Zaawansowane Techniki Programowania

Projekt 1

Mikołaj Zuziak
Informatyka st.2
Gr. 1b

Wstęp

Przedmiotem badania jest aplikacja konsolowa napisana w języku C#, która służy do przetwarzania obrazów poprzez zastosowanie filtru Laplace'a. Program został zaprojektowany jako platforma testowa do analizy wpływu różnych technik zarządzania pamięcią oraz optymalizacji kodu na wydajność aplikacji.

Aplikacja implementuje kilka wariantów algorytmu nakładania filtru Laplace'a na obrazy, różniących się podejściem do zarządzania pamięcią:

1. **Wersja zarządzana (managed)** - wykorzystująca standardowe mechanizmy zarządzania pamięcią przez Garbage Collector w .NET
2. **Wersja zarządzana z użyciem wskaźników (managed fixed)** - korzystająca z bloku fixed do przypięcia pamięci
3. **Wersja niezarządzana (unmanaged)** - używająca bezpośredniego dostępu do pamięci poprzez wskaźniki
4. **Wersja niezarządzana z optymalizacją wskaźników (unmanaged fixed)** - z dodatkowymi optymalizacjami dostępu do pamięci
5. **Wersja niezarządzana z pulą pamięci (unmanaged pooled)** - wykorzystująca mechanizm puli pamięci (memory pooling)

Program umożliwia także włączenie dodatkowych opcji wpływających na zarządzanie pamięcią:

- Przetwarzanie równoległe obrazów (ENABLE_PARALLEL)
- Wymuszenie ręcznego uruchomienia garbage collectora (GC_COLLECT)
- Kompaktowanie sterty obiektów dużych (COMPACT_ONCE)
- Automatyczne zwalnianie zasobów (DISPOSE)
- Wykorzystanie bloku fixed do optymalizacji dostępu do pamięci (USE_FIXED)
- Wykorzystanie puli pamięci (USE_POOLING)

Wszystkie opcje można konfigurować poprzez zmienne środowiskowe, co ułatwia przeprowadzanie testów wydajnościowych bez konieczności rekompilacji kodu.

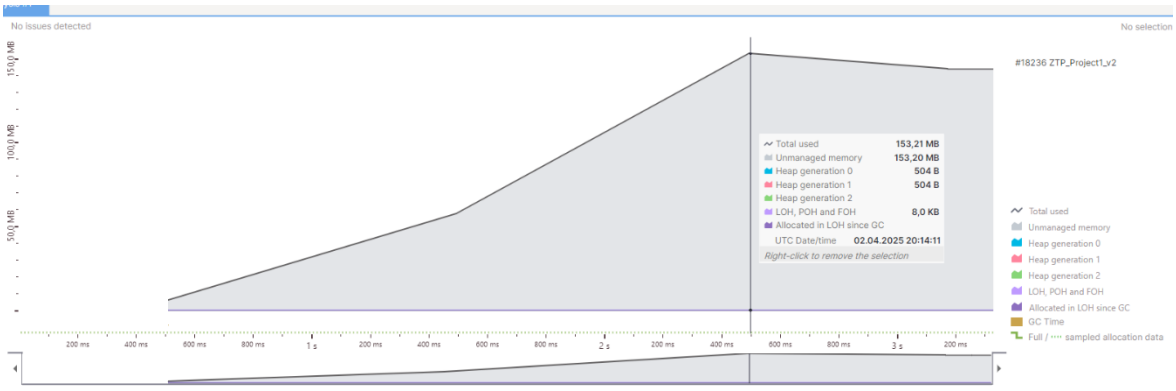
Celem badania jest analiza wpływu powyższych technik na wydajność, zużycie pamięci oraz czas przetwarzania obrazów, co pozwoli na wyciągnięcie wniosków dotyczących optymalizacji aplikacji .NET przetwarzających duże ilości danych.

Wszystkie pomiary będą przeprowadzane dla serii 9 obrazów w formacie JPG.

Wstępne pomiary wydajności

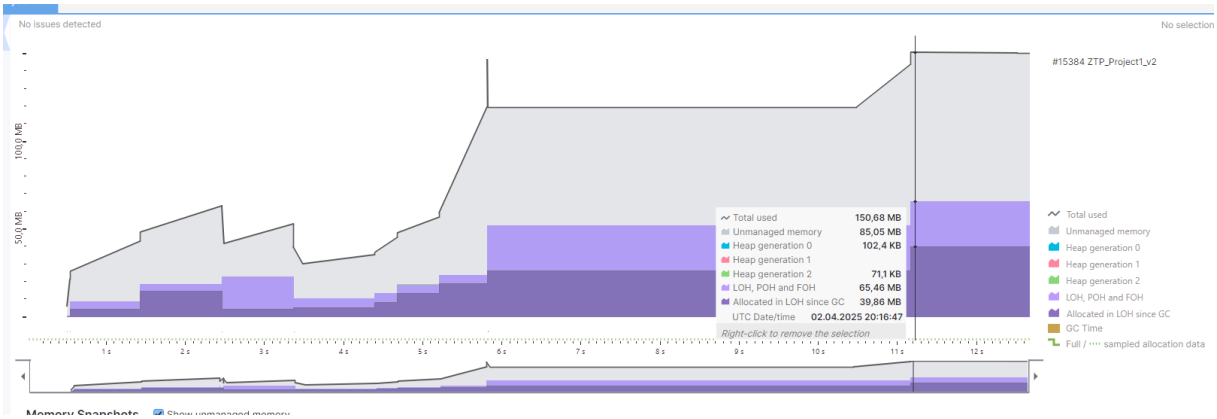
W ramach wstępnego etapu badań przeprowadzono porównanie dwóch podstawowych konfiguracji programu: wersji z zarządzaną pamięcią (managed) oraz wersji z niezarządzaną pamięcią (unmanaged).

Pomiar Unmanaged



Czas: 2540 ms

Pomiar Managed

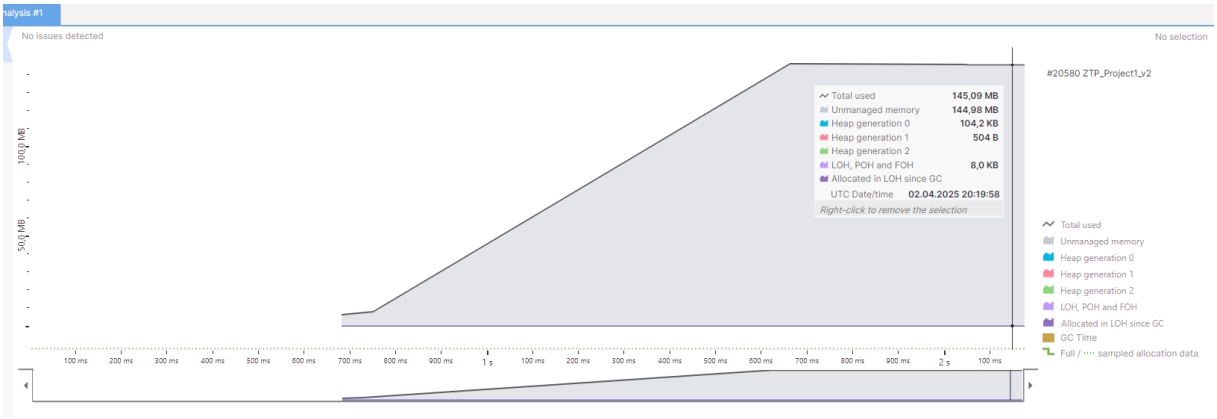


Czas: 11996 ms

Włączenie przetwarzania równoległego

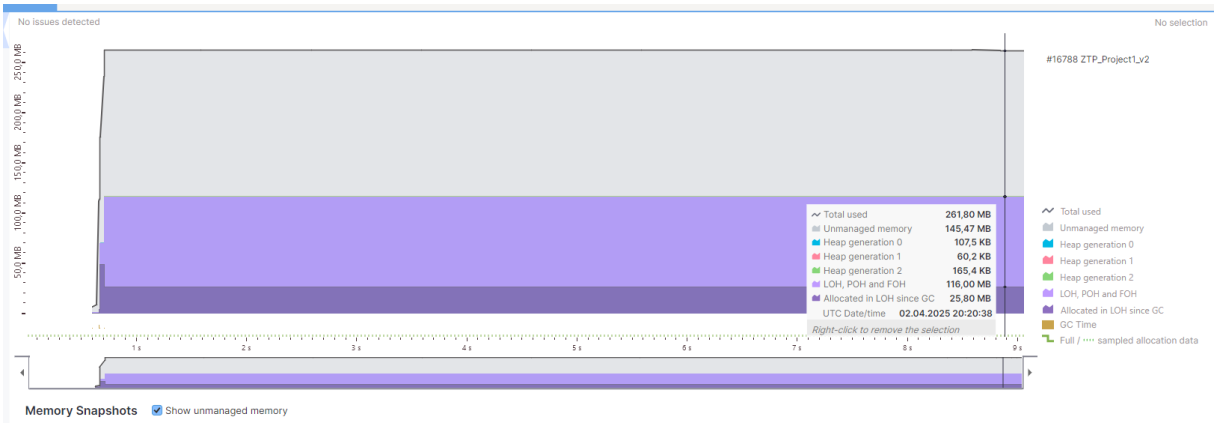
Włączenie opcji ENABLE_PARALLEL=true pozwala na równoczesne przetwarzanie wielu obrazów z wykorzystaniem klasy Parallel.ForEach, co umożliwia efektywne wykorzystanie wielordzeniowych procesorów.

Pomiar Unmanaged



Czas: 1337 ms

Pomiar Managed



Czas: 8213 ms

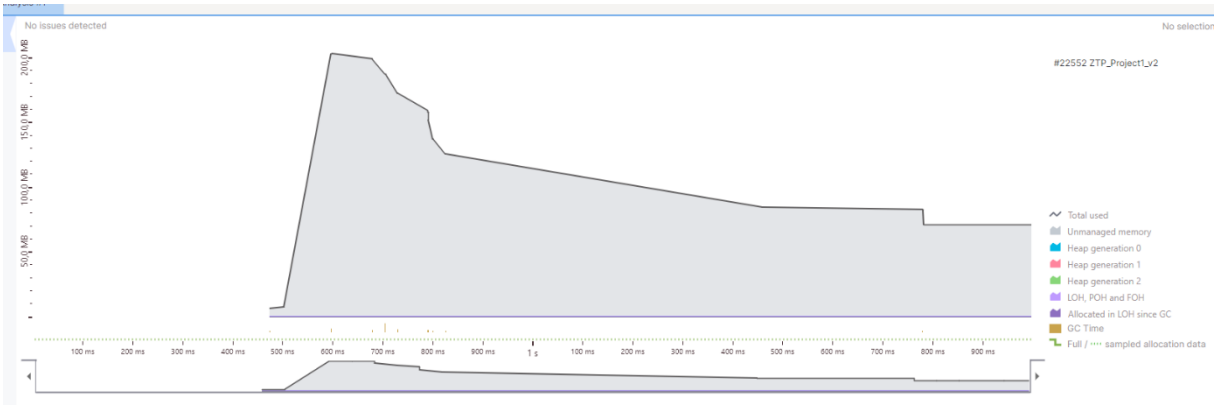
Zastosowywanie Parallel.ForEach powoduje:

- Skrócenie czasu wykonywania programu w obu wersjach.
- Większym zużyciem pamięci w wersji zarządzanej połączonej z jednolitym rozłożeniem zużycia w czasie.
- Równiejszym rozłożeniem zużycia pamięci w wersji niezażądanej.

Ręczne wywoływanie mechanizmu Garbage Collector

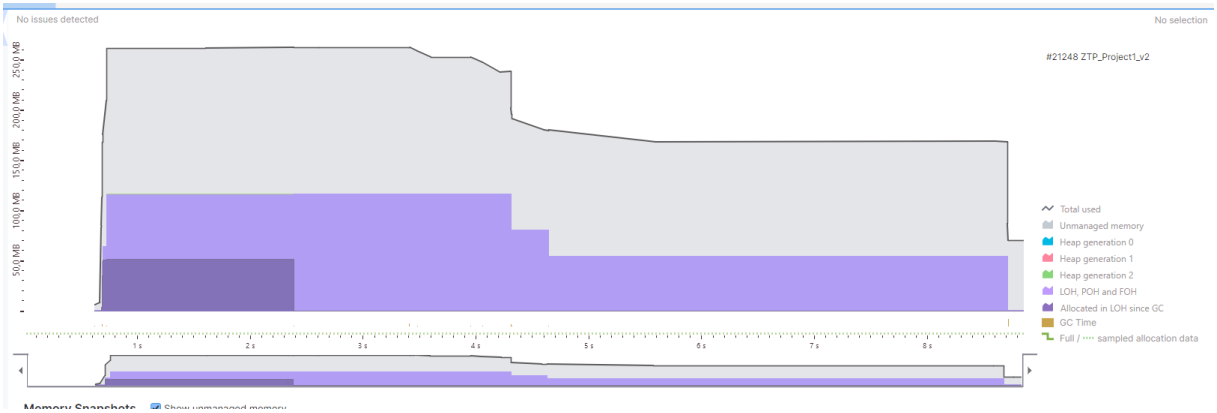
Włączenie opcji GC_COLLECT=true pozwala na ręczne wywołanie mechanizmu Garbage Collector, co umożliwia wskazanie momentu opróżnienia pamięci zarządzanej z zasobów, które nie są już potrzebne.

Pomiar Unmanaged



Czas: 1291

Pomiar Managed



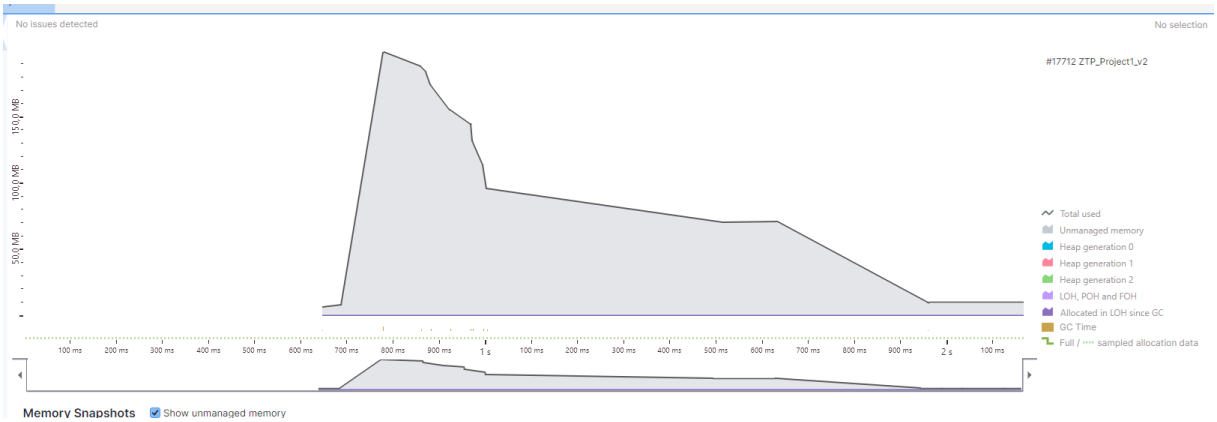
Czas: 8074 ms

Wykres przedstawia momenty wywołania mechanizmu Garbage Collector i stopniowe opróżnianie pamięci w czasie działania programu.

Ręczne wywoływanie metody Dispose

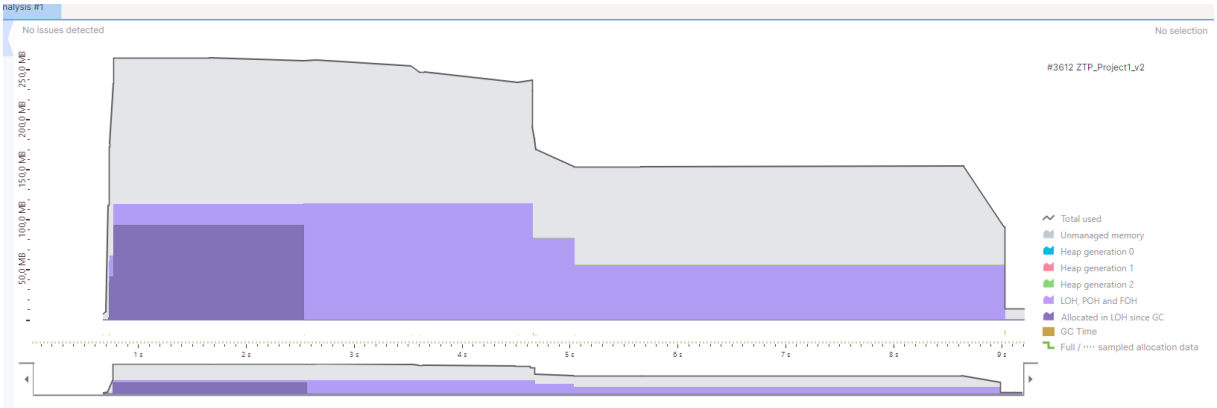
Włączenie DISPOSE=true pozwala na ręczne wywołanie mechanizmu Garbage Collector, co umożliwia opróżnienie pamięci niezażądanej ze wskazanego zasobu.

Pomiar Unmanaged



Czas: 1287ms

Pomiar Managed



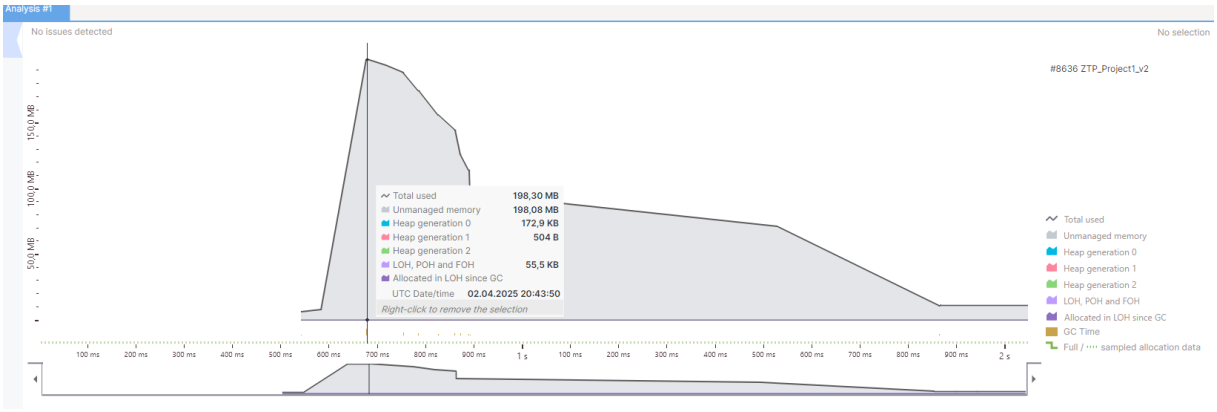
Czas: 8346ms

Wykresy wskazują momenty opróżnienia pamięci niezarządzanej ze wskazanych zasobów.

Zmiana trybu kompaktowania

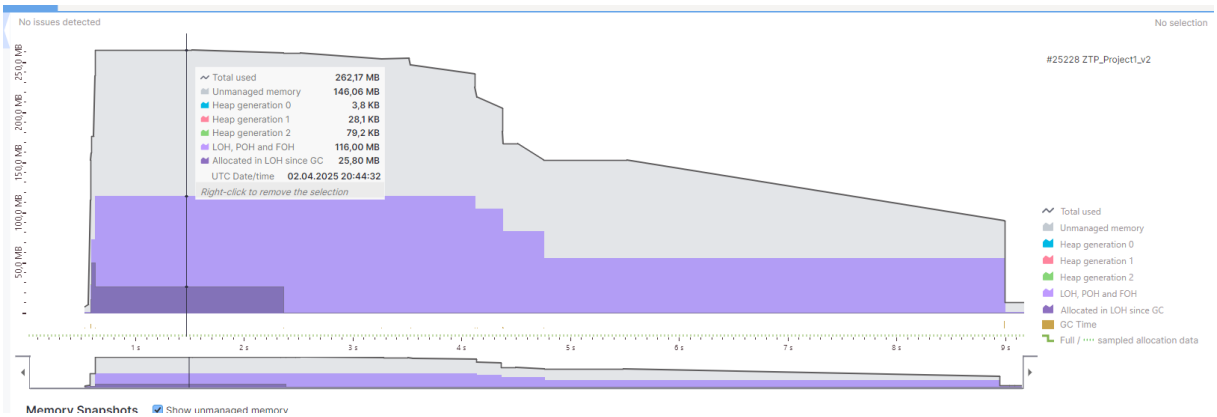
Włączenie COMPACT_ONCE=true pozwala na jednorazowe skompaktowanie dużego stosu obiektów (LOH), co może zredukować fragmentację pamięci. Dzięki temu program potencjalnie zużywa mniej pamięci przy przetwarzaniu wielu dużych obrazów, choć może wprowadzić dodatkowe opóźnienie podczas operacji kompaktowania.

Pomiar Unmanaged



Czas: 1296ms

Pomiar Managed

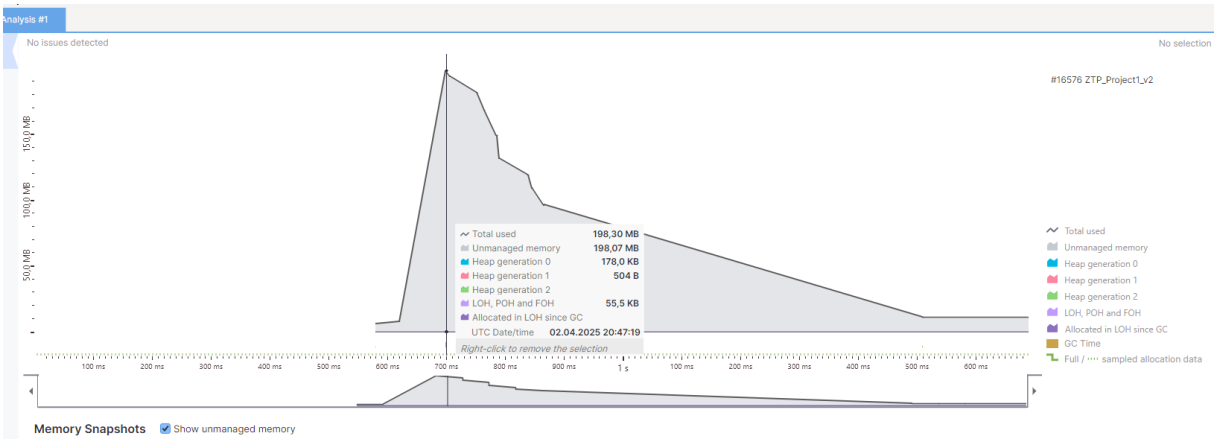


Czas: 8443ms

Wykorzystanie bloku fixed

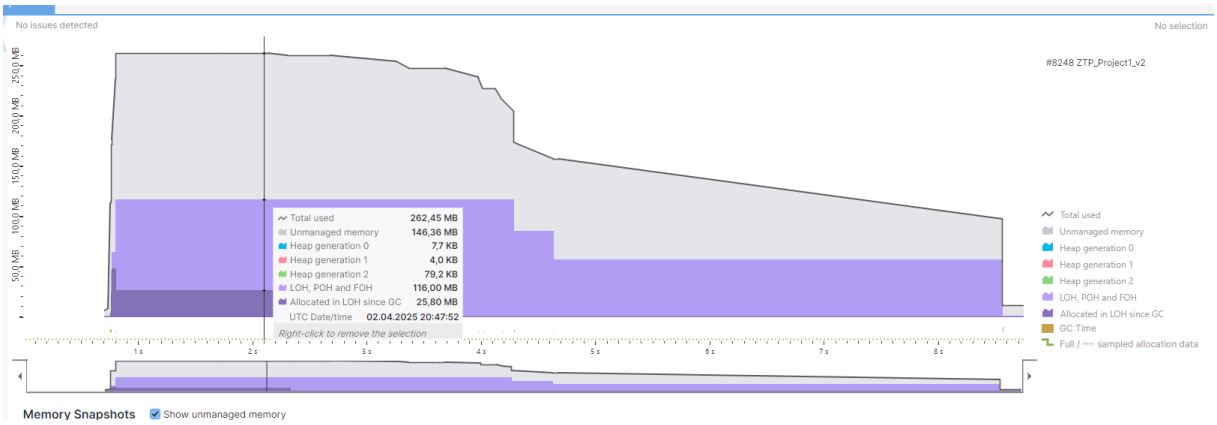
Włączenie USE_FIXED=true pozwala na przypięcie danych w pamięci, zapobiegając ich przenoszeniu przez Garbage Collector podczas operacji na surowych wskaźnikach.

Pomiar Unmanaged



Czas: 904ms

Pomiar Managed



Czas: 7845ms

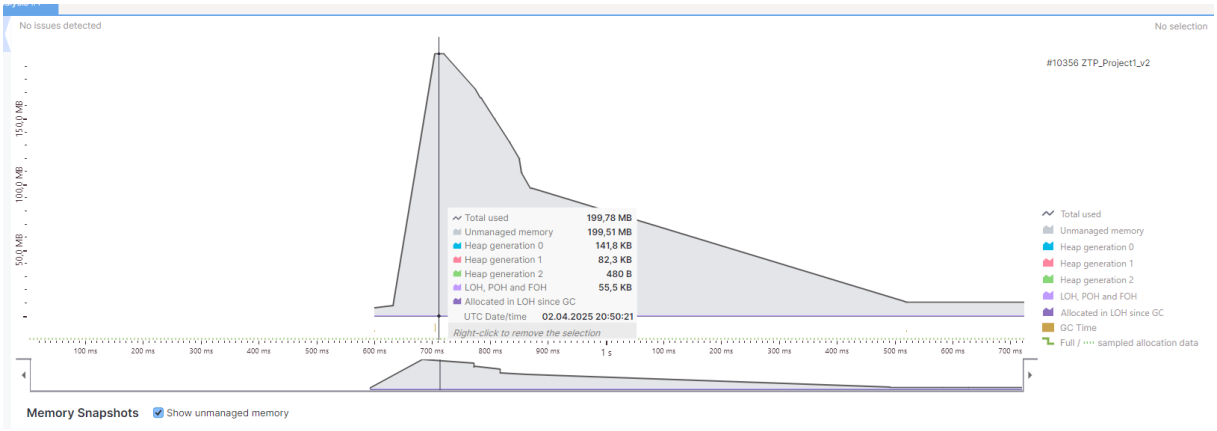
Implementacja unmanaged fixed wykazała najlepsze wyniki wydajnościowe spośród wszystkich badanych wariantów, osiągając czas przetwarzania 904ms, co stanowi poprawę o około 30% w porównaniu do podstawowej wersji niezarządzanej. Ta technika łączy w sobie zalety bezpośredniego dostępu do pamięci poprzez wskaźniki z dodatkowymi optymalizacjami, takimi jak:

- Pinowanie macierzy filtru w pamięci, co eliminuje konieczność jej ponownego ładowania podczas przetwarzania kolejnych pikseli
- Wyeliminowanie redundantnych obliczeń indeksów poprzez zastosowanie przesunięć wskaźnikowych zamiast wielokrotnego mnożenia

Wykorzystanie puli pamięci

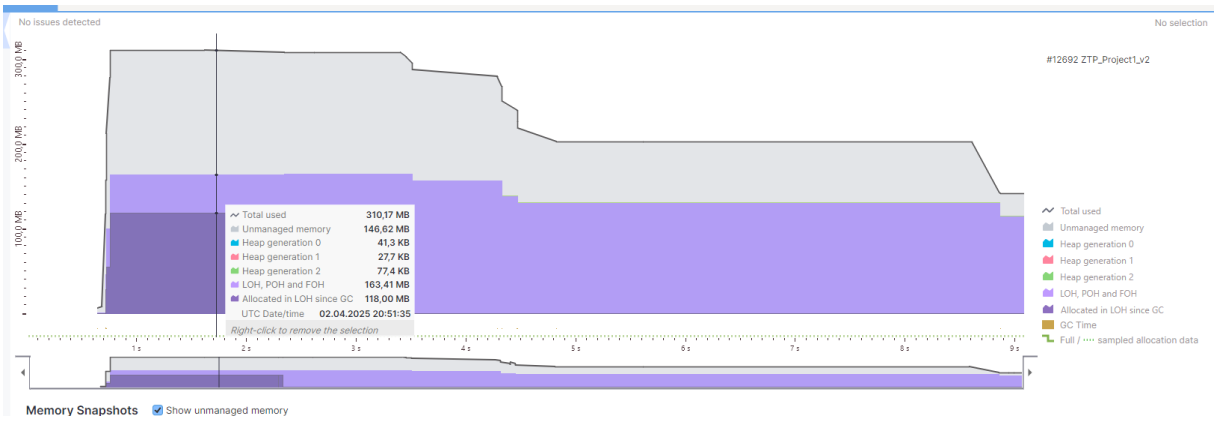
Włączenie USE_POOLING=true pozwala na wielokrotne użycie tych samych buforów pamięci, minimalizując alokacje i fragmentację sterty. Technika ta znacząco redukuje obciążenie Garbage Collectora, szczególnie podczas przetwarzania wielu obrazów, co przekłada się na lepszą wydajność i zmniejszone zużycie pamięci.

Pomiar Unmanaged



Czas: 903ms

Pomiar Managed

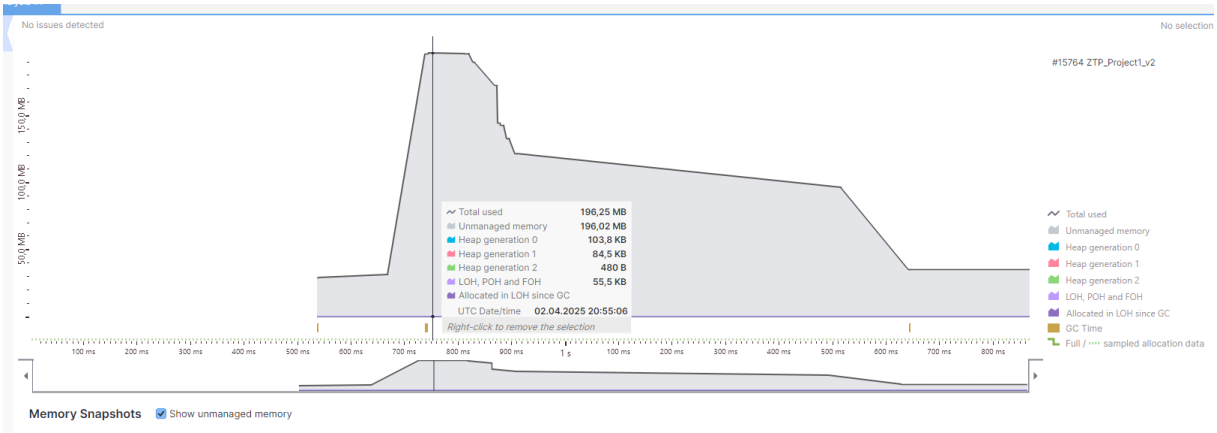


Czas: 8201ms

Włączenie trybu serwerowego

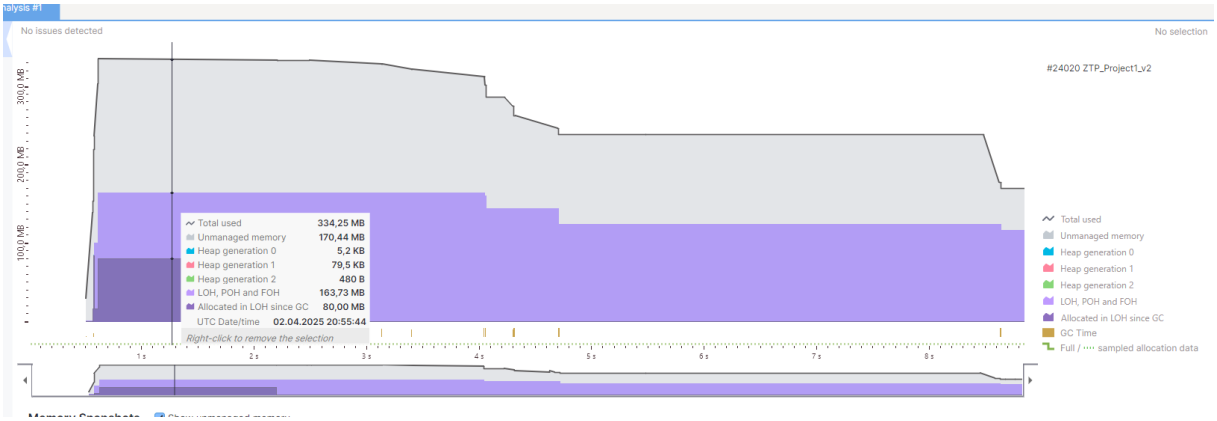
Włączenie trybu serwerowego Garbage Collectora umożliwia równoległe wykonywanie operacji odśmiecania na wielu procesorach, co jest szczególnie korzystne dla aplikacji intensywnie przetwarzających obrazy. Ten tryb optymalizuje wydajność dla programów wymagających dużych zasobów, takich jak nasze przetwarzanie obrazów, zmniejszając czas wstrzymania wątków aplikacji podczas zbierania nieużywanych obiektów.

Pomiar Unmanaged



Czas: 992ms

Pomiar Managed

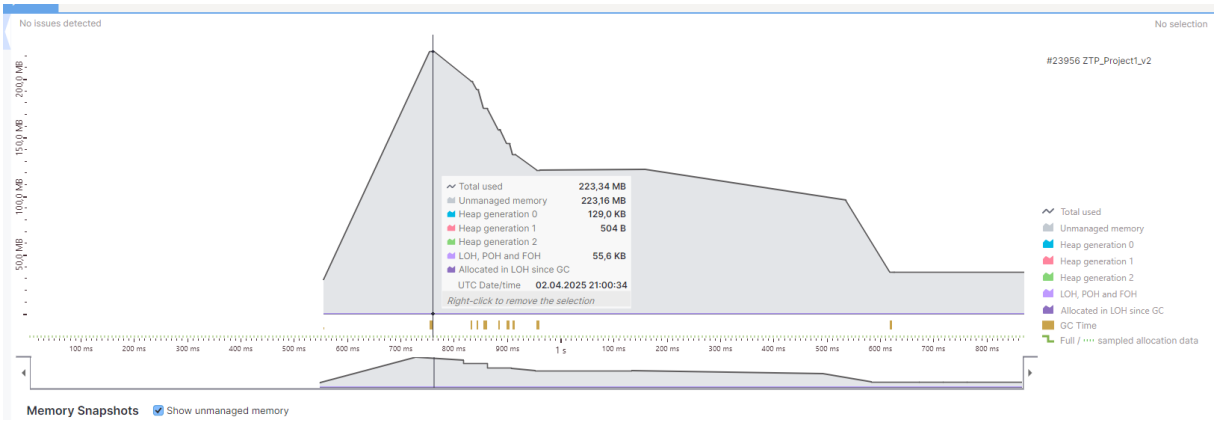


Czas: 8109ms

GC Low Latency

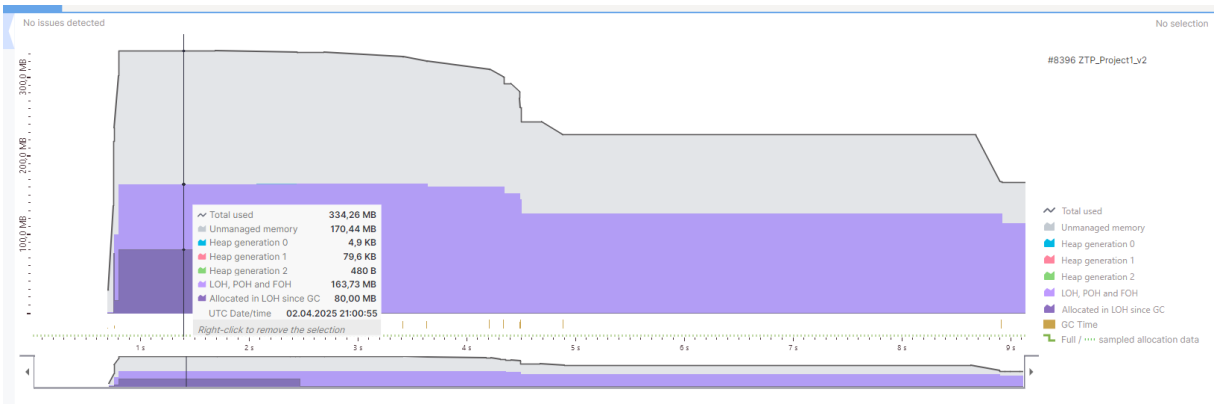
Tryb GC Low Latency tymczasowo zmniejsza częstotliwość i agresywność procesu odśmiecania pamięci, priorytetyzując szybkość wykonania kodu nad natychmiastowym odzyskiwaniem pamięci. To ustawienie jest szczególnie przydatne podczas intensywnego przetwarzania obrazów, ponieważ minimalizuje przerwy w pracy aplikacji związane z działaniem Garbage Collectora, choć może prowadzić do wyższego chwilowego zużycia pamięci.

Pomiar Unmanaged



Czas: 962ms

Pomiar Managed

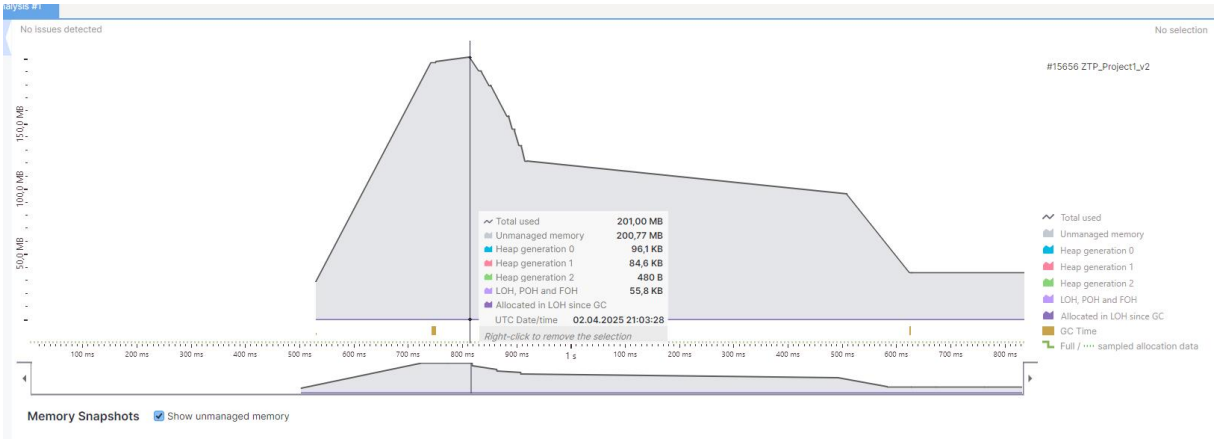


Czas: 8202ms

Sustained Low Latency

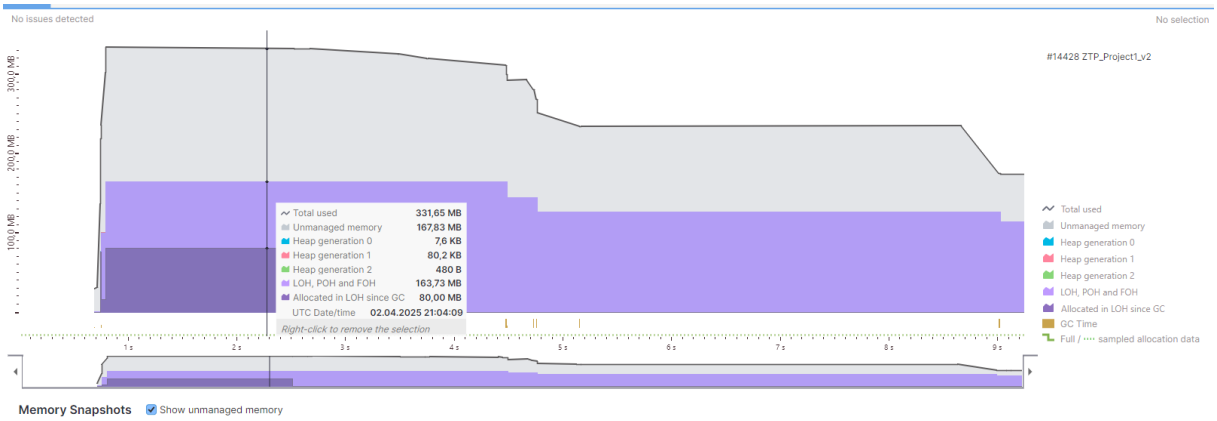
Tryb GC Sustained Low Latency zapewnia długotrwałe, stabilne niskie opóźnienia odpowiedzi aplikacji poprzez zminimalizowanie kompletnych kolekcji GC, co jest szczególnie istotne przy ciągłym przetwarzaniu sekwencji obrazów.

Pomiar Unmanaged



Czas: 970ms

Pomiar Managed



Czas: 8319ms

Wnioski

Na podstawie przeprowadzonych badań wydajnościowych implementacji filtru Laplace'a dla obrazów w środowisku .NET można sformułować następujące wnioski:

1. Pamięć zarządzana vs niezarządzana (VERSION_MANAGED):
   Implementacje używające niezarządzanej pamięci (unmanaged) poprzez bezpośredni dostęp do buforów pikseli konsekwentnie wykazywały wyższą wydajność w porównaniu do wersji zarządzanych. Wynika to z eliminacji narzutu związanego z metodami GetPixel i SetPixel, które wykonują dodatkowe operacje zabezpieczające i konwersje typów.

2. Użycie bloku fixed (USE_FIXED):
   Pinowanie pamięci za pomocą słowa kluczowego fixed zapewniło znaczący wzrost wydajności, szczególnie w przypadku implementacji zarządzanej, gdzie redukowało liczbę potencjalnych przeniesień obiektów przez Garbage Collector. W implementacji niezarządzanej dodatkowe pinowanie kernela filtru również przynosiło korzyści wydajnościowe, choć mniej znaczące.

3. Przetwarzanie równoległe (ENABLE_PARALLEL):
   Wykorzystanie Parallel.ForEach dla przetwarzania wielu obrazów przyniosło znaczące przyspieszenie, szczególnie na systemach wielordzeniowych. Korzyść była tym większa, im więcej obrazów przetwarzano jednocześnie, choć przy bardzo dużej liczbie obrazów mogło dochodzić do konkurencji o zasoby.

4. Zarządzanie odśmiecaniem pamięci (GC_COLLECT):
   Wymuszenie GC.Collect() po każdym przetworzonym obrazie miało dwojaki efekt: z jednej strony zwiększało przewidywalność użycia pamięci i zmniejszało jej maksymalne zużycie, z drugiej jednak wprowadzało dodatkowe opóźnienia. W przypadku przetwarzania wielu dużych obrazów, korzyści z kontrolowanego odśmiecania przeważały nad kosztami.

5. Kompaktowanie dużego stosu obiektów (COMPACT_ONCE):
   Ustawienie GCLargeObjectHeapCompactionMode.CompactOnce pozwalało na zmniejszenie fragmentacji pamięci, co było szczególnie istotne przy przetwarzaniu wielu dużych obrazów. Jednakże, samo kompaktowanie wprowadzało krótkotrwałe, ale zauważalne opóźnienie w działaniu programu.

6. Zwalnianie zasobów (DISPOSE):
   Jawne wywoływanie metody Dispose() na obiektach Bitmap znacząco poprawiało zarządzanie zasobami, szczególnie w scenariuszach przetwarzania dużej liczby

obrazów. Zapobiegało to wyciekowi zasobów systemowych, które nie są automatycznie zarządzane przez Garbage Collector.

7. Wykorzystanie puli pamięci (USE_POOLING):
   Zastosowanie ArrayPool<T> do recyklingu buforów pamięci znacząco redukowało presję na Garbage Collector i fragmentację sterty. Było to szczególnie efektywne w przypadku przetwarzania sekwencyjnego wielu obrazów, gdzie te same bufory mogły być wielokrotnie wykorzystywane.

8. Tryb serwerowy GC:
   Włączenie trybu serwerowego Garbage Collectora poprawiało wydajność na systemach wieloprocesorowych poprzez równoległe wykonywanie operacji odśmiecania. Korzyść była szczególnie widoczna przy włączonym przetwarzaniu równoległym obrazów.

9. Tryb niskiego opóźnienia GC (LOW_LATENCY):
   Tymczasowe ustawienie trybu GCLatencyMode.LowLatency podczas intensywnego przetwarzania obrazów redukowało przerwy spowodowane działaniem Garbage Collectora, co przekładało się na płynniejsze działanie aplikacji, choć kosztem potencjalnie wyższego zużycia pamięci.

10. Tryb długotrwałego niskiego opóźnienia (SUSTAINED_LOW_LATENCY):
    Zastosowanie GCLatencyMode.SustainedLowLatency zapewniało dobry kompromis między wydajnością a zarządzaniem pamięcią podczas długotrwałego przetwarzania sekwencji obrazów, minimalizując częstotliwość pełnych kolekcji bez nadmiernego zwiększania zużycia pamięci.

Optymalna konfiguracja dla większości przypadków testowych obejmowała użycie trybu niezarządzanego z pinowaniem kernela, włączonym przetwarzaniem równoległym, pulą pamięci oraz trybem serwerowym GC z ustawieniem SustainedLowLatency. Takie podejście zapewniało najlepszy stosunek wydajności do zużycia zasobów systemowych przy przetwarzaniu sekwencji dużych obrazów.