# 웹 애플리케이션 성능 개선을 위한 V8 JavaScript 엔진의 아키텍처 레벨 프로파일링

강윤지 O, 홍경환, 신동군
성균관대학교 전자전기컴퓨터공학과
oso41@skku.edu, redc7328@skku.edu, dongkun@skku.edu

# Architecture-Level Profiling of V8 JavaScript Engine for Improving the Performance of Web Application

Yunji Kang O, Gyeonghwan Hong, Dongkun Shin
Department of Electrical and Computer Engineering, Sungkyunkwan University

**요 약**

웹 애플리케이션은 플랫폼에 독립적이기 때문에 개발 및 보수 비용이 낮다는 강점을 지니고 있다. 그러나 웹 애플리케이션은 동적 언어인 JavaScript로 작성되기 때문에 네이티브 애플리케이션과 비교하여 성능이 열세하다는 단점을 지니고 있다. V8 JavaScript 엔진은 인라인 캐싱 기법을 도입하여 타입 검사의 비용을 감소시켰음에도 불구하고 전체 애플리케이션의 수행시간에서 큰 비중을 차지하고 있다. JavaScript 엔진의 최적화 지점을 찾기 위하여 본 논문에서는 V8의 각 구간을 아키텍처 레벨로 프로파일링하는 도구를 제작하였다. 또한 V8의 인라인 캐싱이 효율적으로 수행하는 지 확인하고, 인라인 캐싱 miss에서 병목 구간과 원인을 살펴보았다.

## 1. 서 론

모바일 기술이 급속도로 성장하고 스마트 디바이스의 활용 범위가 점차 확대되면서 웹 기술은 기존 웹 페이지 형태에서 벗어나 웹 애플리케이션으로 발전하고 있다. HTML5, CSS, JavaScript로 구성된 웹 애플리케이션은 플랫폼 및 디바이스에 독립적이라는 특성이 있어 개발 및 유지보수 비용이 감소한다는 장점을 지니고 있다. 그러나 웹 애플리케이션은 네이티브 애플리케이션과 비교하여 성능이 열세하다는 단점을 지니고 있다. 특히 CPU 자원이 제한된 웨어러블 디바이스의 경우, 애플리케이션의 성능 열세는 더욱 무시할 수 없다.

웹 애플리케이션의 구성요소 중 JavaScript는 이미 구축된 HTML 페이지에 동적인 기능을 수행할 수 있도록 한다. JavaScript는 실행 중 객체의 타입이 변경 가능한 동적 언어이기 때문에 변수 접근 시 타입 검사가 필요하다. V8 JavaScript 엔진에서는 인라인 캐싱(Inline Caching; IC)[1] 기법을 도입하여 타입 검사의 과정을 최적화 하고 있다. 그럼에도 불구하고, JavaScript 처리 동작은 전체 웹 애플리케이션의 수행시간에서 약 절반 정도의 큰 비중을 차지하고 있었다.

본 논문에서는 웹 애플리케이션의 성능 병목을 파악하기 위하여 V8 JavaScript 엔진의 구간 별 성능과 인라인 캐싱의 성능을 분석하였다. 기존 연구[2, 3]에서는 단순히 수행시간과 instruction의 개수만을 측정 하였지만, 본 연구에서는 cycle 수와 cache miss와 같은 하드웨어 이벤트 정보를 수집해 아키텍처 레벨의 프로파일링을 수행하였다.

## 2. 웹 엔진

### 2.1 웹 렌더링 엔진의 동작 과정

Blink와 같은 웹 렌더링 엔진은 웹 애플리케이션의 HTML과 CSS 코드를 해석하여 화면에 출력하는 역할을 한다. 웹 엔진의 동작은 크게 Loading, Rendering, Painting, Scripting으로 구성되어 있다. Loading은 데이터를 불러와 HTML와 CSS 문서를 파싱하여 각각 DOM(Document Object Model)과 CSSOM(CSS Object Model)을 구성한다. Rendering은 DOM 트리와 CSSOM를 결합하여 렌더 트리를 구축하고 객체의 정확한 위치와 크기를 계산한다. Painting은 구축된 렌더 트리를 참고하여 화면에 픽셀을 그린다. 마지막으로 Scripting은 JavaScript 코드를 컴파일 하여 수행한다.

### 2.2 V8 엔진과 인라인 캐싱

V8은 동적 프로그래밍 언어인 JavaScript를 구동하는 엔진이다. V8는 최적화 없이 빠르게 컴파일을 수행하는 Full-Codegen 컴파일러와 자주 접근되는 함수에 최적화된 코드를 생성하는 Crankshaft로 구성되어 있다.

JavaScript에서는 실행 중에 변수의 타입을 동적으로 변경 가능하다. 변수는 속성의 집합인 객체로서 관리되며 각 속성은 이름과 값으로 구성되어 있다. 객체의 타입이 변경될 때 마다 속성의 객체 메모리 내 offset이 달라진다. 따라서 객체의 속성 값을 얻어올 때, 객체의 레이아웃(맵)을 탐색하여 해당 속성의 offset을 알아내야 한다. 이 부하를 줄이기 위하여 Full-Codegen 컴파일러에서는 인라인 캐싱(Inline Caching; IC)라는 최적화 기법을 사용한다. IC는 특정 호출 지점에 접근하는 객체들이 주로 같은 타입으로 사용된다는 점을 이용하여 객체의 맵과 해당 속성의 offset을 코드 내부에 캐싱해 다음 접근 시에 빠르게 속성을 접근할 수 있도록 한다.

예를 들어, 그림 1은 JavaScript 코드의 foo.bar의 값을

접근하기 위한 과정을 그린 것이다. 본 예제에서는 cond에 값에 따라 foo 객체가 Hello 또는 World로 정해지는 데, 두 객체는 다른 맵을 가지고 있다. 그림과 같이 IC의 Fast 경로에 Hello 객체의 맵과 속성의 offset 값이 캐싱 되어 있는 경우에 만약 foo가 Hello 객체로 assign 되었다면, 맵 탐색 없이 캐시 된 offset을 통하여 바로 객체의 속성 값을 얻을 수 있다. 반대로 foo라는 객체가 World로 assign 되었다면 LoadIC_Miss() 함수 수행을 통해 Slow 경로로 진입한다. 먼저 bar 속성의 offset을 찾기 위하여 맵을 검색한다. 그 후 탐색한 맵과 속성의 offset을 Fast 경로에 추가 하여 다음 코드 접근 시에 사용 가능하도록 한다. 마지막으로 검색한 offset에 존재하는 객체의 속성에 접근한다.

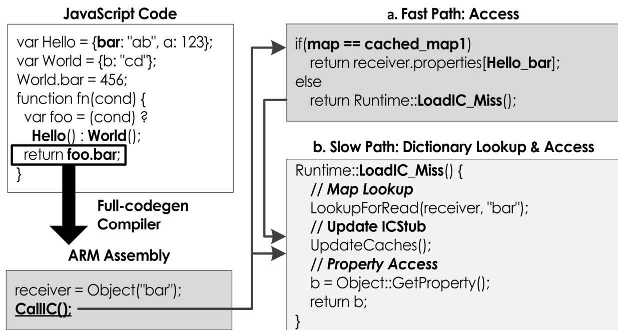



**그림 1 자바스크립트의 값 접근 과정**

IC는 크게 다섯 가지 상태로 나눌 수 있다. 첫 번째로 uninitialized는 맨 처음 초기 상태이며, 한 번 코드를 수행하면 pre-monomorphic 상태가 된다. 여기까지는 Fast 경로에 탐색 코드가 존재하지 않는다. 필요 없는 IC 코드 수행을 막기 위하여 코드가 두 번 이상 수행 되었을 때, IC 코드가 생성된다. monomorphic은 캐시 된 맵이 한 개 존재하는 상태이며, polymorphic은 여러 맵이 존재하는 상태이다. 마지막으로 megamorphic은 맵 비교 횟수를 줄이기 위해서 캐시 된 맵에 대한 해시 테이블을 사용한다.

값을 읽을 때, 속성은 크게 data 속성과 accessor 속성으로 구분된다. data 속성은 V8 객체 내부에 데이터 값이 존재한다. accessor 속성의 경우 accessor 함수가 정의되어 있어 속성을 읽을 때는 getter 함수가, 속성을 수정할 때는 setter 함수가 호출된다. 대표적인 accessor 속성으로는 DOM 객체가 있다. DOM 객체에 접근할 때에는 blink에 정의된 getter/setter 함수를 호출하여 값을 얻어오거나 수정한다.

## 3. 웹 애플리케이션 성능 분석

### 3.1 실험 환경 및 도구

웹 애플리케이션의 성능 분석을 위하여 Exynos Cortex-A9 쿼드코어 프로세서, 2GB 메모리를 사용하는 Odroid-U3 보드에서 실험하였다. 플랫폼으로는 안드로이드 4.4.4를 이용하였고, 웹 애플리케이션을 구동하기 위하여 Crosswalk 15버전을 사용하였다. Crosswalk는 안드로이드, 타이젠 등의 다양한 모바일 플랫폼에서 동작 하는 웹 애플리케이션 런타임 엔진이며, blink 엔진과 V8이 동작하는 크로미움 기반으로 구현되어 있다.

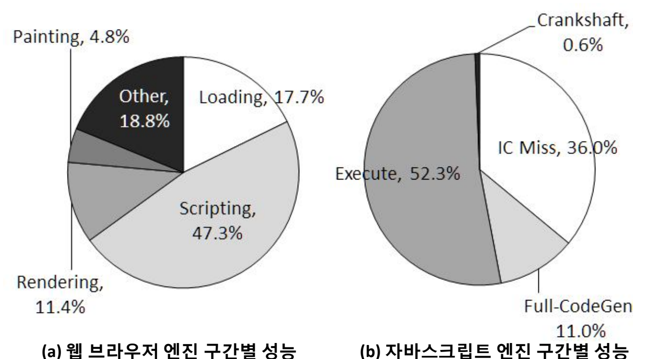
웹 애플리케이션의 동작 구간 별 성능을 측정하기 위하여 크로미움에서 제공하는 함수 트레이서 도구와, V8에서 제공하는 V8 프로파일링 도구를 사용하였다. 웹 애플리케이션으로는 웹 사이트를 로딩하는 bbench[4] 애플리케이션을 사용하였다.

또한 기존 도구에서 cycle 수와 cache miss와 같은 하드웨어 이벤트 정보를 추가로 프로파일링 하기 위해 Exynos Cortex-A9 프로세서가 제공하는 PMU(Performance Monitor Unit)을 이용하여 PMU Tracker 라는 도구를 제작하였다. 기존의 perf[5] 도구는 주기적으로 하드웨어 이벤트 값을 읽어서 프로파일링 하기 때문에 짧은 시간 수행되는 함수 단위로 프로파일링을 수행하기에 적합하지 않다. PMU Tracker는 함수의 시작과 끝의 하드웨어 이벤트 값을 수집하여 함수 수행 동안의 수행 시간 및 하드웨어 이벤트를 측정할 수 있다. 또한 context switch나 CPU migration을 포함한 스케줄링 이벤트를 감지하여 다른 프로세스를 배제한 정확한 값 측정을 수행할 수 있다.

### 3.2 웹 엔진의 구간 별 프로파일링

bbench 애플리케이션에서 ESPN 사이트를 로딩하며 웹 엔진의 각 구간의 실행시간을 측정하였다. 그림 2의 (a)는 크로미움에서 제공하는 함수 트레이서 도구와 PMU Tracker를 이용하여 웹 엔진의 메인 thread의 구간 별 성능을 측정한 것이다. PMU Tracker를 함께 이용하면, 함수 수행 도중의 context switch를 고려하여 성능을 측정할 수 있다. 실험 결과, JavaScript를 처리하는 "Scripting"을 수행하는 시간이 가장 많은 비중(약 50%)을 차지하는 것을 볼 수 있다.

그림 2의 (b)는 V8 프로파일링 도구와 PMU Tracker를 이용하여 V8 엔진의 구간 별 성능을 측정한 결과이다. "Execute"는 컴파일 된 코드를 수행하는 시간, "IC Miss"는 인라인 캐싱 수행 시에, 요청한 타입의 맵과 일치하는 캐싱된 맵이 없어서 slow 경로를 수행하는 시간을 의미한다. 가장 많은 수행 시간을 차지하는 것은 "Execute"이며, 두 번째로는 "IC Miss"가 차지한다.




(a) 웹 브라우저 엔진 구간별 성능 (b) 자바스크립트 엔진 구간별 성능

**그림 2 웹 엔진 구간별 성능**

좀 더 자세한 프로파일링을 위해 수행시간이 짧은 Crankshaft를 제외한 V8의 세 구간의 CPI(Cycle Per Instruction), I-Cache(Instruction Cache) miss, 그리고 D-Cache(Data Cache) miss를 측정하였다. (표 1) CPI와 cache miss를 파악하는 것은 코드 최적화를 할 때 좋은 참고자료가 된다. "Execute"는 수행시간의 비중이 큼에도 불구하고 CPI와 cache miss의 값들이 전체 애플리케이션의 평균과 차이가 나지 않는다. CPI가 낮은 경우, 함수와 데이터 접근의

지역성이 높다는 것을 의미하므로 “Execute”는 많은 cache miss로 인해 시간이 오래 걸리기 보다는 처리하려는 작업의 양이 많아 수행시간이 길다는 것을 알 수 있다. 반대로, “IC Miss”의 경우에는 CPI와 cache miss가 매우 높다. 성능을 높이기 위해서는 함수와 데이터 접근 패턴을 분석하여 cache 지역성을 높이는 방향으로 최적화를 해야 함을 알 수 있다.

표 1 V8 구간의 CPI 및 Miss Rate

| | CPI (Cycle/Inst.) | I-Cache Miss Rate | D-Cache Miss Rate |
|---|---|---|---|
| IC Miss | 2.82 | 6.1% | 3.1% |
| Full-CodeGen | 2.03 | 2.9% | 1.9% |
| Execute | 2.40 | 4.3% | 2.4% |
| Overall Application | 2.25 | 4.1% | 2.1% |

### 3.3 IC 프로파일링

#### 3.3.1. Fast 경로 프로파일링

IC의 성능을 분석하기 위하여 Fast 경로와 Slow 경로로 나누어 프로파일링을 수행하였다. Fast 경로에서는 IC의 hit율 및 hit 비용을 조사하였다. 그림 3은 애플리케이션 수행 동안의 Load IC miss 횟수와 캐싱된 맵 개수에 따른 hit 횟수를 나타낸 것이다. Map N, Hit M은 Fast 경로에 존재하는 캐싱된 맵의 개수가 N개 이며, 그 중 M번째 맵 비교에서 hit가 발생했다는 것을 의미한다. bbench를 수행하는 중의 IC hit율은 약 92%이며 전체 Load 수행의 약 71%는 monomorphic(Map 1) 상태에서 hit 된다.

Hit 비용은 캐싱된 맵들 중 일치하는 맵을 찾는 과정이다. 한 번의 맵 검색으로 hit가 발생하는 비율은 전체 IC의 74%, 두 번째 맵 검색으로 찾은 비율은 약 5% 였다. 또한 해시 테이블로 캐시된 맵을 찾는 megamorphic의 경우에는 약 12%를 차지하였다.

캐싱된 맵의 개수가 2 이상인 polymorphic에서 맵 개수와 hit 발생 위치의 관계를 보았을 때, 가장 최근에 추가된 맵에서 hit가 더 많이 발생하는 경향이 있었다. Map 2의 경우, 마지막에 추가된 두 번째 맵에서 약 70%의 확률로 hit가 발생한다. 특히 Store IC의 Fast 경로에서는 Store의 80%가 최근의 추가된 맵에서 hit가 발생하였다.

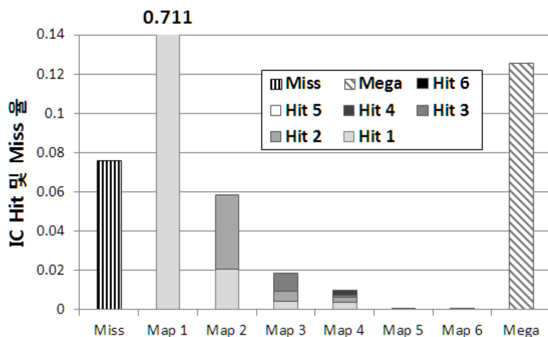



그림 3 Load IC의 Fast 경로에서의 Hit 및 Miss 율

#### 3.3.2 Slow 경로 프로파일링

IC는 속성 접근, 연산, 함수 호출 등에서 활용되고 있다. 프로파일링 결과, 속성을 얻는 Load IC miss의 수행시간이 전체 IC miss의 수행 시간 중 50%를 차지하였으며 CPI 값 또한 2.95로 높다. Load IC miss에서는 속성의 객체 내 메모리 offset을 찾기 위하여 먼저 “Map Lookup”을 시작한다. 그 다음 IC 코드를 추가하는 “Update ICStub” 과정을 거친 후 마지막으로 “Property Access”를 통하여 값을 읽어온다.

표 2는 Load IC의 구간별 성능과 cache miss를 나타낸 것이다. 세 구간 모두 높은 cache miss를 보이고 있다. 특히, Load IC miss에서 가장 많은 수행시간을 차지하는 “Property Access”는 Fast 경로에서도 수행하는 과정이다. (Fast 경로 수행은 “V8 Execute”에 포함) 이 구간을 자세히 분석한 결과, 속성 접근 중에서도 accessor 속성을 접근할 때 많은 수행시간을 소요한다. 이 때, 접근한 accessor 속성은 대부분 DOM 객체의 속성이다. DOM 객체의 값을 접근하기 위해 다양한 종류의 blink getter 함수를 호출하여 DOM 객체를 접근함으로써, cache 지역성이 떨어져 I-Cache 및 D-Cache miss가 크게 발생한다. 이러한 getter 함수 호출을 미리 예측하는 prefetching을 도입하여 cache miss rate를 감소시켜 성능 최적화가 가능할 것으로 예측된다.

표 2 Load IC 프로파일링

| | 전체 App 대비 수행시간 | I-Cache Miss Rate | D-Cache Miss Rate |
|---|---|---|---|
| Map Lookup | 1.8% | 6.4% | 5.9% |
| Update ICStub | 4.2% | 6.9% | 4.1% |
| Property Access | 7.7% | 6.1% | 3.4% |
| Overall Application | 100% | 4.4% | 2.8% |

## 4. 결론

본 논문에서는 웹 애플리케이션의 동작 구간별 프로파일링을 수행하여 성능 병목 지점을 찾아내었다. 프로파일링 결과, JavaScript 엔진의 “IC Miss” 처리 구간이 병목 구간임을 확인할 수 있었다. 특히 “Property Access”은 전체 수행 시간 대비 7.7%을 차지하며, CPI는 2.8으로 전체 어플리케이션의 평균 CPI 대비 높은 구간이다. 값을 접근하는 함수 및 데이터 패턴을 자세히 분석하여 cache 지역성을 높인다면 성능 향상에 큰 도움이 될 것으로 예측된다.

## 참고 문헌

[1] L. Peter Deutsch, et al. “Efficient Implementation of the Smalltalk-80 System”, In POPL, pp. 297-302, 1984.

[2] W. Ahn, et al. “Improving JavaScript performance by deconstructing the type system”, In PLDI. Vol. 49, No. 6, pp. 496-507, 2014.

[3] Dot, Gem, et al. "Analysis and Optimization of Engines for Dynamically Typed Languages", In SBAC-PAD, pp. 41-48, 2015.

[4] A. Gutierrez, et al. "Full-system analysis and characterization of interactive smartphone applications," In IISWC, pp. 81-90, 2011.

[5] S. Eranian, "perfmon2: A flexible performance monitoring interface for linux”, in Proceedings of the Linux Symposium, pp. 269-288, 2006.