저작자표시-비영리-변경금지 2.0 대한민국

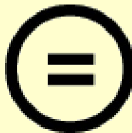
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

코드 공유를 통한 V8 자바스크립트 엔진 메모리 절감

Reducing memory usage by sharing code on V8 JavaScript Engine

2016년 8월

서울대학교 대학원

전기.컴퓨터공학부

서 보 준

코드 공유를 통한 V8 자바스크립트 엔진 메모리 절감

Reducing memory usage by sharing code on V8 JavaScript Engine

지도교수 이 재 진

이 논문을 공학석사학위논문으로 제출함

2016년 4월

서울대학교 대학원

전기.컴퓨터공학부

서보준

서보준의 석사학위논문을 인준함

2016년 6월

위 원 장	문 봉 기	(인)
부위원장	이 재 진	(인)
위 원	권 태 경	(인)

초록

코드 공유를 통한 V8 자바스크립트 엔진 메모리 절감

서보준
서울대학교
전기.컴퓨터공학부 대학원

최근 자바스크립트 엔진은 인터프리팅 방식이 아닌 JIT(Just-In-Time) 컴파일 방식으로 기계어 코드를 생성하여 프로그램을 실행한다. 이 때 생성되는 기계어 코드는 하나의 엔진(프로세스)에서만 사용이 되고 더 이상 필요 없어지는 경우 GC(Garbage Collector)에 의해 지워진다. 여러 개의 엔진(프로세스)이 실행되는 경우, 예를 들어 Chrome 브라우저에서 여러 개의 탭을 생성하여 여러 개의 V8 자바스크립트 엔진이 실행되는 경우, 같은 자바스크립트 코드가 있다면, 같은 기계어 코드가 각각의 프로세스마다 생성되며, 각자의 메모리 공간에 할당된다. 동일한 내용을 중복하여 저장하였기 때문에 이는 메모리 낭비라고 할 수 있다.

본 연구에서는 이러한 동일한 기계어 코드를 여러 V8 자바스크립트 엔진(프로세스)에서 공유함으로써 메모리 사용량을 줄이는 것을 제안한다. 기계어를 공유하기 위해서 V8 런타임에서 컴파일된 코드를 저장하는 방식을 변경하였고, JIT 컴파일러도 수정하였다. 공유 메모리 영역에 바이너리를 할당하고, GC를 적용하여 해제하는 방식을 구현하였다.

Firebase, Kendo 등의 자바스크립트 라이브러리를 사용하는 어플리케이션으로 실험하였고, 4개의 어플리케이션이 동시에 실행되는 상황에서 코드의 메모리 사용량을 기존 대비 약 29.6% 줄였다.

주요어: 자바스크립트, V8, 코드 공유, 공유 메모리

학번: 2013-23118

목차

1. 서론. 1
2. 배경. 3
2.1 V8 Heap 과 GC. 3
2.2 JIT compilation. 6
2.3 컴파일 결과를 위한 자료구조. 8
3. 본론. 12
3.1 Binary chunk. 12
3.2 V8 HeapObject 구조 변경. 14
3.3 V8 JIT 컴파일러 변경. 16
3.4 공유 메모리 영역의 구조. 17
3.5 공유 메모리 영역 GC. 21
4. 실험. 24
4.1 실험 환경. 24
4.2 실험 방법. 25
4.3 실험 결과. 29
5. 결론. 33
참고문헌. 34

표 목차

[표 1]: 기계어에 대한 Code 객체의 주소를 저장한 표. 16
[표 2]: Firebase 어플리케이션. 24
[표 3]: Kendo 어플리케이션. 25
[표 4]: 동시에 실행하는 조합. 25

그림 목차

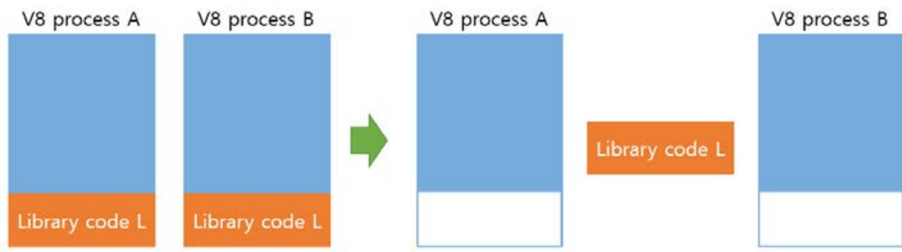
[그림 1]: (좌) 현재의 V8 (우) 본 연구의 구현. 2
[그림 2]: V8 Heap 과 HeapObject 구조. 4
[그림 3]: Incremental marking 과 concurrent and parallel sweeping. 6
[그림 4]: Fast compilation and concurrent optimization. 7
[그림 5]: 컴파일 결과를 위한 자료구조. 9
[그림 6]: Code 객체의 구조. 10
[그림 7]: OOL(Out-Of-Line) constant pool. 11
[그림 8]: 라이브러리 함수 구별 및 컴파일 방법. 13
[그림 9]: 본 연구의 HeapObject 구조. 15
[그림 10]: 공유 메모리 영역 초기화 순서도. 18
[그림 11]: 공유 메모리 영역 구조. 19
[그림 12]: Minifier 로 인하여 같은 함수의 소스코드가 다른 경우. 21
[그림 13]: 공유 메모리 GC 를 위한 자료 구조. 22
[그림 14]: 공유 메모리 GC 예시. 23
[그림 15]: GC 영향으로 본 연구의 메모리 사용량이 더 많은 상황(예시). 27
[그림 16]: 두 가지 측정방식 비교. 28
[그림 17]: Firebase 프로세스 4 개 실행 시 코드의 메모리 사용량. 28
[그림 18]: Firebase periodic memory. 30
[그림 19]: Kendo periodic memory. 30

[그림 20]: Firebase after GC. 31
[그림 21]: Kendo after GC. 31

1. 서론

최근 자바스크립트 엔진의 비약적인 성능 향상으로 많은 어플리케이션이 자바스크립트로 만들어지고 있다. 자바스크립트 어플리케이션의 증가는 안정성과 개발 편의성을 위하여 자바스크립트 라이브러리에 대한 요구로 이어졌다. 이러한 요구에 발 맞추어 많은 자바스크립트 라이브러리가 등장하고 있다. Angular.js, React, Three.js 등등이 새로 등장하여서 많이 사용되고 있으며, 전통적인 자바스크립트 라이브러리인 JQuery는 여전히 많이 사용되고 있다[1, 2, 3, 4, 5]. 자바스크립트 개발자들은 이러한 라이브러리를 이용하여 더욱 쉽게 믿을 수 있는 어플리케이션을 개발할 수 있게 되었다.

어플리케이션에서 라이브러리를 많이 사용한다는 것은 자바스크립트 소스코드 중에서 라이브러리 코드가 차지하는 비율이 커진다는 것을 의미한다. 현재 자바스크립트 엔진은 라이브러리 코드와 어플리케이션 코드를 구별하지 않고, 모두 JIT 컴파일을 통해 자신의 메모리에 올려두고 실행한다. 여러 개의 엔진(프로세스)이 동시에 실행되는 경우 라이브러리 코드에 대해서는 동일한 기계어 코드가 생성되는데, 구별하지 않기 때문에 각 어플리케이션의 메모리 공간에 동일한 기계어 코드가 올라가게 된다. 이를 공유하는 경우 메모리를 절약할 수 있다. 본 연구에서는 [그림 1]과 같이 V8 자바스크립트 엔진에서 중복되는 라이브러리 코드를 공유함으로써 메모리 사용량을 줄이는 것을 제안한다.



[그림 1] (좌) 현재의 V8 (우) 본 연구의 구현

2. 배경

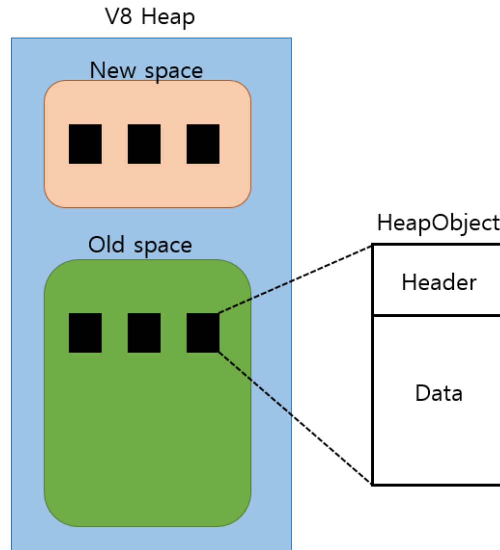
2.1 V8 Heap과 GC

자바스크립트는 프로그래머가 메모리를 직접 관리하지 않고, GC에 의해 메모리를 관리한다[6, 7, 8]. GC는 성능상에 큰 영향을 미치기 때문에 여러 자바스크립트 엔진은 다양한 방법으로 GC를 구현하여 성능을 유지하고자 노력하고 있다[8, 9].

V8에서 관리하는 메모리 공간을 Heap이라 부른다. 이는 전통적인 컴퓨터 공학에서 말하는 Heap 메모리 영역을 의미하는 것이 아니라 V8 자바스크립트 엔진에서 GC를 이용하여 관리하는 메모리 공간을 의미한다[8].

자바스크립트는 타입이 동적으로 결정되는 특성이 있다[6, 7]. 가령 숫자를 저장했던 변수에 문자열을 저장할 수 있다. 이러한 동적 타입 특성은 자바스크립트 프로그램이 좋은 성능을 달성하는 장애 요인인데, 여러 자바스크립트 엔진은 다양한 방법으로 이 문제를 해결하고 있다[8, 10].

V8 자바스크립트 엔진은 [그림2]와 같이 자바스크립트의 모든 타입을 헤더와 데이터로 이루어진 형태로 저장하며 이러한 형태를 HeapObject라 부른다[8, 11]. 헤더에는 데이터의 길이 등의 정보를 담고, 데이터에는 저장하려는 값을 담는다. 가령 문자열의 경우 헤더에는 문자열의 길이 등과 같은 정보가 저장되고, 데이터에는 문자열이 저장된다. 이것은 다양한 동적 타입 특성을 효과적으로 처리하기 위함이다[12]. V8 Heap에는 이러한 HeapObject들이 저장되고 GC에 의해 지워진다.



[그림 2] V8 Heap과 HeapObject 구조

V8의 GC는 generational GC이다[8, 12, 13]. 객체의 life time은 짧거나 길고, 보통인 경우가 거의 없으며, 짧은 경우가 대부분인 것을 알 수 있다[14]. 이것은 생성 된지 얼마 되지 않은 객체는 금방 garbage가 될 확률이 높다는 것을 의미한다[15, 16]. Generational GC는 garbage가 될 확률이 높은 새로 생성된 객체들과 garbage가 될 확률이 낮은 오래된 객체들을 따로 보관하고, 각각 다른 GC알고리즘을 적용하여 GC에 의한 오버헤드를 최소화 하는 방식이다. V8은 [그림 2]와 같이 Heap을 나누어, 새로 생성된 HeapObject를 저장하는 공간을 New space, 오래된 HeapObject를 저장하는 공간을 Old space라 부른다.

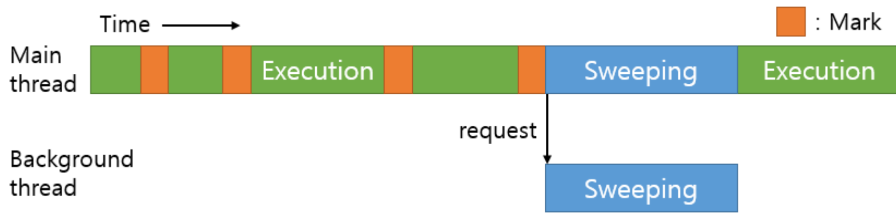
New space 영역은 garbage의 양이 많을 것이므로 GC가 자주 실행되어야 한다. 하지만 잦은 GC의 호출은 성능 저하를 낳는다. 일반적으로 GC의 속도는 공간의 크기에 영향을 받는데, 이러한 성능 저하를 줄이기 위하여 보통 New space의 크기는 작다. 따라서 자주 호출되더라도 빠르게 끝날 수 있도록 구성된다. V8의

new space GC의 경우 Cheney's algorithm[12, 17]이 적용되었고, Scavenge라 불리며, 알고리즘은 다음과 같다.

New space 를 동일한 크기의 to-space 와 from-space 로 나눈다. 모든 할당은 to-space 에 하고, to-space 가 꽉 차면 GC 가 발생한다. GC 과정은 먼저 to-space 와 from-space 를 바꾼다. 그리고 from-space 에서 live object 들은 to-space 로 이동시킨다. 2 번째 살아남은 live object 의 경우 오래 존재하는 object 로 간주하여, old space 로 이동시킨다[12, 17].

V8 의 old space GC 의 경우 mark-sweep 알고리즘이 적용되었고, Full GC 라 불린다. DFS(Depth First Search)를 통하여 모든 live object 를 mark 하고, marking 되지 않은 메모리 영역을 sweep 하는 방식이다. Old space 의 크기가 커질수록 GC 에 걸리는 시간이 늘어난다. GC 는 프로그램의 실행을 멈추고 진행하기 때문에 사용자로 하여금 프로그램이 중간에 멈추는 듯한 현상을 겪게 한다. V8 은 이러한 단점을 보완하기 위하여, Full GC 에 incremental marking 기법과 concurrent and parallel sweeping 기법을 적용하였다[12].

Incremental marking은 [그림 3]와 같이 실행 도중 미리 조금씩 marking을 해 놓는 방식으로 전체 marking 시간은 줄어들지 않지만 한번에 오래 멈추지 않기 때문에 사용자가 멈추는 시간을 느끼기 어렵게 만들어준다. Concurrent and parallel sweeping 기법은 [그림 3]와 같이 sweep을 background thread와 main thread에서 동시에 진행함으로써 프로그램이 멈추는 시간을 단축시킨다.



[그림 3] Incremental marking과 concurrent and parallel sweeping

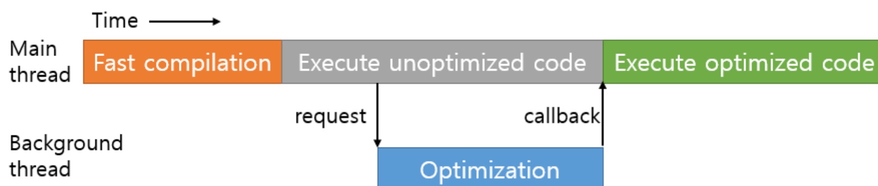
2.2 JIT(Just-In-Time) compilation

자바스크립트는 전통적으로 인터프리팅[18] 방식으로 실행되었다. 인터프리팅 방식이란 컴퓨터 공학에서 사용되는 여러 실행 방식 중 하나로, 일반적으로 전체 소스코드를 컴파일하여 기계어 코드를 생성하고, 생성된 기계어 코드를 실행시키는 방식과 달리, 소스코드를 한 줄씩 직접 실행하는 방식이다. 크게 세 단계인데, 먼저 실행하려는 소스코드를 해석 및 분석(parse)하고, 중간 단계의 언어(IR: Intermediate Representation)로 변경하고, 이 중간 언어를 실행시킬 수 있는 미리 컴파일된 기계어 코드를 실행하는 방식이다. 이러한 여러 단계로 인하여 인터프리팅 방식은 기계어 코드를 직접 실행시키는 것에 비하여 느리다.

인터프리팅 방식의 성능상의 한계로 인하여 최근의 자바스크립트 엔진들은 JIT(Just-In-Time) compilation 기술을 적용하였다[19]. JIT compilation 방식이란 실행 중에 소스코드(자바스크립트 코드)를 기계어 코드로 컴파일하고, 생성된 기계어 코드를 실행하는 방식을 말한다. 한 번 컴파일이 되면 기계어 코드를 실행시킬 수 있으므로 여러 번 실행되는 코드에 대해서는 더 빠르게 실행되지만,

컴파일 시간이 실행시간에 포함되기 때문에 한 번 또는 거의 실행되지 않는 코드에 대해서는 이득이 없다. 이러한 단점을 보완하기 위하여 V8 자바스크립트 엔진에서는 Lazy compilation 기법과 Fast compilation and concurrent optimization 기법을 적용하였다[20, 21]. Lazy compilation 기법은 함수 단위로 컴파일을 하며, 함수가 사용될 때 컴파일을 하고 이미 컴파일된 함수는 바로 실행하는 것을 의미한다. 이로 인하여 실제로 한번도 실행되지 않는 함수는 컴파일하지 않으므로 불필요한 컴파일 오버헤드를 줄임으로 시간과 공간의 낭비를 줄인다.

Fast compilation and concurrent optimization 기법은 컴파일 시간을 단축하기 위한 방법으로 [그림 4]와 같이 처음 컴파일 할 때에는 전통적인 컴파일러에서 적용하는 최적화(예: loop invariant code motion, dead code elimination 등등)를 하지 않은 기계어 코드를 생성하고, 프로파일링을 통하여 많이 실행되는 함수에 대하여 concurrent optimize 하는 방식을 말한다. 이를 통하여 처음 컴파일로 인한 오버헤드를 단축시킬 수 있고, 최적화는 오래 걸리지만 concurrent하게 진행되기 때문에 오버헤드를 감출 수 있다.



[그림 4] Fast compilation and concurrent optimization

최적화되지 않은 코드는 자바스크립트 함수당 하나씩만 생성되는 반면, 최적화된 코드는 타입에 따라 여러 개 생성된다. 최적화 되지 않은 코드가 하나만 존재 할 수 있는 이유는 최적화 되지 않은

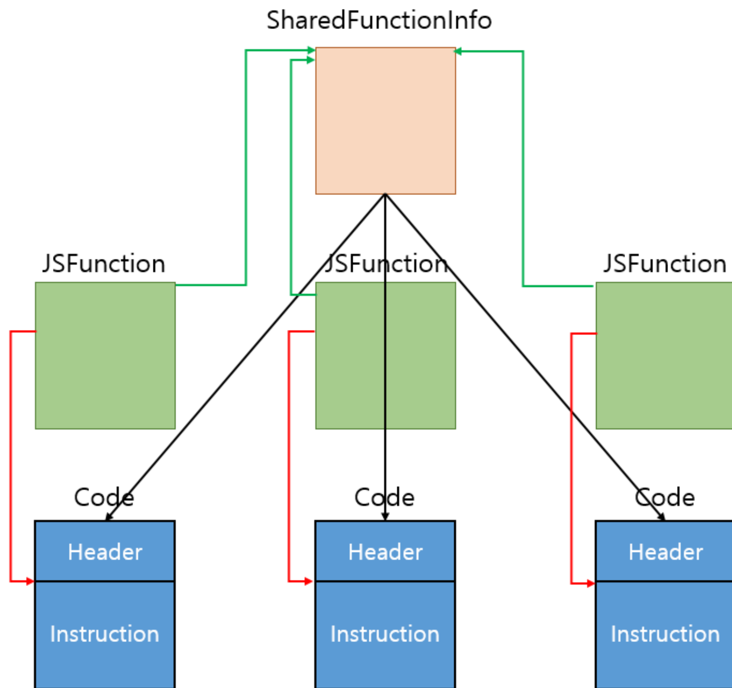
코드는 기계어코드이지만 실행방식이 인터프리팅 방식과 유사하기 때문이다. 최적화되지 않은 코드는 타입이 무엇인지 비교하고, 타입에 따라 연산을 실행시켜주는 기계어 코드를 호출하는 방식으로 실행하기 때문이다. 최적화된 코드는 타입에 따라서 이루어지기 때문에 타입에 따라 여러 개 존재한다. 따라서 V8 에는 자바스크립트 함수 하나당 최적화되지 않은 코드 한 개와 최적화된 코드 여러 개가 존재할 수 있다.

2.3 컴파일 결과를 위한 자료구조

V8은 함수 단위로 컴파일한다. 2.2절에서 설명한 바와 같이 하나의 자바스크립트 함수에 대해서 여러 가지 기계어 코드가 생성될 수 있다. 기계어 코드를 저장하기 위해서는 기계어 코드뿐만 아니라 다른 정보도 같이 저장되어야 하는데, 이러한 정보 중에는 자바스크립트 함수에 따라 결정되는 정보와 그렇지 않은 경우로 나눌 수 있다. 자바스크립트 함수는 동일하기 때문에 자바스크립트 함수에 따라 결정되는 정보는 생성된 여러 기계어 코드들이 서로 공유할 수 있는 정보이다. 이 정보는 SharedFunctionInfo 라는 HeapObject에 저장하며 여러 기계어 코드들이 공유한다. SharedFunctionInfo에는 소스파일 이름(예: jquery.js), 소스파일에서 함수의 시작 위치 등의 정보를 가지고 있다.
기계어 코드마다 필요한 정보는 실행권한이 있어야 하는 정보와 없어도 되는 정보를 분리하여, 두 가지 HeapObject(JSFunction, Code)에 나누어 저장한다. JSFunction에는 기계어의 특성(최적화된 함수인지 여부, 빌트인 함수인지 여부 등)을 저장한다. Code에는

기계어가 저장되어 있고, 헤더에 기계어의 길이 등과 같은 정보를 저장한다.

SharedFunctionInfo, JSFunction, Code 객체가 서로 가리키는 관계는 [그림 5]와 같다. 하나의 자바스크립트 함수마다 하나의 SharedFunctionInfo가 생성되고, 컴파일된 기계어는 Code에 저장되고, 기타 정보는 JSFunction에 저장된다. SharedFunctionInfo에서는 Code 객체의 주소를 리스트의 형태로 보관하여 각각을 모두 가리키고 있다. JSFunction에서는 Code 객체의 주소가 아니라 Code객체의 기계어의 시작주소를 가리키고 있다. 그리고 JSFunction에서는 SharedFunctionInfo를 가리키고 있다.

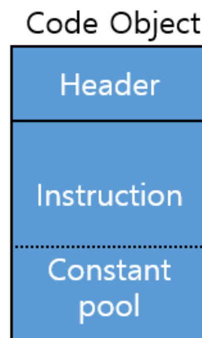


[그림 5] 컴파일 결과를 위한 자료구조

Code 객체 헤더의 크기는 상수 값(64bytes)으로 정해져 있기

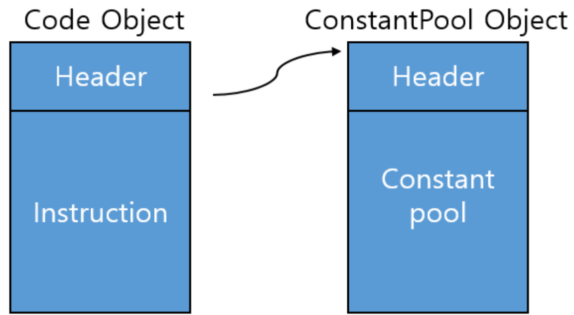
때문에 Code 객체의 시작 주소를 알면 기계어의 시작 주소를 알 수 있고, 기계어의 시작주소를 알면 Code 객체의 시작 주소를 알 수 있다. 그리고 이점을 이용하여 JSFunction에서 Code 객체의 헤더의 정보를 이용하기도 하고, SharedFunctionInfo에서도 기계어의 주소를 이용하기도 한다.

Code 객체는 구조는 [그림 6]과 같이 헤더와 기계어로 구성되어 있다[11]. 헤더에는 기계어의 시작 주소, 기계어의 크기 등과 같은 정보가 저장되어 있고, 기계어 영역에는 기계어 코드를 저장한다. ARM architecture의 경우 indirect addressing만 지원하기 때문에 함수 호출 및 다른 객체 참조를 위한 절대 주소를 constant pool 에 저장하는데 constant pool은 기계어 영역에 포함되어 있다.



[그림 6] Code 객체의 구조

V8에는 OOL constant pool 기능이 있는데, 이 기능은 constant pool이 Code 객체의 기계어 영역에 저장되는 것이 아니라 [그림 7]과 같이 다른 HeapObject인 ConstantPool 객체에 저장되고, Code 객체의 헤더에서 가리키도록 하여 사용한다.



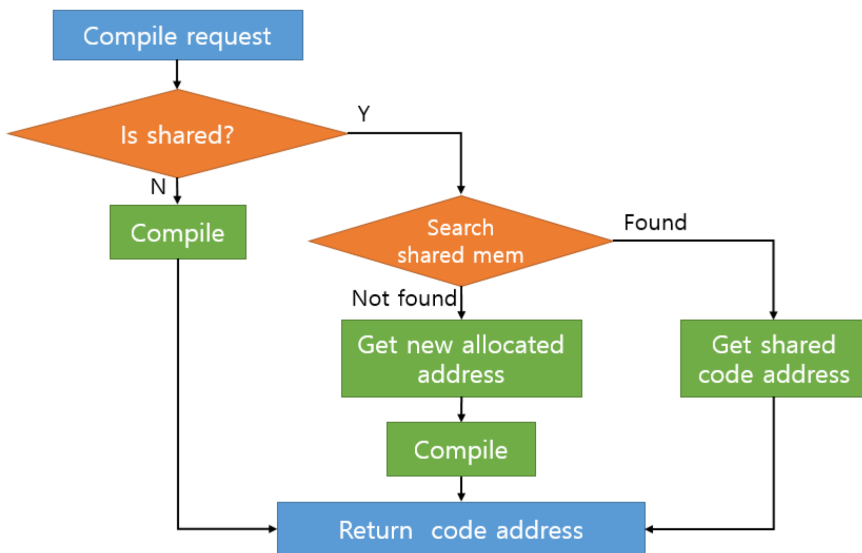
[그림 7] OOL(Out-Of-Line) constant pool

3. 본론

3.1 Binary chunk

본 연구에서는 각 프로세스에서 공유할 수 있는 기계어 코드를 공유하는 것을 그 목적으로 한다. 기계어는 [그림 6]과 같이 Code 객체의 기계어 영역에 있는데, OOL 기능을 켜지 않는 경우, constant pool 영역에는 기계어에서 사용하는 다른 함수 또는 변수의 절대 주소가 저장되어 있다. Constant pool 에서 가리키는 함수 또는 변수는 서로 다른 값이기 때문에 여러 프로세스에서 공유할 수 없다. 가령 프로세스 A 에서 foo()라는 함수의 지역변수 x 에 3 을 대입했다고 해서 프로세스 B 에서 foo()라는 함수의 지역변수 x 의 값이 3 으로 변경되지 않는 것과 같은 이유이다. 따라서 OOL 기능을 이용하여, [그림 7]과 같이 Code 객체의 기계어 영역에서 constant pool 을 분리하고, Code 객체에서 기계어 영역만 공유 메모리 영역에 할당하여 공유하였다. 분리되어 공유 메모리 영역에 할당된 기계어 하나씩을 편의상 binary chunk 라고 부르겠다. Binary chunk 를 공유 메모리 영역에 할당하기 위해서는 binary chunk 가 공유하려는 기계어인지 아닌지 판별 해야 한다. 현재 V8 은 라이브러리를 구별하지 않기 때문에 본 연구에서는 라이브러리 리스트를 미리 작성하고, 이를 이용하여 구별한다. 라이브러리 리스트에는 소스코드의 경로를 저장한다. SharedFunctionInfo 에는 소스코드의 경로가 적혀있는데, 이 값이 리스트에 적혀있는 경우, 라이브러리 함수로 판단한다.

만일 라이브러리가 아니라면 기존의 V8 이 동작하는 것과 같은 방식으로 컴파일하고 실행한다. 만일 라이브러리라면, 이미 컴파일된 기계어 코드가 공유 메모리 영역에 있는지를 확인한다. 만일 아직 컴파일된 기계어 코드가 없다면, 컴파일하여 공유 메모리 영역에 할당한다. 만일 컴파일된 기계어 코드가 있다면, 해당 기계어 코드의 주소 값을 받아서 실행한다. 상기 알고리즘을 순서도로 나타내면 [그림 8]과 같다.



[그림 8] 라이브러리 함수 구별 및 컴파일 방법

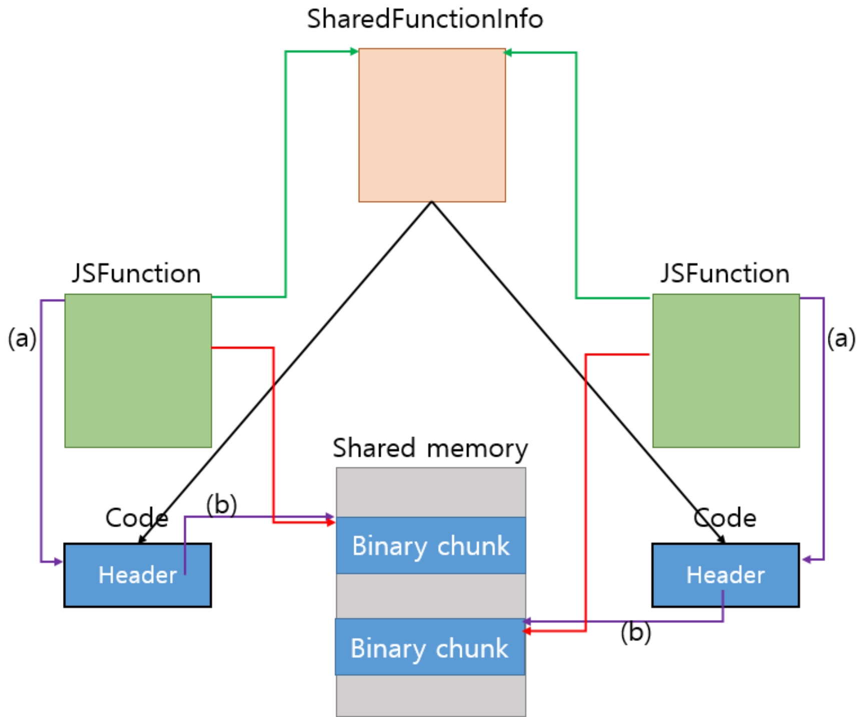
[그림 8]에서 이미 컴파일된 코드가 있는지를 구별하는 방법으로는 hash table 을 이용하였다. Hash table 의 입력 값, 저장 위치 등과 기타 이슈는 3.4 절에 설명한다.

본 연구의 구현에서는 OOL 기능을 사용하기 때문에 다른 프로세스가 생성한 기계어 코드를 공유하여 사용할 수 있지만, constant pool 은 따로 생성해야 한다. Constant pool 은 컴파일

과정이 필요하기 때문에 현재 구현에서는 다른 프로세스가 생성한 기계어 코드를 사용하는 경우에도 컴파일 과정을 거친다.

3.2 V8 HeapObject 구조 변경

2.3 절의 [그림 5]는 본 연구의 구현에 따라 [그림 9]와 같은 형태로 바뀐다. 그런데 2.3 절에서 언급한 바와 같이 Code 객체 헤더의 크기는 상수 값으로 정해져 있기 때문에 SharedFunctionInfo 는 코드 객체의 시작주소에 헤더의 크기를 더하여 기계어의 시작주소를 사용하는 경우가 있고 JSFunction 은 기계어의 시작주소에서 헤더의 크기를 빼서 코드 객체의 시작주소로 사용하는 경우가 있다. 헤더의 크기를 더하거나 빼는 방식으로 접근하는 것은 [그림 9]에서는 잘못된 주소를 참조하는 것을 알 수 있다.



[그림 9] 본 연구의 HeapObject 구조

따라서 본 연구에서는 [그림 9]와 같이 (a)와 (b)가 추가되었다. JSFunction 에서는 Code 객체의 헤더의 주소를 (a)와 같이 저장하고 Code 객체의 헤더의 정보를 이용하는 경우 기존처럼 instruction 의 시작주소에서 헤더의 크기만큼을 빼는 방식이 아닌, (a)를 이용하여 접근한다. Code 객체에는 binary chunk 의 주소를 (b)와 같이 저장하여서, Code 객체에서 binary chunk 를 접근하는 경우 기존처럼 Code 객체의 시작주소에서 헤더의 크기만큼을 더하는 방식이 아닌 (b)를 이용하여 접근한다.

V8 은 자주 쓰이는 함수에 대하여 optimize 하기 위해서 JIT 코드를 실행하다가 V8 런타임으로 넘어와서 프로파일링을 하는데, 이 때 JIT 코드를 실행할 때의 pc(program counter) 레지스터의 값을 이용하여 어떤 Code 객체의 기계어를 실행 중 인지를

파악한다. 이 역시 헤더와 기계어가 분리됨으로 인하여 pc 레지스터의 값으로부터 자신의 Code 객체를 알아오지 못한다. 이를 해결하기 위하여 [표 1]과 같이 기계어의 주소와 그에 대한 Code 객체의 주소를 저장한 표를 따로 관리하였다. 그리고 이를 이용하여 기계어의 주소에서 Code 객체의 주소를 알아낼 수 있도록 하였다.

[표 1] 기계어에 대한 Code 객체의 주소를 저장한 표(예시)

Instruction 주소	Code 주소
0x48D0	0x7E80
0x4620	0x7A20
0x5AE0	0x77A0

3.3 V8 JIT 컴파일러 변경

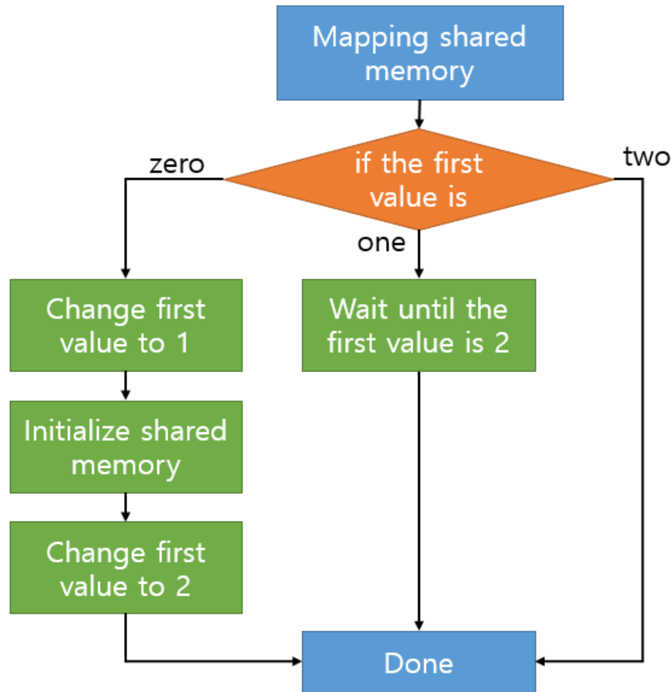
OOL 기능은 2.3 절에서 언급한 바와 같이 코드 객체의 헤더에 ConstantPool 객체의 주소를 저장하여 사용한다. ConstantPool 객체의 주소를 알아야 ConstantPool 에 저장된 값들을 이용할 수 있으므로 함수가 시작될 때 ConstantPool 의 값은 지정된 레지스터(r8)에 저장하고, 그 함수 내에서는 r8 에 상수 값을 더해서 ConstantPool 에 저장된 값들을 이용한다. 함수가 시작될 때 ConstantPool 의 값을 r8 에 저장하는 방법은 함수가 시작될 때의 pc(program counter)값, 즉 기계어의 시작주소로부터 상수 값을 빼는 방식으로 접근한다. 이것은 Code 객체의 헤더와 기계어가 붙어있다는 가정하에서는 정상적으로 동작하지만, 본 연구에서는

Code 객체의 헤더와 기계어를 분리하였기 때문에 정상적으로 동작하지 않는다.

V8 은 함수 호출 시에 r1 레지스터에는 JSFunction 의 주소를 넘기는 특성을 가지고 있다. 본 연구에서는 이 특성을 이용하여, JSFunction 에서 [그림 9]의 (a)와 같이 추가한 Code 객체의 주소를 알아오고, 코드 객체의 주소로부터 ConstantPool 객체의 주소를 r8 레지스터에 저장하는 방식으로 컴파일러를 수정하였다.

3.4 공유 메모리 영역의 구조

각 프로세스는 처음 시작할 때 공유 메모리 영역을 생성하거나 이미 생성된 공유 메모리 영역을 자신의 메모리 영역에 매핑한다. 공유 메모리 영역은 생성될 때 그 값이 모두 0 으로 초기화되는 특성을 이용하여, [그림 10]과 같이 공유 메모리 영역의 가장 첫 번째 값이 0 이면 자신이 생성한 공유 메모리이기 때문에 그 값을 1 로 바꾸어 생성되었음을 표시하고, 초기화를 진행한다. 그리고 초기화가 끝나면 그 값을 2 로 변경하여 초기화가 완료되었음을 표시한다. 공유 메모리의 첫 번째 값이 1 인 경우 다른 프로세스에 의해 생성된 공유 메모리이고, 아직 초기화가 진행 중 이므로 2 가 될 때까지 기다리며, 2 인 경우 초기화가 완료된 것이기 때문에 사용한다.



[그림 10] 공유 메모리 영역 초기화 순서도

공유 메모리 영역은 [그림 11]와 같이 구성된다. 위에서 언급한 바와 같이 첫 번째 값은 생성 및 초기화 여부를 판단하는 값이고, 두 번째 값은 lock value 이다. 세 번째 값은 할당량을 저장하고, 네 번째 값은 공유 메모리 영역 시작부터 어느 영역까지 사용하고 있는지를 저장하고 있다. 세 번째 값은 binary chunk 의 양을 의미하고, 네 번째 값은 각종 빈 공간과 정보 공간을 포함하여 어디부터 어디까지의 메모리 공간이 사용되고 있는지를 의미한다.

[그림 11] 공유 메모리 영역 구조

다섯 번째 값은 free block list 를 저장하고 있다. 본 연구에서는 효율적인 메모리 영역의 관리를 위하여, 공유 메모리 영역을 free list 형태로 관리한다[22]. 현재 구현에서는 0 ~ 63 byte, 64 ~ 255 byte, 256 ~ 1023 byte, 1024 byte 이상 이렇게 4 개의 free list 가 있다. 메모리 할당 요청이 들어오면, 요청한 크기가 속하는 영역의 free block list 에서 요청한 크기를 담을 수 있는 block 을 찾아서 사용하고, 남는 영역은 다시 free block 에 넣는다. 예를 들어 600 bytes 크기의 할당 요청의 경우, 256 ~ 1023 byte 영역에서 가장 먼저 발견된 600 bytes 를 할당 가능한 free block 을 준다. 가령 이 크기가 800 bytes 라고 할 때, 요청을 처리하고 남은 200 bytes 는 그 크기에 맞는 64 ~ 255 byte 영역에 들어가게 된다. 만일 요청한

크기가 속하는 영역에 요청한 크기를 담을 수 있는 block 이 없으면, 그 다음 크기의 영역에서 찾는다. 처음에는 하나의 큰 block 이 존재하는 형태로 시작한다.

0 ~ 63byte 의 공간은 재사용하지 않는데 그 이유는 V8 JIT 컴파일러의 특성상 63 byte 이하의 instruction 은 생성되지 않기 때문이다. 영역을 나누는 기준은 각 리스트에 할당된 block 의 개수가 고르게 분포되도록 실험적으로 설정하였다.

마지막으로 공유 메모리 영역의 여섯 번째 값은 binary chunk 마다의 고유번호와 공유 메모리상의 위치를 저장한 표를 저장하고 있다. 만일 foo()라는 함수가 다른 프로세스에서 이미 컴파일되어 공유 메모리상에 있다고 할 때 foo()라는 함수가 있는지 여부와 있다면 어디에 있는지를 판별하기 위해 사용되는 표이다. Binary chunk 의 고유번호는 hash 값으로 만들어진다. Hash 값을 만들기 위한 input 은 소스코드와 최적화 여부, 최적화된 경우 타입정보를 이용하여 만들어진다. 이 세가지가 동일하다면 같은 binary chunk 이기 때문이다.

하지만 같지 않은 소스코드도 같은 함수인 경우가 존재한다. 자바스크립트 소스코드의 경우 웹에서 많이 쓰이고, 전송이 빈번하다. 따라서 전송 속도를 줄이기 위하여 [그림 12]와 같이 JavaScript minifier 를 이용하여 동일한 기능을 하면서도 변수 명, 함수 명, 주석, 공백, 개행 문자 등을 지움으로써 용량을 최소화 하는 경우가 있다. 따라서 같은 함수임에도 불구하고 소스코드가 다른 경우가 존재한다. JavaScript minifier 도 여러 종류가 존재하므로 원래의 소스코드와 ninifier 로 생성된 소스코드가 다를 뿐만 아니라 서로 다른 minifier 로 생성된 소스코드도 서로 다르다.

[그림 10]는 여러 minifier 중에서 JSMIN[23]으로 압축한 경우이다.

```
/*** before minification ***/
var checker = false;

function test() {
  alert(checker);
}

test(); // execute

/*** after minification ***/
var checker=false;function test(){alert(checker);}
test();
```

[그림 12] Minifier로 인하여 같은 함수의 소스코드가 다른 경우

3.5 공유 메모리 영역 GC

공유 메모리 영역의 GC 는 reference counting[24] 기법과 mark and sweep[25]을 복합하여 구현하였다. 먼저 모든 binary chunk 는 몇 개의 프로세스에서 사용하는지에 대한 정보를 reference count 에 저장한다. 만일 공유한 foo 라는 함수를 A 라는 프로세스와 B 라는 프로세스에서 동시에 사용 중이라면, reference count 값은 2 가 될 것이다. 이와 별개로 mark bit 을 각각의 binary chunk 마다 그리고 프로세스마다 가지고 있는다. [그림 13]은 binary chunk P, Q, R, S 에 대해서 프로세스 A 와 B 가 동시에 실행될 때 reference count 와 mark bit 이 있는 형태를 나타낸다.

	Ref Count	A's markbit	B's markbit
Binary chunk P	2	False	True
Binary chunk Q	1	False	False
Binary chunk R	1	True	False
Binary chunk S	2	False	True

[그림 13] 공유 메모리 GC를 위한 자료구조

각 프로세스에서 Full GC 를 시작할 때 공유 메모리 영역의 자기 프로세스의 mark bit 을 모두 false 로 초기화 하고, mark 를 시작한다. 사용되는 instruction 의 경우 mark 단계에서 true 로 체크된다. 이후 sweep 단계에서 각 instruction 을 방문하면서 자기 프로세스의 mar bit 이 false 인 경우 reference count 를 1 감소시킨다. Reference count 가 0 이 되는 경우 해당 binary chunk 는 모든 프로세스에서 사용하지 않는 것이기 때문에 free block list 에 들어가게 된다.

[그림 14]는 프로세스 A 가 GC 를 실행하는 과정을 단계별로 나타낸 예시다. (a) mark 를 시작하기 전에 모든 mark bit 이 false로 초기화한다. (b) marking이 끝난 상황으로 A에서 사용하는 binary chunk 만 mark bit 이 true 이다. (c) sweeping 이 끝난 상황으로 mark bit 이 false 인 binary chunk 의 reference count 값은 1 씩 감소한다. (d) reference count 가 0 인 경우 free block list 에 들어간다.

	Ref Count	A's markbit	B's markbit
Binary chunk P	2	False	True
Binary chunk Q	1	False	False
Binary chunk R	1	False	False
Binary chunk S	2	False	True

(a) Initialize markbit before marking

	Ref Count	A's markbit	B's markbit
Binary chunk P	2	False	True
Binary chunk Q	1	False	False
Binary chunk R	1	True	False
Binary chunk S	2	False	True

(b) After marking

	Ref Count	A's markbit	B's markbit
Binary chunk P	1	False	True
Binary chunk Q	0	False	False
Binary chunk R	1	True	False
Binary chunk S	1	False	True

(c) After sweeping

	Ref Count	A's markbit	B's markbit
Binary chunk P	1	False	True
Binary chunk R	1	True	False
Binary chunk S	1	False	True

(d) remove binary chunk with 0 reference count

[그림 14] 공유 메모리 GC 예시

현재 공유 메모리 GC 의 경우 compaction 을 구현하지 않았기 때문에 프로그램이 오랫동안 실행되는 경우 재사용이 불가능한 0 ~ 63 bytes 영역이 지속적으로 늘어나 메모리를 낭비시킬 수 있다. 후속연구로 compaction 을 진행하고 버려지는 영역을 재사용 가능하도록 변경 가능하다. 하지만 compaction 을 진행하기 위해서는 공유 메모리 영역을 사용하는 모든 프로세스가 멈춘 상태에서 진행해야 하기 때문에 오버헤드가 매우 클 것으로 예상된다.

4. 실험

4.1 실험 환경

Firebase, Kendo 이렇게 2 가지 라이브러리에 대하여 각각 4 가지 어플리케이션을 이용하여 실험을 진행하였다[26, 27]. 각 라이브러리 별로 2 개, 3 개, 4 개를 동시에 실행하는 경우에 대하여 실험하였다. 같은 라이브러리를 사용해야 공유할 수 있기 때문에 조합은 같은 라이브러리끼리 동시에 실행시키는 형태로만 구성하였다. Firebase, Kendo 의 어플리케이션 이름은 각각 [표 2], [표 3]과 같다. 동시에 실행하는 조합은 [표 4]와 같다. 실험환경은 Nvidia Jetson TK1[28], ARM 보드이며, CPU 2.3GHz, Ram 2GB 이다. 운영체제는 Ubuntu 14.04 이다. V8 은 3.31.1 버전이며, Chromium 브라우저는 41.0.2218.0 버전이다. 각 버전은 2014 년 12 월 버전이다.

[표 2] Firebase 어플리케이션

In short	Full name	Description
A	Drawing	그림판
B	Leader board	점수기록
C	Chat	채팅
D	Tetris	테트리스 게임

[표 3] Kendo 어플리케이션

In short	Full name	Description
E	Aero	사진 열람 및 검색
F	Bootstrap	주문 내역 및 성과 열람 및 수정
G	Sushi	초밥가게 주문
H	Chart	그래프 그리기

[표 4] 동시에 실행하는 조합

프로세스 수	동시에 실행하는 조합
2 개	A+B, C+D, E+F, G+H
3 개	A+B+C, E+F+G
4 개	A+B+C+D, E+F+G+H

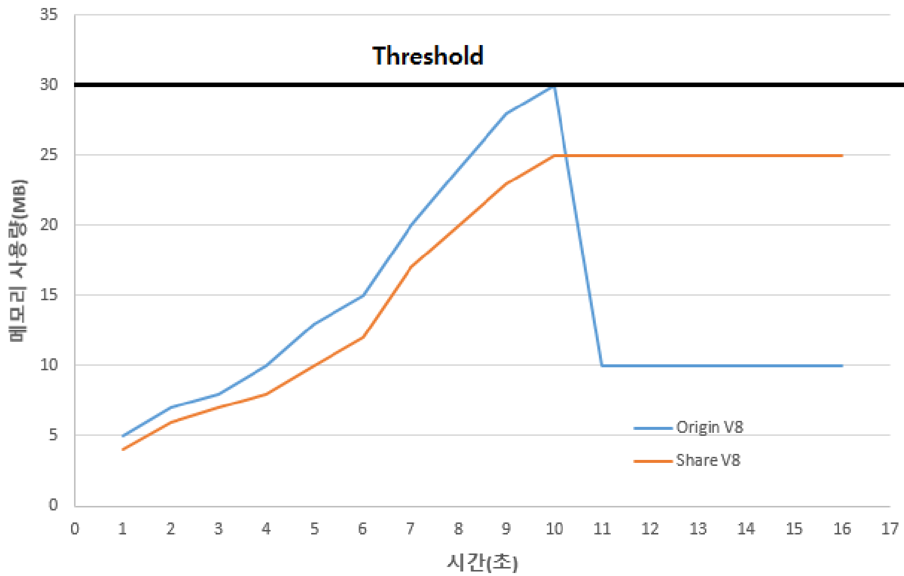
4.2 실험 방법

두 가지 방법으로 메모리 사용량을 측정하였다. 첫 번째 방법은 어플리케이션을 실행하는 중 주기(1 초)적으로 JIT 코드의 메모리 사용량을 측정하는 방법이고, 두 번째 방법은 어플리케이션을 실행하는 중 주기(1 초)적으로 Full GC 를 실행한 뒤, 코드의 메모리 사용량을 측정하는 방법이다. 코드의 메모리 사용량이란, Code 객체의 크기의 합을 말한다. 본 연구에서는 OOL 기능을 이용하여 기존의 V8 과 달리 constant pool 이 Code 객체 바깥에 존재한다. 따라서 본 연구의 구현에서의 코드의 메모리 사용량은 Code 객체의 크기와 ConstantPool 객체의 크기 그리고 공유 메모리

영역의 크기를 합한 값이다. 두 가지 측정 방법은 측정을 하기 전에 강제로 GC 를 부르느냐 마느냐의 차이가 있다. 그리고 같은 절대 시점에서의 각 어플리케이션 코드의 메모리 사용량과 공유 메모리 사용량을 모두 합한 양을 비교하였다.

어플리케이션 4 개가 동시에 실행될 때 Code 의 메모리 사용량은 각 시각에서의 사용량을 합한 값이 된다. 가령 각 어플리케이션이 3 초 시점의 메모리 사용량이 16MB, 28MB, 21MB, 15MB 이고, 공유 메모리 사용량이 5MB 라면, 총 메모리 사용량은 이것을 다 더한 값인 85MB 인 것이다. 기존 V8 의 경우는 어플리케이션 각각의 메모리 사용량이 더 증가하고, 공유 메모리의 크기가 0 일 것이다.

주기적으로 메모리의 양을 측정하는 첫 번째 방법은 보편적인 발상으로 할 수 있는 실험 방법이라 할 수 있다. 하지만 이 방법의 경우 GC 정책에 따라 메모리 사용량이 변동될 수 있다는 단점이 있다. 현재 Full GC 는 동적으로 변하는 threshold 값을 두고, 메모리 사용량이 threshold 에 도달하는 경우에 Full GC 를 실행한다. 따라서 본 연구처럼 메모리를 절약하는 경우에는 메모리를 절약했기 때문에 Full GC 가 호출되지 않는 경우가 발생할 수 있다. 예를 들어 초기 threshold 가 30MB 이고, 메모리 사용량이 [그림 15]와 같은 경우, 10 초 지점에서 기존 V8 의 경우 메모리 사용량이 threshold 에 도달하여 Full GC 가 호출되지만, 본 연구의 경우 메모리 사용량이 절감되었기 때문에 Full GC 가 호출되지 않는다. 그리고 그 이후에 어플리케이션이 메모리를 거의 사용하지 않는다면, [그림 15]와 같이 오히려 메모리 사용량이 늘어난 것과 같은 현상이 나타날 수도 있다.

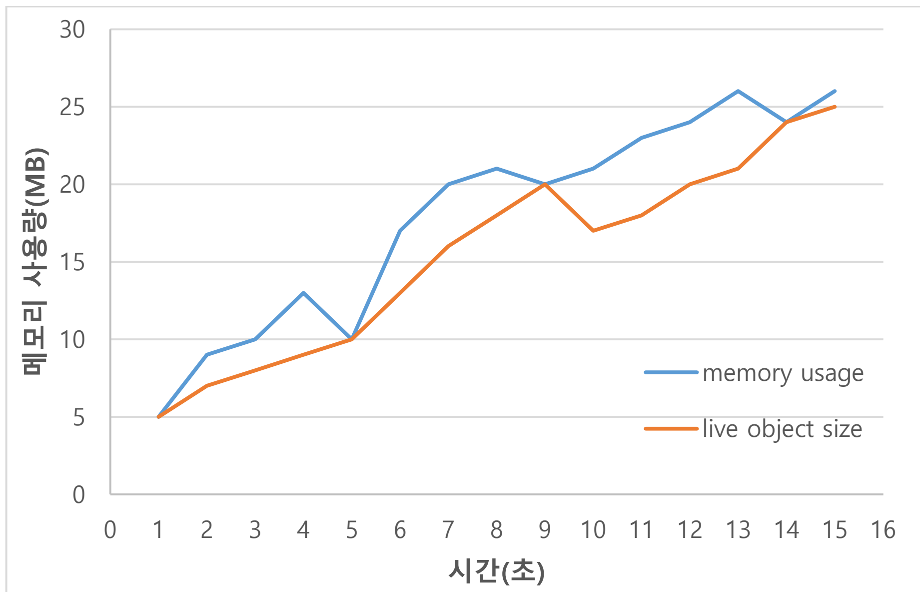


[그림 15] GC 영향으로 본 연구의 메모리 사용량이 더 많은 상황

[그림 15]와 같은 상황을 방지하기 위해서, 첫 번째 실험에서는 현재 메모리 사용량을 계산할 때 공유 메모리 사용량을 추가하여서, 기존과 동일한 시점에 GC 가 호출되도록 하였다. 따라서 기존의 V8 과 본 연구를 공정하게 비교할 수 있다.

하지만 첫 번째 측정방식은 본 연구의 효과를 정확하게 나타낼 수는 없다. 본 연구의 내용과 상관없이 단순히 GC 정책이나 알고리즘이 바뀌기만 하여도 메모리 사용량이 바뀌기 때문이다. 따라서 본 연구의 효과를 정확하게 확인하기 위하여 주기적으로 GC 를 호출하고 그 후에 메모리 사용량을 측정하여 실제 live object 의 크기만을 비교하였다. 이 경우 GC 와 상관없이 메모리 사용량이 측정된다. [그림 16]는 어떤 어플리케이션이 실행될 때 메모리 사용량과 live object 의 크기를 비교한다. GC 가 일어나기 전에는 더 이상 필요 없는 객체라 하더라도 이것이 더 이상 필요

없는 객체인지 아닌지 판단할 수 없기 때문에 계속 메모리를 차지하게 된다. [그림 16]는 5초, 9초, 14초에 GC가 발생한 것으로 가정하고 있는데, 이 시점에서는 더 이상 필요 없는 객체들은 모두 제거되었기 때문에 memory 사용량과 live object 의 크기가 같아진다. 이렇듯 GC 를 실행한 뒤에 측정한 값은 GC 와 상관없이 live object 의 크기만을 측정하기 위한 방법이다. 이 방법은 본 연구의 효과를 정확하게 확인할 수 있다.



[그림 16] 두 가지 측정방식 비교

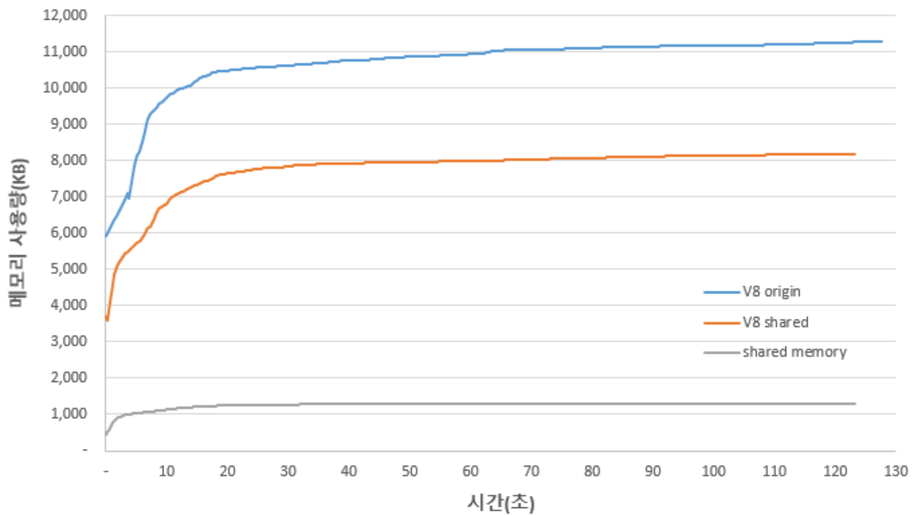
어플리케이션을 실제로 실행시켜주는 것은 xdotool[29]을 이용하여 브라우저를 띄운 상태에서 자동으로 마우스를 클릭 또는 키보드를 타이핑을 할 수 있도록 시나리오를 작성하여 실행 하였다.

각 어플리케이션에서 1 초를 주기로 측정하였으나 각 프로세스에서 측정한 시점이 동일하지 않기 때문에 동일한 시점에서의 메모리 사용량을 계산하기 위하여 측정된 값을 1 초 간격으로 linear interpolation 방식으로 근사 값으로 resampling 하였다.

총 실험은 각각 10 회씩 진행하였으며, 전체 실행시간 동안 메모리 사용량에 대한 평균을 구하였다.

4.3 실험 결과

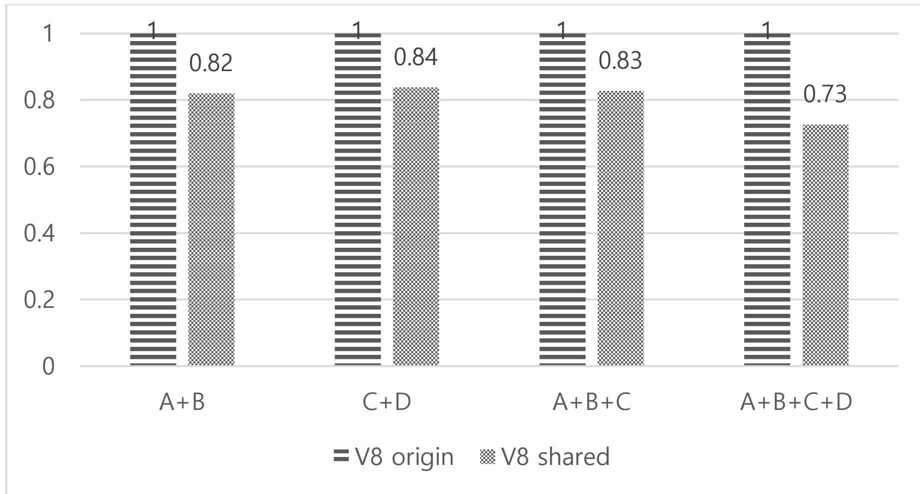
[그림 17]은 시간에 따른 코드의 메모리 사용량을 나타낸다. Firebase 의 4 가지 어플리케이션을 동시에 실행시킬 때 1 초 간격으로 메모리 사용량을 측정한 것이다.



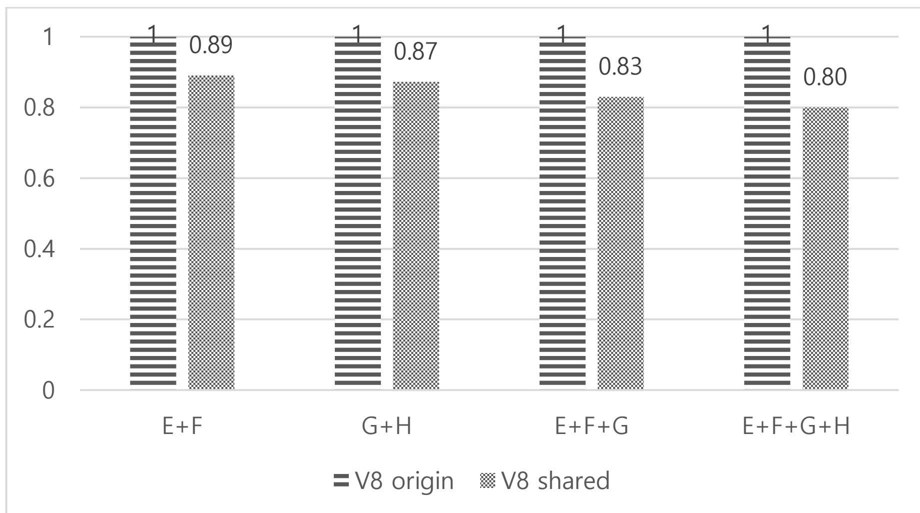
[그림 17] Firebase 프로세스 4개 실행 시 코드의 메모리 사용량

[그림 17]은 공유 메모리가 약 1,000KB 이므로 4 개의 어플리케이션이 동시에 실행되는 경우, 그리고 공유 메모리의 함수를 모든 프로세스에서 사용하는 경우 약 3,000KB 의 메모리가 절감될 것으로 기대되며, 실제로 각 순간마다 메모리 사용량의 차이가 약 3,000KB 인 것을 확인할 수 있다.

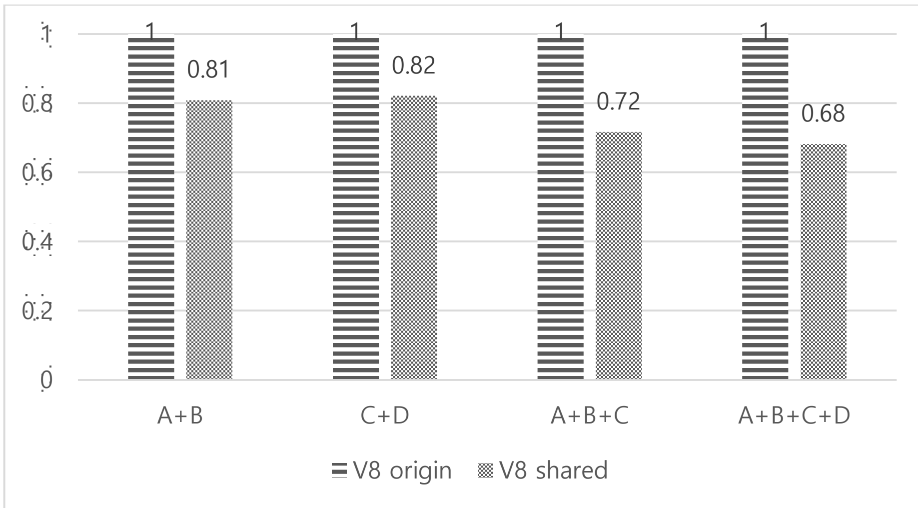
기존의 V8 대비 본 연구의 메모리 사용량을 정규화하여 표현하였다. [그림 18]와 [그림 19]은 Firebase, Kendo 각각에 대해서 1 초 간격으로 메모리 사용량을 측정한 경우를 나타내며, [그림 20]와 [그림 21]은 Firebase, Kendo 각각에 대해서 1 초 간격으로 GC 를 실행한 다음 메모리 사용량을 측정한 경우이다.



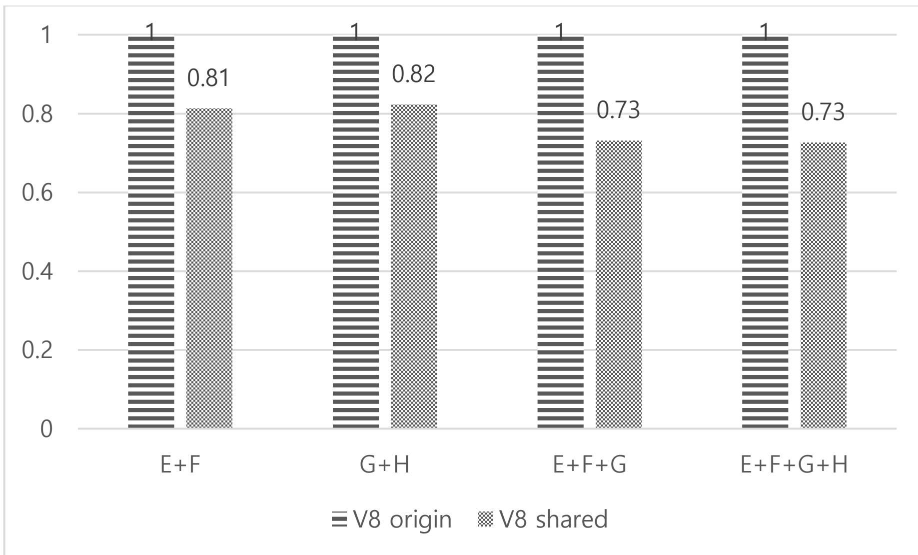
[그림 18] Firebase periodic memory



[그림 19] Kendo periodic memory



[그림 20] Firebase after GC



[그림 21] Kendo after GC

전체적으로 Firebase 어플리케이션이 Kendo 어플리케이션에 비하여 더 좋은 성능을 보이는데, 이것은 Firebase 어플리케이션이 라이브러리 코드를 더 많이 사용하기 때문이다.

더 많은 어플리케이션이 동시에 실행 될수록 더 좋은 성능을 보이는데, 공유된 메모리도 한 번은 사용량에 포함되기 때문이다. 예를 들어 3 개의 프로세스에서 10KB 의 메모리를 공유했다고 할 때, 원래는 30KB 가 사용될 것이 10KB 가 되었기 때문에 절감되는 메모리의 양은 20KB 이다.

주기적으로 메모리를 단순 측정하는 것보다 주기적으로 GC 를 실행하여 live object 의 크기를 측정하는 방식이 더 좋은 성능을 보이는데, 이것은 주기적으로 메모리를 단순하게 측정하는 방식은 GC 의 영향으로 인하여 그 효과를 제대로 보이지 못하였기 때문이다.

주기적으로 메모리를 측정하는 방법은 2 개의 어플리케이션이 동시에 실행되는 경우 기존대비 약 85.5%, 3 개의 어플리케이션이 동시에 실행되는 경우 기존대비 약 82.8%, 4 개의 어플리케이션이 동시에 실행되는 경우 기존대비 약 76.2%의 메모리 사용량을 보인다.

주기적으로 GC 를 실행한 뒤에 메모리 사용량을 측정하는 방법은 2 개의 어플리케이션이 동시에 실행되는 경우 기존대비 약 81.7%, 3 개의 어플리케이션이 동시에 실행되는 경우 기존대비 약 72.4%, 4 개의 어플리케이션이 동시에 실행되는 경우 기존대비 약 70.4%의 메모리 사용량을 보인다.

5. 결론

유용한 JavaScript 라이브러리의 출현으로 편의성과 신뢰성을 위하여 이를 이용한 JavaScript 프로그램이 늘어나고 있다. 라이브러리 코드는 동일한 기계어 코드를 생성하는데, 현재 V8 JavaScript 엔진은 여러 V8 프로세스가 실행되더라도 이를 공유하지 않고 각 프로세스의 메모리 영역에 할당한다. 향후 더 유용한 자바스크립트 라이브러리가 생겨나고, 그 사용량이 늘어난다면, 본 연구는 더욱 큰 의미를 가질 것이다.

본 연구에서는 V8 자바스크립트 엔진에서 JIT 컴파일러에 의하여 생성된 동일한 기계어 코드를 공유함으로써 메모리를 절약하였다. Firebase, Kendo 라이브러리에 대하여 각각 네 가지 어플리케이션을 2 개, 3 개, 4 개를 동시에 실행시키는 환경에서 실험하였고, 2 개, 3 개, 4 개 각각에서 평균 코드 사용량을 18.3%, 27.6%, 29.6% 절감하는 효과를 보인다.

참고 문헌

[1] Angular.js. [Online]. Available: https://angularjs.org

[2] React. [Online]. Available: https://facebook.github.io/react/

[3] Three.js. [Online]. Aavailable: https://threejs.org

[4] JQuery. [Online]. Available: https://jquery.com

[5] J. Resig, (2008). *State of jQuery* [Online]. Available: http://www.slidshare.net/jeresig/state-of-jquery-08-presentation/

[6] ECMAScript. [Online]. Available: http://www.ecmascript.org

[7] D. Flanagan. "Types, Values, and Variables" in *JavaScript: The Definitive Guide*, 6th ed. Oreilly, 2011, ch. 3, pp. 29-56

[8] Google. *Design Elements*. [Online]. Available: https://developers.google.com/v8/design

[9] H. Kim, S. Bak, and J. Lee, "*Lightweight and Block-level Concurrent Sweeping for JavaScript Garbage Collection*". in LCTES, Edinburgh, UK, 2014, pp. 155-164

[10] W. Ahn, J. Choi, T. Shull, M. J. Garzaran, and J. Torrellas, "*Improving JavaScript performance by deconstructing the Type system*". in PLDI, Edinburgh, UK, 2014, pp. 496-507

[11] Google. V8 source code file objects.cc and objects.h. [Online]. Available: https://github.com/v8/v8

[12] J. Conrod. (2014). *A tour of V8: Garbage Collection* [Online]. Available: http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection

[13] R. Jones, R. Lins, “Generational Garbage Collection”, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley, 1996, Ch. 7, pp. 143–181

[14] P. R. Wilson. “*Uniprocessor garbage collection techniques*”, in IWMM, London, UK, 1992, pp. 1–42

[15] H. Lieberman, C. Hewitt. “*A real-time garbage collector based on the lifetimes of objects*”, in CACM, New York, USA, 1983, vol. 26, no. 6, pp. 419–429

[16] D. Ungar, “*Generation Scavenging: A non-disruptive high performance storage reclamation algorithm*”, in SIGPLAN, New York, USA, 1984, vol. 19, no. 5, pp. 157–167

[17] C. J. Cheney, “*A nonrecursive list compacting algorithm*”, in CACM, New York, USA, 1970, vol. 13, no. 11 pp. 677–678

[18] J. McCarthy, “*Recursive Function of Symbolic Expressions and Their Computation by Machine, Part I*”, in CACM, New York, USA, 1960, vol. 3, no. 4, pp. 184–195

[19] J. Aycock, “*A brief history of just-in-time*”, in CSUR, New York, USA, 2003, vol. 35, no. 2, pp. 97–113

[20] M. Arnold, S. J. Fink, D. Grove, M. Hind, P. F. Sweeney. “*A Survey of Adaptive Optimization in Virtual Machines*”, in Proceedings of the IEEE - PIEEE, 2005, vol. 93, no. 2, pp. 449–466

[21] H. Kim, “*JavaScript Compilation and Optimization Techniques for Multicores*”, in SNU Master, Seoul, Korea,

2011

[22] K. C. Knowlton. "*A Fast storage allocator*", in CACM, New York, USA, 1965, vol. 8, no. 10, pp. 623−624

[23] JSMIN. [Online]. Available: http://www.crockford.com/javascript/jsmin.html

[24] R. Jones, R. Lins, "*Reference Counting*", Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley, 1996, Ch. 3, pp. 43−74

[25] R. Jones, R. Lins, "*Mark−Sweep Garbage Collection*", Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley, 1996, Ch. 4, pp. 75−96

[26] Firebase. [Online]. Available: https://www.firebase.com

[27] Kendo. [Online]. Available: http://www.telerik.com/kendo−ui

[28] Nvidia. "*Jetson TK1*". [Online]. Available: http://www.nvidia.com/object/jetson−tk1−embedded−dev−kit.html

[23] xdotool. [Online]. Available: http://www.semicomplete.com/projects/xdotool