자바스크립트 적시 컴파일러를 위한 생성 코드 재사용

오형석, 문수묵
서울대학교 전기컴퓨터공학부
oracle@altair.snu.ac.kr, smoon@altair.snu.ac.kr

Reuse of the Generated Code for JavaScript Just-in-Time Compiler

Hyeong-Seok Oh, Soo-Mook Moon
School of Electronical Engineering and Computer Science
Seoul National University

요 약

모바일 환경에서 웹 브라우저 활용이 늘어나고 웹 페이지를 통한 다양한 서비스가 제공되면서 브라우저에서의 자바스크립트 성능이 중요한 요소가 되었다. 자바스크립트 엔진의 성능 개선을 위해 기존에 사용하던 인터프리터 대신에 머신 코드를 실행 시간에 생성하는 적시 컴파일러(JITC)가 도입되었다. 특히 모바일 환경에서 WebKit의 자바스크립트 엔진인 SquirrelFish Extreme(SFX)이 많이 사용되고 있다. 본 논문에서는 SFX의 성능 향상을 위하여 적시 컴파일러를 통해 생성된 머신 코드를 파일 시스템을 활용하여 파일에 저장하였다가 재사용하는 클라이언트 AOTC(ahead-of-time compilation) 방식을 제시하고 초기 실험 결과를 제시한다.

1. 서 론

최근 모바일 환경이 제공하는 다양한 서비스와 새로운 서비스를 인터넷을 통해 제공하는 것이 늘어나면서, 스마트폰과 같은 모바일 기기에서 데스크탑과 동일한 인터넷 실행환경을 제공하는 풀브라우징(Full-browsing)이 주요 서비스로 각광받고 있다. 또한 다양한 어플리케이션 동작을 위한 모바일 플랫폼으로 웹 브라우저 엔진을 통해 HTML과 XML 및 자바스크립트(JavaScript)를 수행하기도 한다.

한편 데스크탑 환경에서도 AJAX(Asynchronous JavaScript and XML)[1] 기술을 이용하여 서버와 사용자간의 빈번한 데이터 송수신을 통해 사용자의 동작에 반응하는 동적 웹페이지가 증가하고 있다. 여기에 데스크탑 어플리케이션 수준의 서비스를 웹페이를 통해서 제공하는 RIA(Rich Internet Application)[2]를 제공하면서 자바스크립트 활용이 더욱 커지고 있다.

이처럼 다양한 환경에서의 자바스크립트 활용방면이 넓어지면서 웹 브라우저의 자바스크립트 처리 능력이 중요한 사항으로 떠오르게 되었다. 자바스크립트 처리 능력을 향상시키기 위한 방법 중 대표적인 것이 적시 컴파일러(Just-in-Time Compiler, JITC)[3]를 추가하는 것인데, 이는 기존의 인터프리터를 통해 자바스크립트 코드를 실행하는 것에서 벗어나 실행 중에 자바스크립트 코드로부터 기계어 코드를 생성하여 이를 수행하는 것이다. 적시 컴파일러를 도입한 대표적인 공개 자바스크립트 엔진으로 Mozilla Firefox의 자바스크립트 엔진인 SpiderMonkey의 적시 컴파일러인 TraceMonkey[4], Google Chrome의 자바스크립트 엔진인 V8[5], 그리고 WebKit 브라우저 엔진의 자바스크립트 엔진인 JavaScriptCore에 적시 컴파일러 기술을 적용한 SquirrelFish Extreme(SFX)[6]가 있다.

특히 최근 모바일 환경에서 웹브라우저 엔진으로 WebKit을 활용하는 경우가 늘어나면서 자연스럽게 자바스크립트 엔진으로 SFX가 많이 채택되고 있다. 스마트폰에서 WebKit을 주로 사용하는데, 대표적으로 IPhone, Palm pre, S60등이 있다. 이러한 추세는 공개소스(Open-source) 소프트웨어인 WebKit이 모바일 환경에서 만족할만한 성능을 보이고, 플랫폼상에서 어플리케이션 동작 엔진으로 활용하는 데에 유용하기 때문이다.

하지만 여전히 모바일 환경의 자원 제약으로 인해 자바스크립트 엔진 성능의 개선이 필요하다. 또한 메모리 등 한정된 자원을 효과적으로 사용하기 위한 방안이 필요하다.

본 논문에서는 자바스크립트의 적시 컴파일러 성능

본 연구는 서울형산업 기술개발 지원사업(NT080546)의 지원으로 수행되었음.

향상을 위해 클라이언트 AOTC[7]를 활용하여 생성된 코드를 재사용을 적용하는 방법을 제시하고 그 결과를 보인다. 2절에서는 자바스크립트의 특징과 SFX의 컴파일 방식을 소개하고 3절에서는 SFX에 코드 재사용 방식을 적용하는 방법을 제시한다. 그리고 4절에서는 벤치마크에서 이를 적용한 결과를 확인하고 5절에서는 결론과 향후 과제를 제시한다.

2. 자바스크립트의 특징과 SFX의 적시 컴파일러

본 절에서는 자바스크립트의 언어적 특성과 SFX의 적시 컴파일 방식을 소개한다.

2.1. 자바스크립트

자바스크립트는 객체 지향(Object-Oriented) 방식의 스크립트 언어로, 주로 웹 페이지에서 동적 페이지를 제공하기 위한 보조 스크립트로 사용되어 왔다. 자바스크립트를 통해 브라우저와 웹 페이지의 구성을 구조화한 Document Object Model(DOM)에 접근하여 사용자의 입력이나 실행 환경의 변화 등에 따라 DOM을 변경할 수 있기 때문이다. 최근에는 Web 2.0을 구현하는 핵심 기술이라 할 수 있는 Ajax에서 웹 페이지 갱신 없이 서버와 사용자간에 데이터를 송수신 할 수 있도록 하는 XMLHttpRequest 프로토콜을 제공하고, 복잡한 연산을 수행하는 RIA 서비스가 증가하면서 자바스크립트의 활용도는 더욱 높아졌다.

자바스크립트 언어는 기본적으로 C와 유사한 방식으로 작성된다. 그러나 자바스크립트는 동적 언어로서 다양한 동적 특성을 제공하여 언어적으로 차이가 발생하며 이 특성이 컴파일과 실행환경을 복잡하게 하는 큰 원인이 된다.

우선 자바스크립트는 동적 타입을 제공한다. 변수나 오브젝트의 property의 타입이 코드 작성시에 정해지지 않으며 실행중에 수시로 변경될 수 있다. 또한 클래스 기반 객체 지향 언어가 아닌 프토토타입 기반 객체 지향 언어이기 때문에, 오브젝트의 property가 실행중에 추가되는 등 구조가 실행중에 변경될 수 있다. 이러한 동적 특성은 함수(function)에 대해서도 적용되는데, 함수가 실행중에 생성될 수 있고 함수 속에 함수를 정의하여 동적으로 생성해 사용할 수 있는 closure를 제공한다. 아래의 그림 1은 closure의 예이다.

```
function derivative(f, dx) {
    return function(x) { return (f(x+ dx) – f(x)/dx; )
}

enter_with_activation r0 // closure를 포함한 함수 시작
new_func_exp             // closure 함수 생성
tear_off_activation r0   // closure 함수가 상위 함수의
                            변수 접근가능하게 함
ret                  r1  // 함수 return
```

그림 1. Closure 함수와 SFX 바이트코드

예제의 derivative 함수 내에 정의된 새로운 함수는 변수 f와 dx가 자신의 local 변수나 global 변수가 아닌 derivative의 local 변수임에도 참고할 수 있으며, 이런 변수들은 derivative가 수행되지 않더라도 여전히 그 값을 보존해야 한다.

이러한 특성들로 인해 머신 코드를 생성하는 컴파일러를 활용할 때 데이터의 구조와 유효한 기간을 특정할 수 없는 문제가 발생하여, 실행 중에 이러한 것들이 변경될 수 있다는 것을 항상 고려할 수 있도록 코드를 생성하고 실행환경을 구축해야 한다.

2.2. SquirrelFish Extreme과 기계어 생성방식

WebKit은 JavaScriptCore를 자바스크립트 엔진으로 사용하는데 여기에 바이트코드 방식의 인터프리터를 적용한 것이 SquirrelFish이다. SquirrelFish Extreme은 SquirrellFish에 인터프리터 대신에 적시 컴파일러를 적용한 것이다.

SFX는 자바스크립트 코드를 바이트코드로 변환한 뒤에 기계어 코드를 생성한다. SFX의 바이트코드는 Java의 스택 방식 바이트코드와 달리 register 방식의 바이트코드를 사용한다. 따라서 push/pop이 존재하지 않고 가상의 많은 수의 register를 가지고 있다고 가정하여 바이트코드가 생성된다. 이 register들은 SFX내에 자료구조로 메모리상에 register file 형태로 위치하게 된다.

한편 closure를 처리하기 위해 scope chain 자료구조를 둔다. Scope chain은 각 함수 오브젝트를 상호 포함관계에 따라 연결하고 closure 함수를 포함하고 있는 함수의 변수를 closure 함수가 접근할 수 있도록 하는 자료구조를 가진다. 그림 1은 closure를 처리하기 위한 바이트코드를 보여준다.

초기의 SFX는 바이트코드를 처리하기 위한 내부 함수를 호출하고 함수의 인자를 설정하는 기계어 코드를 생성하는 방식을 사용하였다. 이러한 방식을 Context Threading 이라 부르며, 바이트코드를 처리하기 위해 호출하는 함수를 CTI 함수이라 부른다.

Context Threading 방식은 기존의 인터프리터 방식에 비해 branch prediction에서 유리하기 때문에 성능 향상을 얻을 수 있었다.[8]

최근에 공개된 SFX는 몇몇 바이트코드에 대하여 CTI 함수를 호출하는 대신 CTI 함수와 동일한 역할을 하는 기계어 코드를 직접 생성하는 방식을 사용한다. 또한 일부는 바이트코드의 operand 타입에 따라서 특정 타입을 처리하는 기계어 코드와 타입을 만족하지 못할 때 CTI 함수를 호출하는 방식을 사용한다. 이러한 방식을 사용하면 CTI 함수를 호출하기 위해 필요한 부가적인 코드를 실행하는 오버헤드를 줄일 수 있고 생성된 코드 영역에 머무르는 시간이 늘어나면서 캐시 활용 성능을 향상시킬 수 있다.

3. 클라이언트 AOTC를 통한 코드 재활용

본 절에서는 클라이언트 AOTC 개념을 설명하고 SFX에 이를 실현하기 위한 방법을 제시한다

3.1. SFX의 클라이언트 AOTC 구조

클라이언트 AOTC는 코드 저장공간으로 파일시스템을 활용하여 실행환경이 종료되더라도 생성된 코드가 유지될 수 있도록 한다. 다음 번에 같은 웹 페이지를 방문했을 때 같은 자바스크립트 코드를 수행 시에는 적시 컴파일러를 활용하여 코드를 생성하지 않고 파일에 있는 코드를 바로 사용하는 개념이다[7]. 이를 통해 컴파일에 대한 오버헤드가 대부분 사라지게 된다.

SFX는 함수 단위로 컴파일을 수행한다. 코드 생성시에 아직 생성되지 않은 함수에 대한 호출이 있는 부분에 대해서는 컴파일을 할 수 있도록 코드 캐시를 빠져 나와 컴파일 모듈을 호출하는 코드를 생성한다. 그리고 실제 해당 코드가 수행되면 컴파일을 시작한다. 우선 코드 생성기를 통해 임시 버퍼에 각 바이트코드에 대한 기계어 코드를 생성한다. 그리고 생성된 코드 내에서의 jump등에 대해 실제 상대 주소를 패치(patch)하는 작업을 수행하고 임시 버퍼에서 실제 코드 캐시에 공간을 할당하고 생성된 코드를 복사한다. 이후 생성된 코드에 대한 추가 자료구조를 생성하고 생성된 코드 외부에 대한 jump 등에 대해 추가로 패치작업을 수행하여 실행 코드를 완성한다.

클라이언트 AOTC가 동작하도록 하기 위해서는 코드를 파일로부터 읽어 들이거나 생성된 파일에 코드를 저장하는 과정을 추가해야 한다.

우선 코드를 저장하는 시점은 상대주소를 패치한 직후 임시 버퍼가 완성되었을 때로 하여 임시 버퍼 결과를 저장하도록 하였다. 파일은 자바스크립트 코드가 있는 파일명에 새로운 확장자를 추가한 파일에 생성된 코드와 코드에 대한 추가 정보를 각각 저장하여 향후 코드 탐색이 용이하도록 하였다.

코드를 불러들이는 시점은 함수 호출 시에 생성된 코드가 코드 캐시에 없는 것이 확인된 후 적시 컴파일러가 생성되기 직전으로 하였다. 이때 코드에 대한 추가 정보가 들어있는 파일을 검색하여 생성된 코드가 파일에 저장되어있는지 확인하고, 파일에 있는 것이 확인되면 코드가 저장된 파일과 추가정보가 저장된 파일로부터 필요한 정보를 읽어 들이고 코드 캐시에 코드를 생성한다.

3.2. 저장 정보

SFX의 코드 캐시 관리 정책상 코드를 저장했다가 그대로 바로 사용할 수는 없다. SFX는 코드를 페이지 단위로 관리하고, 페이지를 넘는 코드에 대해서는 중간에 jump를 생성하도록 하였다. 또한 하나의 페이지에 여러 개의 코드가 존재할 수 있기 때문에 큰 크기의 머신 코드를 가진 함수의 경우 다시 불러들였을 때 페이지의 경계가 변경되어 중간의 jump 위치가 바뀌어야 할 필요가 생길 수 있다. 한편으로 SFX 내부 함수를 호출하거나 자료구조에 접근하는 코드의 경우 실행할 때마다 그 위치가 달라질 수 있다. 따라서 코드를 유효화 하기 위한 추가 정보가 필요하다.

클라이언트 AOTC를 위한 저장 정보에는 크기가 고정된 정보와 크기가 함수에 따라 달라지는 정보가 있다. 크기가 고정된 정보는 함수를 특정하기 위한 정보 및 머신 코드의 크기 정보 등이 있고, 크기가 변하는 정보는 실제 머신 코드 등이 있다. 이 두가지를 별도의 파일에 저장하여 코드가 파일에 저장되어 있는지를 확인할 때 크기가 고정된 정보가 모여있는 파일을 탐색하도록 하였다. 파일에 저장된 정보는 다음과 같다.

3.2.1. 함수 위치

SFX는 자바스크립트 함수의 실제 이름을 보존하지 않고, 파일 내에서 함수의 위치를 통해 함수를 구분할 수 있다. 이 정보를 탐색하여 머신 코드가 파일에 저장되어 있는지 확인하도록 한다. 또한 머신 코드를 모아둔 파일 내에서 해당 코드의 위치 또한 저장한다.

3.2.2. 머신 코드

적시 컴파일러를 통해 임시 버퍼에 생성된 머신 코드와 머신 코드의 크기를 저장한다.

3.2.3. Constant Pool

ARM과 같은 특정 타겟에 대하여 SFX는 constant pool을 사용한다. Constant pool은 머신 코드가 특정 위치의 값을 읽어 들이거나 특정 위치로 jump할 때의 타겟을 코드 캐시에 저장하여 PC로부터의 상대 위치값을 통해 읽을 수 있도록 한다. 이는 PC로부터 상대 위치를 통해 읽어 들이는 것이 명령어 개수를 줄일 수 있기 때문이다.

따라서 constant pool의 크기와 값, 그리고 constant pool을 참조하는 머신 코드의 위치와 개수를 저장해야 한다.

3.2.4. 코드 패치 및 추가 정보

임시 버퍼에서 코드 캐시로 코드를 복사한 뒤에도 코드 외부에 대한 참조에 대한 코드 패치나 코드에 대한 자료 구조 생성을 위한 추가 정보가 필요하다. 해당 정보들에 대한 크기와 복원을 위한 추가 정보들을 저장해야 한다.

3.2.5. 재배치 정보

위에서 설명한 대로 실행 시에 따라 SFX내의 함수 및 자료의 위치가 달라질 수 있기 때문에 이러한 접근을 하는 코드에 대한 재배치가 필요하다. 그 대표적인 것으로는 CTI 함수가 있다. 이러한 접근은 머신 코드뿐만 아니라 constant pool에 저장된 값도 포함되므로 코드와 constant pool에 대한 재배치 정보를 저장해야 한다.

4. 실험

4.1. 실험 환경

본 논문의 실험은 ARMv7 기반 Cortex-A8[9] CPU를 사용하는 beagle board상에서 이루어졌다. 실험을 위해 WebKit에서 자바스크립트 엔진을 별도로 컴파일하여 stand-alone SFX를 구성하였다.

실험을 위한 벤치마크로 일부의 SunSpider benchmark를 사용하였다.[10]

4.2. 실험 결과

그림 2는 클라이언트 AOTC를 적용하지 않은 SFX(Original)와 클라이언트 AOTC를 적용할 경우 파일을 작성할 때(Dump)와 파일로부터 코드를 읽어 들일 때(Load)의 실행시간을 두 벤치마크에 대해 비교한 것이다.

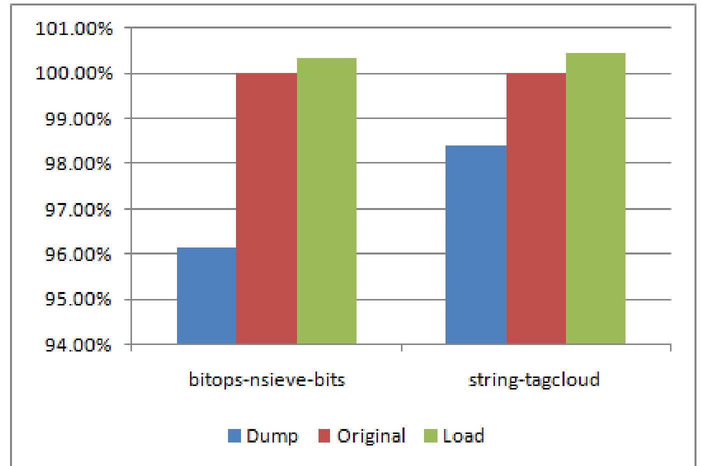

그림 2. 실행 성능 비교

실험결과 클라이언트 AOTC를 적용했을 때에 비해 파일을 작성할 때에는 파일 출력 오버헤드로 인해 약간 느려지고 반대로 코드를 재 상용했을 때는 약간의 성능 향상이 있었다. SunSpider의 특성상 컴파일 오버헤드가 실행시간에서 차지하는 비중이 별로 크기 않기 때문으로 보인다. 그러나 일반적인 웹 페이지 상에서의 자바스크립트 코드는 벤치마크에 비해서 반복구문이 적고 서로 다른 함수 호출이 매우 많기 때문에[11] 컴파일 오버헤드가 매우 클 것이므로 (예를 들어 nationalgeographic.com의 홈 페이지를 로딩하는 데 있어서 679개의 자바스크립트 함수가 수행되며, 이렇게 수백개의 함수가 수행되는 경우가 종종 있음), 클라이언트 AOTC의 효과가 상대적으로 클 것으로 예상된다.

5. 결론 및 향후 과제

웹 페이지 수행을 위한 자바스크립트 성능이 중요해 지면서 자바스크립트 엔진에 적시 컴파일러가 적용되었다. 이로 인해 성능이 향상되었으나 아직 성능 향상을 위한 개선이 필요하다. 본 논문은 자바스크립트 엔진 성능 향상을 위해 파일 시스템을 활용하여 코드를 재활용할 수 있는 기회를 확장하였고 초기 실험 결과를 제시하여 그 가능성을 확인하였다.

Reference

[1] AJAX, https://developer.mozilla.org/En/AJAX
[2] “Rich Internet Applications”, http://www.adobe.com/platform/whitepapers/idc_impact_of_rias.pdf
[3] J. Aycock. "A brief history of Just-in-Time", ACM Computing Surveys, 35(2), Jun 2003.
[4] TraceMonkey,

https://wiki.mozilla.org/JavaScript:TraceMonkey
[5] V8, http://code.google.com/p/v8
[6] SquirrelFishExtreme, http://webkit.org/blog/214/introducing-squirrelfish-extreme
[7] SungHyun Hong, et. al, Java client ahead-of-time compiler for embedded systems, LCTES, 2007
[8] M. Berndl et. al, Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters, Proceedings of the international symposium on Code generation and optimization, p.15-26, March 20-23, 2005
[9] Cortex-A8 Technical Reference Manual, revision: r3p1
[10] SunSpider JavaScript Benchmark, http://www2.webkit.org/perf/sunspider-0.9/sunspider.html
[11] 정원기, et. al, 실제 웹 기반 벤치마크에 의한 자바스크립트 엔진의 성능 평가, 한국정보과학회 한국컴퓨터종합학술대회, 2009