creative commons

COMMONS DEED

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

자바스크립트 엔진을 위한 선택적 선행 컴파일러의 설계 및 구현

A Design and Implementation of Selective Ahead-of-Time Compilation for JavaScript Engines

2012년 8월

서울대학교 대학원
전기·컴퓨터 공학부
박 혁 우

자바스크립트 엔진을 위한
선택적 선행 컴파일러의 설계 및 구현

지도교수 문 수 묵

이 논문을 공학석사학위논문으로 제출함

2012년 8월

서울대학교 대학원
전기·컴퓨터 공학부
박 혁 우

박혁우의 석사학위논문을 인준함
2012년 8월

위 원 장 김 태 환 (인)
부위원장 문 수 묵 (인)
위 원 백 윤 흥 (인)

초 록

자바스크립트 언어는 스크립트 언어 중 하나로 표현이 풍부하며 비전문가들도 쉽게 사용할 수 있어서 웹 환경에서 클라이언트 사이드 프로그래밍 언어로 널리 쓰이고 있다. 특히 웹앱(Web App)과 RIA(Rich Internet Application)의 등장으로 인해 웹 환경에서 자바스크립트 비중이 크게 증가하고 있다.

최근 들어 스마트폰, 태블릿 PC의 대중화로 모바일 기기에서 웹 서비스의 사용이 크게 늘고 있지만 자바스크립트 언어의 동적인 특성으로 인해 제한된 모바일 환경의 하드웨어에서는 자바스크립트 수행이 병목으로 작용하고 있는 것이 사실이다. 이 때문에 자바스크립트 엔진의 수행 성능이 중요한 이슈가 되고 있다.

현재 자바스크립트 엔진은 성능 향상을 위해 실행 중에 자바스크립트 코드로부터 기계어 코드를 생성하여 이를 수행하는 적시 컴파일(JITC, Just-in-Time Compiler) 방식을 채택하고 있다. 그러나 적시 컴파일 방식은 컴파일 오버헤드가 수행 시간에 그대로 포함되며 실제 웹 환경에서는 그 비중이 상당히 크다는 단점이 존재한다.

본 논문에서는 자바스크립트 엔진의 성능 향상을 위하여 선행 컴파일(AOTC, Ahead-of-Time Compiler) 방식을 도입하였다. 선행 컴파일 방식은 한번 생성된 기계어 코드를 저장한 뒤, 다음 수행에서 재사용하는 방식이다. 이 방식은 적시 컴파일 방식의 단점인 컴파일 오버헤드를 줄이는 데 효과가 있다. 실제로 선행 컴파일 방식을 적용한 결과, 성능 향상이 있었지만 보조 기억 장치의 저장 공간을 많이 필요로 하기 때문에 그대로 적용하기엔 무리가 있었다.

선행 컴파일 방식의 한계를 보완하기 위해 여기선 새로운 방식인 선택적 선행 컴파일(Selective-AOTC) 방식을 제안하였다. 선택적 선행 컴파일 방식은 선행 컴파일 방식을 선택적으로 적용하여, 적은

저장 공간을 사용하면서 동시에 최대한의 성능 이득을 얻는 것을 목표로 한다. 선택적 선행 컴파일러의 설계를 위해 컴파일 오버헤드와 생성된 기계어 코드의 크기를 기준으로 하는 선택 휴리스틱을 정하였다. 이를 토대로 구현한 결과 V8 벤치 마크에서 10.8%의 성능 향상이 있었고, 기존 선행 컴파일 방식과 비교할 때 65.1%의 저장 공간을 절약할 수 있었다.

주요어 : 자바스크립트, 자바스크립트 엔진, 적시 컴파일, 선행 컴파일, 선택적 선행 컴파일, 웹 최적화
학번 : 2010-23268

목 차

초 록 ····· i
목 차 ····· iii
표 목 차 ····· iv
그 림 목 차 ····· v

제1장 서론 ····· 1
제2장 SFX 자바스크립트 엔진 ····· 3
2.1. SFX 자바스크립트 엔진 개요 ····· 3
2.2. SFX 자바스크립트 엔진의 적시 컴파일 과정 ····· 7
2.3. SFX 자바스크립트 엔진의 컴파일 오버헤드 ····· 10
제3장 선행 컴파일러 ····· 12
3.1. 선행 컴파일 개요 ····· 12
3.2. 선행 컴파일러 설계 ····· 15
3.3. 중간 언어 단계 선행 컴파일러 ····· 19
3.4. 기계어 코드 단계 선행 컴파일러 ····· 22
3.5. 성능 측정 및 해결 과제 ····· 27
제4장 선택적 선행 컴파일러 ····· 31
4.1. 선택 휴리스틱 ····· 31
4.2. 선택적 선행 컴파일러 설계 ····· 35
4.3. 실험 결과 ····· 37
제5장 결론 및 향후 과제 ····· 41
참 고 문 헌 ····· 43
Abstract ····· 44

표 목 차

<표 1> 기계어 코드에서 참조하는 객체와 할당된 ID 값 25
<표 2> ARM 아키텍처 실험 환경 ····· 27
<표 3> 각 선행 컴파일 방식의 특징 ····· 31
<표 4> ARM 아키텍처 실험 환경 ····· 37

그 림 목 차

[그림 1] SFX Bytecode와 Java Bytecode 비교 ····· 3
[그림 2] SFX Bytecode의 Virtual Register 메모리 모델 · 4
[그림 3] SFX Register File 구조 ····· 4
[그림 4] SFX Interpreter 수행 방식 ····· 5
[그림 5] SFX 전체 수행 과정 ····· 7
[그림 6] SFX 적시 컴파일 과정 ····· 8
[그림 7] V8 벤치마크에서 측정한 컴파일 오버헤드 ····· 10
[그림 8] 웹 사이트에서 측정한 컴파일 오버헤드 ····· 11
[그림 9] 선행 컴파일 과정 ····· 12
[그림 10] 선행 컴파일러의 구조 ····· 15
[그림 11] AOTC 파일 구조 ····· 16
[그림 12] AOTC 파일 포맷 예시 ····· 17
[그림 13] 중간 언어 단계 선행 컴파일 과정 ····· 19
[그림 14] CodeBlock 자료 구조 ····· 20
[그림 15] 기계어 코드 단계 선행 컴파일 과정 ····· 22
[그림 16] ARM 기계어 코드에서 handler function 호출 24
[그림 17] 주소값 재배치 예제 ····· 25
[그림 18] 중간 언어 단계 선행 컴파일 방식의 성능 ····· 28
[그림 19] 기계어 코드 단계 선행 컴파일 방식의 성능 ···· 28
[그림 20] 선행 컴파일 방식에서 AOTC 파일의 크기 ····· 30
[그림 21] 기계어 코드 생성 시간과 기계어 코드의 크기 32
[그림 22] 기계어 코드의 크기와 컴파일 시간의 비율 ····· 32
[그림 23] 선택 휴리스틱의 기준 ····· 33

[그림 24] 선택적 선행 컴파일러의 구조 ····· 35
[그림 25] V8 벤치마크 수행 시간 ····· 38
[그림 26] AOTC 파일의 크기 ····· 39
[그림 27] 기계어 코드 선행 컴파일이 적용된 함수의 개수 40

제1장 서론

자바스크립트 언어는 스크립트 언어 중 하나로 표현이 풍부하며 비전문가들도 쉽게 사용할 수 있어서 웹 환경에서 클라이언트 사이드 프로그래밍 언어로 널리 쓰이고 있다. 특히 웹앱(Web App), RIA(Rich Internet Application)의 등장으로 인해 웹 환경에서 자바스크립트의 비중이 크게 증가하면서 자바스크립트 엔진의 수행 성능이 중요한 이슈가 되고 있다[1].

따라서 효율적인 자바스크립트 코드의 수행을 위해 최근 다양한 기법을 탑재한 고성능 자바스크립트 엔진에 대한 연구가 활발히 이루어지고 있다. 성능 향상을 위한 방안 중 하나로, 수행 중에 자바스크립트 코드로부터 함수나 루프 단위로 기계어 코드를 생성하여 수행하는 적시 컴파일 방식(JITC, Just-in-Time Compiler)이 있다[2]. 적시 컴파일 방식을 도입한 대표적인 공개 자바스크립트 엔진에는 Mozilla Firefox 브라우저의 TraceMonkey 자바스크립트 엔진[3], Google Chrome 브라우저의 V8 자바스크립트 엔진[4], 그리고 Apple Safari 브라우저의 SFX(SquirrelFish Extreme) 자바스크립트 엔진[5]이 해당한다.

적시 컴파일 방식은 인터프리터 방식에 비해 빠른 수행 성능을 보이지만 자바스크립트 언어의 역동적인 특성과 복잡한 언어 스펙 등으로 인해 제한된 모바일 환경의 하드웨어에서는 자바스크립트의 수행이 병목으로 작용하고 있는 현실이다. 특히 적시 컴파일 방식은 수행 중 컴파일을 하기 때문에 컴파일 오버헤드가 그대로 전체 수행에 포함되는 제약이 있다. 이로 인해 컴파일 오버헤드의 비중이 큰 실제 웹에서는 적시 컴파일러를 통해 뚜렷한 성능 향상을 얻는 것이 힘들다[6]. 이 한계점을 극복하고자 나온 방식이 바로 선행 컴파일 방식(AOTC, Ahead-of-Time Compiler)이다.

선행 컴파일러는 실제로 수행이 되기 전에 소스 파일을 컴파일하

여 기계어 코드를 저장한 뒤, 실제 수행 때는 컴파일 과정을 거치지 않고 미리 저장된 기계어 코드를 그대로 불러와 수행하는 방식이다 [7]. 이러한 방식은 실제 수행에서 컴파일 오버헤드를 줄여서 전체 수행 성능을 향상시킬 수 있다는 이점이 있다. 그러나 선행 컴파일 방식은 보조 기억 장치의 저장 공간을 사용해야 하는 제약 또한 존재한다.

본 논문에서는 선행 컴파일 방식을 구현해서 성능과 한계점을 살펴본 뒤, 선행 컴파일 방식을 개선한 선택적 선행 컴파일 (Selective-AOTC) 방식을 제안할 것이다. 선택적 선행 컴파일러는 적은 저장 공간을 사용하는 동시에 최대한의 성능 효과를 얻는 것을 목표로 한다.

이를 위해 2장에서는 SFX 자바스크립트 엔진에 관해 설명하고 3장에서는 선행 컴파일러 방식을 구현하여 성능과 함께 한계점에 대해 살펴본다. 그리고 4장에서는 선택적 선행 컴파일 방식에서 선택 휴리스틱과 함께 설계에 대해 설명하고 실험을 통해 선행 컴파일 방식의 효과에 대해 살펴본다. 그리고 5장에서는 본 연구의 종합적 결과와 함께 향후 연구 과제를 제시한다.

제2장 SFX 자바스크립트 엔진

본 장에서는 SFX 자바스크립트 엔진에 대해 소개하고 자바스크립트 소스 코드의 컴파일 과정에 관해 설명한다.

2.1. SFX 자바스크립트 엔진 개요

Squirrel Fish Extreme(SFX)는 웹킷(WebKit) 엔진에 포함된 자바스크립트 엔진으로서 Apple의 Safari 웹 브라우저에 탑재되어 있다. 기존 인터프리터 방식인 Squirrel Fish 엔진에 적시 컴파일 방식을 추가한 SFX 엔진은 인터프리터와 적시 컴파일, 두 가지 자바스크립트 수행 방식을 지원한다.

현재 SFX는 x86, x64, ARM, MIPS등 여러 아키텍처 환경을 지원하며 자바스크립트 소스 코드를 자체 정의된 중간 언어인 Bytecode로 변환한 후에 인터프리터나 적시 컴파일러 중 한 가지 방식으로 동작한다.

SFX Bytecode(Register based)	Java Bytecode(Stack based)
`enter`	`Method int top()`
`init_lazy_reg r0`	`  aload_0 //push the object`
`init_lazy_reg r2`	`  getfield //pop the object`
`init_lazy_reg r1`	`  iconst_0 //push 0`
`mov r3, "A"(@k0)`	`  iaload //pop the top two value`
`new_object r4`	`  ireturn //pop top value and`
`put_by_id r4, get(@id0), r5`	`  push it to op stack`

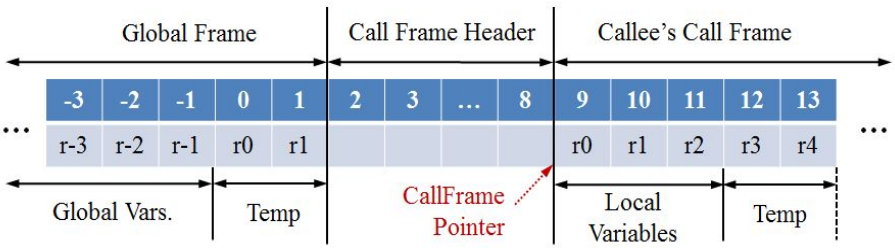
[그림 1] SFX Bytecode와 Java Bytecode 비교

SFX Bytecode는 인터프리터와 적시 컴파일러의 두 가지 방식에서 공통적으로 사용되는 레지스터 기반의 중간 언어로서, 자바의 스택 기반 Bytecode 와는 다르게 Virtual Register, Constant, Identifier 등을 매개 변수로 사용한다[8].

Data Type	Tag (High 32bit)	Data (Low 32bits)
Integer (32bit)	0xffffffff	Integer value
Boolean	0xfffffffe	Boolean value
Null	0xfffffffd	0
Undefined	0xfffffffc	0
Object (JSCell)	0xfffffffb	Pointer

[그림 2] SFX Bytecode의 Virtual Register 메모리 모델

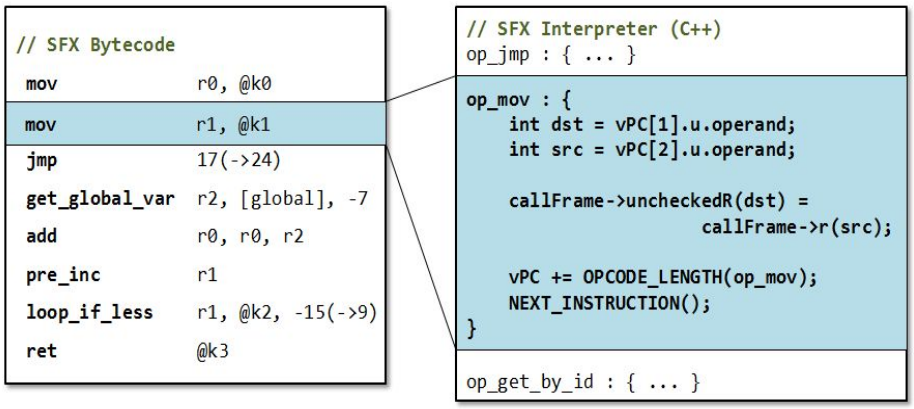
또한 Bytecode는 64bit의 Virtual Register를 사용하는데 자바스크립트의 동적인 데이터 타입을 기록하기 위해 상위 32bit에는 타입 정보를 기록하고 하위 32bit에는 실제 값을 저장하는 형태로 구현하고 있다. 이 중 double형의 데이터는 IEEE754 Representation을 따르고 있고 나머지 타입의 데이터들은 IEEE754 Representation의 Nan encoding 방식을 사용한다.



[그림 3] SFX Register File 구조

Register File은 Bytecode에서 사용하는 Virtual Register들을 저장하기 위한 배열 방식의 자료구조로 C 언어의 스택과 유사한 형태를 가진다. 새로운 함수가 호출되면 해당 함수를 위한 공간이 새로 할당되면서 Register File이 오른쪽으로 확장되고 전역 변수가 추가되면 왼쪽 끝으로 확장된다. 그리고 Call Frame은 현재 수행 중인 함수에서 사용하는 Virtual Register의 범위를 나타내는 포인터로 이 값을 통해 Register File 내에 필요한 데이터에 접근할 수 있다.

SFX에서 자바스크립트의 컴파일은 함수 단위로 이루어진다. 컴파일 시점은 Lazy Compilation 방식을 따르고 있는데 Lazy Compilation은 어떤 함수를 컴파일 하는 중에 함수 내부에 새로운 함수의 정의가 발견되면 그 함수는 바로 컴파일 되지 않고 FunctionType, Prototype 등의 기본적인 자료구조만 저장했다가, 이후 해당 함수가 실제로 호출될 때 함수를 컴파일 하는 방식이다. 컴파일 후에 생성된 코드(Bytecode 또는 기계어 코드)는 코드 캐시에 저장되어 후에 동일한 소스 코드가 수행될 때 컴파일 과정을 거치지 않고 코드 캐시에 저장된 코드를 그대로 재사용한다.



[그림 4] SFX Interpreter 수행 방식

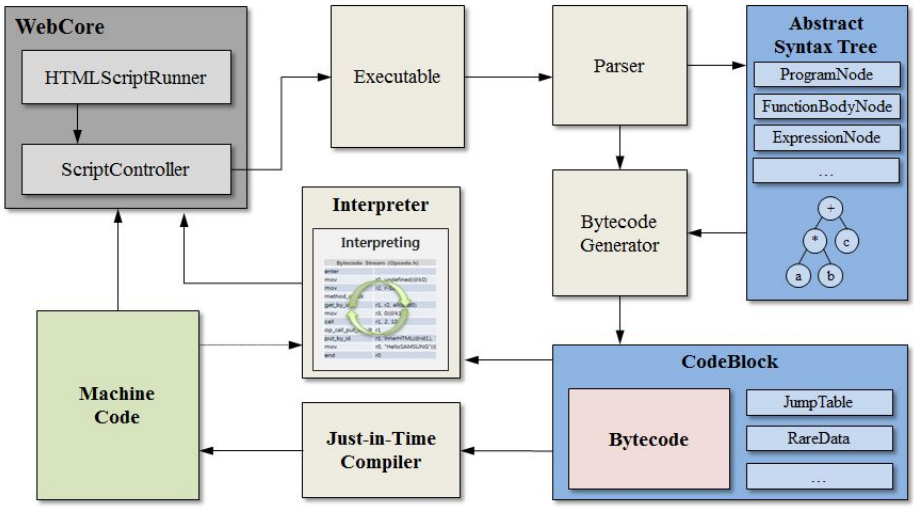
SFX의 인터프리터에서는 생성된 Bytecode를 하나씩 해석하며 미리 정의된 루틴을 순차적으로 수행한다. 인터프리터 방식은 비교적 개발과 디버깅이 용이하나, 소프트웨어 단계에서 자바스크립트 코드를 해석하고 수행하기 때문에 수행 시간이 오래 걸린다는 단점이 존재한다.

적시 컴파일러에서는 Bytecode를 기계어 코드로 변환하는 과정을 한 번 더 거친 뒤 생성된 기계어 코드를 수행한다. 적시 컴파일 방식에서는 기계어 코드로 변환하는 컴파일 오버헤드가 추가되지만 기계어 코드의 수행은 하드웨어 단계에서 이루어진다. 따라서 기계어 코드의 빠른 수행을 통해 반복 수행이 많은 루프나 동일한 함수가 여러번 호출되는 프로그램에서 큰 성능 효과를 볼 수 있다.

실제로 SFX에서 적시 컴파일 방식은 기존의 인터프리터 방식에 비해 벤치마크 기준으로 2-3배의 성능 향상이 있으며 현재 SFX 엔진은 기본적으로 적시 컴파일 방식으로만 수행하도록 빌드 옵션이 되어 있다.

2.2. SFX 자바스크립트 엔진의 적시 컴파일 과정

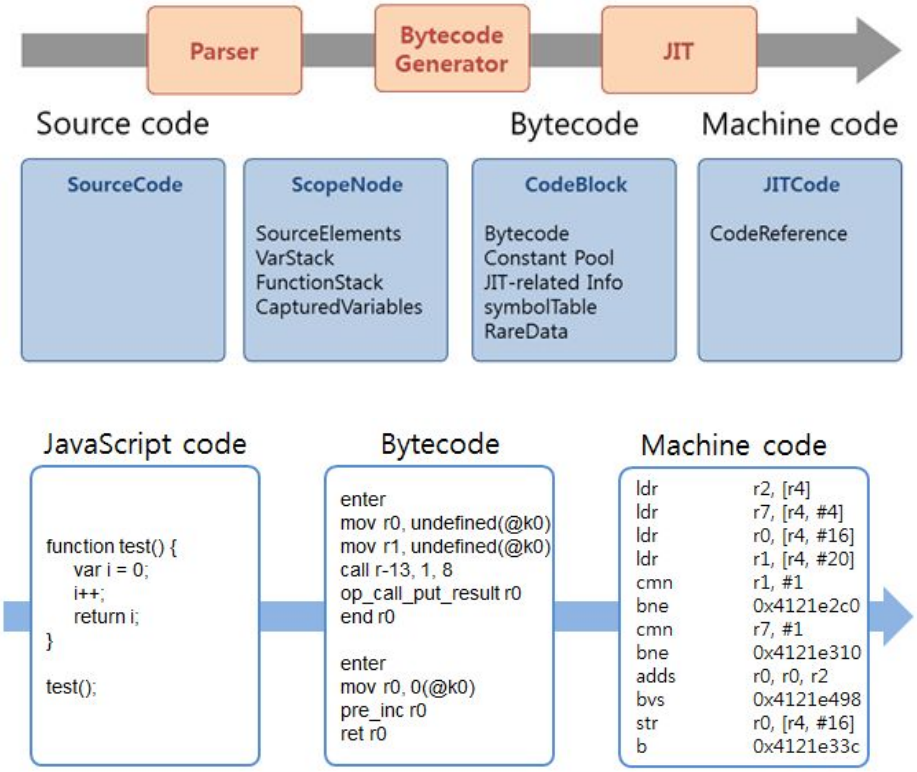
적시 컴파일 방식은 본 연구와 밀접한 관련이 있으므로 본 절에서는 SFX 엔진의 적시 컴파일에 관해 자세히 설명한다.



[그림 5] SFX 전체 수행 과정

[그림 5]는 웹 브라우저에 포함된 SFX 엔진의 자바스크립트 수행 과정을 나타낸다. 웹 브라우저에서 수행할 자바스크립트 소스 코드를 SFX 엔진에 전달하면, SFX에서는 받은 소스 코드를 컴파일 하여 수행한 뒤 그 결과를 다시 웹 브라우저로 넘겨준다.

SFX는 먼저 자바스크립트 소스 코드를 중간 언어인 Bytecode로 변환한다. 그 뒤 SFX 엔진 빌드 시에 정한 옵션에 따라 인터프리터나 적시 컴파일 중 한 가지 방식으로 수행하는데, 적시 컴파일러에서는 Bytecode를 최종적인 기계어 코드로 변환한 뒤 생성된 기계어 코드를 수행한다.



[그림 6] SFX 적시 컴파일 과정

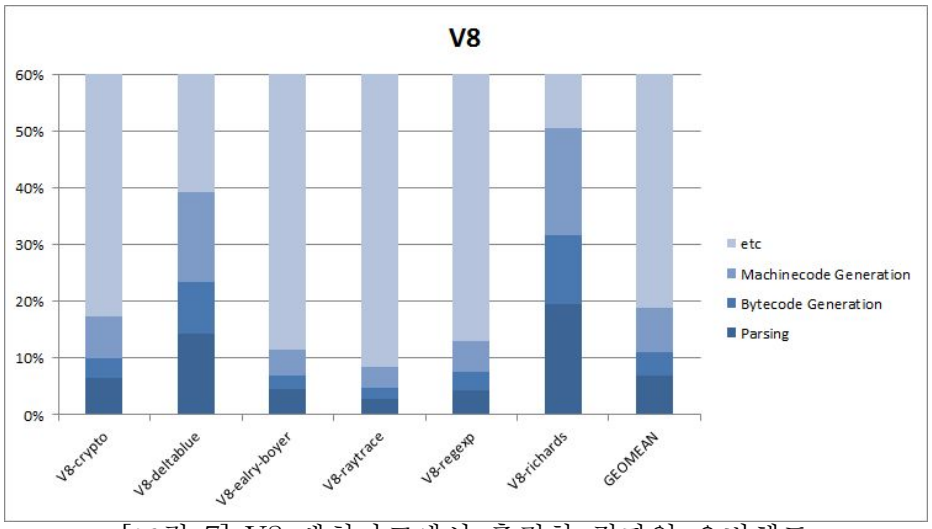
[그림 6]은 SFX 적시 컴파일 과정을 나타낸 것으로 자바스크립트 소스 코드가 최종적인 기계어 코드로 변환되면서 거치게 되는 엔진 모듈들과 생성되는 자료 구조들을 보여준다.

자바스크립트 소스 코드를 Bytecode로 변환하는 과정은 Parser와 Bytecode Generator가 수행한다. Parser는 자바스크립트 소스 코드에 대한 정보가 저장되어 있는 SourceCode 자료구조를 파싱하여 파스 트리(Abstract Syntax Tree) 및 기타 정보를 ScopeNode라는 자료구조에 저장한다[9]. Bytecode Generator는 이를 이용하여 Bytecode 및 기타 정보를 CodeBlock이라는 자료구조에 저장한다. 그리고 적시 컴파일러에서 CodeBlock 자료구조를 이용하여

Bytecode를 최종적인 기계어 코드로 변환하면 이를 힙 영역에 할당된 Executable Pool이라는 코드 캐시에 저장하고 시작 주소를 JITCode 자료구조에 기록한 뒤, 생성된 기계어 코드를 수행한다.

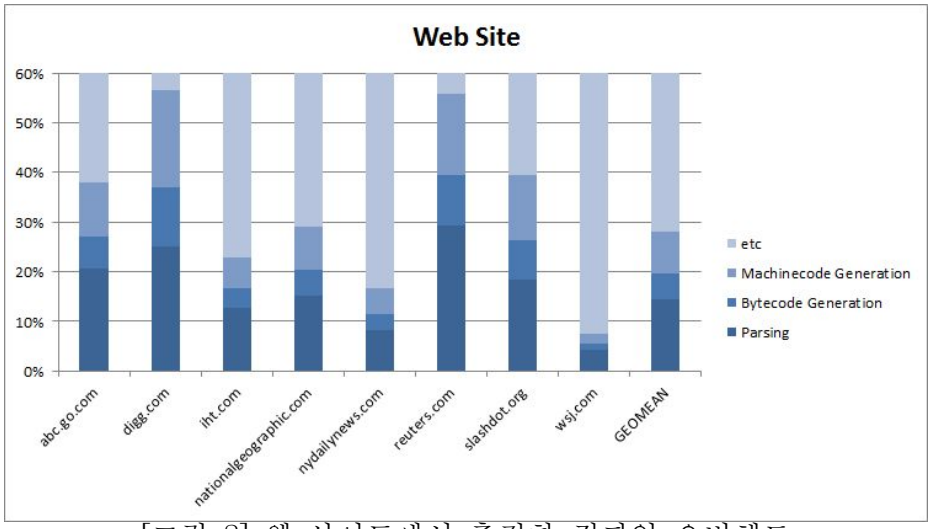
2.3. SFX 자바스크립트 엔진의 컴파일 오버헤드

컴파일 오버헤드는 컴파일 과정에서 소요되는 시간을 의미한다. 자바스크립트 엔진은 자바스크립트 소스 코드를 받아서 실시간으로 컴파일 한 뒤 수행하기 때문에 컴파일 오버헤드가 전체 수행 시간에 그대로 포함된다. 특히 SFX 엔진에서 적시 컴파일 방식으로 수행할 경우 파싱, Bytecode 변환, 기계어 코드 생성의 과정을 거치기 때문에 전체 수행 시간에서 차지하는 컴파일 오버헤드의 비중이 높다.



[그림 7] V8 벤치마크에서 측정한 컴파일 오버헤드

[그림 7]은 ARM 환경에서 V8 벤치마크를 대상으로 SFX 전체 수행 시간 중 차지하는 컴파일 오버헤드를 측정한 결과이다. V8 벤치마크에서 컴파일 오버헤드는 평균 18.7%의 비중을 보인다.



[그림 8] 웹 사이트에서 측정한 컴파일 오버헤드

[그림 8]은 자바스크립트의 비중이 큰 8개의 웹 사이트를 대상으로 SFX 엔진에서 적시 컴파일 방식을 적용했을 때 나타나는 컴파일 오버헤드를 측정한 결과이다. 웹 사이트에서 컴파일 오버헤드는 평균 28.1%를 차지하는데 이는 V8 벤치마크에서 측정된 컴파일 오버헤드보다 높은 비중을 보인다[10].

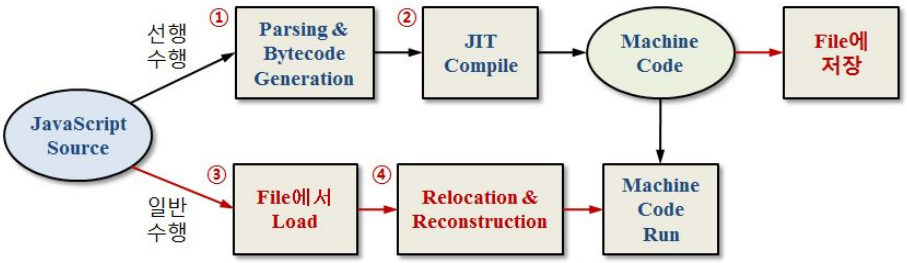
적시 컴파일 방식은 기계어 코드를 생성하여 이를 통해 빠른 수행을 할 수 있다는 장점이 있다. 그러나 위의 실험 결과에서 보듯이 전체 수행에서 차지하는 컴파일 오버헤드 비중이 높다는 단점 역시 존재한다. 적시 컴파일 방식을 보완하여 컴파일 오버헤드는 줄이면서 동시에 기계어 코드로 빠른 수행을 할 수 있다면 적시 컴파일 방식의 성능을 많이 개선할 수 있을 것이다.

다음 장에서는 적시 컴파일 방식의 개선 방안 중 하나로 선행 컴파일 방식을 도입하고 그 효과와 한계에 대해 살펴본다.

제3장 선행 컴파일러

본 장에서는 자바스크립트 엔진의 컴파일 오버헤드를 줄이기 위한 방안으로 선행 컴파일(AOTC, Ahead-Of-Time Compile) 방식을 도입하고 성능 분석과 함께 선행 컴파일의 한계점에 대해 살펴본다.

3.1. 선행 컴파일 개요



[그림 9] 선행 컴파일 과정

기존 적시 컴파일(JITC, Just-In-Time Compilation) 방식은 수행 중에 소스 코드를 컴파일 하여 생성된 기계어 코드를 코드 캐시에 저장하고 사용하는 방식이다. 코드 캐시에 저장된 기계어 코드들은 자바스크립트 프로그램이 종료되기 전까지만 유효하고, 후에 동일한 프로그램이 또 수행될 시 매번 컴파일 과정을 다시 거쳐야 한다. 이렇게 프로그램 수행마다 매번 반복되는 컴파일 과정을 생략하기 위해 나온 것이 선행 컴파일 방식이다.

선행 컴파일(AOTC, Ahead-Of-Time Compilation) 방식은 수행 전에 미리 소스 코드를 컴파일 하여 생성된 기계어 코드를 보조 기

억 장치에 저장한 뒤, 실제로 수행할 시점에서는 컴파일 과정을 생략하고 저장된 기계어 코드를 가져와 수행하는 방식이다. 선행 컴파일 방식은 수행 시 소스 코드를 컴파일 하는 오버헤드를 줄일 수 있어 컴파일 오버헤드가 큰 자바스크립트 엔진에서 전체 수행 성능을 향상시킬 수 있다.

선행 컴파일러의 구성은 적시 컴파일러에 기반을 두고 있다. 프로그램을 처음 수행하게 되면 미리 저장된 기계어 코드가 없으므로 적시 컴파일러 모듈을 통해 컴파일 과정을 거친 뒤에 생성된 기계어 코드를 파일 형태로 보조 기억 장치에 저장하게 된다. 프로그램의 첫 수행 시 거치게 되는 일련의 과정을 선행 수행(FirstRun)이라고 한다. 선행 수행을 하고 난 다음에 또 같은 프로그램을 실행하게 될 경우 기계어 코드가 저장된 파일을 불러와서 컴파일 과정 없이 바로 수행하게 된다. 이 과정을 일반 수행(AOTCRun)이라고 한다.

실제 구현에서는 생성된 기계어 코드를 저장하는 모듈과 저장된 기계어 코드를 읽어 와서 수행하는 모듈 외에 재배치(Relocation)와 재구성(Reconstruction)을 위한 모듈이 추가로 필요하다. 선행 수행 결과 생성되는 기계어 코드에는 명령어 외에 여러 가지 메모리 주소값이 사용된다. 이 메모리 주소값은 일반 수행 시 변경될 수 있어서 정확한 수행을 위해서는 일반 수행에서 유효한 값으로 재배치를 해주어야 한다. 또한 선행 수행에서는 적시 컴파일 과정 중에 기계어 코드 수행에 필요한 자료구조들을 생성한다. 일반 수행에서는 컴파일 과정이 생략되어 필요한 자료구조들이 누락되므로 생략된 자료구조들을 다시 재구성 하는 작업이 필요하다. 선행 수행에서는 재배치와 재구성을 위해 기계어 코드와 함께 관련된 정보들을 같이 저장해야 한다. 그리고 일반 수행에서는 파일에서 기계어 코드와 함께 재배치, 재구성과 관련된 정보들을 읽어 재배치와 재구성을 한 뒤에 수행을 해야 한다.

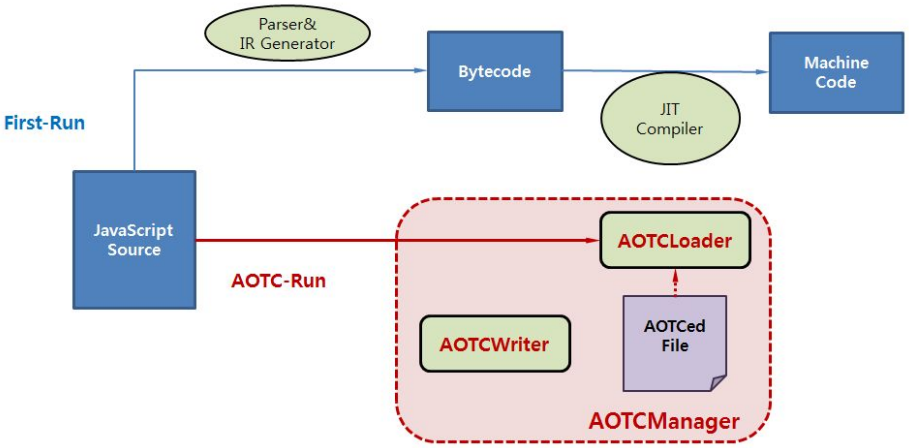
선행 컴파일 방식을 통해 성능의 이득을 얻으려면 컴파일 과정을 생략하여 얻는 수행 시간의 이득이 일반 수행에서 파일을 읽는 오

버헤드와 재배치, 재구성을 수행하는 오버헤드를 합친 것보다 더 커야 한다. 즉, 위의 [그림 9]에서 ①+②의 시간이 ③+④의 시간보다 커야 최종적인 성능 향상을 기대할 수 있다. 따라서 구현에 있어 파일을 읽는 오버헤드와 재배치, 재구성 과정을 효율적으로 설계하는 것이 선행 컴파일 방식의 중요한 해결 과제가 된다.

선행 컴파일 방식은 기계어 코드를 저장하는 시점에 따라 서버 선행 컴파일(S-AOTC, Server Ahead-Of-Time Compilation)과 클라이언트 선행 컴파일(C-AOTC, Client Ahead-Of-Time Compilation)로 나뉜다. 서버 선행 컴파일은 선행 수행을 서버 측에서 수행하고 생성된 기계어 코드를 클라이언트가 받아 와서 수행하는 방식을 의미한다. 클라이언트 선행 컴파일은 서버에서 소스 코드를 받아서 클라이언트 측에서 유휴시간에 미리 선행 수행을 하고 실제 수행에서는 저장된 기계어 코드를 사용하는 방식이다[11]. 서버 선행 컴파일은 서버의 하드웨어 자원을 사용하여 최적화된 양질의 기계어 코드를 생성할 수 있고 클라이언트 측에서는 선행 수행 없이 일반 수행을 바로 할 수 있다는 장점이 있다. 그러나 서버에서는 클라이언트의 하드웨어 환경을 정확히 파악할 수 없기 때문에 선행 수행을 통해 생성된 기계어 코드의 신뢰성을 보장할 수 없는 한계가 존재한다. 한편 클라이언트 선행 컴파일은 클라이언트의 하드웨어 자원을 사용하여 선행 수행을 해야 하지만 같은 하드웨어 환경 내에서 기계어 코드를 생성하기 때문에 신뢰성이 보장되는 장점이 있다.

본 연구에서는 기계어 코드의 신뢰성이 보장되는 클라이언트 선행 컴파일 방식을 SFX 자바스크립트 엔진에 도입하였다. SFX 엔진의 적시 컴파일러에서는 소스 코드가 Bytecode 형태를 거쳐 기계어 코드로 변환되기 때문에 선행 컴파일을 Bytecode 생성 이후와 기계어 코드 생성 이후의 두 곳에 각각 적용할 수 있다. 다음 장에서는 선행 컴파일러를 설계하고, 중간 언어(Bytecode)와 기계어 코드 단계에서의 선행 컴파일러를 구현하여 그 성능을 측정하고 선행 컴파일 방식의 해결 과제에 대해 살펴본다.

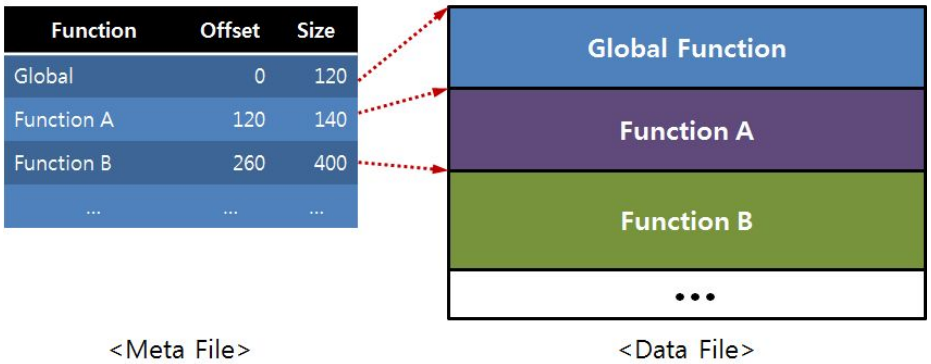
3.2. 선행 컴파일러 설계



[그림 10] 선행 컴파일러의 구조

[그림 10]은 선행 컴파일러의 전체 구조로 기존 적시 컴파일러 방식에 AOTC 모듈을 추가한 모습이다. SFX 엔진에서는 선행 컴파일을 적용하는 시점에 따라 중간 언어(Bytecode)와 기계어 코드의 두 단계로 구분할 수 있으며, AOTC 모듈은 두 곳 모두에서 사용된다.

AOTCManager 모듈은 AOTCWriter 모듈과 AOTCLoader 모듈을 전반적으로 관리하는 모듈이다. AOTCManager 모듈은 자바스크립트 엔진이 소스 코드를 처음 수행할 때 해당 코드에서 실행되는 함수들 중 선행 컴파일 된 함수들의 정보를 읽고 관리하고 있다. 그 후 각 함수의 컴파일이 필요할 때마다 관리하고 있던 선행 컴파일 된 함수들의 정보에서 해당 함수의 정보를 찾아보고 AOTC 데이터를 읽어오거나 기록한다. 또한 AOTCWriter에서 저장할 데이터를 버퍼에 모아 전달하면 이 데이터를 직접 파일에 기록하거나, 파일의 데이터를 읽어와 AOTCLoader에 전달하는 역할도 수행한다.



[그림 11] AOTC 파일 구조

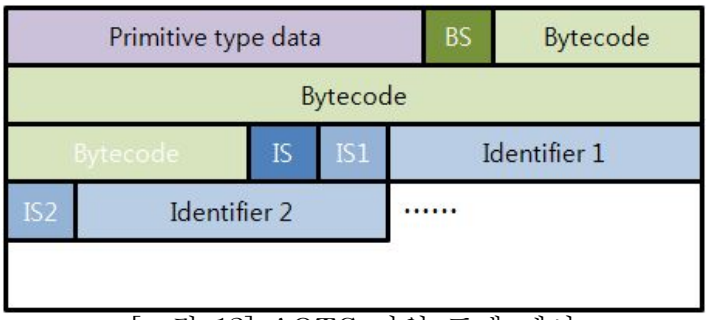
하나의 자바스크립트 실행 문맥에 대해 AOTCManager는 해당 문맥에서 실행되는 함수들을 선행 컴파일 하여 그 결과를 meta 파일과 data 파일에 나누어 저장한다[12]. 현재 SFX 엔진은 함수 단위로 컴파일하고 수행하고 있으므로 선행 컴파일 역시 함수 단위로 파일에 저장하고 읽는다. data 파일에는 자바스크립트 함수가 컴파일 된 정보(AOTC 데이터)가 저장되어 있고, meta 파일에는 각각의 함수에 대한 AOTC 데이터가 data 파일의 어느 위치에 저장되어 있는지를 표시해 둔다. 이를 도식화하면 [그림 11]과 같다.

또한 meta 파일에는 data 파일에 저장된 함수들을 구별할 수 있는 고유한 정보가 포함되어야 한다. 함수의 이름을 구분 기준으로 사용하면 이름이 없는 함수를 처리할 수 없기 때문에 자바스크립트 소스 코드 파일의 URL과 함께 소스 코드 내에서 해당 함수의 위치 정보를 구분 기준으로 사용한다. 또한 자바스크립트에서는 동일한 함수를 일반 함수로 정의할 수도 있고 하나의 객체로 정의할 수도 있는데 두 경우 컴파일 된 결과가 다르기 때문에 컴파일이 함수 호출을 위한 것인지 아니면 객체 생성을 위한 것이었는지의 정보도 meta 파일에 함께 저장한다.

자바스크립트 엔진이 동작을 시작하면 AOTCManager가 해당 소

스 코드에 대한 meta 파일을 읽어와 선행 컴파일 된 함수의 정보를 해시 테이블의 형태로 관리한다. 해시 테이블의 키 값으로 위에서 설명한 URL과 함수의 위치 정보를 사용하고, 해시 테이블의 데이터에는 함수에 대한 AOTC 데이터의 data 파일 내 위치가 저장된다.

이후 함수의 컴파일이 필요한 시점에서 해당 함수가 선행 컴파일 되어 AOTC 파일에 저장되어 있다면, 해시 테이블을 통해 AOTCLoader에서 data 파일에 저장된 AOTC 데이터를 불러와 자료구조를 재구성, 재배치 하는 과정을 거쳐 바로 수행한다. 반면에 선행 컴파일이 되어 있지 않다면, 컴파일을 수행하고 AOTCWriter 모듈을 통해 data 파일과 meta 파일에 정보들을 기록한다.



[그림 12] AOTC 파일 포맷 예시

AOTCWriter는 저장할 자료들을 특정한 포맷에 맞추어 정해진 순서대로 저장한다. primitive type의 정보들은 바로 파일에 저장이 가능하지만 vector에 저장된 정보들은 vector의 크기를 함께 저장하여 차후 Loader에서 정확하게 데이터를 읽어올 수 있도록 한다. [그림 12]는 저장되는 파일의 포맷을 도식화 한 것으로 몇 가지 타입의 정보들이 저장되는 모습을 나타낸다.

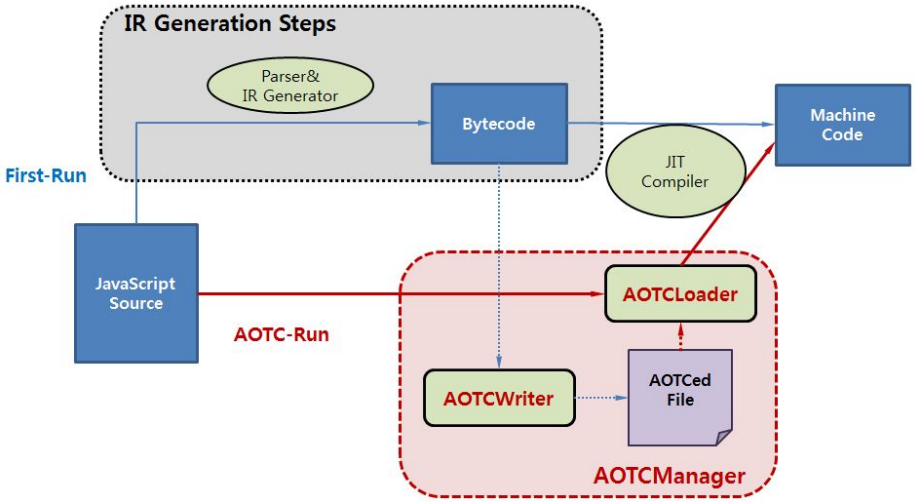
AOTCLoader에서는 data 파일에서 AOTC 데이터를 불러와 필요한 자료구조를 재구성하고 기계어 코드 내에 주소값들을 유효한 값

으로 재배치한다. 데이터를 읽어 올 때는 AOTCWriter에서 정보들을 저장하는 순서에 맞춰서 차례대로 읽어온다.

중간 언어 단계에서의 선행 컴파일러는 선행 수행에서 생성된 Bytecode와 자료 구조들을 AOTCWriter를 통해 저장한다. 그리고 일반 수행에서 AOTCLoader를 통해 AOTC 데이터를 읽어 Bytecode를 복원하고 필요한 자료 구조들을 재구성한다. 복원과 재구성이 끝나면 적시 컴파일러를 통해 기계어 코드를 생성하고 이를 수행한다.

기계어 코드 단계에서의 선행 컴파일러는 선행 수행에서 최종적으로 생성된 기계어 코드와 자료 구조들을 함께 저장한다. 그리고 일반 수행에서 AOTCLoader를 통해 AOTC 데이터를 읽어 필요한 자료 구조들을 재구성하고 기계어 코드를 복원한 후 기계어 코드 내 주소값들을 재배치 해준다. 그 뒤에는 추가적인 컴파일 과정 없이 복원된 기계어 코드를 바로 수행한다.

3.3. 중간 언어 단계 선행 컴파일러



[그림 13] 중간 언어 단계 선행 컴파일 과정

중간 언어(Bytecode) 단계에서 선행 컴파일(IR-Level AOTC)을 적용하면 [그림 13]과 같은 수행 과정을 거친다. 선행 수행(FirstRun)의 경우 자바스크립트 소스 코드가 Parser와 Bytecode Generator를 거쳐 Bytecode 로 변환되면, 적시 컴파일러(JITC)를 통해 기계어 코드로 컴파일 되기 전에 AOTCManager에게 Bytecode 및 관련 자료를 전달한다. 그러면 AOTCManager는 받은 자료들을 AOTCWriter를 통해 보조 기억 장치에 AOTC 파일로 저장해 둔다.

한편 동일한 소스 코드를 다시 수행하는 일반 수행(AOTCRun)의 경우 AOTCManager는 먼저 해당 소스 코드의 AOTC 파일이 존재하는 지를 검사한다. AOTC 파일이 존재한다면 AOTCManager는 이를 판별하여 AOTCLoader를 통해 파일에서 Bytecode와 관련 자료를 읽어온 뒤 적시 컴파일러를 거쳐 수행한다. 이 과정을 통해 컴파일 과정 중 Parsing과 Bytecode 생성 단계를 생략할 수 있다.

```
class CodeBlock {
    ......
    int m_numCalleeRegisters;
    int m_numVars;
    int m_numCapturedVars;
    int m_numParameters;
    bool m_isConstructor;
    int m_thisRegister;
    ......
    bool m_isStrictMode;
    CodeType m_codeType;
    SourceProvider* m_source;
    unsigned m_sourceOffset;

    Vector<Instructions> m_instructions; // Bytecode

    Vector<GlobalResolveInfo> m_globalResolveInfos;
    Vector<unsigned> m_jumpTargets;
    Vector<Identifiers> m_identifiers;
    Vector<WriteBarrier<Unknown> > m_constantRegisters;
    Vector<WriteBarrier<FunctionExecutable> >
m_functionDecls;
    Vector<WriteBarrier<FunctionExecutable> >
m_functionExprs;

    SymbolTable* m_symbolTable;

    struct RareData {   // RareData
        Vector<HandleInfo> m_exceptionHandlers;
        Vector<WriteBarrier<RegExp> > m_regexps;
        Vector<Vector<JSValue> > m_constantBuffers;
        Vector<SimpleJumpTable> m_immediateSwitchJumpTables;
        Vector<SimpleJumpTable> m_characterSwitchJumpTables;
        Vector<StringJumpTable> m_stringSwitchJumpTables;
        Vector<ExpressionRangeInfo> m_expressionInfo;
        Vector<LineInfo> m_lineInfo;
        Vector<CallReturnOffsetToBytecodeOffset>
m_callReturnIndexVector;
    } m_rareData;
}
```

[그림 14] CodeBlock 자료 구조

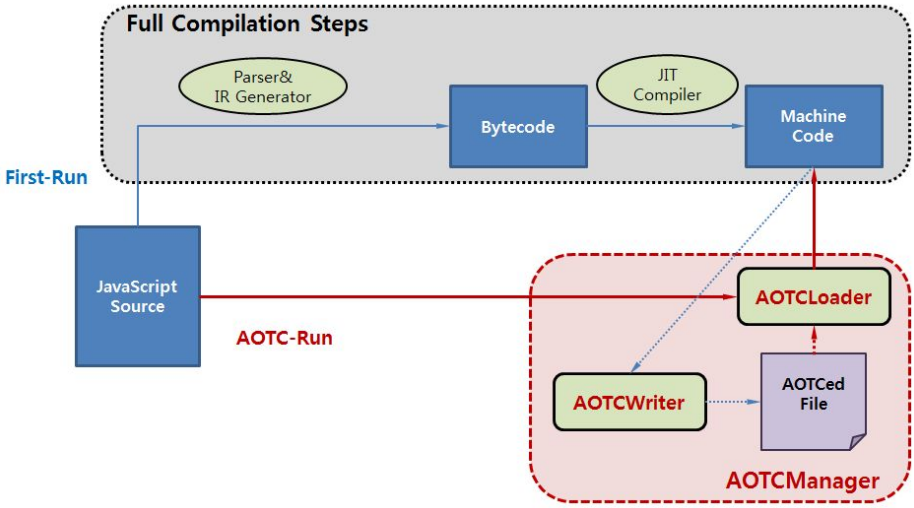
중간 언어 단계에서의 선행 컴파일러는 Bytecode 생성 과정이 끝난 후 생성되는 CodeBlock 자료구조를 저장한다. CodeBlock 자료구조에는 생성된 Bytecode를 비롯하여 [그림 14]와 같이 다양한 정보들이 저장된다. CodeBlock에 저장된 정보들은 Bytecode에서 기계어

코드를 생성하는데 쓰이거나, 기계어 코드의 수행 중 최적화 수행과 관련하여 참조된다.

선행 수행에서 AOTCWriter를 통해 Bytecode를 포함하여 CodeBlock 자료 구조의 정보들을 AOTC 파일에 저장하면, 일반 수행에서는 AOTCLoader를 통해 AOTC 파일을 읽어 Bytecode와 CodeBlock 자료 구조를 재구성 한다. 여기에선 기계어 코드를 저장하고 복원하지 않기 때문에 주소값 재배치의 작업은 하지 않는다.

중간 언어 단계에서의 선행 컴파일 방식은 Bytecode 단계에 선행 컴파일을 적용하여 전체 컴파일 과정 중에 Parsing, Bytecode 생성 과정을 생략한다. 전체 컴파일 과정을 생략하는 기계어 코드에서의 선행 컴파일 방식과 비교하면 얻는 성능 이득이 적지만, 저장하는 AOTC 데이터의 크기가 작고 AOTCLoader에서 수행하는 복원 및 재구성 작업이 간편하다는 장점이 있다.

3.4. 기계어 코드 단계 선행 컴파일러



[그림 15] 기계어 코드 단계 선행 컴파일 과정

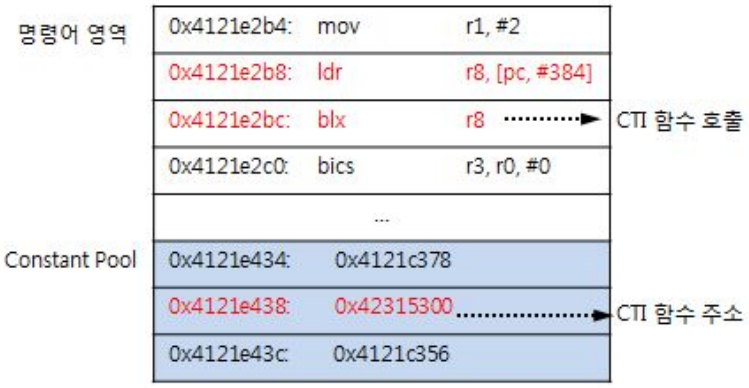
기계어 코드 단계에서 선행 컴파일(Full AOTC)은 중간 언어 단계에서의 선행 컴파일 방식에 기반을 두고 이를 확장하여 구현한다. 선행 수행(FirstRun)의 경우 자바스크립트 소스 코드가 Parser, Bytecode Generator, 적시 컴파일러(JITC)를 거쳐 최종적인 기계어 코드로 변환되면 AOTCManager에게 기계어 코드 및 관련 자료를 전달한다. 그러면 AOTCManager는 받은 자료들을 AOTCWriter를 통해 보조 기억 장치에 AOTC 파일로 저장해 둔다.

한편 동일한 소스 코드를 다시 수행하는 일반 수행(AOTCRun)의 경우 AOTC 파일이 존재한다면 AOTCManager는 이를 판별하여 AOTCLoader를 통해 파일에서 기계어 코드와 관련 자료를 읽어온 뒤 복원한 기계어 코드를 바로 수행한다. 이 과정을 통해 전체 컴파일 과정을 생략할 수 있다.

기계어 코드 단계에서는 총 세 가지의 데이터를 저장한다. 이 세 가지 데이터는 재구성이 필요한 자료구조(CodeBlock 포함), 기계어 코드, 재배치와 관련된 정보들이다. 이 중 CodeBlock 자료구조의 경우 적시 컴파일로 생성된 기계어 코드들은 수행 중에 CodeBlock 자료구조의 정보를 참조하기 때문에 중간 언어 단계의 선행 컴파일러와 동일하게 CodeBlock의 데이터를 저장해야 한다. 다만 Bytecode는 기계어 코드에서 더 이상 필요하지 않기 때문에 이 단계에선 제외된다.

재배치 정보들은 기계어 코드 내에서 사용되는 주소값과 관련된 정보들이다. 적시 컴파일러를 통해 생성된 기계어 코드는 수행 속도를 빠르게 하기 위해 함수 호출이나 객체의 정보에 접근할 때 타겟의 주소로 직접 분기하는 branch 명령어를 사용한다. 즉, 기계어 코드에는 현재 수행하고 있는 문맥의 함수나 객체의 주소 정보들이 포함된다. SFX에서 기계어 코드가 참조하는 주소는 기계어 코드에서 수행하기 어려운 부분을 보완해주는 C로 구현된 handler function(CTI 함수)과 SFX 내부에서 생성하는 여러 객체들이 있다.

이런 특성으로 인해 선행 수행에서 기계어 코드를 저장하고 난 뒤에 일반 수행에서 이를 불러와 해당 기계어 코드를 수행할 때 함수와 객체의 주소가 달라져 저장되었던 주소 정보가 유효하지 않게 된다. 따라서 AOTCLoader에서 저장된 파일을 읽어올 때 기계어 코드에서 참조하는 주소 정보들을 현재 수행에서 사용하는 주소로 교체해주는 재배치 작업을 반드시 해주어야 한다. 이를 위해 AOTC 파일에 기계어 코드 중 어느 곳에서 함수 및 객체의 주소값을 사용하는지, 그리고 이들이 어떤 데이터를 참조하는지에 대한 재배치 정보를 추가로 저장해야 한다.



[그림 16] ARM 기계어 코드에서 handler function 호출

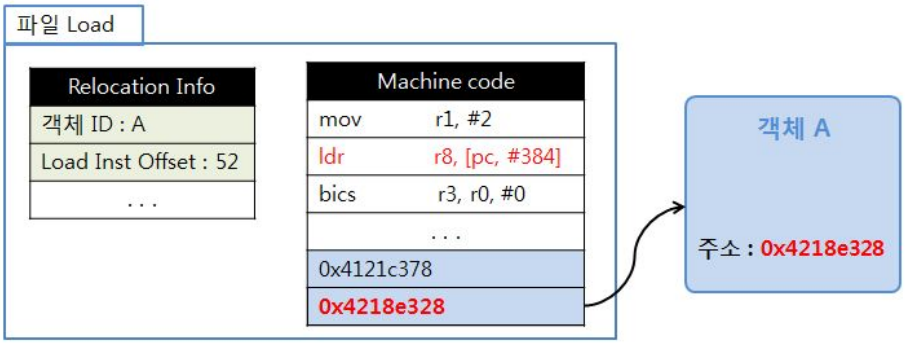
SFX에서 생성하는 ARM 기계어 코드는 [그림 16]과 같이 명령어 영역과 Constant Pool로 분리되어 있다. ARM에서는 operand로 사용할 값이 커서 32 비트 명령어의 immediate field에 채워 넣을 수 없는 경우 operand의 값을 Constant Pool에 저장하고 이를 LDR 명령어를 통해 레지스터로 불러오는 방식을 사용한다. ARM은 32 비트 크기의 주소를 사용하므로 기계어 코드에서 참조하는 주소값들은 모두 Constant Pool에 저장되고, LDR 명령어를 통해 접근된다.

따라서 AOTC 파일에 저장되는 주소값 관련 재배치 정보는 LDR 명령어의 위치와 참조하는 데이터 종류에 관한 ID 값이 된다. LDR 명령어의 위치는 기계어 코드의 시작 지점으로부터 offset 값으로 저장하고, ID 는 참조하는 데이터 별로 하나씩 정수 값을 부여해 저장한다. 즉, 기계어 코드에서 참조하는 handler function의 경우 함수 각각에 대해 고유 ID값을 할당하여 사용하고, 객체의 주소값도 마찬가지로 각 객체 별로 참조하는 객체 내부 데이터에 따라 ID 값을 부여해 사용한다. 기계어 코드에서 주소로 참조하는 객체의 종류와 각각에 할당된 ID 값은 아래 <표 1>에 정리되어 있다.

Object	참조하는 Object Member	ID
CodeBlock	Identifier	0
	FuncExpr	1
	FuncDecl	2
	RegExp	3
	GlobalObject	4
	CodeBlock	5
	ResolveInfo	6
GlobalData	FuncVPtr	0
	StringVPtr	1
	ArrayVPtr	2
JSValue	Constant Register	0
	Constant Register Address	1

<표 1> 기계어 코드에서 참조하는 객체와 할당된 ID 값

선행 수행에서 AOTCWriter를 통해 기계어 코드를 포함하여 재구성이 필요한 자료 구조와 재배치 관련 정보들을 AOTC 파일에 저장하면, 일반 수행에서는 AOTCLoader를 통해 AOTC 파일을 읽어 기계어 코드를 복원하고 재구성, 재배치 과정을 수행한다.



[그림 17] 주소값 재배치 예제

재배치 작업은 AOTC 파일에 저장된 재배치 정보를 이용하여 기계어 코드에서 사용하는 주소값들을 현재 수행에서의 주소값으로 맞춰주는 과정이다. [그림 17]은 주소값 재배치의 과정을 나타낸 예제로 AOTC 파일에는 기계어 코드와 함께 재배치 정보들이 저장되어 있다. 일반 수행에서 AOTC 파일에 저장되어 있는 기계어 코드를 복원하면, 그 뒤 재배치 과정을 수행하게 된다. 재배치 과정에서는 먼저 재배치 정보들 중 LDR 명령어의 위치 정보로부터 Constant Pool에 있는 주소값의 위치를 파악한다. 그리고 같이 저장된 ID 값을 통해 대상 주소값을 현재 문맥에 맞는 주소값으로 바꿔준다.

기계어 코드 단계에서의 선행 컴파일 방식은 최종적으로 생성된 기계어 코드에 선행 컴파일을 적용하기 때문에 전체 컴파일 과정을 생략할 수 있다. 컴파일 과정 중 일부만 생략하는 중간 언어 단계에서의 선행 컴파일 방식과 비교하면 얻는 성능 이득이 훨씬 크지만, 저장하는 AOTC 데이터의 크기가 크고 AOTCLoader에서 수행하는 재배치 및 재구성 작업이 복잡하다는 단점이 존재한다.

3.5. 성능 측정 및 해결 과제

Board	OMAP 3530 Beagle Board
OS	Ubuntu Linux kernel 2.6.31
CPU	ARMv7 Cortex-A 600Mhz
Main Memory	256MB
Flash Memory	4GB
SFX Engine	revision 89682
Benchmark	V8 test

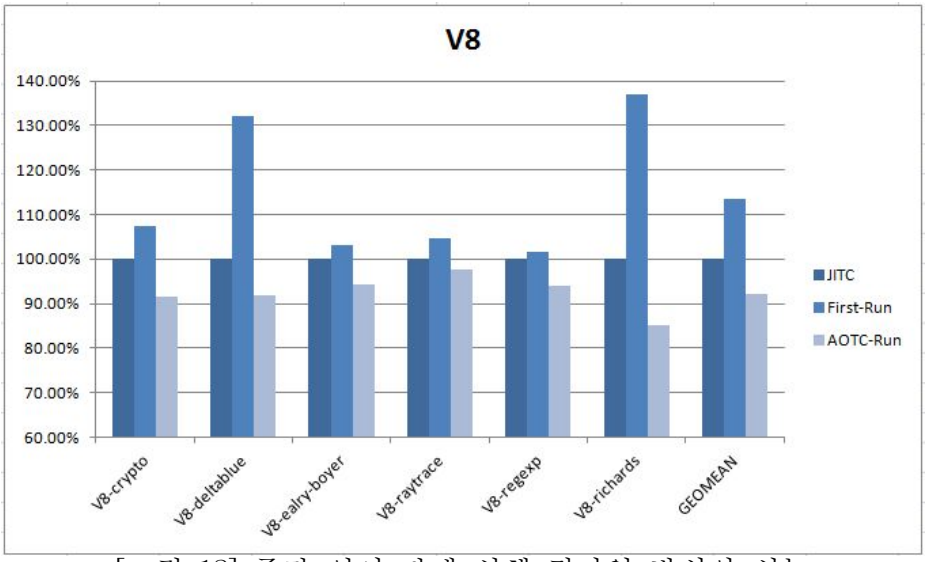
<표 2> ARM 아키텍처 실험 환경

앞에서 구현한 선행 컴파일러의 성능을 측정하기 위해 현재 모바일 환경에서 많이 쓰이고 있는 ARM 아키텍처에서 실험을 진행하였다. 자세한 실험 환경은 <표 2>에 나와 있다.

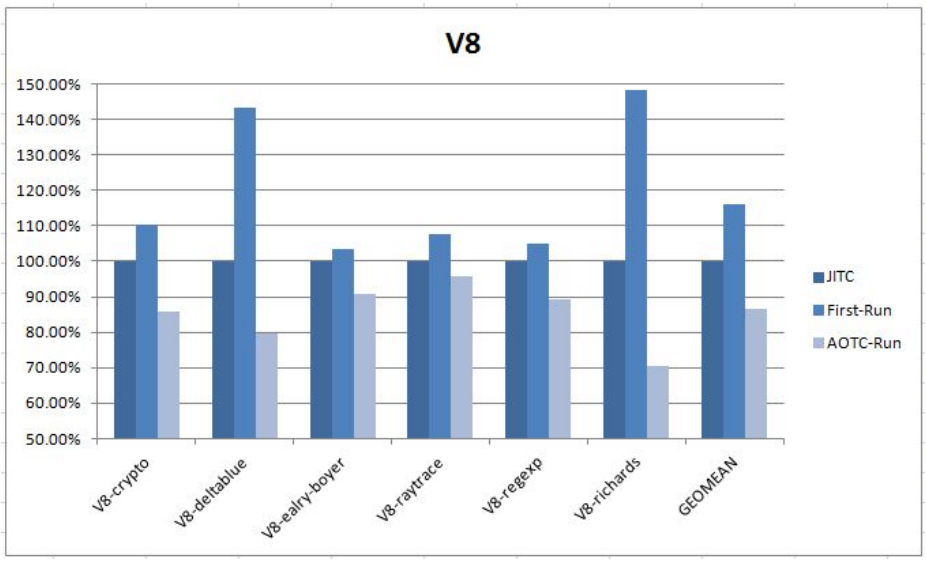
실험은 대표적인 자바스크립트 벤치마크인 V8를 수행하여 걸린 시간을 측정하였다. 실험 대상으로는 세 가지 방식을 설정하고 이들 간의 성능을 비교, 분석하였다. 세 가지 방식에는 기존 적시 컴파일 방식의 SFX, 중간 언어 단계에서 선행 컴파일을 구현한 SFX, 기계어 코드 단계에서의 선행 컴파일을 구현한 SFX가 속한다. 기존 적시 컴파일 방식의 수행 시간을 기준으로 두고 중간 언어 선행 컴파일과 기계어 코드 선행 컴파일의 수행 시간을 비교하였다.

또한 성능 외에도 선행 컴파일 방식을 적용했을 때 AOTC 파일이 사용하는 보조 기억 장치의 저장 공간을 측정하였다. 여기에선 자바스크립트 소스 코드의 크기를 기준으로 AOTC 파일의 크기를 비교하였다.

실험 결과를 통해 중간 언어 단계에서의 선행 컴파일 방식과 기계어 코드 단계에서의 선행 컴파일 방식은 성능과 저장 공간 사용에 있어서 상반된 특징을 가지는 것을 확인할 수 있었다.



[그림 18] 중간 언어 단계 선행 컴파일 방식의 성능

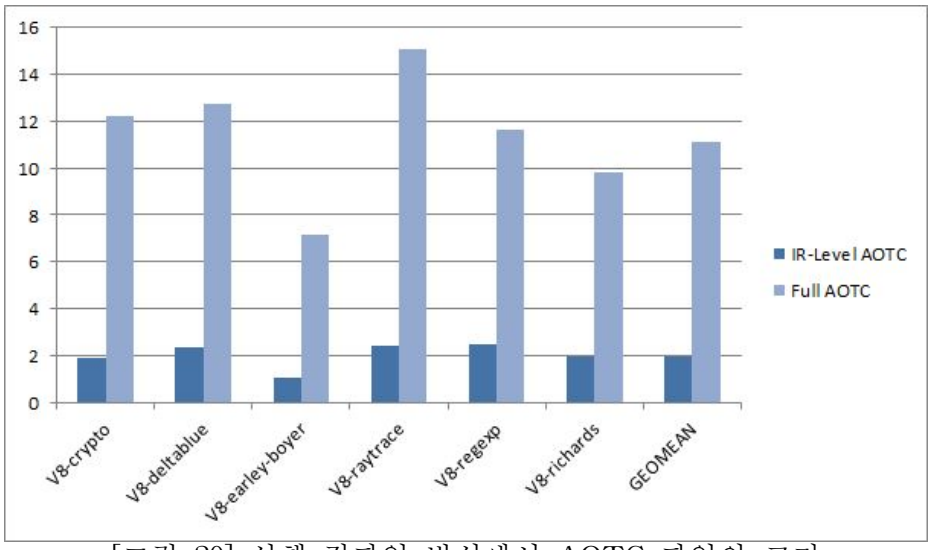


[그림 19] 기계어 코드 단계 선행 컴파일 방식의 성능

[그림 18]은 중간 언어 단계 선행 컴파일러와 기존 적시 컴파일러 방식을 비교한 결과이다. 그래프에서 JITC는 적시 컴파일러 방식을 나타내고 First-Run은 중간 언어 선행 컴파일러의 선행 수행을, AOTC-Run은 일반 수행을 의미한다. V8 벤치마크 결과를 살펴보면 모든 벤치마크에서 중간 언어 선행 컴파일 방식이 기존 적시 컴파일 방식보다 성능 효과가 있음을 확인할 수 있다. 선행 수행(Pre-Run)에서는 AOTC 데이터들을 파일에 저장하는 오버헤드로 인해 적시 컴파일 방식보다 더 느린 성능을 나타냈지만, 일반 수행(AOTC-Run)에서는 컴파일 오버헤드가 줄어서 평균 7.71%의 성능 개선 효과가 있었다.

[그림 19]는 기계어 코드 단계 선행 컴파일러와 기존 적시 컴파일러 방식을 비교한 결과이다. V8 벤치마크 결과를 살펴보면 앞의 중간 언어 단계 선행 컴파일 방식과 동일하게 모든 벤치마크에서 성능 효과가 있었다. 기계어 코드 단계 선행 컴파일 방식의 선행 수행에서는 AOTC 파일을 저장하는 오버헤드로 인해 성능이 더 느려졌지만, 일반 수행에서는 평균 15.21%의 성능 향상이 있었다.

기계어 코드 단계 선행 컴파일 방식에서 일반 수행을 할 경우 전체 컴파일 과정을 생략하기 때문에 중간 언어 단계 선행 컴파일 방식보다 더 좋은 성능을 나타낸다. 그러나 기계어 코드 단계 선행 컴파일은 전체 컴파일 과정에서 생성되는 필요한 자료구조들과 기계어 코드, 재배치·재구성 정보들을 다 저장해야 하므로 중간 언어 단계 선행 컴파일보다 AOTC 파일을 위한 저장 공간이 많이 필요하다.



[그림 20] 선행 컴파일 방식에서 AOTC 파일의 크기

[그림 20]은 자바스크립트 소스 코드의 크기를 기준으로, 중간 언어 선행 컴파일 방식과 기계어 코드 선행 컴파일 방식에서 생성한 AOTC 파일의 크기를 나타낸다. 중간 언어 선행 컴파일 방식에서는 소스 코드 파일의 크기를 기준으로 평균 2.33배의 저장 공간을 사용하고, 기계어 코드 컴파일 방식에서는 평균 11.57배의 저장 공간을 사용한다. PC와 달리 보조 기억 장치의 용량이 충분하지 않은 내장형 시스템에서 위의 선행 컴파일 방식을 적용할 경우, AOTC 데이터를 위해 사용해야 하는 공간이 늘어난다는 문제가 있다.

앞선 결과를 종합해 볼 때, 기계어 코드 선행 컴파일과 중간 언어 선행 컴파일은 각각 성능과 저장 공간의 사용 측면에서 장단점을 지니고 있음을 알 수 있다. 기계어 코드 선행 컴파일의 경우 성능은 좋지만 저장 공간을 많이 사용한다. 반면 중간 언어 선행 컴파일의 경우 저장 공간은 적게 사용하지만 기계어 코드에 비해서는 성능이 낮다. 따라서 본 논문에서는 이러한 결과로부터 기계어 코드 선행 컴파일과 중간 언어 선행 컴파일의 장점을 취하여 최적의 성능 효과를 얻을 수 있는 선택적 선행 컴파일 방식을 제안하고자 한다.

제4장 선택적 선행 컴파일러

본 장에서는 선행 컴파일 방식을 보완하기 위해 선택적 선행 컴파일(Selective AOTC) 방식을 제안하고 실험을 통해 그 효과에 대해 살펴본다.

4.1. 선택 휴리스틱

	성능 이득	**저장 공간 사용**
중간 언어 선행 컴파일	낮다	적음
기계어 코드 선행 컴파일	높다	많음

<표 3> 각 선행 컴파일 방식의 특징

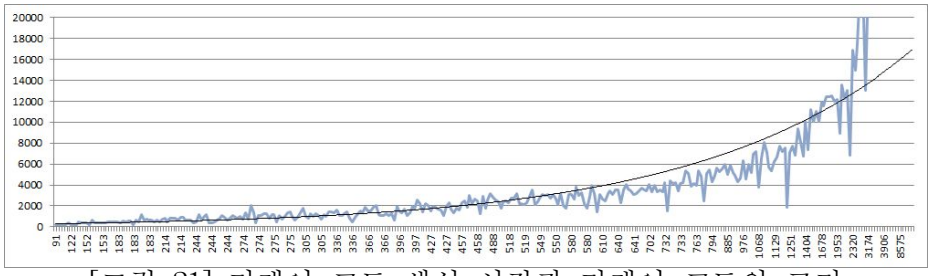
위의 <표 3>은 3장에서 구현한 중간 언어 단계 선행 컴파일과 기계어 코드 단계 선행 컴파일의 특징을 정리한 것이다. 중간 언어 선행 컴파일은 기계어 코드 선행 컴파일에 비해 낮은 성능 향상을 보이지만 AOTC 파일에서 사용하는 저장 공간이 적다. 반대로 기계어 코드 선행 컴파일은 중간 언어 선행 컴파일에 비해 높은 성능 향상을 보이지만 많은 양의 저장 공간을 사용한다.

선택적 선행 컴파일러는 저장 공간 사용의 효율을 지닌 중간 언어 선행 컴파일의 장점과 성능이 좋은 기계어 코드 선행 컴파일의 장점을 함께 적용하는 방식이다. 선택적 선행 컴파일러는 저장 공간 사용은 줄이면서도 최적의 성능을 내는 것을 목표로 한다.

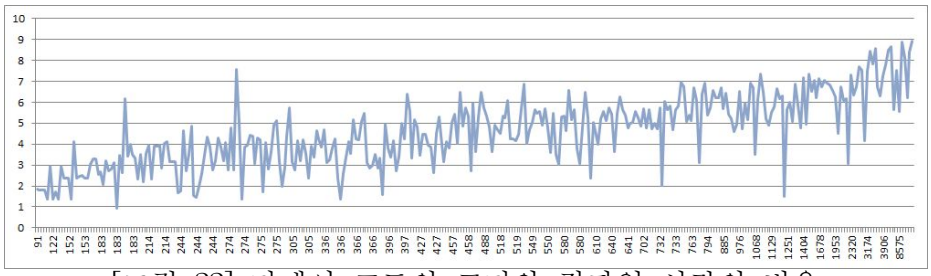
이를 위해 선택적 선행 컴파일러는 함수를 단위로 하여, 각 함수

별로 중간 언어 선행 컴파일 방식과 기계어 코드 선행 컴파일 방식을 선택적으로 적용한다. 이때 선행 컴파일 방식의 선택에 관련된 기준이 선택 휴리스틱이다. 선택 휴리스틱은 각 함수별로 중간 언어 선행 컴파일과 기계어 코드 선행 컴파일 방식을 적용할 때 얻는 이득을 비교하여, 더욱 큰 효과를 볼 수 있는 선행 컴파일 방식을 선택한다.

여기선 성능의 효율을 우선시하여, 기계어 코드 컴파일 방식을 적용하기에 적합한지의 여부를 기준으로 각 선행 컴파일 방식의 이득을 비교한다. 즉, 기계어 코드 컴파일 방식을 적용하여 얻는 효과가 크다면 기계어 코드 컴파일 방식의 이득이 크다고 판단하고 반대의 경우에는 중간 언어 컴파일 방식의 이득이 더 크다고 간주한다.



[그림 21] 기계어 코드 생성 시간과 기계어 코드의 크기



[그림 22] 기계어 코드의 크기와 컴파일 시간의 비율

[그림 21]은 V8 벤치마크를 수행했을 때 각 함수 별로 기계어 코드를 생성하는 컴파일 시간과 생성된 기계어 코드의 크기 사이의 관계를 나타낸다. [그림 21]에서 가로축은 컴파일 시간, 세로축은 생성된 기계어 코드의 크기를 나타낸다.

[그림 22]는 V8 벤치마크에서 각 함수별로 생성된 기계어 코드의 크기(바이트 단위)와 컴파일 시간(us 단위)의 비율값을 나타낸다. 이때 평균값은 0.235(byte/us)이며, 평균값과 편차가 큰 함수들도 다수 분포하고 있다.

위의 두 실험 결과를 종합하면 기계어 코드의 크기와 컴파일 시간은 대체적으로 비례 관계이지만, 비율에 있어서는 편차가 존재함을 확인할 수 있다. 여기서 컴파일 시간과 기계어 코드 크기의 비율은 기계어 코드 선행 컴파일을 적용할 때의 효율을 의미한다. 성능 효과가 크지만 저장 공간을 많이 사용하는 기계어 코드 선행 컴파일 방식은 기계어 코드 컴파일 시간이 길고 생성된 기계어 코드의 크기가 작은 함수에 적용할 때 큰 효과를 볼 수 있다. 반면에 기계어 코드 컴파일 시간이 짧고 생성된 기계어 코드의 크기가 큰 경우에는 기계어 코드 선행 컴파일 방식을 적용하여 얻는 효과가 미미하기 때문에 중간 언어 선행 컴파일 방식을 적용하는 것이 적합하다.

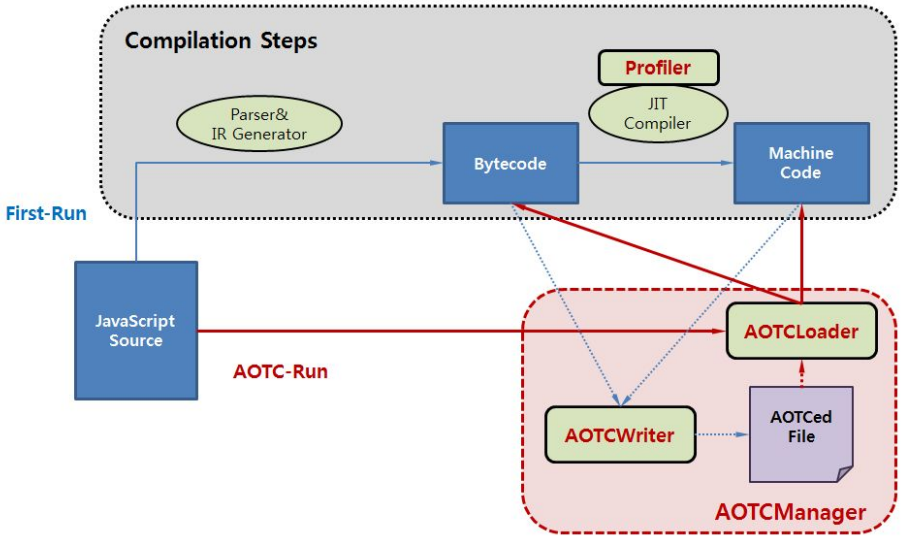
$$\frac{\text{Machinecode Compile Time}}{\text{Machinecode Size}}$$

[그림 23] 선택 휴리스틱의 기준

선택 휴리스틱의 기준을 수식으로 정리한 것이 [그림 23]이다. 선택 휴리스틱은 기계어 코드 컴파일 시간(Machinecode Compile Time)과 생성된 기계어 코드 크기(Machinecode Size)의 비율을 기준으로 삼는다.

선행 수행에서 함수를 컴파일 때 컴파일 시간과 기계어 코드 크기의 비율을 구하여 이 값이 특정 Threshold 값을 넘을 경우, 이는 휴리스틱에 의해 기계어 코드 선행 컴파일을 적용하기에 효율적이라 판단되어 기계어 코드 단계에 선행 컴파일을 적용하고 반대의 경우에는 중간 언어 단계에 선행 컴파일을 적용한다.

4.2. 선택적 선행 컴파일러 설계



[그림 24] 선택적 선행 컴파일러의 구조

[그림 24]는 선택적 선행 컴파일러의 전체 구조로, 3장에서 구현한 중간 언어 선행 컴파일러와 기계어 코드 선행 컴파일러를 함께 적용한 모습이다. 또한 선택 휴리스틱과 관련된 정보를 얻기 위해 적시 컴파일러(JITC)에 Profiler 모듈이 추가되었다. Profiler 모듈은 선택 휴리스틱과 관련하여, 기계어 코드 컴파일 시간과 기계어 코드의 크기 정보를 수집한다.

선택적 선행 컴파일 방식에서 선행 수행(First-Run)의 경우 일단 자바스크립트 소스 코드가 Parser, Bytecode Generator, 적시 컴파일러를 거쳐 최종적인 기계어 코드로 변환되는 전체 컴파일 과정을 수행한다. 그 뒤 Profiler 모듈을 통해 기계어 코드의 컴파일 시간과 생성된 기계어 코드의 크기 정보를 얻어 선택 휴리스틱의 비율값을 계산하고, 이 값을 토대로 선행 컴파일 방식을 선택적으로 적용한

다. 이 비율값이 미리 정한 특정 Threshold 보다 작을 경우 중간 언어 단계 선행 컴파일을 적용하여 Bytecode와 재구성과 관련된 자료 구조들을 AOTC 파일에 저장한다. 반대로 비율값이 Threshold 보다 클 경우 기계어 코드 단계 선행 컴파일을 적용하여 생성된 기계어 코드와 함께 재구성, 재배치 관련 정보들을 AOTC 파일에 저장한다.

일반 수행(AOTC-Run)의 경우 AOTCManager는 AOTC 파일에서 해당 함수에 관한 AOTC 데이터를 읽어온다. 그리고 저장된 AOTC 데이터의 종류를 판별하여 중간 언어 선행 컴파일 방식이 적용된 AOTC 데이터일 경우, Bytecode를 복원하고 자료 구조를 재구성 한 뒤 적시 컴파일러를 거쳐 수행한다. 반대로 기계어 코드 선행 컴파일이 적용된 경우, 기계어 코드를 읽고 재배치·재구성 과정을 거쳐 복원된 기계어 코드를 바로 수행한다.

선택적 선행 컴파일러는 선행 수행에서 AOTC 데이터를 저장할 때 해당 데이터가 중간 언어 선행 컴파일이 적용된 것인지, 아니면 기계어 코드 선행 컴파일이 적용된 데이터인지를 알려주는 태그 정보를 같이 저장한다. 그리고 일반 수행에서 AOTC 데이터를 읽어오면 위의 태그 정보를 먼저 판별하여 중간 언어 선행 컴파일이 적용된 경우 Parsing, Bytecode 생성 단계를 생략하고 적시 컴파일 과정을 수행하고, 반대로 기계어 코드 선행 컴파일이 적용된 경우에는 전체 컴파일 과정을 생략하고 복원된 기계어 코드를 바로 수행한다.

선택 휴리스틱을 적용할 때 비교의 기준이 되는 Threshold 값은 엔진 내부에 특정값으로 미리 지정되어 있다. Threshold 값을 정하는 방식에는 여러 가지가 있지만 여기서는 미리 구한 소스 코드 내 함수들의 선택 휴리스틱 비율의 평균값을 사용한다.

4.3. 실험 결과

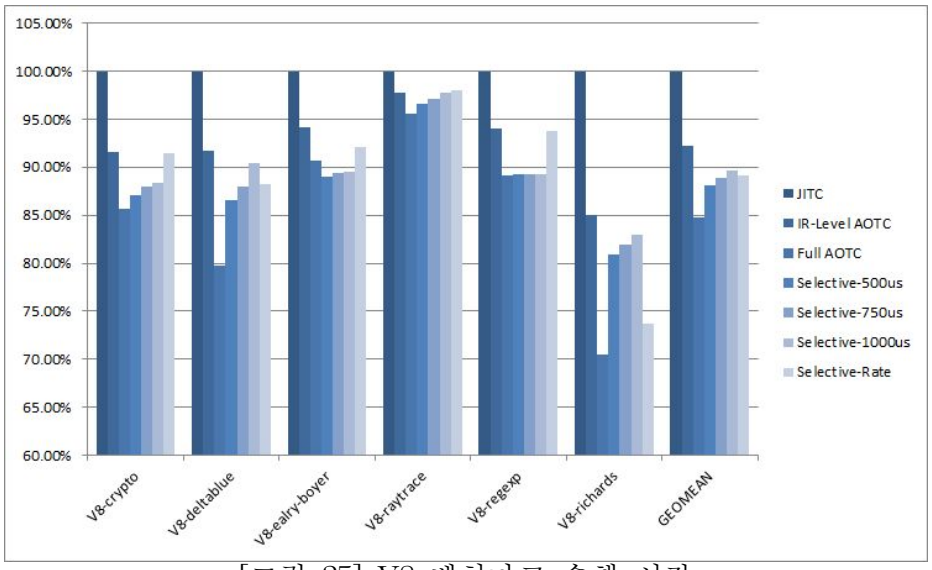
Board	OMAP 3530 Beagle Board
OS	Ubuntu Linux kernel 2.6.31
CPU	ARMv7 Cortex-A 600Mhz
Main Memory	256MB
Flash Memory	4GB
SFX Engine	revision 89682
Benchmark	V8 test

<표 4> ARM 아키텍처 실험 환경

선택적 선행 컴파일러의 효용성을 확인하기 위해 현재 모바일 환경에서 많이 쓰이고 있는 ARM 아키텍처에서 실험을 진행하였다. 자세한 실험 환경은 <표 4>에 나와 있다.

실험을 진행한 SFX 엔진은 총 네 가지이다. 여기엔 기존 적시 컴파일러 방식을 적용한 엔진, 선행 컴파일을 적용한 엔진, 컴파일 시간과 기계어 코드 크기에 관한 비율로 선택 휴리스틱을 적용한 엔진, 컴파일 시간만을 기준으로 선택 휴리스틱을 적용한 엔진이 속한다. 실험은 대표적인 자바스크립트 벤치마크인 V8를 수행하여 수행 시간과 저장 공간 사용량, 기계어 코드 선행 컴파일이 적용된 함수의 개수를 측정하였다.

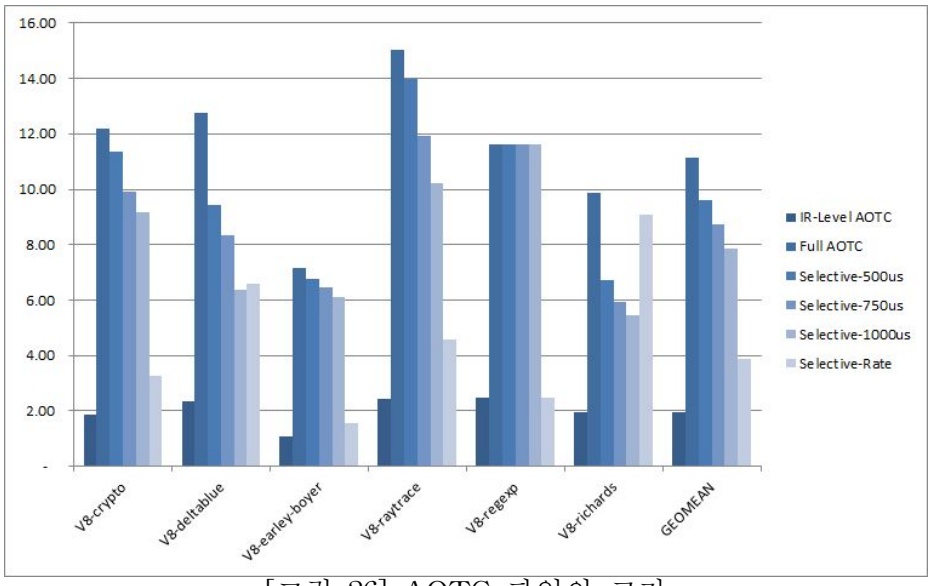
컴파일 시간을 기준으로 한 선택 휴리스틱은 각 함수의 기계어 코드 컴파일 시간과 특정 Threshold값을 비교하여 선행 컴파일 방식을 선택적으로 적용한다. 여기서 컴파일 시간은 기계어 코드 선행 컴파일을 적용하여 얻는 성능 이득을 의미한다. 컴파일 시간을 기준으로 한 휴리스틱은 저장 공간 사용량은 고려하지 않고 단순히 컴파일 시간만을 기준으로 정한 휴리스틱으로, 앞에서 제시한 비율을 기준으로 한 휴리스틱의 효과를 검증하기 위해 실험에 추가하였다.



[그림 25] V8 벤치마크 수행 시간

컴파일 시간을 기준으로 선택 휴리스틱을 적용한 엔진의 경우 Threshold 값을 500us, 750us, 1000us으로 나누어 실험하였다. 그리고 앞 절에서 제시한 컴파일 시간과 생성된 기계어 코드 사이의 비율을 선택 휴리스틱의 기준으로 적용한 엔진에선 0.2(byte/us)의 값을 Threshold로 정하고 실험하였다.

[그림 25]는 각 방식을 적용한 엔진의 일반 수행에서 V8 벤치마크 수행 시간을 나타낸다. 실험 결과 기존 적시 컴파일 방식의 수행 시간을 기준으로 기계어 코드 선행 컴파일은 15.21%, 중간 언어 선행 컴파일은 7.71%, 500us를 Threshold로 한 선택적 선행 컴파일은 11.89%, 750us를 Threshold로 한 선택적 선행 컴파일은 11.15%, 1000us를 Threshold로 한 선택적 선행 컴파일은 10.37%, 마지막으로 비율을 기준으로 한 선택적 선행 컴파일은 10.84%의 성능 향상이 있었다. 선택적 선행 컴파일을 적용한 엔진들은 중간 언어 선행 컴파일과 기계어 코드 선행 컴파일 방식을 같이 적용하기 때문에 중간 언어 선행 컴파일과 기계어 코드 선행 컴파일 결과들 사이에 위치한 성능을 나타냈다.

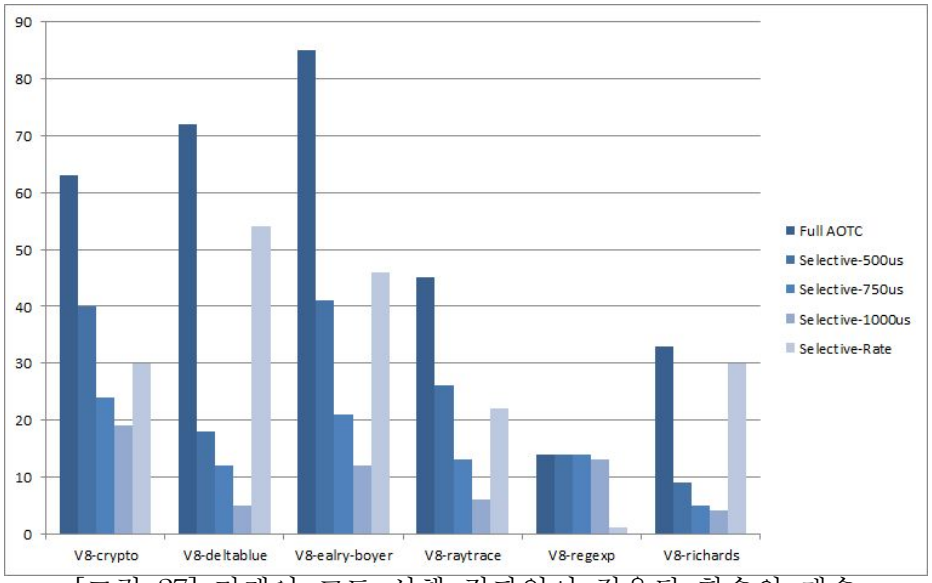


[그림 26] AOTC 파일의 크기

[그림 26]은 각 방식의 엔진에서 AOTC 파일로 사용하는 저장 공간의 크기를 나타낸다. 실험 결과 자바스크립트 소스 파일의 크기를 기준으로 중간 언어 선행 컴파일은 1.95배, 기계어 코드 선행 컴파일은 11.15배, 500us를 Threshold로 한 선택적 선행 컴파일은 9.61배, 750us를 Threshold로 한 선택적 선행 컴파일은 8.71배, 1000us를 Threshold로 한 선택적 선행 컴파일은 7.84배, 마지막으로 비율을 기준으로 한 선택적 선행 컴파일은 3.9배의 저장 공간을 AOTC 파일로 사용하였다. 선택적 선행 컴파일 방식을 적용한 엔진들은 앞의 [그림 25]의 결과와 비슷하게 중간 언어 선행 컴파일과 기계어 코드 선행 컴파일 결과들 사이의 크기를 AOTC 파일로 사용하고 있었다.

[그림 25]와 [그림 26]의 결과를 종합해보면 컴파일 시간을 휴리스틱의 기준으로 하는 선택적 선행 컴파일에서는 Threshold의 값이 커질수록 기계어 코드 선행 컴파일이 적용되는 함수의 비중이 줄어 중간 언어 선행 컴파일의 결과와 비슷해지는 경향을 보인다. 그리고 비율을 적용한 선택적 선행 컴파일과 컴파일 시간을 기준으로 하는

선택적 선행 컴파일을 비교해보면 성능 측면에서는 비슷한 효과를 보인다. 그러나 AOTC 파일의 크기에서는 비율을 기준으로 하는 선택적 선행 컴파일이 컴파일 시간을 기준으로 하는 선행 컴파일보다 절반 이하의 공간을 사용하는 것을 확인할 수 있다. 이를 토대로 본 연구에서 제안한 비율을 기준으로 하는 선택적 선행 컴파일 방식은 기계어 코드 선행 컴파일보다 적은 저장 공간을 사용하면서 동시에 중간 언어 선행 컴파일보다 높은 성능을 보여, 성능과 저장 공간 사용에 있어 효과적임을 알 수 있다.



[그림 27] 기계어 코드 선행 컴파일이 적용된 함수의 개수

[그림 27]은 V8 벤치마크에서 기계어 코드 선행 컴파일이 적용된 함수의 개수를 나타낸다. 비율을 적용한 선택적 선행 컴파일에서는 기계어 코드 선행 컴파일이 적용되는 함수의 개수가 특별한 경향을 나타내지는 않았다. 이 결과는 기계어 코드 선행 컴파일이 적용되는 함수의 개수는 선택적 선행 컴파일 방식에서 중요한 요소가 아니며, 함수 별로 어떤 선택 휴리스틱을 적용하는지가 전체 성능과 깊은 관련이 있음을 보여준다.

제5장 결론 및 향후 과제

최근 들어 웹앱(Web App), RIA(Rich Internet Application)의 등장으로 인해 웹 환경에서 자바스크립트의 비중이 크게 증가하면서 자바스크립트 엔진의 수행 성능이 중요한 이슈가 되고 있다. 성능을 향상시키기 위해서 기계어 코드를 생성하고 이를 수행하는 적시 컴파일러 방식이 도입되었는데, 이 방식은 컴파일 시간이 수행 시간에 그대로 포함되는 특성을 지닌다. 이 특성으로 인해 컴파일 오버헤드의 비중이 높은 실제 웹 환경에서는 적시 컴파일러 방식으로 두드러지는 성능의 향상을 기대하기 힘든 것이 사실이다.

본 연구의 목적은 기존 적시 컴파일 방식을 개선하여 자바스크립트 엔진의 성능을 향상시키는 것이다. 이를 위해서 컴파일 오버 헤드를 줄여서 전체 수행 성능을 향상시키는 선행 컴파일 방식을 SFX 엔진에 적용하였다.

선행 컴파일 방식을 중간 언어 단계와 기계어 코드 단계에 각각 적용한 결과 컴파일 오버헤드를 줄여 수행 성능을 향상시킬 수 있었지만, 보조 기억 장치의 저장 공간을 사용해야 하는 문제점을 보였다. 또한 중간 언어 선행 컴파일과 기계어 코드 선행 컴파일 방식은 성능 향상과 저장 공간 사용에 있어 서로 상반되는 특징을 나타내서 이를 그대로 적용하기엔 무리가 있었다.

선택적 선행 컴파일 방식은 선행 컴파일 방식을 보완하여 적은 저장 공간을 사용하면서 동시에 최적의 성능 효과를 얻는 것을 목표로 한다. 컴파일 시간과 기계어 코드의 크기를 바탕으로 선택 휴리스틱을 적용하여 함수 별로 중간 언어 선행 컴파일과 기계어 코드 선행 컴파일 방식을 선택적으로 적용하였다. 그 결과 10.8%의 성능 향상과 동시에 기계어 코드 선행 컴파일보다 60.9% 줄어든 저장 공간을 사용하는 것을 확인할 수 있었다. 선택적 선행 컴파일을 통해 컴파일 오버헤드를 줄여 기존 적시 컴파일의 성능을 개선하였고 선

행 컴파일 방식의 문제점인 저장 공간의 사용을 효율적으로 줄일 수 있었다.

후속 연구에서는 선행 컴파일 방식과 관련된 여러 가지 최적화 방안을 구현할 수 있다. 그 가운데서도 서버 선행 컴파일 방식(Server-AOTC)과 멀티 쓰레드(Multi-Thread) 수행 방식에 대한 연구가 구체화 될 수 있다.

서버 선행 컴파일 방식은 서버에서 선행 수행을 하여 강력한 하드웨어 성능을 바탕으로 최적화 수준이 높은 기계어 코드를 생성하고, 클라이언트에서는 서버에서 선행 수행 결과 생성된 기계어 코드를 받아 그대로 사용하는 방식이다. 이를 통해 클라이언트에서는 선행 수행 없이 최적화 수준이 높은 기계어 코드를 바로 수행할 수 있다.

다음으로 멀티 쓰레드 수행은 멀티 코어 환경에서 쓰레드를 나누어 선행 컴파일을 수행하는 방식이다[13]. 일반 수행에서 기계어 코드를 수행하는 쓰레드와 파일에서 기계어 코드를 읽어오는 쓰레드를 분리하여 동시에 수행하면 성능 향상의 최적화에 기여할 수 있다.

본 연구에서 제시한 선택적 선행 컴파일 방식은 자바스크립트 엔진의 성능 향상을 위한 새로운 대안이 될 수 있다. 웹이 진보함에 따라 자바스크립트의 비중이 증가하고 있는 현재의 웹 환경을 위해서 선행 컴파일 방식과 함께 성능 관련 최적화 연구를 후속으로 진행할 필요가 있다.

참 고 문 헌

[1] R. Chugh, J. Meister, R. Jhala and S. Lerner, Staged Information Flow of JavaScript, In Proceedings of PLDI, 2009

[2] J. Aycock, A brief history of Just-in-Time, ACM Computing Surveys, 35(2), 2003

[3] TraceMonkey, https://wiki.mozilla.org/JavaScript:TraceMonkey

[4] V8, http://code.google.com/p/v8

[5] SFX, http://webkit.org/

[6] G. Richards, S. Lebresne, B. Burg and J. Vit다, An Analysis of the Dynamic Behavior of JavaScript Programs, In Proceedings of PLDI, 2010

[7] 정동헌 외, 내장형 시스템을 위한 자바 AOTC의 설계와 구현, 추계학술발표회 논문집, 한국정보과학회, 2005

[8] Bytecode, http://en.wikipedia.org/wiki/Bytecode

[9] Abstract Syntax Tree, http://en.wikipedia.org/wiki/Abstract_syntax_tree

[10] P. Ratanaworabhan, B. Livshits, B. Zorn, JSMeter: Comparing the Behavior of JavaScript Benchmarks with RealWeb Applications, In USENIX Conference, 2010

[11] SungHyun Hong, et. al, Java Client Ahead-of-Time Compiler for Embedded Systems, LCTES, 2007

[12] 오형석, 문수묵, 자바스크립트 적시 컴파일러를 위한 생성 코드 재사용, 한국컴퓨터종합학술대회 논문집, 한국정보과학회, 2010

[13] B. Igor, J. Tobias, K. Edler, K. Staphen, F. Bjorn and T. Nigel, Generalized Just-in-Time Trace Compilation using a Parallel Task Farm in a Dynamic Binary Translator, In Proceedings of PLDI, 2011

Abstract

JavaScript language which is one of scripting language, is expressive and easy to use even for non-expert and is widely used for client-side programming in Web environment. In particular, due to appearance of Web App and RIA, JavaScript portion of Web environment has been increased.

As mobile environment is getting critical in IT industry, a performance of mobile becomes important. A performance of Web service is one of the most important factors, because web services are commonly used in mobile devices. But dynamic property of JavaScript limits its performance, especially in mobile environment

Just-in-Time Compiler(JITC) is developed for JavaScript performance. JITC makes machine code from JavaScript source code and execute machine code during runtime. However, In JITC method, compile overhead is included during runtime and it's portion is significant in the Web environment.

In this paper, we adapted Ahead-of-Time Compiler(AOTC) in order to improve the performance of JavaScript engine. AOTC saves machine code generated during compilation, and then reuse it. AOTC method can decrease compile overhead which is disadvantage of JITC. After the implementation of AOTC method, we found that it improved engine performance. But AOTC also need storage space in secondary storage, so it is hard to adapt AOTC method.

To overcome the limits of AOTC method, we proposed Selective-AOTC which hasn't developed yet. By selectively

applying AOTC method, we can get maximum performance while using less storage space. To implement Selective-AOTC, we proposed select heuristic which use compile overhead and machine code size information. Our Selective-AOTC had 10.8% performance gain and decreased 65.1% of storage usage compared to JITC method in V8 benchmark.

Keywords : JavaScript, JavaScript Engine, Just-in-Time Compilation, Ahead-of-Time Compilation, Selective AOTC, Web Optimization