AtomicitySanitizer: 멀티쓰레드 C 프로그램에 대한 실행연속성 위반 결함 검출 도구

AtomicitySanitizer: Effective Runtime Atomicity Violation Detector for Multithreaded C Programs

홍신[1], 김윤호[2], 김문주[2], 윤석영[3], 정한웅[3], 박사천[3]

한동대학교 전산전자공학부[1], KAIST 전산학부[2], 삼성전자[3]

hongshin@handong.edu, moonzoo@cs.kaist.ac.kr, kimyunho@kaist.ac.kr,
seok0.yoon@samsung.com, hw7884.jung@samsung.com, sachem.park@samsung.com

요 약

실행연속성위반은 멀티쓰레드 프로그램에서 발생하는 동시성 결함의 일종으로 한 쓰레드가 특정 공유 자원을 연속적으로 접근 하는 중에 다른 쓰레드가 의도치 않게 간섭을 할 경우 오류가 발생하는 결함이다. 실행연속성위반은 가장 빈번하게 발생하는 동시성 결함의 한 종류이지만, 멀티쓰레드 C/C++ 프로그램을 대상으로, 결함검출을 지원하는 도구가 부족한 실정이다. 본 논문에서는 멀티쓰레드 프로그램의 실행을 관찰함으로써 실행연속성 위반 결함을 효과적인 검출하는 AtomicitySanitizer 결함검출기의 동적 분석과 구현을 설명한다. 또한, AtomicitySanitizer를 소프트웨어 개발 과정에 적용함으로써 해당 도구가 실제 동시성 결함 검출에 효과적임을 보인 사례연구를 소개한다.

1. 서 론

멀티코어 아키텍처의 발전과 보급에 따라 이를 효과적으로 활용하기 위해 많은 소프트웨어 시스템이 멀티쓰레드 프로그램 형태로 개발되고 있다[1]. 멀티쓰레드 프로그래밍이란 복수 개의 실행 흐름, 즉 복수 개의 쓰레드(thread)를 동시에 실행하는 프로그램으로, 복수 개의 프로세서를 효과적으로 구동할 수 있게 할 뿐만 아니라, 동시적 입력에 따른 프로그램의 반응성, 여러 개의 하드웨어의 동시적 관리를 통한 운용효율성(utilization)이 높은 프로그램을 수월하게 작성할 수 있어 실제 소프트웨어 개발에 널리 사용되고 있다[2]. 이와 같은 장점에도 불구하고, 멀티쓰레드 프로그래밍에서는 각 쓰레드의 공유 자원(shared resource)에 대한 접근에 오동작이 발생할 경우, 쓰레드 간의 의도하지 않은 상호작용이 발생하여 동시성 오류(concurrency error)가 발생하게 되는 데, 이러한 동시성 오류의 증상은 비결정적인 쓰레드 실행 순서(non-deterministic thread scheduling)에 따라 민감하게 나타나므로, 오류의 탐지와 디버깅이 어려워 소프트웨어 품질 관리에 난점으로 알려져 있다.

실행연속성위반(atomicity violation)은 쓰레드 간 실행경합(race condition)에 따른 동시성 결함의 한 종류로, 한 쓰레드가 특정 공유 데이터(shared data)를 연속적으로 접근 하는 중에 다른 쓰레드가 의도치 않게 간섭을 할 경우 오류가 발생하는 결함이다. 실제 멀티쓰레드 프로그램 중에는 특정 쓰레드가 특정 공유 데이터를 점유하여, 다른 쓰레드의 간섭이 발생하지 않는 다는 가정 하여서 마치 비동시적 순차적 프로그램과 같이 연속적인 읽기, 쓰기 명령을 내리는 실행연속성보장 코드 구간(atomic region)이 자주 작성된다. 하지만, 실행연속성보장 코드 구간에 대한 실제 쓰레드 동기화(synchronization)가 해당 공유 데이터 접근의 실행 연속성(atomicity)을 제대로 보장하지 않아서, 쓰레드 간의 간섭이 발생하게 되고 이 때 실행이 비정상적 동작으로 이어져 프로그램 오동작이 발생하게 된다. 2006년 Lu 등의 조사 연구 결과에 따르면, 실행연속성위반은 오픈소스 소프트웨어 개발 과정 중에서 가장 빈번히 발생하는 동시성 결함의 유형으로 밝혀졌으며[3], 지난 수년간 여러 선행 연구를 통해 실행연속성위반을 검출하는 다양한 정적/동적 분석 기법들이 제시되어왔다[4]. 하지만, 현재 멀티쓰레드 C 프로그램을 대상으로 실행연속성위반의 검출을 효과적으로 지원하는 분석도구가 공개 되지 않아, 해당 결함의 효과적인 분석이 어려운 상황이다.

본 논문에서는 C 언어로 작성된 멀티쓰레드 프로그램의 실행을 관찰함으로써 실행연속성 위반 결함으로 의심 되는 코드를 자동으로 검출하는 AtomicitySanitizer이라는 새로운 동작 결함 검출 도구를 LLVM[5]을 기반으로 개발하고 실제 소프트웨어 테스팅 과정에 적용한 사례를 소개한다. AtomicitySanitizer은 LLVM 기반 동적 분석 프레임워크인 Sanitizer 상에서 구현되어, 임베디드 소프트웨어와 같이 복잡하고 다양한 프로그램 요소를 포함하고 있는 실제 프로젝트에 대해서도 효과적인 오류 검출 기능을 제공하였다. 또한, AtomicitySanitizer은 동시성 결함 분석과 디버깅 과정에 효과적인 인터페이스를 추가로 제공함으로써, 실제 개발자가 효과적이며 효율적으로 문제 상황을 분석할 수 있도록 지원하도록 설계되었다.

본 논문의 2장에서는 실행연속성위반 결함 검출을 위한 동적 분석 모델을 소개한 후, 3장에서는 이를 도구로 구현한 AtomicitySanitizer를 설명한다. 4장에서는 구현한 AtomicitySantizer를 실제 소프트웨어 테스팅에 적용한 사례를 통해 개발한 도구의 실용성을 논의한다. 마지막 5장에서는 결론으로 본 논문을 마무리 한다.

2. 동적 결함 검출 기법

2.1. 실행 연속성 위반 결함

실행연속성(atomicity)이란 멀티쓰레드 프로그램의 특정 코드구간의 실행이 어떠한 쓰레드 스케쥴링 상황에서도 다른 쓰레드의 영향을 받지 않아야 한다는 요구사항이다. 실행연속성이 요구되는 코드 구간(atomic code region)은 동기화 명령을 적확히 사용하여 통해 실행연속성을 항상 보장하면서도, 실행효율성을 고려해 가능한 다른 쓰레드와 동시에 실행되도록 구현되어야 한다. 실행연속성 위반 결함이란 잘못된 동기화 명령으로 실행연속성이 의도된 코드 구간의 실행이 스케쥴링에 따라 상이한 결과가 발생하는 결함을 뜻한다.

그림1은 실행 연속성 위반 결함의 예로, `write()` 함수 전체의 실행은 실행연속성이 의도되었으나, 두 쓰레드에서 이를 동시에 실행할 때, 의도치 않은 간섭이 발생하는 경우이다. `write()` 함수는 2행에서 `find_unused()`는 현재 사용되지 않는 하나의 블록을 찾은 후, 3행에서 `assign()` 함수를 실행하여 해당 파일에 배정한다. 이 때 `find_unused()`와 `assign()` 함수 각각은 동기화가 이루어져 쓰레드 간 간섭이 발생하지 않는다고 하더라도, 두 함수의 연속적 실행을 보장하는 동기화 명령이 존재하지 않을 경우, 그림1과 같은 쓰레드 스케쥴링이 발생가능하다. 이 때 만약 `Thread 1`과 `Thread 2`가 서로 다른 두 개의 파일에 대해 `write()`를 수행하나 `find_unused()`에서 동일한 블록을 반환할 경우, 같은 블록이 두 개의 서로 다른 파일에 할당하는 오동작이 발생한다.

본 논문의 연구는 삼성전자의 지원을 받아 수행되었음.

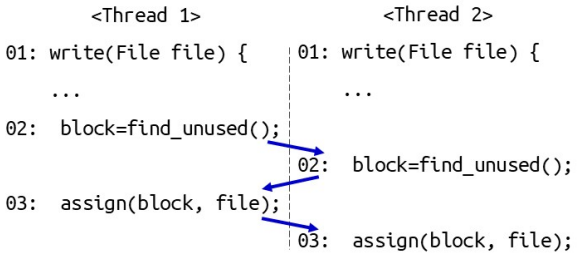

그림 1. 실행 연속성 위반 결함의 예

2.2 결함 검출을 위한 동적 분석

실행연속성위반을 인한 오류는 비결정적 쓰레드 스케쥴링 상황에서 특정한 쓰레드 간 실행 순서 관계가 발생할 때만 발생하기 때문에, 수많은 쓰레드 스케쥴링 중 임의의 상황만을 관찰하는 일반적인 테스팅 으로는 오류의 효과적인 검출이 어렵다. 제한된 테스트 실행 정보를 바탕으로 실행 연속성 위반을 효과적으로 검출하기 위한 방식으로 (1) 테스트 실행에서 프로그램에서 발생 가능한 다양한 동작을 포괄하는 모델을 추출한 후, (2) 추출된 모델을 바탕으로 발생 가능할 것으로 예상되는 쓰레드 스케쥴링을 생성하고, (3) 생성된 쓰레드 스케쥴링 상황에서 실행 연속성 위반을 탐지하는 예측적 분석(predictive analysis)을 통하여 실행 연속성 위반 결함을 효과적으로 검출할 수 있다[4].

본 연구에서는 실행연속성 결함의 예측적 분석을 위해 동적 분석 체계를 다음과 같이 정의하여 사용하였다. 본 분석은 멀티쓰레드 프로그램 실행에서 (1) 메모리 접근 명령, (2) 쓰레드 제어 명령, (3) 쓰레드 간 동기화 명령이 발생하는 경우, 이를 실행한 쓰레드 정보, 연관된 코드 라인 정보, 연관된 메모리 주소 정보를 추출하여 다음과 같은 동시성 실행 모델을 생성 한다:

- 프로그램은 유한 개의 메모리 주소 M와 유한 개의 쓰레드 집합 T을 가지며, 고유의 식별자를 가지는 각 쓰레드는 유한 개의 명령 순열로 구성된다.
- 각 명령은 연산자와 피연산자의 쌍으로 정의되며, 그 종류는 다음과 같다:
  - 메모리 주소 $m \in M$에 대해 *read(m), write(m)*
  - 메모리 주소 $m \in M$에 대해 *lock(m), unlock(m)*
  - 쓰레드 식별자 $t \in T$에 대해 *fork(t), join(t)*
- 하나의 실행연속성을 갖는 코드 구간은 한 쓰레드 내의 연속적인 명령 순열로 표현된다.
- Happens-before 관계. 두 명령 p와 q는 다음 네 가지 중 하나의 조건에 속할 때 $p < q$가 성립 된다:
  - p가 q와 같은 쓰레드 내에서 선행할 때
  - p가 *fork(t)*이고 q가 쓰레드 t 에 속한 명령일 때
  - p가 쓰레드 t 의 명령이며, q가 *join(t)*일 때
  - $p < r$ 이며 $r < q$ 인 명령 r 이 존재할 때
- Mutual-exclusion 관계. 동시 실행이 불가능한 명령의 관계를 기술하는데 필요한 함수 L은 다음과 같다:
  - 명령 p에 대한 $L(p)$는 해당 명령이 실행될 때 이를 실행하는 쓰레드가 점유하는 lock 메모리 주소들의 집합임
  - 한 쓰레드 속한 두 명령 p와 q에 대해서 $L(p,q)$는 p에서부터 q까지 연속적으로 해당 쓰레드가 점유하는 lock 메모리 주소들의 집합임
- 실행연속성 위반 결함[4]. 한 실행연속구간에 속한 p와 p', 그리고 p와 다른 쓰레드에 속한 q가 다음 조건을 모두 만족할 때, (p, q, p')은 결함조건을 만족함:
  - p, q, p'의 피연산자 메모리 주소가 동일함
  - p 혹은 q의 연산자가 write이어야 함
  - p' 혹은 q의 연산자가 write이어야 함
  - $q \not< p$ 이고 $p' \not< q$ 이어야 함
  - $L(p, p') \cap L(q) = \varnothing$ 이어야 함

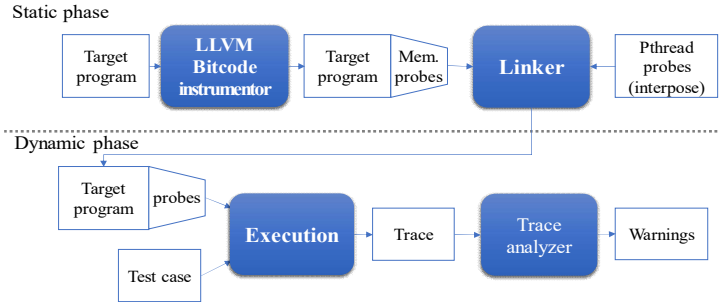

그림 2. AtomicitySanitizer의 구조와 실행 흐름

3. 동적 결함 검출 도구 AtomicitySantizer[1)]

3.1. AtomicitySanitizer의 동작과 구현

본 연구에서는 앞서 제안한 동적 결함 검출 분석 기법을 멀티쓰레드 C/C++ 프로그램에 적용할 수 있는 동적 결함 검출 도구인 AtomicitySanitizer를 개발하였다. 그림 2는 AtomicitySanitizer의 주요 컴포넌트와 실행흐름을 개괄적으로 설명하고 있다. AtomicitySanitizer는 크게 정적 단계에 이은 동적 단계의 두 단계로 작동 된다. 정적 단계에서 AtomicitySantizer는 검증대상 C/C++ 프로그램의 소스코드를 입력으로 받은 후 LLVM 중간코드(LLVM Bitcode) 수준의 프로그램 수정(instrumentation)을 통해 프로그램 트레이스를 추출하는 탐침(probe)를 삽입한다. 탐침은 프로그램 모델 생성에 필요한 런타임 정보를 추출하는 역할을 하는데, AtomicitySanitizer는 정적으로 주소가 할당된 메모리, 포인터로 접근하는 메모리에 대한 명령, 동기화 명령, 쓰레드 제어 명령, 함수의 호출 전과 후에 탐침을 삽입한다. 프로그램 수정 모듈은 LLVM IR Pass 형태로 구현되었다. 프로그램 수정의 결과로는 탐침이 삽입된 LLVM 중간코드가 생성된다. 생성된 LLVM 중간코드는 탐침에 대한 런타임 모듈과 링크되어 실행 가능한 객체 코드로 완성된다.

동적 단계에서는 탐침이 부착되어 완성된 검증대상 프로그램을 테스트 케이스로 실행시킨다. 실행 중에 탐침은 프로그램 내부 정보를 별도의 파일에 출력하여 프로그램 모델 생성에 필요한 런타임 정보를 추출한다. 각 실행에 대하여 추출된 런타임 정보는 분석 모듈로 전달되어 프로그램 모델로 생성된 후, 2장에서 정의한 실행연속성 결함 조건에 맞는 명령이 있는 지 검사가 수행된다. 이 때 실행연속성 결함 조건이 발견된 경우, 관련된 명령의 정보를 사용자에게 경보(warning)로 출력한다.

3.2. 사용자 인터페이스

사용자는 검증대상 프로그램 코드에 AtomicitySanitizer가 제공하는 API 함수를 호출하는 형태로 실행연속구간을 정의할 수 있다. AtomicitySanitizer는 `atomic_begin()`와 `atomic_end()`를 API함수를 제공하는데, AtomicitySanitizer는 한 쓰레드가 이 두 함수를 실행하는 사이의 명령을 하나의 실행연속구간으로 인식하여 분석을 수행한다. `atomic_begin()`과 `atomic_end()`가 중첩적으로 실행될 경우에는 가장 먼저 실행된 `atomic_begin()`와 `atomic_end()` 사이를 실행연속구간으로 정의한다.

AtomicitySanitizer는 경보 정보를 사용자의 관심에 맞게 제공하기 위하여 다음 2가지 방법을 제공 한다:

- 결함을 구성하는 명령의 구체적인 정보를 제공하기 위해, 코드 라인 정보는 물론 사용자가 지정한 k 개의 호출 스택(call stack) 정보를 제공한다. 호출 스택 정보는 함수의 시작과 종료 시점에 실행되는 탐침이 발생하는 런타임 정보를 통해 구성된다.
- AtomicitySanitizer 라이브러리에서 `atomic_observe(m)` API 함수를 제공하고, 검증대상 프로그램 내에서 해당 API가 사용된

1) AtomicitySanitizer는 향후 오픈소스 프로젝트 형태로 공개할 계획이다.

표 2. 결함 검출 분석 결과

Versions	Atomicity Sanitizer	Thread Sanitizer	AtomicitySanitizer ∪ ThreadSanitizer
Original	0	0	0
Fault-seeded	14	30	3

경우, *m*으로 주어진 메모리 주소에 해당하는 실행연속성 위반 결함만을 선별하여 사용자에게 경보로 제공한다.

4. 사례 연구

4.1. 검증 대상

본 연구에서는 AtomicitySanitizer을 실제 프로젝트 상황에 적용함으로써 도구의 실효성을 확인하였다. 본 사례 연구에서 AtomicitySanitizer를 적용한 검증대상 프로젝트는 계산량이 많은 작업을 멀티코어 프로세서 상에서 병렬적으로 배치하고 그 결과를 종합하는 분산작업용 쓰레드 풀링(thread pooling) 라이브러리다. 검증대상 프로그램은 응용 프로그램이 분산처리를 효과적으로 지원할 수 있도록 (1) 개별 쓰레드를 할당 받는 명령, (2) 계산에 필요한 데이터를 쓰레드 개별 공간에 수신하는 명령, (3) 계산을 수행하는 명령, (4) 사용한 쓰레드를 쓰레드 풀에 반납하는 명령의 총 4가지 대표적인 명령을 지원하는 API를 제공한다. 검증 대상 시스템은 ARM 아키텍쳐 임베디드 시스템을 대상으로 하여, 전체 코드는 1만 라인 이상의 C/C++ 코드로 POSIX Thread 모델을 사용하였다.

4.2. 실험 설정

사례연구에는 검증대상 프로그램을 그대로 사용한 버전(original version)과 실험을 위해 동시성 결함을 의도적으로 삽입한 버전(fault-seeded version)의 2가지 버전을 사용하였다. 결함을 삽입한 버전의 경우, 개별 쓰레드를 할당 받는 명령에 연관된 코드에 존재하는 동기화 명령을 의도적으로 삭제함으로써 동시성 결함을 의도적으로 발생시켰다.

검증대상 프로그램이 여러 개의 쓰레드에서 동시에 사용되는 경우를 효과적으로 테스팅 하기 위해서, 본 사례 연구에서는 2개의 쓰레드에서 검증대상 프로그램의 API를 동시에 사용하는 테스트 케이스를 작성하여 사용하였다. 각 쓰레드는 검증대상 프로그램이 제공하는 4개의 API 함수를 1회씩 순차적으로 수행하도록 하였다. 이 때 쓰레드 간의 상호작용이 발생할 가능성을 높이기 위해, 2개의 쓰레드가 동일한 계산 데이터를 읽어서 사용하도록 설정하였다. 앞서 설명한 멀테쓰레드 테스트 케이스 2개의 검증대상 프로그램 버전 각각에 대하여 10회씩 실행하며 AtomicitySanitizer를 구동하여 동적분석을 수행하였다. 이 때, 결함 검출 결과를 비교하기 위하여 데이터레이스 결함검출기인 ThreadSanitizer[6]도 동시에 실행하여 결과를 얻었다.

4.3. 분석 결과

표1은 2개의 검증대상 프로그램 버전에 대해 테스트케이스 실행 10회 동안 검증대상 코드에 대하여 발견된 결함의 개수다. 표1의 2행에서 알 수 있듯이, 검증대상 프로그램을 그대로 분석한 결과(original version), 어떠한 실행연속성결함도 발견되지 않았으며, 동시에 어떠한 데이터레이스 문제도 발견되지 않음을 알 수 있었다.

반면, 검증대상 프로그램에 의도적으로 결함을 삽입한 버전(fault-seeded version)의 경우, 총 14개의 독립적인 실행연속성위반 결함이 검출되었다. 이들을 개별적으로 분석한 결과는 다음과 같다:

- AtomicitySanitizer만 발견한 7개의 결함의 경우, 실제적인 문제를 일으킬 수 있는 문제적 요소를 올바르게 지적한 것으로 분석됨. 이들은 구체적인 결함 정보로부터 특정 쓰레드가 연속적 실행 구간(atomic code block)을 실행하는 중에 다른 쓰레드에 의해 어떻게 간섭을 받게 되는지에 대한 구체적인 정보가 제시되는 것을 확인할 수 있었음.

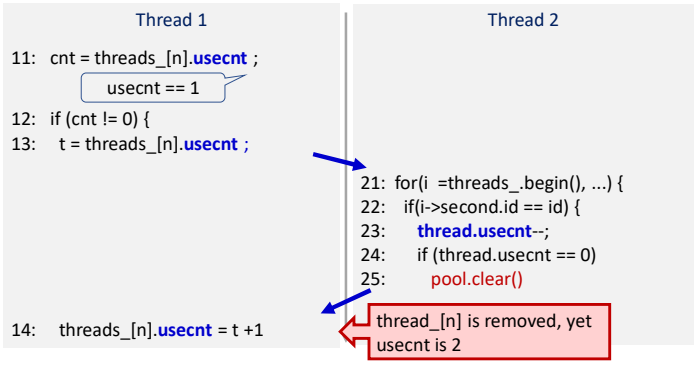

그림 3. 결함 삽입 버전에서 발견된 실행연속성위반 결함 사례

- AtomicitySantizier만 발견한 나머지 7개 결함은 오경보로 판별됨. 이들은 공유 변수에 해당하는 메모리 주소가 실행 시점에 따라 비공유 상태(non-shared)로 운용되는 경우가 있는데(예: 변수 초기화), 이를 동적 분석 모델에서 올바르게 검사하지 못하여 발생한 것으로 분석됨.
- AtomicitySanitizer와 ThreadSanitizer의 결과를 비교한 결과, AtomicitySantizier가 발견한 7개의 실제 결함 중 3개의 결함은 ThreadSanitizer가 발견한 결함과 중복이 되었으며, 나머지 4개는 AtomicitySanitizer가 고유한 결함 상황을 발견한 것이었음.

위 결과를 통해, AtomicitySanitizer가 실제 멀티쓰레드 C 프로그램에서 효과적으로 결함을 검출할 수 있으며, 기존의 ThreadSanitizer와는 독립적으로 고유한 유형의 동시성 결함을 검출함을 확인할 수 있었다. 예를 들어, 그림 3은 결함을 의도적으로 삽입한 버전에 대한 분석에서 AtomicitySanitizer만이 검출한 결함 사례다. 이 결함은 동일한 쓰레드 자원에 대해서 할당 명령(`Thread 1`)과 반환 명령(`Thread 2`)이 동시에 수행되는 상황에서 예기치 않은 쓰레드 간 간섭이 발생함으로써, 쓰레드 할당 명령에 필요한 메모리 자원이 삭제되어 프로그램 실패(`crash`)가 발생하는 오류 증상을 나타내었다. 그림 3에서 설명하는 바와 같이, 이 결함은 실행연속성 위반으로, `Thread 1`의 `11`행에서 `usecount`가 `0`이 아님을 확인한 후 해당 자원을 `14`행에서 다시 참조하기 전, `Thread 2`에서 `usecount`를 `0`으로 변경하고(`23`행) 해당 자원을 삭제(`25`행)하는 실행 패턴이 발생하여 오류증상이 발생하였다.

5. 결론

본 논문에서는 멀티쓰레드 프로그램의 실행을 관찰함으로써 실행연속성 위반 결함을 효과적인 검출하는 AtomicitySanitizer 결함검출기의 동적 분석과 구현을 설명하였다. 또한, 구현된 AtomicitySanitizer를 소프트웨어 개발 과정에 적용함으로써 해당 도구가 실제 동시성 결함 검출에 효과적임을 보인 사례연구를 소개하였다. 향후 연구에서는 여러 개의 서로 다른 실행 정보를 복합하여 실행연속성 위반 결함 검출의 정확도를 향상 시키는 새로운 동적 분석 기법과 쓰레드 스케쥴 자동 생성 기법과 AtomicitySantizer를 복합적으로 사용하여 결함 검출을 향상 시키는 방안에 대하여 연구할 계획이다. 또한 AtomicitySanitizer를 오픈소스 프로젝트로 공개할 계획이다.

참조문헌

[1] H. Sutter, J. Larus, Software and the Concurrency Revolution, Queue, 3(7), 2005

[2] P. Godefroid, N. Nagappan, Concurrency at Microsoft - An Exploratory Survey, CAV 2008 Workshop on Exploiting Concurrency Efficiently and Correctly, 2008

[3] S. Lu et al., Learning from Mistakes: a Comprehensive Study on Real World Concurrency Bug Characteristics, ASPLOS 2008

[4] S. Hong, M. Kim, A Survey of Race Bug Detection Techniques for Multithreaded Programmes, STVR, May 2015

[5] The LLVM Compiler Infrastructure Project, https://llvm.org

[6] K. Serebryany et al., Dynamic Race Detection with LLVM Compiler, RV 2011