SunSpider 벤치마크를 통한 자바스크립트 엔진의 성능 평가

(Performance Evaluation of JavaScript Engines Using SunSpider Benchmarks)

정 원 기 † 이 성 원 †
(Won Ki Jung) (Seong-Won Lee)

오 형 석 † 오 진 석 †
(Hyeong-Seok Oh) (Jin Seok Oh)

문 수 묵 ††
(Soo-Mook Moon)

요 약 최근 RIA(Rich Internet Application)의 등장으로 인해 자바스크립트 코드의 복잡도가 증가함에 따라 이를 위한 고성능 자바스크립트 엔진들이 경쟁적으로 발표되고 있다. 또한 이들 엔진의 성능 측정을 위한 도구로서 SunSpider 벤치마크가 널리 사용되고 있다. 본 논문에서는 대표적인 고성능 자바스크립트 엔진인 Mozilla의 TraceMonkey, Google의 V8, 그리고 Apple의 SquirrelFish Extreme에 대해 자바스크립트 코드 수행 방식을 비교하고 SunSpider 벤치마크를 이용해 각 엔진의 성능을 측정한다. 또한 각 엔진들의 수행 방식과 SunSpider 각각의 코드 특성을 토대로 하여 성능 결과를 분석하여 각 엔진의 장단점을 평가한다.

키워드 : 자바스크립트 엔진, RIA, SunSpider, 성능 평가

Abstract The recent deployment of RIA (Rich Internet Application) is often involved with the complex JavaScript code, which leads to the announcement of high performance JavaScript engines for its efficient execution. And the Sunspider benchmark is being widely used for the performance evaluation of these JavaScript engines. In this paper, we compare the execution methods of three high-performance JavaScript engines, Mozilla TraceMonkey, Google V8, and Apple SquirrelFish Extreme, and measure their performances using the SunSpider benchmark. We also evaluate the pros and cons of each engine, based on its execution method and the code characteristics of the SunSpider benchmarks.

Key words : JavaScript Engine, RIA, SunSpider, Performance Evaluation

1. 서 론

최근 인터넷 기술에서 눈에 띄는 변화로 Google Docs나 Maps와 같은 데스크탑 어플리케이션 수준의 웹 어플리케이션인 RIA(Rich Internet Application) 기술의 등장을 꼽을 수 있다. RIA 기술은 인터넷에서 웹 2.0의 개념이 주류로 등장하는 가운데 이를 가능하게 하는 수단으로서 주목받고 있다[1]. RIA를 구현하는 데 있어 현재 가장 많이 사용되는 웹 클라이언트 사이드 프로그래밍 언어로서 자바스크립트(JavaScript)를 꼽을 수 있다. 자바스크립트는 기존 웹 프로그래밍에서도 자주 사용되었지만 주로 DOM(Data Object Model) 객체에 접근하는 간단한 코드만을 수행하도록 사용되었던데 반해, 최근 페이지를 유지한 채로 클라이언트와 서버간의 데이터 전송을 가능하게 하는 AJAX(Asynchronous JavaScript and XML) 기술이 유용하게 사용되면서 자바스크립트의 활용 범위가 넓어지고 있다. 특히 RIA를 구현하는 데 있어 핵심적인 역할을 담당하게 됨에 따라 웹 어플리케이션에서 자바스크립트의 비중이 상당히 높아지게 되었다.

RIA 기술의 등장에 따라 자바스크립트 코드의 복잡도가 증가하면서 웹 브라우저에서 자바스크립트 코드를 실행하는 자바스크립트 엔진의 성능이 관심의 대상이 되고 있다. 이러한 변화에 발맞추어 최근 다양한 기법을 사용한 고성능 엔진들이 하나 둘씩 발표되고 있다. 현재 대표적인 자바스크립트 엔진으로는 Mozilla FireFox 3.1 브라우저에 내장된 TraceMonkey[2], Google Chrome 브라우저에 내장된 V8[3], 그리고 Apple WebKit 팀의 Safari 4.0 브라우저에 내장된 SquirrelFish Extreme을 꼽을 수 있다[4]. 이들 엔진들은 고성능 인터프리터를

· 본 연구는 서울형산업 기술개발 지원사업(NT080546)의 지원으로 수행되었음
· 이 논문은 제36회 추계학술발표회에서 '자바스크립트 엔진의 성능 평가'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 서울대학교 전기컴퓨터공학부
dream9903@altair.snu.ac.kr
oracle@altair.snu.ac.kr
jiingoh@altair.snu.ac.kr
swlee@altair.snu.ac.kr
†† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
smoon@snu.ac.kr

논문접수 : 2009년 12월 24일
심사완료 : 2010년 3월 2일

사용하거나 실행할 자바스크립트 코드를 실시간으로 기계어 코드로 번역하는 적시JIT(Just-in-Time) 컴파일러 등을 사용하여 자바스크립트의 빠른 수행을 꾀하고 있다.

또한 각 오픈 소스 진영에서 자바스크립트 엔진을 경쟁적으로 발표함에 따라 각 엔진의 성능을 평가하기 위한 벤치마크 툴에 대한 관심도 높아지고 있는데, 현재 알려져 있는 대표적인 자바스크립트 벤치마크 툴로서 WebKit SunSpider, Mozilla Dromaeo, Google V8 Test가 주로 사용되고 있고[5], 이 중에서 가장 먼저 발표된 SunSpider 벤치마크가 자바스크립트 엔진의 성능을 측정하는 툴로서 가장 널리 사용되고 있다.

SunSpider 벤치마크는 브라우저와는 별도로 독립적으로 동작하는 자바스크립트 엔진의 성능 측정을 위해 DOM 객체에 접근하는 코드를 제외한 순수 자바스크립트 코드를 사용하며, 암호화 알고리즘, 문자열 및 정규식 처리 관련, 이진 연산이나 수학 알고리즘 등을 반복 수행하는 26개의 세부 테스트 프로그램으로 구성되어 있으며[6], 이들 세부 프로그램은 간단한 연산을 많은 횟수 반복하는 코드와 복잡한 알고리즘을 수행하는 코드가 모두 존재하여 기존의 웹 페이지에서 쓰이던 간단한 자바스크립트 코드와 RIA에서의 복잡한 코드 모두에 대한 자바스크립트 엔진의 성능을 예측하는 데 유용한 도구로 쓰이고 있다.

본 논문에서는 TraceMonkey, V8, SquirrelFish Extreme의 세 가지 엔진의 수행 방식을 비교 분석한다. 또한 SunSpider 벤치마크를 이용해 각 엔진의 성능을 측정하고, 엔진들의 수행 방식과 SunSpider 벤치마크 코드 특성을 바탕으로 하여 성능 결과를 분석한다.

2. 자바스크립트 엔진의 수행 방식

본 절에서는 1절에서 언급한 세 자바스크립트 엔진의 동작 및 최적화 방식을 소개한다.

2.1 TraceMonkey(Mozilla FireFox)

TraceMonkey는 기본적으로 자바스크립트 소스 코드를 바이트코드 형식으로 변환한 뒤 명령어를 하나씩 실행하는 인터프리터 방식을 사용한다. 그러나 인터프리터를 수행하다가 루프와 같이 자주 반복되는 코드, 즉 핫 경로(hot path)를 만나면 이를 기계어로 번역하여 수행하는 선택적인 JIT 컴파일 방식을 사용한다[2].

핫 경로를 찾아 컴파일 하는 과정은 그림 1과 같이 수행된다. 먼저 루프와 같은 핫 경로는 주로 후방 분기(backward branch)의 대상이 되는 경우가 많으므로, 인터프리터로 바이트코드를 실행하는 도중 후방 분기를 특정 횟수 이상 만나게 되면 JIT 컴파일 모니터가 이 핫 경로의 시작으로 인식한다[7].

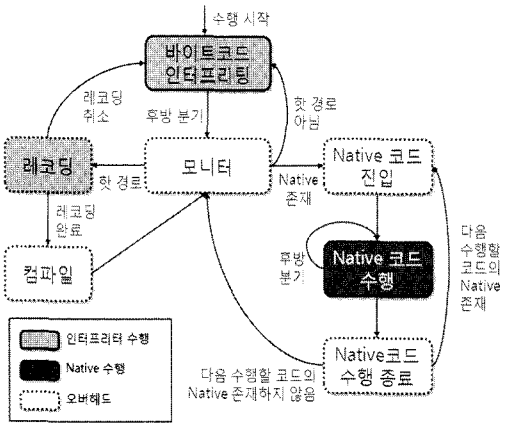

그림 1 TraceMonkey의 컴파일 방식

핫 경로에 해당하는 후방 경로를 만나 컴파일을 수행할 때 우선적으로 레코딩(recording) 작업을 수행하는데, 이는 핫 경로에 해당하는 바이트코드들을 다시 한 번 인터프리터로 수행하면서 각 바이트코드를 중간 코드 형태로 바꾸는 작업이다. 핫 경로에 대해 레코딩 작업을 하고 다시 후방 분기를 만나 루프의 시작 지점으로 되돌아오면 레코딩을 완료하고 모아두었던 중간 코드를 기계어로 컴파일한 뒤 기계어로 코드를 수행한다.

이와 같은 방식으로 컴파일을 수행하게 되면 자주 수행되지 않는 코드는 컴파일하지 않기 때문에, 컴파일로 인한 오버헤드가 적다는 장점이 있다. 또한 핫 경로에 대해서는 바이트코드를 기계어로 바꾸어 수행하기 때문에 인터프리터 방식에 비해 매우 높은 성능을 가져올 수 있다. 또한 핫 경로에 대해 변수의 타입 구체화(specialization) 기능을 사용해 컴파일된 코드의 빠른 수행을 돕는다. 기본적으로 자바스크립트는 변수 타입이 정해지지 않고 언제든 바뀔 수 있는 동적 타입(dynamic-typed) 언어이기 때문에 대부분의 연산들을 수행할 때 피연산자의 타입을 체크하여 연산을 수행하는 비효율적인 방식으로 수행되어야 한다. 그러나 TraceMonkey는 핫 경로를 컴파일할 때 현재 각 변수들이 가진 타입을 기반으로 최적화된 코드를 생성하고 수행 도중 이 타입이 변하지 않는지를 체크하는 머신코드만을 삽입한다. 실제로 루프의 수행 중에 변수의 타입이 바뀌는 것은 드문 일이므로 TraceMonkey의 이러한 방식의 코드 생성은 효율적인 수행을 가져온다[8].

2.2 V8(Google Chrome)

V8은 TraceMonkey와 달리 수행할 모든 자바스크립트 코드에 대해 JIT 컴파일을 사용하는 방식을 택한다[3]. 즉 인터프리터가 존재하지 않으며, 코드를 수행하다 함수 호출이 발생하면 호출된 함수가 이전에 컴파일되

지 않았다면 기계어로 컴파일하고 수행한다. 호출된 함수가 이전에 이미 컴파일 되었다면 컴파일된 코드를 호출하게 된다.

모든 자바스크립트 코드를 수행 도중 컴파일 하기 때문에 자주 호출되는 함수는 한 번 컴파일한 뒤 다시 사용할 수 있기 때문에 성능의 이득을 볼 수 있지만, 적게 호출되는 함수의 경우 오히려 컴파일로 인한 오버헤드가 발생할 수 있다. 이는 핫 경로에 대해서만 컴파일하는 TraceMonkey의 수행 방식과 비교될 수 있다.

2.3 SquirrelFish Extreme(Apple Safari)

SquirrelFish Extreme(이하 SFX)은 TraceMonkey와 마찬가지로 바이트코드 인터프리터의 형태로 자바스크립트 코드를 실행하던 기존의 SquirrelFish 엔진에서 발전된 형태로서, 컨텍스트 쓰레딩이라는 기법을 사용해 기존의 인터프리터에서 사용하던 코드를 이용해 컴파일된 코드를 만들어 내는 방식을 사용한다[4].

최근의 인터프리터들이 바이트코드를 수행할 때 다이렉트 쓰레딩(direct threading)이라는 방식을 사용하는데, 다이렉트 쓰레딩은 점프시 간접 분기(indirect branch)를 많이 사용하여 분기 예측(branch prediction)의 실패가 자주 일어나 x86, ARM과 같은 깊은 파이프라인 프로세서에는 많은 성능의 저하를 가져올 수 있다.

이러한 문제를 해결하기 위해 SFX에서는 그림 2와 같이 컨텍스트 쓰레딩(context threading) 방식을 사용한다[9]. 컨텍스트 쓰레딩은 다이렉트 쓰레딩과는 달리 각 바이트코드 처리 루틴을 하나의 함수로 구현하고, 각 바이트코드마다 해당 처리 루틴 함수를 호출하는 방식으로 바이트코드를 수행한다. 이때 함수의 리턴은 다이렉트 쓰레딩의 간접 분기보다 분기 예측이 쉽기 때문에 이득을 볼 수 있다. 이의 구현을 위해 바이트코드를 함수 호출로 표현한 컨텍스트 쓰레드 테이블(context threaded table)을 미리 작성하고 이를 수행한다.

함수 호출 명령어로 이루어진 컨텍스트 쓰레드 테이블을 만들 때 간단한 처리 루틴을 가지는 바이트코드의 경우에는 코드 영역의 바이트코드 처리 루틴 코드를 테이블에 복사하거나 처리 루틴과 같은 일을 하는 기계어 코드를 직접 테이블에 생성하기도 하기 때문에 컨텍스트 쓰레드 테이블을 생성하는 작업을 일종의 JIT 컴파일로 볼 수 있다.

컨텍스트 쓰레드 테이블을 만드는 과정은 V8과 마찬가지로 함수 단위로 이루어지며 각 함수가 처음 호출되었을 때 동적으로 만들어진다. 또한 TraceMonkey와 달리 수행할 모든 코드를 컴파일하기는 하지만, 컨텍스트 쓰레드 테이블을 만드는 과정이 비교적 단순하여 SFX의 JIT 컴파일은 모든 기계어 코드를 직접 생성하는 V8보다는 비교적 빠를 수 있다.

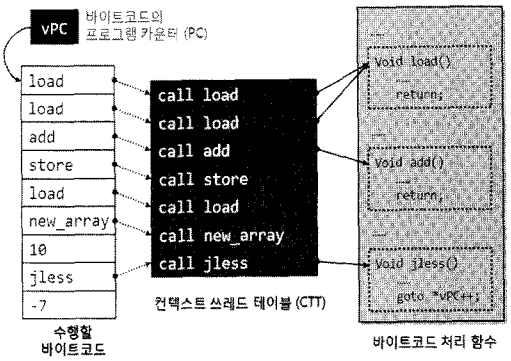

그림 2 SFX의 컨텍스트 쓰레딩 방식

3. SunSpider 벤치마크를 통해 측정한 자바스크립트 엔진의 성능 결과 및 비교 분석

그림 3은 SunSpider 벤치마크를 통해 자바스크립트 엔진의 성능을 측정한 결과이다. Intel Core2 Quad CPU 2.33GHz, 메인 메모리 4GB 환경에서 리눅스 기반 엔진들로 성능을 측정하였다. 성능 측정을 위해 사용한 엔진 버전은 TraceMonkey(GECKO_1_9_2_BASE), V8(1.3.7), SFX(Revision 48534)이다. 그래프의 가로축은 각각의 성능 측정을 위해 사용된 세부 벤치마크 프로그램의 이름이며, 세로축은 TraceMonkey의 JIT 컴파일 옵션을 끄고 순수 인터프리터만으로 실행했을 때를 기준으로 하여 각 엔진들이 몇 배나 빠른 성능을 보이는지를 나타내었으며, 높은 값일수록 좋은 성능임을 나타낸다. 성능 측정 결과 평균적으로 TraceMonkey가 기준보다 3.14배, V8이 4.92배, SFX가 4.32배 더 빠른 것으로 측정되었다.

이제 SunSpider의 각 벤치마크의 특성을 바탕으로 하여 엔진들의 수행 방식이 성능 결과에 어떠한 영향을 미치는지 분석해보도록 하겠다.

3.1 TraceMonkey와 다른 엔진들의 비교

먼저 선택적 컴파일을 수행하는 TraceMonkey와 수행할 모든 코드를 컴파일하는 V8, SFX를 비교해 보자.

TraceMonkey에서 월등히 높은 성능을 보인 Bitops-3bits-in-byte, Bitops-bitwise-and 벤치마크는 단순한 비트 연산을 많은 횟수 반복하는 코드로 이루어져 있다. 이들 벤치마크는 모두 하나의 짧은 루프만 존재하며, 이를 각각 12만 8천 회, 60만 회 반복한다. 많은 횟수 반복하는 짧은 루프의 경우 레코딩 오버헤드가 상대적으로 작고, 기계어 코드로 수행하는 구간이 길기 때문에 TraceMonkey에서 좋은 성능을 보인다. 마찬가지로 Math-cordic 벤치마크 역시 30만 회 반복하는 짧은 루프가 존재하여 좋은 성능을 보인다.

Math-spectral-norms 벤치마크는 수행하는 루프의

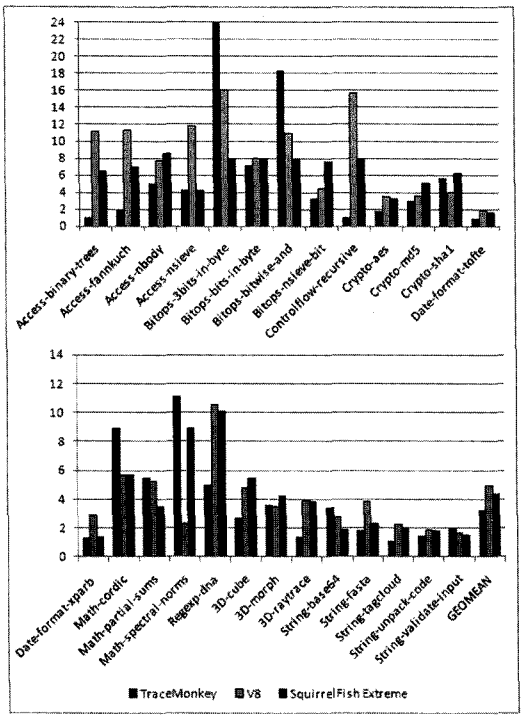

그림 3 SunSpider를 통해 측정한 자바스크립트 엔진의 성능 결과

개수가 많고 반복 횟수가 적어 상대적으로 레코딩 오버헤드가 크지만, 루프 내의 코드가 거의 함수 호출로 이루어진다는 특징이 있다. TraceMonkey는 함수 단위가 아닌 핫 경로 단위로 컴파일하기 때문에 루프 안의 함수 호출은 자동으로 인라인된다. 반면에 함수 단위로 컴파일하는 다른 두 엔진은 함수 호출을 할 때 컴파일 된 코드를 찾고 건너뛰는 작업 등 부가적인 절차가 필요하기 때문에 상대적으로 TraceMonkey보다 비효율적이다.

TraceMonkey에서 좋은 성능을 보이는 여섯 개의 벤치마크에서는 타입 구체화 기능이 잘 적용된다. Bitops의 두 벤치마크는 정수 타입으로, Math의 두 벤치마크에서는 double 타입, 그리고 String의 두 벤치마크에서는 문자열 타입으로 구체화가 이루어진다. 2.1절에서 언급했듯이 타입 구체화가 적용되는 경우 타입 검사를 하는 코드를 생략할 수 있어 보다 최적화된 기계어 코드를 생성할 수 있다.

반면에 몇몇 벤치마크들에서는 TraceMonkey의 성능이 가장 떨어지는 것으로 측정되었는데, 그 원인 중 한 가지로 레코딩 오버헤드를 꼽을 수 있다. 루프 내의 코드가 길고 반복 횟수가 적은 경우 기계어로 수행하는 구간에 비해 상대적으로 레코딩 오버헤드가 크게 작용한다. 대표적으로 Access-nbody, Crypto-aes, Crypto-md5, Date-format-tofte, 3D-cube, 3D-raytrace, String-unpack-code를 꼽을 수 있다.

또한 다중 루프의 경우 TraceMonkey는 외부 루프와 내부 루프에 대한 트레이스를 따로 만들어 관리하기 때문에 외부 루프와 내부 루프 간의 전환에 추가적인 작업이 필요하여 오버헤드가 작용한다. Bitops-bits-in-byte, Bitops-nsieve-bit, Access-fannkuch, Access-nseive, 그리고 Crypto-aes 벤치마크에서 다중 루프를 사용하여 낮은 성능을 보인다. 단, Bitops-bits-in-byte의 경우는 다중 루프로 인한 성능 저하가 있지만 타입 구체화의 효과를 받아 다른 엔진들과 비슷한 성능을 낸다고 볼 수 있다.

마지막 원인으로는 TraceMonkey가 핫 경로에 대해 레코딩을 시도하다가 실패하는 경우가 몇 가지 존재한다. 재귀호출의 경우 루프와 같이 취급될 수 있지만, 아직 TraceMonkey에서는 재귀호출에 대한 컴파일은 제공하지 않는다. 재귀호출이 사용된 벤치마크는 3D-cube, 3D-raytrace, Access-binary-trees, String-tagcloud, Controlflow-recursive의 5개이다. 또한 자바스크립트 내장 함수인 eval() 함수와 정규식을 사용한 String.replace() 함수가 포함된 루프에 대한 컴파일을 지원하지 않는다. 이들 함수가 사용된 벤치마크는 Date-format의 두 벤치마크와 String-unpack-code이다. 이들 총 8개 벤치마크에서는 원래 컴파일 되었어야 할 핫 경로를 컴파일하지 못하고 대부분의 코드를 인터프리터로 수행하기 때문에 TraceMonkey에서 낮은 성능을 보인다.

3.2 V8과 SFX의 비교

V8과 SFX 모두 수행할 모든 자바스크립트 코드에 대해 함수 단위로 컴파일 하는 방식을 사용한다는 점에서 비슷하지만, V8은 모든 함수에 대해서 직접 코드를 만들어 내기 때문에 컴파일 오버헤드가 다소 큰 대신 생성되는 코드의 질이 좋아질 수 있다. 반면에 SFX는 미리 컴파일된 코드를 사용하기 때문에 오버헤드가 적지만 생성되는 코드의 질은 V8보다 떨어질 수 있다.

Access-fannkuch, Access-nsieve, String-base64, String-fasta와 Math-partial-sums와 같은 벤치마크의 경우 함수 개수가 적고, 함수 내의 루프가 많은 반복을 수행하는 경우이기 때문에 생성되는 코드의 질이 성능에 절대적인 영향을 미치게 될 것이라고 예측할 수 있다. 다섯 개의 벤치마크 모두 다섯 번 이내로 호출되는 함수들로만 구성되어 있고, 수행시간의 대부분은 함수 내의 루프에서 소요된다. 이 벤치마크들에서는 V8이 SFX보다 높은 성능을 보인다.

그러나 Access-nbody, Crypto-md5, Crypto-sha1, 같은 벤치마크들은 함수의 개수가 많고 SFX에서 보다 높은 성능을 보인다. 앞서 언급한 것처럼 함수의 개수가

적을 때는 컴파일 오버헤드의 영향이 적지만 함수의 개수가 많아질 수록 컴파일 오버헤드의 영향이 커지게 된다. 따라서 벤치마크의 함수 개수가 V8과 SFX의 성능 차이에 영향을 미칠 것이라고 판단할 수 있다.

표 1은 SunSpider의 일부 벤치마크들의 함수 개수와 가장 높은 성능을 보이는 엔진을 표시한 결과이다. 몇 가지 예외를 제외하면 함수 개수가 많은 벤치마크에서는 SFX가, 함수 개수가 적은 벤치마크에서는 V8이 보다 높은 성능을 보이는 경향성을 발견할 수 있다.

위 표에서 나타나는 경향성에 어긋나는 몇 가지 벤치마크들을 찾아볼 수 있는데, Bitops-nsieve-bit나 Bitops-bits-in-byte, 3D-morph와 같은 벤치마크는 함수의 개수가 2-4개 정도로 적음에도 불구하고 SFX에서의 성능이 가장 빠르게 나타났다.위 벤치마크들의 공통점은 주로 곱하기나 나누기, 시프트 연산과 같은 산술 연산이 수행시간의 대부분을 차지한다는 점이다. Math-cordic, Math-spectral-norm, 3D-cube, 3D-raytrace 벤치마크에서도 산술 연산이 많은 부분 사용되었고, 이들 벤치마크 모두 SFX가 높은 성능을 보이거나 V8과 거의 비슷한 성능을 보였다.

산술 연산에 대해 SFX가 좋은 성능을 보이는 원인은 2.3절에서 언급한 SFX의 컴파일 방식에서 찾을 수 있다. SFX의 경우 기계어 코드를 직접 만들어내지 않고, 인터프리터용 바이트코드 수행 코드를 컨텍스트 쓰레드 테이블에 복사하여 기계어 코드를 만들어 내는 방식을 사용하는데, 이런 방식이 대부분의 경우에는 직접 코드를 만들어 내는 것보다 코드의 질이 떨어질 가능성이 높지만 산술 연산들에 대한 수행 코드는 엔진 코드를 컴파일 하는 과정에서 최적화 작업을 충분히 수행할 가능성이 높기 때문에 코드의 질이 더 좋을 수도 있다.

표 1 일부 벤치마크의 함수 개수에 따른 성능 결과

벤치마크	함수 개수	가장 높은 성능을 나타내는 엔진
Crypto-md5	21	SFX
Crypto-sha1	18	SFX
Access-nbody	12	SFX
Date-format-xparb	6	V8, SFX
String-fasta	5	V8
Access-binary-trees	4	V8
Access-nsieve	4	V8
Controlflow-recursive	4	V8
Bitops-nsieve-bit	4	SFX
Bitops-3bits-in-byte	3	V8
Bitops-bits-in-byte	3	V8, SFX
Access-fannkuch	2	V8
Math-partial-sums	2	V8
3D-morph	2	SFX
Bitops-bitwise-and	1	V8

4. 결론 및 향후 연구 방향

본 논문에서는 고성능 자바스크립트 엔진들의 수행 방식의 차이로 인해 SunSpider 벤치마크에서의 어떠한 성능 결과를 가져오는지를 분석해 보았다. 이러한 분석은 차후 자바스크립트 엔진의 성능 향상에 관한 연구에 많은 도움이 될 수 있을 것이다. 다만 SunSpider 벤치마크가 실제 웹 페이지의 자바스크립트와는 차이가 있을 것이므로 이에 대한 보완 평가가 필요할 것이다. 그러나 RIA처럼 많은 연산을 자바스크립트가 수행하는 환경에서는 SunSpider가 비교적 대표성을 가질 수 있으므로 유용한 평가 결과라고 판단된다.

또한 본 논문에서의 실험은 데스크탑 환경에서 수행되었는데, 최근에는 데스크탑 환경 뿐 아니라 휴대폰이나 IPTV와 같은 내장형 환경에서의 풀 브라우징(Full browsing)이 이슈로 떠오르고 있다. 데스크탑에 비해 제한적인 하드웨어 자원 때문에 내장형 환경에서의 자바스크립트 엔진의 성능은 보다 중요하다. 따라서 본 논문에서의 연구를 내장형 환경으로 확장하여 수행하는 것도 좋은 연구 과제가 될 수 있을 것이다.

참 고 문 헌

[1] Rich Internet Applications, "http://www.adobe.com/platform/whitepapers/idc_ impact_of_rias.pdf"

[2] Mozilla Wiki - TraceMonkey, "http://wiki. mozilla.org/Javascript:TraceMonkey"

[3] V8 Javascript Engine, "http://code.google.com/apis/v8"

[4] The Webkit Open Source Project "http://webkit.org"

[5] Dromaeo : Javascript Performance Testing, "http://dromaeo.com"

[6] SunSpider JavaScript Benchmark, "http://www2.webkit.org/perf/sunspider-0.9/ sunspider.html"

[7] A. Gal, C. W. Probst, M. Franz "HotpathVM: An Effective JIT Compiler for Resource-constrained Devices," *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pp.144-153, 2006.

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, M. Frantz "Trace-based Just-In-Time Type Specialization for Dynamic Languages," *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.465-478, 2009.

[9] M. Berndl, B. Vitale, M. Zaleski, A. D. Brown "Context Threading: A flexible and efficiend dispatch technique for virtual machine interpreters," *Proceedings of the International Symposium on Code Generation and Optimization*, pp.15-26, 2005.