석사학위논문

애플리케이션 성능 향상을 위한 자바 가상 기계의 가비지 컬렉터 활용 방안 연구

-오픈 소스 소프트웨어 OpenJDK 8, 11, 17 중심으로-

2023년

한성대학교 지식서비스&컨설팅대학원

스 마 트 융 합 컨 설 팅 학 과

스마트융합기술컨설팅전공

송 승 헌

석 사 학 위 논 문
지도교수 원종혁

애플리케이션 성능 향상을 위한 자바 가상 기계의 가비지 컬렉터 활용 방안 연구

-오픈 소스 소프트웨어 OpenJDK 8, 11, 17 중심으로-

A Study on the Application of Java Virtual Machine Garbage Collector for Improving Application Performance

;Focusing on Open Source Software OpenJDK 7, 11, 17

2023년 6월 일

한성대학교 지식서비스&컨설팅대학원

스 마 트 융 합 컨 설 팅 학 과

스마트융합기술컨설팅전공

송 승 헌

석 사 학 위 논 문
지도교수 원종혁

애플리케이션 성능 향상을 위한 자바 가상 기계의 가비지 컬렉터 활용 방안 연구

-오픈 소스 소프트웨어 OpenJDK 8, 11, 17 중심으로-

A Study on the Application of Java Virtual Machine Garbage Collector for Improving Application Performance

;Focusing on Open Source Software OpenJDK 7, 11, 17

위 논문을 컨설팅학 석사학위 논문으로 제출함

2023년 6월 일

한성대학교 지식서비스&컨설팅대학원

스 마 트 융 합 컨 설 팅 학 과

스마트융합기술컨설팅전공

송 승 헌

송승헌의 컨설팅학 석사학위 논문을 인준함

2023년 6월 일

심사위원장 김 승천 (인)

심 사 위 원 황 호영 (인)

심 사 위 원 원 종혁 (인)

국 문 초 록

애플리케이션 성능 향상을 위한 자바 가상 기계의 가비지 컬렉터 활용 방안 연구

- 오픈소스 소프트웨어 OpenJDK 8, 11, 17 중심으로 -

한성대학교 지식서비스&컨설팅대학원

스 마 트 융 합 컨 설 팅 학 과

스 마 트 융 합 기 술 컨 설 팅 전 공

송 승 헌

현대사회는 인터넷의 발달로 인해 디지털 혁신이 가속화 되고 있으며 애플리케이션을 통해 다양한 서비스를 접하게 되면서 삶의 질 향상을 이루게 되었다. 이러한 시대적 흐름에 의해 사용자 중심으로 새로운 애플리케이션이 개발되어 왔고 전 세계가 하나의 네트워크 속에서 연결되어 실시간으로 정보를 주고받는 클라우드 시대를 본격적으로 맞이하게 되었다.

그리고 디지털전환 이라는 트렌드와 발맞추어 초고속 정보통신시스템의 발전으로 초연결, 초고속, 초저지연 서비스의 구현을 맞이하게 되었다. 초연결 사회로의 기대가 급속히 증가하였으며 초고속 인터넷 서비스가 제공되면서 그 기대는 커지게 되었다. 더욱이 COVID-19 시대를 접하면서 비대면 커뮤니케이션 비중이 높아지게 되면서 애플리케이션이 현대화 되고 미래지향적 발전방향을 맞이하게 되었다. 이는 신규 서비스를 보다 빠르게 제공해야하는 경쟁시대를 맞이하게 되었다는 것을 뜻한다. 하지만 급속히 증가하는 사용자로 인하여 애플리케이션의 응답이 지연되는 문제점이 발생하게 되었고 여러

원인 중 하나로 메모리 적체 현상에 대한 개선 요구가 크게 증가하게 되었다. 사용자들의 요구사항에 대응하기 위해 복잡한 시스템이 구현되는 과정이 반복되면서 메모리 적체 현상이 발생하게 되었고 요구사항에 부합할수록 애플리케이션 개발 단계부터 이러한 문제를 항상 고려하면서 시스템을 개발해야 하는 번거로움이 커지게 되었다. 자바는 출시된 지 28년이 지나고 있지만 여전히 전 세계적으로 높은 점유율을 보이고 있다. 그렇다 보니 자바로 프로그래밍을 한 애플리케이션 서비스가 많아지게 되어 사용자의 불편사항 또한 비례적으로 높아지게 된 이유가 되었다.

본 연구는 소프트웨어로 구현되어 있는 애플리케이션을 사용하는 과정에서 메모리 적체와 같은 구조적 문제로 인해 응답시간이 지연되는 현상을 해결하고자 하였으며 실증 연구를 통한 성능개선 방안을 제시하고자 한다. 메모리 성능을 최적화하기 위한 방법은 오래전부터 대응방안이 적용되어 왔지만 예전 프로그램들은 구현에 어려움이 있었다. 그러나 자바로 프로그래밍을 하게 되면서 개선된 메모리 관리기법으로서 가비지 컬렉션(Garbage Collection)이 개발되었다. 메모리 관리만으로도 프로그램 성능이 향상되었으며 애플리케이션을 사용하는 사용자들의 불만을 크게 개선할 수 있는 효과를 볼 수 있게 되었다.

본 연구의 실험은 자바 프로젝트에서 가장 많이 사용하고 있으며 Long Term Support에 해당하는 자바 8, 11, 17 버전의 Parallel, G1, Z 가비지 컬렉터를 선정하였다. 각기 다른 힙 메모리 크기(2GB, 4GB, 8GB, 16GB)로 할당된 Tomcat 프로세스에서 가비지 컬렉션 정지 시간, 가비지 컬렉션 횟수, 응답 시간을 실험 후 결과를 분석하여 가비지 컬렉터 적용 방안을 구현하였다. 평가를 위해 Apache Jmeter와 Scouter를 사용 하였으며 사례별로 적용된 가비지 컬렉터를 활용한 애플리케이션의 성능 향상 지표를 실증하고자 한다.

【주요어】 자바 가상 기계, 가비지 컬렉션, 가비지 컬렉터, 자바, 애플리케이션

목 차

I. 서 론 ····· 1

1.1 연구의 배경 및 목적 ····· 1
1.1.1 연구의 배경 및 목적 ····· 1
1.1.2 연구의 방법 및 구성 ····· 3

II. 이론적 배경 ····· 4

2.1 애플리케이션 정의 ····· 4
2.2 애플리케이션 개발 언어 ····· 4
2.2.1 애플리케이션 개발 언어 사용현황 ····· 4
2.2.2 애플리케이션 개발 인력현황 ····· 6
2.2.3 자바 개발 언어 특징 ····· 7
2.2.4 OpenJDK와 OracleJDK 버전 동향 ····· 8
2.3 자바 가상 기계 ····· 10
2.3.1 자바 가상 기계 정의 ····· 10
2.3.2 자바 가상 기계 동작원리 ····· 11
2.3.3 자바 코드 실행과정 ····· 11
2.4 가비지 컬렉션 ····· 12
2.4.1 가비지 컬렉션 정의 ····· 12
2.4.2 가비지 컬렉션 종류 및 특징 ····· 16
2.4.2.1 Parallel 가비지 컬렉터 ····· 16
2.4.2.2 G1 가비지 컬렉터 ····· 16
2.4.2.3 Z 가비지 컬렉터 ····· 17
2.4.2.4 가비지 컬렉션 관련 에러 ····· 19
2.5 가비지 컬렉션 선행연구 ····· 22
2.5.1 가비지 컬렉션 선행연구 ····· 22
2.5.2 선행연구 분석결과 ····· 23
2.5.3 힙 메모리 적체 해결 방안 연구 ····· 25

III. 가비지 컬렉터 실험 설계 및 구현 ····· **26**

3.1 실험 환경 및 구성 ····· 26
3.1.1 실험 환경 및 구성 ····· 26
3.2 실험 결과 및 분석 ····· 28
3.2.1 실험 결과 및 분석 ····· 28

IV. 결 론 ····· **39**

4.1 결론 ····· 39
4.1.1 결론 및 시사점 ····· 39
4.1.2 연구의 한계점 및 향후 연구방향 ····· 40

참 고 문 헌 ····· **41**

ABSTRACT ····· 43

표 목 차

[표 1-1] 가비지 컬렉터 실험 구성 및 설정 ····· 3
[표 2-1] TIOBE index ····· 5
[표 2-2] 개발 언어별 소프트웨어 전문인력 ····· 6
[표 2-3] Java Support Roadmap ····· 9
[표 2-4] 자바 기본 가비지 컬렉션 ····· 9
[표 2-5] 가비지 컬렉션 쓰레드 ····· 13
[표 2-6] Java OutOfMemory Error 종류 ····· 19
[표 2-7] Java OutOfMemory Error Tomcat Log Message ····· 20
[표 2-8] 가비지 컬렉션의 성능 개선 관련 선행연구 ····· 22
[표 2-9] 정보시스템의 용량산정 및 성능 ····· 24
[표 2-10] 테스트 및 시험운영부 ····· 25
[표 3-1] 가비지 컬렉터 실험 구성 및 설정 ····· 26
[표 3-2] 실험 시스템 사양 ····· 27
[표 3-3] 가비지 컬렉션 파라미터 및 힙 구성 ····· 27
[표 3-4] 힙 크기 2GB에서 가비지 컬렉션 결과 ····· 28
[표 3-5] 힙 크기 4GB에서 가비지 컬렉션 결과 ····· 30
[표 3-6] 힙 크기 8GB에서 가비지 컬렉션 결과 ····· 32
[표 3-7] 힙 크기 16GB에서 가비지 컬렉션 결과 ····· 34
[표 3-8] 실험 유형별 애플리케이션 평균 응답시간 ····· 36

그 림 목 차

[그림 2-1] TIOBE index ····· 5
[그림 2-2] 자바 버전 별 사용률 ····· 7
[그림 2-3] 자바 벤더 별 점유율 ····· 8
[그림 2-4] 자바 프로그램 동작 구조 ····· 10
[그림 2-5] 클래스 로더 동작원리 ····· 11
[그림 2-6] 자바 코드 실행과정 ····· 12
[그림 2-7] Heap Memory Structure ····· 13
[그림 2-8] Garbage Collection Process ····· 16
[그림 2-9] G1 가비지 컬렉션 단계 ····· 17
[그림 2-10] Z 가비지 컬렉터 힙 메모리 영역 ····· 18
[그림 2-11] Z 가비지 컬렉션 단계 ····· 18
[그림 3-1] 2GB로 할당된 Parallel 컬렉터의 힙 메모리 사용량 ····· 29
[그림 3-2] 2GB로 할당된 G1 컬렉터의 힙 메모리 사용량 ····· 29
[그림 3-3] 2GB로 할당된 Z 컬렉터의 힙 메모리 사용량 ····· 29
[그림 3-4] 4GB로 할당된 Parallel 컬렉터의 힙 메모리 사용량 ····· 30
[그림 3-5] 4GB로 할당된 G1 컬렉터의 힙 메모리 사용량 ····· 31
[그림 3-6] 4GB로 할당된 Z 컬렉터의 힙 메모리 사용량 ····· 31
[그림 3-7] 8GB로 할당된 Parallel 컬렉터의 힙 메모리 사용량 ····· 33
[그림 3-8] 8GB로 할당된 G1 컬렉터의 힙 메모리 사용량 ····· 33
[그림 3-9] 8GB로 할당된 Z 컬렉터의 힙 메모리 사용량 ····· 34
[그림 3-10] 16GB로 할당된 Parallel 컬렉터의 힙 메모리 사용량 ····· 35
[그림 3-11] 16GB로 할당된 G1 컬렉터의 힙 메모리 사용량 ····· 35
[그림 3-12] 16GB로 할당된 Z 컬렉터의 힙 메모리 사용량 ····· 36
[그림 3-13] 힙 메모리 2,4,8,16GB에서 각 컬렉터의 평균 정지 시간 ····· 38

Ⅰ. 서론

1.1 연구의 배경 및 목적

1.1.1 연구의 배경 및 목적

스마트폰 기술의 발달을 포함한 IT 기술이 발전으로 제한된 자원을 탑재한 소형 기기들의 사용이 급증하게 되면서 이러한 소형 기기들의 성능을 개선하기 위한 많은 연구가 활발히 진행되었다(최형규, 2008). 성능 개선을 위해 애플리케이션 개선을 위한 개발 시도가 증가하였으며 Python, C, Java, C++ 등과 같은 프로그래밍 언어의 활용률이 높아졌다. 이중에서 자바는 운영체제에 영향을 받지 않고 독립적인 플랫폼으로서 개발이 가능하기 때문에 1995년 처음 공개된 이래로 28년 동안 전 세계적으로 많이 사용되면서 사용자들의 인기를 받았다(TIOBE Index, 2023). 자바는 오랜 기간 플랫폼 개발 언어로서 광범위하게 사용되어 왔으며 많은 개발자와 커뮤니티가 확보되었기 때문에 현재까지도 자바의 활용도는 상위권으로 인식되어 지고 있다. 애플리케이션 개발 언어 중에서 점유율 1위를 차지하였으며 최근에도 3위의 점유율을 유지할 정도로 그 활용도가 크다고 할 수 있다.

자바 프로그램의 장점으로 특정 플랫폼에 종속되지 않아 여러 환경에서 사용할 수 있고, 자바 가상 기계([JVM]:Java Virtual Machine)에 의해 자바 프로세스의 메모리가 자동으로 관리된다는 점이다(Oracle, 2023). 국내의 경우 현재 운영되는 많은 시스템이 자바 프로젝트로 이루어져 있고(소프트웨어정책연구소, 2022), 새로 도입되는 차세대나 고도화 프로젝트의 경우에도 기존 시스템과의 호환을 위한 자바 프로젝트가 많이 도입되고 있는 상황이다.

하지만 오랜 세월만큼 사용의 불편함도 발견되었으며 그 중 하나가 프로그램 동작 시 가비지 컬렉션 수행을 위한 지연 현상이 생기는 것이다. 이는 자바 언어 특성상 프로그래밍 동작이 수행되면 생성된 객체의 참조

내용이 지워지지 않고 계속 남게 되면서 메모리 용량을 차지하게 되는 현상이다. 따라서 메모리의 여유 공간이 부족해지고 프로그램 동작 시 응답이 늦어지는 지연 현상이 발생하게 되었다.

또한 자바 프로그램은 시스템을 위해 직접 컴파일된 C/C++ 프로그램보다 해석 오버헤드로 인해 선천적으로 느리게 실행된다는 특징을 가지고 있다(양병선, 2017). 이 때문에 메모리 공간을 차지하고 있는 불필요한 객체에 대하여 시스템 개발 과정 중간에 개발 소스를 수정하거나 지속적으로 모니터링을 하면서 튜닝을 하는 등 시스템 운영자의 불편함이 가중되게 되었다. 특히 큰 규모의 대국민 프로젝트 개발 산출물에서는 이러한 불필요한 메모리 적재로 인한 문제를 해결하기 위한 방법으로서 시스템 전체적인 성능 저하로 인한 손실 정도를 예측하고 대비하기 위한 단계의 프로세스 비중을 높이게 되었다.

자바 시스템을 포함한 소프트웨어 시스템 개발 방식은 분석 〉 설계 〉 구현 〉 테스트 방식으로 진행되며 주목받는 애자일 개발 방식 또한 요구사항에 대한 기능을 작게 나눈 분석 〉 설계 〉 구현 〉 테스트로 진행된다(기석, 2011). 각 개발 방법론의 테스트 단계는 애플리케이션 서비스 사용자의 불편함을 해소하고 성능과 안전성을 확보하는 단계라 할 수 있다. 자바 기반 시스템은 애플리케이션 성능 향상을 위한 여러 가지 방법 중의 하나로 가비지 컬렉션 부분을 개선 할 수 있다(김태용, 2016).

자바 시스템의 가비지 컬렉션 메커니즘이 수행될 때에는 일정시간의 정지 시간이 발생하며 정지 시간동안 서비스되고 있는 애플리케이션의 실행이 중단이 된다(Oracle, 2023). 따라서 정지 시간이 적게 걸릴수록 애플리케이션의 응답 시간과 처리량이 향상됨을 알 수 있다(김형원, 2011).

이에 본 연구에서는 가비지 컬렉션 기법으로 성능 개선을 구현하는 실험을 통해 가비지 컬렉션을 적용하는 정도에 따라 효과적으로 메모리 관리를 할 수 있는 개선 방안을 연구하게 되었다.

연구 방법으로는 가비지 컬렉터(Parallel, G1, Z)를 적용하여 가비지 컬렉션 수행시간, 가비지 컬렉션 횟수, 응답시간을 실증 분석하였으며 애플리케이션 성능 향상을 위한 최적의 가비지 컬렉터 적용 방법을 연구 하였다.

1.1.2 연구의 방법 및 구성

본 논문의 구성은 다음과 같다.

연구 방법은 가비지 컬렉터 적용에 따른 성능 수치를 비교를 할 수 있도록 [표 1-1]과 같이 구성한다. 부하 테스트 도구인 Jmeter를 사용하여 동일한 부하(Thread) 주고, Tomcat 프로세스에 할당되는 힙 메모리를 (2GB, 4GB, 8GB, 16GB) 로 달리하여 각 가비지 컬렉터 메커니즘이 동작하도록 설계하였다. 그리고 Tomcat 프로세스에서 남겨지는 가비지 컬렉션 로그를 통해 각 가비지 컬렉터 마다의 정지 시간 및 가비지 컬렉션 횟수를 확인하고 성능모니터링 도구인 Scouter를 사용하여 애플리케이션 응답 시간 비교를 진행하였다.

[표 1-1] 가비지 컬렉터 실험 구성 및 설정

구분	항목	가비지 컬렉터	힙 크기
1	OpenJDK 1.8.0.332 Tomcat 9.0.73	Parallel	2GB, 4GB, 8GB, 16GB
2	OpenJDK 11.0.2 Tomcat 9.0.73	G1	2GB, 4GB, 8GB, 16GB
3	OpenJDK 17.0.2 Tomcat 9.0.73	Z	2GB, 4GB, 8GB, 16GB

Ⅱ. 이론적 배경

2.1 애플리케이션

애플리케이션은 사용자가 원하는 기능을 수행하기 위해 필요한 정도로 제작된 소프트웨어로서 사용자가 직접 사용할 수 있는 응용 소프트웨어를 말하며 사용자가 컴퓨터를 사용하여 어떠한 일을 하려고 할 때 사용되는 거의 모든 프로그램을 지칭한다.

일반적으로 알려져 있는 대표적인 프로그램인 운영체제나 컴퓨터 시스템의 운영관리를 지원하는 다양한 유틸리티 프로그램 프로그램 언어와는 다르게 구분된다. 애플리케이션 소프트웨어는 입력되고 있는 실시간 데이터 또는 저장된 데이터를 처리하는 프로그램이라 정의할 수 있다(IT용어사전, 2023).

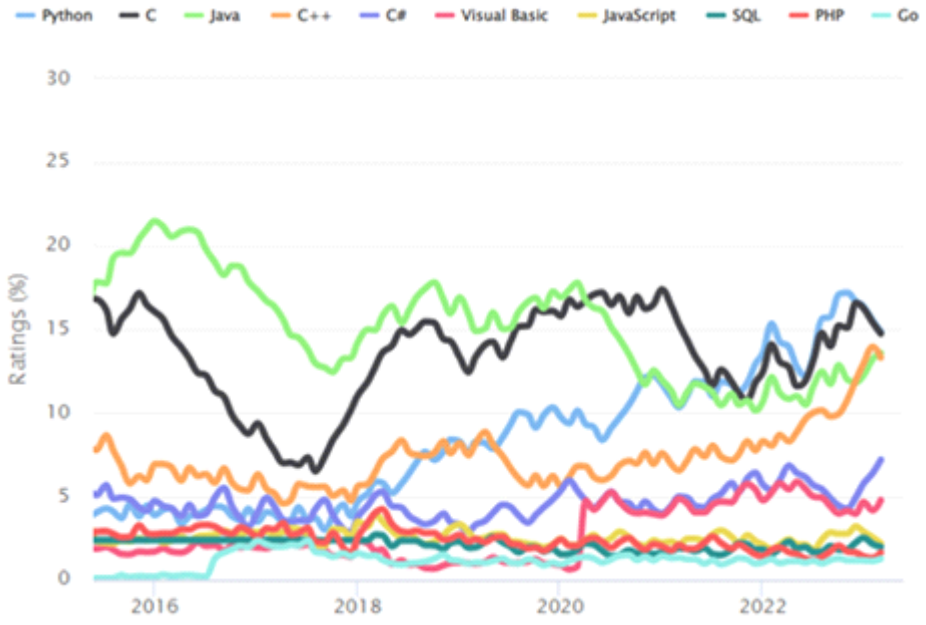
이러한 애플리케이션 소프트웨어는 문서처리, 사무계산, 데이터베이스, 과학기술용 수치 해석, 게임, 그래픽, 멀티미디어, CAD(Computer Aided Design), CAM(Computer Aided Manufacturing) 등 대부분의 산업분야의 정보처리 기반에 사용되고 있다.

2.2 애플리케이션 개발 언어

2.2.1 애플리케이션 개발 언어 사용현황

애플리케이션 개발 언어 사용현황은 [그림 1-1]과 같이 검색 엔진을 바탕으로 개발 언어의 인기도를 측정하는 TIOBE index[1] 통계를 살펴보면 2023년 자바 언어는 전년 대비 2.37% 증가한 13.56%(3순위) 점유율을 기록한 것으로 집계되고 있다. 자바는 2020년 까지 가장 높은 점유율을 차지하였으며 최근에는 Python, C, C++과 비슷한 점유율을 보이고 있지만 최상위 점유율을 계속 유지하고 있음을 알 수 있다.

1) TIOBE Software BV 사에서 엔지니어 수, 교육과정, 제품 공급업체 수를 기반으로, 검색엔진(Google,Bing,Yahoo, Wikipedia, Amazon, YouTube, BIDU)을 이용하여 개발 언어의 인기도를 측정하는 지표



[그림 2-1] TIOBE Index (TIOBE Index, 2023)

한편 1위는 Python 언어이며 2022년도에 이는 가장 높은 점유율(14.83%)을 보이고 있다. 2위는 C 언어로서 2022년도에 이어 연속 2위의 높은 점유율(14.73%)을 나타내고 있다. 2023년에는 1위 Python 언어와의 점유율 격차가 거의 없기 때문에 향후 점유율 변동이 예상되기도 한다.

[표 2-1] TIOBE Index (TIOBE Index, 2023)

프로그래밍 언어	점유율	전년 대비 증감률	2022.03 지수	2023.03 지수
Python	14.83%	+0.57%	1	1
C	14.73%	+1.67%	2	2
Java	13.56%	+2.37%	3	3
C++	13.29%	+4.64%	4	4
C#	7.17%	+1.25%	5	5

2.2.2 애플리케이션 개발 인력 현황

성능 구현이 좋고 호환성이 높은 애플리케이션 언어라 할지라도 실제 설계 구현 및 코딩작업을 하는 개발 인력의 참여 정도가 줄어들면 애플리케이션의 개선된 알고리즘 구현에 한계가 발생하면서 지속성 또한 줄어들게 된다. 따라서 개발 인력 현황에 대한 지표를 기준으로 애플리케이션 개발 언어가 향후 얼마나 점유율을 높게 유지할 수 있는지를 예측해 볼 수 있으며 개선을 위한 알고리즘 개발의 필요성이 얼마큼 높은지를 유추해 볼 수 있다.

2022년 소프트웨어정책연구소에서 조사된 소프트웨어 산업 실태조사에 따르면 [표 2-2] 과 같이 국내 자바 기반 시스템을 다루는 개발 전문 인력은 약 13만 명으로 Java(39.1%), C/C++(33.6%), Java Script(25.4%)의 순서로 높게 조사되었다. 자바 언어는 1995년 출시되었고 여전히 국내에서 최상위 수준의 인력 수급을 요구하는 개발 언어라는 것을 알 수 있다.

[표 2-2] 개발 언어별 소프트웨어 전문인력 (소프트웨어정책연구소, 2022)

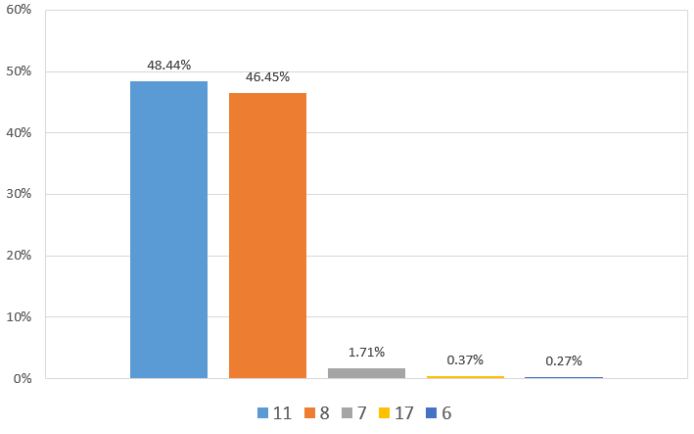
구분	2017년	2018년	2019년	2020년	2021년
C/C++	90.7	103.9	99.8	102.2	118.1
C#	-	-	52.0	56.3	64.3
Java	121.4	122.4	134.2	123.6	137.5
Java Script	53.7	72.0	83.2	78.5	89.3
PHP	17.3	25.0	22.8	20.5	24.1
HTML5	3.9	13.8	17.5	14.8	17.3
Object-C	9.8	16.5	21.4	17.9	18.7
Visual 계열[2]	26.5	21.3	7.9	5.9	5.9
파이썬	5.9	8.5	12.6	10.2	10.8
기타[3]	7.2	11.6	10.3	7.3	8.0

2) Visual Basic, Visual C++,NET

2.2.3 자바 개발 언어 특징

자바의 장점은 특정 플랫폼에 종속되지 않아 여러 환경에서 사용할 수 있는 특징과 자바 가상 기계에 의해 자바 프로세스의 메모리가 자동으로 관리된다는 특징이 있다(Oracle, 2023). 또한 플랫폼 의존 부분을 제외하면 C, C++ 기반의 프로그래밍도 자바 언어로 간결하게 작성 가능하다는 특징이 있다(한판암 외, 1995). 국내의 경우 현재 운영되는 많은 시스템이 자바 프로젝트로 이루어져 있으며 새로 도입되는 프로젝트 및 고도화 프로젝트의 경우에도 기존 시스템과의 호환을 위해 자바 프로젝트가 많이 도입되고 있다.

NR State of Java report[4] 에서 조사된 자바 버전의 사용 현황은 위 [그림 2-2]와 같다. 2022년 자바 8, 자바 11이 가장 일반적으로 사용되는 버전으로 조사되었다. 2018년 출시된 자바 11은 Long Term Service(LTS) 버전으로 현재 운영 환경에 있는 애플리케이션의 48.44% 비율로 사용되고 있고, 이는 2020년의 11.11%에서 37.33% 만큼 증가한 수치이다. 자바 8은 46.45%로 2위로 조사되었고, 2020년 84.48%의 점유율에서 38.03% 감소하였다. 자바 7는 1.71%, 자바 17는 0.37%, 자바 6는 0.27%로 조사되었다.

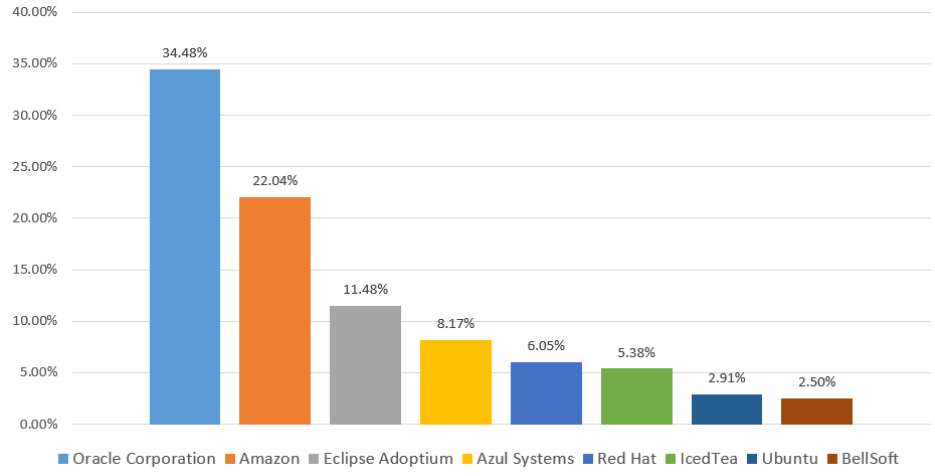


[그림 2-2] 자바 버전 별 사용율 (New Relic, 2022)

3) Assembly language, Delphi/Object Pascal, MATLAB, Pascal, R 등

4) New Relic사 에서 성능 데이터를 제공하는 애플리케이션의 데이터를 바탕으로 집계한 지표

[그림 2-3]는 2022년 자바 벤더 별 점유율로서 Oracle은 34.48%로 가장 큰 점유율을 조사되었고 Amazon은 22.04%로 조사되었다. Oracle은 2020년 75% 점유율에 비해 감소하였으며 Amazon을 포함하여 여러 Vendor가 배포한 JDK의 점유율은 증가하였지만 이는 OpenJDK 기반으로 개발된 특징이 있다.



[그림 2-3] 자바 벤더 별 점유율 (New Relic, 2022)

2.2.4 OpenJDK와 OracleJDK 버전 동향

JDK(Java Development Kit)는 JRE(Java Runtime Environment) 환경에 추가 라이브러리들이 포함된 제품이다. OpenJDK는 Open Java Development Kit로 GNU General Public License version2 라이선스를 따르는 공개 소프트웨어이다. OracleJDK는 Oracle Binary Code License Agreement 라이선스를 따르는 유료 소프트웨어이다. 같은 버전에 대한 기본 소스 구현체는 OpenJDK, OracleJDK가 동일하며, OracleJDK의 경우 LTS 버전 정책으로 장기간 동안 버그 패치를 지원받을 수 있는 장점이 있다. 자바 시스템을 도입 및 개발하는 경우에는 보안 및 패치에 대한 이슈로 LTS 버전을 기반으로 시스템 개발이 된다. [표 2-3] Java Support Roadmap을 보면,

LTS에 해당하는 JDK버전은 7, 8, 11, 17이다.

[표 2-3] Java Support Roadmap (Oracle, 2023)

Ver	출시 일	지원 기간
7 (LTS)	July 2011	July 2022
8 (LTS)	March 2014	December 2030
9 (non-LTS)	September 2017	Not Available
10 (non-LTS)	March 2018	Not Available
11 (LTS)	September 2018	September 2026
12 (non-LTS)	March 2019	Not Available
13 (non-LTS)	September 2019	Not Available
14 (non-LTS)	March 2020	Not Available
15 (non-LTS)	September 2020	Not Available
16 (non-LTS)	March 2021	Not Available
17 (LTS)	September 2021	September 2029
18 (non-LTS)	March 2022	Not Available
19 (non-LTS)	September 2022	Not Available
20 (non-LTS)	March 2023	Not Available

OpenJDK버전에 따른 기본 가비지 컬렉터는 [표 2-4]의 자바 기본 가비지 컬렉션과 같다. OpenJDK 8는 Parallel 가비지 컬렉터로 동작하며, OpenJDK 11, 17는 G1 가비지 컬렉터로 기본 동작된다.

[표 2-4] 자바 기본 가비지 컬렉터 (OpenJDK, 2023)

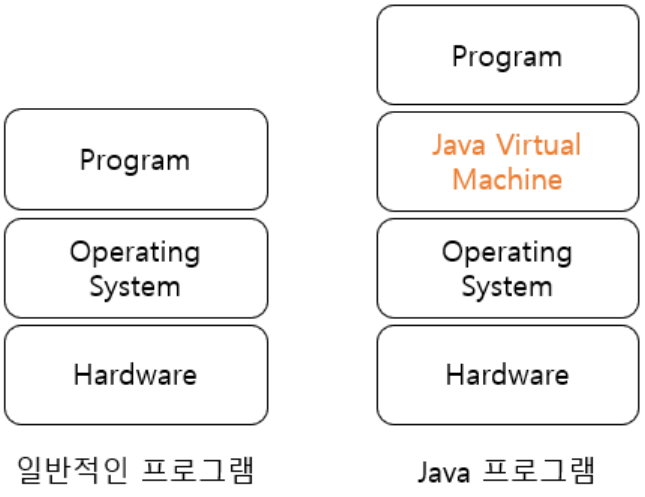
버전	기본 가비지 컬렉터
OpenJDK 8	Parallel GC
OpenJDK 11	G1 GC[5]
OpenJDK 17	G1 GC

5) JDK 11, 17 버전에서는 운영체제의 CPU가 1개일 경우 Serial 가비지 컬렉터, 2개 이상일 경

2.3 자바 가상 기계

2.3.1 자바 가상 기계 정의

자바가 등장하기 전까지 C/C++ 의 특성상 하드웨어나 운영체제에 따라 같은 기능의 소스를 각 하드웨어에 맞게 작성해야하는 불편함이 존재했다. 이 문제를 해결하기 위해 자바 언어가 개발되었다. 자바의 가장 큰 특징으로 [그림 2-4]와 같이 어떠한 플랫폼이던 상관없이 바이트 코드가 플랫폼 독립적 이라는 것이다. 이러한 특징을 구현하기 위해서는 자바 가상 기계가 필요하다. 자바 가상 기계는 컴파일된 바이트 코드를 실행시켜주는 자바 애플리케이션을 실행하는 표준 환경을 의미한다(Oracle, 2023). 소형 기기와 같은 임베디드 환경에서는 K-가상 기계가 사용되었고(최인범 외, 2004) 현재는 같은 Micro Edition 환경에서 구현된 여러 임베디드 자바 가상 기계가 사용된다.

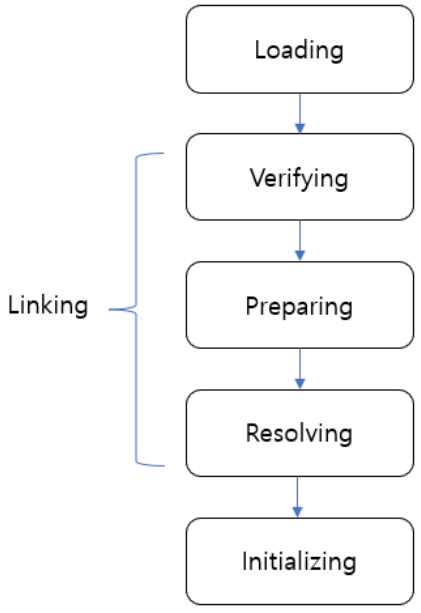


[그림 2-4] 자바 프로그램 동작 구조 (Oracle, 2023; 연구자 재구성)

우, G1 가비지 컬렉터가 기본 방식이다.

2.3.2 자바 가상 기계 동작원리

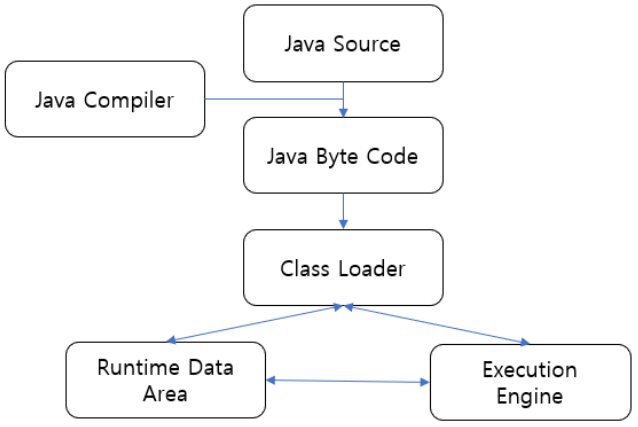
자바 가상 기계 내부의 클래스 로더는 동적으로 클래스 읽어들이는 런타임 로드 및 링크하는 특징이 있다. 클래스 로더에는 계층적으로 상위 클래스 로더가 있다. 캐시된 메모리를 확인하여 클래스가 이미 로드 되었는지를 먼저 확인한 후, 상위 클래스 로더에 위임하여 필요한 클래스를 확인하여 [그림 2-5]과 같이 단계별로 역할을 수행한다. 첫 번째, 로드 영역은 클래스 파일을 가져와서 자바 가상 기계의 메모리 영역에 로드 한다. 디스크에 반복적으로 로드 할 필요가 없게 되어 성능이 향상되는 부분이다. 두 번째, 검증 영역은 읽어 들인 클래스가 명세대로 .class 파일 형태가 유효한지 검사한다. 세 번째, 클래스가 필요로 하는 메모리 할당을 위한 준비를 한다. 네 번째, 분석 영역은 심볼릭 레퍼런스를 실제 레퍼런스로 변경한다. 다섯 번째, 초기화 영역은 클래스와 인터페이스 변수를 초기화 한다.



[그림 2-5] 클래스 로더 동작원리 (Oralce, 2023; 연구자 재구성)

2.3.3 자바 코드 실행과정

자바 코드 실행 과정은 네 단계로 구분할 수 있다. 첫 번째, 작성한 자바 파일을 자바 컴파일러를 통해 class 파일 형태의 자바 바이트 코드로 컴파일 한다. 두 번째, 컴파일된 class 파일 형태의 바이트 코드를 자바 가상 기계의 클래스 로더에게 전달한다. 세 번째, 클래스 로더는 필요한 클래스를 로딩 및 링크하여 자바 가상 기계 메모리 영역에 올린다. 네 번째, 실행 엔진은 자바 가상 기계 메모리에 올라온 바이트 코드를 실행한다.



[그림 2-6] 자바 코드 실행과정 (Oracle, 2023; 연구자 재구성)

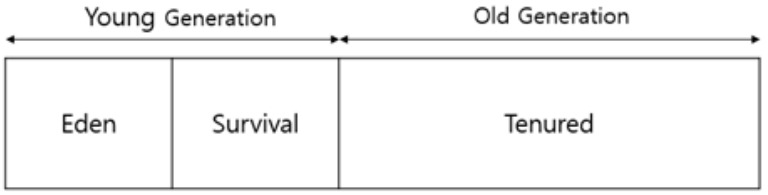
2.4 가비지 컬렉션

2.4.1 가비지 컬렉션 정의

가비지 컬렉션은 힙 메모리 공간에서 더 이상 사용하지 않는 객체를 정리하여, 다시 힙 메모리 공간을 확보하는 메커니즘이다. C/C++의 언어의 경우, 힙 메모리 할당과 반납 부분을 개발자가 직접 설정하지만, 자바 가상 기계는 힙 메모리에 대하여 자동으로 가비지 컬렉션을 한다. 가비지

컬렉션이 수행되는 도중에는 가바지 컬렉션 처리를 하는 쓰레드 외에는 다른 서비스 쓰레드들은 잠시 멈추는 시간인 정지 시간(Puase Time)이 생긴다(Oracle, 2023). 자바 시스템에서는 가비지 컬렉션에 대하여 적절한 옵션 조정을 통해 정지 시간을 가능한 줄여야 하고, 사용자들의 애플리케이션을 사용에 불편함이 없도록 해야 한다(이용수 외, 2002).

Hotspot 자바 가상 기계의 힙 구조는 [그림 2-7]과 같이 Eden, Survival, Tenured 영역으로 나뉜다. Eden 영역은 새로 생성된 객체가 처음 위치하는 곳이다.



[그림 2-7] Heap Memory Structure (Oracle, 2006)

Eden 영역의 객체 수용 크기가 넘어서게 되면 객체 참조 여부를 확인한다. 이때, 참조가 존재하는 객체는 가비지 컬렉션이 발생하면서 Survivor 영역으로 이동하게 된다. Eden 영역과 Survivor 영역의 객체 참조 여부 확인을 반복하면서 수용 크기를 넘어서면, Tenured 영역으로 객체가 이동된다. Eden, Survival 영역은 비교적 참조가 적은 Young Generation 영역이며, Tenured는 참조가 어느 정도 진행된 Old Generation 영역에 해당한다.

Hotspot 자바 가상 기계에서 수행되는 가비지 컬렉션 쓰레드는 아래 [표2-5]로 확인할 수 있다. 가비지 컬렉터의 쓰레드 수의 경우 자바 가상 기계가 실행되는 플랫폼 환경에 따라 변화되어 실행된다.

[표 2-5] 가비지 컬렉션 쓰레드

가비지 컬렉터	쓰레드 정보
Parallel	"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x0000000002789000 nid=0x5784 runnable

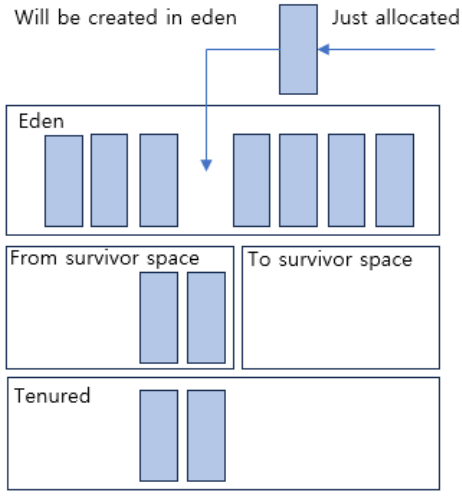
	"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x000000000278a800 nid=0x6824 runnable "GC task thread#2 (ParallelGC)" os_prio=0 tid=0x000000000278c800 nid=0x29ac runnable "GC task thread#3 (ParallelGC)" os_prio=0 tid=0x000000000278f000 nid=0x3f60 runnable "GC task thread#4 (ParallelGC)" os_prio=0 tid=0x0000000002791000 nid=0x6fa4 runnable "GC task thread#5 (ParallelGC)" os_prio=0 tid=0x0000000002792800 nid=0x4968 runnable "GC task thread#6 (ParallelGC)" os_prio=0 tid=0x0000000002795800 nid=0x4ee8 runnable "GC task thread#7 (ParallelGC)" os_prio=0 tid=0x0000000002796800 nid=0x6780 runnable
G1	"GC Thread#0" os_prio=2 cpu=31.25ms elapsed=36.35s tid=0x0000021307e14000 nid=0x3cac runnable "GC Thread#1" os_prio=2 cpu=31.25ms elapsed=35.33s tid=0x000002132dc33000 nid=0x3840 runnable "GC Thread#2" os_prio=2 cpu=31.25ms elapsed=35.33s tid=0x000002132da38000 nid=0x2b04 runnable "GC Thread#3" os_prio=2 cpu=31.25ms elapsed=35.34s tid=0x000002132d9bd800 nid=0x6ed4 runnable "GC Thread#4" os_prio=2 cpu=31.25ms elapsed=35.34s tid=0x000002132d9be000 nid=0x40e8 runnable "GC Thread#5" os_prio=2 cpu=15.63ms elapsed=35.34s tid=0x000002132d9be800 nid=0x3178 runnable "G1 Main Marker" os_prio=2 cpu=0.00ms elapsed=36.35s tid=0x0000021307e72000 nid=0x3f10 runnable "G1 Conc#0" os_prio=2 cpu=15.63ms elapsed=36.35s tid=0x0000021307e77800 nid=0x22cc runnable "G1 Conc#1" os_prio=2 cpu=15.63ms elapsed=34.72s tid=0x000002132d96e000 nid=0x4cf8 runnable "G1 Refine#0" os_prio=2 cpu=0.00ms elapsed=36.35s tid=0x000002132bc79800 nid=0x58dc runnable "G1 Refine#1" os_prio=2 cpu=0.00ms elapsed=35.34s tid=0x000002132d9c0000 nid=0x17cc runnable

	"G1 Young RemSet Sampling" os_prio=2 cpu=0.00ms elapsed=36.35s tid=0x000002132bc7a800 nid=0x3920 runnable "VM Periodic Task Thread" os_prio=2 cpu=0.00ms elapsed=36.16s tid=0x000002132c85c800 nid=0x4bd4 waiting on condition
Z	"ZDirector" os_prio=2 cpu=0.00ms elapsed=22.23s tid=0x0000021a47d993b0 nid=0x2b70 runnable "ZDriver" os_prio=2 cpu=0.00ms elapsed=22.23s tid=0x0000021a47d989e0 nid=0x58f8 runnable "ZStat" os_prio=2 cpu=0.00ms elapsed=22.23s tid=0x0000021a47d99d10 nid=0x6424 runnable "ZUnmapper" os_prio=2 cpu=0.00ms elapsed=22.25s tid=0x0000021a47d967a0 nid=0x6918 runnable "ZUncommitter" os_prio=2 cpu=0.00ms elapsed=22.25s tid=0x0000021a47d970f0 nid=0x3b74 runnable "ZWorker#0" os_prio=2 cpu=0.00ms elapsed=22.25s tid=0x0000021a47d86880 nid=0x3cac runnable "ZWorker#1" os_prio=2 cpu=0.00ms elapsed=22.25s tid=0x0000021a47d87da0 nid=0x3f10 runnable "RuntimeWorker#0" os_prio=2 cpu=0.00ms elapsed=22.24s tid=0x0000021a47d9b810 nid=0x40d4 runnable "RuntimeWorker#1" os_prio=2 cpu=0.00ms elapsed=22.24s tid=0x0000021a47d9c9c0 nid=0x3840 runnable "RuntimeWorker#2" os_prio=2 cpu=0.00ms elapsed=22.24s tid=0x0000021a47d9d2c0 nid=0x2b04 runnable "RuntimeWorker#3" os_prio=2 cpu=0.00ms elapsed=22.24s tid=0x0000021a47d9e360 nid=0x6ed4 runnable "RuntimeWorker#4" os_prio=2 cpu=0.00ms elapsed=22.24s tid=0x0000021a47da08c0 nid=0x4bd4 runnable

2.4.2 가비지 컬렉션 종류 및 특징

2.4.2.1 Parallel 가비지 컬렉터

Parallel 컬렉터는 자바 8의 기본 가비지 컬렉터이고, 자바 옵션 지정(-XX:+UseParallelGC) 으로 컬렉터 선택이 가능하다. Young Generation 영역에 대해서 여러 개의 쓰레드를 사용하여 가비지 컬렉션을 수행하는 컬렉터 이다. CPU 개수만큼의 가비지 컬렉션 쓰레드를 사용하고, 여러 쓰레드를 사용하는 만큼 가비지 컬렉션 처리량을 늘리는데 목표를 둔다. [그림 2-8] 가비지 컬렉션 프로세스의 Tenured 영역에 대해서 Mark-sweep-compact 알고리즘이 수행된다. Mark-sweep-compact는 Old 영역에서 가비지 컬렉션 대상 객체에 대해 식별 및 지운 뒤, 남아있는 객체들을 한곳으로 압축하는 방식이다. Mark-sweep은 객체 참조 그래프의 크기에 따라 달라 질 수 있어, Mark 작업에 소비되는 시간은 비결정적인 특징이 있다(이상윤, 2006).

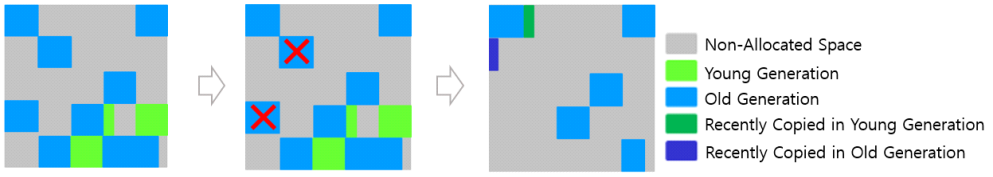


[그림 2-8] Garbage Collection Process (Oracle, 2006; 연구자 재구성)

2.4.2.2 G1 가비지 컬렉터

G1 컬렉터는 자바 9 버전의 기본 가비지 컬렉터이며, 자바 옵션 지정

(-XX:+UseG1GC)으로 컬렉터 선택이 가능하다. 정지 시간을 최소화하는 것을 목표로 하고, 힙 메모리는 비연속적인 영역들의 집합(Region)으로 구성되며, 최소 크기는 1MB이고 최대 크기는 32MB로 할당된다. 이는 여러 집합들의 크기가 쉽게 조정되기 위해서이다. 집합 내 객체 참조에 대한 정보는 Remember Set에서 관리된다.



[그림 2-9] G1 가비지 컬렉션 단계 (Oracle, 2011)

G1 가비지 컬렉션 단계는 [그림 2-9]과 같다. 연두색 집합은 Young Generation 객체가 포함되고, 하늘색 집합에는 Old Generation 객체가 포함된다. 첫 번째 단계에서는 Old Generation 집합 여러 개가 존재하는 상태이다. 새로운 객체는 참조 여부에 따라, 빈 공간에 새로운 연두색 집합으로 할당된다. 두 번째 단계에서는 Concurrent Marking 작업을 진행하여 참조되는 객체가 없는 집합이 있는지 확인한다. 참조가 없다면 해당 집합에 대한 할당을 해제하여 영역의 공간을 확보한다. 세 번째 단계는 Reclaim 및 Compaction 단계로 하단의 연두색 집합인 Young Generation에 대해서 가비지 컬렉션이 수행되고, 그 결과 상단 초록색의 Survivor 집합의 형태로 압축되었다. 또한, Young 집합에 가까이 위치했던 일부 Old 집합에 대해서도 가비지 컬렉션이 수행되어 상단의 파란색으로 Old 집합이 압축되어 생성된다.

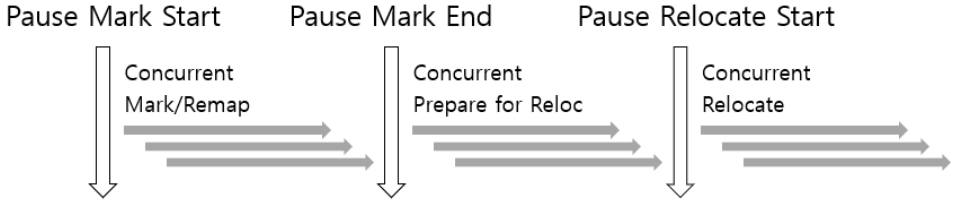
2.4.2.3 Z 가비지 컬렉터

Z 가비지 컬렉터의 특징은 저 지연성, 밀리초 미만의 최대 일시 중지 시간을 목표로 한다. 자바 11버전 이상부터 옵션(-XX:+UseZGC) 지정으로 컬렉터 선택이 가능하다. 힙, 루트 세트에 크기에 비례하여 정지 시간이 증가하지

않아, 8MB 작은 메모리에서부터 16TB의 큰 힙 메모리 크기가 할당될 수 있는 애플리케이션에도 적용이 가능하다. 아래 [그림 2-10] Z 가비지 컬렉터 힙 메모리 영역처럼 하늘색의 Small (2MB) 영역, 남색의 Medium(32MB) 영역, 파랑색의 Large(N x 2MB) 영역들로 구성된다.

[그림 2-10] Z 가비지 컬렉터 힙 메모리 영역 (Oracle, 2018)

[그림 2-11]은 Z 가비지 컬렉션의 단계 별 수행을 나타낸다. Pause Mark Start는 Root-set에서 객체 Mark 표시를 하는 초기 단계이다. Concurrent Mark/Remap는 Root-set의 객체에 참조를 가진 모든 객체에 Mark를 표시하는 단계이다. Pause Mark End 는 새로운 객체에 대해서 Mark를 표시하고, 약한 참조에 대해 정리 작업이 수행된다. Concurrent Prepare for Reloc 는 재배치 할 영역을 탐지하고, Pause Relocate Start 단계는 모든 Root 참조의 재배치를 진행한다. Concurrent Relocate는 Load barriers를 사용하여 모든 객체를 재배치 및 참조를 수정한다.



[그림 2-11] Z 가비지 컬렉션 단계 (Oracle, 2018)

2.4.2.4 가비지 컬렉션 관련 에러

힙 메모리 공간이 부족하게 되면, OutOfMemoryError 예외처리가 발생하면서 객체를 더 이상 할당할 수 없게 된다. OutOfMemoryError 예외처리는 자바 클래스 로드를 지원하기 위한 기본 메모리가 충분하지 않은 경우거나, 자바 가상 기계에서 더 이상 새 객체 생성을 위한 공간을 만들 수가 없는 상황에 발생하고, 가비지 컬렉션 수행에 과도한 시간이 소요되고, 해제되는 메모리가 거의 없는 경우에도 발생한다. 자세한 에러 예외 상황의 내용은 아래 [표 2-6]와 같다. OutOfMemoryError가 발생할 경우, 적용된 가비지 컬렉터의 튜닝에 대해 고려 할 수 있지만, 힙 메모리 할당의 크기 조정 부분과 애플리케이션의 메모리 누수 지점도 같이 확인하는 작업이 필요하다.

[표 2-6] Java OutOfMemory Error 종류 (Oracle, 2014)

java.lang.OutOfMemoryError 메시지 종류	설명
Java heap space	객체를 힙에 할당할 수 없는 경우이며, 지정된 힙 크기(또는 지정되지 않은 경우 기본 크기)가 응용 프로그램에 충분하지 않은 구성인지 확인이 필요하다. 객체 수명이 긴 응용 프로그램의 경우, 응용 프로그램이 객체에 대한 참조를 보유하고 있어 가비지 컬렉션 대상이 되지 않으므로, 프로그래밍된 부분에서 객체의 메모리 누수 의심되는 부분을 확인해야 한다.
Metaspace	클래스의 메타 데이터를 할당을 할 수 없을 경우에 발생한다. JDK 7 버전의 PermGen 영역에서 Metaspace 영역으로 대체되었다.
GC Overhead limit exceeded	가비지 컬렉션이 계속 수행 중이고, 가비지 컬렉션 후에도, 힙 메모리가 2% 미만으로 복구 되는 경우 발생한다. 새 객체 할당을 위한 여유 공간이 거의 없는 경우에 발생한다.
Requested array size	힙 메모리 크기 보다 큰 배열을 할당하는 경우에 발생한다.

exceeds VM limit	
request size bytes for reason. Out of swap space?	힙 메모리에 할당에 실패하는 소스 모듈 이름이 출력된다. 잘못 컴파일된 메서드로 자바 가상 기계가 충돌하는 경우에도 발생한다.

[표 2-7]를 보면 Tomcat WAS 로그에서 OufOfMemoryError 발생의 원인이 Java heap space 내용으로 발생된 것을 알 수 있다. 누수 지점은 애플리케이션 gc.jsp 소스의 7번째 줄에서 예외 처리가 된 내용을 확인 할 수 있다.

자바 가상 기계에서 수행하는 가비지 컬렉션 외에 애플리케이션에서 System.gc() 메서드를 직접 호출을 하여도 가비지 컬렉션이 가능하다. 하지만, System.gc() 메서드 호출과 동시에 발생하는 정지 시간이 자바 가상 기계에서 수행하는 가비지 컬렉션 정지 시간보다 오래 수행되는 경우가 많다. 이 경우에는 -XX:+DisableExplicitGC 옵션 사용으로 명시적인 가비지 컬렉션을 비활성화시켜 가비지 컬렉션 수행 성능을 높일 수 있다.

[표 2-7] Java OutOfMemory Error Tomcat Log Message

```
SEVERE [http-nio-8010-exec-4]
org.apache.catalina.core.StandardWrapperValve.invoke
Servlet.service() for servlet [jsp] in context with path [] threw
exception
[javax.servlet.ServletException: java.lang.OutOfMemoryError:
Java heap space] with root cause
java.lang.OutOfMemoryError: Java heap space
Stacktrace:] with root cause
java.lang.OutOfMemoryError: Java heap space
at org.apache.jsp.gc_jsp._jspService
(gc_jsp.java:124)
at org.apache.jasper.runtime.HttpJspBase.service
(HttpJspBase.java:70)
at javax.servlet.http.HttpServlet.service
(HttpServlet.java:596)
```

```
at org.apache.jasper.servlet.JspServletWrapper.service
JspServletWrapper.java:466)
at org.apache.jasper.servlet.JspServlet.serviceJspFile
(JspServlet.java:379)
at org.apache.jasper.servlet.JspServlet.service
(JspServlet.java:327)
at javax.servlet.http.HttpServlet.service
(HttpServlet.java:596)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter
(ApplicationFilterChain.java:209)
at org.apache.catalina.core.ApplicationFilterChain.doFilter
(ApplicationFilterChain.java:153)
...
at org.apache.coyote.AbstractProcessorLight.process
(AbstractProcessorLight.java:63)
at org.apache.coyote.AbstractProtocol$ConnectionHandler.process
(AbstractProtocol.java:926)
at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun
(NioEndpoint.java:1791)
at org.apache.tomcat.util.net.SocketProcessorBase.run
(SocketProcessorBase.java:49)
at org.apache.tomcat.util.threads.ThreadPoolExecutor.runWorker
(ThreadPoolExecutor.java:1191)
at org.apache.tomcat.util.threads.ThreadPoolExecutor$Worker.run
(ThreadPoolExecutor.java:659)
at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run
(TaskThread.java:61)
at java.base/java.lang.Thread.run
(Thread.java:833)

Servlet.service() for servlet [jsp] in context with path [] threw
exception [An exception occurred processing [/gc.jsp] at line [7]
```

2.5 가비지 컬렉션 선행연구

2.5.1 가비지 컬렉션 선행연구

[표 2-8] 가비지 컬렉션의 성능 개선과 관련한 선행연구의 결과로, 가비지 컬렉션은 애플리케이션 성능에 영향을 미치는 요소임을 확인 할 수 있다.

[표 2-8] 가비지 컬렉션의 성능 개선 관련 선행연구

년도	주요 연구 내용
백대현,박희상,양희권,이철훈 (2002)	Solaris SPARC 운영체제에서 마크-회수 알고리즘을 개선한 마크-회수 압축 알고리즘을 이용하여 가비지 컬렉션을 설계 및 가비지 콜렉션 구현 동작 확인.
이은화 (2004)	generation 기법을 사용한 자바 플랫폼에서 Young generation의 크기 조정에 따른 성능 측정 결과를 바탕으로, 효율적인 방법을 제안 함. 동일한 힙의 크기가 할당될 때, Young generation 크기 조정함으로서 쓰레기 수집의의 횟수와 실행시간의 성능이 향상 됨.
이상윤 (2006)	한국전자통신연구원에서 개발된 임베디드용 자바 가상 머신(EVM)에서 구현된 메모리 관리 기법을 제안하여, 객체 생성 수가 많아짐에 따라 단편화 율이 거의 일정한 결과를 보임.
최형규 (2014)	Laze worst fit, Biased allocator, ahead-of-time heap expansion 기법을 제안하여, 쓰레기 수집기의 호출을 줄여 사용자 반응성과 메모리 관리 성능을 개선하는 연구를 진행. 해당 기법이 반영된 Android 환경에서 기본 응용프로그램들을 이용하여 평가 했을 때, 쓰레기 수집기의 수행 시간은 최대 31% 줄이는 결과를 보임.
김태용 (2016)	HP-UX 운영체제와 자바 가상 기계 메모리 관리 측면에서 애플리케이션의 Life Cycle 및 자바 힙 메모리 영역에 생성되는 자바 객체들의 가비지 컬렉션 패턴을 분석하여 성능 향상 및 시스템의 안정성을 향상시키고자 함. 자바 가상 기계 옵션 튜닝을 통해 비니지스 영역에서 사용되고 있는 각각의 사례들의 경우, 성능 향상 측면에서 최고 200% 이상 개선됨.
민철기,엄현영,한혁	자바 가상 기계의 쓰레기 수집 방법 중 Parallel Compact에 대해 분석하고 문제점에 대해 논의 함.

(2021)	Parallel Compact 영역에 대하여, 영역 내부 compact와 영역 간 compact로 나누어 설계 및 구현하였고, 실험을 통해 Full GC 수행시간이 최대 43%까지 줄어드는 결과를 보임.

* 자료 : 선행연구를 토대로 연구자 재구성

2.5.2 선행연구 분석결과

이은화(2004)는 1M~64M로 힙의 크기를 달리 조정하여 가비지 컬렉션 성능 측정을 하였다. 또한 동일한 힙 크기일 때 Young Generation 영역의 크기를 전체 힙 크기 대비 20%~50%로 달리 조정하였고, 실행 시간의 성능을 측정하였다. 그 결과 힙의 크기가 커질수록 가비지 컬렉션 횟수가 줄고, 가비지 컬렉션 실행 시간은 많이 소요되었다. 동일한 힙 크기 인 경우 Young Generation을 조정을 했을 시 성능이 향상되었다. 최형규(2014)는 가상 시스템에서 Lazy Worst Fit allocator를 채택할 경우 가비지 컬렉션시에 Overhead가 감소한 결과를 보였다. 김태용(2016)은 hp-ux Operating System 환경에서 자바 가상 기계의 가비지 컬렉션 패턴을 분석하여, 애플리케이션의 성능향상 측면에서 개선점을 확인하였다. Parallel 가비지 컬렉터와 Concurrent Mark-Sweep 가비지 컬렉터가 실험에 채택되었고, 높은 처리량이 요구되는 환경에서는 Young Generation과 Old Generation의 비율을 1:1 조정이 성능 향상을 결과를 보였다. 다만, Unix인 HP-UX Operation System의 메모리 관리 측면의 연관성이 고려가 필요하다. 민철기, 염헌영, 한혁(2021)은 JDK 8u222 버전에서 Parallel Compact에 대한 가비지 컬렉션을 병렬 쓰레드 부분과 싱글 쓰레드로 처리하는 것으로 나누어 실험하였다. 워크로드에 따라 Full 가비지 컬렉션 수행시간이 최대 43%까지 줄어드는 결과를 보인다. 선행 연구 공통적으로 가비지 컬렉션과 관련한 allocator나 Generation 영역에 대한 비율 조정을 통하여 가비지 컬렉션 처리에 대한 성능 개선의 결과를 확인하였다.

다만, 특정 운영체제 종속성을 탈피하기 위한 Windows, macOS 및 여러 Linux 배포 판에서 사용가능한 JDK 채택이 필요해 보이며 할당되는 힙 크기

의 경우도 현재 적용되고 있는 운영 시스템의 환경 범위에 맞춘 크기로 적용이 고려되어야 한다. 채택되는 JDK 버전도 최근 시점 기준으로 버그 및 개선 사항이 반영된 출시 버전과 가장 많이 사용되고 있는 JDK 버전으로 선정이 필요하겠다.

또한, 국내 행정안전부에서 배포된 [표2-9]의 행정 및 공공기관 정보시스템 구축·운영 지침[6] 에서는 애플리케이션의 요구 응답시간(Response Time)은 정보시스템의 성능 평가 항목 중의 하나로 제시되어 있다. 응답시간은 어떤 작업이 실행 가능한 시점부터 그 작업이 완료되는 시점까지 경과된 시간이며 애플리케이션 서버가 사용자에게 요청 결과를 반환하는 데 걸리는 시간을 말한다. 응답 시간이 빠를수록 처리되는 분당 요청 수(Transaction Per Second)는 증가하게 된다. 전 국민을 대상으로 하는 공공·대민 서비스의 경우 응답시간이 사용자 만족으로 이어지게 된다. 따라서 응답 시간에 영향을 미치는 요소 중 하나인 가비지 컬렉션의 정지 시간을 줄이는 것이 중요하다.

[표 2-9] 정보시스템의 용량산정 및 성능 (행정안전부 행정 및 공공기관 정보시스템 구축·운영 지침, 2022)

세부 평가항목	검토여부	검토내용
현행 업무 및 시스템에 대한 용량 및 성능 관련 자료를 기술하였는가?		
시스템의 용량산정을 위해 정보시스템의 사용자 수, 최대 동시 접속자수, 요구 응답시간, 데이터양 등을 기술하였는가?		
향후 업무확장, 인원 및 데이터 증가 정도를 기술하였는가?		

행정기관 및 공공기관 정보시스템 구축에서는 구축된 시스템의 품질 확보를 위해 부하 테스트로 응답 시간에 대한 성능 시험을 진행한다.

6) 행정기관 등의 장이 정보시스템을 구축·운영함에 있어서 준수해야 할 기준, 표준 및 절차와 법 제49조제1항에 따른 상호 운용성 기술평가에 관한 사항을 정함을 목적으로 하는 지침

[표 2-10] 테스트 및 시험운영부 (행정안전부 행정·공공기관 웹사이트·구축 운영 가이드, 2021)

○ 사업완료 전에 성능 기능 및 보안 요구사항 등이 충실히 구현되었는지 , 확인하고 요구사항을 충족하지 못하는 경우에는 개선 요구하고 안정적인 서비스 제공을 위하여 일정기간 시험 운영을 실시하는 것이 필요 ○ 부하 테스트로 응답시간 시간당 처리량 자원 사용량 등 성능시험 -응답시간: 사용자 관점에서 작업 처리를 요청한 시간으로부터 이를 시스템 처리하여 결과를 보여줄 때까지 소요된 시간 -시간당 처리량: 시스템이 성공적으로 처리한 단위 시간당 요청 처리 건수 페이지 (처리건수, 요청 처리건수), 이 때 시간당 최대 얼마만큼 처리할 수 있는지 최댓값 사용 -자원사용량: 자원(CPU,메모리 등)들의 용량 중 실제 사용하고 있는 값의 비율

2.5.3 힙 메모리 적체 해결 방안 연구

힙 메모리 적체 문제를 해결하기 위해서는 가비지 컬렉션 관련 옵션 조정을 통한 적용 방법이 있다. Young Generation의 크기를 다르게 설정하거나, Young Generation과 Old Generation의 비율을 조정하는 법이 있다. 하지만 실제 운영되는 자바 기반 애플리케이션 성격에 따라 모니터링을 선행 한 후에 가비지 컬렉션 영역에 대한 세부 조정을 해나가는 방식이 필요하며 옵션 조정의 목표는 결국 가비지 컬렉션 정지 시간을 줄이는 것이 된다.

따라서 본 실증 연구의 특징은 선행연구와는 다른 특정 운영체제의 종속성을 탈피한 Windows, macOS 및 여러 Linux 배포 판에서 사용가능한 JDK를 채택하여 동일한 애플리케이션에 대하여 3가지의 가비지 컬렉터를 적용한 후 정지 시간 발생 정도를 중심으로 성능 실험을 하는데 독창적 의미가 있다.

Ⅲ. 가비지 컬렉터 실험 설계 및 구현

3.1 실험 환경 및 구성

3.1.1 실험 환경 및 구성

가비지 컬렉터 실험 설계는 아래 [표 3-1]과 같이 3가지로 구성한다. LTS 버전의 구현체가 되는 OpenJDK 8, 11, 17 버전을 선택하였고 WAS 서버로는 엔터프라이즈 환경에서 오픈소스 소프트웨어로 많이 사용되고 있는 Tomcat을 선정하였다. 각각의 실험구성에서 가비지 컬렉터를 Parallel, G1, Z 컬렉터로 선택하고 동일한 애플리케이션에 대하여 정지 시간, 가비지 컬렉션 횟수를 비교 한다.

[표 3-1] 가비지 컬렉터 실험 구성 및 설정

구분	항목	가비지 컬렉터
실험구성 1	OpenJDK 1.8.0.332 Tomcat 9.0.73	Parallel
실험구성 2	OpenJDK 11.0.2 Tomcat 9.0.73	G1
실험구성 3	OpenJDK 17.0.2 Tomcat 9.0.73	Z

성능 모니터링 도구로 선정한 Scouter는 시스템의 가용한 자원을 기반으로 단위 시간당 많은 요청을 서비스 할 수 있는지에 대한 측정과 애플리케이션의 응답시간을 확인하는데 활용이 가능한 오픈 소스 소프트웨어이다. 실험 구성에 부하를 주기 위하여 부하테스트 도구로는 JMeter를 사용한다. JMeter는 애플리케이션은 테스트 기능 동작과 성능을 측정하도록 설계된 오픈소스 소프트웨어로서 서버 또는 객체의 부하 강도를 테스트 할 수 있다. 구체적인 실험 시스템의 하드웨어 사양과 소프트웨어 버전은 [표 3-2]와 같다.

[표 3-2] 실험 시스템 사양

구분	항목	사양
HW	OS	Windows 11
	CPU, Memory	Intel Core i7-1260P Memory 32GB
SW	JDK	OpenJDK 1.8.0.332 OpenJDK 11.0.2 OpenJDK 17.0.2
	WAS	Tomcat 9.0.73
	부하 테스트도구	Apache JMeter 5.5
	성능 모니터링도구	Scouter 2.17, 2.18

[표 3-3]와 같이 실험 구성에 적용된 가비지 컬렉터에 가비지 컬렉션 파라미터 옵션을 적용한다. 해당 옵션의 Tomcat WAS 디렉터리 구조의 catalina.bat 파일에 옵션 값으로 지정하며 가비지 컬렉션 수행 기록을 Tomcat WAS 로그에 자세하게 출력하도록 한다. 힙 크기가 2GB, 4GB, 8GB, 16GB로 다르게 할당된 환경에서 각 가비지 컬렉터의 정지 시간과 가비지 컬렉션 횟수를 비교 진행하였다.

[표 3-3] 가비지 컬렉션 파라미터 및 힙 크기 구성

구분	Garbage Collector	Garbage Collection Parameter	Heap Size
실험구성 1	Parallel	-verbose:gc -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintGCDetails -Xloggc	2GB,4GB, 8GB,16GB
실험구성 2	G1	-verbose:gc -Xloggc	2GB,4GB, 8GB,16GB
실험구성 3	Z	-verbose:gc -Xlog:gc*	2GB,4GB, 8GB,16GB

3.2 실험 결과 및 분석

3.2.1 실험 결과 및 분석

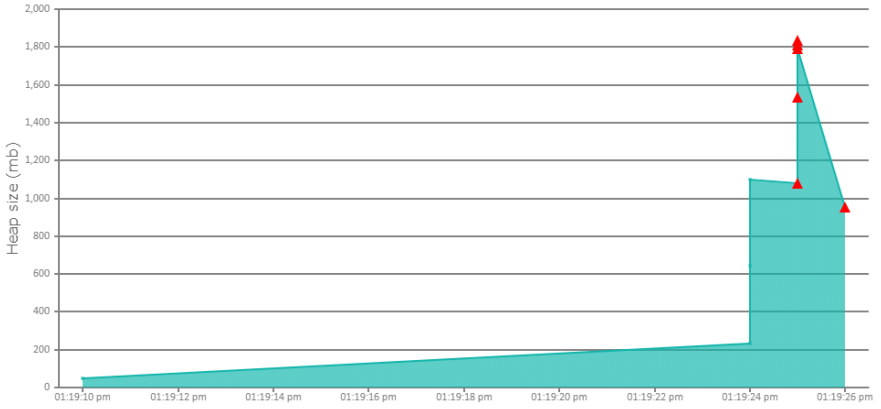
[표 3-4]의 실험 1-1은 힙 크기 2GB로 할당된 메모리에서의 Tomcat WAS 프로세스의 가비지 컬렉션의 정지 시간 측정 결과이다. Parallel 컬렉터로 적용 시 평균 가비지 컬렉션 시간은 56.9ms 소요되었고 Z 컬렉터로 적용한 경우 평균 0.0264ms가 소요되었다. 가비지 컬렉션 횟수의 경우 G1 컬렉터가 550회로 가장 많이 수행되었고 Parallel 컬렉터가 13회로 가장 적게 수행되었다. Scouter로 측정된 Parallel , G1, Z 컬렉터의 애플리케이션의 평균 응답시간은 3ms~4ms 정도로 큰 차이를 보이지 않았다.

[표 3-4] 힙 크기 2GB에서 가비지 컬렉션 결과

구분	Garbage Collector	Heap Size	Avg GC Pause Time	Max GC Pause Time	GC Count	App평균 응답시간
실험 1-1	Parallel	2GB	56.9ms	170ms	13	4ms
	G1	2GB	15ms	64.1ms	550	4ms
	Z	2GB	0.0264ms	0.596ms	117	3ms

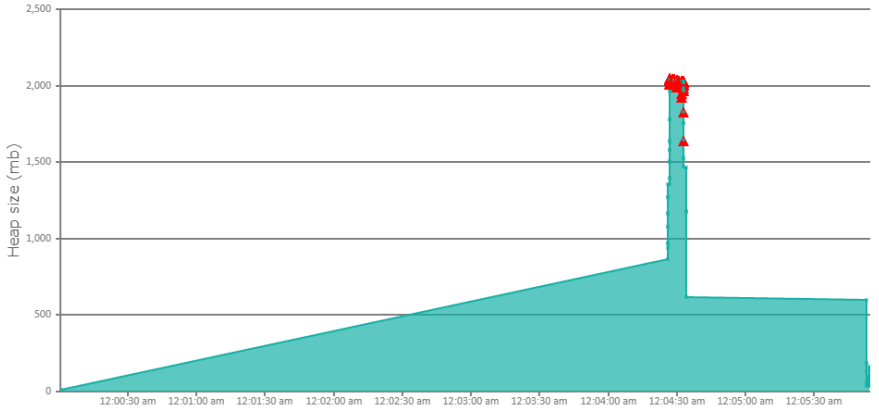
가비지 컬렉션 수는 가비지 컬렉션이 수행된 횟수를 의미하는데 Parallel 컬렉터가 G1, Z 컬렉터보다 적게 수행되었다. Parallel 경우 Minor 가비지 컬렉션과 Major 가비지 컬렉션이 포함된 횟수이다. G1 가비지 컬렉션의 경우 Young 가비지 컬렉션, Concurrent, Remark, Cleanup 이 포함된 횟수이며 Z 가비지 컬렉션의 경우 Pause Mark End, Pause Mark Start, Pause Relocate Start이 포함된 횟수이다.

[그림 3-1]는 2GB 할당된 힙 메모리에서 Parallel 컬렉터 적용 시 가비지 컬렉션 후 힙 메모리 사용률을 나타낸 그래프이다. 최대 1.8GB 메모리 사용률을 보였고, 가비지 컬렉션이 진행되어 힙 메모리의 가용 메모리를 확보하였다.



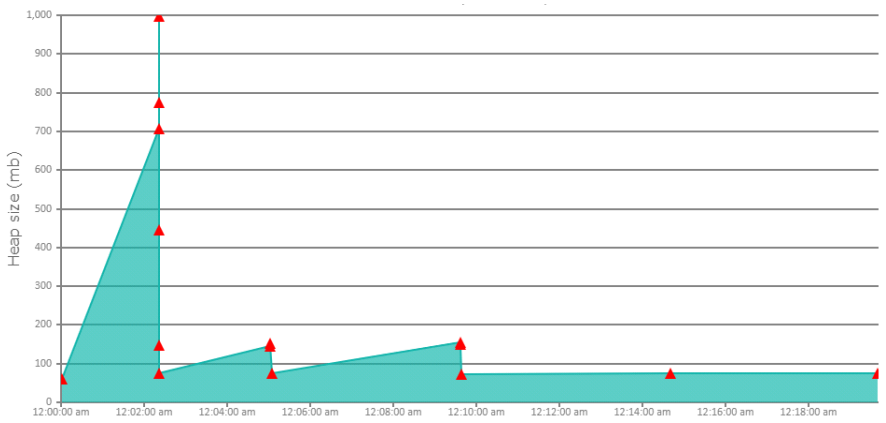
[그림 3-1] 2GB로 할당된 Parallel 컬렉터의 힙 메모리 사용량

[그림 3-2]에서는 G1 컬렉터가 최대 2GB에 해당하는 메모리 사용률을 보였다.



[그림 3-2] 2GB로 할당된 G1 컬렉터의 힙 메모리 사용량

[그림 3-3]의 Z 컬렉터의 경우, 2GB에서 최대 사용률 1GB 이하로 힙 메모리가 관리되었다. 이는 사용되지 않고 남은 1GB 크기만큼 추가적으로 객체를 생성 및 정리 공간으로 사용 가능하다는 의미이다. Parallel 컬렉터, G1 컬렉터의 경우 부하 발생 시 짧은 시간이었지만 모든 힙 메모리 공간을 사용한 것과 차이가 있다.



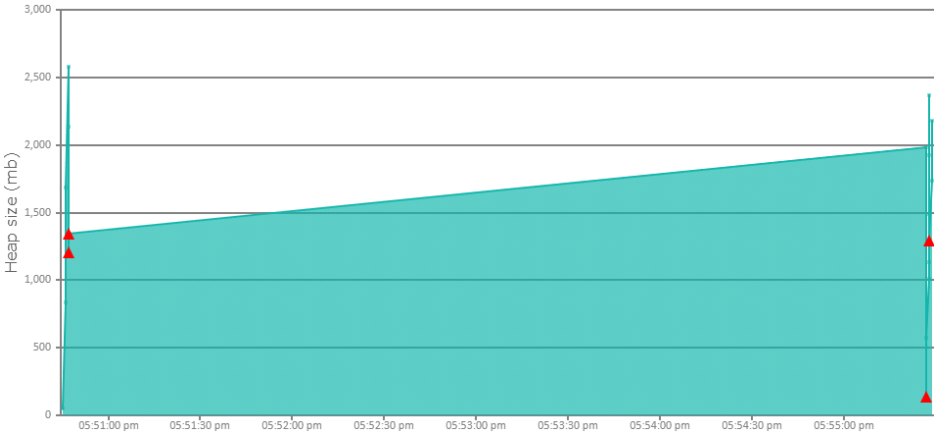
[그림 3-3] 2GB로 할당된 Z 컬렉터의 힙 메모리 사용량

[표 3-5]의 실험 1-2은 힙 크기 4GB로 할당된 메모리에서의 Tomcat WAS 프로세스의 가비지 컬렉션의 정지 시간 측정 결과이다. Parallel 컬렉터로 적용 시 평균 가비지 컬렉션 시간은 68.9ms 소요되었고 Z 컬렉터로 적용한 경우 평균 0.0141ms가 소요되었다. 최대 가비지 컬렉션 시간은 Parallel 컬렉터가 130ms로 G1, Z 컬렉터보다 오래 걸려 수행되었다. 가비지 컬렉션 횟수의 경우, G1 컬렉터가 226회로 가장 많이 수행되었으며 Parallel 컬렉터가 18회로 가장 적게 수행되었다. Scouter로 측정된 각 컬렉터의 애플리케이션의 평균 응답시간은 Parallel 컬렉터가 645ms로 가장 오래 측정되었다. G1 컬렉터의 응답시간은 9ms로 측정되었고 Z 컬렉터의 응답시간은 2GB, 4GB 환경에서 모두 3ms~4ms 정도로 큰 차이를 보이지 않았다.

[표 3-5] 힙 크기 4GB에서 가비지 컬렉션 결과

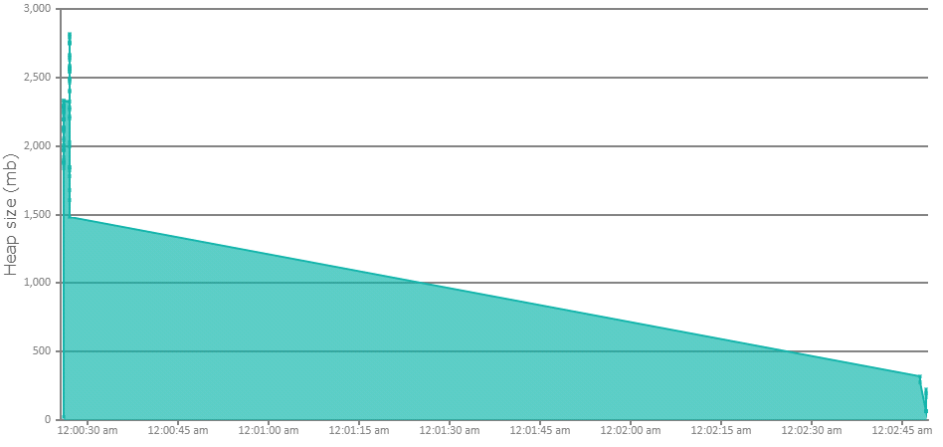
구분	Garbage Collector	Heap Size	Avg GC Pause Time	Max GC Pause Time	GC Count	App평균 응답시간
실험 1-2	Parallel	4GB	68.9ms	130ms	18	645ms
	G1	4GB	14.5ms	20.7ms	226	9ms
	Z	4GB	0.0141ms	0.0360ms	81	4ms

[그림 3-4]는 4GB 할당된 메모리에서 Parallel 컬렉터 적용 시 힙 메모리 사용률을 나타낸 그래프이다. 최대 2.6GB 메모리 사용률을 보였으며 재차 가비지 컬렉션이 일어난 구간에서도 2.4GB 사용률을 보였다.



[그림 3-4] 4GB로 할당된 Parallel 컬렉터의 힙 메모리 사용량

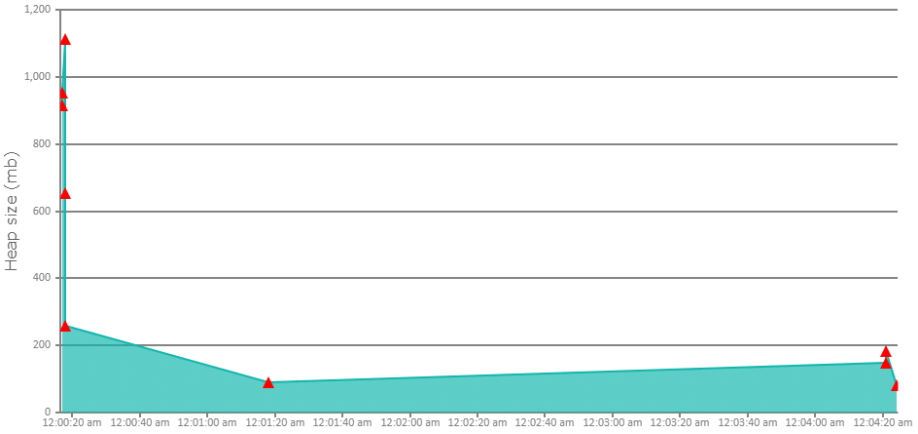
[그림 3-5]에서는 G1 컬렉터에 할당된 최대 2.8GB에 해당하는 메모리 사용률을 보였으며 가비지 컬렉션이 진행되어 점차 가용 메모리를 확보하였다.



[그림 3-5] 4GB로 할당된 G1 컬렉터의 힙 메모리 사용량

[그림 3-6]의 Z 컬렉터의 경우 최대 1.1GB 이하로 힙 메모리가 관리되고 있다. 부하 발생시 Parallel 컬렉터, G1 컬렉터와 비교하여 약 2배의 메모리

공간을 여유 공간으로 확보하였다.



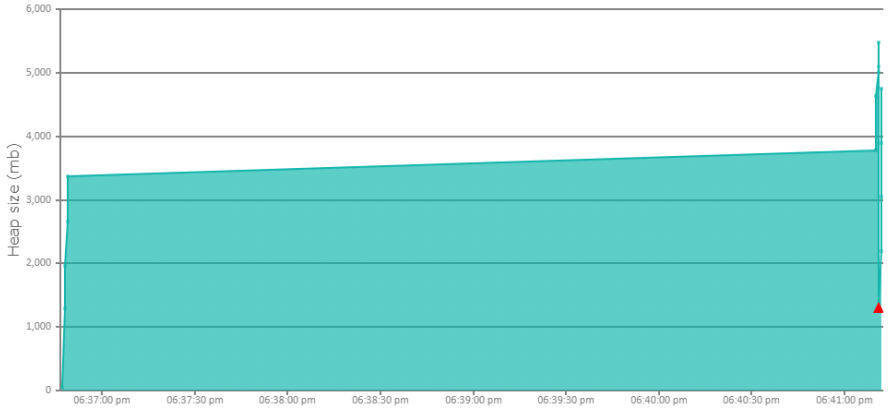
[그림 3-6] 4GB로 할당된 Z 컬렉터의 힙 메모리 사용량

[표 3-6]의 실험 1-3은 힙 크기 8GB로 할당된 메모리에서의 Tomcat WAS 프로세스의 가비지 컬렉션의 정지 시간 결과이다. Parallel 컬렉터로 적용 시 평균 가비지 컬렉션 시간은 155ms 소요되었고 Z 컬렉터로 적용한 경우 평균 0.0156ms가 소요되었다. 애플리케이션의 평균 응답시간은 Z 컬렉터가 220ms로 Parallel, G1 컬렉터와 비교하여 가장 적게 측정되었다.

[표 3-6] 힙 크기 8GB에서 가비지 컬렉션 결과

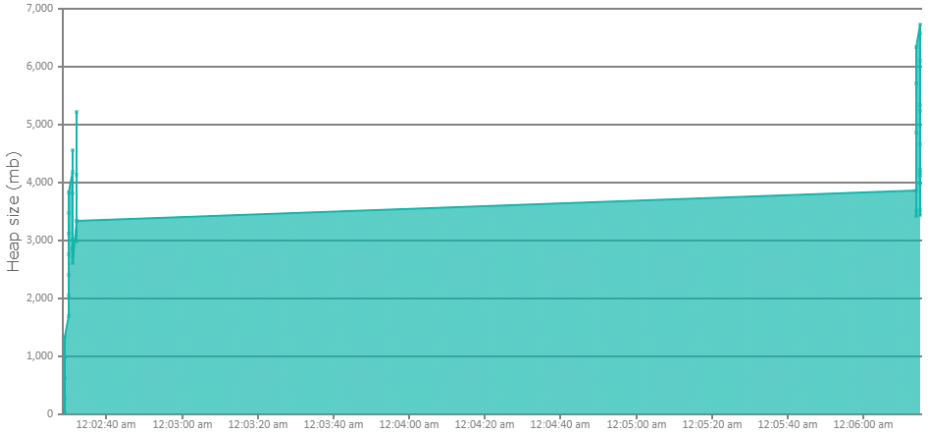
구분	Garbage Collector	Heap Size	Avg GC Pause Time	Max GC Pause Time	GC Count	App평균 응답시간
실험 1-3	Parallel	8GB	155ms	300ms	15	561ms
	G1	8GB	52.1ms	152ms	59	484ms
	Z	8GB	0.0156ms	0.0610ms	99	220ms

[그림 3-7]는 8GB 할당된 메모리에서 Parallel 컬렉터 적용 시 가비지 컬렉션 후의 힙 메모리의 사용률이다. 평균 3.5GB 메모리 사용을 보이다가 최대 5.5GB 메모리까지 사용되었다.



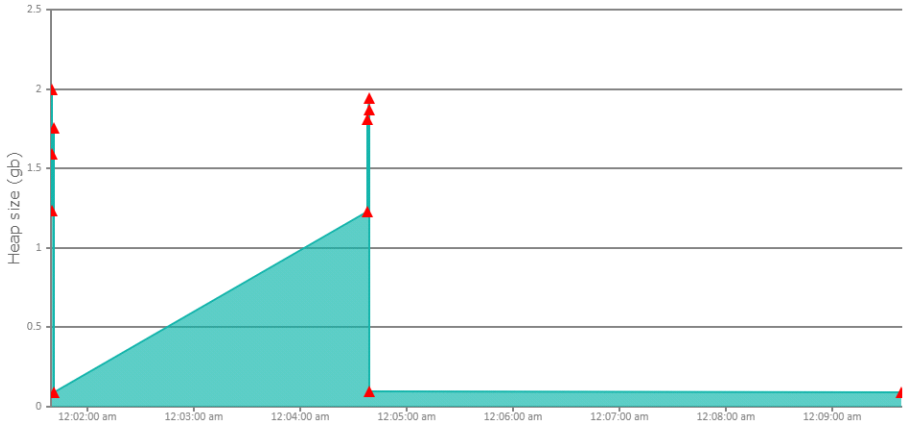
[그림 3-7] 8GB로 할당된 Parallel 컬렉터의 힙 메모리 사용량

[그림 3-8]의 G1 컬렉터의 경우에서도 Parallel 컬렉터와 비교와 유사한 최대 6.8GB에 해당하는 메모리 사용률을 보였다.



[그림 3-8] 8GB로 할당된 G1 컬렉터의 힙 메모리 사용량

[그림 3-9]의 Z 컬렉터의 경우 최대 2GB 이하로 힙 메모리가 관리되고 있다. 부하 발생시 Parallel 컬렉터, G1 컬렉터와 비교하여 약 3배의 메모리 공간을 여유 공간으로 확보하였다.



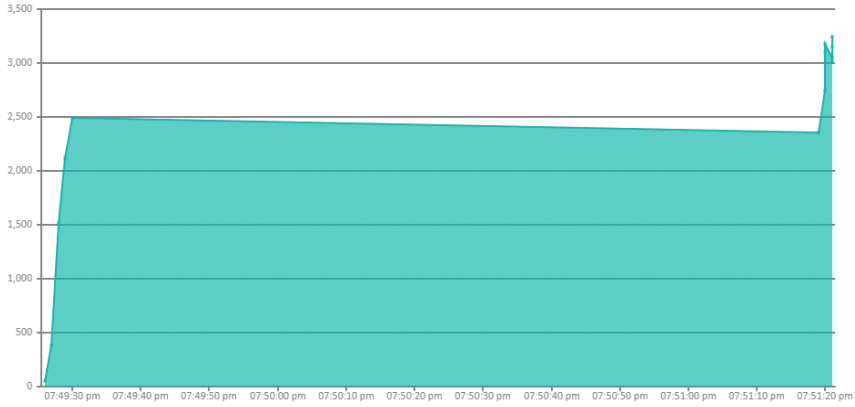
[그림 3-9] 8GB로 할당된 Z 컬렉터의 힙 메모리 사용량

[표 3-7]의 실험 1-4은 힙 크기 16GB로 할당된 메모리에서의 Tomcat WAS 프로세스의 가비지 컬렉션의 정지 시간 측정 결과이다. 애플리케이션의 평균 응답시간은 Parallel, G1, Z 컬렉터 모두 300~400ms 이내로 측정되었다.

[표 3-7] 힙 크기 16GB에서 가비지 컬렉션 결과

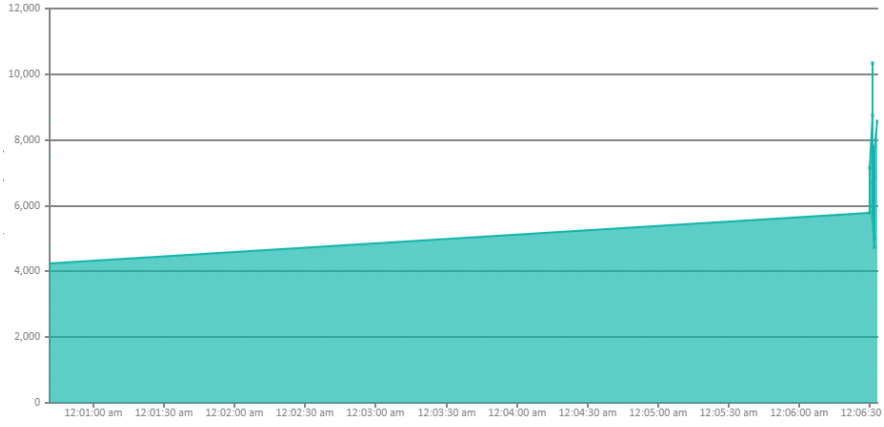
구분	Garbage Collector	Heap Size	Avg GC Pause Time	Max GC Pause Time	GC Count	App평균 응답시간
실험 1-4	Parallel	16GB	111ms	170ms	11	315ms
	G1	16GB	71.9ms	244ms	38	413ms
	Z	16GB	0.0164ms	0.0410ms	63	316ms

[그림 3-10]는 16GB 할당된 메모리에서 Parallel 컬렉터 적용 시 힙 메모리 사용률을 나타낸다. 평균적으로 2.5GB 메모리가 사용되었으며 최대 3.3GB 메모리까지 사용되었다.



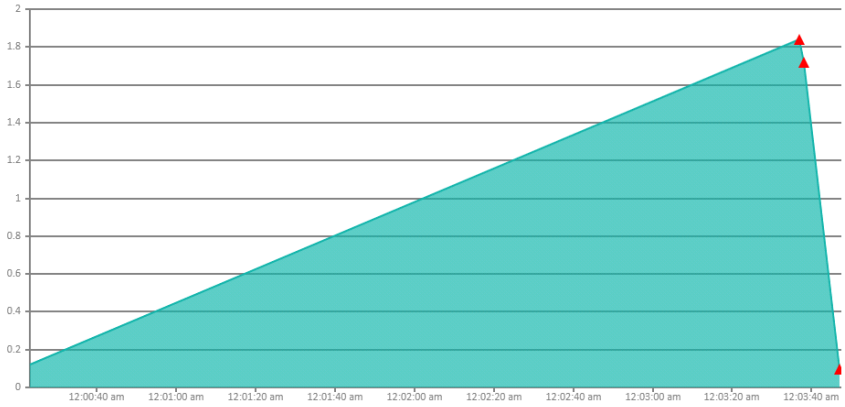
[그림 3-10] 16GB로 할당된 Parallel 컬렉터의 힙 메모리 사용량

[그림 3-11]에서 G1 컬렉터의 경우 최대 10.5GB에 해당하는 메모리가 사용되었다. 이는 Parallel 컬렉터의 최대 메모리 사용률의 약 3배에 해당하는 메모리 크기이다.



[그림 3-11] 16GB로 할당된 G1 컬렉터의 힙 메모리 사용량

[그림 3-12]의 Z 컬렉터의 경우 최대 1.9GB 이하로 힙 메모리가 관리되고 있다. G1 컬렉터보다 최대 메모리 사용률의 약 5배 해당하는 여유 공간이 가용 공간으로 확보되었다.



[그림 3-12] 16GB로 할당된 Z 컬렉터의 힙 메모리 사용량

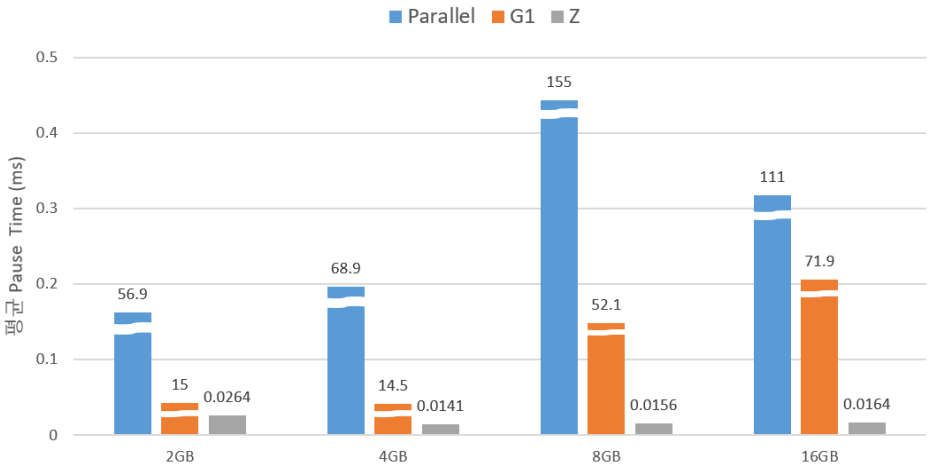
[표 3-8]는 실험1-1, 실험1-2, 실험 1-3, 실험 1-4 결과를 2GB, 4GB, 8GB, 16GB 순서로 가비지 컬렉터 별로 정리한 내용이다. 같은 크기의 메모리가 할당된 경우에는 Jmeter의 쓰레드 수를 달리하여 정지 시간과 애플리케이션의 응답시간을 측정하였다. 쓰레드 수가 커진다는 것은 가비지 컬렉터가 적용이 된 Tomcat WAS 프로세스의 부하가 증가한다는 의미로 평균 정지 시간, 가비지 컬렉션 횟수, 애플리케이션의 평균 응답 시간이 모두 증가하는 결과를 보였다. 할당되는 힙 메모리 크기가 커질수록 객체를 정리하는 공간의 범위도 넓어지게 되어 가비지 컬렉션 정지 시간이 증가하게 된다. 또한 힙 메모리 크기가 크게 할당되는 경우에 Z, G1, Parallel 컬렉터 순서로 애플리케이션 평균 응답시간도 적게 소요되었다.

[표 3-8] 실험 유형별 애플리케이션 평균 응답시간

구분	Garbage Collector	Heap Size	Thread	Avg GC Pause Time	Max GC Pause Time	GC 수	App 평균 응답시간
1	Parallel	2GB	100	56.9ms	170ms	13	4ms
2	Parallel	4GB	100	16.7ms	30ms	3	3ms
3	Parallel	4GB	200	68.9ms	130ms	18	645ms
4	Parallel	8GB	100	25ms	40ms	2	8ms
5	Parallel	8GB	200	20ms	60ms	3	4ms

6	Parallel	8GB	400	155ms	300ms	15	561ms
7	Parallel	16GB	100	40ms	40ms	1	10ms
8	Parallel	16GB	200	35ms	50ms	2	19ms
9	Parallel	16GB	400	60ms	140ms	3	14ms
10	Parallel	16GB	800	114ms	190ms	5	270ms
11	Parallel	16GB	1600	111ms	170ms	11	315ms
12	G1	2GB	100	15ms	64.1ms	550	4ms
13	G1	4GB	100	8.85ms	28.8ms	10	2ms
14	G1	4GB	200	14.5ms	20.7ms	226	9ms
15	G1	8GB	100	12.5ms	55.8ms	7	1ms
16	G1	8GB	200	16.3ms	33.5ms	7	62ms
17	G1	8GB	400	52.1ms	152ms	59	484ms
18	G1	16GB	100	20.1ms	24.0ms	3	2ms
19	G1	16GB	200	44.6ms	94.1ms	6	9ms
20	G1	16GB	400	85.3ms	229ms	13	7ms
21	G1	16GB	800	99ms	178ms	33	564ms
22	G1	16GB	1600	71.9ms	244ms	38	413ms
23	Z	2GB	100	0.0264ms	0.0596ms	117	3ms
24	Z	4GB	100	0.0150ms	0.0790ms	72	7ms
25	Z	4GB	200	0.0141ms	0.0360ms	81	4ms
26	Z	8GB	100	0.0130ms	0.0180ms	9	15ms
27	Z	8GB	200	0.0125ms	0.0250ms	18	23ms
28	Z	8GB	400	0.0156ms	0.0610ms	99	220ms
29	Z	16GB	100	0.0133ms	0.0250ms	9	6ms
30	Z	16GB	200	0.0143ms	0.0290ms	9	11ms
31	Z	16GB	400	0.0172ms	0.0340ms	18	5ms
32	Z	16GB	800	0.0179ms	0.0490ms	54	371ms
33	Z	16GB	1600	0.0164ms	0.0410ms	63	316ms

[그림 3-13]는 2GB, 4GB, 8GB, 16GB에서 각 컬렉터의 평균 정지 시간 측정 결과이다. 2GB에서는 Z, G1, Parallel 컬렉터 순서로 평균 정지 시간과 최대 가비지 컬렉션 정지 시간이 적게 소요되었다. 4GB로 할당된 쓰레드 수 200의 동일한 부하에서는 Z 컬렉터는 0.0141ms로 가장 적은 정지 시간이 소요되었다. Z 컬렉터는 쓰레드 수 100, 400, 800, 1600 에서도 Parallel, G1 컬렉터과 비교하여 정지 시간이 적게 소요되었다.



[그림 3-13] 힙 메모리 2, 4, 8, 16GB에서 각 컬렉터의 평균 정지 시간

Ⅳ. 결론

4.1 결론

4.1.1 결론 및 시사점

본 연구에서는 자바 LTS 버전에 해당하는 8, 11, 17의 각 가비지 컬렉터에 대해 분석하고 실험을 통해 애플리케이션의 성능에 영향을 미치는 정지 시간 및 평균 응답 시간을 확인하였다. 구현된 결과는 Apache Jemeter 5.5와 Scouter 2.17, 2.18을 동일한 부하를 주어 실험을 진행하였다. 결과로는 G1 컬렉터의 경우 Parallel 컬렉터를 사용 했을 때보다 정지 시간이 최대 79% 줄어들었고 평균 61% 줄어들었다. Z 컬렉터의 경우에는 2GB에서 8GB까지 다양한 힙 메모리 크기가 할당 된 경우에도 크기에 관계없이 정지 시간이 평균 0.03ms 이하로 가장 적은 시간이 소요되었다.

시스템에 구성된 애플리케이션의 성능에 영향을 미치는 요소는 다양하다. 본 연구에서 다루는 부분은 가비지 컬렉터 적용 관점에서 실험 구성하였고 정지 시간과 애플리케이션의 응답 시간을 측정하였다. 따라서 실제 운영되고 있는 자바 시스템 환경에서는 하드웨어의 성능, 네트워크의 성능, 업무 시스템에 성격, 객체 생성 주기에 따른 제약 사항이 존재한다.

생명 주기가 짧고 빈번한 객체 생성이 필요한 자바 애플리케이션 시스템 환경에서는 Parallel 컬렉터보다 G1, Z 컬렉터를 적용하면 가비지 컬렉션 과정에서의 정지 시간이 줄어들게 된다. 자바 8 시스템 환경에서는 애플리케이션 소스의 재 컴파일 없이도 기본 컬렉터인 Parallel 컬렉터에서 G1 컬렉터로 변경하여 컬렉터 적용이 가능하다. 마찬가지로 자바 11 기반 시스템 환경에서는 기존 운영하고 있는 업무의 애플리케이션 소스의 재 컴파일 없이도 기본 컬렉터인 G1 컬렉터에서 Z 컬렉터로 변경하여 컬렉터 적용이 가능하다. 컬렉터 적용 방안을 통해 힙 메모리 공간이 부족하여 발생하는 OOME(Out Of Memory Error) 현상이 줄어들게 되므로 메모리

적체 현상을 해결할 수 있다. 또한 줄어든 정지 시간만큼 애플리케이션의 응답 시간이 개선되고 처리량 향상을 기대할 수 있다. 차세대 또는 고도화 프로젝트의 애플리케이션 소스나 현재 운영되는 시스템의 애플리케이션 소스 측면의 변경사항 없이도 사용자의 불편함을 해소하고 성능과 안전성을 증가시키는 방안 중의 하나가 되겠다.

4.1.2 연구의 한계점 및 향후 연구방향

최근 온-프레미스(On-premise) 환경에서 클라우드 환경으로 시스템 도입이 많아지면서 MSA(MicroService Architecture) 지향 개발 방식의 전환이 진행되고 있다. MSA 개발 방식은 큰 서비스를 작은 서비스로 나누고 개별 배포를 가능하게 하는 방식이다. 자바 시스템 또한 마찬가지로 구조적으로 경량화를 추구하는 MSA 기본 사상에 따라 할당되는 힙 메모리 크기의 할당이 작아져야만 한다. 본 연구의 실험 결과에서 Z 컬렉터 적용 시 힙 메모리 크기가 커질수록 애플리케이션의 성능 개선 기대 효과가 커지는 결과를 보였다. MSA 환경의 2GB 미만(500MB, 1GB)의 작은 힙 메모리 크기가 할당되는 자바 시스템의 경우에도 가비지 컬렉터에 따라 정지 시간과 애플리케이션 응답시간에 비교할 때 어떤 결론이 도출되는지 후속 연구가 필요하다. 작은 메모리 환경에서도 Z 컬렉터의 정지 시간 개선 효과가 확인된다면 각 기능 단위로 나누어진 애플리케이션의 응답 시간의 향상을 가져올 수 있게 될 것이며 향후 시스템의 접속자 수 증가나 데이터 증가에도 보다 높은 품질이 확보될 수 있을 것으로 기대한다.

참 고 문 헌

1. 국내문헌

기 석. (2011). 동시 테스팅을 이용한 애자일 방법론의 스크럼 개발 프로세스. 고려대학교 정보통신대학원 석사학위논문.

김태용. (2016). hp-ux 에서 동작하는 Java Virtual Machine 의 Garbage Collection 사례들 패턴 분석을 통한 성능향상 연구. 고려대학교 컴퓨터정보통신대학원 석사학위논문.

김형원. (2011). 사용자 관점의 애플리케이션 성능평가 지표에 관한 연구. 숭실대학교대학원 박사학위논문.

민철기, 염헌영, 한혁. (2021). “멀티 코어를 위한 효율적인 JAVA Full GC Compact 방법” 정보과학회 컴퓨팅의 실제 논문지 제 27 권 2021.3, 129-136.

양병선. (2017). Java virtual machine optimizations for Java and dynamic languages, 서울대학교 전기컴퓨터공학부, 박사학위논문.

이상윤, 원희선, 최병욱. (2006). EVM에서의 자바 동적 메모리 관리기 및 쓰레기 수집기의 구현 및 성능 분석. 정보처리학회논문지 A 제 13-A 권 제4호.

이용수. (2002). 임베디드 자바가상머신을 위한 가비지 콜렉션 설계 및 구현. 한국정보과학회 논문집, 제29권 제2호.

이은화, 윤성대 (2004). 자바 플랫폼에서 효율적인 쓰레기 수집기. 부경대학교 산업대학원 석사학위논문

최인범. (2004). KVM 의 효율적인 힙 메모리 관리를 위한 가비지 콜렉션의 설계 및 구현. 한국정보처리학회 논문집, Vol.11 No.2

최형규, 문수묵. (2008). 메모리가 적은 자바 시스템을 위한 자동 동적 메모

리 관리 기법. 정보과학회논문지 제35권 8호.
최형규. (2014). 가상머신의 메모리 관리 최적화. 서울대학교 전기컴퓨터공학부, 박사학위논문.
한판암, 김정종. (1996). “자바의 객체지향 프로그래밍 특성” 경남대학교 공업기술연구소 연구논문집 Vol14 No.1, 81-87

2. 국외문헌

Java Platform Standard Edition 8 Documentation (2023). Oracle
Monica Beckwith. Garbage First Garbage Collector Tuning (2013). Oracle
Oracle, “HotSpot Virtual Machine Garbage Collection Tuning Guide - White Paper”
Per Lidén & Stefan Karlsson. The Z Garbage Collector (2018). Oracle

ABSTRACT

A Study on the Application of Java Virtual Machine Garbage Collector for Improving Application Performance

- Focusing on Open Source Software OpenJDK 8, 11, 17 -

Song, Seung-Heon

Major in Smart Convergence Technology Consulting

Dept. of Smart Convergence Consulting

The Graduate School of Knowledge Service Consulting

Hansung University

In modern society, digital innovation is accelerating due to the development of the Internet, and the quality of life has been improved by accessing various services through applications.

Moreover, in the age of COVID-19, the proportion of non-face-to-face communication has increased and this inconvenience has also increased. This means that as applications are modernized and future-oriented development directions are presented, we are faced with an era of competition to provide new services more quickly.

However, due to the rapidly increasing number of users, there

has been a problem of delaying the response of the application. As one of the many causes, the demand for improvement in memory has been greatly increased. The process of implementing an increasingly complex system to respond to the needs of users has been repeated. From the application development stage, the difficulty of developing a system continued, always considering memory problems. It has been 28 years since Java was launched, and it still has a high market share worldwide. Therefore, there are many application services programmed in Java, and user inconvenience is proportionally the highest.

The purpose of this study is to solve the problem of delayed response time due to problems such as memory in the process of using software-implemented applications and to suggest a performance improvement.Methods for optimizing memory performance have long been applied, but old programs have been difficult to implement. However, with Java programming, Garbage Collection was developed as an improved memory management technique. Memory management has improved program performance, and it has an improvement effect that can greatly resolve the complaints of users using the application.

The experiment in this study is most commonly used in the Java project and selected Parallel, G1, and Z garbage collectors in Java 8, 11, and 17 versions corresponding to Long Term Support. In the Tomcat process allocated with different heap memory sizes (2GB, 4GB, 8GB, 16GB), the garbage collection Pause time, GC Count, and the response time were compared and analyzed to implement a garbage collector application method. Apache Jmeter and Scouter were used for evaluation. I would like to study the performance improvement of applications using garbage collectors applied on a case-by-case.

【Keywords】Java Virtual Machine, Garbage Collection, Garbage Collector, Java, Application