Uniwersytet Ekonomiczny
we Wrocławiu

KIERUNEK STUDIÓW

INFORMATYKA W BIZNESIE

Kamil Breczko
Nr albumu 179792

PRACA MAGISTERSKA

Analiza i ocena wydajności aplikacji dla firm zaprojektowanych w języku Java w wersji 8 i 11

Promotor:
dr inż. Kamal Matouk
Katedra Zarządzania Procesami

WROCŁAW 2021

Analysis and evaluation of application performance for companies designed in Java version 8 and 11

Abstract

Nowadays, we can observe a rapid development of technology, more and more memory reserves, more efficient processors. New programming paradigms, programming languages and subsequent versions of these languages are emerging. Despite this, the performance issue in software still exists. With the development of technology, the complexity of applications and operations it performs has increased. This paper is an evaluation and performance analysis of applications designed in the Java programming language, where its two versions have been compared. The Java programming language is a popular choice among business applications. As part of the performance analysis, a scientific experiment was carried out, preceded by an analysis of the state of the literature. All the changes made to the performance from version 8 to version 11, including changes to the runtime, were included in the literature review and experiment.

Streszczenie

W obecnych czasach możemy zaobserwować gwałtowny rozwój technologii, coraz to większe zapasy pamięci, bardziej wydajne procesory. Powstają nowe paradygmaty programowania, języki programowania i kolejne wersje tych języków. Pomimo tego, problem wydajności w oprogramowaniu nadal istnieje. Wraz z rozwojem technologii wzrosła złożoność aplikacji i operacji jakie ona wykonuje. Niniejsza praca dyplomowa stanowi ocenę i analizę wydajnościową aplikacji zaprojektowanych w języku programowania Java, gdzie między sobą zostały porównane jego dwie wersje. Język programowania Java jest popularnym wyborem wśród aplikacji dla firm. W ramach analizy wydajnościowej został przeprowadzony eksperyment naukowy poprzedzony analizą stanu literatury. W przeglądzie literatury oraz w eksperymencie zostały uwzględnione wszystkie wprowadzone zmiany wpływające na wydajność od wersji 8 do wersji 11 z uwzględnieniem zmian w środowisku uruchomieniowym.

SPIS TREŚCI

WSTĘP

Rozwój informatyki wraz z postępującą cyfryzacją spowodował wzrost zainteresowania programowaniem komputerów. Powstają coraz to nowsze maszyny, urządzenia, komputery i podzespoły, które sterowane są oprogramowaniem zaprojektowanym przez programistów. Wraz z rozwojem technologii powstają nowe paradygmaty programowania, języki programowania i kolejne wersje tych języków, które ułatwiają wytwarzanie efektywnego, czystego i zrozumiałego kodu dla innych osób. Dynamiczny rozwój języków programowania i nauki o programowaniu stawia wysoki próg wejścia dla nowych programistów.

Nie każdy zdaje sobie sprawę, że przy każdej kolejnej opublikowanej wersji danego języka programowania, oprócz wprowadzania zmian w składni, twórcy języków wprowadzają różnego rodzaju optymalizacje. W obecnych czasach możemy zaobserwować gwałtowny rozwój technologii, coraz większe zapasy pamięci i bardziej wydajne procesory. Pomimo tego, problem wydajności w oprogramowaniu nadal istnieje. Wraz z rozwojem technologii wzrasta złożoność aplikacji i operacji jakie ona wykonuje.

Żyjemy w czasach pandemii, w dzisiejszych czasach wiele firm głównie rozwija się dzięki informatyce i komunikacji poprzez internet. Takimi firmami są banki, sklepy internetowe, firmy kurierskie czy restauracje. Większość firm, jeśli decyduje się na wytworzenie oprogramowania, to stawia na stabilne języki programowania.

Celem pracy jest analiza i porównanie wydajnościowe dwóch głównych wersji Java 8 i 11, wyciągnięcie wniosków z otrzymanych wyników oraz ich interpretacja. Niewątpliwe, że zmiany opublikowane przy wydaniu Javy 8 wprowadziło rewolucję w środowisku Java. Głównym przesłaniem pracy jest postawiona teza: analiza porównawcza wydajności programów zaprojektowanych i uruchomionych na środowiskach obu wersji Javy pokaże, że warto migrować i korzystać z nowszych wersji Javy oraz pozwoli na podejmowanie korzystnych decyzji dla nowych i istniejących projektów.

Język programowania Java jest jeden z najpopularniejszych języków do budowania aplikacji webowych. Ten język programowania w ostatnim czasie rozwija się bardzo prężnie, co umacnia swoją pozycję. To już standard, że środowisko Java jest zainstalowane na komputerach osobistych, telefonach oraz serwerach. Programiści znacznie chętniej wybierają go do nauki, a menadżerowie projektów oraz właściciele produktów wybierają jako główny język programowania w swoich projektach.

W obecnych czasach wydajność i czas działania aplikacji jest bardzo istotna dla biznesu jak i dla klientów, którzy używają takich aplikacji. Zazwyczaj aplikacje sięgają kilkunastu tysięcy linii kodu źródłowego. Nikt nie może sobie pozwolić na sytuację, gdzie aplikacja na środowisku produkcyjnym zużywa wszystkie zasoby procesora, środowisko uruchomieniowe zajmuje całą pamięć operacyjną, aplikacja dalej uruchamia się zbyt długo, pojawiają się przerwy w działaniu lub czas odpowiedzi jest niewystarczający. Wtedy, nie zawsze dobrym pomysłem jest dokładanie dodatkowych zasobów. Dlatego w analizie porównawczej zostało uwzględnione zarządzanie pamięcią RAM i czas działania aplikacji. Przeprowadzone badania pomogły odpowiedzieć na pytania badawcze z rozdziału 2, które zostały sformułowane i postawione po wykonanym przeglądzie literatury.

Praca składa się z trzech rozdziałów. Rozdział 1 jest poświęcony przeglądzie literatury, gdzie wstępnie została nakreślona dziedzina niniejszej pracy. Rozdział został podzielony na dwie główne części. W pierwszej części wprowadzono do środowiska Java. Udzielono odpowiedzi na pytania: czym jest Java i z jakich składa się składników. Został wytłumaczony nagły wzrost popularności i rozwój języka programowania Java. Przedstawiono istotne informacje dla menadżerów projektowych odnośnie do zmian w licencjonowaniu i wersjonowaniu. Ostatnia część została poświęcona przeglądowi zmian między wersją Javy 8 a 11, gdzie skupiono się głównie na elementach, które mają bezpośredni wpływ na wydajność aplikacji. W pracy została wykorzystania zarówno literatura zagraniczna jak i krajowa.

W rozdziale 2 został zdefiniowany obszar badawczy. Na podstawie analizy stanu literatury postawiono pytania badawcze oraz zostały opracowane wstępne spostrzeżenia. Kolejna cześć rozdziału rozpoczyna się od przeglądu wskaźników do pomiaru wydajności aplikacji w języku Java, następnie przedstawiono metody i narzędzia do testowania wydajności, które w ramach pracy zostały wybrane do badań. Na koniec rozdziału został omówiony plan i sposób przeprowadzania badań, które umożliwią odpowiedzieć na postawione pytania badawcze. W ramach opisu sposobu przeprowadzenia badań przedstawiono opis i implementację algorytmów do pomiaru wydajności aplikacji.

Rozdział 4 stanowi prezentację oraz interpretację wyników badań zrealizowanych w ramach rozdziału 3. Rozdział został podzielony na trzy części zgodnie z wymienionymi spostrzeżeniami. Rozdziały dzielą się ze względu na: zmiany w bibliotekach, zmiany w modułach *garbage collector* oraz zmiany w *kompilatorze Just-in-Time.* Przedstawiono wyniki wraz z omówieniem oraz ich interpretacją.

Na koniec wyciągnięto końcowe wnioski, rekomendacje odnośnie do migracji do nowszych wersji, wskazano odpowiedzi na postawione pytania badawcze w rozdziale 3 oraz

zasugerowano dalsze kierunki badań. W niniejszej pracy zostały zastosowane następujące metody badawcze:

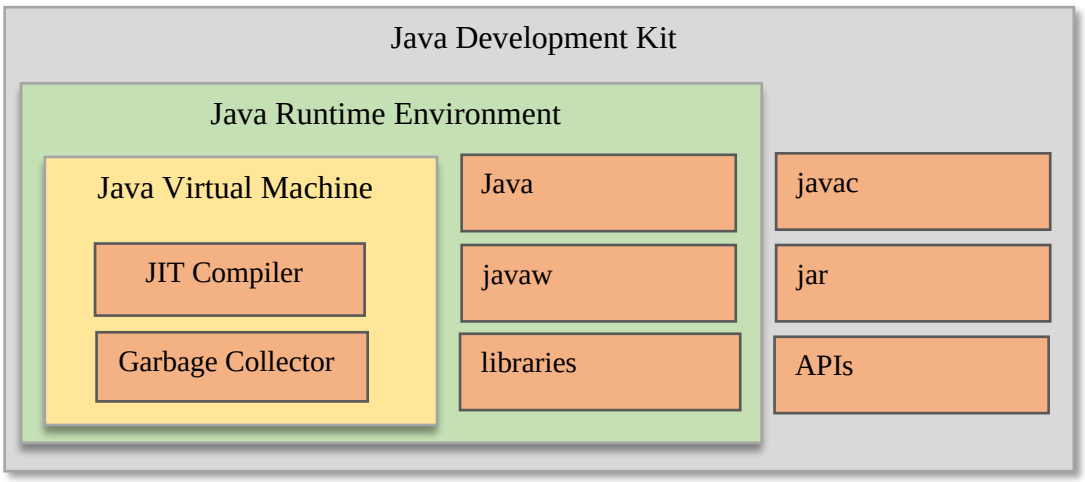
- analiza literatury – przegląd literatury w zakresie zmian wydajnościowych między wersjami języka programowania,
- analiza – rozpatrywanie i wybór algorytmów pod kątem wydajnościowym,
- eksperyment naukowy, symulacja komputerowa – wytworzenie kodu źródłowego, skompilowanie do kodu bajtowego zgodnie z wersją języka programowania, uruchomienie zbudowanej aplikacji w wcześniej skonfigurowanym środowisku uruchomieniowym,
- porównanie – ustalenie podobieństw i różnić w wydajności aplikacji,
- interpretacja – wyciągnięcie wniosków z przeprowadzonych badań i wyjaśnienie otrzymanych wyników z eksperymentu naukowego.

1. PRZEGLĄD ZMIAN MIĘDZY JAVA 8 A JAVA 11

Niniejszy rozdział został poświęcony przeglądzie literatury. W podrozdziałach zostały omówione następujące tematy: wprowadzenie do Java, popularność wersji Java na produkcji, różnica między Java Development Kit a Java Runtime Environment, zmiany w wersjonowaniu i licencjonowaniu. Następnie został omówiony przegląd najważniejszych zmian od wersji 8 do wersji 11, które mają bezpośredni wpływ na wydajność aplikacji.

1.1. Wprowadzenie do języka Java

W języku potocznym pod nazwą Java kryje się wiele modułów, rozumie się nie tylko jako język programowania, ale także jako cały pakiet programistyczny - Java Development Kit, w skrócie JDK. Produkt dostępny jest dla wielu systemów operacyjnych stworzone przez firmę Sun, a którego obecnym właścicielem jest Oracle Corporation. W niniejszej pracy skupiamy się jedynie na edycji standardowej (Java Platform, Standard Edition – zwana wcześniej jako J2SE). Pakiet JDK zawiera w sobie środowisko uruchomieniowe (Java Runtime Environment, w skrócie JRE) oraz zestaw narzędzi niezbędnych do kompilowania oraz budowania gotowych uruchamialnych plików. Zostało to przedstawione na rysunku 1. Do wersji Java 11, zestaw JRE był niezależny i można było pobrać w celu uruchomienia zbudowanych plików *jar*. Począwszy od wersji 11 należy pobrać cały pakiet JDK lub utworzyć niestandardowe środowisko JRE. Język programowania Java służy do wytwarzania kodów źródłowych kompilowanych do kodu bajtowego, czyli postaci wykonywanej przez maszynę wirtualną (Java Virtual Machine, w skrócie JVM).



Rys. 1. Komponenty zawarte w Java Development Kit

Źródło: opracowanie własne.

Niewątpliwie Java pozycjonuje się jako jeden z najpopularniejszych języków programowania, głównie do tworzenia aplikacji internetowych. Rysunek 2 przedstawia najpopularniejsze języki w ostatnich latach według indeksu popularności TIOBE. Patrząc na statystyki, Java od 20 lat mieści się w TOP 5 najpopularniejszych języków. W marcu 2021 roku uzyskał 2 miejsce wśród wszystkich języków. W 2015 roku został ogłoszony jako „język programowania roku". Warto w tym momencie dodać, że pierwsze wydaje Javy 8 miało miejsce w marcu 2014 roku.

Mar 2021	Mar 2020	Change	Programming Language	Ratings	Change
1	2	▲	C	15.33%	-1.00%
2	1	▼	Java	10.45%	-7.33%
3	3		Python	10.31%	+0.20%
4	4		C++	6.52%	-0.27%
5	5		C#	4.97%	-0.35%
6	6		Visual Basic	4.85%	-0.40%
7	7		JavaScript	2.11%	+0.06%
8	8		PHP	2.07%	+0.05%
9	12	▲	Assembly language	1.97%	+0.72%
10	9	▼	SQL	1.87%	+0.03%

Rys. 2. Najpopularniejsze języki programowania według indeksu popularności TIOBE

Źródło: [1]

[1] Mishra, P., *TIOBE Index for March 2021*, [w.] https://www.lambdatest.com/blog/top-10-java-testing-frameworks [30.03.2021]

1.2. Wersjonowanie i licencjonowanie

Mając na uwadze rozwój języków programowania i technologii, sponsorzy Java, czyli Oracle, nie mogli dopuścić, aby rynek został przejęty przez inne języki programowania i wyparto język Java. Jeszcze do roku 2017 losy Javy były niepewne, ponieważ nowe wersje były wypuszczane w przedziale mniej więcej co 5 lat, a obok rosły w siłę inne języki programowania. Ostatnią główną wersją była Java 8 z 2014 roku. Dopiero w 2017 roku została wypuszczona Java w wersji 9 podczas której zmieniło się podejście do wersjonowania i licencjonowania Javy. W roku 2018 świat doczekał się Javy wersji 11 na nowych zasadach.

Wraz z wypuszczeniem Javy 9, sponsorzy ogłosili dwie ważne zmiany. Pierwsza zmiana dotyczy wersjonowania. Zostało ogłoszone, że nowe wersje będę udostępniane dwa razy w roku z krótkim wsparciem oraz co trzy lata z wydłużonym wsparciem. Firma Oracle zapowiedziała, że będzie wspierała Java 8 do 2022 roku, a Java 11 do 2023 roku, z możliwością wykupienia dodatkowego rozszerzonego wsparcia. Zostało to przedstawione na rysunku 3. Druga istotna zmiana dotyczy licencjonowania. Firma Oracle zapowiedziała, że Java będzie darmowa tylko do wytwarzania i testowania oprogramowania. Od stycznia 2019 roku nie będzie już publikowała aktualizacji do użytku komercyjnego. Produkt Java nie będzie już udostępniany na licencji *Binary Code License,* ale na licencji komercyjnej *Oracle Technology Network License,* czyli zgodnie z zasadami licencji to „licencja ta nie zezwala na używanie Javy do jakiegokolwiek użytku komercyjnego takiego jak procesowanie danych, używanie w systemach produkcyjnych czy do innych aplikacji biznesowych używanych wewnątrz firmy” [2]. W przypadku używania Javy produkcyjnie trzeba będzie zakupić subskrypcję od firmy Oracle lub skorzystać z innych podmiotów, które korzystając ze źródeł OpenJDK budują, udostępniają i wspierają binarne dystrybucje JDK. Jednym z takich podmiotów jest AdoptOpenJDK, skąd zostały pobrane omawiane wersje Javy i użyte w niniejszej pracy do przeprowadzenia badań. Rysunek 4 przedstawia plan pomocy technicznej projektu AdoptOpenJDK na obecne wersje Javy. Warto zaznaczyć, że firma Oracle sponsoruje zespół implementujący OpenJDK, a źródła są udostępniane na licencji *GPLv2+CE* [3].

[2] M. Waksmański, *Czy Java jest nadal darmowa? Jeśli nie, to co teraz? Które JDK wybrać?,* [w.] https://devrev.pl/czy-java-jest-nadal-darmowa/ [01.04.2021]
[3] J. Tokarski, *„Płatna Java” podana na zimno,* [w.] https://www.pgs-soft.com/pl/blog/platna-java-podana-na-zimno/ [01.04.2021]

Oracle Java SE Support Roadmap*†			
Release	**GA Date**	**Premier Support Until**	**Extended Support Until**
7	July 2011	July 2019	July 2022*****
8**	March 2014	March 2022	December 2030
9 (non-LTS)	September 2017	March 2018	Not Available
10 (non-LTS)	March 2018	September 2018	Not Available
11 (LTS)	September 2018	September 2023	September 2026
12 (non-LTS)	March 2019	September 2019	Not Available
13 (non-LTS)	September 2019	March 2020	Not Available
14 (non-LTS)	March 2020	September 2020	Not Available
15 (non-LTS)	September 2020	March 2021	Not Available
16 (non-LTS)	March 2021	September 2021	Not Available
17 (LTS)	September 2021***	September 2026****	September 2029****

Rys. 3. Plan pomocy technicznej Oracle Java SE

Źródło: [4]

First Availability		Latest Release	Next Release	End of Availability [1]
Java 8 (LTS)	Mar 2014	jdk8u282-b08 19th Jan 2021	jdk8u292 20th Apr 2021	At Least May 2026 [1]
Java 9	September 2017	jdk-9.0.4+11 16th January 2018	N/A	Mar 2018
Java 10	Mar 2018	jdk-10.0.2+13 17th Jul 2018	N/A	September 2018
Java 11 (LTS)	September 2018	jdk-11.0.10+9 19th Jan 2021	jdk-11.0.11 20th Apr 2021	At Least Oct 2024 [1]
Java 12	Mar 2019	jdk-12.0.2+10 16th Jul 2019	N/A	September 2019
Java 13	September 2019	jdk-13.0.2+8 14th Jan 2020	N/A	Mar 2020
Java 14	Mar 2020	jdk-14.0.2+12 14th Jul 2020	N/A	Sep 2020
Java 15	Sep 2020	jdk-15.0.2+7 20th Oct 2020	N/A	Mar 2021
Java 16	Mar 2021	jdk-16+36	jdk-16.0.1 20th Apr 2021	Sep 2021
Java 17 (LTS)	Sep 2021	N/A	jdk-17 14th Sep 2021	TBC[1]

Rys. 4. Plan pomocy technicznej AdoptOpenJDK

Źródło: [5]

[4] Oracle Corporation, *Oracle Java SE Support Roadmap*, [w.] https://www.oracle.com/java/technologies/java-se-support-roadmap.html [31.03.2021]

[5] AdoptOpenJDK, *AdoptOpenJdk Java SE Support Roadmap*, [w.] https://adoptopenjdk.net/support.html [31.03.2021]

1.3. Przegląd zmian

Niniejszy podrozdział został poświęcony przeglądowi zmian od wersji 8 do wersji 11 języka programowania Java. W przeglądzie zmian skoncentrowano się na istotnych zmianach, które mogą mieć wpływ na wydajność aplikacji. Ze względu na liczbę wprowadzonych zmian podzielono podrozdział na sekcje zgodnie z znajdującymi się komponentami w pakiecie JDK: *Garbage Collector*, kompilator *Just-in-Time*, optymalizacje na etapie wykonywania kodu, optymalizacje na etapie kompilacji do kodu bajtowego, zmiany w bibliotekach.

Aby przybliżyć działanie poniżej omawianych komponentów należy prześledzić cykl wytwarzania oprogramowania i zrozumieć jaką rolę odgrywa wirtualna maszyna Java, czyli Java Virtual Machine. Pierwszym krokiem w wytwarzaniu oprogramowania jest napisanie kodu źródłowego przez programistę. To właśnie w tym miejscu najczęściej popełniane są błędy wydajnościowe. Wprowadzone mechanizmy i zmiany w wersjach mają na celu zadbać o optymalizację tego kodu. Następnym krokiem jest zbudowanie artefaktu przy pomocy kompilatora *javac.* Wynikiem kompilacji kodu źródłowego przez *javac* są pliki z rozszerzeniem *class*, czyli kod bajtowy. Ostatnim krokiem jest uruchomienie kodu bajtowego na wirtualnej maszynie Java, który następnie będzie interpretowany i cały czas analizowany pod względem wydajnościowym. Jeszcze przed samym uruchamianiem programu maszyna wykonuje następujące procesy:

- wczytywanie plików z rozszerzeniem *class* do pamięci JVM,
- weryfikacja definicji klas,
- przygotowanie plików, czyli tworzenie i inicjalizacja wszystkich pól statycznych,
- rozwiązywanie zależności, który zapewnia, że każdy typ jest gotowy do wykorzystania.

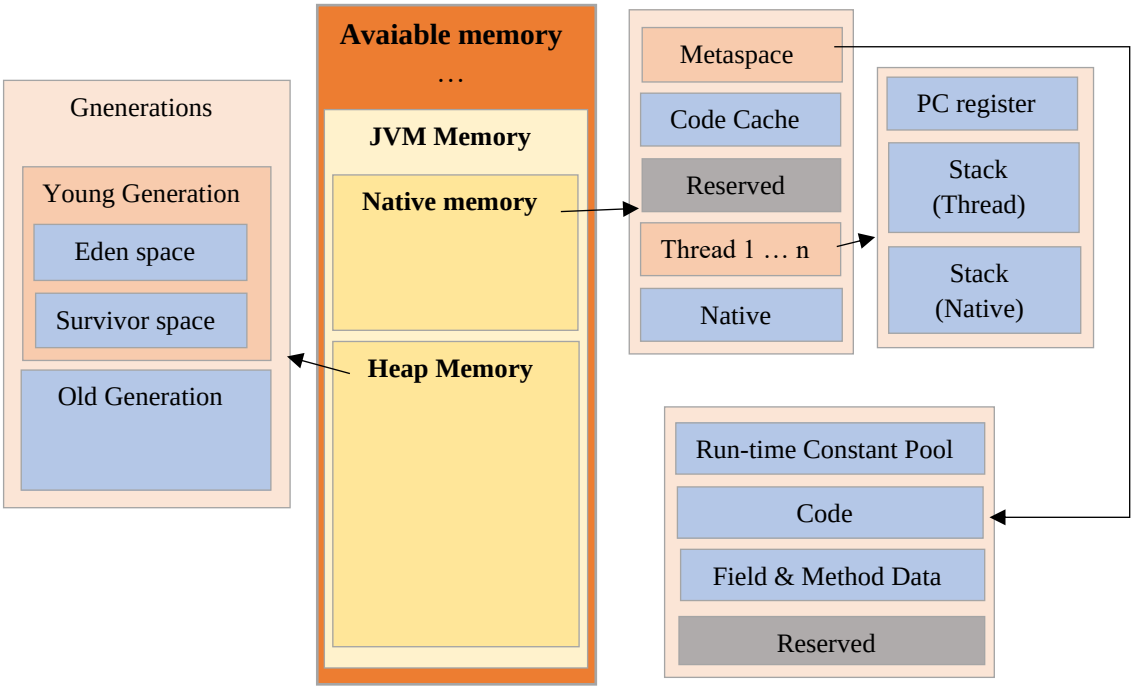
Dzięki takiemu rozwiązaniu, program może zostać uruchomiony na każdej platformie, na której jest możliwość zainstalowania wirtualnej maszyny Java. Istnieje kilka produktów, które implementują specyfikacje Wirtualnej Maszyny Java. Referencyjną implementacją wirtualnej maszyny Java jest HotSpot rozwijany przez projekt OpenJDK. Innymi słowy większość dostępnych produktów rozszerza HotSpot dodając funkcjonalności specyficzne dla konkretnych grup docelowych. Do takich implementacji należy Eclipse OpenJ9, Amazon Corretto oraz Azul Zulu. W niniejszej pracy został użyty JVM z pakietu OpenJDK [6].

[6] J.Kubryński, *Co każdy programista Java powinien wiedzieć o JVM*, „Programista", nr. 03/2015, s. 24

1.3.1. Odśmiecanie pamięci

Istnieją dwie strategie zarządzaniem pamięcią dynamiczną. Pierwsza strategia polega na obarczeniu tą odpowiedzialnością programistę. To programista tworzy i czyści pamięć na stercie. Niewątpliwie jest to wydajne rozwiązania, ale mniej czytelne i wygodne podczas wytwarzania kodu. Druga strategia polega na przeniesieniu tej odpowiedzialności do niezależnego modułu działającego obok wykonywanego programu. To właśnie ta strategia została zaimplementowana w Javie. Dlatego w odróżnieniu od niektórych języków programowania, programista pisząc w języku Java nie musi martwić się o zwalnianie pamięci, ponieważ maszyna wirtualna Javy robi to za niego. Jednym z mechanizmów używanych do tych celów przez wirtualną maszynę Javy jest *Garbage Collector*, w skrócie GC. Zadaniem *Garbage Collector'a* jest dbanie o alokowaną pamięć, czyszczenie z nieużywanych plików oraz wydajność aplikacji. Moduł *Garbage Collector* towarzyszy od początku istnienia Javy i rozwija się wraz z nowymi wydaniami. Aby zrozumieć działanie i wprowadzone zmiany w modułach GC trzeba najpierw zrozumieć, jak dzieli się obszar pamięci w Javie.

Całą pamięć dzieli się na kilka poziomów. Podstawowym podziałem obszaru pamięci w wirtualnej maszynie Javy jest to podział na: sterta i dane poza stertą (pamięć natywną). Dokładnie przedstawia to rysunek 5.

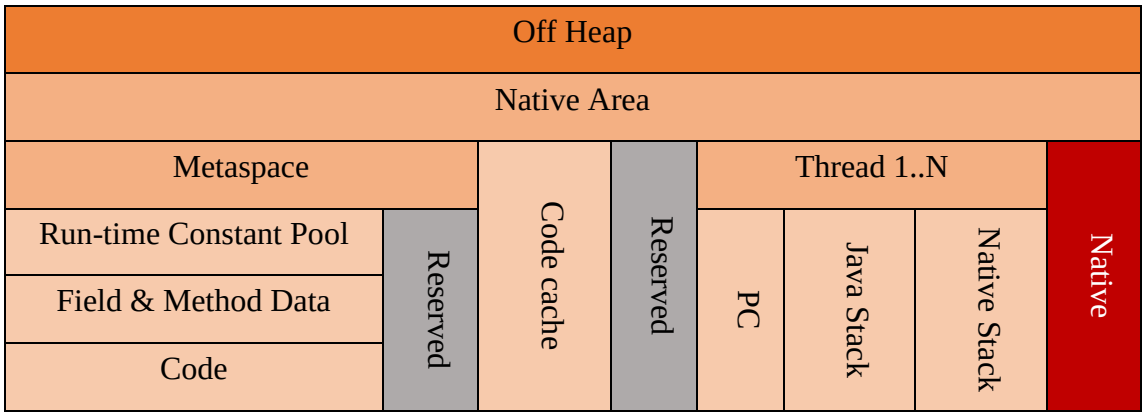


Rys. 5. Obszar pamięci wirtualnej maszyny Java

Źródło: opracowanie własne.

W dość mocnym uproszczeniu, w pamięci poza stertą umieszczone są byty potrzebne do wsparcia działania wirtualnej maszyny i reprezentacji jej struktur wewnętrznych. Możemy ją podzielić na 2 grupy: pamięć współdzielona i autonomiczna. Pamięć współdzielona dzieli się na dodatkowe segmenty: segment *Metaspace* i segment *Code cache*. W segmencie *Metaspace* przechowywane są dane statyczne takie jak: kody klas, metody, pola, literały znakowe, referencje czy stałe. W segmencie *Code cache* przechowywane są zoptymalizowane i skompilowane kody aplikacji. Ten segment jest mocno powiązany z modułem *kompilatora Just-in-Time*, który szerzej będzie omówiony w następnym podrozdziale. Kolejną grupą pamięci natywnej jest pamięć autonomiczna, czyli pamięć przypisana do każdego wątku działającego na maszynie wirtualnej. Zawiera ramki wywołań, zmienne lokalne i referencję do aktualnie wykonywanej instrukcji kodu [7]. Schemat pamięci poza stertą został zobrazowany na rysunku 6.

[7] D. Rudczyk, *Obszary pamięci Maszyny Wirtualnej Javy (JVM),* [w.] https://softwareskill.pl/obszary-pamieci-maszyny-wirtualnej-javy-jvm [14.04.2021]



Rys. 6. Schemat pamięci poza stertą

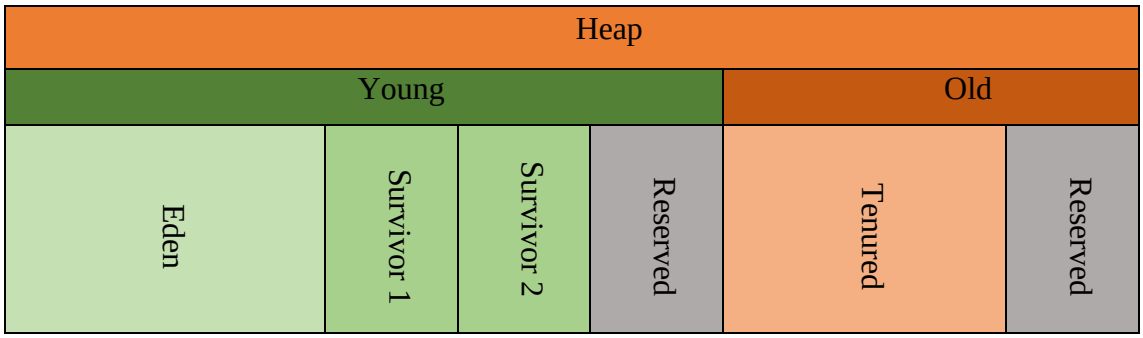
Źródło: opracowanie własne.

Drugą i bardziej interesującą częścią pamięci jest sterta. To właśnie tutaj przechowywane są wszystkie instancje klas utworzone w trakcje wykonywania programu. Sterta ma największy wpływ na wydajność aplikacji niezależnie od wybranego języka programowania, ponieważ pamięć w stercie dość często może się zmieniać, ulegać fragmentacji i przepełniać. Obszar pamięci może rosnąć wraz z przydzielaniem i zwiększaniem zasobów dla wirtualnej maszyny Java. Ze względu na konieczność przeglądania całej tej powierzchni, zarządzania nią i w obawie o wydajność aplikacji to podzielono stertę na generacje. Stertę dzieli się na dwie części: przestrzeń dla nowych i starych obiektów, tak jak pokazano to na rysunku 7. Dodatkowo przestań dla nowych obiektów dzielona jest na podprzestrzenie:

- Eden,
- przestrzeń przetrwalników 0,
- przestrzeń przetrwalników 1.

Początkową przestrzenią, przez którą każdy obiekt musi przejść jest Eden. W tym obszarze żyją nowe obiekty. Wraz z działaniem aplikacji, poszczególne obszary będą się zapełniać, aż do momentu uruchamiania mechanizmu oczyszczania pamięci. Mechanizm oczyszczania pamięci usunie nieużywane obiekty w konkretnym obszarze i awansuje do następnej generacji. W każdej z tych części można obrać inną strategię usuwania obiektów i promowania do wyższej generacji. Takie podejście wychodzi z dwóch założeń:

- większość obiektów szybko staje się nieużyteczna z punktu widzenia aplikacji,
- odwołania starych obiektów do nowych są bardzo rzadkie.



Rys. 7. Schemat sterty

Źródło: opracowanie własne.

Do rozwiązania problemu oczyszczania pamięci dynamicznej opracowano wiele rodzajów algorytmów. Jednym z nich jest algorytm wektorowy, który polega na reprezentowaniu obiektów i zależności jako grafy. Taki mechanizm został zaadaptowany w Javie. Większość powstałych modułów *Garbage Collector* pracuje podobnie i w uproszczeniu proces wygląda następująco:

- W pierwszym kroku, algorytm szuka obiektów źródłowych, które będą korzeniami w grafie. Korzeniem może być uruchomiony wątek albo zmienna lokalna.
- W następnym kroku algorytm przechodzi po korzeniach i oznacza obiekty jako używane.
- Następnie algorytm ponownie przegląda zaznaczone korzenie, ale tym razem zaznacza wszystkie powiązane z nim obiekty jako używany.
- Na koniec iteruje po wszystkich obiektach z pamięci i sprawdza czy taki obiekt został oznaczony. Jeśli obiekt nie został oznaczony, to wtedy zostaje usunięty z pamięci.
- Opcjonalnym krokiem jest reorganizacja pamięci programu, aby uniknąć fragmentacji.

W dużym skrócie moduł znajduje wszystkie żywe obiekty, usuwa pozostałe i ewentualnie reorganizuje pamięć [8].

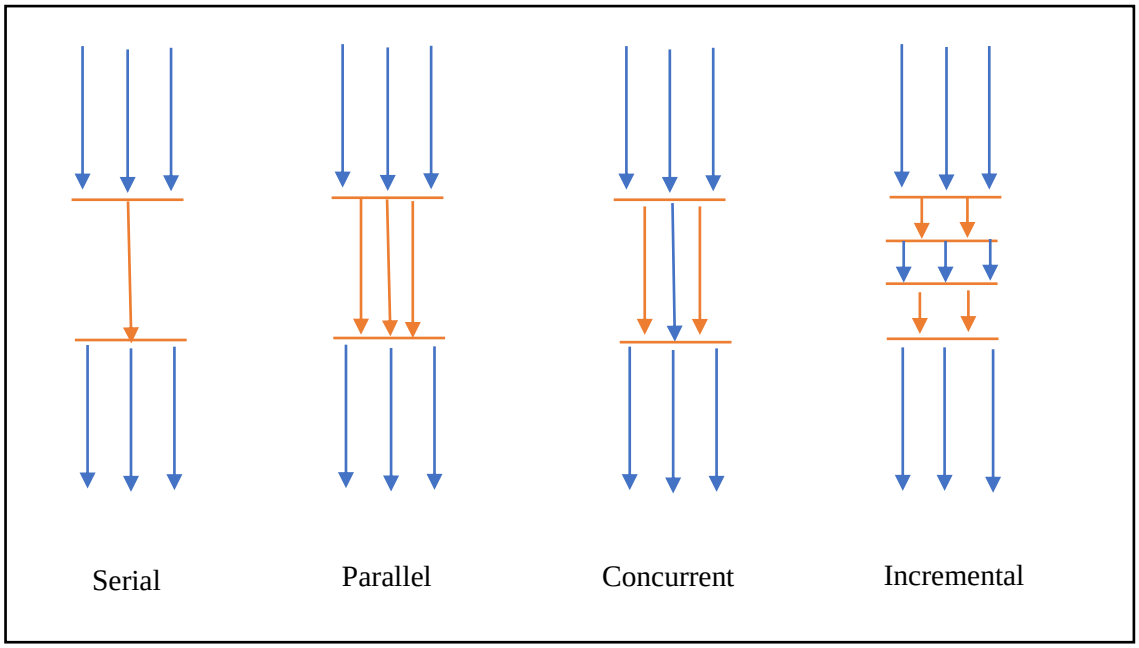
Moduły oczyszczania pamięci w Javie możemy podzielić na cztery kategorie w zależności od trybu pracy:

- tryb szeregowy – wszystkie wątki są zatrzymywane, a wyłączny dostęp do pamięci ma wątek GC,
- tryb równoległy – wszystkie wątki są zatrzymywane, a wyłączny dostęp do pamięci mają wątki GC. Różni się od trybu szeregowego zwiększoną ilością wątków dla GC,

[8] S. Nayak, *Garbage Collection in Java – What is GC and How it Works in the JVM*, [w.] https://www.freecodecamp.org/news/garbage-collection-in-java-what-is-gc-and-how-it-works-in-the-jvm/ [15.04.2021]

- tryb współbieżny – pozwala na dalsze wykonywanie wątków aplikacyjnych z jednoczesnym uruchomieniem wątków GC. Taki tryb nie zatrzymuje aplikacji, a jedynie zmniejsza przepustowość.
- tryb przyrostowy – wykonywanie na zmianę wątki aplikacyjne i GC [9].

Powyższy podział ilustruje rysunek 8.



Rys. 8. Podział modułów do czyszczenia pamięci w zależności od trybu pracy

Źródło: opracowanie własne.

Oprócz podziału na tryb pracy, moduły czyszczenia pamięci klasyfikuje się według sposobu usuwania nieużywanych obiektów. Wyróżniamy:

- *Mark-Sweep* – w pierwszym kroku oznacza obiekty używane przez wirtualną maszynę, a następnie czyści pamięć, usuwając obiekty porzucone w pierwszym kroku,
- *Mark-Sweep-Compact* – wykonuje te same kroki jakie w *Mark-Sweep*, z tą różnicą, że dokonuje realokację danych tak aby pamięć nie pozostała pofragmentowana,
- *Mark-Copy* – kopiuje używane obiekty do nowego miejsca na stercie [10].

W zależności od wybranego trybu pracy i strategii usuwania nieosiągalnych obiektów, pojawiają się następujące problemy wydajnościowe:

[9] J.Kubryński, *Co każdy programista Java powinien wiedzieć o JVM: zarządzanie pamięcią*, „Programista", nr. 04/2015, s. 21-22

[10] M. Marczak, *JVM Garbage Collector. Wprowadzenie*, [w.] https://bulldogjob.pl/news/404-jvm-garbage-collector-wprowadzenie [15.04.2021]

- przepustowość – zmniejszenie rzeczywistej szybkości wykonywania wątków aplikacyjnych,
- latencja - opóźnienie w działaniu programu, wynikająca z przerwania wątków aplikacyjnych na rzecz uruchomienia wątków modułu oczyszczenia pamięci,
- fragmentacja pamięci – efekt uboczny, powstały po czyszczeniu pamięci. Problem fragmentacji pamięci jest uzależniony od wyboru strategii usuwania nieużywanych obiektów. Nierozwiązanie tego problemu, będzie powodowało, że JVM będzie spędzał więcej czasu na szukaniu odpowiedniego miejsca do alokowania obiektu.

Przy wyborze trybu pracy zawsze wybieramy pomiędzy przepustowością a latencją, zaś wybór strategii usuwania nieużywanych obiektów wpływa głównie na efekt pofragmentowanej pamięci [11].

Java JDK zawiera cztery moduły do oczyszczania pamięci:

- *Serial Garbage Collector (SerialGC),*
- *Parallel Garbage Collector (ParallelGC),*
- *CMS Garbage Collector (CMSGC),*
- *G1 Garbage Collector (G1GC).*

W zależności od potrzeby można wybrać jeden z nich. Jeśli użytkownik nie wybierze konkretnego modułu do oczyszczania pamięci to wirtualna maszyna ustawi domyślny, który może różnić się w zależności od wersji Javy. W niniejszej pracy zostaną przybliżone dwa z nich. Pierwszy jest to *Parallel Garbage Collector*, a drugi to *G1 Garbage Collector* [12].

Moduł *Parallel Garbage Collector* jest domyślnym algorytmem do oczyszczania pamięci w Javie 8. Algorytm zwany jest także algorytmem wysokiej przepustowości. W przypadku młodej generacji moduł używa strategii *Mark-Copy*, czyli kopiowania żywych obiektów do nowego miejsca na stercie. W przypadku starej generacji używa *Mark-Sweep-Compact*, czyli oznacza obszar zajmowanych przez osiągalne obiekty, następnie usuwa obiekty porzucone w pierwszym kroku a na końcu eliminuje fragmentacje pamięci. Moduł działa w trybie równoległym, a więc zatrzymuje wątki aplikacyjne i używa je do czyszczenia pamięci. Dodatkowo dzieli wątki na wykonywanie konkretnych zadań. Dla pojedynczego wątku przydziela wyczyszczenie pamięci z starej generacji, a następnie wyeliminowanie fragmentacji, zaś resztę wątków przydziela do młodej generacji w celu posprzątania pamięci [13].

[11] *M. Marczak, JVM Garbage Collector. Wprowadzenie*, [w.] https://bulldogjob.pl/news/404-jvm-garbage-collector-wprowadzenie [15.04.2021]
[12] A.Altvater, *Co to jest Java Garbage Collection? Jak to działa, sprawdzone metody, samouczki i nie tylko*, [w.] https://stackify.com/what-is-java-garbage-collection/ [15.04.2021]
[13] M. Marczak, *Algorytmy GC. Serial, Parallel, CMS*, [w.] https://bulldogjob.pl/news/424-algorytmy-gc-serial-parallel-cms [15.04.2021]

Moduł *G1 Garbage Collector* jest dość nowym rozwiązaniem. Został wprowadzony w Javie wersji 7, a ustawiony jako domyślny moduł do oczyszczania pamięci w Javie wersji 9. Celem zaimplementowania modułu *G1 Garbage Collector* jest zastąpienie modułu *CMS Garbage Collector*, który w JEP 291 został oznaczony jako przestarzały [14]. Algorytm zwany jest także jako algorytm niskiej latencji. Działa w trybie równoległym i współbieżnym jak CMS, ale pod spodem działa zupełnie inaczej w porównaniu do starszych modułów. Moduł G1 to regionalizowany i generacyjny moduł odśmiecania pamięci, oznacza to, że algorytm jest odpalany na regionach o jednakowych wielkości zachowując podejście oparte na generacji obiektów. Rozmiary regionów mogą się wahać od 1MB do 32MB w zależności od rozmiaru sterty. Dodatkowym warunkiem jest utworzenie nie więcej niż 2048 regionów, a więc dla sterty o rozmiarze 2GB, *G1 Garbage Collector* ustali 2048 regionów o rozmiarze 1MB. Struktura sterty też uległa zmianie i została podzielona na 3 główne części:

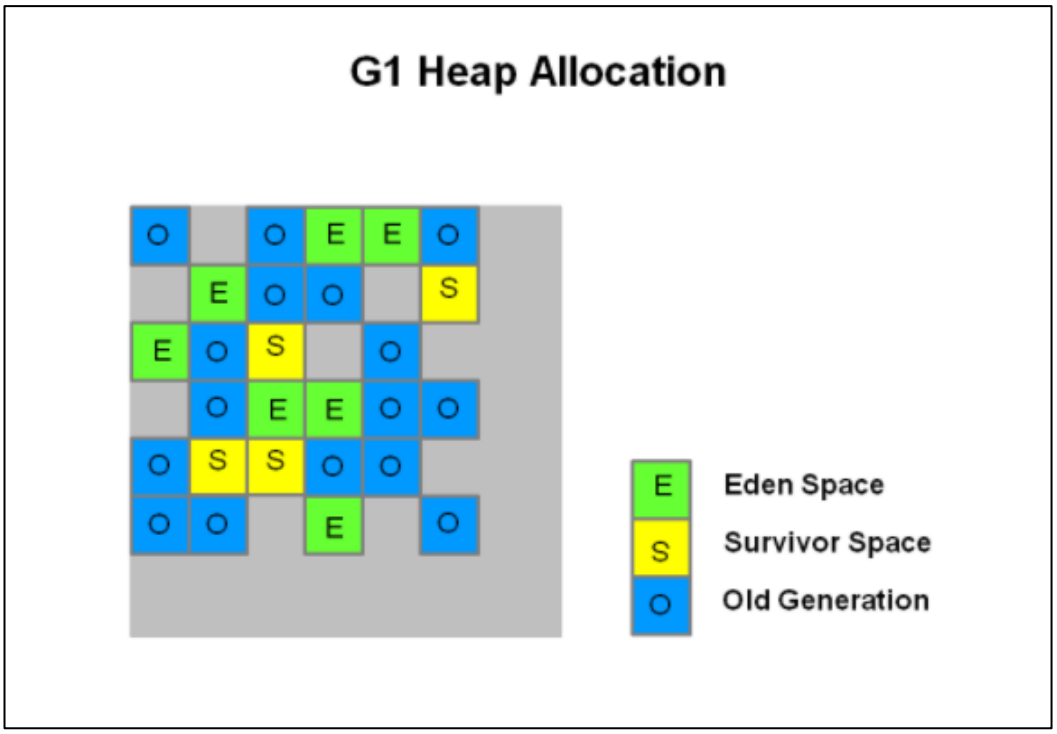
- Eden,
- Przestrzeń przetrwalników,
- Stara Generacja.

Sam algorytm pracuje w czterech głównych fazach:

- W pierwszej fazie zaznacza obiekty źródłowe blokując wątki aplikacyjne.
- W fazie drugiej zaznacza używane obiekty wychodząc od korzenia. Celem jest zaznaczenie w trybie współbieżnym jak największej ilości obiektów. Następnie obiera za cel te regiony, które zapewnią dużą ilość wolnego miejsca.
- W trzeciej fazie, wątki aplikacyjne są zatrzymywane i dokonywana jest końcowa wersja zaznaczania.
- Dopiero w czwartej fazie, algorytm usuwa nieużywane obiekty, kopiuje do jednego lub więcej regionów w celu promocji i zmniejszenia fragmentacji. Ostatnia faza jest wykonywania współbieżnie.

Algorytm stara się pracować cały czas w trybie współbieżnym i unikając pracy na całej stercie, aby skrócić czas przerwy i zwiększyć przepustowość [15].

[14] Oracle Corporation, *JEP 291: Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector,* [w.] https://openjdk.java.net/jeps/291 [15.04.2021]
[15] M. Beckwith, *Garbage First Garbage Collector Tuning,* [w.] https://www.oracle.com/technical-resources/articles/java/g1gc.html [15.04.2021]



Rys. 9. Schemat sterty dla G1 Garbage Collector

Źródło: [16]

Zwykle obiekty przydzielane są do danego regionu aż do jego zapełnienia, aby później moduł GC mógł uruchomić pracę w obrębie danego regionu i oczyścić pamięć. Jednak nie zawsze ta reguła się sprawdza. Moduł *G1 Garbage Collector* inaczej traktuje obiekty o dużych rozmiarach. Jeśli obiekt jest większy niż połowa rozmiaru regionu to taki obiekt jest traktowany wyjątkowo, dzięki czemu dostęp do obiektów jest zazwyczaj szybszy. Takie obiekty nazywane są jako *humongous objects*. Algorytm GC napotykając obiekt wykonuje następujące kroki:

- Przydziela bezpośrednio do starej generacji.
- Tworzony jest oddzielny region, który składa się z kilka sąsiadujących regionów.
- Utworzony region może pomieścić tylko jeden ogromny obiekt, co oznacza, że przestrzeń w ostatnim z połączonych regionów może być prawie pusta.

Algorytm *G1 Garbage Collector* obowiązuje jako domyślny moduł od Javy wersji 9. Zmiana została wprowadzona w JEP 248 [17]. W kolejnych wersjach Javy wprowadzono drobne poprawki. W JEP 307 wprowadzono do modułu tryb równoległy [18]. Oznacza to, że, jeśli moduł

[16] Portal Oracle, *Getting Started with the G1 Garbage Collector*, [w.] https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html [15.04.2021]
[17] Oracle Corporation, *JEP 248: Make G1 the Default Garbage Collector*, [w.] https://openjdk.java.net/jeps/248 [15.04.2021]
[18] Oracle Corporation, *JEP 307: Parallel Full GC for G1*, [w.] https://openjdk.java.net/jeps/307 [15.04.2021]

nie będzie mógł wystarczająco szybko odzyskać pamięci, będzie zmuszony do zatrzymania wątków aplikacyjnych i wyczyścić pamięć [19]. Wraz z Java 11 wprowadzono eksperymentalnie dwa nowe moduły odśmiecania pamięci: *Epsilon Garbage Collector* i *Z Garbage Collector*.

Moduł *Epsilon Garbage Collector* został wprowadzony w JEP 318. Obsługuje alokację pamięci, ale nie implementuje żadnego mechanizmu odzyskiwania pamięci. Po wyczerpaniu dostępnej sterty maszyna JVM zostanie zamknięta. Moduł jest przydatny w przypadku usług krótkotrwałych, niezajmujących dużej przestrzeni i aplikacji, o dość niskim priorytecie [20].

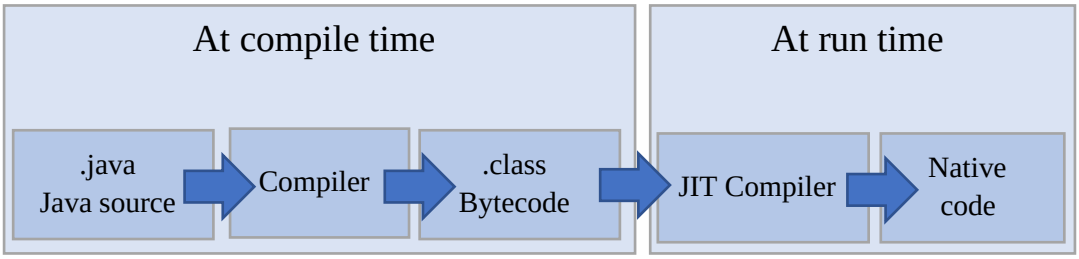
Moduł *Z Garbage Collector* został wprowadzony w JEP 333. Celem powstania modułu jest osiągnięcie niskiego opóźnienia, poprzez unikania zatrzymywania wątków aplikacyjnych. Twórcy założyli za cel, że czas pauzy nie przekroczy 10 milisekund, a redukcja przepustowości aplikacji osiągnie nie więcej niż 15% w porównaniu do korzystania z *G1 Garbage Collector*. Moduł *Z Garbage Collector* jest odpowiedni dla aplikacji, które wymagają małych opóźnień i używają sterty liczącej w terabajtach [21].

1.3.2. Kompilator Just-in-Time

Kolejnym bardzo ważnym komponentem w wirtualnej maszynie Javy jest kompilator *Just-in-Time*. Kompilator *Just-in-Time* (w skrócie JIT) jest jednym z głównych elementów, który wpływa bezpośrednio na wydajność aplikacji Java i odegrał olbrzymią rolę w temacie wydajności. Kompilator działa w czasie wykonywania programu. Pomimo interpretowania kodu bajtowego przez Java, to w niektórych sytuacjach kod optymalizowany przez *kompilator JIT* może być szybciej wykonywany niż kod wytworzony w językach niższego poziomu. Ważnym aspektem, dzięki któremu kompilator może działać jest fakt uruchamiania aplikacji na wirtualnej maszynie Java w trybie interpretowania kodu bajtowego. Dzięki wirtualnej maszynie Java zbierane są statystyki o wykonywanym kodzie. W momencie, w którym metody osiągną odpowiedni próg wywołań to rozpoczyna się optymalizacja. Jedną z optymalizacji jest uruchomienie *kompilatora JIT* w celu skompilowania kodu bajtowego Java do natywnego kodu wykonywalnego, który wykonywany jest bezpośrednio przez procesor, gdzie czas

[19] Rob, *Java 10 improvements to Garbage Collection explained in 5 minutes*, [w.] https://blog.idrsolutions.com/2018/04/java-10-improvements-to-garbage-collection-explained-in-5-minutes/ [15.04.2021]
[20] Oracle Corporation, *JEP 318: Epsilon: A No-Op Garbage Collector (Experimental)*, [w.] https://openjdk.java.net/jeps/318 [15.04.2021]
[21] Oracle Corporation, *JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)*, [w.] https://openjdk.java.net/jeps/333 [15.04.2021]

wykonywania jest nawet 10-krotnie szybszy. Proces kompilowania kodu źródłowego został przedstawiony na rysunku 10 [22].



Rys. 10. Proces kompilowania kodu źródłowego

Źródło: opracowanie własne.

Oprócz skompilowania do kodu natywnego, kompilator będzie próbował wykonać optymalizacje na poziomie kodu bajtowego. Do najważniejszych i najpopularniejszych optymalizacji, które zostały do tej pory wprowadzone należą [23]:

- eliminacja martwego kodu,
- rozwijanie pętli,
- zagnieżdżanie metod,
- grupowanie blokad,
- eliminacja blokad,
- ostrzenie typów [24].

W maszynie wirtualnej dostarczonej przez firmę Oracle standardowo możemy znaleźć kompilator klienta, zwany również C1 i kompilator serwera, zwanym C2. Obecna instalacja Javy używa obu kompilatorów. C1 jest zaprojektowany tak, aby szybciej działał i szybciej wprowadzał zmiany kosztem tworzenia mniej zoptymalizowanego kodu. Jeśli liczba wywołań ponownie będzie rosła to wirtualna maszyna ponownie skompiluje kod, ale tym razem przy użyciu kompilatora C2. Kompilator C2 zajmuje więcej czasu, ale tworzy lepiej zoptymalizowany kod. Od kilku lat, klienci nie doczekali się większych ulepszeń w kompilatorze, ze względu na poziom trudności w jego utrzymywaniu. Sam kompilator został zaprojektowany w języku programowania C++ [25].

[22] K. Ishizaki, A. Hayashi, G. Koblents, V. Sarkar, Compiling and Optimizing Java 8 Programs for GPU Execution, IEEE, 2015, s. 1-3
[23] J.Kubryński, *Co każdy programista Java powinien wiedzieć o JVM*, „Programista", nr. 03/2015, s. 24-26
[24] A. Shipilëv, JVM Anatomy Quarks, [w.] https://shipilev.net/jvm/anatomy-quarks/ [15.04.2021]
[25] *M. Aboullaite, Understanding JIT compiler (just-in-time compiler),* [w.] https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/ [15.04.2021]

Długo nie trzeba było czekać, aż powstanie alternatywna opcja dla domyślnego *kompilatora JIT*. W Javie wersji 9 o kodzie JEP 243 opracowano oparty na języku Java interfejs kompilatora JVM, zwany JVMCI [26]. Nowa zmiana umożliwia wprowadzenie kompilatora napisanego w języku Java przez maszynę JVM jako kompilator dynamiczny. To, co faktycznie pozwala JVMCI, to wyłączenie standardowego narzędzia i wymianę na nowy, bez konieczności zmiany czegokolwiek w JVM. Interfejs jest dość prosty i zawiera metodę *compileMethod* z jednym parametrem. Jako dane wejściowe otrzymuje kod bajtowy wraz z istotnym informacjami do przetworzenia, a zwraca kod maszynowy. Zostało to przedstawione na rysunku 12 [27].

```
package jdk.vm.ci.runtime;

import jdk.vm.ci.code.CompilationRequest;
import jdk.vm.ci.code.CompilationRequestResult;

public interface JVMCICompiler {
    int INVOCATION_ENTRY_BCI = -1;

    CompilationRequestResult compileMethod(CompilationRequest request);
}
```

Rys. 11. Kod źródłowy przedstawiający interfejs JVMCI

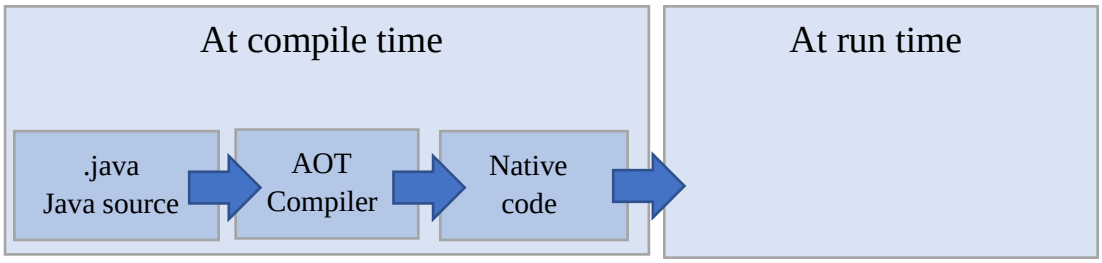
Źródło: opracowanie własne.

Wraz z wprowadzeniem interfejsu *kompilatora JIT*, za ciosem twórcy wprowadzili kilka innych eksperymentalnych zmian. Pierwsza daje możliwość użycia opcjonalnego kompilatora Graal, który został stworzony przez Oracle w ramach projektu GraalVM [28]. Graal to wysokowydajny *kompilator JIT*, którego celem jest zastąpienie standardowego kompilatora. Został napisany w języku Java, co oznacza, że otrzymamy wszystkie zalety pisania aplikacji w Java w porównaniu do języka C++, czyli wyjątki zamiast awarii i brak prawdziwych wycieków pamięci. Dodatkowo wsparcie dla IDE oraz możliwość debuggowania

[26] Oracle Corporation, *JEP 243: Java-Level JVM Compiler Interface*, [w.] https://openjdk.java.net/jeps/243 [15.04.2021]
[27] Baeldung, *Deep Dive Into the New Java JIT Compiler – Graal*, [w.] https://www.baeldung.com/graal-java-jit-compiler [15.04.2021]
[28] R. Larsson, Evaluation of GraalVM Performance for Java Programs, Department of computer science and media technology (CM), 2020, s. 1-6, [w.] https://www.diva-portal.org/smash/get/diva2:1457592/FULLTEXT01.pdf [17.04.2021]

i profilowania. Najważniejsza zaletą jest ciągły rozwój i możliwość wprowadzenia nowych optymalizacji. Zmiana została zarejestrowana jako eksperymentalna o kodzie JEP 317 [29].

Kolejną zmianą jest wprowadzenie kompilatora *Ahead-of-Time*, zwany kompilatorem AOT. Został wpisany do listy poprawek jako JEP 295. [30] Narzędzie ma na celu poprawę tak zwanego okresu rozgrzewania i skompilowanie klas do kodu natywnego już przed uruchomieniem maszyny wirtualnej. Proces kompilowania kodu źródłowego przy pomocy *kompilatora AOT* został przedstawiony na rysunku 12. Podobnie jak w poprzedniej zmianie został tutaj użyty *kompilator AOT* z projektu GraalVM. Świetnym przypadkiem użycia *kompilatora AOT* są krótko działające programy, które kończą wykonywanie jeszcze przed uruchomieniem *kompilatora JIT*. Temat przedwczesnego kompilowania do kodu natywnego nie zostanie uwzględniony w badaniach niniejszej pracy [31].



Rys. 12. Proces kompilowania kodu źródłowego przy pomocy kompilatora AOT

Źródło: opracowanie własne.

W Javie 9 dla kompilatora i wirtualnej maszyny Javy wprowadzono segmentacje pamięci podręcznej kodu. Pamięć podręczna kodu to obszar pamięci, w której przechowywany jest wygenerowany natywny kod aplikacji przez *kompilator JIT*. Kod zmiany to JEP 197 [32]. Celem jest podział sterty kodu na odrębne segmenty: niemetodowy, profilowany i nieprofilowany. Zostało to przedstawione w tabeli 1. Efektem wprowadzonych zmian jest krótszy czas oczyszczania pamięci, zmniejszona fragmentacja zoptymalizowanego kodu, lepsza kontrola

[29] Oracle Corporation, *JEP 317: Experimental Java-Based JIT Compiler*, [w.] https://openjdk.java.net/jeps/317 [15.04.2021]
[30] Oracle Corporation, *JEP 295: Ahead-of-Time Compilation*, [w.] https://openjdk.java.net/jeps/295 [15.04.2021]
[31] C. Seaton, *Understanding How Graal Works - a Java JIT Compiler Written in Java*, [w.] https://chrisseaton.com/truffleruby/jokerconf17/ [15.04.2021]
[32] Oracle Corporation, *JEP 197: Segmented Code Cache*, [w.] https://openjdk.java.net/jeps/197 [15.04.2021]

nad zużyciem pamięci maszyny JVM, a co za idzie to skrócenie czasu skanowania skompilowanych metod oraz poprawienie wydajności [33].

Tabela 1. Podział pamięci podręcznej kodu na segmenty

	Code Cache		
	Non-Method Code	Profiled Code	Non-Profiled Code
Compiled Code	Compiler buffers, bytecode interpreter	Lightly optimized profiled methods	Fully optimized non-profiled methods
Lifespan	Parmanent	Short lifetime	Long lifetime

Źródło: opracowanie własne.

1.3.3. Kompilacja i czas wykonywania

Tak jak zostało wspomniane w podrozdziale 1.3.2, wirtualna maszyna Javy w tracie działania aplikacji zbiera informacje o wykonywanym kodzie co umożliwia optymalizacje na etapie wykonywania kodu. Od wersji 9 wprowadzono kilka istotnych zmian działających w trakcie wykonywania programu, które mają wpływ na wydajność. Także uwzględniono zmiany, które wpływają na program w trakcie kompilowania kodu źródłowego do kodu bajtowego [34]. Do najważniejszych zmian należy:

- JEP 254: *Compact Strings* – zmiany wprowadzone w wersji 9, które obejmują komponent *core-libs/java.lang*. Wprowadzone ulepszenie zmienia wewnętrzną reprezentację klasy *String* i klas pochodnych. Dokładnie mówiąc to pojedynczy znak z klasy *String* będzie można reprezentować jako jeden bajt zamiast dwóch bajtów. Skutkować będzie to zmniejszeniem ilości miejsca wymaganego do przechowywania ciągu znaków nawet o połowę. Główny cel twórców to poprawa wydajności. Optymalizacja będzie miała miejsce tylko dla znaków o kodowaniu *Latin-1*. W kodowaniu *Latin-1*, można przedstawić większość języków wychodzących z alfabetu łacińskiego [35].

[33] E. Lavieri, P. Verhas, Mastering Java 9, *Segmented code cache [JEP 197]*, Packt, [w.] https://subscription.packtpub.com/book/application_development/9781786468734/2/ch02lvl1sec19/segmented-code-cache-jep-197 [15.04.2021]
[34] Microsoft Corporation, *Reasons to move to Java 11*, [w.] https://docs.microsoft.com/en-us/azure/developer/java/fundamentals/reasons-to-move-to-java-11 [17.04.2021]
[35] Oracle Corporation, *JEP 254: Compact Strings*, [w.] https://openjdk.java.net/jeps/254 [15.04.2021]

- JEP 310: *Application Class-Data Sharing* – zmiany wprowadzone w wersji 10 [36]. Celem projektu jest rozszerzenie istniejącej funkcji *Class-Data Sharing*, która została wprowadzona w 5 wydaniu Javy. Wprowadzając rozszerzenie, będzie możliwe umieszczenie klas aplikacji we współużytkowanym archiwum. A co za tym idzie, to przyśpieszenie czasu uruchomienia aplikacji i oszczędność miejsca w pamięci używanej przez metadane klasy Java. Biorąc pod uwagę wytwarzane w korporacjach aplikacje produkcyjne może mieć to znaczny wpływ na wydajność [37].
- JEP 312: *Thread-Local Handshakes* – zmiana wprowadzona w wersji 10. JEP dotyczy środowiska uruchomieniowego i nie wpływa na interfejs API. Celem projektu jest umożliwienie wykonania wywołania zwrotnego w wątkach bez wykonywania globalnego punktu bezpiecznego maszyny wirtualnej. Pozwoli to na osiągnięcie mniejszego opóźnienia poprzez zmniejszenie liczby globalnych punktów bezpieczeństwa [38].
- JEP 220: *Modular Run-Time Images* – koncepcja modularnych obrazów środowiska uruchomieniowego została wprowadzona w wersji 9. Zmiany pozwalają na generowanie niestandardowych obrazów JRE, które zawierają tylko moduły wymagane do uruchomienia i działania aplikacji. Wprowadzenie modułów ma na celu poprawić wydajność, bezpieczeństwo i oszczędność pamięci [39].

1.3.4. Zmiany w podstawowych bibliotekach

Oprócz samych zmian w wirtualnej maszynie Javy, twórcy wprowadzili drobne poprawki do podstawowych bibliotek załączonych w JDK. Zmiany wprowadzone w bibliotekach mogą mieć mniejsze lub większe znaczenie w wydajności aplikacji napisanych w Java od wersji 9. Do najważniejszych zmian należy:

- JEP 193: *Variable Handles* i JEP 266: *More Concurrency Updates* – zmiany wprowadzone wraz z wersją 9 i obejmują komponenty *core-libs/java.lang* i *core-libs / java.util.concurrent*. JEP 193 definiuje standardowe sposoby do wywołania

[36] Oracle Corporation, *JEP 310: Application Class-Data Sharing*, [w.] https://openjdk.java.net/jeps/310 [15.04.2021]
[37] N. Samoylov, M. Sanaulla, *Using application class-data sharing*, W Java 11 Cookbook - Second Edition, Packt, [w.] https://subscription.packtpub.com/book/application_development/9781789132359/1/ch01lvl1sec16/using-application-class-data-sharing [15.04.2021]
[38] Oracle Corporation, *JEP 312: Thread-Local Handshakes*, [w.] https://openjdk.java.net/jeps/312 [15.04.2021]
[39] Oracle Corporation, *JEP 220: Modular Run-Time Images*, [w.] https://openjdk.java.net/jeps/220 [15.04.2021]

odpowiedników dla zmiennych z pakietów *java.util.concurrent.atomic* i *sun.misc* [40]. Wprowadzają ulepszenia w zakresie współbieżności i równoległości w aplikacjach. Założonymi celami są: bezpieczeństwo, integralność, wydajność i użyteczność. [41]

- JEP 269: *Convenience Factory Methods for Collections* – zmiany wprowadzone w wersji 9 i obejmuje komponent *core-libs / java.util:collections*. Definiuje API, które ułatwi tworzenie instancji kolekcji z małą liczbą elementów. Za pomocą wprowadzonego API, można utworzyć kompaktowe i niemodyfikowalne kolekcje. Jak sami twórcy wspomnieli, celem nie jest polepszenie wydajności, ale użycie może wpłynąć na wykonywany kod [42].
- JEP 285: *Spin-Wait Hints* – zmiana wprowadzona w wersji 9, które obejmuje komponent *core-libs/java.lang* [43]. Definiuje interfejs API, który pozwala zasugerować systemowi, że znajduje się w pętli *spin*. Pętla spinowa lub technika zajętego oczekiwania to pętla, w której określony warunek jest wielokrotnie sprawdzany. Przykładem takiej pętli jest kod znajdujący się na rysunku 13. Za pomocą tego mechanizmu w niektórych procesorach można poprawić wydajność wykonywania kodu [44].

```
class EventHandler {
    volatile boolean eventNotificationNotReceived;
    void waitForEventAndHandleIt() {
        while ( eventNotificationNotReceived ) {
            java.lang.Thread.onSpinWait();
        }
        readAndProcessEvent();
    }

    void readAndProcessEvent() {
        // Read event from some source and process it
          . . .
    }
}
```

Rys. 13. Przykład pętli spin

Źródło: [45]

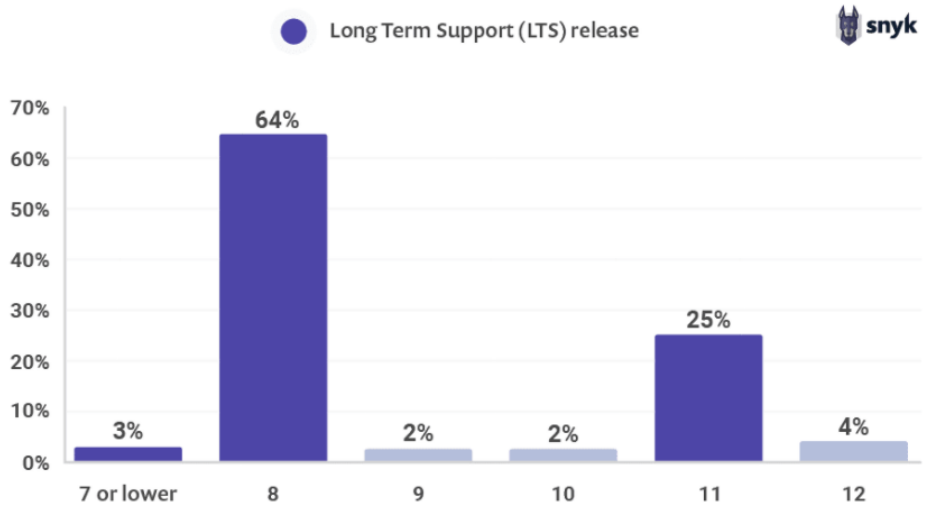
[40] Oracle Corporation, *JEP 193: Variable Handles*, [w.] https://openjdk.java.net/jeps/193 [15.04.2021]
[41] Oracle Corporation, *JEP 266: More Concurrency Updates*, [w.] https://openjdk.java.net/jeps/266 [15.04.2021]
[42] Oracle Corporation, *JEP 269: Convenience Factory Methods for Collections*, [w.] https://openjdk.java.net/jeps/269 [15.04.2021]
[43] Oracle Corporation, *JEP 285: Spin-Wait Hints*, [w.] https://openjdk.java.net/jeps/285 [15.04.2021]
[44] M. Balci, *Spin-Wait Hints in Java*, [w.] https://metebalci.com/blog/spin-wait-hints-in-java/ [15.04.2021]
[45] Oracle Corporation, Class Thread, [w.] https://docs.oracle.com/javase/10/docs/api/java/lang/Thread.html [15.04.2021]

- JEP 321: *HTTP Client* – zmiana wprowadzona w wersji 11 i obejmuje komponent *core-libs/java.net.* Zapewnia nowe API klienta http, które oprócz zaimplementowania *http/2* i *websocket* postawiono za cel przystosowanie się do aktualnych wymagań bezpieczeństwa i wydajności. Warunkiem wprowadzenia tej zmiany, było spełnienie celu, który wymagał od nowego API, aby zużycie pamięci było równe lub niższe od istniejącego rozwiązania [46].
- JEP 332: *Transport Layer Security (TLS) 1.3* – zmiany wprowadzone w wersji 11 dla komponentu *security-libs / javax.net.ssl.* Celem było zaimplementowanie *Transport Layer Security* w wersji 1.3. Skutkuje to poprawą bezpieczeństwa i wydajności w porównaniu z poprzednimi wersjami [47].

1.4. Zarys problemu wykorzystania Javy

Pomimo tak dużych różnic pomiędzy wersją 8 i 11, to nadal Java w wersji 8 dominuje na rynku według przeprowadzonych ankiet. W 2020 roku serwis snyk.io przeprowadził badania wśród programistów. Badania dotyczyły wersji Javy używanej w środowisku produkcyjnym. Opierając się na statystykach przedstawionych na rysunku 14 to jedynie 16% badanych programistów zadeklarowało używanie Javy w wersji z krótkim wsparciem technicznym lub wersji starszej niż 8. Jedna czwarta programistów używa wersji 11 w produkcji, a aż 64% używa wersji 8. Statystyki potwierdzają stwierdzenie, że liderzy projektów chętniej wybierają Java w wersji stabilniejszej z zapewnionym długoterminowym wsparciem. Dla 51% respondentów głównym argumentem przed wykonaniem migracji to „obecna konfiguracja działa dobrze, więc zmiana nie jest potrzebna”. Pozostała część twierdzi, że koszt migracji wydaje się zbyt wysoki lub serwery nie obsługują najnowszych wersji JDK.

[46] Oracle Corporation, *JEP 321: HTTP Client*, [w.] https://openjdk.java.net/jeps/321 [15.04.2021]
[47] Oracle Corporation, *JEP 332: Transport Layer Security (TLS) 1.3*, [w.] https://openjdk.java.net/jeps/332 [15.04.2021]



Rys. 14. Statystyki używania wersji Java SE w środowisku produkcyjnym

Źródło: [48]

Java to dynamicznie rozwijający się język programowania. Na czas pisania pracy najbardziej popularną wersją Javy jest wersja 8 oraz 11. Wersja 11 oferuje korzyści nie tylko w jakości wytwarzanego kodu, a także na poziomie środowiska uruchomieniowego. Teoretycznie pomiędzy wersjami poprawiło się wiele aspektów pod względem wydajnościowym. Pomimo tego duża część użytkowników dalej decyduje się na pozostaniu przy wersji 8. Nie zapominając o jednej z najważniejszych cech Javy, którą jest kompatybilność wsteczna. Dzięki temu migrowanie kodu aplikacji powinno być stosunkowo proste.

Po przeglądzie literatury można stwierdzić, że w literaturze powstała nisza, dotycząca porównania obu wersji pod względem wydajnościowym i stosunkowo małe skupienie uwagi na wydajność wytwarzanego oprogramowania. Celem badań jest odpowiedź na powstałą niszę, poprzez przeprowadzenie analizy wydajności algorytmów zaprojektowanych w wspomnianych wersjach.

[48] SnykBlog, *Which Java SE version do you use in production for your main application?*, [w.] https://snyk.io/blog/developers-dont-want-to-leave-java-8-as-64-hold-firm-on-their-preferred-release [31.03.2021]

2. BADANIA WYDAJNOŚCIOWE OPROGRAMOWANIA

W niniejszym rozdziale został zdefiniowany obszar badawczy. Omówione są kolejno: cel i zakres badań, pytania badawcze oraz wstępne spostrzeżenia oparte na przeglądzie literatury. Następne przeprowadzono przegląd wskaźników do pomiaru wydajności języka Java oraz narzędzi, które wspomogą przeprowadzenie badań. Na koniec rozdziału został przedstawiony plan wykonanych badań, które opierają się na metodach symulacji komputerowej.

2.1. Cel i zakres badań

Przedmiotem badania są aplikacje zaprojektowane w języku programowania Java w wersji 8 oraz w wersji 11. Celem badań jest analiza i ocena wydajności dwóch najnowszych wersji LTS dla tego języka.

Język programowania Java rozwija się bardzo dynamicznie w ostatnim czasie, a potwierdza to przegląd literatury. Twórcy platformy Java wprowadzają wiele udogodnień dzięki czemu jakość i czytelność kodu znacznie wzrasta, a proporcjonalnie zmniejszają się koszty tworzenia i utrzymania aplikacji. Drugim zapominanym filarem przez wielu programistów i menadżerów projektowych jest obszar, który kryje wirtualna maszyna Javy. Z każdą kolejną publikacją wersji Javy często wiążą się zmiany, które gołym okiem nie widać, a wpływają znacząco na wydajność działania aplikacji.

Zebrane informacje z przeglądu literatury oraz przeprowadzonego eksperymentu pomogą zdiagnozować źródła efektywnego wytwarzania i działania oprogramowania. Podany obszar badawczy pozwoli poznać wpływ zmiany wersji Java na wydajność oprogramowania.

2.2. Pytania badawcze

Przeprowadzone badania mają odpowiedzieć na poniżej postawione pytania badawcze, które zostały opracowane na podstawie przeanalizowanego stanu literatury.

1. Czy warto migrować stare aplikacje napisane przed laty do nowszych wersji Javy?
2. Czy zawsze przy projektowaniu aplikacji należy wybierać najnowszą wersję Javy?
3. Czy wraz z nowszą wersją Javy pojawia się lepsza wydajność aplikacji?
4. Czy zmiana wersji Javy wpływa na czas działania aplikacji?
5. Czy zmiany w domyślnych bibliotekach mają wpływ na wydajność aplikacji?

6. Jaki wpływ ma wybór modułu czyszczenia pamięci na czasy wykonywania aplikacji?
7. Czy wymiana *kompilatora JIT* będzie miał duży wpływ na wydajność aplikacji?

2.3. Wstępne spostrzeżenia

Na podstawie przeanalizowanego stanu literatury wyciągnięto następujące wstępne spostrzeżenia:

- Między obiema wersjami wprowadzono zmiany w domyślnych bibliotekach. Biblioteki mogą różnić się sposobem implementacji i użytymi strukturami, co zapewne będzie miało wpływ na wydajność aplikacji.
- Obie wersje różni sposób reprezentowania klasy *String*. W porównaniu do poprzednich wersji, w Javie 11 klasy reprezentujące ciągi znaków przechowują znaki jako jeden bajt dla każdego znaku zakodowanego jako *Latin-1*. Dlatego będzie odczuwalne mniejsze zużycie pamięci i zmniejszenie aktywności usuwania elementów bezużytecznych.
- Ze względu na sposób implementacji nowego modułu usuwania bezużytecznych obiektów z sterty będziemy oczekiwali poprawy wydajności aplikacji podczas operowania na bardzo dużych obiektach. W przypadku operacji na obiektach o mniejszych rozmiarach możemy odczuwać nieznaczne pogorszenie wydajności na korzyść Java 8.
- Ze względu na dłuższą przerwę w zmianach dla *kompilatora JIT* i udostępnienie portu do wprowadzenia zewnętrznego modułu optymalizacji kodu, będziemy oczekiwać poprawy wydajności w przypadku użycia *kompilatora Graal*.
- W Java 11 wprowadzono segmentację pamięci podręcznej kodu, dzięki temu może się polepszyć wydajność aplikacji w momencie uruchamiania *kompilatora JIT* w Java 11 w porównaniu do starszych wersji.

2.4. Metody i narzędzia do badań wydajnościowych

Analiza i porównywanie wydajności aplikacji jest dziedziną obszerną i rozwianą od bardzo dawna. Powstało wiele artykułów i badań na ten temat. Zaczynając od analizy porównawczej różnych języków programowania, poprzez porównanie środowisk komputerowych, a kończąc na systemach operacyjnych. Zanim zostanie wyjaśnione w jaki sposób można przeprowadzać testy wydajnościowe należy wyjaśnić czym jest test

wydajnościowy. Testy wydajnościowe należą do grupy testów niefunkcjonalnych i służą do sprawdzenia zachowania aplikacji przez zastosowanie pewnego obciążenia.

Pomiary wydajności technologii możemy wykonać na kilka sposobów. Możemy wyróżnić dwa główne podejścia: *black box* oraz *white box*. W modelu testów *black box* tester nie ma żadnych informacji o wewnętrznym działaniu aplikacji. Ten model testów najczęściej używany jest w przypadku, gdy technologie w których została wytworzona aplikacja mocno się różnią lub celem testów są wyższe warstwy aplikacji na przykład przetwarzanie żądań http. Model *white box* charakteryzuje się projektowaniem testów wydajnościowych z myślą o testowanych narzędziach i użytych technologiach. Przykładem mogą posłużyć tutaj narzędzia z funkcją profilowania pamięci w Javie za pośrednictwem JVMTI. Taka możliwość zostało wprowadzone w Javie 11 pod kodem JEP 331, które celem jest zapewnienie wydajne próbowanie alokacji sterty Java [49]. W przypadku aplikacji serwerowych w obu modelach można używać zewnętrznych narzędzi do pomiarów wydajności [50]. W skład badań wydajnościowych wchodzą:

- współczynniki wydajności, które wpływają na wydajność aplikacji,
- metryki,
- metody i narzędzia do pomiarów wydajności.

W niniejszej pracy testy wydajnościowe zostały przeprowadzone metodą *white box*. Poniżej zostały wymienione istotne czynniki, które wpływają na wydajność programów Java:

- zużycie pamięci,
- zajętość procesora.

Do sporządzania wyników skorzystano z dwóch narzędzi przeznaczonych do aplikacji uruchamianych w wirtualnej maszynie Java. Pierwszym narzędziem, na którym oparte zostały testy jest biblioteka JMH – Java Microbenchmark Harness. JMH to narzędzie do budowania, wykonywania automatycznie powtarzalnych wywołań kodu, symulowania kontekstu produkcyjnego i monitorowania kodu za pomocą mikro-testów. Dzięki temu jest możliwość utworzenia miarodajnych i precyzyjnych testów wydajnościowych unikając problemów z usuwaniem kodu przez *kompilator JIT* czy symulowanie dużej ilości wywołań danej metody. Produkt został opracowany przez projekt OpenJDK [51].

[49] Oracle Corporation, *JEP 331: Low-Overhead Heap Profiling*, [w.] http://openjdk.java.net/jeps/331 [17.04.2021]
[50] A. Haiut, *White Box Vs. Black Box in Load Testing*, [w.] https://www.blazemeter.com/blog/white-box-vs-black-box-load-testing [15.04.2021]
[51] Oracle Corporation, *Code Tools: jmh*, [w.] https://openjdk.java.net/projects/code-tools/jmh/ [12.05.2021]

Drugim pomocniczym narzędziem jest VisualVM. Narzędzie pozwala na wizualne monitorowanie zasobów serwera oraz integruje polecenia JDK wiersza poleceń i lekkie funkcje profilowania. Podobnie jak wcześniejsze narzędzie zostało opracowane przez projekt OpenJDK [52].

2.5. Plan badań

Symulacja komputerowa została przeprowadzona z wykorzystaniem dwóch wersji Java. Obie wersje zostały pobrane od podmiotu AdoptOpenJDK.

Pierwsza wersja:

- openjdk version "1.8.0_292",
- OpenJDK Runtime Environment (AdoptOpenJDK) (build 1.8.0_292-b10),
- OpenJDK 64-Bit Server VM (AdoptOpenJDK) (build 25.292-b10, mixed mode).

Druga wersja:

- openjdk version "11.0.11" 2021-04-20,
- OpenJDK Runtime Environment AdoptOpenJDK-11.0.11+9 (build 11.0.11+9),
- OpenJDK 64-Bit Server VM AdoptOpenJDK-11.0.11+9 (build 11.0.11+9, mixed mode).

W celu odwzorowania środowiska produkcyjnego, w którym uruchamiane są aplikacje zaprojektowane dla firm, skorzystano z usług Google Cloud. Testy zostały przeprowadzone na dwóch różnych odizolowanych instancjach o tych samych parametrach. Szczegóły maszyn wirtualnych znajdują się w tabeli 2.

Tabela 2. Szczegóły maszyny wirtualnej

Typ maszyny	e2-standard-4
Pamięć RAM	32 GB
Procesor	Intel Xeon E5 v4 (Broadwell E5), 8 procesorów wirtualnych, bazowa częstotliwość 2.2GHz z przyspieszeniem do 2.8GHz
System operacyjny	Canonical, Ubuntu, 20.04 LTS
Dysk rozruchowy	10GB

Źródło: opracowanie własne.

[52] Oracle Corporation, *VisualVM*, [w.] https://visualvm.github.io/ [12.05.2021]

Do przeprowadzenia badań zostały zdefiniowane i zaimplementowane algorytmy. Testy zostały wykonane przy użyciu tego samego kodu źródłowego. Zbudowane i uruchomione na odpowiedniej wersji Javy. Każdy test został uruchomiony z tą samą konfiguracją Javy, z wyjątkiem testów, w których jawnie napisano, że zostały użyte inne parametry. Szczegóły uruchomienia testów w tabeli 3.

Tabela 3. Parametry do uruchomienia testów

Parametry do uruchomienia testów	
Opis	Parametr
Ustawia maksymalny rozmiar sterty na 2GB. Zawsze ustawiane.	-Xmx2g
Ustawia początkową wielkość sterty Java na 2GB. Zawsze ustawiane.	-Xms2g
Wstępnie wyzerowanie każdej strony sterty na żądanie podczas inicjalizacji maszyny JVM, a nie przyrostowo podczas wykonywania aplikacji.	-XX:+AlwaysPreTouch
Ustawia odpowiedni moduł do oczyszczania pamięci.	-XX:+UseG1GC; -XX:+UseParallelGC; -XX:+UseSerialGC; -XX:+UseZGC
Włącza *Graal* jako *kompilator JIT*. Ustawiane przy wybranych testach.	-XX:+UnlockExperimentalVMOptions; -XX:+UseJVMCICompiler

Źródło: opracowanie własne.

Konfiguracja narzędzia JHM dla wszystkich testów wygląda podobnie. Każdy test porównawczy wykorzystuje 5 iteracji rozgrzewających wirtualną maszynę Javy po 10 sekund i 15 iteracji pomiarowych po 10 sekund. Testy zostały wykonane tylko jeden raz z wykorzystaniem jednego wątku. Wynikiem danego testu jest średnia ze wszystkich iteracji. W celu obniżenia poziomu szumów w testach porównawczych wymuszono „dotkniecie" każdej strony sterty podczas inicjalizacji maszyny JVM, pozwoli to na uniknięcie zmiany rozmiaru i postojów związanych z zatwierdzania pamięci.

W celu przeprowadzenia badań porównawczych utworzone zostały 11 zbiorów testów. Testy zostały tak zaprojektowane, aby uwzględnić jak najwięcej wprowadzonych zmian do Javy 11, zaczynając od porównania modułów GC, poprzez zmiany w bibliotekach i API, a kończąc na testach kompilatorów. W zbiorze nie zabrakło algorytmów, które zdarza się

wykorzystywać w środowisku produkcyjnym. Algorytmy zostały zaprojektowane zgodnie z zaleceniami od JHM w celu uniknięcia optymalizacji, które nie mogłyby się zdarzyć w systemie produkcyjnym. Jednym z takich przykładów mogą być metody, które nie zwracają żadnych wyników. Kompilator traktuje takie operacje jako martwy kod, co powoduje pominięcie jego wykonywania.

Aby podkreślić jaki wpływ mają zmiany w bibliotekach na wydajność aplikacji, zaprojektowano i poddano analizie następujące testy:

- *CountUppercaseBenchmark [CUB]* – zliczenie N razy wystąpień wielkich liter w ciągu znaków przekazanym przez argument. Implementacja przedstawiona została na listingu 1.

Listing 1. Zliczanie n razy wystąpień wielkich liter w ciągu znaków

```
1.  public long countUppercase(Plan plan) {
2.      long total = 0;
3.      for (int i = 0; i < plan.iterations; i++) {
4.          total += plan.value.chars().filter(Character::isUpperCase).count();
5.      }
6.      return total;
7.  }
```

Źródło: opracowanie własne.

- *LevenshteinDistanceBenchmark [LDB]* – implementacja algorytmu obliczającego odległość edycyjną. Algorytm przyjmuje 2 argumenty w postaci dwóch ciągów znaków a wynikiem jest liczba oznaczająca odległość edycyjną dwóch ciągów [53].
- *SplittingListBenchmark [SLB]* - przepakowanie listy elementów do niemodyfikowalnych zbiorów 5-elementowych. W przypadku Javy 11 użyto nowego api. Kod źródłowy został przedstawione na listingu 2.

Listing 2. Przepakowanie listy elementów do niemodyfikowalnych zbiorów 5-elementowych

```
1.    public List<Set<Integer>> split(Plan plan) {
2.        final List<Set<Integer>> result = new LinkedList<>();
3.        final int[] numbers = plan.numbers;
4.        for (int i = 0; i < numbers.length; i = i + 5) {
5.            final Set<Integer> values = toSet(
6.                    numbers[i], numbers[i + 1], numbers[i + 2],
7.                    numbers[i + 3], numbers[i + 4]
```

[53] Baeldung, *How to Calculate Levenshtein Distance in Java?*, [w.] https://www.baeldung.com/java-levenshtein-distance [12.05.2021]

```
            );
            result.add(values);
        }
        return result;
    }

    // metoda w wersji Java 11
    private <T> Set<T> toSet(T v1, T v2, T v3, T v4, T v5) {
        return Set.of(v1, v2, v3, v4, v5);
    }

    // metoda w wersji Java 8
    private <T> Set<T> toSet(T v1, T v2, T v3, T v4, T v5) {
        Set<T> set = new HashSet<>();
        set.add(v1);
        set.add(v2);
        set.add(v3);
        set.add(v4);
        set.add(v5);
        return Collections.unmodifiableSet(set);
    }
```

Źródło: opracowanie własne.

- *StreamBenchmark [SB]* – zbiór testów wykonujące proste operacje na *stream'ach*. Implementacja została przedstawiona na listingu 3. Zbiór zawiera następujące testy:
    - *mapToPair* [*test1*] - przepakowanie elementów do listy par. Algorytm został przedstawiony na listingu 3.

Listing 3. Przepakowanie listy elementów do par

```
public List<Pair<Integer, Integer>> mapToPair(Plan plan) {
    return IntStream.range(0, plan.array.length)
            .boxed()
            .map(index -> new Pair<>(index, plan.array[index]))
            .collect(Collectors.toList());
}
```

Źródło: opracowanie własne.

- *plusOne* [*test2*] - dodanie wartości 1 do każdego elementu w liście. Algorytm został przedstawiony na listingu 4.

Listing 4. Dodanie liczby 1 do każdego elementu listy

```
1.  public List<Integer> plusOne(Plan plan) {
2.      return Arrays.stream(plan.array)
3.              .map(number -> number + 1)
4.              .boxed()
5.              .collect(Collectors.toList());
6.  }
```

Źródło: opracowanie własne.

- *sort* [*test3*] – sortowanie kolekcji elementów. Algorytm został przedstawiony na listingu 5.

Listing 5. Algorytm sortowania oparty na stream

```
1.  public List<Integer> sort(Plan plan) {
2.      return Arrays.stream(plan.array)
3.              .sorted()
4.              .boxed()
5.              .collect(Collectors.toList());
6.  }
```

Źródło: opracowanie własne.

- *StringConcatenationBenchmark [SCB]* – naiwne scalanie ciągów znaków o różnych wielkościach. Test ma na celu potwierdzić jedno z wymienionych spostrzeżeń, a dokładnie czy zmiana reprezentacji ciągów znaków wpłynie znacząco na wydajność. Algorytm został przedstawiony na listingu 6.

Listing 6. Konkatenacja ciągów znaków

```
1.  public String concatenateStrings(Plan plan) {
2.      String result = "";
3.      for (String value : plan.values) {
4.          result = result + value;
5.      }
6.      return result;
7.  }
```

Źródło: opracowanie własne.

W celu ustalenia jaki wpływ ma wybór modułu *garbage collectora* na wydajność aplikacji zaimplementowano do analizy następujące testy:

- *TreeBenchmark [TB]* - test ma na celu sprawdzić wydajność modułów GC w przypadku obiektów głęboko zagnieżdżonych. Podczas testów tworzone są zrównoważone drzewa binarne o 1023 wierzchołkach i 1024 liści. Podczas jednej iteracji tworzone są 1024 takich drzew. W jednym teście uruchamiane są 10 iteracji. Implementacja została przedstawiona na listingu 7.

Listing 7. Algorytm tworzący listę drzew

```
1.  static class Product {
2.      private final Product product1;
3.      private final Product product2;
4.      private final int total;
5.      private final byte[] bytes;
6.
7.      public Product(Product product1, Product product2, int total, byte[] bytes) {
8.          this.product1 = product1;
9.          this.product2 = product2;
10.         this.total = total;
11.         this.bytes = bytes;
12.     }
13.
14.     public static Product createRecursive(int total) {
15.         if (total <= 1) {
16.             return new Product(null, null, total, new byte[512]);
17.         }
18.
19.         final Product product1 = Product.createRecursive((total / 2));
20.         final Product product2 = Product.createRecursive((total / 2));
21.         return new Product(product1, product2, total, new byte[512]);
22.     }
23. }
24.
25. public void createNewObjectsWithRecursive(Plan plan, Blackhole blackhole) {
26.     for (int iter = 0; iter < 10; iter++) {
27.         List<Product> objects = new ArrayList<>(plan.numberOfObjects);
28.         for (int i = 0; i < plan.numberOfObjects; i++) {
29.             objects.add(Product.createRecursive(plan.size));
30.         }
31.         blackhole.consume(objects);
32.     }
33. }
```

Źródło: opracowanie własne.

- *ArrayBenchmark [AB]* – zbiór testów, które mają na celu sprawdzenie wydajności aplikacji podczas odczytywania, zapisywania i zamianę elementów miejscami w tablicy. W przypadku zamiany elementów miejscami w tablicy posłużył algorytm sortowania bąbelkowego. Zbiór zawiera następujące testy:
    - *read [test1]* – odczytuje wszystkie elementy z tablicy i sumuje. Algorytm został przedstawiony na listingu 8.

Listing 8. Czytanie elementów tablicy

```
1.  public long read(Plan plan) {
2.      long sum = 0;
3.      for (int i = 0; i < plan.array.length; i++) {
4.          sum += plan.array[i];
5.      }
6.      return sum;
7.  }
```

Źródło: opracowanie własne.

    - *readAndWrite [test2]* – do każdego elementu z tablicy dodaje wartość przekazaną poprzez argument. Algorytm został przedstawiony na listingu 9.

Listing 9. Modyfikacja elementów tablicy

```
1.  public int[] readAndWrite(Plan plan) {
2.      for (int i = 0; i < plan.array.length; i++) {
3.          plan.array[i] = plan.element + plan.array[i];
4.      }
5.      return plan.array;
6.  }
```

Źródło: opracowanie własne.

- *swap [test3]* – sortowanie bąbelkowe, które polega na porównaniach elementów z tablicy i zamianę miejscami. Algorytm został przedstawiony na listingu 10.

Listing 10. Zamiana miejscami elementów w tablicy na przykładzie sortowania bąbelkowego

```
1.  public int[] swap(Plan plan) {
2.      final int[] unsortedArray = plan.unsortedArray;
3.      final int length = unsortedArray.length;
4.
5.      for (int i = 0; i < length; i++) {
6.          for (int j = 1; j < (length - i); j++) {
7.              if (unsortedArray[j - 1] > unsortedArray[j]) {
8.                  final int temp = unsortedArray[j - 1];
9.                  unsortedArray[j - 1] = unsortedArray[j];
10.                 unsortedArray[j] = temp;
11.             }
12.         }
13.     }
14.     return unsortedArray;
15. }
```

Źródło: opracowanie własne.

- *write [test4]* – podmienia wszystkie elementy z tablicy wartością przekazaną poprzez argument. Algorytm został przedstawiony na listingu 11.

Listing 11. Zbiór testów dla ArrayBenchmark

```
16. public int[] write(Plan plan) {
17.     for (int i = 0; i < plan.array.length; i++) {
18.         plan.array[i] = plan.element;
19.     }
20.     return plan.array;
21. }
```

Źródło: opracowanie własne.

- *AllocationBenchmark [ALLB]* – zbiór testów, które mają na celu porównanie zachowania modułów GC w przypadku alokowania małych jak i dużych obiektów na stercie. Implementacje zapełniają około 70% sterty obiektami o rozmiarach 512B, 1KB, 2KB, 10KB, 100KB, 1MB, 2MB, 10MB i 100MB. Zbiór zawiera następujące testy:
  - *createNewObjects [test1]* – w każdej iteracji tworzona jest tablica o rozmiarze X bajtów i przekazywana do przetwarzania innej metodzie, co oznacza, że cykl

życia takiej tablicy jest bardzo krótki. Liczba iteracji jest w taki sposób ustalana, aby sterta była wypełniona 4 razy po 70% całkowitej pojemności. Tak zaprojektowany algorytm wymusi uruchomienie modułu do oczyszczania pamięci w trakcie testu. Implementacja została przedstawiona na listingu 12.

Listing 12. Alokowanie obiektów

```
1.  public void createNewObjects(Plan plan, Blackhole blackhole) {
2.      for (int i = 0; i < plan.numberOfObjects * 4; i++) {
3.          blackhole.consume(new byte[plan.size]);
4.      }
5.  }
```

Źródło: opracowanie własne.

- *fillHeap [test2]* – test zawiera dwie zawierające się w sobie pętle. W wewnętrznej pętli tworzone jest N obiektów i dodawane do listy, aby zapełnić 70% sterty. Następnie lista jest przekazywana do innej metody, co oznacza, że obiekty już nie będą używane. Cały ten cykl powtarzany jest 4 razy, aby moduł do oczyszczania pamięci uruchomił się w trakcie testu. Implementacja została przedstawiona na listingu 13.

Listing 13. Alokowanie obiektów w liście

```
1.  public void fillHeap(Plan plan, Blackhole blackhole) {
2.      for (int iter = 0; iter < 4; iter++) {
3.          List<byte[]> objects = new ArrayList<>(plan.numberOfObjects);
4.          for (int i = 0; i < plan.numberOfObjects; i++) {
5.              objects.add(new byte[plan.size]);
6.          }
7.          blackhole.consume(objects);
8.      }
9.  }
```

Źródło: opracowanie własne.

Do porównania *kompilatorów JIT* wybrano następujące testy:

- *FibonacciBenchmark [FIB]* - implementacja algorytmu obliczania n-tej liczby ciągu *fibonacciego* za pomocą rekurencji ogonowej. Listing 14 przedstawia implementacje danego testu.

Listing 14. Obliczanie n-tej liczby fibonacciego

```
1.  public long runFibonacci(FibonacciPlan fibonacciPlan) {
2.      return fib(fibonacciPlan.iterations, 0, 1);
3.  }
4.
5.  private long fib(long n, int a, int b) {
6.      if (n == 0) {
7.          return a;
8.      }
9.      if (n == 1) {
10.         return b;
11.     }
12.     return fib(n - 1, b, a + b);
13. }
```

Źródło: opracowanie własne.

- *JitCompilerBenchmark [JITB]* – zbiór testów utworzonych w celu porównania jakości optymalizacji kodu przez *kompilatory JIT*. Zbiór testów został przedstawiony na listingu 10. Testy mają na celu zmuszenie *kompilatora JIT* do następujących operacji:
  - *eliminateConditionsAlwaysTrue* [*test1*] - usunięcie warunków zawsze prawdziwych. Implementacja została przedstawiona na listingu 15.

Listing 15. Algorytm mający na celu wymusić usunięcie zbędnych warunków

```
1.  public long eliminateConditionsAlwaysTrue(SimplePlan state) {
2.      long sum = 0;
3.      for (Integer number : state.numbers) {
4.          if (number == null) {
5.              return Integer.MIN_VALUE;
6.          }
7.          if (number > state.sum && number < Integer.MAX_VALUE) {
8.              return number;
9.          }
10.         if (number < Integer.MIN_VALUE) {
```

```
11.             return Integer.MIN_VALUE;
12.         }
13.         if (sum < Long.MAX_VALUE) {
14.             if (sum > Long.MIN_VALUE) {
15.                 sum = number + sum;
16.             }
17.         }
18.     }
19.     return sum;
20. }
```
Źródło: opracowanie własne.

- *extractOperationOutOfLoop [test2]* - wyciągnięcia wyrażeń poza pętle. Implementacja została przedstawiona na listingu 16.

Listing 16. Algorytm mający na celu wyciągnięcie wyrażeń poza pętlę

```
1.  public double extractOperationOutOfLoop(SimplePlan state) {
2.      double sum = 0;
3.      double result = 0;
4.      double x = 100;
5.      for (int i = 1; i < state.iterations; i++) {
6.          double v1 = (Math.log((i + 1) * x) + Math.log(x)) * 2;
7.          double v2 = Math.log((i + 1) * x) * Math.log(x);
8.          sum += v1 / v2;
9.          if (i == state.iterations - 1) {
10.             result = Math.log(sum);
11.         }
12.     }
13.     return result;
14. }
```
Źródło: opracowanie własne.

- *reduceLoops* [*test3*] - zredukowanie wszystkich pętli do jednej. Implementacja została przedstawiona na listingu 17.

Listing 17. Algorytm mający na celu wymuszenie redukcji pętli

```
1.  public long reduceLoops(SimplePlan state) {
2.      long sum1 = 0;
3.      long sum2 = 0;
4.      long sum3 = 0;
5.      long iterations = 0;
6.      for (int i = 1; i < state.integerMaxValue; i++) {
7.          sum1 = i + 1;
```

```
8.      }
9.      for (int i = 1; i < state.integerMaxValue; i++) {
10.         sum2 += i;
11.     }
12.     for (int i = 1; i < state.integerMaxValue; i++) {
13.         sum3 -= i;
14.     }
15.     for (int i = 1; i < state.integerMaxValue; i++) {
16.         iterations = i;
17.     }
18.     return iterations / (sum1 + sum2 + sum3);
19. }
```

Źródło: opracowanie własne.

- *reduceSynchronizedBlocks* [*test4*] - zredukowanie synchronicznych bloków. Implementacja została przedstawiona na listingu 18.

Listing 18. Algorytm ma na celu zredukowanie synchronicznych bloków

```
1.  public long reduceSynchronizedBlocks() {
2.      long result = 0;
3.      synchronized (this) {
4.          long x = ThreadLocalRandom.current().nextInt();
5.          synchronized (this) {
6.              long y = ThreadLocalRandom.current().nextInt();
7.              synchronized (this) {
8.                  long z = ThreadLocalRandom.current().nextInt();
9.                  result = x + y + z;
10.             }
11.         }
12.     }
13.     synchronized (this) {
14.         long x = ThreadLocalRandom.current().nextInt();
15.         synchronized (this) {
16.             long y = ThreadLocalRandom.current().nextInt();
17.             synchronized (this) {
18.                 long z = ThreadLocalRandom.current().nextInt();
19.                 result = result + x - y - z;
20.             }
21.         }
22.     }
23.     return result;
24. }
```

Źródło: opracowanie własne.

- *reduceYoungObjects* [*test5*] - optymalizacje na podstawie obiektów o krótkich czasie życia. Implementacja została przedstawiona na listingu 19.

Listing 19. Algorytm mający na celu wymusić optymalizacje na podstawie krótkotrwałych obiektów

```
1.  public long reduceYoungObjects(SimplePlan state) {
2.      long sum = 0;
3.      for (int i = 0; i < state.iterations; i++) {
4.          final RandomNumber randomNumber1 = new RandomNumber();
5.          final RandomNumber randomNumber2 = new RandomNumber();
6.          sum += randomNumber1.getNumber() + randomNumber2.getNumber();
7.      }
8.      return sum;
9.  }
```

Źródło: opracowanie własne.

- *redundantCode* [*test6*] - usunięcie nieużywanego kodu. Implementacja została przedstawiona na listingu 20.

Listing 20. Algorytm mający na celu usunięcie nieużywanego kodu

```
1.  public String redundantCode(SimplePlan state) {
2.      final StringBuilder firstChars = new StringBuilder();
3.      for (Integer number : state.numbers) {
4.          final Integer searched = search(state.numbers, number);
5.          Supplier<String> stringSupplier = () -> "x";
6.          final String value = getValueFromSupplier(stringSupplier);
7.          final char c = value.charAt(0);
8.          firstChars.append(c);
9.      }
10.     return firstChars.toString();
11. }
12.
13. private Integer search(List<Integer> numbers, Integer searchedNumber) {
14.     for (Integer number : numbers) {
15.         if (number.equals(searchedNumber)) {
16.             return searchedNumber;
17.         }
18.     }
19.     return searchedNumber;
20. }
21.
22. private String getValueFromSupplier(Supplier<String> stringSupplier) {
```

```
23.     if (stringSupplier == null) {
24.         throw new IllegalArgumentException();
25.     }
26.     return stringSupplier.get();
27. }
```
Źródło: opracowanie własne.

- *CalculatorBenchmark [CALC]* – test zawiera wszystkie przypadki z *JitCompilerBenchmarks*. Implementacja sumuje przekazane do metody liczby typu *BigDecimal* wykonując poboczne operacje z brakiem skutków ubocznych.

Wymienione testy zostały uruchomione w następujących kombinacjach:

- Java 8, moduł *ParallelGC*, domyślny *kompilator JIT*,
- Java 8, moduł *G1GC*, domyślny *kompilator JIT*,
- Java 8, moduł *SerialGC*, domyślny *kompilator JIT*,
- Java 11, moduł *ParallelGC*, domyślny *kompilator JIT*,
- Java 11, moduł *G1GC*, domyślny *kompilator JIT*,
- Java 11, moduł *SerialGC*, domyślny *kompilator JIT*,
- Java 11, moduł *ZGC*, domyślny *kompilator JIT*,
- Java 11, moduł *ParallelGC*, *kompilator Graal*,
- Java 11, moduł *G1GC*, *kompilator Graal*,
- Java 11, moduł *SerialGC*, *kompilator Graal*.

Niestety, nie udało się uruchomić kombinacji Java 11 z modułem ZGC oraz kompilatorem Graal, ponieważ takie połączenie nie jest obsługiwane przez kompilator Graal. Dlatego moduł ZGC został uruchomiony w Java 11 z domyślnym kompilatorem dla ogólnego zestawienia.

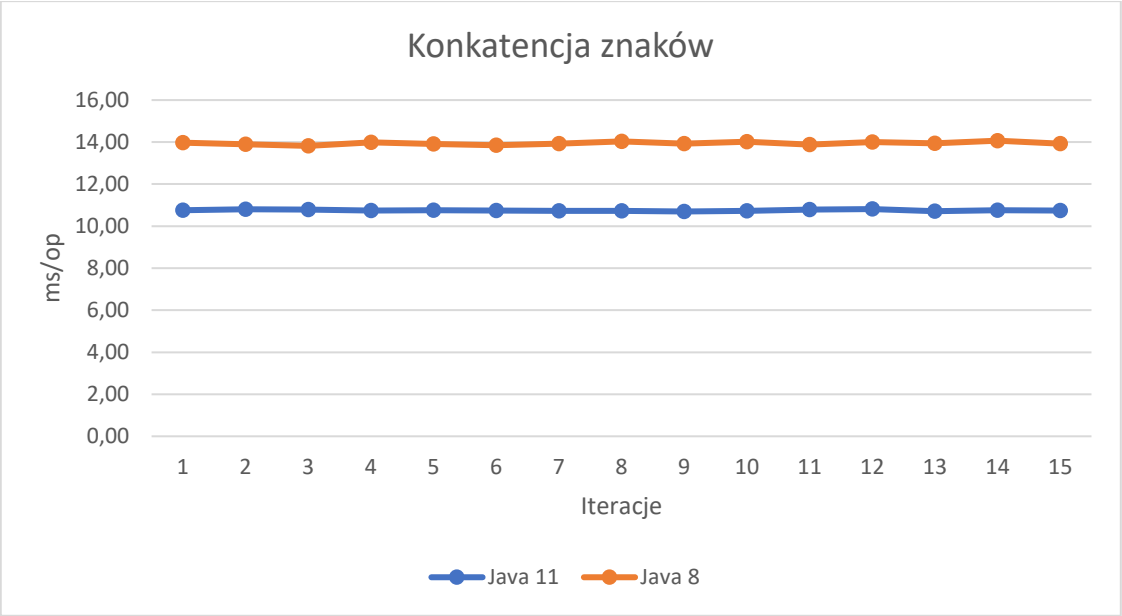
3. ANALIZA I INTERPRETACJA WYNIKÓW BADAŃ

W niniejszym rozdziale została przeprowadzona analiza i interpretacja wyników otrzymanych po wykonanych badaniach z rozdziału 3. Rozdział został podzielony na 3 podrozdziały, które odpowiadają ustalonym pytaniom badawczym i spostrzeżeniom. W pierwszy podrozdziale skoncentrowano się na zmianach w bibliotekach i API. Drugi podrozdział skupia się na modułach do czyszczenia pamięci z nieużywanych obiektów. Na koniec została przeprowadzona interpretacja wyników testów pod względem jakości zoptymalizowanego kodu przez *kompilatory JIT*.

3.1. Wpływ zmian w bibliotekach na czas wykonywania

Wbrew pozorom, zmiany w bibliotekach oraz API zazwyczaj mają duży wpływ na jakoś i wydajność wytwarzanego kodu. Poniżej zostaną przeanalizowane wyniki testów, które odnoszą się bezpośrednio do wprowadzonych zmian i użycia standardowych bibliotek. Wszystkie testy zostały przeprowadzone przy domyślnych konfiguracjach, a więc testy z użyciem Java 8 zostały przeprowadzone z modułem *ParallelGC*, a testy z użyciem Java 11 zostały przeprowadzone z modułem *G1GC*.

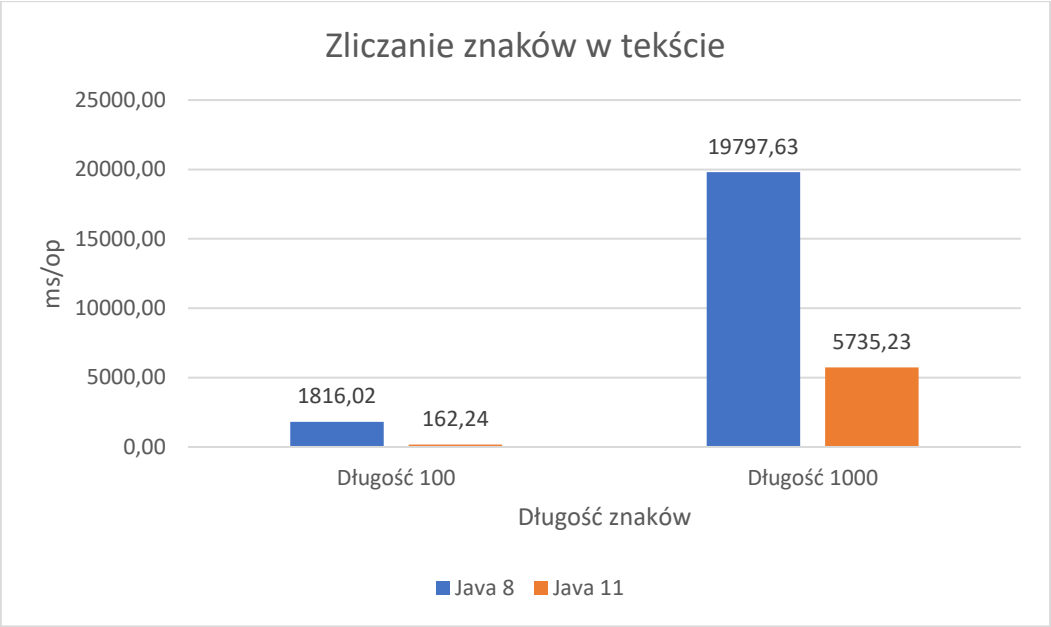
Najbardziej interesującymi testami jest konkatenacja znaków oraz zliczanie znaków w tekście. Dla konkatenacji 100 ciągów znaków o długości 10 wyniki wyszły zaskakujące na korzyść Java 11. Średni czas wykonywania zadania wyniósł 13.35 milisekund w przypadku Java 8, a 6.77 milisekund w przypadku Java 11. To daje nam prawie 97% zysku na wydajności. Błąd pomiarów w obu wyniósł nie więcej niż 8 mikrosekund. Przebieg kolejnych iteracji przedstawia wykres 1.



Wykres 1. Wykres skonstruowany na podstawie zebranych pomiarów dla testu konkatenacji ciągów znaków o długości 10 (Java z użyciem domyślnego modułu GC)

Źródło: opracowanie własne.

Podobne zjawisko można zaobserwować przy zliczaniu znaków w tekście. Dla ciągów znaków o długości 100 wynik dla Javy 8 wyszedł na poziomie 1816 milisekund, a dla Javy 11 to 162 milisekund. W przełożeniu na aplikacje produkcyjne różnica może być bardziej zauważalna. Otrzymane pomiary można łatwo wytłumaczyć. Na wydajność aplikacji pracujących na ciągach znaków ma bezpośredni wpływ sposób ich reprezentowania. Dzięki zmianie rozmiaru znaku z 2 bajtów na 1 bajt w przypadku znaków kodowanych w *Latin-1* oszczędzamy nawet o połowę rozmiaru sterty. Im mniejszy rozmiar struktur danych tym wydajniejszy jest nasz algorytm. Porównanie pomiarów dla testu zliczenia znaków w tekście znajduje się na wykresie 2.



Wykres 2. Wykres skonstruowany na podstawie zebranych pomiarów dla testu zliczania znaków w tekście (Java z użyciem domyślnego modułu GC)

Źródło: opracowanie własne.

W przeprowadzonych badaniach pojawiły się wyniki, które wpływają na niekorzyść nowszej wersji Javy. Tymi wynikami są testy przeprowadzone na strumieniach. Większość z tych testów dotyczy bezpośrednio tematu modułów oczyszczania pamięci z bezużytecznych obiektów. Największa różnica ma miejsce w algorytmie, który mapuje element na parę składającą się z indeksu i danego elementu. Różnica wynosi 45,05 milisekund, które daje około 68% straty do Javy 8. Zaś w przypadku testu sprawdzającego zmianę API dla tworzenia niemodyfikowalnej listy elementów zysk jest znikomy i można stwierdzić, że obie wersje są porównywalne. Jedyną korzyść jest polepszona jakość wytwarzanego kodu. Więcej szczegółowych informacji znajduje się w tabeli 4.

Tabela 4. Zestaw wyników testów opracowanych dla porównania zmian w bibliotekach i API (Java z użyciem domyślnego modułu GC)

		Java 8		Java 11		
benchmarks	**param**	**score [us/op]**	**error**	**score [us/op]**	**error**	**diff**
CUB	100	1816015,266	55051,272	**162243,143**	1466,682	**1019%**
	1000	19797627,749	251102,252	**5735226,739**	268360,853	**245%**
LDB	100	354,685	0,893	**313,146**	0,980	**13%**
	1000	35871,434	122,133	**31384,222**	70,342	**14%**
	10000	**3661797,018**	15342,689	7914118,281	17398,475	**-54%**
SLB	N/A	37701,013	1083,184	**37431,486**	184,100	**1%**
SB.test1	N/A	**22400,008**	117,663	69451,974	565,819	**-68%**
SB.test2	N/A	**13407,053**	46,819	21060,557	125,217	**-36%**

SB.test3	N/A	**31800,454**	1296,810	38531,185	880,804	**-17%**
SCB	10	1335,755	7,191	**677,556**	2,640	**97%**
	100	13943,303	71,964	**10753,530**	34,834	**30%**
	1000	332591,810	200800,060	**118971,990**	1177,786	**180%**

Źródło: opracowanie własne.

3.2. Wpływ garbage collector'a na czas wykonywania

W poniższym podrozdziale zostaną przeanalizowane badania pod kątem wpływu wyboru *garbage collectora* na czas wykonywania programów. W tym celu zaprojektowano i zaimplementowano testy, które manipulują stertą.

Pierwszym zestawieniem jest porównanie zmian w module *G1GC* wprowadzonych od wersji 8 do wersji 11. Badania zostały przeprowadzone z pomocą Javy 8 oraz Javy 11 z konfiguracją, która ustawia moduł *G1GC* jako używany *garbage collector*. W jednym z testów wyszła nieoczekiwana sytuacja. Podczas uruchamiania testu *AllocationBenchmark.fillHeap* z parametrem do tworzenia tablic o rozmiarze 1MB wystąpił wyjątek *java.lang.OutOfMemoryError: Java heap space*, który oznacza, że zabrakło pamięci dla tworzonych obiektów. Szczegóły i listę aktywnych ramek stos z omawianej sytuacji prezentuje rysunek 15. Sytuacja miała miejsce w każdej wersji Java niezależnie od wybranego *kompilatora JIT*. Wspólną konfiguracją był moduł do odśmiecania pamięci z nieużywanych obiektów – *G1 Garbage Collector*. Aby podkreślić powagę sytuacji, to wyjątek powtórzył się nawet przy domyślnych ustawieniach w Java 11. Powodem takiej sytuacji jest sposób działania modułu *G1GC*. Algorytm ma na celu wypełnić 70% pamięci obiektami rozmiaru 1MB. Warto zaznaczyć, że taka sytuacja miała miejsce tylko przy module G1. Opierając się na wiedzy z przeanalizowanej literatury i działania testu możemy przeprowadzić następujące rozumowanie:

- Jeśli rozmiar sterty jest równy 2GB, to 70% z całkowitego rozmiaru to 1433MB.
- Jeśli celem jest zapełnienie 70% sterty obiektami o rozmiarze 1MB, to musiałoby powstać około 1433 obiektów.
- Jeśli całkowity rozmiar pamięci wynosi 2GB, to według przeanalizowanej literatury powinno powstać 2048 regionów o wielkości 1MB.
- Jeśli rozmiar obiektu jest 1MB, to taki obiekt jest większy od połowy rozmiaru regionu, czyli 512B. A więc obiekty tworzone w teście traktowane są jako *Humongous object*.

- Tablica, aby mogła być zaalokowana na stercie potrzebuje dodatkowo 16 bajtów pamięci dla nagłówka.
- Znając fakty, możemy wywnioskować, że rozmiar regionu jest wynikiem połączenia dwóch sąsiednich regionów, czyli obiekt znajduje się w regionie o rozmiarze 2MB.
- Cofając się do przeanalizowanej literatury, wiemy, że ogromny region może pomieścić tylko jeden ogromny obiekt.
- Jeśli algorytm chce zaalokować 1433 obiektów o rozmiarze 1MB + 16B nagłówka, to potrzebne jest co najmniej 1433 ogromnych regionów o wielkości 2MB.

Przeprowadzając takie rozumowanie, możemy wywnioskować, że algorytm próbował zaalokować 1433 obiektów na regionach o wielkości 2MB. Z czego udało mu się zaalokować część, ponieważ G1 dysponował tylko 1024 regionami o wielkości 2MB, stąd brak pamięci. Z jednej strony ułatwia dostęp do takich obiektów dzięki czemu zyskujemy na wydajności, ale z drugiej strony należy mieć na uwadze, że w skrajnych przypadkach może marnować pamięć.

```
# JMH version: 1.29
# VM version: JDK 11.0.11, OpenJDK 64-Bit Server VM, 11.0.11+9
# VM invoker: /home/kamykbb/jdk-11.0.11+9/bin/java
# VM options: -Xms2g -Xmx2g -XX:+AlwaysPreTouch
# Blackhole mode: full + dont-inline hint
# Warmup: 5 iterations, 10 s each
# Measurement: 15 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: com.github.kbreczko.jmhbenchmarks.benchmarks.allocation.AllocationBenchmark.fillHeap
# Parameters: (size = 1048576)

# Run progress: 6.52% complete, ETA 07:53:29
# Fork: 1 of 1
# Warmup Iteration   1: <failure>

java.lang.OutOfMemoryError: Java heap space
    at com.github.kbreczko.jmhbenchmarks.benchmarks.allocation.AllocationBenchmark.fillHeap(AllocationBenchmark.java:52)
    at com.github.kbreczko.jmhbenchmarks.benchmarks.allocation.jmh_generated.AllocationBenchmark_fillHeap_jmhTest.fillHeap_thrpt_jmhStub(AllocationBenchmark_fillHeap_jmhTest.java:161)
    at com.github.kbreczko.jmhbenchmarks.benchmarks.allocation.jmh_generated.AllocationBenchmark_fillHeap_jmhTest.fillHeap_Throughput(AllocationBenchmark_fillHeap_jmhTest.java:104)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:566)
    at org.openjdk.jmh.runner.BenchmarkHandler$BenchmarkTask.call(BenchmarkHandler.java:470)
    at org.openjdk.jmh.runner.BenchmarkHandler$BenchmarkTask.call(BenchmarkHandler.java:453)
    at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
    at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:515)
    at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
    at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
    at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
    at java.base/java.lang.Thread.run(Thread.java:829)
```

Rys. 15. Szczegóły uruchomienia i ślad stosu podczas wypełniania sterty obiektami o rozmiarze 1MB

Źródło: opracowanie własne.

Pomijając powyższy przypadek to można zaobserwować wyższą wydajność w Javie 11. Średni zysk w Javie 11 używając *G1GC* utrzymuje się w okolicy 23% w porównaniu z Java 8. Niewielkie straty można zauważyć podczas sortowania tablicy, czyli zamianę miejscami

elementów w tablicy oraz przy alokowaniu obiektów o rozmiarze 10KB i 10MB w liście. Porównanie wyników testów znajduje się w tabeli 5.

Tabela 5. Zestaw wyników testów opracowanych dla porównania modułów GC (Java z użyciem modułu G1GC)

		Java 8		Java 11		
benchmarks	**param**	**score [us/op]**	**error**	**score [us/op]**	**error**	**diff**
ALLB.test1	512	768801.114	6753.390	**729519.388**	2322.582	**5%**
	1024	1063990.941	8829.632	**968745.884**	9316.594	**10%**
	2048	1098395.234	8153.260	**1025497.372**	5174.879	**7%**
	10240	1029311.599	7510.325	**976876.427**	2810.247	**5%**
	102400	945843.998	6374.984	**899330.674**	2456.317	**5%**
	1048576	1356417.548	4514.849	**1048851.624**	17398.759	**29%**
	2097152	1257837.349	7399.449	**989866.264**	25482.391	**27%**
	10485760	1027894.445	87909.846	**952409.169**	14115.680	**8%**
	104857600	949499.278	5060.059	**909170.123**	13834.388	**4%**
ALLB.test2	512	5204410.358	44244.206	**1832942.496**	9606.389	**184%**
	1024	1953862.855	47492.923	**1963001.936**	26390.082	**0%**
	2048	1926509.536	30987.374	**1851758.993**	17390.380	**4%**
	10240	1688806.444	24768.280	2089813.272	12237.966	**-19%**
	102400	1663598.738	18311.342	**1369634.534**	6582.855	**21%**
	1048576	N/A	N/A	N/A	N/A	N/A
	2097152	N/A	N/A	N/A	N/A	N/A
	10485760	**1085001.888**	61048.576	1239919.113	21222.557	**-12%**
	104857600	1049201.951	7236.447	**960800.141**	21198.750	**9%**
AB.test1	N/A	383.237	0.468	**375.507**	0.203	**2%**
AB.test2	N/A	169.608	0.775	**165.450**	0.662	**3%**
AB.test3	N/A	**201.765**	0.163	311.168	0.165	**-35%**
AB.test4	N/A	184.483	0.800	**171.774**	0.735	**7%**
TB	N/A	13179822.616	1883128.594	**4005711.876**	20565.236	**229%**

Źródło: opracowanie własne.

Kolejnym zestawieniem jest porównanie obu wersji z użyciem modułu *ParallelGC*, który jest domyślnym modułem w Javie 8. W porównaniu z poprzednim zestawieniem, otrzymujemy wyniki bardzo zbliżone do siebie, ale dalej na korzyść Javy 11. Istnieją jednak testy z większą rozbieżnością. Takim przykładem jest dodawanie małych obiektów do listy. Średni zysk przy użyciu Javy 11 wynosi około 48%. W przypadku obiektów większych od 2KB różnica znacznie się zmniejsza, a w niektórych przypadkach na korzyść dla starszej wersji. Porównanie wyników testów znajduje się w tabeli 6.

Tabela 6. Zestaw wyników testów opracowanych dla porównania modułów GC (Java z użyciem modułu ParallelGC)

		Java 8		Java 11		
benchmarks	**param**	**score [us/op]**	**error**	**score [us/op]**	**error**	**diff**
ALLB.test1	512	723860.215	3531.505	**687903.387**	3108.977	**5%**
	1024	1016460.531	15596.684	**971998.303**	8159.932	**5%**
	2048	1068448.572	3789.821	**1026112.555**	3722.027	**4%**
	10240	1005466.892	4356.055	**973184.114**	4465.307	**3%**
	102400	923532.485	2804.778	**892353.493**	3878.406	**3%**
	1048576	917001.393	3886.247	**883924.956**	3613.257	**4%**
	2097152	915812.427	3892.928	**885422.027**	3235.928	**3%**
	10485760	**827106.210**	117839.163	880834.600	3310.275	**-6%**
	104857600	901537.212	4500.190	**867253.535**	3102.297	**4%**
ALLB.test2	512	7176379.875	420817.406	**4978806.827**	127250.963	**44%**
	1024	5043676.256	60253.280	**3425713.766**	36358.272	**47%**
	2048	3813754.370	91885.664	**2492051.269**	49281.704	**53%**
	10240	**1841447.959**	19612.008	1897976.947	23269.032	**-3%**
	102400	**1516120.086**	11019.827	1801573.154	23446.924	**-16%**
	1048576	**1588598.292**	19430.838	1666092.988	17580.319	**-5%**
	2097152	1699437.915	69374.195	**1643198.910**	44408.505	**3%**
	10485760	1646111.548	101313.658	**1630555.466**	25000.212	**1%**
	104857600	1790163.751	11891.987	**1725891.271**	10975.249	**4%**
AB.test1	N/A	384.691	0.955	**375.293**	0.191	**3%**
AB.test2	N/A	170.168	0.784	**165.437**	0.465	**3%**
AB.test3	N/A	**201.778**	0.101	271.077	2.434	**-26%**
AB.test4	N/A	183.112	0.983	**179.109**	1.078	**2%**
TB	N/A	4035611,727	18320,149	**3812228,103**	22850,552	**6%**

Źródło: opracowanie własne.

Następnym zestawieniem jest porównanie obu wersji z użyciem modułu *SerialGC*. Wyniki wyszły jak najbardziej pożądane z racji tego, że nie wprowadzono znacznych zmian w module *SerialGC* od wersji 8. Przy obliczaniu zysków nieznacznie przeważa Java 11 z zyskiem na poziomie od 1% do 8%. Największa różnica tym samym największa strata występuje w teście sortowania tablicy. Strata do Javy 8 wynosi aż 27%. Porównanie wyników testów znajduje się w tabeli 7.

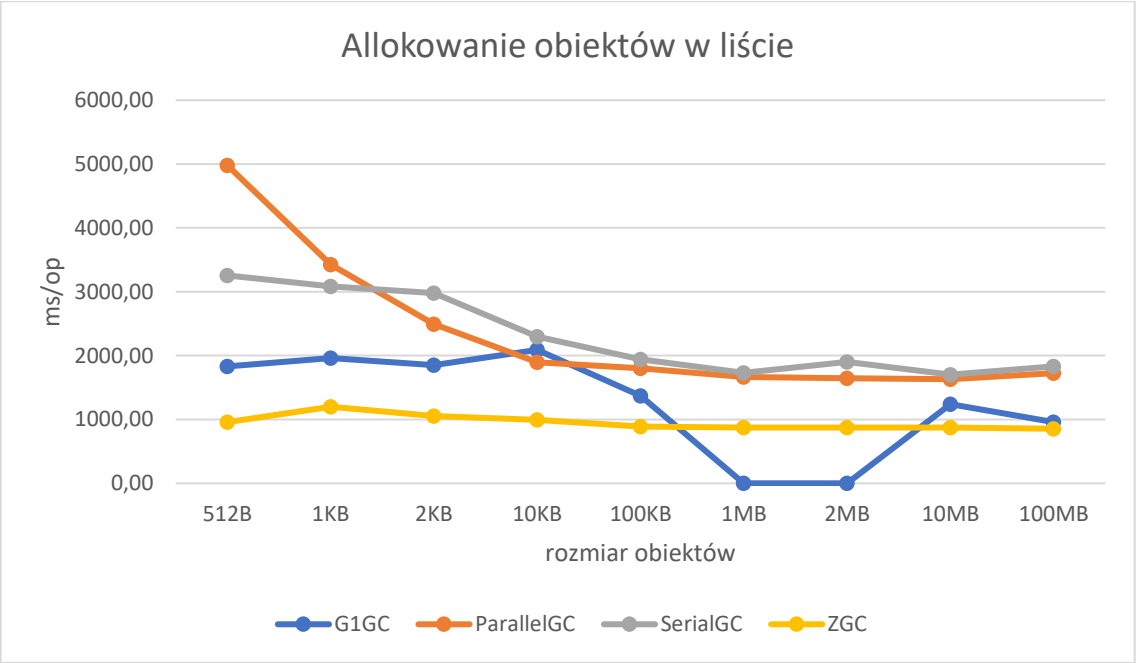
Tabela 7. Zestaw wyników testów opracowanych dla porównania modułów GC (Java z użyciem modułu SerialGC)

		Java 8		Java 11		
benchmarks	**param**	**score [us/op]**	**error**	**score [us/op]**	**error**	**diff**
ALLB.test1	512	669364.176	1179.137	**636461.620**	1012.419	**5%**
	1024	948885.014	3941.520	**904902.979**	7826.835	**5%**
	2048	1006358.865	3298.877	**968487.370**	2248.553	**4%**
	10240	943830.153	1190.705	**912656.834**	1786.950	**3%**
	102400	865058.030	1357.389	**837558.821**	2168.215	**3%**
	1048576	856987.014	1411.191	**829922.068**	2273.657	**3%**
	2097152	858512.529	984.316	**828650.705**	2317.330	**4%**
	10485760	**772895.094**	112181.569	821580.501	2527.239	**-6%**
	104857600	804639.413	2495.580	**775142.956**	1797.384	**4%**
ALLB.test2	512	3499479.966	98818.402	**3255188.618**	6129.965	**8%**
	1024	3319599.802	30377.150	**3086646.160**	40509.503	**8%**
	2048	3158528.078	40350.988	**2978666.748**	44566.266	**6%**
	10240	2420440.665	10568.615	**2298917.656**	6859.976	**5%**
	102400	2013921.185	15678.992	**1939580.823**	4726.348	**4%**
	1048576	1789918.110	11793.443	**1732796.798**	12152.021	**3%**
	2097152	1915018.556	65900.057	**1899292.816**	8427.572	**1%**
	10485760	1732786.645	98794.879	**1702470.368**	7919.557	**2%**
	104857600	1880105.964	14065.338	**1828839.997**	11777.602	**3%**
AB.test1	N/A	384.385	2.053	**375.286**	0.166	**2%**
AB.test2	N/A	169.651	0.536	**165.211**	0.587	**3%**
AB.test3	N/A	**202.943**	0.157	276.632	1.377	**-27%**
AB.test4	N/A	180.747	4.063	**179.274**	1.395	**1%**
TB	N/A	9650683.565	55739.834	**8989756.670**	34594.679	**7%**

Źródło: opracowanie własne.

Ostatnim i najciekawszym zestawieniem jest porównanie wszystkich modułów GC w Javie. Porównanie wyników testów znajduje się w tabeli 8. Z racji tego, że Java 8 nie obsługuje modułu *ZGC* oraz nie posiada najnowszych zmian w *G1GC*, to analiza i interpretacja wyników została przeprowadzona na podstawie badań w Javie 11. Wykres 3 przedstawia otrzymane pomiary podczas testu alokowania obiektów na liście. Z wykresu można odczytać, że najbardziej pożądanym modułem do tego celu okazuje się moduł *ZGC*, który utrzymuje średni czas na wysokości 900 milisekund. Moduł *ZGC* aktualnie jest jeszcze w fazie eksperymentalnej. Kolejnym moduł, który bardzo dobrze sobie poradził przy postawionym zadaniu jest *G1GC*. Domyślny moduł z Javy 11 osiągnął wynik na poziomie 1617 milisekund, pomijając testy, które nie mógł ukończyć. Na wykresie można zauważyć, że większość modułów miała problem z alokacją małych obiektów. W tym przypadku im obiekty większe tym wydajność ulega poprawieniu. Wynikać to może z właściwości, że moduły bazują na generacjach. Wtedy mniejszych obiektów jest więcej na stercie i wymaga więcej pracy dla

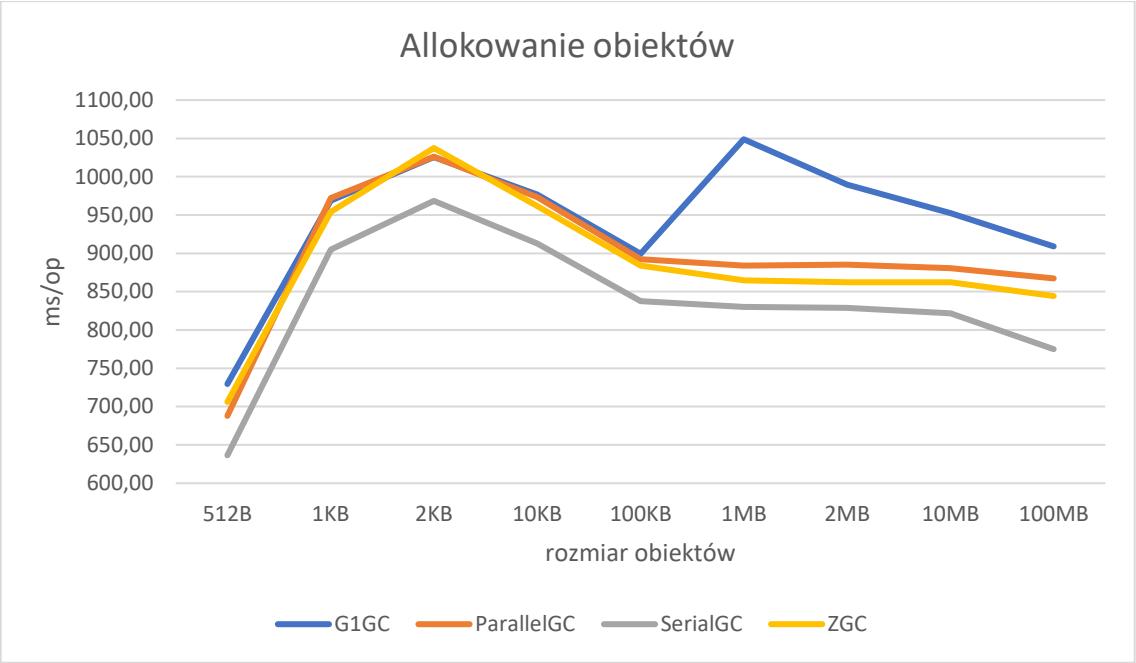
modułów oczyszczania pamięci. Warto zaznaczyć, że moduł *ParallelGC* korzysta z strategii *Mark-Copy* w młodej generacji. A to oznacza, że kopiuje używane obiekty do nowego miejsca na stercie co prawdopodobnie mocno odbija się na wydajności w omawianym przypadku. Porównanie pomiarów dla testu alokowania obiektów w liście znajduje się na wykresie 3.



Wykres 3. Wykres skonstruowany na podstawie zebranych pomiarów dla testu alokowania dużych obiektów w liście

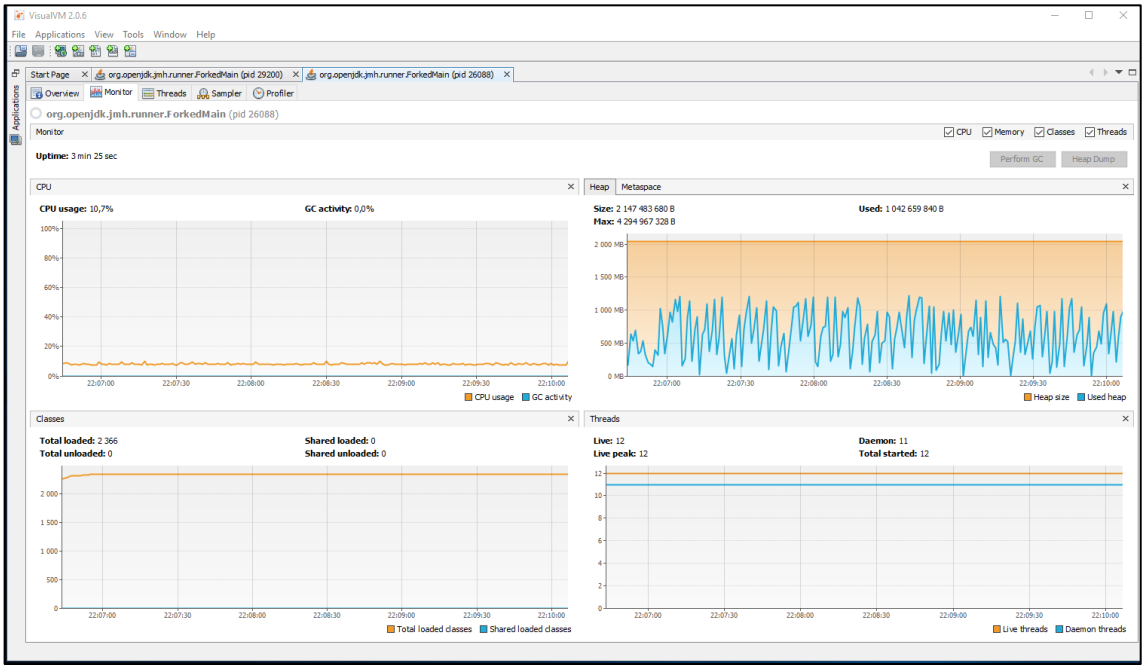
Źródło: opracowanie własne.

Inaczej wygląda sytuacja w przypadku alokowania krótkotrwałych obiektów na stercie. Najbardziej wyróżniającym modułem jest tutaj moduł *SerialGC*. Niezależnie od rozmiaru obiektów wynik jest lepszy o około 50 milisekund od pozostałych modułów. W przypadku innych modułów to wyniki są zbliżone. Najbardziej widać to przy obiektach o mniejszych rozmiarach. Rysunek 16 przedstawia obciążenie zasobów serwera podczas wypełniania sterty małymi obiektami. Dopiero po osiągnięciu rozmiaru 1MB można zauważyć znaczne pogorszenie wydajności w przypadku modułu *G1GC*. Prawdopodobnie wynika to z tego samego powodu, dlaczego nie mogły się zakończyć zadania w poprzednim teście. Porównanie pomiarów dla testu alokowania obiektów znajduje się na wykresie 4.



Wykres 4. Wykres skonstruowany na podstawie zebranych pomiarów dla testu alokowania obiektów

Źródło: opracowanie własne.



Rys. 16. Zrzut ekranu aplikacji VisualVM podczas monitorowania działania testu alokowania obiektów

Źródło: opracowanie własne.

Poza badaniem alokowania nowych obiektów na stercie, poddano analizie czas zapisu i odczytu elementów z tablicy. Wyniki okazują się być dość zbliżone do siebie. W przypadku

odczytu różnica wynosi do 1 mikrosekundy, aczkolwiek najlepiej prezentuje się moduł *SerialGC*, a po nim *ParallelGC*. W badaniach sprawdzających wydajność zapisu elementów do tablicy sytuacja wygląda odwrotnie, a różnica sięga do 8 mikrosekund. Najlepiej prezentuje się *ZGC*, a po nim *G1GC* ze stratą 0.2 mikrosekundy. Warto dodać, że w porównaniu do poprzednich testów to złożoność algorytmu jest mniejsza, co może spowodować dużą różnicę w środowisku produkcyjnym, gdzie operuje się na złożonych algorytmach. Ostatnim testem, który warto poruszyć jest alokowanie złożonych obiektów, a dokładnie mówiąc zrównoważone binarne drzewa o wysokości równej 9. Obiekty są o tyle trudne dla modułów GC, że posiadają zależności, a więc moduły muszą przejść po wszystkich obiektach i zaznaczyć, że dany obiekt jest używany. Najlepiej poradził sobie z tym problemem moduł *ZGC* wykonując test w 2,95 sekundy. Na drugim miejscu znajduje się moduł *ParallelGC* osiągając czas 3.81 sekundy, a po nim z stratą 0.19 sekundy moduł G1GC.

Tabela 8. Zestaw wyników testów opracowanych dla porównania modułów GC (Java z uwzględnieniem wszystkich modułów GC).

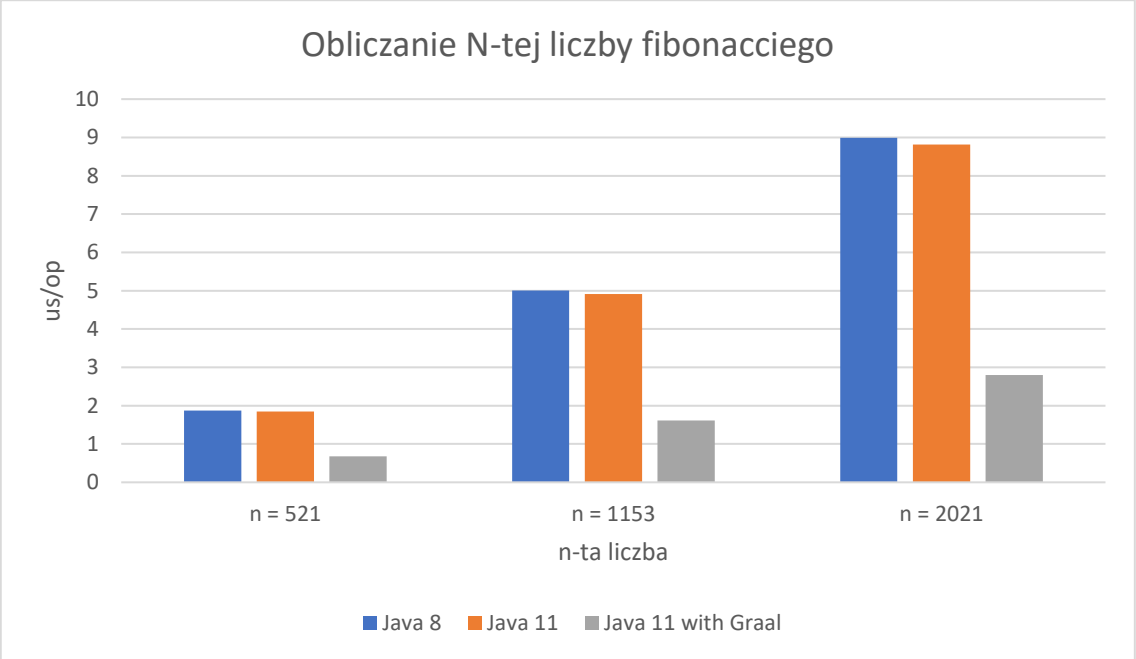
		G1GC	ParallelGC	SerialGC	ZGC
benchmarks	**param**	**score [us/op]**	**score [us/op]**	**score [us/op]**	**score [us/op]**
ALLB.test1	512	**729519.388**	687903.387	**636461.620**	706092.688
	1024	968745.884	**971998.303**	**904902.979**	954218.086
	2048	1025497.372	**1026112.555**	**968487.370**	1037460.655
	10240	**976876.427**	973184.114	**912656.834**	961784.892
	102400	**899330.674**	892353.493	**837558.821**	883947.461
	1048576	**1048851.624**	883924.956	**829922.068**	864949.776
	2097152	**989866.264**	885422.027	**828650.705**	862211.560
	10485760	**952409.169**	880834.600	**821580.501**	862265.339
	104857600	**909170.123**	867253.535	**775142.956**	844130.107
ALLB.test2	512	1832942.496	**4978806.827**	3255188.618	**959264.655**
	1024	1963001.936	**3425713.766**	3086646.160	**1199127.661**
	2048	1851758.993	2492051.269	**2978666.748**	**1052010.076**
	10240	2089813.272	1897976.947	**2298917.656**	**994727.209**
	102400	1369634.534	1801573.154	**1939580.823**	**888454.969**
	1048576	N/A	1666092.988	**1732796.798**	**875132.108**
	2097152	N/A	1643198.910	**1899292.816**	**871902.800**
	10485760	1239919.113	1630555.466	**1702470.368**	**875587.203**
	104857600	960800.141	1725891.271	**1828839.997**	**854917.087**
AB.test1	N/A	375.507	375.293	**375.286**	**375.563**
AB.test2	N/A	165.450	165.437	**165.211**	**165.684**
AB.test3	N/A	**311.168**	**271.077**	276.632	301.215
AB.test4	N/A	171.774	179.109	**179.274**	**171.596**
TB	N/A	4005711.876	3812228,103	**8989756.670**	**2951321.179**

Źródło: opracowanie własne.

3.3. Wpływ kompilatora Just-in-Time na czas wykonywania

W poniższym podrozdziale zostaną przeanalizowane badania pod kątem optymalizacji wykonywanego kodu przez *kompilator JIT*. Badania zostały przeprowadzone z domyślnym kompilatorem w Javy 8 oraz Java 11 i dodatkowo z kompilatorem Graal w Java 11.

Pierwsze zestawienie będzie bazowało na teście obliczania n-tej liczby Fibonacciego. Tak jak można byłoby się spodziewać różnica między Java 11 a Java 8 jest znikoma i wynosi maksymalnie 0.1 sekundy na korzyść Javy 11. Inaczej wygląda porównanie z Java 11 przy użyciu Graal. Niezależnie od przekazywanego parametru, to czas wykonywania przy użyciu Graal jest 4 razy rzędu szybszy niż przy użyciu domyślnego kompilatora. Powodem takiego wyniku jest zdecydowanie optymalizacja na poziomie kompilatora. Komilator Graal prawdopodobnie wykrył, że ma do czynienia z rekurencją ogonową, którą może zamienić na pętle iteracyjną. Po zmianie na pętlę iteracyjną daje to kolejne kroki do optymalizacji. Wyniki z danego testu sporządzone na wykresie 5.

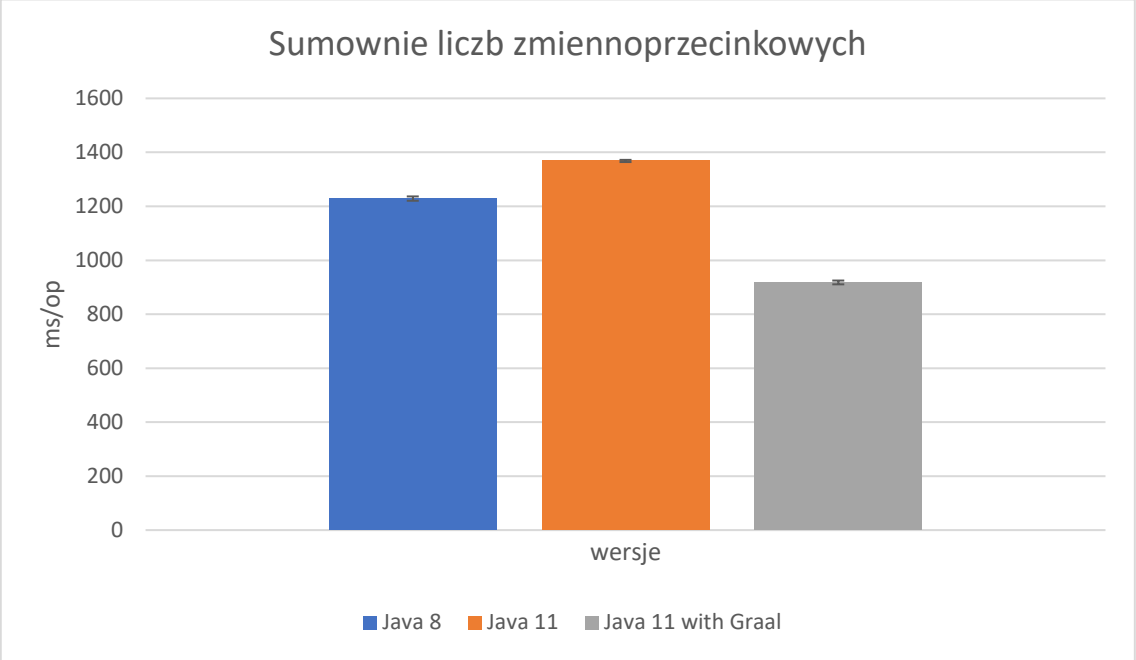


Wykres 5. Wykres skonstruowany na podstawie zebranych pomiarów dla testu obliczania n-tej liczby Fibonacciego (Java z użyciem modułu ParallelGC)

Źródło: opracowanie własne.

Kolejne zestawienie bazuje na metodzie sumowania dużych liczb zmiennoprzecinkowych. Podczas sumowania, algorytm wykonuje inne poboczne operacje,

które zawierają się w *JitCompilerBenchmarks*. Podobnie jak w poprzednim teście, kombinacja Java 11 wraz z kompilatorem Graal przeważyła nad Java 8 i Java 11 z domyślnym kompilatorem. Algorytm przy użyciu kompilatora Graal wykonał się w 916 milisekund. Ten sam kod uruchomiony w Java 8 osiągnął 1228 milisekund, a najgorzej wypadł w Java 11 osiągając 1368 milisekund. Wyniki z danego testu sporządzone na wykresie 6.



Wykres 6. Wykres skonstruowany na podstawie zebranych pomiarów dla testu sumowania liczb zmiennoprzecinkowych (Java z użyciem modułu ParallelGC)

Źródło: opracowanie własne.

Rozbijając test sumowania liczb zmiennoprzecinkowych na części składowe możemy dowiedzieć się więcej informacje na temat optymalizacji *kompilatora Gaal*. W wynikach dla *JitCompilerBenchmarks* nie zawsze *kompilator Graal* wykonywał kod w najkrótszym czasie. W porównaniu z Java 8 i Java 11 uruchomioną z domyślnym kompilatorem, Java 11 z *kompilatorem Graal* wygrała 2 testy z 6 możliwych. Tymi testami są: wymuszenie wyciągnięcia wyrażeń poza pętlę oraz zredukowanie synchronicznych bloków. W przypadku wyciągania wyrażeń poza pętle w zestawieniu z Java 11 zysk wynosi na poziomie 95%, a przy Java 8 zysk na poziomie 49%. Z kolei w przypadku redukcji synchronicznych bloków w zestawieniu z Java 11 to aż 2277%, a przy Java 8 to 2227%. Łatwo można zauważyć, że *kompilator Graal* łatwo radzi sobie z podanymi operacjami. W przypadku pozostałych metod w porównaniu z domyślnym *kompilatorem JIT* osiągnął straty. Największa strata wynosi 37%

podczas usuwania warunków, które zawsze są prawdziwe. Zestawienia kompilatora Graal z Java 11 znajdują się w tabeli 9, a z Java 8 w tabeli 10.

Tabela 9. Zestaw wyników testów opracowanych dla porównania kompilatorów JIT (Java 11 z modułem G1GC)

		Java 11		Java 11 with Graal		
benchmarks	**param**	**score [us/op]**	**error**	**score [us/op]**	**error**	**diff**
FIB	521	1,846	0,001	**0,678**	0,001	**172%**
	1153	4,917	0,007	**1,612**	0,001	**205%**
	2021	8,989	0,018	**2,799**	0,002	**221%**
JITB.test1	N/A	**1882,352**	0,966	2999,608	1,854	**-37%**
JITB.test2	N/A	62092,631	23,396	**31843,481**	197,772	**95%**
JITB.test3	N/A	**1704018,982**	1011,732	1761135,716	698,594	**-3%**
JITB.test4	N/A	0,523	0,001	**0,022**	0,001	**2277%**
JITB.test5	N/A	**7416,136**	4,668	10404,857	4,901	**-29%**
JITB.test6	N/A	**35276,061**	2891,565	40007,316	4540,068	**-12%**
CALCB	N/A	1183786,851	3368,408	**921872,181**	7753,329	**28%**

Źródło: opracowanie własne.

Tabela 10. Zestaw wyników testów opracowanych dla porównania kompilatorów JIT (Java 8 z modułem ParallelGC)

		Java 8		Java 11 with Graal		
benchmarks	**param**	**score [us/op]**	**error**	**score [us/op]**	**error**	**diff**
FIB	521	1,871	0,001	**0,678**	0,001	**176%**
	1153	5,007	0,008	**1,611**	0,001	**211%**
	2021	8,989	0,014	**2,797**	0,001	**221%**
JITB.test1	N/A	**1900,744**	1,694	2997,618	0,719	**-37%**
JITB.test2	N/A	46992,929	31,566	**31499,241**	4,916	**49%**
JITB.test3	N/A	**1715009,03**	1144,341	1760113,4	559,038	**-3%**
JITB.test4	N/A	0,512	0,003	**0,022**	0,001	**2227%**
JITB.test5	N/A	**8365,769**	6,059	10394,202	4,463	**-20%**
JITB.test6	N/A	**36804,444**	2681,592	37800,099	4388,522	**-3%**
CALCB	N/A	1228669,09	8311,967	**917919,062**	7822,334	**34%**

Źródło: opracowanie własne.

ZAKOŃCZENIE

W każdym kolejnym wydaniu Javy możemy zauważyć sporo nowych zmian. Nowe stabilne wersje otrzymujemy co 3 lata. Od czasu Java 8 zmieniła się polityka licencjonowania, aktualizacji i wsparcia. Oprócz tego wprowadzono zmiany dla środowiska uruchomieniowego, bibliotek i API. Większość popularnych bibliotek używanych w środowisku produkcyjnym zaktualizowała już wersję Javy do najnowszych, dzięki czemu nasza aplikacja będzie w stanie korzystać z najnowszych bibliotek.

Celem niniejszej pracy było zbadanie wydajności wirtualnej maszyny Javy oraz porównanie jej najnowszych wersji. Napisanie dobrych testów do analizy porównawczej nie jest rzeczą łatwą. Istnieje wiele optymalizacji na poziomie samej Javy, ale też na poziomie procesora i innych zasobów na serwerze. Też ciężko jest objąć wszystkie przypadki użycia w kilku testach. W porównaniach zestawione zostały ze sobą pomiary czasów wykonywania implementacji różnych algorytmów. Na podstawie przeanalizowanej literatury i popartych przykładami można zaobserwować większą konfigurowalność wirtualnej maszyny Java i poprawę wydajności aplikacji. Przykładem na pewno posłuży możliwość wymiany *kompilatora JIT* pozostając przy domyślnej wirtualnej maszynie Javy oraz wybranie odpowiedniego modułu do oczyszczania sterty z obiektów bezużytecznych. Warto dodać, że każdy z tych modułów można w dowolny sposób dostosowywać do aplikacji. Zmieniając rozmiar regionów w przypadku G1GC lub ustawianie liczby wątków używanych przez *garbage collector*. Oprócz *kompilatora JIT* i modułów do czyszczenia sterty, można zauważyć znaczną poprawę wydajności od strony aktualizacji bibliotek oraz sposobu reprezentowania niektórych struktur danych. Sporządzone testy pokazały znaczny wzrost wydajności podczas działania na ciągach znaków. To dobry moment na migrację aplikacji z Java 8 do Java 11, uwzględniając fakt, że nowe wydania w większości są kompatybilne wstecz. Aktualizacja do najnowszych wersji zapewni utrzymanie bezpieczeństwa, wydajności aplikacji oraz jakości utrzymywanego kodu.

Na potrzeby niniejszej pracy został utworzony specjalny projekt z kodem źródłowym omawianych algorytmów. Zaprojektowane i zaimplementowane testy porównawcze można dowolnie rozszerzać oraz modyfikować. Praca pokazuje jedynie część przypadków jakie zachodzą w środowisku produkcyjnym. Istnieje możliwość osiągnięcia lepszych rezultatów powstałych testów poprzez kolejne konfiguracje maszyny wirtualnej, modułów oczyszczania pamięci lub kompilatorów. Przeprowadzone badania można rozszerzyć o budowanie

niestandardowych okrojonych pakietów JRE, kompilację AOT lub całkowitą wymianę wirtualnej maszyny na inną. Ponadto testy można rozszerzyć o porównanie powszechnie używanych bibliotek do wytwarzania aplikacji internetowych działających po stronie serwera. W celu rozszerzenia badań zaplanowano porównanie obu wersji z użyciem technologii webowych: *spring boot*, *microprofile*, *akka* oraz *vert.x*. Używając wymienionych bibliotek można wykonać testy wydajnościowe typu *black box*, symulując duży ruch na serwerze i mierząc czas odpowiedzi na żądania http.

BIBLIOGRAFIA

1. Aboullaite, M. (2017, 8 31). *Understanding JIT compiler (just-in-time compiler).* Pobrano z lokalizacji https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/
2. AdoptOpenJDK. (brak daty). *Support.* Pobrano z lokalizacji https://adoptopenjdk.net/support.html
3. Altvater, A. (2017, 5 11). *What is Java Garbage Collection? How It Works, Best Practices, Tutorials, and More.* Pobrano z lokalizacji https://stackify.com/what-is-java-garbage-collection/
4. Baeldung. (2020, 8 6). *Deep Dive Into the New Java JIT Compiler – Graal.* Pobrano z lokalizacji https://www.baeldung.com/graal-java-jit-compiler
5. Baeldung. (2021, 1 9). *How to Calculate Levenshtein Distance in Java?* Pobrano z lokalizacji https://www.baeldung.com/java-levenshtein-distance
6. Balci, M. (brak daty). *JEP 285: Spin-Wait Hints in Java*. Pobrano z lokalizacji https://metebalci.com/blog/spin-wait-hints-in-java/
7. Beckwith, M. (2013, 8). *Garbage First Garbage Collector Tuning.* Pobrano z lokalizacji https://www.oracle.com/technical-resources/articles/java/g1gc.html
8. Haiut, A. (2014, 7 21). *White Box Vs. Black Box in Load Testing*. Pobrano z lokalizacji https://www.blazemeter.com/blog/white-box-vs-black-box-load-testing
9. Ishizaki, K., Hayashi, A., Koblents, G. i Sarkar, V. (2015). *Compiling and Optimizing Java 8 Programs for GPU Execution.* San Francisco: IEEE.
10. Kubryński, J. (2015, 03). Co każdy programista Java powinien wiedzieć o JVM. *Programista*, strony 24-27.
11. Kubryński, J. (2015, 04). Co każdy programista Java powinien wiedzieć o JVM: zarządzanie pamięcią. *Programista*, strony 20-23.
12. Larsson, R. (2020). *Evaluation of GraalVM Performance for Java Programs.* Department of computer science and media technology (CM). Pobrano z lokalizacji https://www.diva-portal.org/smash/get/diva2:1457592/FULLTEXT01.pdf
13. Lavieri, E. i Verhas, P. (2017). Segmented code cache [JEP 197]. W *Mastering Java 9.* Packt. Pobrano z lokalizacji https://subscription.packtpub.com/book/application_development/9781786468734/2/ch02lvl1sec19/segmented-code-cache-jep-197
14. Marczak, M. (2018, 8 8). *Algorytmy GC. Serial, Parallel, CMS.* Pobrano z lokalizacji https://bulldogjob.pl/news/424-algorytmy-gc-serial-parallel-cms
15. Marczak, M. (2019, 12 6). *JVM Garbage Collector. Wprowadzenie.* Pobrano z lokalizacji https://bulldogjob.pl/news/404-jvm-garbage-collector-wprowadzenie
16. Microsoft Corporation. (2019, 11 19). *Reasons to move to Java 11.* Pobrano z lokalizacji https://docs.microsoft.com/en-us/azure/developer/java/fundamentals/reasons-to-move-to-java-11
17. Mishra, P. (2020, 9 15). *Top 10 Java Unit Testing Frameworks for 2021*. Pobrano z lokalizacji https://www.lambdatest.com/blog/top-10-java-testing-frameworks/

18. Nayak, S. (2021, 1 22). *Garbage Collection in Java – What is GC and How it Works in the JVM*. Pobrano z lokalizacji https://www.freecodecamp.org/news/garbage-collection-in-java-what-is-gc-and-how-it-works-in-the-jvm/
19. Oracle Corporation. (2017, 8 17). *JEP 193: Variable Handles*. Pobrano z lokalizacji https://openjdk.java.net/jeps/193
20. Oracle Corporation. (2017, 4 28). *JEP 197: Segmented Code Cache.* Pobrano z lokalizacji https://openjdk.java.net/jeps/197
21. Oracle Corporation. (2017, 9 22). *JEP 220: Modular Run-Time Images.* Pobrano z lokalizacji https://openjdk.java.net/jeps/220
22. Oracle Corporation. (2017, 9 12). *JEP 248: Make G1 the Default Garbage Collector*. Pobrano z lokalizacji https://openjdk.java.net/jeps/248
23. Oracle Corporation. (2017, 4 24). *JEP 266: More Concurrency Updates*. Pobrano z lokalizacji https://openjdk.java.net/jeps/266
24. Oracle Corporation. (2017, 6 26). *JEP 269: Convenience Factory Methods for Collections*. Pobrano z lokalizacji https://openjdk.java.net/jeps/269
25. Oracle Corporation. (2017, 8 23). *JEP 285: Spin-Wait Hints*. Pobrano z lokalizacji https://openjdk.java.net/jeps/285
26. Oracle Corporation. (2018, 10 5). *JEP 295: Ahead-of-Time Compilation.* Pobrano z lokalizacji https://openjdk.java.net/jeps/295
27. Oracle Corporation. (2018, 3 29). *JEP 307: Parallel Full GC for G1*. Pobrano z lokalizacji https://openjdk.java.net/jeps/307
28. Oracle Corporation. (2018, 8 17). *JEP 310: Application Class-Data Sharing*. Pobrano z lokalizacji https://openjdk.java.net/jeps/310
29. Oracle Corporation. (2018, 3 28). *JEP 317: Experimental Java-Based JIT Compiler*. Pobrano z lokalizacji https://openjdk.java.net/jeps/317
30. Oracle Corporation. (2018, 9 24). *JEP 318: Epsilon: A No-Op Garbage Collector (Experimental)*. Pobrano z lokalizacji https://openjdk.java.net/jeps/318
31. Oracle Corporation. (2018, 9 5). *JEP 331: Low-Overhead Heap Profiling.* Pobrano z lokalizacji http://openjdk.java.net/jeps/331
32. Oracle Corporation. (2018, 9 17). *JEP 332: Transport Layer Security (TLS) 1.3*. Pobrano z lokalizacji https://openjdk.java.net/jeps/332
33. Oracle Corporation. (2019, 9 16). *JEP 243: Java-Level JVM Compiler Interface*. Pobrano z lokalizacji https://openjdk.java.net/jeps/243
34. Oracle Corporation. (2019, 8 21). *JEP 312: Thread-Local Handshakes*. Pobrano z lokalizacji https://openjdk.java.net/jeps/312
35. Oracle Corporation. (2020, 3 5). *JEP 254: Compact Strings*. Pobrano z lokalizacji https://openjdk.java.net/jeps/254
36. Oracle Corporation. (2020, 4 6). *JEP 291: Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector.* Pobrano z lokalizacji https://openjdk.java.net/jeps/291
37. Oracle Corporation. (2020, 9 15). *JEP 321: HTTP Client*. Pobrano z lokalizacji https://openjdk.java.net/jeps/321
38. Oracle Corporation. (2020, 3 13). *JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)*. Pobrano z lokalizacji https://openjdk.java.net/jeps/333

39. Oracle Corporation. (brak daty). *Class Thread*. Pobrano z lokalizacji https://docs.oracle.com/javase/10/docs/api/java/lang/Thread.html
40. Oracle Corporation. (brak daty). *Code Tools: jmh.* Pobrano z lokalizacji https://openjdk.java.net/projects/code-tools/jmh/
41. Oracle Corporation. (brak daty). *VisualVM.* Pobrano z lokalizacji https://visualvm.github.io/
42. Rob. (2018, 4 10). *Java 10 improvements to Garbage Collection explained in 5 minutes.* Pobrano z lokalizacji https://blog.idrsolutions.com/2018/04/java-10-improvements-to-garbage-collection-explained-in-5-minutes/
43. Rudczyk, D. (brak daty). *Obszary pamięci Maszyny Wirtualnej Javy (JVM).* Pobrano z lokalizacji https://softwareskill.pl/obszary-pamieci-maszyny-wirtualnej-javy-jvm
44. Samoylov, N. i Sanaulla, M. (2018, 9). Using application class-data sharing. W *Java 11 Cookbook - Second Edition.* Packt. Pobrano z lokalizacji https://subscription .packtpub.com/book/application_development/9781789132359/1/ch01lvl1sec16/using-application-class-data-sharing
45. Seaton, C. (2017, 11 3). *Understanding How Graal Works - a Java JIT Compiler Written in Java.* Pobrano z lokalizacji https://chrisseaton.com/truffleruby /jokerconf17/
46. Shipilëv, A. (2020, 1 8). *JVM Anatomy Quarks.* Pobrano z lokalizacji https://shipilev.net/jvm/anatomy-quarks/
47. Tokarski, J. (2018, 10 11). *„Płatna Java" podana na zimno.* Pobrano z lokalizacji https://www.pgs-soft.com/pl/blog/platna-java-podana-na-zimno/
48. Vermeer, B. (2020, 2 5). *Which Java SE version do you use in production for your main application?* Pobrano z lokalizacji https://snyk.io/blog/developers-dont-want-to-leave-java-8-as-64-hold-firm-on-their-preferred-release
49. Waksmański, M. (2020, 10 25). *Czy Java jest nadal darmowa? Jeśli nie, to co teraz? Które JDK wybrać?* Pobrano z lokalizacji https://devrev.pl/czy-java-jest-nadal-darmowa/

WYKAZ RYSUNKÓW

WYKAZ TABEL

WYKAZ WYKRESÓW

WYKAZ LISTINGÓW