JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 13 (2019) 359-365
Wysłane: 2019-09-16
Przyjęte: 2019-09-23

Porównanie wydajności wybranych algorytmów oczyszczania pamięci w Wirtualnej Maszynie Javy

Igor Kopeć*, Jakub Smołka

Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. W językach z automatycznym zarządzaniem pamięcią ważną rolę pełni odśmiecacz pamięci - mechanizm odpowiedzialny za usuwanie nieużywanych obiektów z pamięci. Algorytmy odzyskiwania pamięci są rozwijane od wielu lat i dążą do zmaksymalizowania wydajności aplikacji. W niniejszym artykule przedstawiono i porównano wydajność pięciu algorytmów automatycznego zwalniania pamięci występujących w Javie w wersji 12 na trzech aplikacjach o różnym czasie życia obiektów. Analizie została poddana szybkość aplikacji, narzut pracy odzyskiwaczy pamięci oraz przepustowość aplikacji przy dużym obciążeniu.

Słowa kluczowe: odzyskiwanie pamięci; Wirtualna Maszyna Javy; wydajność aplikacji

*Autor do korespondencji.

Adres e-mail: igor.kopec@pollub.edu.pl

A performance comparison of garbage collector algorithms in Java Virtual Machine

Igor Kopeć*, Jakub Smołka

Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. In programming languages with automatic memory management garbage collection plays an important role of cleaning unused memory. Garbage collection algorithms have been developed for many years and aim to maximize the application’s performance. This paper presents and compares a performance of five garbage collection algorithms present in current version of Java 12 in three applications with different object lifetime span. The analysis covered the system responsiveness, garbage collector workload and application throughput at high application load.

Keywords: garbage collecting; Java Virtual Machine; application performance

*Corresponding author.

E-mail address: igor.kopec@pollub.edu.pl

1. Wstęp

Języki programowania stale się rozwijają i stawiają sobie za cel szybsze i efektywniejsze wytwarzanie oprogramowania. W coraz większym stopniu programiści mogą skupić się wyłącznie na dostarczaniu właściwych rozwiązań, gdyż niskopoziomowymi aspektami technicznymi zajmuje się niezawodny silnik aplikacyjny lub sam język. W językach z automatycznym zarządzaniem pamięcią ważną rolę pełni odśmiecacz pamięci - mechanizm odpowiedzialny za zapełnianie oraz usuwanie obiektów z pamięci. Duże ilości pamięci operacyjnej i coraz wydajniejsze procesory powodują, że programiści nie skupiają się na pisaniu wydajnego kodu do momentu, kiedy system zaczyna działać w nieoczekiwany sposób. Wtedy niezbędne staje się zrozumienie mechanizmów stojących za automatycznym zarządzaniem pamięcią jak i samych mechanizmów odzyskiwania pamięci.

Ze względu na pojawienie się nowej wersji Javy a w niej nowego odśmiecacza pamięci nasuwa się pytanie o ich wydajność, różnice w ich działaniu a także ich wpływ na aplikacje. W ramach szerszego porównania w niniejszym artykule zbadano wszystkie algorytmy odzyskiwania pamięci dostępne w Javie w wersji 12.

Artykuł ma za zadanie porównać wydajność i skalowalność pięciu odśmiecaczy pamięci na wirtualnej maszynie HotSpot na przykładzie trzech aplikacji o różnej budowie.

2. Przegląd literatury

Istnieje wiele badań poświęconych odzyskiwaniu pamięci w językach z automatycznym zarządzaniem pamięcią. Dużą część z nich stanowią prace skierowane w język Java i Wirtualną Maszynę Javy skupiające się na działaniu i wydajności zawartych w niej algorytmów odzyskiwania pamięci. Największą część badań skierowanych w problematykę odśmiecania pamięci były prace porównawcze. W składzie literatury, która pozwoliła przyjrzeć się bliżej problematyce odzyskiwania pamięci znalazły się prace porównujące wydajność konkretnych algorytmów i ich wpływ na działanie aplikacji bazując na predefiniowanych testach wzorcowych [2, 4, 5, 6, 7, 8, 9]. Przyjrzano się także pracom autorów implementacji odzyskiwaczy pamięci,

w których porównywano ich osiągnięcia względem istniejących już algorytmów [3, 11]. W kilku pracach badano zgodność algorytmów z teorią generacyjną [1], skalowalność poszczególnych algorytmów [4] a także badania skupiające się na wydajności odzyskiwania w procesach całkowitego odzyskiwania pamięci [6, 7, 10]. Żadna z prac nie uwzględniała wydajności algorytmów względem rozkładu czasu życia obiektów. Nie zbadano również zachowania algorytmów, które działają w aplikacjach działających wbrew teorii generacyjnej [12]. Temat skalowalności algorytmów także nie został dokładnie zbadany biorąc pod uwagę tylko liczbę dostępnych rdzeni procesora bez uwzględnienia zwiększania nakładu pracy.

3. Cel badań

Celem badania było zmierzenie wydajności pięciu algorytmów odzyskiwania pamięci w Javie w wersji 12 na podstawie trzech aplikacji: aplikacji tworzącej obiekty, które będą znajdować się w pamięci przez krótką chwilę, aplikacji tworzącej obiekty, z których większość nigdy nie zostanie usunięta z pamięci oraz aplikacji tworzącej obiekty, których „długość życia" – czyli liczba przetrwanych cykli odzyskiwania – będzie oscylować w granicach nie pozwalających na przeniesienie danego obiektu do innej części pamięci.

Kolejnym celem było zmierzenie skalowalności algorytmu. Na podstawie wyżej wymienionych aplikacji zbadano, jak zmienia się wydajność algorytmu odzyskiwania pamięci w przypadku stopniowego zwiększania nakładu pracy.

Porównanie miało także na celu wskazać, które algorytmy będą najbardziej odpowiednie do zastosowania w aplikacjach o konkretnym sposobie działania.

4. Pamięć w Wirtualnej Maszynie Javy

Pamięć w Wirtualnej Maszynie Javy podzielona jest na stertę (ang. *heap*) oraz na pamięć natywną (ang. *off-heap*). Pierwsza przechowuje obiekty i jest w zasięgu działania odzyskiwaczy pamięci. Jest tworzona przy starcie Wirtualnej Maszyny Javy i rośnie wraz z liczbą utworzonych obiektów. Sterta jest głównym przedmiotem zainteresowań programistów, gdyż to ona ma największy wpływ na wydajność. Druga zaś przechowuje dane wspomagające pracę całej aplikacji nie będących obiektami i jest poza zasięgiem automatycznego odzyskiwania. Do pamięci natywnej zaliczamy tzw. Meta Space, schowek kompilatora JIT, stosy wątków oraz współdzielone biblioteki.

Sterta dzielona jest na przestrzeń młodą (ang. *young*) i starą (ang. *old*) i jest skutkiem zastosowania hipotezy generacyjnej [1]. Dalej pamięć młoda podzielona została na eden i dwie przestrzenie przejściowe (nazywane przetrwalnikowymi). Eden to miejsce, w którym umieszczane są obiekty nowo utworzone. Nowe obiekty pozostają w przestrzeni eden do momentu pierwszego cyklu odzyskiwania pamięci. Obiekty które po pierwszym cyklu nie zostały usunięte, zostają przeniesione do kolejnego segmentu pamięci młodej - pierwszej przestrzeni przetrwalnikowej (ang. *survivor 0*). Przetrwanie obiektu w kolejnych cyklach czyszczenia pamięci skutkuje jego przeniesieniem do kolejnych przestrzeni przetrwalnikowych aż do momentu w którym obiekt jest uznawany za obiekt stary, po czym zostaje wypromowany do starszej generacji. Przestrzeń starej generacji (zwana także przestrzenią *tenured*) jest przestrzenią ciągłą, w której znajdują się najdłużej żyjące obiekty aplikacji.

Pamięć natywna jest częścią pamięci zarządzaną głównie przez wirtualną maszynę. Dalsza dekompozycja pamięci natywnej stanowi rozdzielenie na obszary wspólne dla całej wirtualnej maszyny oraz prywatne obszary pojedynczych wątków. Pierwszą przestrzenią jest *Meta Space*, gdzie znajdują się podprzestrzenie zawierające definicje klas, metod i pól, literały znakowe, stałe i referencje oraz kody klas. Nie posiada górnej granicy wielkości pamięci - ogranicza ją tylko pamięć systemu operacyjnego. Kolejną, ważną z punktu widzenia wydajności jest przestrzeń schowka kompilatora JIT , która to stanowi kod zoptymalizowany przez kompilator JIT. Ostatnią częścią pamięci natywnej jest przestrzeń wykorzystywana przez poszczególne wątki, która składa się ze stosu (ang. *stack*) oraz z licznika operacji (ang. *program counter*). Stosem jest kolejka LIFO zawierająca ramki - tablicę zmiennych lokalnych oraz wartości zwracane przez wywoływane metody. Licznik operacji natomiast posiada referencję do aktualnie wykonywanej instrukcji. Każdy wątek posiada swój stos wywołań metod.

Empiryczne badania [1] dowiodły, iż najwięcej jest obiektów najkrócej żyjących, natomiast odwołania do starych obiektów są stosunkowo rzadkie. Obserwacje te są określane mianem hipotezy generacyjnej i implikują podział pamięci na starą i młodą generację ze względu na ich odmienną charakterystykę. Segmentacja pamięci pozwala na zastosowanie różnych algorytmów odzyskiwania pamięci do różnych generacji. Każdy algorytm jest zoptymalizowany pod konkretne właściwości danej generacji. Podział pozwala także na selektywne uruchamianie sprzątania dla określonych segmentów.

5. Algorytmy odzyskiwania pamięci

Według oficjalnej dokumentacji w Javie 12 istnieje pięć odśmiecaczy pamięci - Szeregowy (ang. *Serial*), Równoległy (ang *Parallel*), CMS (*Concurrent Mark Sweep*), G1 (*Garbage First)* i Shenandoah.

Algorytm szeregowy

Algorytm szeregowy (ang. *Serial*) jest najprostszym algorytmem działającym tylko w jednym wątku. Nie uzyskuje lepszej wydajności na maszynach o większej liczbie rdzeni procesora, ale jest dobrym wyborem w środowisku jednowątkowym oraz wtedy, gdy na jednym komputerze liczba działających Wirtualnych Maszyn jest równa lub większa liczbie dostępnych wątków procesora. Używając tego algorytmu w środowisku wielowątkowym blokujemy pracę innych wątków. Ten odzyskiwacz pamięci zatrzymuje całą aplikację na czas swojego działania [14].

Algorytm równoległy

Algorytm równoległy (ang. *Parallel*) jest podobny do algorytmu szeregowego ale różni go to, że do odzyskiwania używa wielu wątków. Zarówno młoda i stara generacja czyszczona jest równolegle. Pomimo tego, iż algorytm na czas działania zatrzymuje całą aplikację zrównoleglenie pracy powoduje, że przerwy są proporcjonalnie krótsze. Mając dostępnych X rdzeni, odśmiecacz użyje X wątków. Do najefektywniejszej pracy kolektor potrzebuje przynajmniej trzech wątków - wtedy dopiero zauważalna jest różnica pomiędzy algorytmem szeregowym [14].

Algorytm Concurrent Mark Sweep

Algorytm Concurrent Mark Sweep (CMS) jest algorytmem współbieżnym i próbuje wykonywać większą część swojej pracy nie przerywając działania aplikacji. Osiąga to przez to, iż w starej generacji stara się nie używać częstego kopiowania i kompaktowania obiektów, tylko coraz bardziej zwiększa dostępną pamięć dla aplikacji. W przypadku gdy ilość zajmowanego miejsca jest dostatecznie duża wykonywane jest całkowite czyszczenie kopiująco-kompaktujące. Algorytm większość pracy wykonuje równolegle z działaniem aplikacji. Niestety wykonując odzyskiwanie współbieżnie, wątki algorytmu zabierają część czasu procesora wątkom aplikacji, więc wydajność aplikacji używającej tego algorytmu zmniejsza się. Stanowi to dobry kompromis, jeśli aplikacja nie powinna mieć większych przestojów [14].

Algorytm G1

Algorytm G1 (ang. *Garbage first*) działa na podobnej zasadzie co algorytm CMS lecz jest jeszcze bardziej współbieżny i stosuje odmienny podział generacyjny sterty niż wcześniej wymienione algorytmy. Pozwala programiście samemu określić czas pauzy w każdym cyklu odzyskiwania. Sterta dla tego algorytmu nie jest już dzielona na młodą i starą generację lecz na regiony o takiej samej wielkości. W zależności od wielkości dostępnej pamięci wielkość regionu waha się od 1MB do 32MB po to, aby liczba regionów wynosiła 2048. Wyróżnia się pięć rodzajów regionów - *eden, przetrwalnikowy, ogromny, stara generacja* i *nieużywany*. Podobnie jak w innych algorytmach, nowe obiekty alokowane są w edenie. Gdy się starzeją przechodzą do regionu przetrwalnikowego, a następnie do regionu starej generacji. Obiekty, których wielkość przekracza 50% miejsca regionu alokowane są w regionie ogromnym - jest to sytuacja dość rzadka. Regiony nieużywane czekają na przydzielenie im jednej z pozostałych ról oddając nieużywaną pamięć systemowi operacyjnemu [14].

Algorytm Shenandoah

Algorytm Shenandoah jest algorytmem redukującym czas pauz do minimum poprzez wykonywanie większości swojej pracy współbieżnie do działania aplikacji. Zrównolegleniu uległa tutaj również faza kompaktowania obiektów, wskutek czego czasy pauz nie zależą już dłużej od wielkości dostępnej pamięci. Algorytm stworzono z myślą o stertach osiągających wielkość powyżej 100GB. Jednym z założeń tego algorytmu jest osiąganie czasu trwania wszystkich pauz odzyskiwania nie większych niż 10 ms. dla małych oraz dla dużych obszarów pamięci. Co za tym idzie, Shenandoah może być używany tam, gdzie najczęściej używane dziś algorytmy (CMS, G1) po prostu nie są wystarczająco wydajne. Z drugiej strony Shenandoah jest mało wydajny dla małych aplikacji [15].

Podział pamięci na regiony wygląda podobnie jak w algorytmie G1. Zasada działania jest również podobna. Różnicą jest to, że Shenandoah nie przestrzega zasady, iż większość obiektów umiera szybko, ponieważ nie wszystkie aplikacje działają zgodnie z tym założeniem. Algorytm nie dzieli stosu na młodą i starą generacje. Region może zawierać zarówno młode jak i stare obiekty. W związku z tym proces odzyskiwania jest zawsze procesem czyszczenia całego stosu (ang *Full garbage collection*) [15].

6. Metoda badawcza

W celu przeprowadzenia badań stworzono trzy aplikacje, które różniły się rozkładem czasu trwania obiektów w pamięci. Każda aplikacja została przetestowana używając wszystkich algorytmów oczyszczania pamięci wspomnianych w rozdziale piątym w ich podstawowej konfiguracji. Aplikację testowano pod kątem czasu wykonanego testu, oraz *przepustowości* aplikacji - stosunku czasu działania aplikacji do czasu pauzy systemu związanej z pracą odzyskiwacza pamięci. Prześledzono także działania algorytmu odzyskiwania pamięci. Ponadto zbadano skalowalność każdej aplikacji dla poszczególnych algorytmów oczyszczania stopniowo zwiększając nakład pracy. Wszystkie testy poprzedzono pięcioma iteracjami próbnymi w celu „rozgrzania" Wirtualnej Maszyny Javy. W celu uzyskania poprawnego wyniku, każdy przypadek testowy został wykonany dziesięć razy. Następnie wyniki zostały uśrednione.

Wyniki uzyskano analizując zapis plików dziennika generowanych przez algorytm odzyskiwania pamięci oraz poprzez dostarczany wraz z Javą program *jStat* [13] służący do monitorowania stanu pamięci aplikacji.

6.1. Opis aplikacji testowych

Pierwszą aplikację zaprojektowano tak, aby tworzyła bardzo dużo małych obiektów. Aplikacja sprawdza, jak algorytm poradzi sobie z odzyskiwaniem dużej liczby obiektów w młodej generacji.

Listing nr 1 przedstawia test aplikacji tworzącej obiekty krótko żyjące. Test dodaje do listy podaną liczbę obiektów klasy akumulator. Klasa akumulator posiada zmienną typu Long i metodę dodającą do akumulatora losową wartość liczbową. Po wyjściu z funkcji zmienna *rand* jest uznawana jako obiekt martwy. W następnym kroku, lista akumulatorów jest iterowana a na każdym z nich wykonywana jest funkcja dodająca losową wartość. W taki sposób test tworzy 1,5 miliona małych obiektów. Następnie wartości wszystkich obiektów są sumowane do zmiennej *sum*. Na końcu test usuwa wszystkie elementy z listy.

Przykład 1. Listing przedstawiający kod testu pierwszej aplikacji

```
public void startTest() {
  List<Accumulator> accumulators = new LinkedList<>();
  //dodanie obiektów Akumulator do listy
  for (int i = 0; i < numberOfAccumulators; i++) {
    accumulators.add(new Accumulator());
  }

  //wywołanie 10000 razy funkcji addValue() akumulatora
  AtomicLong tempSum = new AtomicLong();
  accumulators.forEach(accumulator -> {
    //3000 wywołań funkcji
    for (int i = 0; i < numberOfAddedValues; i++) {
      tempSum.addAndGet(accumulator.addValue());
    }
  });

   long sum = accumulators.stream()
      .mapToLong(Accumulator::getAccumulatedAmount)
      .sum();
   BigInteger bigInteger = new
   BigInteger(String.valueOf(sum));
   Iterator<Accumulator> it = accumulators.iterator();
   while (it.hasNext()) {
    Accumulator next = it.next();
    it.remove();
  }
}
```

Drugą aplikację zbudowano tak aby przechowywała dużo obiektów w pamięci jak najdłużej po to, aby wymusić promocję obiektów do starej generacji.

Test aplikacji tworzącej obiekty długożyjące przedstawiono na listingu nr 2. Celem testu jest stworzenie określonej liczby węzłów w drzewie. W pętli na przemian dodawanych jest 50 dużych obiektów do binarnego drzewa. Następnie usuwany jest losowo jeden węzeł. Usunięcie węzła powoduje, że jego potomkowie także są usunięci z węzła. Wymusza to proces odzyskiwania pamięci. Elementy które są najbliżej korzenia drzewa mają dużą szansę na przetrwanie dostatecznie długo, aby zostać zakwalifikowane do przeniesienia do starej generacji. Ciągłe tworzenie obiektów i alokacja w starej generacji wymusza proces oczyszczania nie tylko młodej ale i starej generacji stosu.

Przykład 2. Listing przedstawiający kod testu drugiej aplikacji

```
public void startTest() {
  try {
    while (nodeCounter.get() < maxNumberOfNodes) {
      //dodanie elementów
      int randomInt =
   ThreadLocalRandom.current().nextInt(50);
   for (int i = 0; i < randomInt; i++) {
   Node node = new
   Node(nodeCounter.getAndIncrement());
    tree.insert(node);
    overallSum += (Long) node.getResultValue();
  }
      //usunięcie węzła
  Iterator iterator = tree.inorderIterator();
  ArrayList<Node> elements = new ArrayList<>();
  while (iterator.hasNext()) {
    elements.add((Node) iterator.next());
  }
  double rand = ThreadLocalRandom.current().nextDouble(0,
1.01);
  Node element;
  if (rand < 0.05) {
    element = chooseElement(elements, lastXPercent(0.4));
  } else if (rand < 0.10) {
    element = chooseElement(elements, lastXPercent(0.2));
  } else {
    element = chooseElement(elements,
this::cut100OrTenPercent);
  }
  tree.delete(element);

    }
  } catch (InterruptedException e) {
    e.printStackTrace();
  } finally {
    log.debug("Sum: {}", overallSum);
  }
}
```

Trzecią aplikacje skonstruowano tak, aby pracę wykonywała na wielu wątkach w celu zwielokrotnienia liczby wykonywanych operacji oraz liczby alokowanych obiektów o średnim czasie życia.

Aplikacja tworzy obiekty które od razu kwalifikują się do odzyskania przez algorytm odzyskiwania pamięci oraz takie, które w tej pamięci pozostają na pewien czas. Test został skonstruowany tak, aby długość życia danego obiektu oscylowała w granicach progu zakwalifikowania obiektu do przeniesienia do starej generacji. W praktyce heurystyki odśmiecacza pamięci nie są w stanie jednoznacznie stwierdzić czy obiekty będą przeniesione do starej generacji pamięci, więc algorytm nie może zastosować odpowiednich optymalizacji. Test ma za zadanie sprawdzić zachowanie algorytmu w aplikacji działającej wbrew teorii generacyjnej. Każdy wątek wykonuje takie samo zadanie przedstawione w listingu nr 3. Badanie wykonano na czterech wątkach - taką liczbę wątków posiadał komputer na którym przeprowadzono badania.

Przykład 3. Listing przedstawiający kod testu trzeciej aplikacji

```
public Object doOperation() {
   char[] chars = getCharArray(objectSize);
   List<String> liveList = new
ArrayList<String>(myNumLive);

  // Utworzenie listy
  for (int i = 0; i < myNumLive; i++) {
    liveList.add(new String(chars));
  }
  for (int j = 0; j < 5_000_000; j++) {
    // Tworzymy dużą liczbę obiektów
    for (int i = 0; i < fractionLive; i++) {
      String garbageObject = new String(chars);
    }
    // W losowym miejscu zamieniamy obiekty w liście
    // na inne
    int index = (int) (Math.random() * myNumLive);
    liveList.set(index, new String(chars));
  }
  return liveList;
}
```

Dla powyższych aplikacji wykonano podane scenariusze w dwóch przypadkach. Pierwszy zakładał użycie stałych parametrów dla wszystkich algorytmów w celu zbadania wydajności. Były to odpowiednio:

- 500 000 obiektów akumulator i 3 000 operacji na każdym obiekcie,

- 100 000 dodanych węzłów do drzewa. 50 obiektów dodawanych w każdym cyklu,
- Cztery wątki aplikacyjne.

Drugi przypadek badał skalowalność aplikacji i zakładał użycie zmiennych parametrów odpowiednio:

- 10 000 - 12 400 000 obiektów akumulator i 3 000 operacji na każdym obiekcie,
- 100 000 dodanych węzłów do drzewa i 100 - 50 000 obiektów dodawanych w każdym cyklu,
- 1 - 8 wątków aplikacyjnych.

7. Analiza porównawcza

W analizie porównano:

- Całkowity czas testu i przepustowość aplikacji,
- Narzut czasu odzyskiwania i liczbę procesów odzyskiwania dla generacji młodej i starej,
- Średni i maksymalny czas pauz systemu,
- Skalowalność poszczególnych algorytmów.

W tabeli 1, 2 i 3 został przedstawiony średni czas wykonania całego testu wydajnościowego na trzech aplikacjach a także przepustowość aplikacji dla poszczególnych algorytmów.

Tabela 1. Czas wykonania testu i przepustowość aplikacji dla poszczególnych algorytmów dla pierwszej aplikacji

Algorytm	Czas testu [s]	Przepustowość
Szeregowy	43,83	96,66%
Równoległy	46,82	99,28%
CMS	44,68	98,38%
G1	53,74	98,13%
Shenandoah	48,33	98,72%

Tabela 2. Czas wykonania testu i przepustowość aplikacji dla poszczególnych algorytmów dla drugiej aplikacji

Algorytm	Czas testu [s]	Przepustowość
Szeregowy	51,87	96,74%
Równoległy	52,02	97,67%
CMS	49,09	98,40%
G1	51,87	98,99%
Shenandoah	58,77	88,14%

Tabela 3. Czas wykonania testu i przepustowość aplikacji dla poszczególnych algorytmów dla trzeciej aplikacji

Algorytm	Czas testu [s]	Przepustowość
Szeregowy	38,2	21,51%
Równoległy	35,8	40,65%
CMS	25,0	62,70%
G1	24,9	77,76%
Shenandoah	90,4	21,27%

Tabele 4, 5 i 6 przedstawiają całkowity czas poświęcony na odzyskiwanie pamięci oraz liczbę procesów odzyskiwania pamięci. Poniższe wyniki zostały rozdzielone na młodą i starą generację.

Tabela 7, 8 i 9 przedstawia średni czas pauz występujących w danych testach wydajnościowych oraz maksymalny zarejestrowany czas pauzy w młodej i starej generacji pamięci.

Tabela 4. Narzut czasu i liczba odzyskiwań dla poszczególnych algorytmów dla pierwszej aplikacji

Algorytm	Narzut czasu odzyskiwania [s]		Liczba procesów odzyskiwania	
	młoda gen.	stara gen.	młoda gen.	stara gen.
Szeregowy	1,929	0,000	124,30	0,0
Równoległy	0,446	0,000	103,90	0,0
CMS	1,095	0,000	127,4	0,0
G1	1,717	0,000	113,60	0,0
Shenandoah	0,000	10,216	0,0	404,20

Tabela 5. Narzut czasu i liczba odzyskiwań dla poszczególnych algorytmów dla drugiej aplikacji

Algorytm	Narzut czasu odzyskiwania [s]		Liczba procesów odzyskiwania	
	młoda gen.	stara gen.	młoda gen.	stara gen.
Szeregowy	3,293	1,845	93,7	15,4
Równoległy	2,969	1,850	128,6	17,4
CMS	3,007	0,493	473,4	54,5
G1	2,530	0,000	328,9	0,0
Shenandoah	0,000	2,835	0,0	153,3

Tabela 6. Narzut czasu i liczba odzyskiwań dla poszczególnych algorytmów dla trzeciej aplikacji

Algorytm	Narzut czasu odzyskiwania [s]		Liczba procesów odzyskiwania	
	młoda gen.	stara gen.	młoda gen.	stara gen.
Szeregowy	26,688	7,199	66,6	4,2
Równoległy	20,181	3,472	287,1	866,2
CMS	11,415	0,989	227,7	4,1
G1	16,303	9,687	61,0	1,3
Shenandoah	0,000	36,116	0,0	489,7

Tabela 7. Maksymalny i średni czas pauz dla poszczególnych algorytmów dla pierwszej aplikacji

Algorytm	Średni czas pauz [ms]	Maksymalny czas pauzy [ms]	
		młoda gen.	stara gen.
Szeregowy	15,29	212,7	0,0
Równoległy	0,45	471,8	0,0
CMS	8,58	619	0,0
G1	15,26	198,3	0,0
Shenandoah	0,80	0,0	4,41

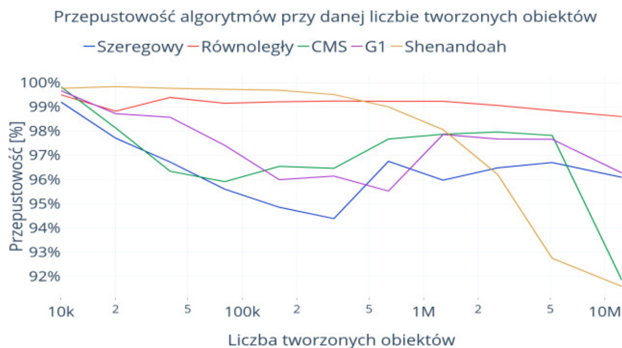
Tabela 8. Maksymalny i średni czas pauz dla poszczególnych algorytmów dla pierwszej aplikacji

Algorytm	Średni czas pauz [ms]	Maksymalny czas pauzy [ms]	
		młoda gen.	stara gen.
Szeregowy	56,0	128,7	334,2
Równoległy	33,9	100,1	247,9
CMS	7,9	49,8	241,2
G1	6,8	69,2	0,0
Shenandoah	1,0	0,0	6,2

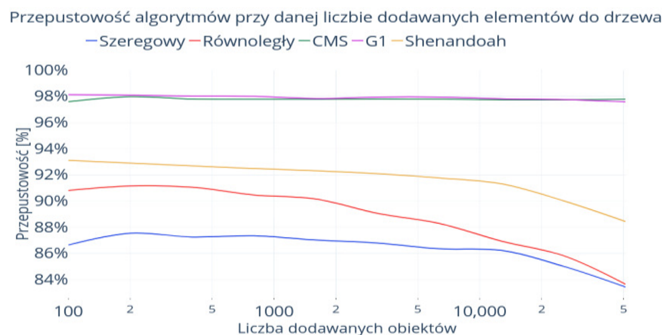
Tabela 9. Maksymalny i średni czas pauz dla poszczególnych algorytmów dla pierwszej aplikacji

Algorytm	Średni czas pauz [ms]	Maksymalny czas pauzy [ms]	
		młoda gen.	stara gen.
Szeregowy	480,0	4502,2	4252,7
Równoległy	320,0	501,4	1524,4
CMS	49,4	1361,0	58,1
G1	106,3	185,2	0,0
Shenandoah	5,2	0,0	25,8

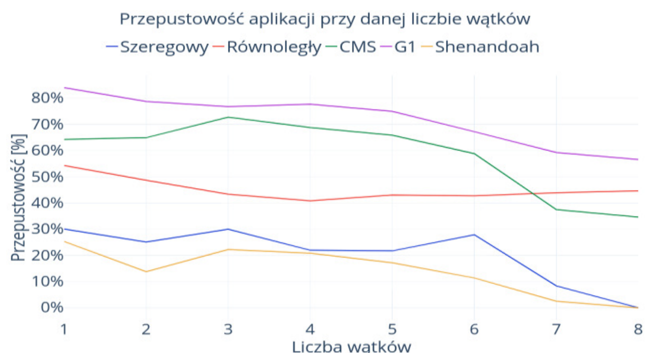
W ramach testów wydajnościowych trzech aplikacji, zbadano także zmianę przepustowości algorytmów na zwiększające się obciążenie pracą. Wyniki testów skalowalności przedstawia rysunek 1, 2 i 3.



Rys. 1. Wykres przedstawiający zmieniającą się przepustowość przy danej liczbie tworzonych obiektów w aplikacji pierwszej.



Rys. 2. Wykres przedstawiający zmieniającą się przepustowość przy danej liczbie dodawanych elementów w aplikacji drugiej.



Rys. 3. Wykres przedstawiający zmieniającą się przepustowość przy danej liczbie wątków roboczych w aplikacji trzeciej.

8. Wnioski

Wyniki pokazują, jak bardzo wykonanie testu różniło się w przypadku zastosowania różnych algorytmów odzyskiwania pamięci. Algorytmy współbieżne CMS i G1 charakteryzują się najwyższą przepustowością i dobrą skalowalnością i warto z nich korzystać niemal we wszystkich typach aplikacji. We wszystkich przypadkach algorytmy współbieżne osiągały bardzo krótkie pauzy, zazwyczaj o długości niezauważalnej przez użytkownika aplikacji. Ich doskonały wynik zawdzięczany jest wysokiej współbieżności, co pozwalała im na działanie wraz z działającą aplikacją. Cechują ich także niższe maksymalne pauzy i liczba procesów odzyskiwania pamięci. Algorytmy CMS i G1 najlepiej poradziły sobie z testem aplikacji drugiej i trzeciej. W pierwszej natomiast ustąpiły algorytmowi równoległemu. Algorytmy współbieżne przystosowane są do efektywnego odzyskiwania starej generacji. Ze względu, że wyniki obu algorytmów są podobne, ciężko jest wskazać który algorytm sprawdza się lepiej. CMS osiągał mniejsze czasy pauz, natomiast w przypadku użycia algorytmu G1 przepustowość aplikacji była wyższa.

Najmniejsze czasy pauz osiągał najnowszy algorytm Shenandoah, który niezależnie od rodzaju aplikacji czy obciążenia potrafił osiągać stałe wartości pauz nie przekraczające średniej wartości 10 ms. Najdłuższa odnotowana pauza nie przekraczała 25 ms., kosztem wolniejszego wykonania testu i zmniejszonej przepustowości.

Algorytmy nie współbieżne takie jak szeregowy czy równoległy mogą natomiast z powodzeniem być wykorzystywane w aplikacjach w których nie przechowuje się sporej liczby obiektów w pamięci na długi czas, a także w środowiskach jednoprocesorowych. Sporą wadą tych algorytmów jest to, że powodują one nieuniknione długie pauzy systemu najbardziej zauważalne przy odzyskiwaniu starej generacji. Z wyników można odczytać, że algorytm równoległy jako algorytm odzyskujący pamięć w wielu wątkach, powinien być lepszym wyborem od algorytmu szeregowego w każdym przypadku.

Analizując wyniki trzech pierwszych aplikacji stwierdzono, że algorytmy działają poprawnie w aplikacjach, które podążają za teorią generacyjną - większość obiektów jest szybko niepotrzebna a obiektów znajdujących się w pamięci dłużej jest mniej. Dobrze to widać na przykładzie aplikacji trzeciej, gdzie czas życia obiektów został uśredniony. Wyniki trzeciej aplikacji pokazują, że heurystyki algorytmów nie są dobrze zoptymalizowane do tego typu aplikacji.

Literatura

[1] Appel A. W.: Simple generational garbage collection and fast allocation. Software: Practice and Experience, nr 2/1989, Tom 19, s. 171-183.

[2] Carpen-Amarie M., Marlier P., Felber P., Thomas G.: A Performance Study of Java Garbage Collectors on Multicore Architectures, Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, 2015, s. 20-29.

[3] Detlefs D., Flood C., Heller S., Printezis T.: Garbage-first garbage collection. In Proceedings of the 4th international symposium on Memory management. ACM. 2004, s. 37-48.

[4] Gidra L., Thomas G., Sopena J., Shapiro M.: Assessing the scalability of garbage collectors on many cores. In Proceedings of the 6th Workshop on Programming Languages and Operating Systems, ACM, 2011, s. 7.

[5] Grgic H., Mihaljevic B., Radovan A.: Comparison of garbage collectors in Java programming language, 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, 2018.

[6] Li H., Wu M., Chen H.: Analysis and Optimizations of Java Full Garbage Collection, Proceedings of the 9th Asia-Pacific Workshop on Systems, 2018, s. 18.

[7] Li H., Wu M., Zang B., Chen H.: ScissorGC: Scalable and Efficient Compaction for Java Full Garbage Collection, 2019.

[8] Suo K., Rao J., Jiang H., Srisa-an W.: Characterizing and optimizing hotspot parallel garbage collection on multicore systems. EuroSys, 2018, s. 35-1.

[9] Tauro C., Prabhu M., Saldanha V.: CMS and G1 Collector in Java 7 Hotspot: Overview, Comparisons and Performance Metrics. International Journal of Computer Applications, 43(11), 2012.

[10] Yu Y., Lei T., Zhang W., Chen H., Zang B.: Performance analysis and optimization of full garbage collection in memory-hungry environments. In ACM SIGPLAN Notices, ACM. Nr 7/2016, Tom 51, s. 123-130.

[11] Flood C. H., Kennke R., Dinn A., Haley A., Westrelin R.: Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, ACM, 2016, s. 13.

[12] Struktura pamięci Wirtualnej Maszyny Java , https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf [24.03.2019].

[13] Dokumentacja programu jStat, https://docs.oracle.com/javase/7/docs/technotes/tools/share/jstat.html [24.03.2019].

[14] Hunt C., Beckwith M., Parhar P., Rutisson B.: Java Performance Companion, Addison-Wesley Professional, 2016.

[15] Opis algorytmu Shenandoah https://wiki.openjdk.java.net/display/shenandoah/Main 24.03.2019.