Culry: C/C++ 소스코드 수준 기록-재현 도구

한정경 ○ 김용일 이성호
충남대학교 컴퓨터융합학부
wjdrud2532@naver.com, yongil.k01@gmail.com, eshaj@cnu.ac.kr

Culry: C/C++ Source Level Record-Replay Tool

JungKyoung Han○ Yongil Kim Sungho Lee
The Division of Computer Convergence. Chungnam National University

요 약

C/C++ 프로그램에서 비결정론적(nondeterministic) 요인으로 인해 오류 재현이 불가능한 경우, 오류의 원인 파악에 많은 시간과 자원을 필요로 한다. 본 논문에서는 기록-재현(record-replay)과 그림자 메모리(shadow memory) 기법을 사용하여 프로그램 실행 상태를 모니터링하는 도구 Culry를 제안한다. Culry는 C/C++ 소스코드를 컴파일한 LLVM 비트코드(bitcode)에 LLVM 명령(instruction)을 삽입한다. LLVM 명령이 삽입된 비트코드로부터 컴파일 된 실행파일은 명령 삽입 전과 같은 동작을 수행하며 프로그램 실행 상태를 기록한 데이터 파일을 생성한다. 이후 C/C++ 소스코드와 기록 데이터 파일을 매핑(mapping)하여 사용자에게 디버깅 기능을 제공한다. Culry는 프로그램 실행 흐름을 역방향으로 추적하거나 오류 발생 당시의 상황을 정확히 재현하여 오류 재현이 어려운 결함의 원인 파악을 돕는다.

1. 서 론

C/C++ 프로그램에서, 메모리를 직접 관리하거나 네트워크 통신, 동시성(concurrent) 같은 비결정론적(nondeterministic) 요인으로 인해 오류 재현이 어려운 경우[1], 오류를 식별하고 해결하는 데 많은 시간과 자원을 필요로 한다.

본 논문에서는 기록-재현(record-replay)과 그림자 메모리(shadow memory) 기법을 사용하여 프로그램 실행 상태를 모니터링 하는 도구 Culry를 제안한다. Culry는 C/C++ 소스코드를 컴파일한 LLVM 비트코드(bitcode)에 프로그램 실행 상태 기록을 위한 LLVM 명령(instruction)을 삽입한다. 명령이 삽입된 비트코드로부터 컴파일 된 실행파일은 명령 삽입 전과 같은 동작을 수행하며 프로그램 실행 상태를 기록한 데이터 파일을 생성한다. 이후 C/C++ 소스코드와 기록 데이터 파일을 매핑(mapping)하여 CLI(command line interface)로 사용자와 상호작용하며 디버깅 기능을 제공한다.

Culry는 프로그램 실행 흐름을 역방향으로 추적하거나 오류 발생 당시의 상황을 정확히 재현하여 오류 재현이 어려운 결함의 원인 파악을 돕는다.

1

2. 배경지식

기록-재현 기법은 프로그램의 실행 정보를 기록하여 실행 상황을 동일하게 재현하는 기법이다. 해당 기법은 단일 프로세스[2, 3], 가상환경[4], 분산 환경에 적용되어왔다[5]. 기록-재현은 원본 메모리와 동일한 구조를 갖는 가상 메모리인 그림자 메모리를 활용한다. 그림자 메모리는 재현 단계에서 프로그램의 상태를 나타내는데 활용된다.

이 논문은 2023년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(IITP-2022-0-01200)

3. C/C++ 소스코드 수준 기록-재현 도구

그림 1은 Culry의 전반적인 동작 과정을 도식화한 그림이다. Culry는 디버그 옵션으로 컴파일 된 LLVM 비트코드를 입력으로 받는다. 디버그 옵션은 LLVM 명령과 C/C++ 소스코드 간의 매핑을 위한 메타 데이터를 생성한다. 명령 삽입 단계에서, 입력으로 받은 비트코드에 프로그램 실행 정보 기록을 위한 LLVM 명령을 삽입한다. 명령이 삽입된 비트코드 파일을 컴파일 후 실행하면 원본 프로그램과 동일한 동작을 수행하며, 실행 도중에 프로그램의 상태를 외부 데이터 파일에 기록한다. 재현 단계에서, 기록 데이터 파일과 C/C++ 소스코드를 입력으로 받아, 매핑을 통해 소스코드 각 실행라인에 해당하는 그림자 메모리를 생성한다. 이후 CLI를 통해 사용자와 상호작용하며 소스코드 라인에 해당하는 그림자 메모리를 출력하는 방식으로 디버깅 기능을 제공한다.

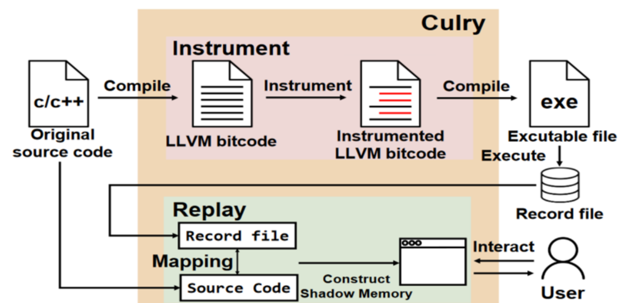

그림 1 Culry 동작 과정

3.1 LLVM 명령 삽입(Instrument)

이 과정에서는 프로그램 실행 시 수행되는 각 명령의 정보를 기록하기 위한 추가적인 LLVM 명령을 삽입한다. 기록할

정보로는 각 명령의 C/C++ 소스코드 위치와 명령의 대상이 되는 변수의 정보가 포함되며, 변수의 정보로는 이름, 타입 그리고 실행 시 결정되는 변수의 주소와 값이 포함된다. 변수의 타입이 배열(array)인 경우 차원(dimension)의 정보를 추가로 기록한다.

```
line 1.  num1 = 10;              ⟹   line 1.  store i32 10, i32* %num1, align 4, !dbg !997
line 2.  num2 = 20;                   line 2.  store i32 20, i32* %num2, align 4, !dbg !999
line 3.  num3 = num1 + num2;          line 3.  %0 = load i32, i32* %num1, align 4, !dbg !1002
                                      line 4.  %1 = load i32, i32* %num2, align 4, !dbg !1003
                                      line 5.  store i32 %add, i32* %num3, align 4, !dbg !1001
```

그림 2 C++ 소스코드와 컴파일 된 LLVM bitcode

그림 2는 int 타입 변수를 대상으로 값 할당 및 덧셈을 수행하는 C++ 소스코드와 그것을 컴파일한 비트코드를 나타낸다. 소스코드의 1, 2, 3번 라인은 각각 비트코드의 1, 2, 3~5번 라인으로 컴파일 된다. 각 명령어의 메타데이터는 해당 명령과 매핑 되는 소스코드의 라인 및 컬럼을 나타낸다. 예를 들어, !dbg !997로부터 라인 및 컬럼(1, 7) 정보를 얻을 수 있다. Culry는 비트코드 전체를 순회하며 메모리에 접근하는 명령어인 store, load를 대상으로 명령어 다음 라인에 LLVM 명령을 삽입한다.

```
line 1.  store i32 10, i32* %num1, algin 4, !dbg !997
line 2.   record("store", "num1", "i32", 10, %num1, 1, 7)
line 3.  store i32 20, i32* %num2, algin 4, !dbg !999
line 4.   record("store", "num2", "i32", 20, %num2, 2, 7)
line 5.  store i32 %add, i32* %num3, align 4, !dbg !1001
line 6.  %0 = load i32, i32* %num1, align 4, !dbg !1002
line 7.   record("load", "num1", "i32", %0, %num1, 3, 14)
line 8.  %1 = load i32, i32* %num2, align 4, !dbg !1003
line 9.   record("load", "num2", "i32", %1, %num2, 3, 21)
line 10. %add = add nsw i32 %0, %1, !dbg !1004
line 11. store i32 %add, i32* %num3, align 4, !dbg !1001
line 12.  record("store", "num3", "i32", %add, %num3, 3, 7)
```

그림 3 기록을 위한 명령이 삽입된 LLVM bitcode

그림 3은 LLVM 명령이 삽입된 비트코드를 단순화한 것이다. record()는 프로그램 실행 정보 기록을 위해 삽입한 명령을 나타내며 인자로 전달된 정보들을 외부 데이터 파일에 기록한다. 라인 2는 라인 1의 store 명령어를 대상으로 record()를 삽입한 것이다. record()의 인자로 명령어(store), 저장 대상 변수 이름(num1), 저장 대상 변수 타입(i32), 저장되는 값(10), 저장 대상 변수의 주소(%num1), 매핑 되는 소스코드의 라인 및 컬럼 정보(1, 7)를 전달한다. 명령어, 이름, 타입, 값, 주소 정보는 비트코드 명령으로부터, 라인 및 컬럼 정보는 메타데이터로부터 가져온다. 저장되는 값이 상수가 아닌 변수에 담겨 있는 경우, 라인 12와 같이 해당 변수 값을 읽어와 기록한다. 라인 7은 라인 6의 load 명령어를 대상으로 record()를 삽입한 것이다. 기록 정보는 store와 유사하나, 저장 대상 변수 대신 값을 읽어올 변수에 대한 정보(num1, i32, %num1)와 변수로부터 읽어온 값(%0)을 인자로 전달한다.

```
line 1.  %arrayidx = &arr[12][13]
line 2.  store i32 5, i32* %arrayidx, align 4, !dbg !1838
line 3.  record("store", "arr", "i32", 12, 13, 5, %arr, 50, 130)
```

그림 4 배열 대상, 기록을 위한 명령을 삽입한 LLVM bitcode

그림 4는 배열의 store 명령어에 대해 record()를 삽입한 비트코드를 단순화한 것이다. 배열 대상의 store, load 명령은 컴파일 시 배열의 특정 인덱스 주소를 가져오는 명령(라인 1)과 해당 주소에 값을 저장하는 명령(라인 2)로 분화된다. Culry는 배열에 대한 명령을 별도로 인식하고, 값을 읽거나 저장할 배열과 해당 인덱스 정보를 함께 기록하도록 명령을 추가한다. 예를 들어, 라인 3의 record()에서는 라인 1의 정보를 참조하여 라인 2 명령의 저장 대상 변수(%arrayidx)가 아닌 본래 배열의 변수(arr)와 배열의 차원 정보(12, 13)을 함께 기록한다.

그림 5는 프로그램 실행 정보를 기록한 기록 데이터 파일의 구조를 나타낸다. 기록 데이터 파일은 LLVM 명령이 삽입된 프로그램 실행 시 생성되며, 파일 내 각 라인은 명령어가 실행된 순서대로 나타난다. 그림 5의 라인 1~5와 6은 각각 그림 3과 그림 4의 record()로 기록된 결과이다. 모든 라인에서, C/C++ 소스코드 위치와 변수 정보가 기록되며 라인 6에서, 배열의 경우 차원이 추가적으로 기록되는 것을 확인할 수 있다.

```
line 1.  store num1 i32 10 0x16fafa4f8 1 7
line 2.  store num2 i32 20 0x7ff7baa4d 2 7
line 3.  load num1 i32 10 0x16fafa4f8 3 14
line 4.  load num2 i32 20 0x7ff7baa4d 3 21
line 5.  store num3 i32 30 0x16efda514 3 7
line 6.  store arr i32 12 13 5 0x16efda629 2 9
```

그림 5 기록 데이터 파일의 구조

3.2 재현(Replay)

재현 단계에서는 기록 데이터 파일에서 기록 데이터 리스트(record data list)를 생성하고, 이를 사용하여 그림자 메모리를 생성한다.

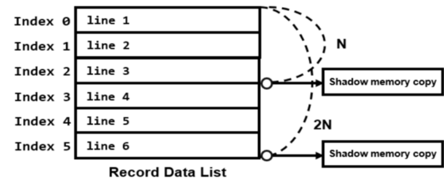

그림 6 그림자 메모리 생성 구조

그림 6은 그림자 메모리 생성 구조를 도식화한 것이다. 기록 데이터 리스트의 인덱스(index)를 기반으로 그룹화 크기(N)마다 하나의 그림자 메모리를 생성한다. 초기 그림자 메모리는 0~(N-1) 까지의 기록 데이터를 기반으로 생성되고, 이후, N개의 인덱스마다 기존 그림자 메모리에 기록 데이터를 누적 적용하여 새로운 그림자 메모리를 생성한다. 예를 들어, N이 3인 경우, 인덱스 0~2의 기록 데이터로부터 인덱스 2의 그림자 메모리가 생성되고, 인덱스 5의 그림자 메모리는 인덱스 2의 그림자 메모리에 인덱스 3~4의 기록 데이터를 적용하여 생성된다.

```
1   InitShadowMemories(recordList, N)
2       shadowMemList = []
3       shadowMem = empty
4       acc = []
5       for idx = 0 to size of  recordList do
6           recordUnit = recordList [idx]
7           acc.Add(recordUnit)
8           if idx % N == 0 then
9               shadowMem = ConstructShadowMem(shadowMem, acc)
10              shadowMemList.Add(shadowMem)
11              acc = []
12          end if
13      end for
14      return shadowMemList
```

그림 7 그림자 메모리 생성

그림 7은 그림자 메모리 생성 과정을 나타낸 의사 코드이다. 입력으로 기록 데이터 리스트와 N을 받는다. 기록 데이터 리스트의 크기만큼 리스트를 순회하면서, 직전에 생성된 그림자 메모리와 그룹화 크기(N)개의 기록 데이터를 누적하여 새로운 그림자 메모리를 생성 후 그림자 메모리 리스트에

추가한다.

그림 8는 그림자 메모리 계산 과정을 나타낸 의사 코드이다. 입력으로 사용자 입력, 기록 데이터 리스트, 그림자 메모리 리스트, N을 받는다. 사용자 입력은 디버깅 시 라인 간의 이동을 수행하며, 이동할 라인은 `CalcCurLine()`를 통해 계산된다. 해당 라인의 그림자 메모리는 가장 가까운 이전 라인의 그림자 메모리부터 해당 라인까지의 기록 데이터를 누적하여 실시간으로 만들어 제공한다. 따라서, N의 크기가 클수록 그림자 메모리 저장에 필요한 메모리 사용량은 감소하는 반면, 특정 라인의 그림자 메모리 계산에 필요한 연산은 증가한다.

```
1   CalcShadowMemory(userInput, recordList, shadowMemList, N)
2     curLine = CalcCurLine(userInput)
3     recordUnitIdx = GetRecordUnitIdx(recordList, curLine)
4     shadowMem = shadowMemList[recordUnitIdx / N]
5     uptStartIdx = recordUnitIdx - uptCnt
7     for i = 1 to recordUnitIdx % N to
8       recordUnit = recordList[uptStartIdx + i]
9       shadowMem = Apply(shadowMem, recordUnit)
10    end for
11    return shadowMem
```

그림 8 그림자 메모리 계산

4. 평 가

4.1 성능측정

본 논문에서는 Culry의 성능 평가를 위해 K framework c-semantics(https://github.com/kframework/c-semantics) 벤치마크 프로그램 및 직접 작성한, 메모리 접근이 빈번한 프로그램을 대상으로 실행 시간을 비교하였다. 측정에는 Apple M1 CPU와 8GB RAM을 탑재한 MacBook Air(2020)를 사용하였다.

Name	Native	Record	SlowR
1-unsequenced-side-effect	4	6	+50%
2-buffer-overflow	5	24	+380%
3-array-in-struct	3	5	+67%
4-buffer-underflow-external	4	5	+25%
5-buffer-overflow-environment	3	5	+67%
6-int-overflow	3	6	+100%
7-out-of-lifetime	4	5	+25%
8-int-overflow-tricky	4	6	+50%
9-memory-leak	3	4	+33%
10-add-500	6	39	+550%
11-add-5000	11	347	+3054%
12-add-10000	17	638	+3652%
13-add-20000	25	1252	+4908%

표 1 원본 프로그램과 LLVM 명령이 삽입된 프로그램 동작 시간 측정

표 1은 성능 측정 결과를 나타낸다. 1~9는 벤치마크, 10~13은 덧셈 연산을 각각 500, 5000, 10000, 20000번 수행하는 프로그램이다. Native와 Record는 각각 LLVM 명령 삽입 전, 후 프로그램의 실행시간을 밀리초(ms) 단위로 나타낸 것이다. 실행시간은 10회 측정하여 중간 값을 사용하였다. SlowR은 Native 대비 Record의 증가한 시간 비율을 기재하였다. 실험 결과, 1~9에서 최소 25%, 최대 380%, 평균 88.5%, 10~13에서 최소 550%, 최대 4908%, 평균 3041%가 증가한 것으로 나타났다. 즉, 메모리를 읽고 쓰는 작업에 비례하여 성능 저하가 큰 것을 확인했다. 반면, 메모리 접근이 적을 수록 적은 성능 저하를 보였다.

4.2 의미 보존 평가

4.1의 실험 코드를 대상으로 의미 보존 평가를 진행하였다. 각 대상 프로그램의 메모리 변화를 야기하는 구문 사이에 변화된 메모리 값을 출력하는 구문을 추가하여, 원본 프로그램과 LLVM 명령이 삽입된 프로그램의 실행 의미를 비교하였다. 실험 결과, 모든 프로그램에서 LLVM 명령 삽입 전 후의 의미가 동일함을 확인했다.

5. 향후 연구 및 결론

본 논문에서는 기록-재현 및 그림자 메모리 기법을 사용하여 C/C++ 프로그램의 실행 흐름과 상태를 모니터링하는 도구 Culry를 제안한다. Culry는 메모리 정보에 접근하는 명령어를 대상으로, 명령어의 동작을 프로그램 실행 중에 기록하는 LLVM 명령을 삽입한다. 명령이 삽입된 비트코드로부터 컴파일 된 실행파일은 프로그램 실행 상태를 기록한 데이터 파일을 생성한다. 이후 기록 데이터 파일과 C/C++ 소스코드를 매핑하여 사용자와 상호작용을 수행하며 디버깅 기능을 제공한다. 기록을 위한 LLVM 명령이 삽입된 프로그램 대상의 성능 측정 결과, 벤치마크에서는 평균 88.5%, 메모리 연산이 많은 프로그램에서는 평균 3041%의 성능 저하가 발생함을 확인했다. 또한, 의미 보존 평가를 통해 명령 삽입 후에도 프로그램의 의미가 보존됨을 확인했다.

Culry는 메타데이터가 없거나 코드가 존재하지 않는 라이브러리 함수 몸체의 경우 명령 삽입 대상에서 제외한다. 따라서, 라이브러리 함수의 실행 직후 프로그램이 비정상 종료되는 경우, 해당 시점의 그림자 메모리를 올바르게 생성할 수 없는 문제가 있다. 라이브러리 함수 호출 이후의 메모리 변화를 기록하도록 연구를 진행할 예정이다.

결과적으로, Culry가 지원하는 디버깅 기능은 프로그램 오류 원인 파악에 소모되는 개발자의 시간과 자원을 줄여 안전한 프로그램 개발에 기여할 것으로 기대한다.

참 고 문 헌

[1] Torres Lopez, Carmen, Gurdeep Singh, Robbert, Marr, Stefan, Gonzalez Boix, Elisa and Scholliers, Christophe, “Multiverse debugging: Non-deterministic debugging for non-deterministic programs”, 33rd European Conference on Object-Oriented Programming, pages 15-19, July 2019.

[2] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. “Flashback: A lightweight extension for rollback and determinsitic replay for software debugging”, In USENIX Annual Tech. Conf., June 2004.

[3] Larry D. Wittie, “Debugging distributed C programs by real time replay”, In ACM workshop on parallel and distributed debugging, pages 57-67, May 1988.

[4] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen, “Revirt: Enabling intrusion analysis through virtual-machine logging and replay”, In 5th Symp. on Op. Sys. Design and Impl. (OSDI), December 2002.

[5] Michiel Ronsse, Koen De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmuller. “Record/replay for non-determinsitic program executions”,