A Study of Java`s Non-Java Memory

Remigiusz Mytyk

Agenda

1. Wprowadzenie
2. Struktura pamięci
3. MARUSA
4. Micro-benchmarki
5. Macro-benchmarki
6. Modyfikacja JVM
7. Wnioski

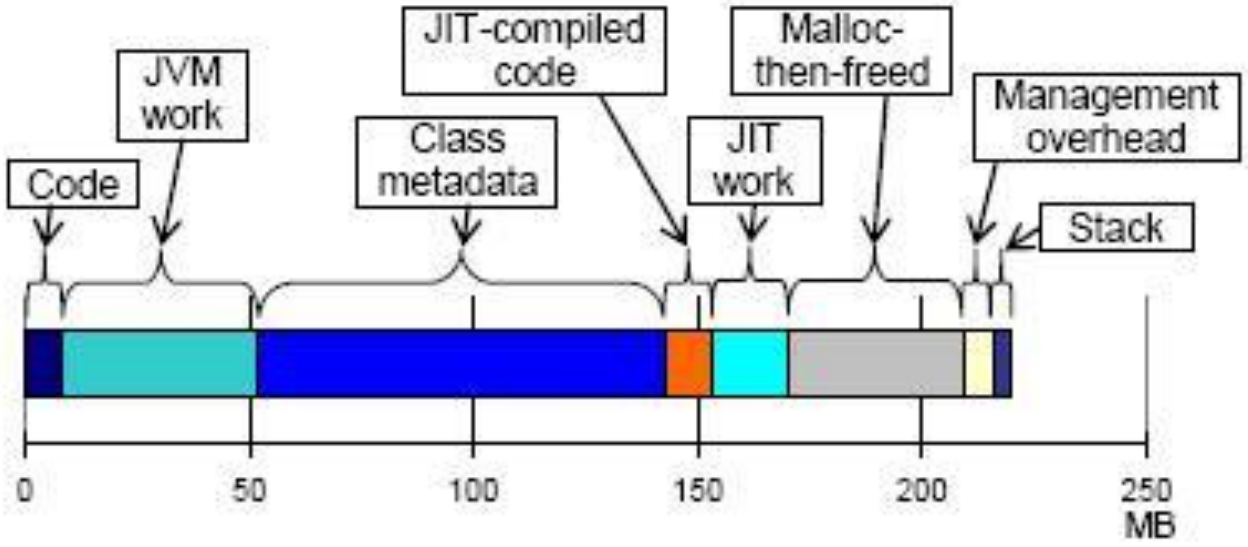
Wprowadzenie

- Wiele osób spotkało się zapewne w Javie z wyjątkiem out-of-memory
- Z reguły powodem jest przepełnienie sterty, na której znajdują się obiekty tworzone podczas wykonywania programu
- Ale przyczyną może być również wyczerpanie pamięci określanej dalej jako NIEJAVOWA(NJ)
- Na przykład w skutek załadowanie zbyt dużej liczby klas

Przeznaczenie pamięci NJ

- Biblioteki dzielone
- Metadane załadowanych klas
- Skompilowany kod JIT
- Optymalizacja wywołań refleksyjnych
- Przechowywanie direct byte buffer

Budowa pamięci NJ



•Rozkład pamięci NJ po 9minutach działania aplikacji J2EE Apache DayTrayder na WebSphere Application Server

•Rozmiar tej pamięci wynosi 210 MB prawie tyle samo co domyślna wartość sterty

•Nie wszyscy programiści są świadomi takiego stanu rzeczy =)

Podział pamięci NJ

1. Obszar kodu
   - Kod natywny załadowany z wykonywalnych plików i bibliotek
   - Dane z załadowane z bibliotek dzielonych

2. Obszar roboczy JVM
   - Dane używane na potrzebę JVM
   - Pamięć zaalokowana przez standardową bibliotekę Javy
   - Metody JNI użytkowników

Podział pamięci NJ

3. Obszar metadanych klas
   - Dane załadowane z plików klas
   - Bytecode, znaki UTF-8, pule stałych, tablice metod

4. Obszar skompilowanego kodu JIT
   - Kod natywny wygenerowany przez kompilator JIT
   - Dane do tego kodu

Podział pamięci NJ

5. Obszar roboczy kompilatora JIT
   - Pośrednia reprezentacja kompilowanych metod

6. Obszar malloc-then-freed
   - Pamięć zajęta poprzez wywołanie malloc()
   - Następnie zwolniona używając free()
   - Zarządzanie tą pamięcią zależy od libc

Podział pamięci NJ

7. Obszar narzutu systemowego
   - Pamięć wykorzystywana przez OS do zarządzania pamięcią procesu
   - Nagłówek malloca
   - Nieużywane części stron pamięci

8. Obszar stosu
   - Stos Javy i C
   - Ramki stosu Javy w zależności od implementacji JVM

MARUSA

- W celu zliczania pamięci należącej do poszczególnych obszarów autorzy stworzyli narzędzie MARUSA - Memory Analyzer for Redundant,Unused, and String Areas
- Zbiera statystyki zarówno z OS jak i JVM
- Wizualizuje wyniki podzielone na wymienione wcześniej kategorie

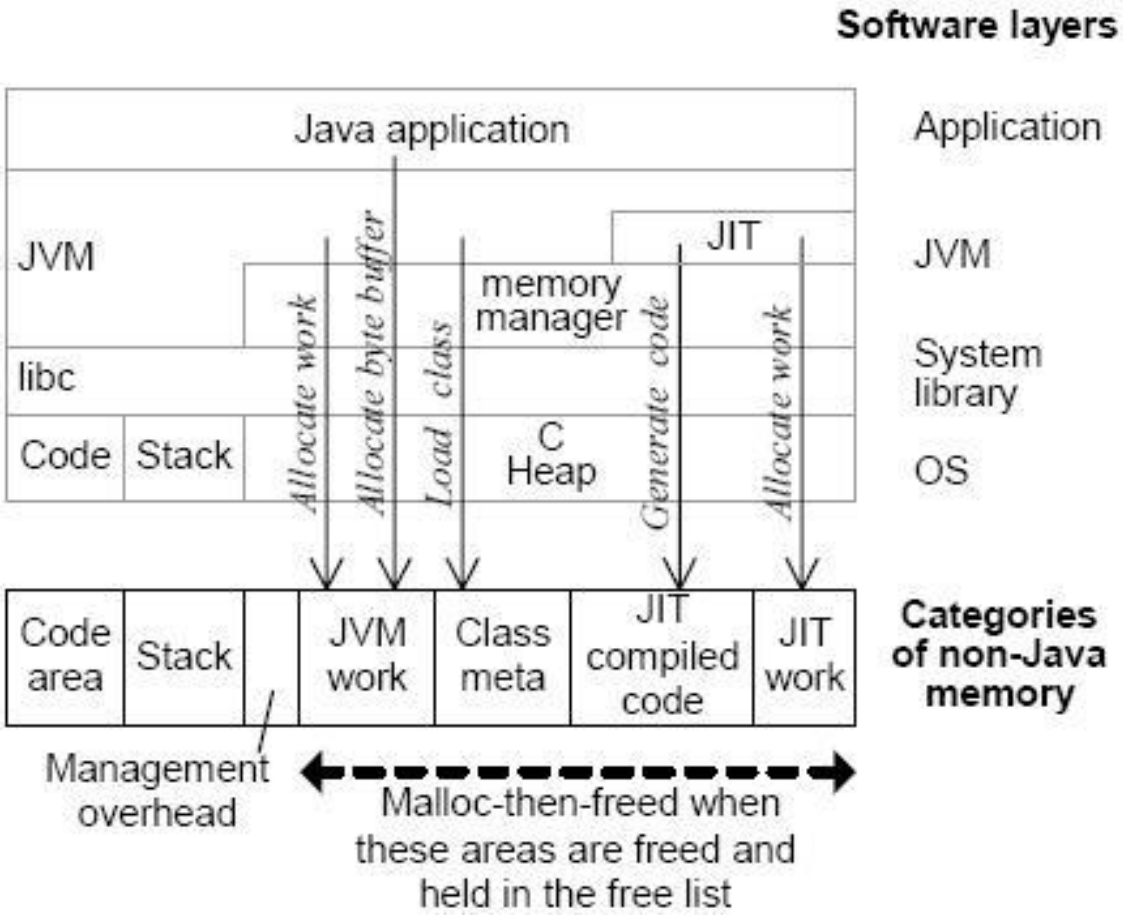
Metodologia mierzenia pamięci NJ

- Autorom zależało na pełnej analizie pamięci przydzielonej do procesu JVM

Zbieranie informacji na temat obszarów pamięci:

1. Informacje z poziomu OS – zakresy pamięci i ich atrybuty
2. Informacje poziomu JVM – jakie komponenty alokowały pamięć
3. Powiązanie ze sobą informacji z punktu 1 i 2

- Duże, nowoczesne aplikacje posiadają własne wewnętrzne managery pamięci



Informacje poziomu OS

- Rozmiar i atrybuty bloków pamięci należących do JVM
- MARUSA używa maps z filesystemu /proc do uzyskania zakresów pamięci i ich atrybutów
- Stan fizycznych stron za pomocą pageinfo również z /proc

Informacje poziomu JVM

- Debugowanie – rozmiar metadanych klas, skompilowanego kodu JIT
- Modyfikacja JVM – dokładne informacje o alokowanej i zwalnianej pamięci, żądania do wewnętrznego managera pamięci

Zliczanie pamięci NJ

- MARUSA posiada mapę w której są informacje na temat każdego byte`u przydzielonego w aplikacji
- Na podstawie atrybutów program zlicza pamięć należącą do poszczególnych obszarów

Micro-benchmarki

- Zależności pomiędzy rozmiarem pamięci NJ a operacjami wykonywanymi w programach napisanych w Javie.

- Benchmarki te analizują wykorzystanie:
  - obszaru metadanych klas
  - obszaru roboczego JVM i obszaru malloc-then-freed

Hardware environment	
Machine	IBM BladeCenter LS21
CPU	Dual-core Opteron (2.4 GHz), 2 sockets
RAM size	4 GB
Software environment	
OS	SUSE Linux Enterprise Server 10.0
Kernel version	2.6.16
JVM	IBM J9 Java VM for Java 6 (SR7), 32bit

Table 2. Execution environment for x86.

Hardware environment	
Machine	IBM BladeCenter JS21
CPU	Dual-core PowerPC 970MP (2.5 GHz), 2 sockets
RAM size	8 GB
CPU and memory allocated to the tested virtual machine	
CPU	2 CPUs
Memory	2 GB
Software environment	
OS	RedHat Enterprise Linux 5.4
Kernel version	2.6.18
JVM	IBM J9 Java VM for Java 6 (SR7), 32bit

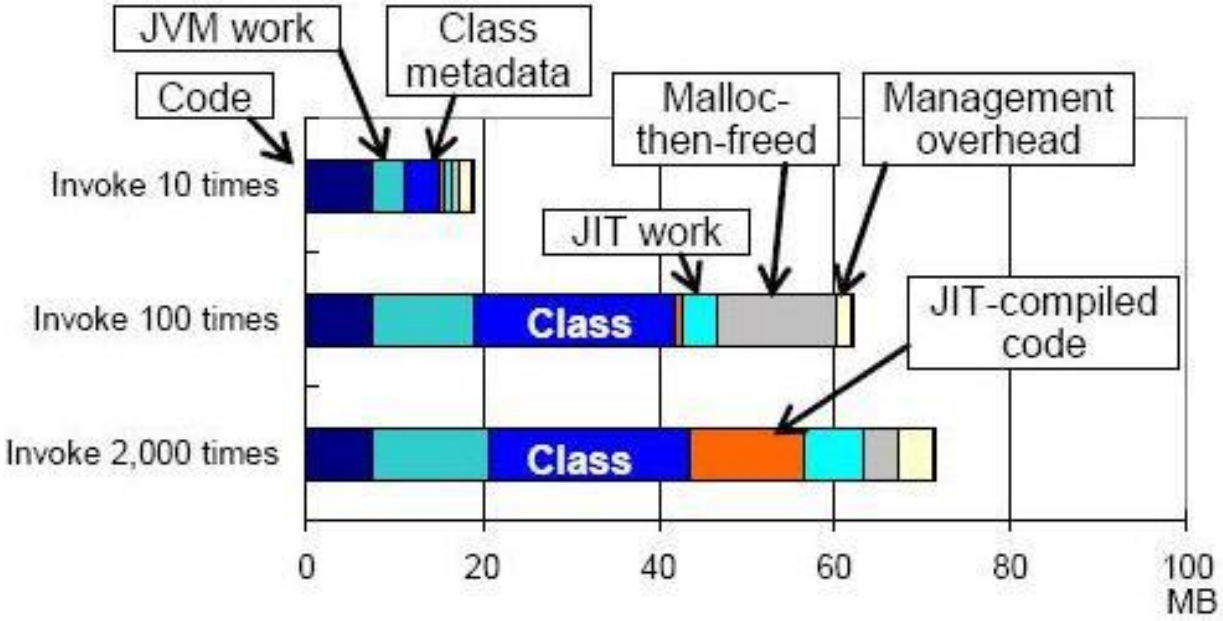
Table 3. Execution environment for POWER.

Środowiska

Micro-benchmark dla obszaru metadanych klas

- Wpływ mechanizmu refleksji na rozmiar pamięci NJ
- Wywołamy gettery i settery dla 6000 pól za pomocą obiektów klasy java.lang.reflect.Method
- Zmierzymy rozmiar wykorzystanej pamięci wywołując te metody 10, 100 i 2000 razy

Wyniki dla środowiska x86

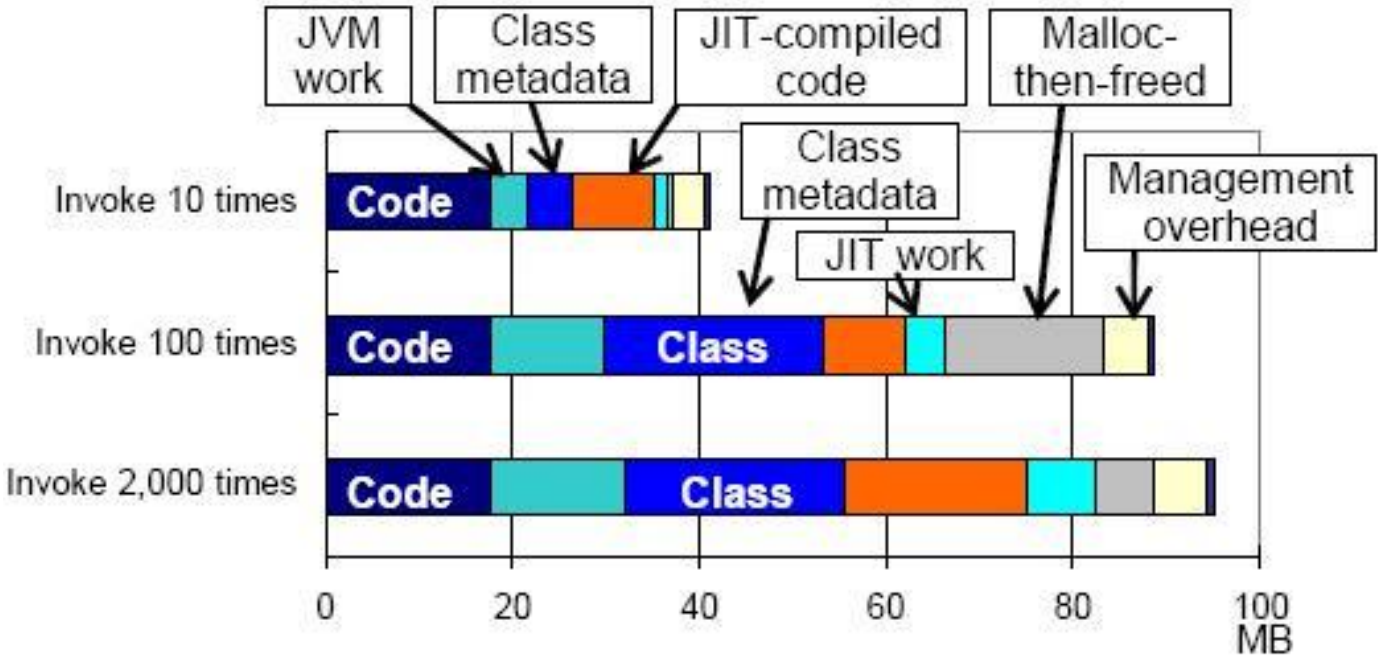


Metadane klas:
- 3.9 MB dla 10
- 21.8 MB dla 100 i 2000

Skompilowany kod JIT
- 0.8 MB dla 10 i 100
  12.3 MB dla 2000

- Optymalizacja wywołań refleksyjnych
- Optymalizacja przez kompilator JIT
- Wzrost kosztów przy 2000 wywołaniach o 29.2 MB i 43% rozmiaru całej wykorzystanej pamięci

Wyniki dla środowiska POWER

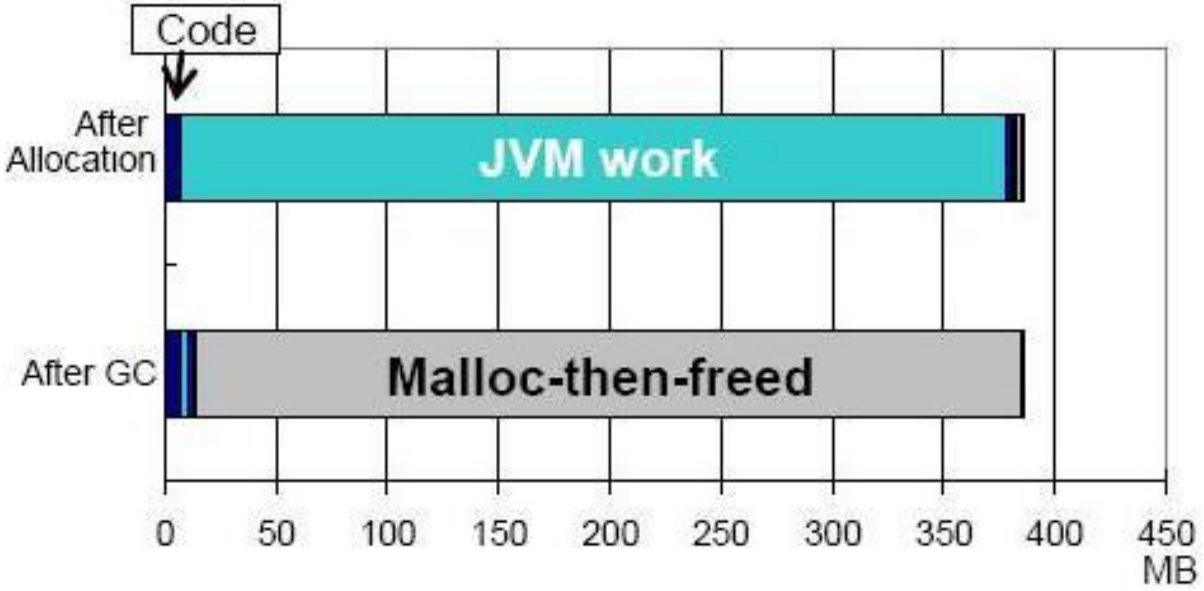


• Domyślnym rozmiarem strony w RedHat Enterprise Linux 5 for POWER wynosi 64 KB
  - Zwiększa procent trafień w TLB
  - Zwiększa również wykorzystanie pamięci – wewnętrzna fragmentacja
• Różnica w implementacji przydziału pamięci dla obszaru skompilowanego kodu JIT. (Przydzielane są większe bloki pamięci)

Micro-benchmark dla obszaru roboczego JVM i obszaru malloc-then-freed

- Zachowanie pamięci NJ podczas alokacji i zwalniania *direct byte buffers*
- *Alokujemy 10000 buforów , każdy po 32 KB, przy stercie Javy rozmiaru 8MB*
- *Następnie po zwolnieniu uruchamiamy garbage collector wywołując System.gc()*

Wyniki dla środowiska x86

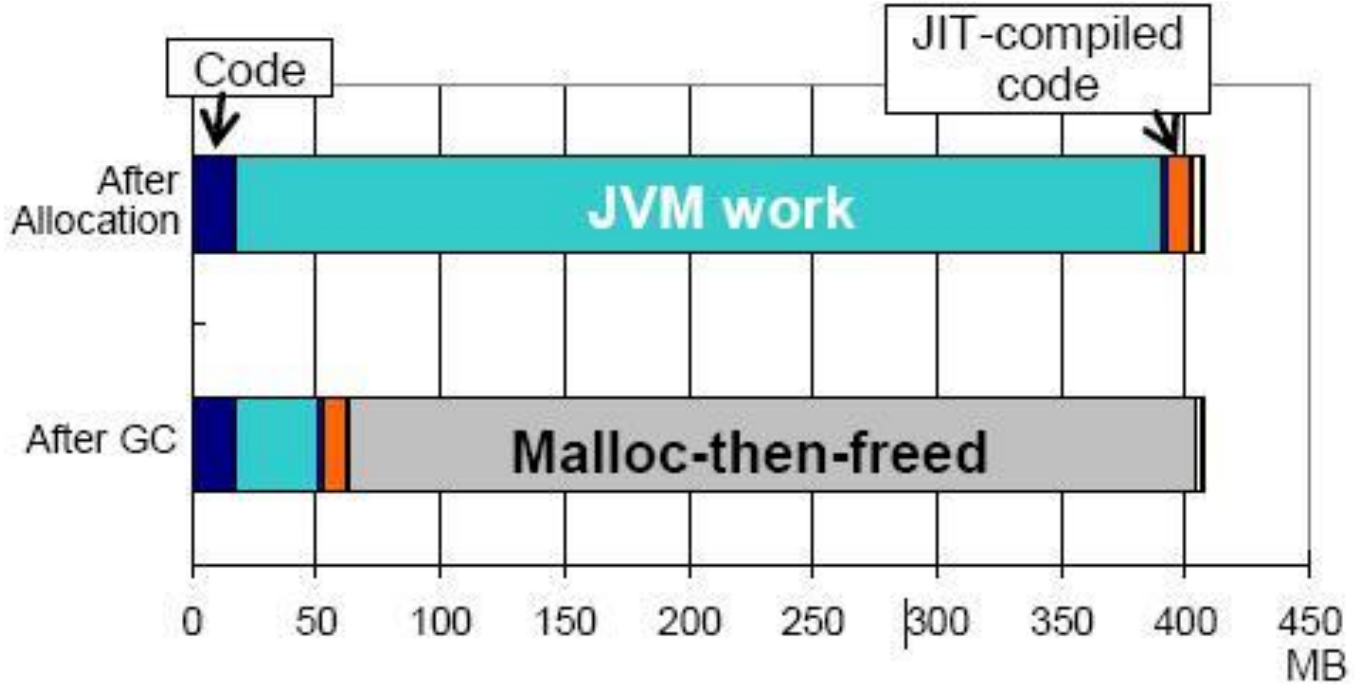


Sterta javy: 8 MB
Obszar roboczy JVM: 354.3MB z czego 351.7 MB to pamięć buforów

Po uruchomieniu GC:
Obszar roboczy JVM: -352 MB
Obszar malloc-then-freed: +350 MB

- Duże zużycie pamięci niewidoczne dla programów i narzędzi do debugowania
- Zużycie pamięci po wywołaniu garbage collectora nie zmieniło się

Wyniki dla środowiska POWER

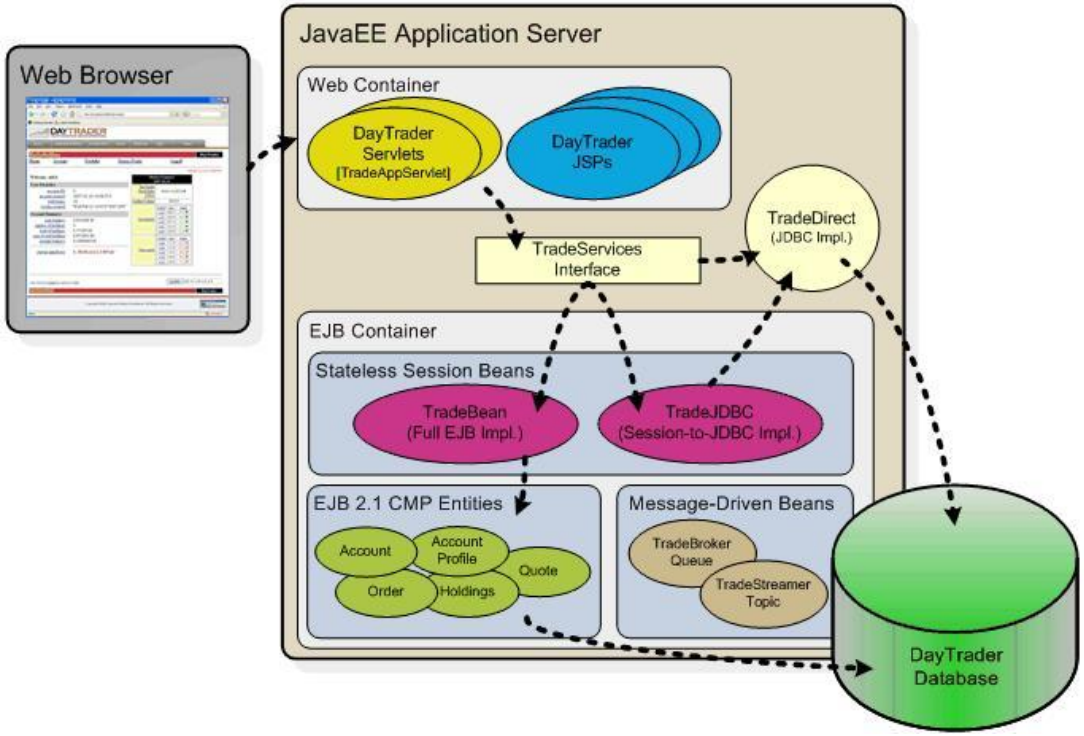


•Na stercie pozostało około 900 obiektów direct byte buffer pomimo odpalenia garbage collectora

Macro-benchmarki

- Sprawdzimy jak zachowuje się pamięć NJ podczas pracy większych aplikacji.
- Przetestujemy Apache DayTrader na WebSphere Application Server
- Benchmarki DaCapo
- Przetestujemy je na takich samych środowiskach jak micro-benchmarki

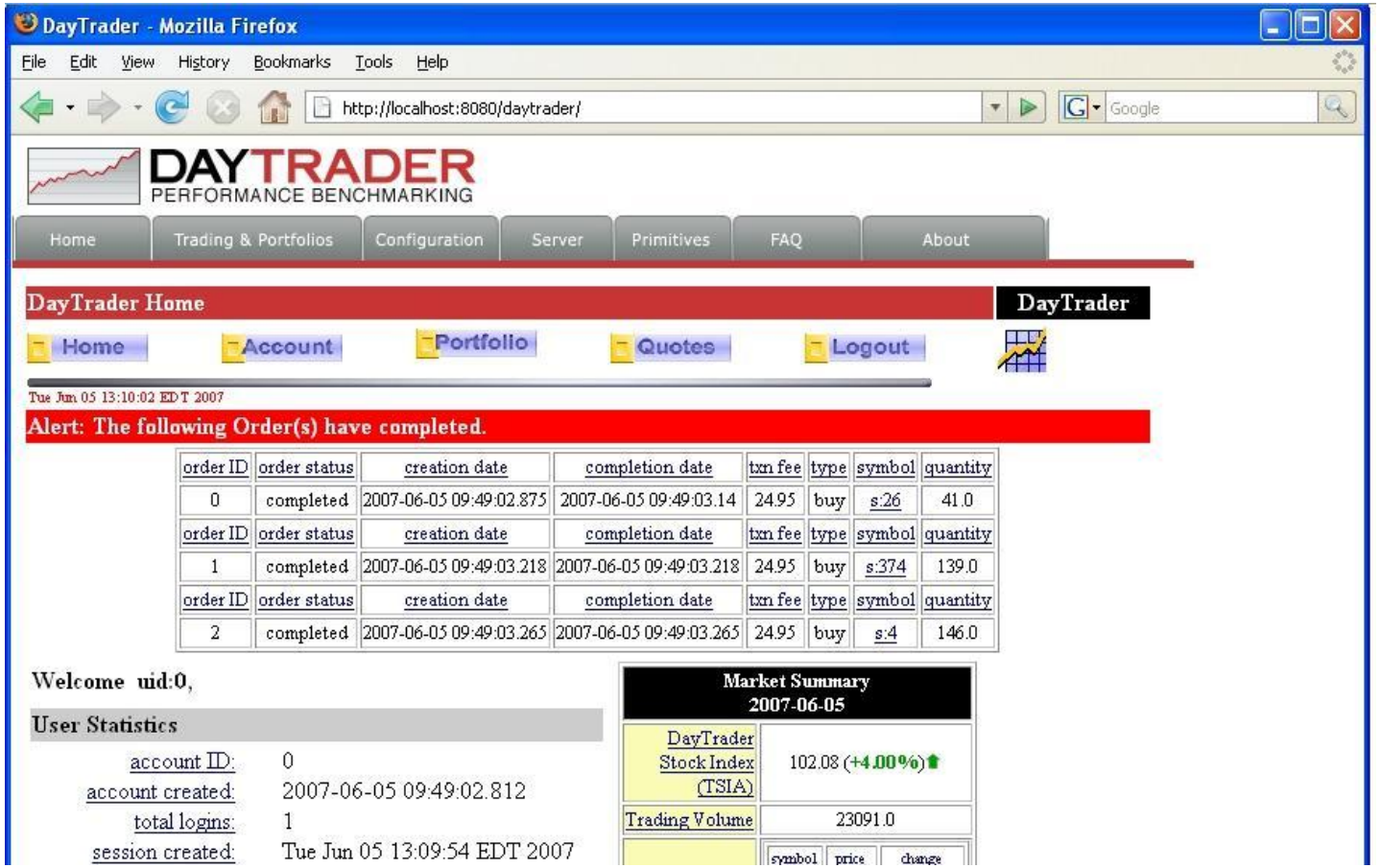
Apache DayTrader



Złożona aplikacja J2EE wykorzystująca szereg technologii:

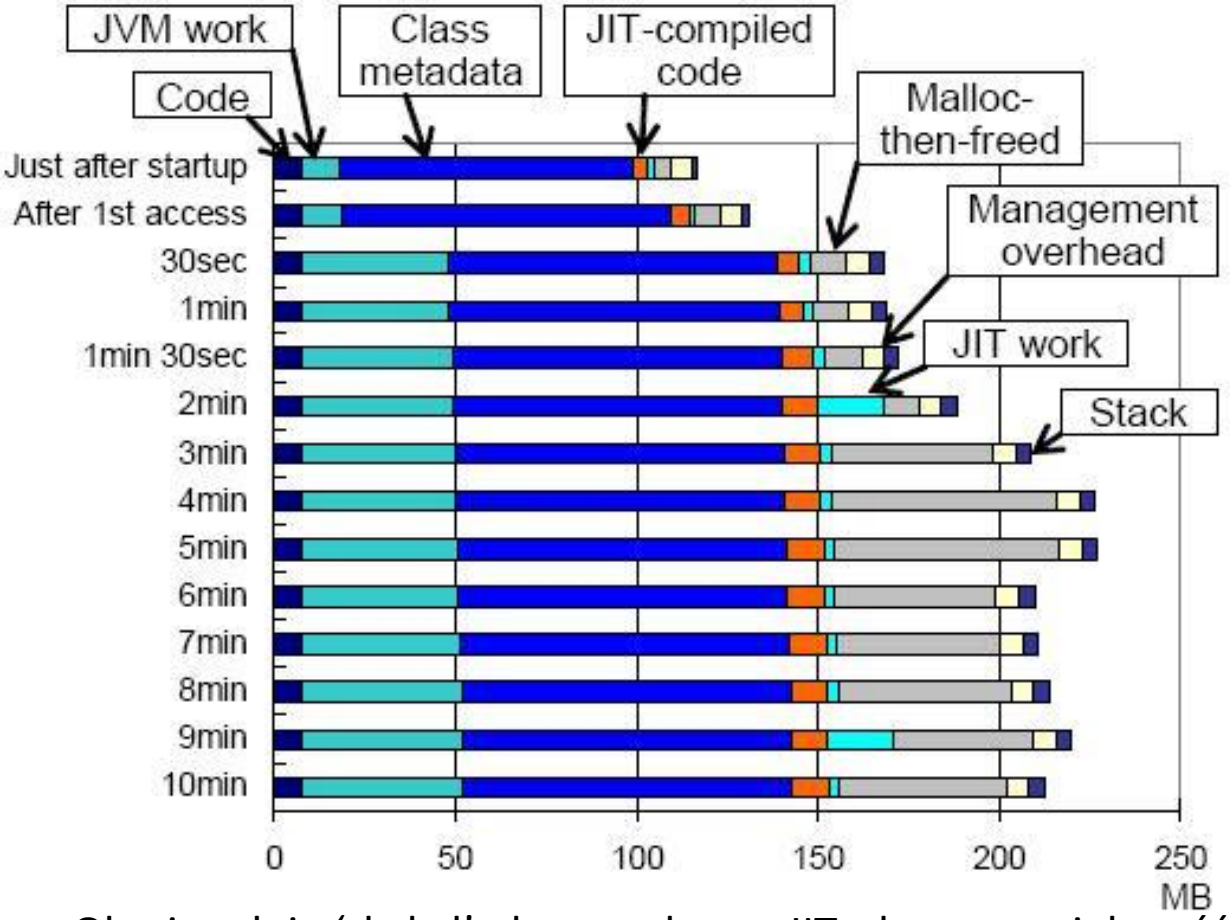
- Warstwa prezentacji - Java Servlets and JavaServer Pages (JSPs)
- Logika biznesowa i warstwa persystencji - Java database connectivity (JDBC), Java Message Service (JMS), Enterprise JavaBeans (EJBs) i Message-Driven Beans (MDBs)

Apache DayTrader



- My podczas 10 minutowego testu generowaliśmy ruch w aplikacji za pomocą 30 wątków
- Maksymalny rozmiar sterty był ustawiony na 256 MB

Apache DayTrader na x86

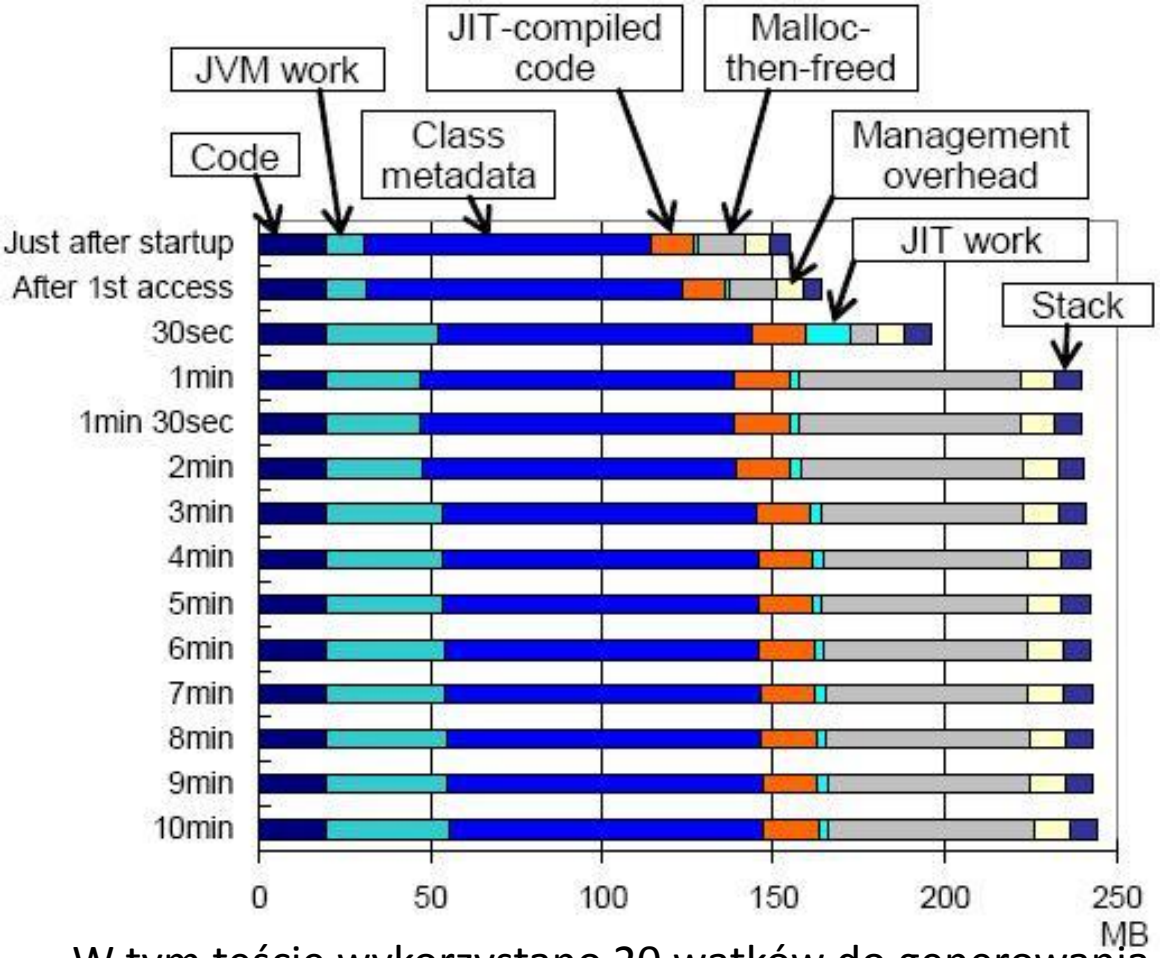


Trzy największe obszary:

- Obszar metadanych klas – największy ze wszystkich tuż po uruchomieniu
- Obszar roboczy JVM – wzrósł po 30sec za sprawą Direct Byte Buffers
- Następnie urósł obszar malloc-then-freed

Okazjonalnie ‘skakał’ obszar roboczy JIT, ale przez większość czasu zajmował znikomą pamięć

Apache DayTrader na POWER



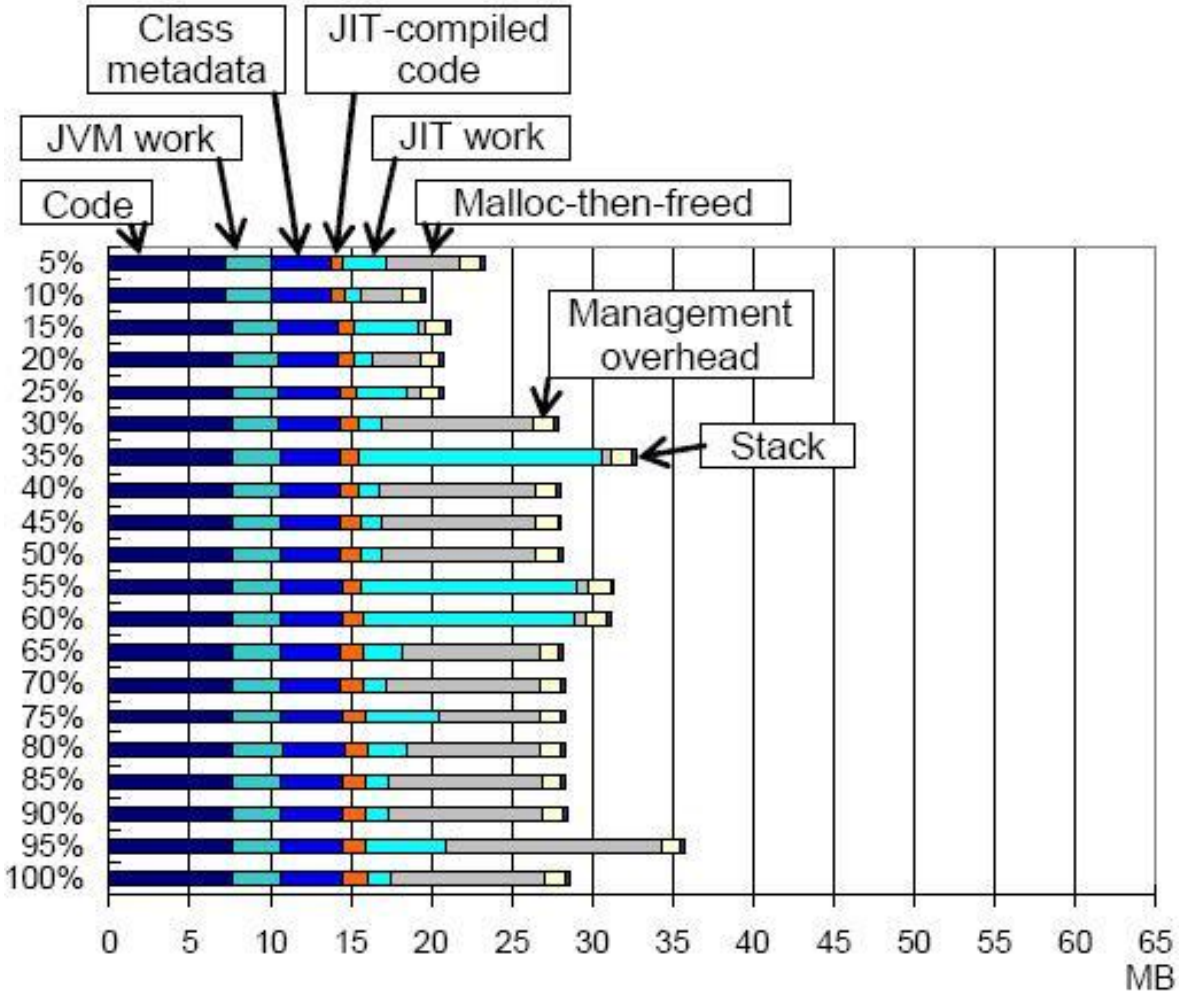
- Skoki w obszarze roboczym JVM
- Wzrost przez Direct Byte Buffer po 30sec i po 2min
- Tymczasowe struktury – alokacja przy 30sec i zwolnienie po 1min
- Stos większy o 4 MB – większy rozmiar strony
- 155 wątków dla WAS

W tym teście wykorzystano 20 wątków do generowania ruchu, ponieważ maszyna dla platformy POWER była wolniejsza od tej dla x86. Natomiast wykorzystanie CPU w obu przypadkach zostało utrzymane na poziomie 90%.

DaCapo Benchmarks

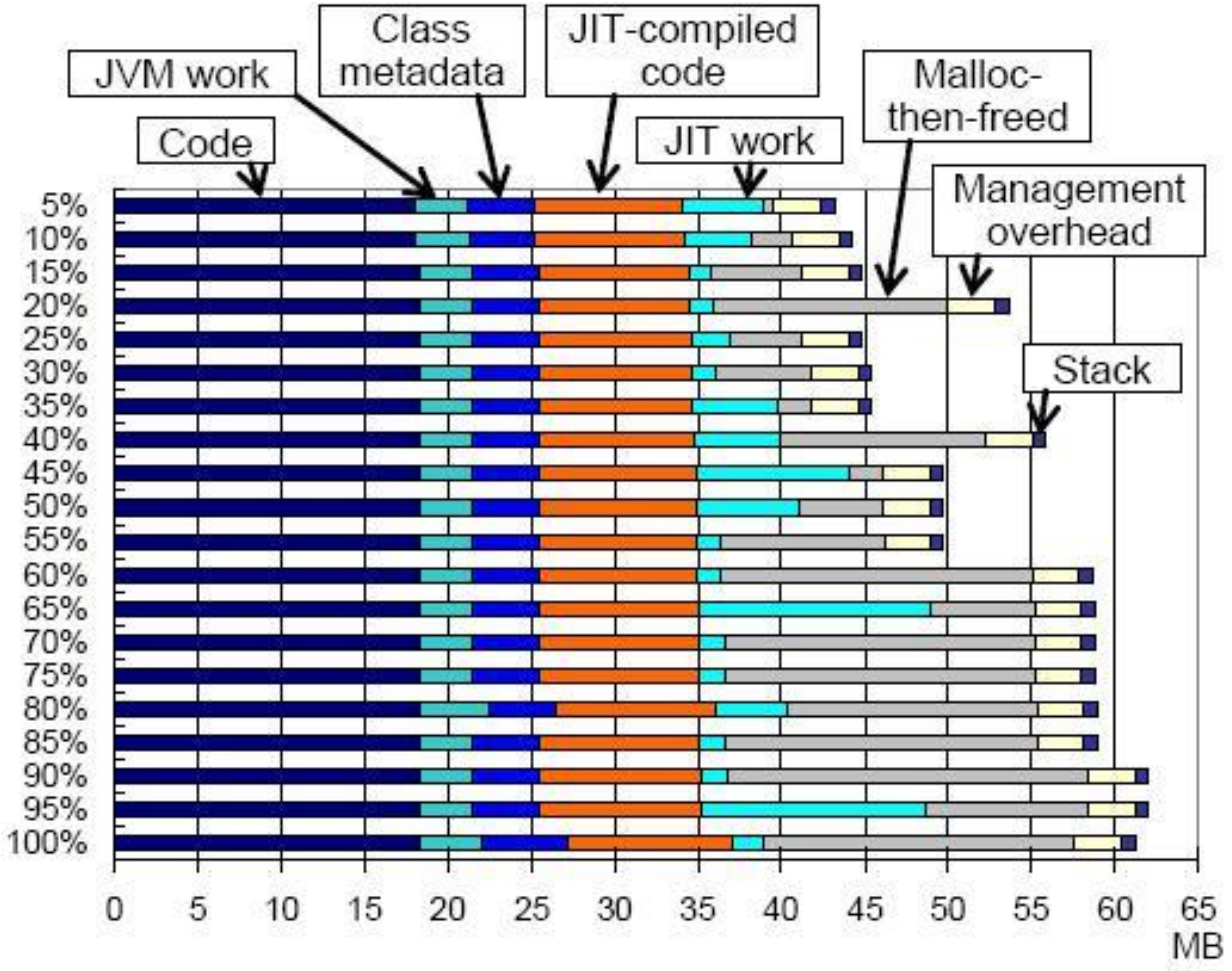
- Zestaw użytkowych, opensource`owych aplikacji, które umożliwiają testowanie pamięci poprzez jej różnorodne i zaawansowane metody obciążania
- Wyniki dla benchmarku *bloat,* ponieważ dla innych benchmarków zaobserwowano podobne zachowanie pamięci NJ
- Benchmark ten alokuje 990 MB danych bytecodu
- Sterta javy ustawiona na 13 MB

DaCapo Benchmark dla x86



- Zależność pomiędzy rozmiarem obszaru roboczego JIT i malloc-then-freed
- Rozmiar malloc-then-freed zależy od:
- Alokacji i zwalniania pamięci przez kompilator JIT
- Algorytmu użytego do zarządzania listą wolnej pamięci w *libc*

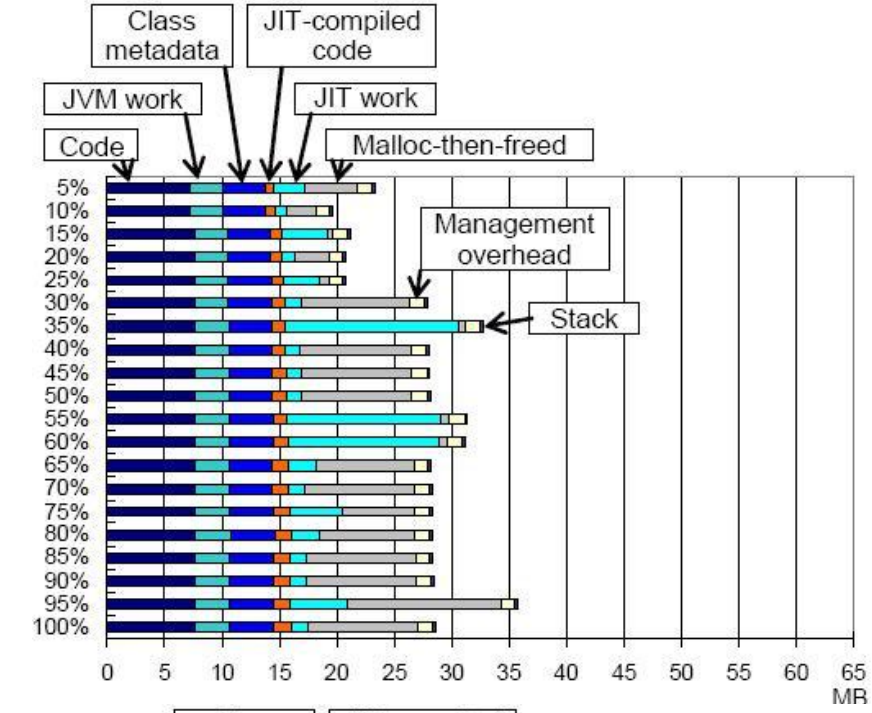
Benchmark DaCapo dla POWER

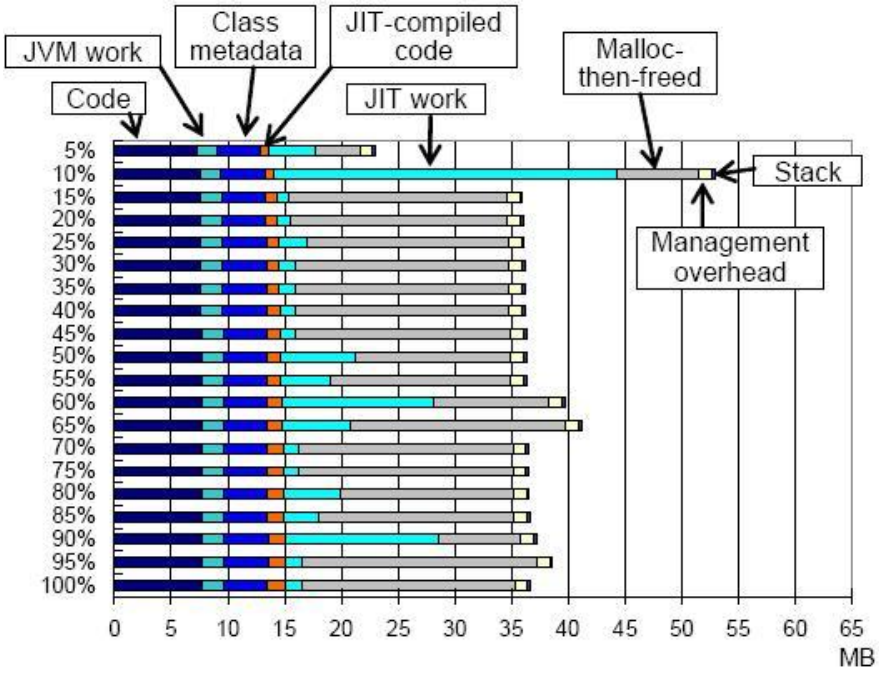


- Bardziej agresywna optymalizacja na platformie POWER skutkuje zwiększeniem obszaru malloc-then-freed
- Obszar kodu jak i skompilowanego kodu JIT większe z przyczyn omawianych wcześniej

Redukcja obszaru malloc-then-freed

- Znaczny rozmiar pamięci zajmowany przez ten obszar
- Trudność z wyznaczeniem prawdziwego rozmiaru pamięci zajmowanego przez program
- Rozmiar tego obszaru podlega również trudnym do przewidzenia skokom





Dwa wywołania tego samego benchmarka DaCapo

•Znaczne różnice w rozmiarach obszaru malloc-then-freed

Redukcja obszaru malloc-then-freed

- Obszar malloc-then-freed trzymany na liście wolnej pamięci zarządzanej przez libc
- Brak w API możliwości przekazania czy zwalniany blok pamięci powinien być przechowywany do ponownego użycia, czy powinien być zwrócony do OS
- Utrudnia efektywne zarządzanie pamięcią pomiędzy aplikacją i libc

madvise() system call

- Możemy zmniejszyć obszar zajmowany przez malloc-then-freed przez bezpośrednie wysłanie prośby do OS, aby usunął fizyczne strony z pamięci procesu.
- W Linuxie możemy skorzystać z systemowego wywołania madvise().
- Chociaż obszary znajdujące się na liście wolnej pamięci dalej będą zajmować przestrzeń adresową, zwolniona zostanie fizyczna pamięć.

madvise() system call

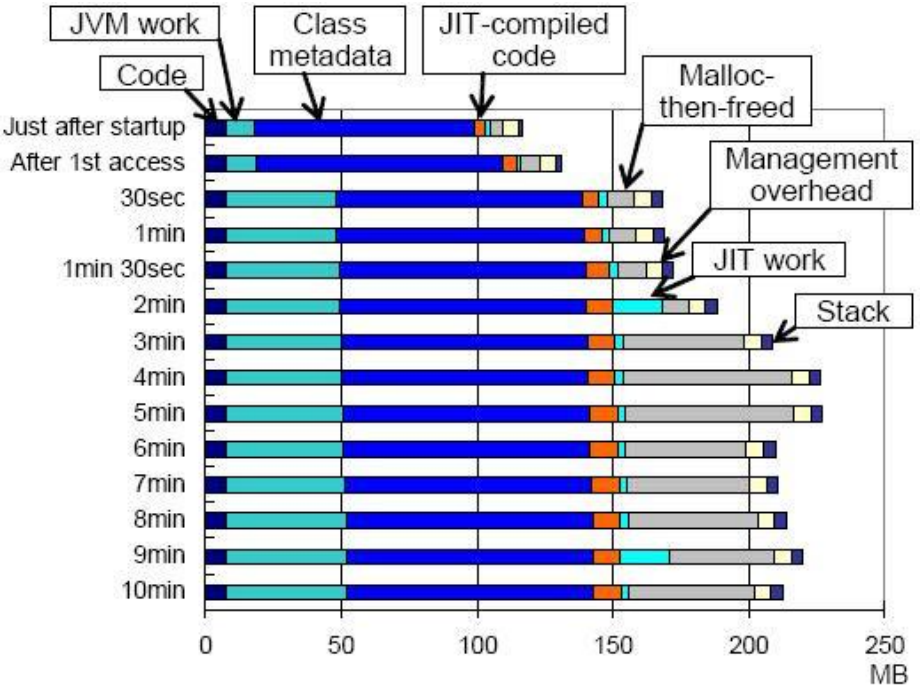
- Technika używana przy ogólnym menagerze pamięci i dla sterty javy
- My zastosujemy to dla obszaru roboczego JIT
  - Produkuje większość obszaru malloc-then-freed
  - Ograniczamy liczbę wywołań systemowych

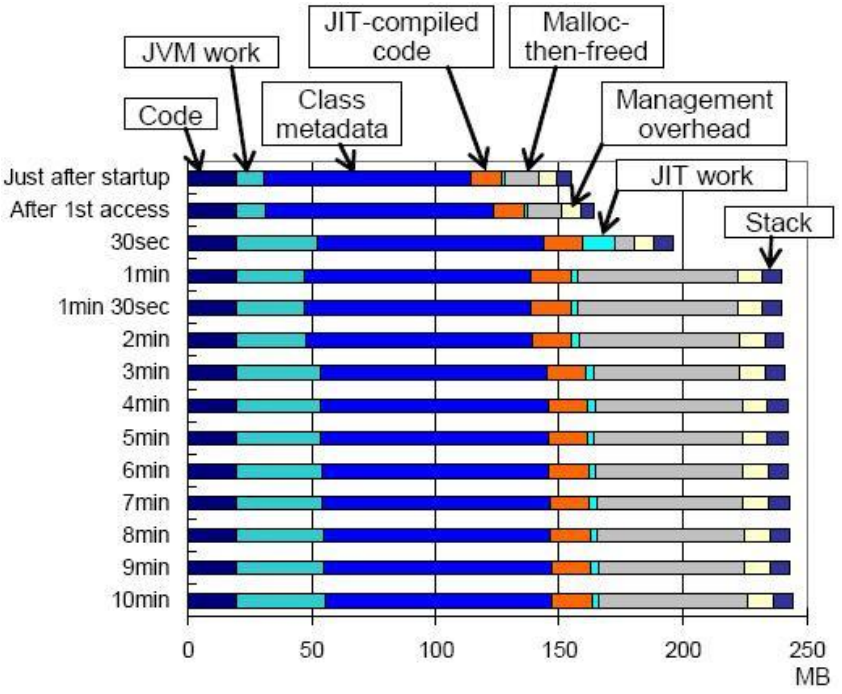
madvise() system call w Linuxie

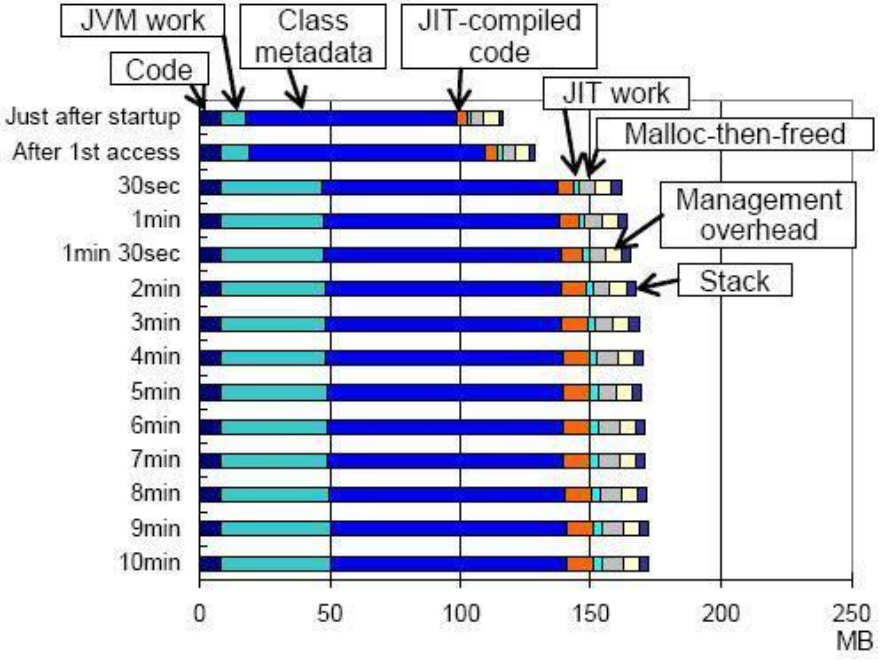
- Aplikacja informuje jądro jak zamierza używać dany obszar pamięci. Jądro może zastosować różne techniki usprawniające jak np. czytanie z wyprzedzeniem itp., ale może również zignorować taką informację
- Opcje zależą od konkretnego OS
- MADV_DONTNEED:
  - wskazane strony pamięci nie będą w najbliższym czasie użytkowane.
  - wirtualne adresy pozostaną zarezerwowane, fizyczne strony zostaną natomiast zwolnione

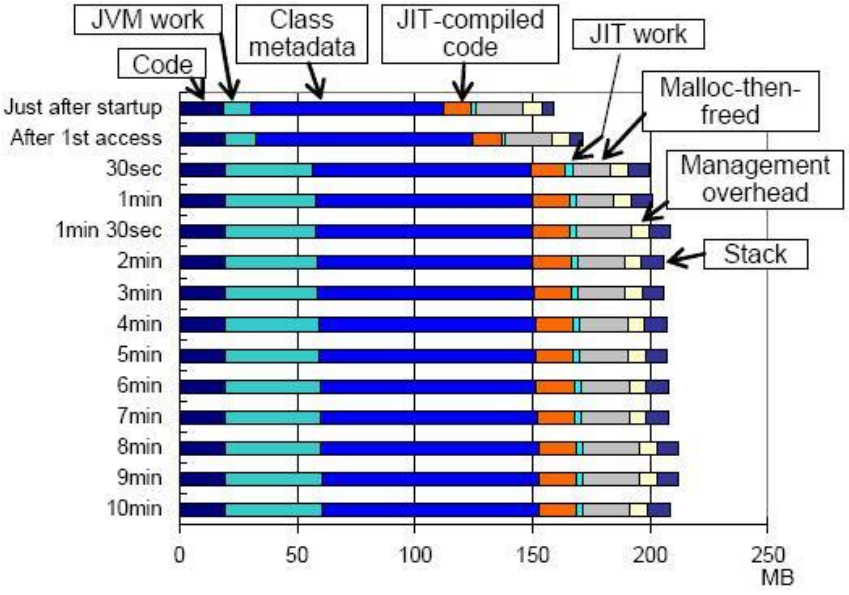
Modyfikacja JVM

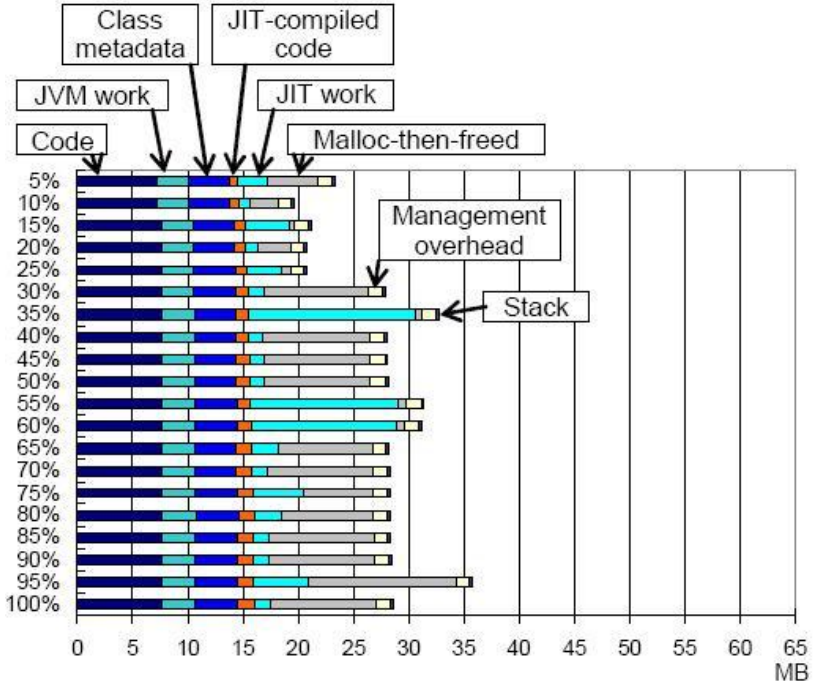
- Wywołujemy madvise() kiedy chcemy zwolnić blok pamięci
- Ograniczamy się do bloków zwalnianych przez obszar roboczy JIT

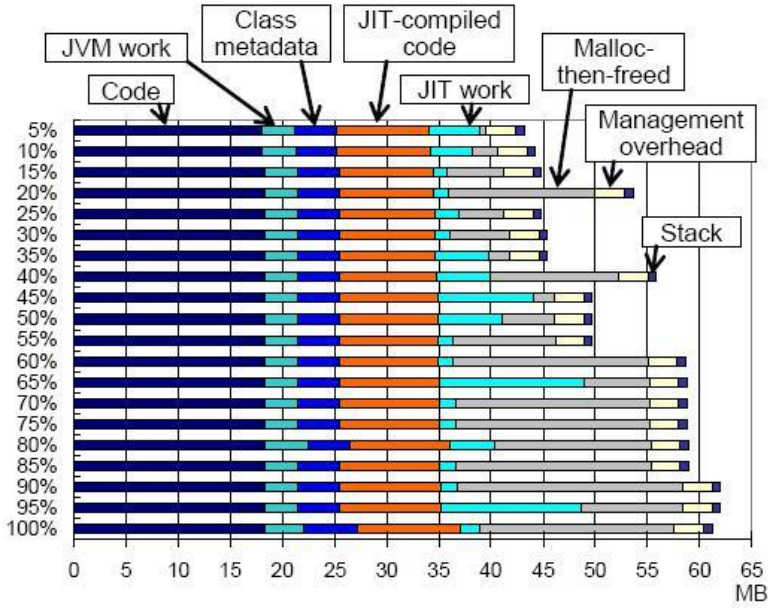

JVM work
Code
Class metadata
JIT-compiled code
Malloc-then-freed
Management overhead
JIT work
Stack
Just after startup
After 1st access
30sec
1min
1min 30sec
2min
3min
4min
5min
6min
7min
8min
9min
10min
0
50
100
150
200
250
MB

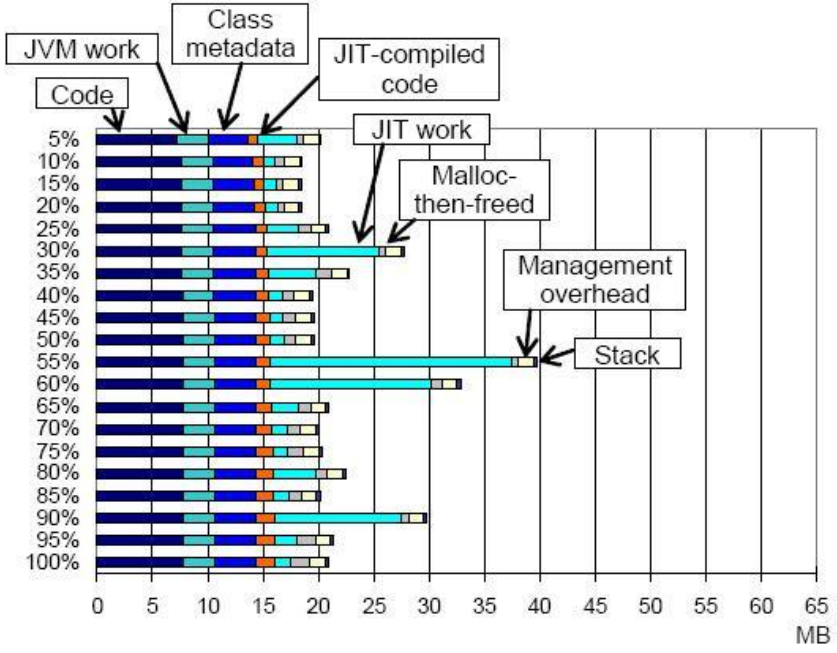

JVM work
Code
Class metadata
JIT-compiled code
Malloc-then-freed
Management overhead
JIT work
Stack
Just after startup
After 1st access
30sec
1min
1min 30sec
2min
3min
4min
5min
6min
7min
8min
9min
10min
0
50
100
150
200
250
MB

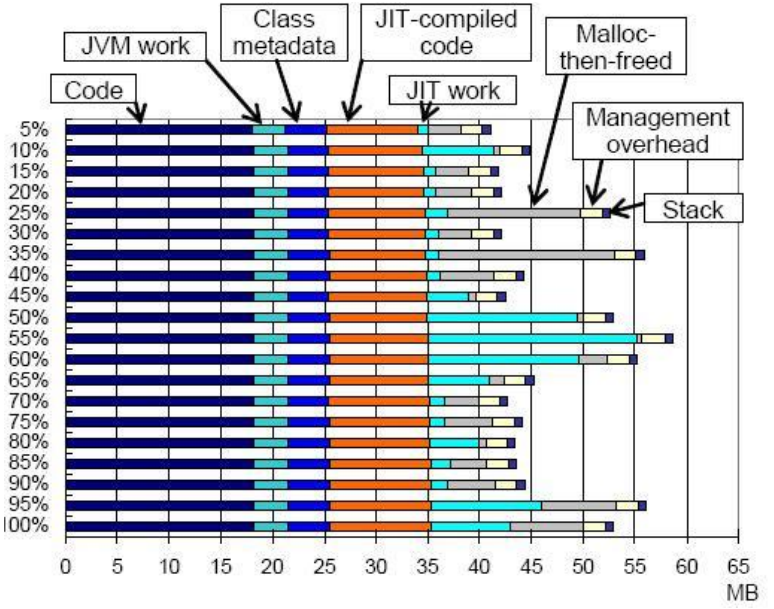

JVM work
Code
Class metadata
JIT-compiled code
JIT work
Malloc-then-freed
Management overhead
Stack
Just after startup
After 1st access
30sec
1min
1min 30sec
2min
3min
4min
5min
6min
7min
8min
9min
10min
0
50
100
150
200
250
MB


JVM work
Code
Class metadata
JIT-compiled code
JIT work
Malloc-then-freed
Management overhead
Stack
Just after startup
After 1st access
30sec
1min
1min 30sec
2min
3min
4min
5min
6min
7min
8min
9min
10min
0
50
100
150
200
250
MB

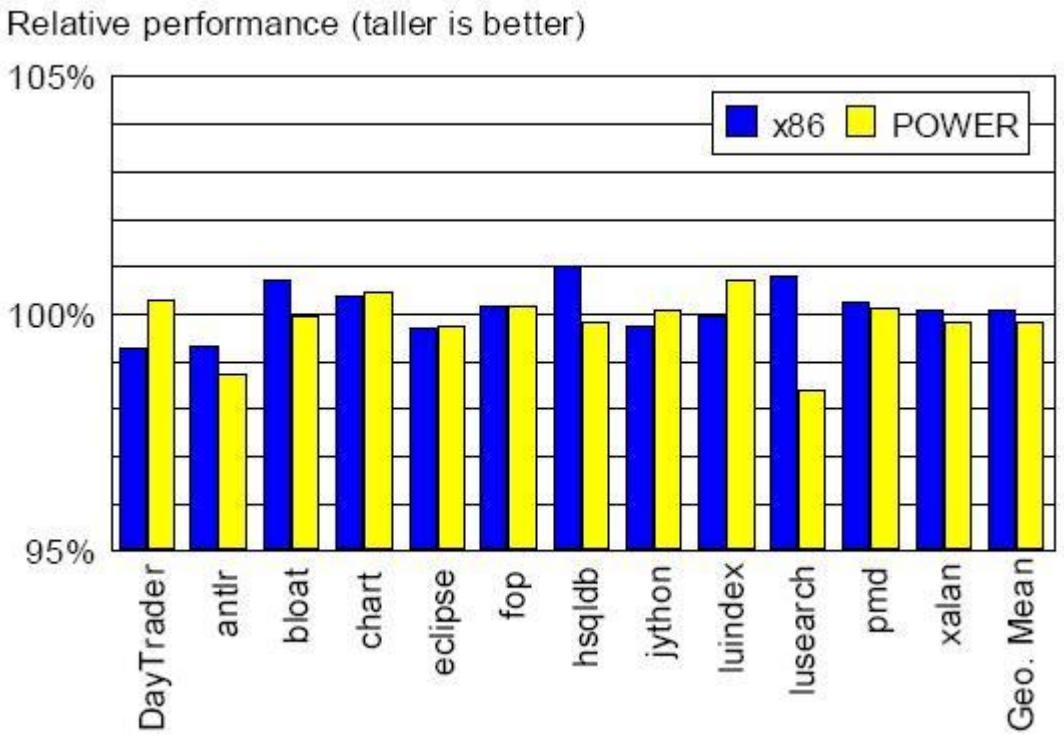

Class metadata
JIT-compiled code
JVM work
JIT work
Code
Malloc-then-freed
Management overhead
Stack
5%
10%
15%
20%
25%
30%
35%
40%
45%
50%
55%
60%
65%
70%
75%
80%
85%
90%
95%
100%
0 5 10 15 20 25 30 35 40 45 50 55 60 65
MB


JVM work
Class metadata
JIT-compiled code
Malloc-then-freed
Code
JIT work
Management overhead
Stack
5%
10%
15%
20%
25%
30%
35%
40%
45%
50%
55%
60%
65%
70%
75%
80%
85%
90%
95%
100%
0 5 10 15 20 25 30 35 40 45 50 55 60 65
MB


JVM work
Class metadata
JIT-compiled code
Code
JIT work
Malloc-then-freed
Management overhead
Stack
5%
10%
15%
20%
25%
30%
35%
40%
45%
50%
55%
60%
65%
70%
75%
80%
85%
90%
95%
100%
0 5 10 15 20 25 30 35 40 45 50 55 60 65
MB


JVM work
Class metadata
JIT-compiled code
Malloc-then-freed
Code
JIT work
Management overhead
Stack
5%
10%
15%
20%
25%
30%
35%
40%
45%
50%
55%
60%
65%
70%
75%
80%
85%
90%
95%
100%
0 5 10 15 20 25 30 35 40 45 50 55 60 65
MB

Wpływ na wydajność



Średnie różnice pomiędzy wywołaniami z madvise() i bez wynoszą 1% i 1.8% odpowiednio dla x86 i POWER

- Porównaliśmy czasy wykonań poszczególnych benchmarków
- Wyniki wskazują, że wprowadzone zmiany mają niewielki wpływ na wydajność

Swapping

- Wydaje się, że nie musimy się już martwić o duże rozmiary pamięci w obszarze malloc-then-freed, ponieważ OS powinien odbierać od nas wolne strony
- Nie wiadomo jednak czy nasze rozwiązanie nie spowoduje wymiany zbyt dużej liczby stron

Swapping

- Zostały więc przeprowadzone kolejne testy, które dodatkowo wskazały znaczącą poprawę w liczbie wymienianych stron

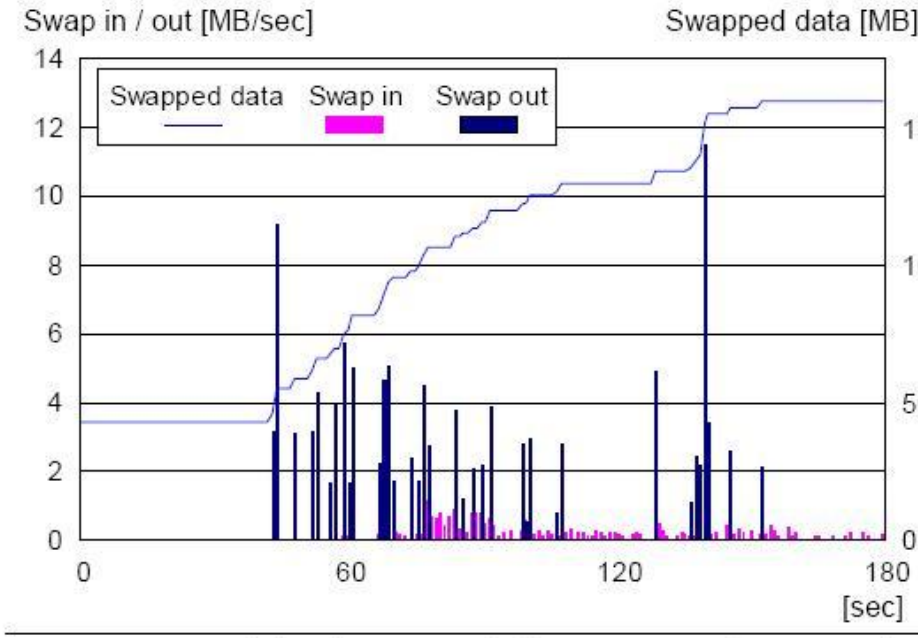


Figure 17. Disk I/O rate and the amount of swapped data during execution of two WAS processes running Apache DayTrader when madvise() was not called.

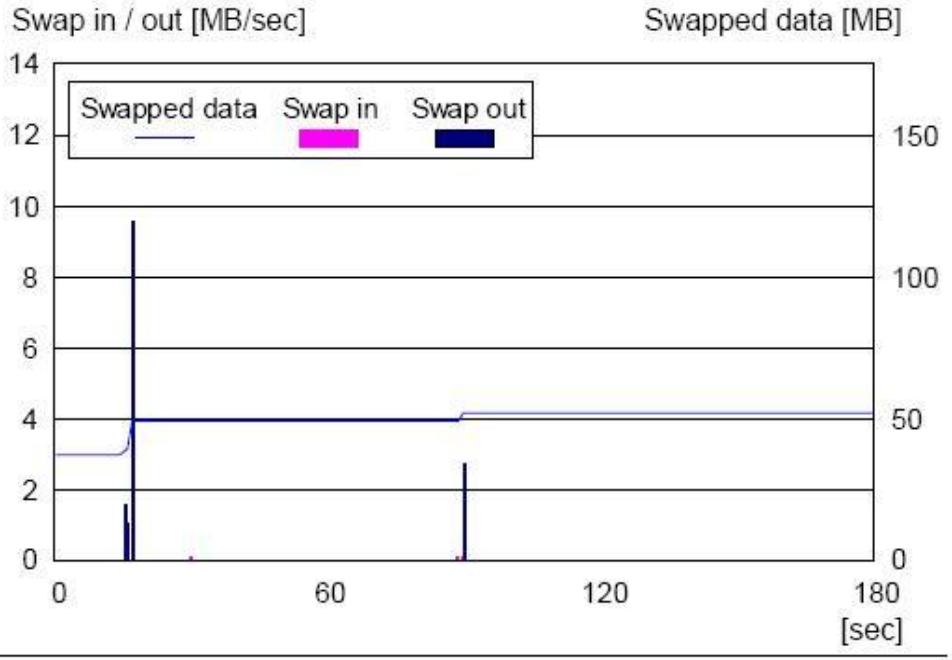


Figure 18. Disk I/O rate and amount of swapped data during execution of two WAS processes running Apache DayTrader when madvise() was called for the JIT work area

Wnioski

- Rozmiar pamięci NJ może równie duży jak rozmiar sterty dla wielu programów
- Pamięć NJ jest używana z wielu różnych powodów
  - Może to oznaczać wiele różnych przyczyn błędów takich jak wyjątki out-of-memory
- Coraz więcej optymalizacji jak w przypadku wywołań refleksyjnych oznacza również często coraz większe narzuty na przykład w pamięci NJ

Wnioski

- Zauważyliśmy również, że system zarządzania pamięcią libc, może mieć wpływ na rozmiar pamięci NJ
- Sugeruje to konieczność lepszej integracji różnych warstw MMS

Dziękuję za uwagę