■ 2013년도 학생논문 경진대회 수상작

크로스-브라우저 프로파일링을 통한 웹 어플리케이션 성능버그 탐지

(Performance Bug Detection in Web Applications through Cross-browser Profiling)

박 용 배 † 홍 신 † 김 문 주 ††
(Yongbae Park) (Shin Hong) (Moonzoo Kim)

요 약 웹 어플리케이션의 성능을 보장하기 위해서 성능버그를 찾아야 한다. 하지만 기존 테스팅 방법으로 성능버그를 찾고 분석하는 방법은 통제 불가능한 요인이 많은 웹 어플리케이션에 적용하기 힘들다. 이러한 문제점을 극복하기 위해서, 본 논문은 여러 웹 브라우저에서 자바스크립트 프로그램의 성능을 프로파일링하여 성능버그를 찾는 *크로스-브라우저 프로파일링 기법*을 제안한다. 이 기법을 통해서 서로 다른 웹 브라우저에서 실행시간 차이가 큰 자바스크립트 함수를 찾는다면, 코드를 더 최적화하여 성능을 향상시킬 수 있을 것이라고 추론하였다. 이 기법을 The Organizer와 WordPress에 적용하는 사례연구를 통해서 5개의 새로운 성능버그를 찾고 성능이 향상시켰다.

키워드: 성능버그, 프로파일링, 웹 어플리케이션, 자바스크립트 프로그램

Abstract Detecting performance bugs is desirable for web applications. However, identifying and diagnosing web application performance issues by traditional methods is difficult, because performance quantification depends on many factors uncontrollable by testers. To overcome these difficulties, we propose a new technique *cross-browser profiling* by which a performance bug is detected by comparing performance profiling results across various web-browsers. We conjecture that a JavaScript function whose execution time varies largely depending on web-browser would have features able to optimize in code level. We present the case study of the proposed technique with two web application, the Organizer and Wordpress, to demonstrate that the technique effectively detects performance bugs. In case study, total 5 performance bugs are newly detected and the program performances are improved for fixing these bugs.

Keywords: performance bug, profiling, web application, JavaScript program

· 본 연구는 미래부가 지원한 2013년 정보통신·방송(ICT) 연구개발사업, 지식경제부/한국산업기술평가관리원 IT R&D 프로그램(10041752, 초소형·고신뢰 OS와 고성능 멀티코어 OS를 동시 실행하는 듀얼 운영체제 원천 기술 개발), 미래부/한국연구재단의 중견연구자지원사업-핵심연구(NRF-2012R1A2A2A01046172), 그리고 정보통신산업진흥원의 IT/SW 창의연구과정(NIPA-2012-H0503-12-1006)의 연구비 지원으로 수행되었음

† 학생회원 : 한국과학기술원 전산학과
yongbae2@gmail.com
hongshin@gmail.com
†† 종신회원 : 한국과학기술원 전산학과 교수
moonzoo@cs.kaist.ac.kr
(Corresponding author임)

논문접수 : 2013년 6월 3일
심사완료 : 2013년 9월 11일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제19권 제11호(2013.11)

1. 서 론

소프트웨어 성능(performance)은 소프트웨어가 올바른 기능을 수행할 때 자원(예: 시간, 메모리)을 얼마만큼의 효율적으로 사용하는 지 평가하는 품질요소다. 소프트웨어 성능은 소프트웨어 동작정확성과 함께 소프트웨어의 품질을 결정하는 중요한 품질요소 중 하나이다. 최근 활발히 개발되고 있는 웹 어플리케이션은 높은 사용자경험 만족도가 요구되는 시스템으로, 소프트웨어와 사용자가 얼마만큼 끊임없이 상호작용할 수 있는 지가 유사한 웹 어플리케이션 간에서 중요한 경쟁요소가 된다. 최근 조사에 따르면 웹 어플리케이션의 성능은 서비스 회사 이익의 최대 9%까지 영향을 줄 수 있다고 알려져 있다[1]. 웹 어플리케이션의 성능을 잘 관리하는 회사는 그렇지 않은 회사보다 평균 9.6만 달러를 더 투자하고 있으며, 성능 향상으로 1.17억 달러를 더 벌어들인다. 이러한 현상은 웹 어플리케이션이 느리게 동작하면 이를 사용하는 고객의 만족도가 낮아지는 것은 물론, 웹 어플리케이션을 사용하는 직원의 생산성 또한 감소하기 때문에 나타난다.

성능버그(performance bug)는 소프트웨어가 올바른 기능을 수행할 때 필요 이상의 자원 소모를 발생시키는 결함이다[2]. 이러한 성능버그를 찾는 방법 중 하나로 성능 테스팅이 제안되어 왔지만, 성능 테스팅은 성능버그 검출과 디버깅에 효율이 낮다는 문제점이 있다[3]. 최근 연구에 따르면 성능버그에 대한 디버깅은 다른 종류의 버그에 대한 디버깅보다 평균 1.32배 더 많은 시간이 걸린다고 보고되었다[4]. 특히 웹 어플리케이션의 경우, 소프트웨어가 다양한 하드웨어/소프트웨어 환경에서 실행되기 때문에 프로그램 내에 존재할 수 있는 다양한 종류의 성능버그를 개발자가 수작업으로 탐색하는 데에 더 많은 비용이 소요될 것으로 보인다. 따라서, 웹 어플리케이션에 대해 자동으로 성능버그를 탐지하는 기법이 절실히 요구되는 실정이다.

본 논문에서는 테스팅을 통해 생성한 실행정보를 분석하여 웹 어플리케이션의 성능버그를 자동으로 검출하는 *크로스-브라우저 프로파일링 기법*을 소개한다. 크로스-브라우저 프로파일링 기법은 다양한 웹 브라우저를 이용하여 웹 어플리케이션의 특정 기능에 대한 다양한 종류의 실행정보를 생성하고, 그 결과를 비교 분석을 통해 웹 어플리케이션에서 성능버그 후보로 탐지하는 기법으로, 사용자의 성능버그 탐지와 디버깅 지원을 목표로 한다. 제안한 기법은 세 가지 웹 브라우저(Internet Explorer, Firefox와 Chrome)를 사용하여 크로스-브라우저 프로파일링을 구현하였다. 제안한 기법을 실제 웹 어플리케이션인 The Organizer[5]와 WordPress[6]를 대상으로 적용하는 사례 연구를 통해 효과적인 성능버그를 탐지가 가능함을 소개한다. 사례연구 결과, The Organizer에서 18개의 성능버그 후보 중에서 실제 성능버그 2개를 확증하였고, WordPress에서 30개의 성능버그 후보 중에서 3개의 실제 성능버그를 확증하였다.

본 논문의 구성은 다음과 같다. 2장에서는 성능버그와 관련된 연구를 소개한다. 3장에서는 성능 프로파일링, 성능버그 후보를 정의한다. 4장에서는 성능 프로파일링을 이용한 성능버그 후보 검출 방법을 소개하며 5장에서는 본 논문에서 제시한 성능 프로파일링과 성능버그 후보 검출 방법을 The Organizer와 WordPress에 적용하여 성능버그 후보를 찾고 그중에서 성능향상이 가능한 성능버그에 대해서 설명한다. 6장에서는 성능버그 오경보 경우와 효율적인 디버깅을 위한 성능버그 정렬 방안을 논의한다. 마지막으로 7장에서는 결론을 제시한다.

2. 관련 연구

웹 어플리케이션이나 그래픽 사용자 인터페이스(Graphic User Interface; GUI)와 같은 대화식 프로그램(interactive program)에서 성능버그를 검출하기 위하여 테스트 수행 결과를 분석하는 기법들이 최근 개발되고 있다. 일반적으로, 성능버그는 기능을 변경하지 않는 간단한 코드 수정으로 프로그램의 성능을 효과적으로 향상시킬 수 있는 결함으로 정의할 수 있다[2]. 전통적인 성능버그 탐지기법에서는 프로그램의 전체 동작에 대한 총 실행시간을 측정한 후, 전반적인 동작의 총 실행시간에서 가장 많은 부분을 차지하는 요소(예: 함수)인 핫스포트(hot spot)를 성능버그 후보로 찾는 방법을 사용해왔다. 하지만 이와 같은 기법은 특정 동작의 반응지연으로 특징되는 종류의 성능버그의 효과적인 탐지가 어렵다[7]. 최근, 반응성 프로그램에서 효과적인 성능버그 검출을 위해 총 실행시간 대신 소요시간(latency)을 측정하는 기법과 대기시간(wait time)을 측정하여 분석하는 기법이 개발되고 있다.

소요시간(latency)은 프로그램의 특정기능이나 특정요소의 실행의 시작과 종료까지 걸린 시간을 뜻한다. 기능별(혹은 프로그램 요소별) 소요시간을 측정하여, 프로그램의 전반적인 실행시간이나 실행의 전체 소요 시간을 측정에서는 발견하기 어려운 반응성 이상을 검출하고 분석하도록 지원하는 기법이 제안되고 있다. LagHunter[7]는 GUI 프로그램의 성능버그를 검출하기 위하여 테스트 실행에서 GUI 프로그램에서 함수별 소요시간을 측정한 후, 이를 활용하여 소요시간이 기준점 이상인 함수와 관련된 호출스택(callstack)을 분석하여 긴 소요시간을 발생시키는 요소를 성능버그로 검출한다. AppInsight[8]는 모바일 어플리케이션의 성능을 측정하여 병목현상을

찾아내는 도구이다. 자바스크립트 실행 모니터링 프레임워크인 AjaxScope[9]는 다양한 실행환경에서 발생할 수 있는 자바스크립트 성능버그 검출을 위하여 함수별로 소요시간을 모니터링 하는 기능을 제공한다.

특정기능의 소요시간을 측정하고 이를 평균과 표준편차를 이용하여 기능이 느리게 실행되었는지와 정상적으로 실행되었는지를 구별하여 성능이 비정상적으로 느려지는 현상을 찾는 연구가 있다[10]. 이 연구에서 웹 서비스 사용자의 입력마다 소요시간을 측정하고 평균보다 심각하게 느려지는 현상이 생기면 관리자에게 보고하여 대처할 수 있도록 하였다. 그리고 후속 연구로 성능이 비정상적으로 느려지는 현상의 원인을 찾는 연구가 있다[11]. 소요시간과 더불어 웹 서비스를 제공하는 각 서버의 부하를 측정하여 어느 서버에서 병목현상이 생기는지 찾고 웹 서비스 개발자가 성능버그를 더 쉽게 찾을 수 있도록 도와준다.

대기시간(wait time)은 프로그램의 특정기능이나 특정요소의 실행의 시작과 종료 사이에 기능이 수행되지 못하고 지연되는 시간을 뜻한다. 대기시간은 주로 멀티쓰레드 프로그램과 같이 동시에 수행되는 작업이 공유자원을 획득하기까지 기다리는 과정에서 발생한다. 공유자원 점유 방식에 따라 프로그램이 같은 기능을 수행하면서도 서로 다른 대기시간이 발생할 수 있으므로, 불필요한 대기시간의 발생을 성능버그로 검출하는 기법이 개발이 제안되고 있다. StackMine[12]은 GUI 프로그램에서 성능버그를 검출하기 위하여 함수별 대기시간 발생을 측정한다. 그 이후 측정된 실행 정보를 분석할 때, 특정한 시간 이상의 대기시간이 발생한 함수를 성능결함 검출대상으로 분석한다. Doloto[13]는 웹 어플리케이션이 네트워크에서 코드를 다운로드 받아 실행하는 특성으로 인하여 발생할 수 있는 불필요한 대기시간 발생을 성능버그로 파악하고, 이를 줄이기 위한 효율적인 코드 분리 방법을 제안한다.

본 연구에서 제안하고자 하는 기법은, 소개한 기존 연구와 같이 자바스크립트의 소요시간(latency)을 측정한다는 면에서 공통점을 가진다. 하지만 본 연구에서 제안하는 기법은 소요시간을 설정된 기준치와 비교하여 성능 이상을 판별하지 않는다. 본 연구에서 제안하는 기법은 여러 개의 웹 브라우저를 통해 동일한 기능을 수행하는 여러 개의 실행을 생성하고 이를 바탕으로 성능버그를 정의한 점에서 기존 연구와 구별된다. 그리고 소요시간을 사용하여 성능버그를 파악하는 것은 각 브라우저에서 사용자 입력의 속도를 같게 할 수 없기 때문에 입력 속도에 영향을 받을 수 있는 대기시간 보다 더 정확하다. 본 연구에서 소개하는 성능버그 정의는 기존 연구에서 사용된 프로그램 분석 기법들과는 독립적이므로 기존의 다양한 기법들과 복합적으로 사용될 수 있을 것으로 보인다.

성능버그의 특성에 관한 연구로, 실제 웹브라우저 프로그램 개발에서 발생한 버그 리포트를 성능버그 리포트와 성능과 관련 없는 버그 리포트로 분류한 후 각 분류에 해당하는 100개의 버그 리포트를 수집하여 소프트웨어 개발 과정에서 버그 특징을 연구가 소개되었다[14]. 이 논문에 따르면, 성능버그와 관련된 오류 보고는 성능버그가 있음을 확인하는데 더 오랜 시간이 걸린다. 특히 개발자가 테스트한 결과에서는 이상이 없는 것으로 나타나는 경우가 성능과 관련 없는 버그 리포트보다 더 많았다. 그리고 성능버그가 다시 발생하는 빈도가 높았으며, 성능버그 사이에 연관성은 더 적었다.

웹 어플리케이션에서 성능 향상을 위해 웹 페이지나 자바스크립트에서 최적화 프로그래밍 패턴이나 성능버그 패턴의 제안이 있다[15]. DynaTrace[16]와 YSlow[17]는 웹 어플리케이션 실제 실행을 모니터링하여 이와 같은 패턴을 자동으로 탐지해준다. 그러나 이러한 패턴을 정적 분석을 통해 찾는 것은 자바스크립트의 동적인 특성 때문에 매우 어려운 것으로 보인다[18].

3. 성능버그 후보 정의

이 장에서는 웹 어플리케이션 실행 모델과 성능 프로파일링에 대해서 정의한 후, 동일한 웹 어플리케이션을 여러 브라우저에서 성능 프로파일링 한 결과를 바탕으로 성능버그 후보를 정의하고자 한다.

3.1 웹 어플리케이션 실행 모델

웹 어플리케이션 실행 모델은 테스팅을 통하여 측정한 프로그램의 성능 정보를 분석하기 위한 목적으로 테스팅 과정에서 발생하는 웹 어플리케이션 실행 결과를 추상화한 표현한다. 웹 어플리케이션은 HTML 문서, 자바스크립트 코드, CSS, 이미지, 동영상 등으로 구성되어 있으며 웹 브라우저는 사용자가 입력한 주소에서 웹 어플리케이션을 가져온 후에, HTML 문서를 순차적으로 해석(parsing)하면서 웹 페이지를 생성하고 화면에 웹 페이지를 표시한다. 그리고 웹 브라우저는 자바스크립트 코드가 있다면 이를 실행한다. 사용자는 생성된 페이지에 입력을 줌으로써 내용을 변경할 수 있는데, 이 때 입력은 순차적으로 생성된 유한한 이벤트들이다. 각 이벤트에 대한 반응으로 자바스크립트 함수가 선언된 경우, 브라우저는 해당 자바스크립트 함수를 실행함으로써 웹 페이지의 상태를 변화시킨다.

웹 어플리케이션 실행 모델에서 웹브라우저의 동작은 정의 1과 같이 표현될 수 있다. 웹브라우저는 초기 웹페이지, 웹 어플리케이션 프로그램과 사용자 입력(이벤트 순열)을 입력 받아서 화면에 결과 웹 페이지를 생성하

는 함수이다. 본 모델에서는 사용자 입력으로 주어지는 이벤트는 순열과 같은 순서로 실행되며(synchronous execution), 웹 어플리케이션 프로그램의 동작인 비결정성(non-determinism)을 가지지 않는 것으로 가정한다.

정의 1. 웹 브라우저 b는 사용자로부터 주어진 웹 어플리케이션 $application$을 실행하고 초기 웹 페이지 $page_0$에서 사용자 입력 $input$을 수행하면, 연속된 웹 페이지 $page_\sigma$를 화면에 순서대로 표시한다.

$$b(application, page_0, input) = page_\sigma$$

3.2 성능 프로파일링

웹 어플리케이션의 성능 프로파일링은 웹 어플리케이션이 실행하는 주기 동안 사용하는 총 자원을 측정한 결과이다. 본 기법에서는 자바스크립트 프로그램의 여러 가지 성능 중 실행 시간에 집중하며, 특별히 함수별 실행시간을 성능으로 측정하였다. 성능으로 측정하는 자바스크립트 함수의 실행시간은 정의 2와 같다. 정의 2에서 보는 바와 같이 함수 f의 소요시간은 함수 f가 호출된 시점으로부터 함수f가 반환되는 시점의 차이로, 함수 f의 실행과 함수 f가 호출하는 함수의 실행시간을 모두 포함하는 포괄 실행시간(inclusive time)이다. 한 함수가 여러 번 호출되었다면 각 호출의 실행시간의 평균이 함수 f의 실행시간이다. 그리고 함수가 호출되는 문맥(context)은 고려하지 않았다. 자바스크립트에서 익명함수(anonymous function)이라는 이름이 없는 함수를 선언하여 사용할 수 있다. 이 경우 편의상 함수가 선언된 파일이름과 파일 내에서 선언된 부분의 줄 번호로 함수 이름을 대신하였다.

정의 2. 브라우저 b에서 웹 어플리케이션 $application$을 실행하고 초기 웹 페이지 $page_0$에서 사용자 입력 $input$으로 성능 프로파일링을 수행하면 연속된 웹 페이지 $page_\sigma$와 호출된 자바스크립트 함수의 실행시간 $metrics$을 얻을 수 있다.

$$b(application, page_0, input) = (page_\sigma, metrics)$$

여기서 $metrics(f)$는 호출된 자바스크립트 함수 f의 평균 포괄 실행시간(inclusive execution time)이다.

3.3 성능버그 후보

본 기법에서는 성능버그 후보를 동일한 함수의 소요시간이 웹브라우저에 따라 소요시간의 차이가 기준치를 넘는 함수로 정의한다. 기본적으로 한 함수를 실행하는 시간은 웹 브라우저에 따라 차이가 있다. 이 때, 특정 함수에 대하여, 웹 브라우저에 따른 실행시간 차이가 일반적인 함수에서 발생하는 실행시간 차이보다 크다면, 해당 함수는 특정 브라우저에서 같은 기능을 수행하는데 필요이상의 시간비용을 소모하는 현상으로 볼 수 있다[2]. 크로스-브라우저 프로파일링 기법은 웹브라우저에 따른 성능 차이가 큰 프로그램 요소는 코드 수정을 통해 성능을 개선할 수 있는 여지가 크다고 판단하고, 이러한 프로그램 요소를 성능버그 후보로 탐지한다. 웹 어플리케이션의 경우, 동일한 코드에 대해서도 웹 브라우저의 최적화 방식에 따라 불필요한 세부 과정을 줄임으로써 실행 시간이 단축되는 경우가 많다. 따라서 본 기법에서는, 특정 웹 브라우저의 최적화가 효과적으로 적용되는 코드의 경우, 개발자가 같은 기능을 제공하는 보다 단순한 코드를 사용함으로써 모든 웹 브라우저에 빠르게 동작하는 코드로 변경할 가능성이 큰 것으로 예상하여 성능버그로 탐지하였다. 이와 같이, 크로스-브라우저 프로파일링을 통한 성능버그 탐지 기법은, 함수의 절대적인 소요시간의 다소만으로 성능버그를 탐지하지 않고, 프로그램 수정을 통해서 성능을 줄일 수 있는 가능성을 추론함으로써 성능버그 탐지에 있어서 기존의 프로파일링 기법을 보완하였다.

여러 웹 브라우저에서 성능 프로파일링 결과로 정의 3과 같이 성능비율(performance ratio)을 정의한다. 그리고 성능버그 기준(performance bug threshold)으로 성능비율이 일정 범위를 넘어서는 함수를 정의 4와 같이 성능버그 후보라 한다. 정의 3에서 웹 페이지가 동일함을 판단하는 기준은 각 웹페이지의 크기를 비교하여 5%이내의 차이를 보이면 동일한 웹 페이지로 하였다.

정의 3. 웹 브라우저 집합 $B=\{b_1, b_2, ..., b_n\}$이 주어졌을 때, 각 웹 브라우저에서 웹 어플리케이션 $application$을 실행하여 초기 웹 페이지 $page_0$를 시작으로 동일한 입력 $input$으로 성능 프로파일링을 하여 연속된 웹 페이지 p_i와 자바스크립트 함수 실행시간 m_i를 결과로 얻었다고 하자(i는 1부터 n까지의 정수).

$$b_1(application, page_0, input) = (p_1, m_1)$$
$$\cdots$$
$$b_n(application, page_0, input) = (p_n, m_n)$$

성능 프로파일링 결과가 모든 웹 브라우저에서 웹 페이지의 결과가 같음을 의미하는 다음 수식을 만족하면

$$\bigwedge_{i=1}^{n} p_{old,i} = p_{new,i}$$

함수 f의 웹 브라우저 b_i와 웹 브라우저 b_j에 대한 성능비율 $ratio_{b_i/b_j}(f)$은 다음과 같다.

$$ratio_{b_i/b_j}(f) = \frac{m_i(f)}{m_j(f)}$$

정의 4. 웹 브라우저 집합 $B=\{b_1, b_2, ..., b_n\}$이 있고 성능버그 기준 $threshold$을 정했을 때,

$$\bigvee_{i=1}^{n} \bigvee_{j=1, j\neq i}^{n} (ratio_{b_i/b_j}(f) \geq threshold)$$

를 만족하는 함수 f를 성능버그 후보라 한다.

정의 4에 의해서 어떤 함수가 특정 브라우저에서 실행시간이 다른 웹 브라우저와 비교하였을 때 성능버그 기준보다 빠르거나 느리다면 성능버그 후보이다. 예를 들어 $threshold$가 4라고 할 때, 어떤 함수 f_1의 $ratio_{IE/FF}(f_1)$가 5라면 f_1은 성능버그 후보이며, $ratio_{IE/FF}(f_2)$가 0.2인 함수 f_2는 $ratio_{FF/IE}(f_2)$가 5이고, $ratio_{FF/IE}(f_2)$가 성능버그 기준보다 크기 때문에 f_2 또한 성능버그 후보이다.

4. 크로스-브라우저 성능버그 검출 방법

이번 장에서는 성능 프로파일링을 통해 웹 어플리케이션에서 성능버그 후보를 탐지하고 그 중에서 성능버그를 찾는 방법을 소개한다. 먼저 본 논문에서 크로스-브라우저 성능 프로파일링을 위한 프레임워크는 (1) 성능 프로파일링을 위한 프로그램 수정, (2) 웹 브라우저에서 웹 어플리케이션을 실행하고 사용자 입력을 수행하는 테스팅, (3) 성능 프로파일링 결과 분석을 순차적으로 수행하여 성능버그 후보를 탐지한다. 다음 각 절에서는 프레임워크의 각 단계를 순서대로 설명하도록 한다.

4.1 프로그램 수정

프로그램 수정(program instrumentation) 방식 성능 프로파일링은 프로그램에 코드를 추가하여 측정하려는 함수의 실행시간을 구하는 방법이다.

웹 브라우저에서 사용가능한 프로파일러 대신에 프로그램 수정 방식을 사용한 이유는 프로파일러가 자바스크립트 성능에 영향을 주기 때문이다. Internet Explorer와 Chrome은 프로파일러가 내장되어 있으며, Firefox의 경우 Firebug[19]라는 플러그인의 프로파일러 기능을 사용할 수 있다. 이처럼 각 웹 브라우저마다 프로파일러가 다르기 때문에 웹 브라우저 때문이 아닌 프로파일러의 차이에 의해서 잘못된 성능버그를 찾을 수 있다. 예를 들어 다음은 try-catch가 있는 예제 코드이며 4번 줄에서 예외를 발생시키면 5번 줄에서 예외를 처리하게 되며, 이를 10,000번 반복하는 코드이다.

```
1 function exampleTryCatch() {
2  for(var i = 0; i < 10000 ; i++) {
3  try {
4  throw "error";
5  } catch (e) { }
6  }
7 }
```

이 자바스크립트 함수를 20번 호출하고 각 호출시 실행시간을 프로파일러와 프로그램 수정 방식으로 측정한 결과는 표 1과 같다. 여기서 두 가지 방식 모두 실행시간은 포괄 시간이며 3가지 측정값은 20번 호출한 시간의 평균값이다. 프로파일러를 사용하였을 때, Firefox가

표 1 프로그램 수정 방식과 기존 프로파일러의 성능 프로파일링 결과(단위: ms)

Table 1 Performance profiling results of program instrumentation and traditional profiler

	IE	FF	CH
Program instrumentation(w/o profiler)	90.79	3.08	2.10
Program instrumentation(with profiler)	92.49	657.31	2.40
Profiler	97.26	691.13	1.80

Internet Explorer보다 7.17배 느리지만, 프로파일러를 사용하지 않았을 때 Firefox가 오히려 29.5배 빠른 것으로 나타났다. 따라서 다른 자바스크립트 코드의 성능에 영향을 주지 않는 프로그램 수정 방식이 기존 프로파일러보다 더 정확하게 성능을 측정하는 것을 알 수 있으며 본 논문에서는 기존 프로파일러 대신에 프로그램 수정 방식을 사용하였다.

프로그램 수정 방식 성능 프로파일링은 시간을 측정하고자 하는 함수의 시작과 끝 부분에 시간을 측정하는 함수를 호출하고, 시작과 끝의 시간 차이로 시간을 측정하려는 함수의 포괄 시간(inclusive time)을 구한다. 프로그램 수정 방식에 필요한 자바스크립트 코드의 의사코드(pseudo code)는 그림 1과 같다. 1, 2번 줄에서는 측정한 실행시간을 기록할 변수, 실행시간을 구분할 고유번호를 저장할 변수, 결과를 출력할 변수를 선언하며, 4~10번 줄 `probe_start()` 함수를 측정 구간이 시작하는 부분에서 호출하면 측정구간의 고유번호, 이름과 시

```
 1 var prove_printArea;var prove_callId = 1;
 2 var prove_time = Array();
 3
 4 function probe_start(name) {
 5   var temp_id = prove_callId++;
 6   prove_time[temp_id] = Array();
 7   prove_time[temp_id]['name'] = name;
 8   prove_time[temp_id]['start'] = currentTime();
 9   return temp_id;
10 }
11
12 function probe_end(id) {
13   prove_time[id]['end'] = currentTime();
14   if( !prove_printArea ) {
15     prove_printArea = createPrintArea();
16   }
17   prove_printArea.println(
18     id + "," +
19     prove_time[id]['name'] + "," +
20     prove_time[id]['start'] + "," +
21     prove_time[id]['end']
22   );
23 }
```

그림 1 프로그램 수정 방식의 프로파일링을 위한 의사코드

Fig. 1 Pseudo code for program instrumentation performance profiling

작된 시각을 기록한다. 12~23번 줄 probe_end() 함수를 측정 구간이 끝나는 부분에서 호출하면, 종료된 시각을 구하고 고유번호에 해당하는 측정구간의 이름, 시작 시각, 종류 시각을 출력한다. 다음은 측정 코드를 삽입하는 방법을 보여주는 사용 예제이며 3번 줄에 있는 명령이 걸리는 시간을 측정한다.

```
1 function toBeTested() {
2  var probe_id = probe_start("toBeTested");
3  something_to_test();
4  probe_end(probe_id);
5 }
```

4.2 테스팅

테스팅은 웹 어플리케이션의 수정된 자바스크립트 프로그램을 여러 웹 브라우저에서 실행하고, 동일한 사용자 입력을 수행하여 성능 프로파일링 결과를 저장하는 것이다.

웹 브라우저에서 웹 어플리케이션을 실행하기 위한 사용자 입력은 Selenium[20]을 사용하여 동일한 입력을 여러 번 반복하였다. Selenium은 Firefox 플러그인을 제공하며, 플러그인에 사용자 입력을 기록하는 기능과 기록된 입력을 재생하는 기능이 있다. 또한 기록을 Java, C# 프로그램으로 변환하여 Firefox, Internet Explorer, Chrome 등에서 웹 브라우저를 자동으로 조작하는데 사용할 수 있다. 반복문을 통해서 테스트를 자동으로 여러 번 수행할 수도 있다. 또한 Selenium은 실행중인 웹 어플리케이션의 웹 페이지의 전체 또는 일부분을 읽을 수 있으며, 이 기능을 이용하여 프로그램 수정 방식에서 출력된 결과를 읽어 파일로 저장하고 전체 웹 페이지의 크기 차이가 5% 이내인지를 확인하여 동일한 웹 페이지를 출력하는지를 판단한다.

4.3 결과 분석

성능 프로파일링 결과는 각 함수의 평균 실행시간을 구하고, 각 브라우저에서 함수 실행시간으로 성능 비율을 구한다. 웹 어플리케이션이 브라우저에 따라 성능이 다르기 때문에 성능 비율의 평균의 2~4배 이내에서 성능버그 기준을 정하고 성능 비율이 기준을 넘는 함수 성능버그 후보로 한다.

분석시 측정코드에 의한 오차를 고려하여 측정코드에 의한 영향을 많이 받는 경우는 성능버그에서 제외 한다. 측정코드에 의한 오차는 그림 1의 8번 줄에서 시작시간을 측정하는 함수가 실행되고 측정구간으로 들어가기까지 소요되는 시간과 측정구간이 끝나고 13번 줄에서 종료 시각을 측정하는 함수가 실행될 때까지 소요되는 시간이다. 두 가지 소요시간의 합을 별도로 측정하고 모든 웹 브라우저에서 함수의 실행시간이 오차로 인한 소요시간의 10배 이하라면 함수는 성능버그 후보에서 제외한다. 본 논문의 실험에서 측정코드에 의한 오차를 측정한 결과는 표 2와 같다. 이를 기준으로 성능버그 후보의 실행시간이 Internet Explorer에서 0.056 ms 이내, Firefox에서 0.195 ms 이내, Chrome에서 0.029 ms 이내라면 성능버그 후보에서 제외하였다.

표 2 프로그램 수정 방식에 사용하는 측정 코드의 실행 시간 오차(단위: ms)

Table 2 Error of execution time by program instrumentation

Internet Explorer	Firefox	Chrome
0.0056	0.0195	0.0029

5. 사례연구를 통한 평가

본 논문에서 제안한 기법이 실제 웹 어플리케이션에 대하여 효과적으로 성능버그를 검출하는 지 평가하기 위하여 두 가지 사례 연구를 수행하였다. 사례연구를 통하여 확인하고자 하는 구체적인 평가 질문은 다음과 같다.

가. 실제 웹 어플리케이션에서 웹 브라우저에 따라 현저한 성능 차이를 보이는 성능버그 후보가 얼마만큼 존재하는가?

나. 실제 웹 어플리케이션에서 성능버그 후보로 발견된 코드를 수정할 경우 얼마만큼의 성능 향상을 보일 수 있는가?

5.1 실험 대상

사례연구를 위하여 실제 웹 어플리케이션인 The Organizer[5]와 WordPress[6]를 실험 대상으로 사용하였다. The Organizer는 일정 관리 시스템으로 일정 등록, 삭제, 노트 등의 기능이 있다. The Organizer에는 9개의 자바스크립트 소스코드 파일로 구성되어 있으며, 코드의 크기는 1,107 코드라인이며 79개의 함수가 있다. WordPress는 컨텐츠 관리 시스템으로 블로깅, 출판 등의 기능을 제공하는 웹 어플리케이션이다. 실험에 사용한 WordPress 버전은 3.5이다. WordPress의 외부 라이브러리를 제외한 자바스크립트 소스코드 파일은 59개이며, 크기는 13,390 코드라인, 1,648개의 함수로 구성되어 있다.

두 가지 실험대상의 실행정보 생성을 위해, 구체적인 사용 시나리오를 설계하여 테스팅을 수행하였다. The Organizer를 이용한 사례연구에서는 시나리오를 실행대상으로 사용하였다. 구체적으로, The Organizer 테스트 1회에 노트 추가/삭제, 할 일 추가/삭제, 연락처 추가/삭제, 일정 추가/삭제, 비밀번호 변경 기능을 순서대로 실행하는 사용 시나리오를 수행하였다. WordPress를 이용한 사례연구에서 사용한 사용 시나리오는 블로그 글 추가, 변경, 삭제를 순서대로 실행하였다. WordPress에는 “Post” 기능을 사용하는 도중에 웹 브라우저가 서버

로부터 받은 모든 자바스크립트 소스코드 파일 21개를 대상으로 분석하였다.

5.2 실험 설계

본 논문에서 제안한 성능버그 검출 기법을 앞서 제안한 두 가지 항목에 대하여 평가하기 위하여 다음과 같이 두 단계의 실험을 설계하였으며, The Organizer와 WordPress에 대해서 다음의 2단계로 실험을 수행하였다.

- 단계 1: 성능버그 후보 검출
  (1) 성능버그 탐지를 위하여 실험 대상 프로그램에 수행 시간 측정 코드를 삽입하였다.
  (2) 설계한 사용 시나리오를 따라 100회 실행을 생성하였다.
  (3) 100회의 실행 정보를 바탕으로 성능버그 기준을 판별하였다.
  (4) 판별된 성능버그 기준에 따라서 성능이상을 보이는 함수를 성능버그로 보고하였다.
- 단계 2: 검출된 성능버그 후보 검증
  (1) 검출된 각 버그를 분석하여 성능향상 가능성 여부를 판별하였다.
  (2) 성능향상 가능성이 있는 것으로 판별된 각 성능버그에 대해 프로그램을 수정하였다.
  (3) 수정된 프로그램을 설계한 사용 시나리오를 따라 100회 실행하며 성능을 측정한다.

검출된 성능버그 검증 단계에서 검출된 모든 성능버그의 실제 여부를 판별하고, 성능 향상이 가능하도록 수정하는 데에는 해당 프로그램에 대한 구체적인 배경 지식이 필요하다. 따라서 본 실험에서는 일부 성능버그에 대해서 선별적으로 사례연구를 수행하였다.

정확한 성능 측정을 위하여 성능 프로파일러를 프로그램 수정을 통해 테스트 대상 프로그램에 삽입하였다. 수동으로 성능 측정 코드를 삽입해야 하였으며, 학부생 한명이 측정코드를 삽입하는데 The Organizer의 경우 30분, WordPress의 경우 약 4.1시간 소요되었으며, 이는 함수 1개당 17초가 평균적으로 소요된 것이다. 이 작업은 추후 자바스크립트 구문 분석 도구 등을 사용하여 완전 자동화가 가능하다.

테스트에 사용한 웹 브라우저는 Internet Explorer 10, Firefox 17.0.1, Chrome 25이다. 그리고 테스트는 Intel Core i5 1.6GHz CPU와 4GB 램을 장착한 장비에서 시행하였다.

5.3 The Organizer 사례 연구

5.3.1성능버그 후보 검출

The Organizer에 79개의 함수에 측정코드를 삽입하였으며, 실행 결과로 총 62개의 함수가 호출되었다(측정코드를 삽입한 함수의 78%). 그림 2는 각 함수의 실행시간을 바탕으로 성능비율을 구한 결과이며 실행시간이 짧은 함수는 측정코드에 의한 오차를 기준으로 제외하였다. 성능 프로파일링에서 전체 호출된 함수 $F=\{f_1,\cdots f_n\}$의 성능비율 평균은 다음과 같다.

$$\sum_{i=1}^{n} ratio_{IE/FF}(f_i)/n = 2.12$$

$$\sum_{i=1}^{n} ratio_{FF/CH}(f_i)/n = 1.62$$

$$\sum_{i=1}^{n} ratio_{IE/CH}(f_i)/n = 3.35$$

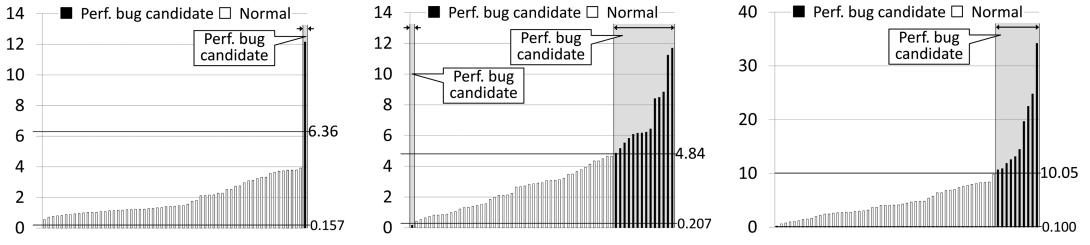
이를 바탕으로 성능버그 기준을 각 웹 브라우저간의 평균 성능비율의 3배로 정하였으며 다음과 같다.

$$threshold_{IE/FF} = 2.12 \times 3 = 6.36$$

$$threshold_{FF/CH} = 1.62 \times 3 = 4.84$$

$$threshold_{IE/CH} = 3.35 \times 3 = 10.05$$

성능버그 기준을 통해서 The Oragnizer에서 총 79개의 함수 중, 18개의 함수를 성능버그 후보를 찾았다(18/79=23%). 그림 2에서 성능버그 후보는 속이 채워진 막대로 표시되어 있다. 그림 3은 18개의 성능버그 후보를 벤 다이어그램으로 표현하였다. 3가지 성능버그 기준을 모두 넘는 경우는 없었으며, 성능버그 기준 CH/FF와 IE/CH를 동시에 넘는 함수가 7개가 있으며, IE/FF와 IE/CH를 동시에 넘는 함수가 1개 있다.



(a) $ratio_{IE/FF}(f_i)$ (b) $ratio_{FF/CH}(f_i)$ (c) $ratio_{IE/CH}(f_i)$

그림 2 The Organizer 실행 중에 호출된 함수의 성능비율

Fig. 2 Performance ratio of functions which are called in the execution of The Organizer

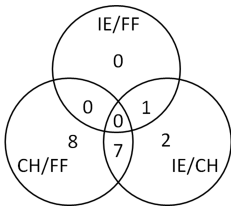


그림 3 The Organizer 성능버그 후보 벤 다이어그램
Fig. 3 Venn diagram of performance bug candidates of The Organizer

5.3.2 성능버그 후보 검증

18개의 성능버그 후보 중에서 `showPleaseWait()` 함수와 `hidePleaseWait()` 함수가 실제 성능버그임을 확인하였다. `showPleaseWait()` 함수의 경우, 성능비율 $ratio_{IE/CH}$(showPleaseWait)이 34.19로, 최고치를 보여, 성능버그 후보로 탐지되었으며, `hidePleaseWait()`는 성능비율 $ratio_{IE/CH}$(hidePleaseWait)가 6.16로 나타나서 성능버그 후보로 탐지되었다.

`showPleaseWait()` 함수는 사용자가 요청한 자료를 서버에서 가져오기 위해서 AJAX를 이용하여 통신하는 도중에 잠시 기다리라는 메시지를 표시하거나 숨기기 위해서 사용한다. 그림 4(a)와 같이 "New Note" 버튼을 클릭하여 `showPleaseWait()`를 호출하면 현재 표시중인 내용은 사라지고 "Please wait..."이라는 메시지가 표시된다. 이러한 기능을 구현하기 위해서 기존 소스코드는 그림 4(b)와 같이 밑줄 친 부분에서 DOM 객체(object)에 직접 접근하여 DOM 객체에 적용중인 CSS 속성을 수정하는 방식을 사용하였으며, 이는 DOM 객체의 CSS 클래스(class)를 변경하여 DOM 객체의 스타일을 변경하는 방식보다 느리다. CSS 클래스를 변경하는 방식으로 구현하기 위해서 다음과 같이 `special_hide` 라는 CSS 클래스를 만들었다.

```
1 <style id="special_hide" >.special_hide { display:
  none; }</style>
```

그리고 DOM 객체를 `special_hide` 클래스에 포함, 제거하는 방식으로 메시지를 숨기고 표시하는 방식을 사용하여 성능버그를 제거할 수 있다. 그림 3(c)은 밑줄 친 부분이 수정되어 성능버그가 제거된 코드를 보여준다.

`hidePleaseWait()` 함수는 `showPleaseWait()` 함수와 정반대의 기능을 수행하며, `showPleaseWait()` 함수 동일한 방법으로 성능을 향상시킬 수 있다.

그림 5는 성능버그를 제거하기 전과 후의 포함 실행시간을 보여준다. 성능버그를 제거함으로써 Firefox와 Chrome에서의 성능은 비슷하지만 Internet Explorer에서 두 함수의 실행시간이 감소하였다.

5.3 WordPress 사례 연구

5.3.1 성능버그 후보 검출

WordPress에 878개의 함수에 측정코드를 삽입하고, 테스트 결과로 392개의 함수가 호출되었다(측정 코드를 삽입한 함수의 44.6%). 그림 6은 각 함수의 실행시간을 바탕으로 성능비율을 구한 결과이며 실행시간이 짧은 함수는 측정코드에 의한 오차를 기준으로 제외하였다. 성능 프로파일링에서 전체 호출된 함수 $F=\{f_1,\cdots f_n\}$의 성능비율 평균은 다음과 같다.

$$\sum_{i=1}^{n} ratio_{IE/FF}(f_i)/n = 1.61$$

$$\sum_{i=1}^{n} ratio_{FF/CH}(f_i)/n = 1.23$$

$$\sum_{i=1}^{n} ratio_{IE/CH}(f_i)/n = 3.17$$

이를 바탕으로 성능버그 기준을 각 웹 브라우저간의 평균 성능비율의 3배로 정하였다.

(a) Role of `showPleaseWait()`

```
1a function showPleaseWait() {
2a   tabsButtonsEnabled = false;
3a   $("mainContent").style.display = "none";
4a   $("pleaseWait").style.display = "block";
5a } // End showPleaseWait().
```

(b) `showPleaseWait()` with performance bug

```
1b function showPleaseWait() {
2b   tabsButtonsEnabled = false;
3b   $('mainContent').className = 'special_hide';
4b   $("pleaseWait").className = '';
5b } // End showPleaseWait().
```

(c) `showPleaseWait()` without performance bug

그림 4 성능버그 `showPleaseWait()`의 기능과 성능버그 수정
Fig. 4 Role of performance bug `showPleaseWait()` and fix of performance bug

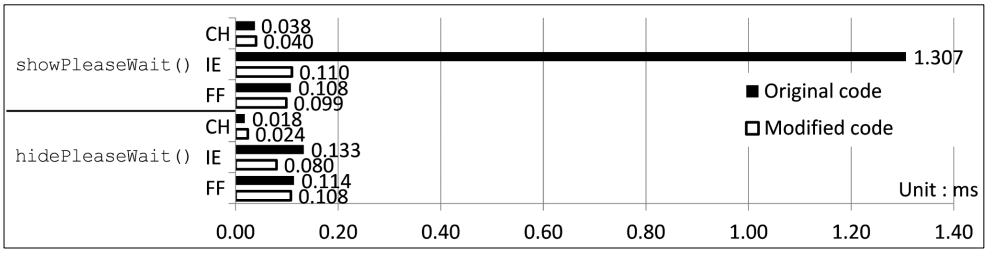
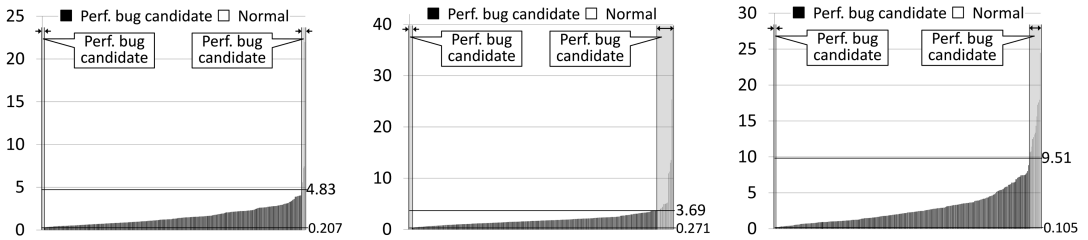


그림 5 The Organizer에서 성능버그 디버깅을 통한 성능 향상
Fig. 5 Performance improvement of performance bugs of The Organizer



(a) $ratio_{IE/FF}(f_i)$ (b) $ratio_{FF/CH}(f_i)$ (c) $ratio_{IE/CH}(f_i)$

그림 6 WordPress 실행 중에 호출된 함수의 성능비율
Fig. 6 Performance ratio of functions which are called in the execution of WordPress

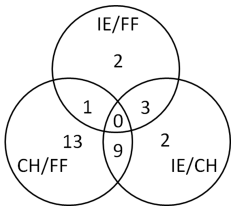


그림 7 WordPress 성능버그 후보 벤 다이어그램
Fig. 7 Venn diagram of performance bug candidates of WordPress

$$threshold_{IE/FF} = 1.61 \times 3 = 4.83$$
$$threshold_{FF/CH} = 1.23 \times 3 = 3.69$$
$$threshold_{IE/CH} = 3.17 \times 3 = 9.51$$

성능버그 기준을 통해서 WordPress에서 30개의 성능버그 후보를 찾았으며(30/392=7.6%), 그림 6에서 성능버그 후보는 음영 안의 막대이다. 그림 7은 30개의 성능버그 후보를 벤 다이어그램으로 표현하였다. 3가지 성능버그 기준을 모두 넘는 경우는 없었으며, 성능버그 기준 CH/FF와 IE/CH를 동시에 넘는 함수가 9개, IE/FF와 IE/CH를 동시에 넘는 함수가 3개, IE/FF와 CH/FF를 동시에 넘는 함수 1개가 있다.

5.4.2 성능버그 후보 검증

30개의 성능버그 후보 중에서 `visibility()` 함수, `clean()` 함수, `set()` 함수가 성능버그임을 확인하였다. 이 절의 다음 부분은 성능버그에 대해서 기존 소스코드의 기능, 성능버그가 검출된 이유, 성능버그의 디버깅에 대해서 설명한 후에 성능이 향상된 결과를 보여준다. 그리고 본 논문에서 찾은 성능버그 정보는 WordPress 버그 리포트로 제출한 상태이다.

• `visibility()` 함수

이 함수는 $ratio_{IE/CH}$(visibility)값이 12.49로 나왔기 때문에 성능버그로 검출되었으며, 선언된 지역변수가 조건에 따라 사용되지 않을 수 있는 것이 원인이다. 앞서 설명한 `showPleaseWait()`와 `hidePleaseWait()` 함수와 비슷하게, `visibility()` 함수는 특정 DOM 객체가 화면에 표시하거나 숨기는 역할을 한다. 이 함수의 소스코드는 그림 8(a)와 같으며, 밑줄 친 2a번 줄에서 `views`와 `hide` 변수는 4a번 줄의 조건문이 참일 때만 사용된다. 따라서 조건문 안에서 두 변수를 선언하면 성능이 향상된다. 테스트 1회에서 `visibility()` 함수는 23회 호출되지만 조건문이 참이 되는 경우는 그중 4회였다.

그림 8(b)는 성능버그를 제거한 소스코드이다. 밑줄 친 부분과 같이 조건문 내부에서 변수를 선언하여 그림 9에서 모든 웹 브라우저에서 모두 성능향상이 있음을 알 수 있다.

• `clean()` 함수

이 함수는 $ratio_{CH/IE}$(clean)이 10.2로 높게 나왔기 때문에 성능버그로 탐지되었으며, 성능버그의 원인은 정규

```
1a visibility: function() {
2a   var region = this.region,
       view = this.controller[ region ].get(),
       views = this.views.get(),
       hide = ! views || views.length < 2;
3a
4a   if ( this === view )
5a     this.controller.$el.toggleClass( 'hide-' +
   region, hide );
6a },
```

(a) visibility() with performance bug

```
1b visibility: function() {
2b   var region = this.region,view = this.controller[
   region ].get();
3b   if ( this === view ) {
4b     var views = this.views.get(),hide = ! views ||
   views.length < 2;
5b     this.controller.$el.toggleClass( 'hide-' +
   region, hide );
6b   }
7b },
```

(b) visibility() without performance bug

```
1a clean : function(tags) {
2a   var comma = postL10n.comma;
3a   if ( ',' !== comma )
4a     tags = tags.replace(new RegExp(comma, 'g'), ',');
5a   tags = tags.replace(/\s*,\s*/g, ',').replace
   (/,+/g, ',').replace(/[,\s]+$/,
   '').replace(/^[,\s]+/, '');
6a   if ( ',' !== comma )
7a     tags = tags.replace(/,/g, comma);
8a   return tags;
9a },
```

(c) clean() with performance bug

```
1b clean : function(tags) {
2b   var comma = postL10n.comma;
3b   if ( ',' !== comma )
4b     tags = tags.replace(new RegExp(comma, 'g'), ',');
5b   tags = tags.replace(/[\s]*,[,\s]*/g,
   ',').replace(/[,\s]+$/, '').replace(/^[,\s]+/, '');
6b   if ( ',' !== comma )
7b     tags = tags.replace(/,/g, comma);
8b   return tags;
9b },
```

(d) clean() without performance bug

```
1a set : function(name, value, expires, path, domain,
   secure) {
2a   var d = new Date();
3a   if ( typeof(expires) == 'object' &&
   expires.toGMTString ) {
4a     expires = expires.toGMTString();
5a   } else if ( parseInt(expires, 10) ) {
6a     d.setTime( d.getTime() + ( parseInt(expires, 10)
   * 1000 ) );
7a     expires = d.toGMTString();
8a   } else {
9a     expires = '';
10a  }
       ....
17a },
```

(e) set() with performance bug

```
1b set : function(name, value, expires, path, domain,
   secure) {
2b   if ( typeof(expires) == 'object' &&
   expires.toGMTString ) {
3b     expires = expires.toGMTString();
4b   } else {
5b     var parsed = parseInt(expires, 10);
6b     if ( parsed ) {
7b       var d = new Date();
8b       d.setTime( d.getTime()+(parsed*1000) );
9b       expires = d.toGMTString();
10b    } else {
11b      expires = ''; }
12b  }
       ....
19b },
```

(f) set() without performance bug

그림 8 WordPress의 성능버그 후보 디버깅

Fig. 8 Debugging of performance bugs in WordPress

표현식을 이용한 문자열 처리가 최적화되어있지 않은 것이다. clean() 함수는 블로그 글을 분류하기 위해 사용하는 태그(tag)에서 중복된 쉼표나 공백을 제거한다. 소스코드는 그림 8(c)이며 5a번 줄과 같이 4개의 정규표현식을 사용하였다.

동일한 기능은 밑줄 친 2개의 정규표현식을 하나로 합쳐서 구현 할 수 있으며, 그림 8(d)과 같이 성능버그를 제거하였다. 성능버그를 제거하여 그림 9에서 실행시간이 0.05 ms 단축되었다.

• set() 함수

이 함수는 성능비율 $ratio_{FF/CH}$(clean)이 32.9로 나왔기 때문에 성능버그로 검출되었으며, 성능버그의 원인은 동일한 계산을 반복하며, 조건에 따라 사용되지 않을 수 있는 지역변수를 선언한 것이다. set() 함수는 웹 브라우저의 쿠키를 추가하는 함수이다. set() 함수의 소스코드는 그림 8(e)와 같다. 쿠키는 웹 브라우저에 임시로 데이터를 저장할 수 있는 기능이며 일정 시간이 지나면 지워지도록 만료 시각을 정할 수 있다. 이 만료 시각을 계산하기 위해서 현재 시간에 쿠키가 유지되어야할 시간을 더해서 만료 시각을 정하며, 현재 시각을 구하기 위해서 2번 줄에서 new Date()를 통해 현재 시각이 저장된 객체를 생성한다. 여기서 5a번 줄의 조건문이 참일 때만 new Date()로 객체를 생성하면 실행시간이 줄어든다. 그리고 parseInt(expires, 10) 함수는 2번 호출되지만(5a번 줄과 6a번 줄) 한번 호출하고 그 값을 저장하여 다시 사용할 수 있다[15].

성능버그를 제거한 소스코드는 그림 8(f)와 같다. 5b번 줄에서 parseInt(expires, 10) 함수의 결과 값이 저장되며, 6b와 8b번 줄에서 저장된 값을 사용한다. 그리고 7b번 줄에 있는 new Date() 함수는 조건문이 참

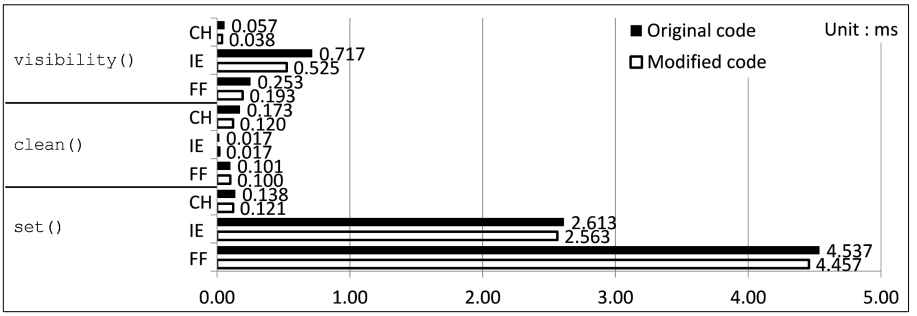


그림 9 WordPress에서 성능버그 디버깅을 통한 성능 향상
Fig. 9 Performance improvement of performance bugs of WordPress

일 때만 호출된다. 소스코드를 변경하였을 때 그림 9에 보이듯이 Firefox, Internet Explorer, Chrome 모두 실행시간이 약간 감소하였다.

6. 논의사항

6.1 성능버그 후보 검출 기법의 정확도

본 논문에서 제안한 성능버그 검출 기법은 거짓 양성 경보(false positive)와 거짓 음성 경보(false negative)를 생성할 수 있다.

본 기법은 웹 브라우저 별로 다른 기능을 제공하도록 작성된 웹 어플리케이션에 대하여 거짓 양성 경보를 생성할 수 있다. 본 기법은 검증 대상 웹 어플리케이션의 모든 함수가 웹 브라우저에 상관없이 동일한 동작을 한다는 가정을 하고 성능버그를 찾았다. 하지만 웹 어플리케이션이 실행 환경 변수를 참조하여 웹 브라우저에 특화된 동작을 제공하도록 프로그램 되어있을 수 있다. 이와 같은 경우, 웹 브라우저마다 실행하는 동작의 내용이 다르므로, 실행 시간의 차이가 버그라고 볼 수 없다.

웹 브라우저에 따라 실행 시간의 차이는 발생하지만 이를 개선할 수 있는 코드가 불가능한 성능버그 후보의 경우도 거짓 양성 경보에 해당한다. 성능버그는 웹 어플리케이션 내에 존재하는 비효율적인 프로그래밍으로 말미암아 같은 동작을 수행하는데 필요이상의 시간이 소요되는 코드를 뜻한다. 하지만 웹 브라우저가 대체 불가능한 특정 명령을 수행함에 있어서 다른 웹 브라우저보다 월등히 높은 성능을 보일 경우, 성능버그와 상관없는 코드가 성능버그 후보로 탐지될 수 있다.

본 논문에서 제안한 기법은 웹 어플리케이션의 실행정보를 바탕으로 분석을 수행하므로, 실행 생성 과정에서 실행되지 않은 코드 영역에서 존재하는 성능 버그에 대해서 거짓 음성 결과가 가능하다. 성능버그를 최대한 검출하기 위해서는 모든 함수의 동작이 포함되도록 다양한 사용 시나리오를 실행 생성 과정에서 사용해야 한다.

본 논문의 기법에서 함수 단위로 실행시간을 측정하기 때문에 거짓 음성이 발생할 수 있다. 예를 들어 명령문 2개가 있는 성능버그인 함수가 있고 Firefox에서 첫 번째 명령문이 Chrome에 비해서 느리게 실행되지만, 두 번째 명령문은 반대로 Firefox보다 Chrome에서 느리게 실행된다면 상쇄되어서 성능버그로 감지되지 않을 수 있다. 현재 기법에서는 성능을 함수 단위보다 측정하고 있지만, 더 세밀한 단위(예를 들어, 구문 단위)로 시간 성능을 측정하고 비교하면 이와 같은 거짓 음성 경보를 줄일 수 있다.

6.2 성능버그 후보 정렬 방안

성능버그 탐지를 통하여 프로그램의 효과적인 성능 향상을 이루기 위해서는, 탐지된 성능버그를 중요도에 따라 사용자에게 제공하는 것이 중요하다. 기본적으로, 본 논문에서 제안한 기법에서는 평균 실행시간 비율이 큰 함수를 우선적으로 정렬할 수 있다. 이에 더하여, 성능버그로 탐지된 함수 들 중 다음의 특성을 가지는 함수는 성능버그로 의심도가 높을 것으로 예상되므로, 우선적으로 정렬할 경우, 사용자가 효율적으로 성능 향상을 위한 디버깅을 수행할 수 있을 것으로 예상된다.

먼저 성능비율을 통한 비교에서 많은 웹 브라우저에서 일관되게 성능비율의 차이가 큰 함수에 우선순위를 부여하는 방법이 있다. 모든 웹 브라우저에서 성능비율의 차이가 많이 난다면, 개발자가 각 웹 브라우저의 특성에 맞는 코드를 작성하지 않았기 때문일 수 있기 때문이다. 또한 다양한 웹 브라우저 중에서 성능향상이 가능한 방법 하나를 찾는 것이 단 하나의 웹 브라우저에서 성능향상이 가능한 방법을 찾는 것보다 더 쉽다.

또한 절대적인 실행시간이 긴 함수에 높은 우선순위를 부여하는 방법이 있다. 절대적인 실행시간이 길수록 웹 어플리케이션이 느리게 동작하는 원인이 될 가능성이 커지므로, 이를 먼저 해결하는 것이 실행시간이 짧은 함수를 고치는 것 보다 품질향상에 더 유익하다. 사용자

에게 특정 시간 이상 걸리는 함수만을 우선적으로 보여주는 방법을 사용하면, 사용자가 성능향상에 더 중요한 함수를 확인할 수 있다.

그리고 반복적으로 호출되는 함수에 높은 우선순위를 부여하는 방법도 있다. 절대적인 실행시간이 짧다고 하더라도 반복되는 함수라면 큰 성능 향상을 보일 수 있다. 특히 반복적으로 호출되는 함수가 특정 상황에서만 느려지는 경우가 존재할 수 있으며, 성능비율을 통한 비교에서는 평균 실행시간을 기준으로 판단하기 때문에 성능 버그로 탐지하지 못하거나 낮은 성능비율이 나타난다. 그러나 개발자가 특정 상황에서 급격하게 느려지는 현상을 알지 못하고 함수를 작성할 수도 있기 때문에 반복적으로 호출되는 함수 중에서 실행 시간이 크게 차이가 나는 함수에 높은 우선순위를 부여하는 방법을 사용하면, 특정 상황에서 발생하는 성능버그를 더 효율적으로 찾을 수 있다.

7. 결 론

본 논문은 웹 어플리케이션에서 성능 저하를 발생시키는 성능버그를 테스트를 통해 자동적으로 탐지하는 크로스-브라우저 프로파일링 기법을 개발하고 실험을 통하여 기법의 유효성을 평가한 내용을 소개하였다. 크로스-브라우저 프로파일링 기법은 특정 브라우저에서 성능이 급격히 저하되는 함수를 성능버그로 정의하고 이를 성능 프로파일링을 통해 찾는 방법이다.

크로스-브라우저 프로파일링 기법을 구현하고 3개의 웹 브라우저에서 실제 웹 어플리케이션 테스팅에 적용한 결과 The Organizer에서 18개의 성능버그를 찾았으며 WordPress에서 30개의 성능버그를 찾았다. 그리고 그 중에 성능 향상이 가능한 실제 성능버그를 The Organizer에서 2개, WordPress에서 3개를 찾음으로써 본 논문에서 제시한 방법이 성능버그를 찾고 웹 어플리케이션의 성능을 향상시키는데 도움이 되는 것을 확인하였다.

향후연구로 본 기법의 거짓 양성 경보와 거짓 음성 경보를 줄이는 방법에 관한 연구와 추가적인 실행정보 분석을 통해 성능버그 후보 정렬 방안을 구현하는 연구를 진행할 계획이다.

References

[1] Aberdeen Group, "Application Performance Management: The Lifecycle Approach Brings IT and Business Together," Jun. 2008.

[2] G. Jin, L. Song, X. Shi, J. Scherpelz, S. Lu, "Understanding and Detecting Real-World Performance Bugs," *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[3] C. Jones, "Software Quality and Software Economics," *Software Tech News*, vol.13, no.1, pp.10-14, Apr. 2010.

[4] S. Zaman, B. Adams, A. E. Hassan, "Security Versus Performance Bugs: A Case Study on Firefox," *Proc. IEEE Working Conference on Mining Software Repositories (MSR)*, 2011.

[5] F. Zammetti, Practical Ajax Projects with Java Technology, 1st Ed., Apress, Berkeley, 2006.

[6] WordPress.org, WordPress ver. 3.5 [Online]. Available: http://wordpress.org/download/source (downloaded 1 Dec. 2012)

[7] M. Jovic, A. Adamoli, M. Hauswirth, "Catch Me If You Can: Performance Bug Detection in the Wild," *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.

[8] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, S. Shayandeh, "AppInsight: Mobile App Performance Monitoring in the Wild," *Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[9] E. Kiciman, B. Livshits, "AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications," *ACM Transactions on The Web*, vol.4, no.4, Sep. 2010.

[10] C.-P. Bezemer, A. Zaidman, "System Load Characterization Using Low-Level Performance Measurements," *Delft Univ. of Technology, Tech. Rep. TUD-SERG-2012-006*, 2012.

[11] C.-P. Bezemer, A. Zaidman, A. Hoeven, A. Graaf, M. Wiertz, R. Weijers, "Locating Performance Improvement Opportunities in an Industrial Software-as-a-Service Application," *Proc. IEEE International Conference on Software Maintenance (ICSM)*, 2012.

[12] S. Han, Y. Dong, S. Ge, D. Zhang, T. Xie, "Performance Debugging in the Large via Mining Millions of Stack Traces," *Proc. International Conference on Software Engineering (ICSE)*, 2012.

[13] B. Livshits, E. Kiciman, "Doloto: Code Splitting for Network-Bound Web 2.0 Applications," *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2008.

[14] S. Zaman, B. Adams, E. Hassan, "A Qualitative Study on Performance Bugs," *Proc. IEEE Working Conference on Mining Software Repositories (MSR)*, 2012.

[15] N. C. Zakas, High Performance JavaScript, 1st Ed., O'Reilly Media, Sebastopol, 2010.

[16] YSlow [Online]. Available: http://yslow.org (downloaded 1 Dec. 2012)

[17] Compuware dynaTrace AJAX Edition, [Online]. Available: http://www.compuware.com/ (downloaded 1 Dec. 2012)

[18] G. Richards, S. Lebresne, B. Burg J. Vitek, "An

Analysis of the Dynamic Behavior of JavaScript Programs," *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.
[19] Mozilla, Firebug, [Online]. Available: https://getfirebug.com/ (downloaded 1 Dec. 2012)
[20] Selenium [Online]. Available: http://seleniumhq.org/. (downloaded 1 Dec. 2012)

박 용 배
정보과학회논문지 : 컴퓨팅의 실제 및 레터
제 19 권 제 8 호

홍 신
2007년 KAIST 전산학과 학사. 2010년 KAIST 전산학과 석사. 2010년~현재 KAIST 전산학과 박사과정. 관심분야는 동시성 프로그램 테스팅, 소프트웨어 검증

김 문 주
정보과학회논문지 : 컴퓨팅의 실제 및 레터
제 19 권 제 8 호 참조