석사학위논문
Master's Thesis

심볼릭 배열 인덱스 참조 연산의 최적화를 통한 Concolic 테스팅의 커버리지 개선

Coverage Improvement of Concolic Testing by Optimizing Symbolic Array Index Operations

2016

김태진 (金 泰 蓁 Kim, Taejin)

한국과학기술원

Korea Advanced Institute of Science and Technology

석사학위논문

심볼릭 배열 인덱스 참조 연산의 최적화를 통한 Concolic 테스팅의 커버리지 개선

2016

김태진

한국과학기술원

전산학부

심볼릭 배열 인덱스 참조 연산의 최적화를 통한 Concolic 테스팅의 커버리지 개선

김태진

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2016년 06월 20일

심사위원장 김문주

심사위원 한태숙

심사위원 류석영

Coverage Improvement of Concolic Testing by Optimizing Symbolic Array Index Operations

Kim, Taejin

Advisor: Kim, Moonzoo

A thesis submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in School of Computing

Daejeon, Korea
June 20, 2016

Approved by

Kim, Moonzoo
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

---

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my dissertation contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS
20144372

김태진. 심볼릭 배열 인덱스 참조 연산의 최적화를 통한 Concolic 테스팅의 커버리지 개선. 전산학부. 2016년. 44+iv 쪽. 지도교수: 김문주. (한글 논문)

Kim, Taejin. Coverage Improvement of Concolic Testing by Optimizing Symbolic Array Index Operations. School of Computing. 2016. 44+iv pages. Advisor: Kim, Moonzoo. (Text in Korean)

초 록

Concolic 테스팅 기법은 심볼릭 실행과 실제 실행 값을 이용하여 테스트 입력 값을 자동으로 생성하는 기법으로 실제 산업 분야에 활발히 적용되고 있기 때문에 이 기법을 개선하여 더 높은 커버리지를 달성하는 것이 중요하다. 이 concolic 테스팅 기법에서 더 높은 커버리지를 달성을 막는 주요 원인으로는 테스팅 실행 시 발생하는 심볼릭 변수의 값 실제화가 있다. 값 실제화는 테스팅 도구가 지원하지 않는 연산이 심볼릭 실행 경로에 포함될 때 발생하는데, 올바른 다음 테스트 입력 값을 생성하지 못하기도 하므로 예상한 실행 경로를 수행하는데 실패할 수 있다.

이러한 문제점을 극복하기 위하여, 이 논문에서는 concolic 테스팅 도구 중 하나인 CREST에서 지원하지 않던 심볼릭 배열 인덱스 참조 연산을 적용하여 더 높은 커버리지를 달성한다. 이 심볼릭 배열 인덱스 참조 연산은 메모리 사용량이 크다는 문제점이 존재하는데, 이것은 연산 수행 시마다 스냅샷이라 부르는 해당 배열의 현재 심볼릭 메모리 상태 복사본이 생성되기 때문이다. 따라서 이 연산의 메모리 사용량을 줄이고자 최적화 기법 2가지를 제시하였다. 첫 번째 기법은 참조 연산 수행 시마다 생성되는 스냅샷들 중 동일한 스냅샷들을 제거하는 기법이다. 두 번째 기법은 이전에 생성된 스냅샷과 비교해 변경된 원소만을 새로운 스냅샷에 저장하는 기법이다. 실험에서 삼성 플래시 메모리의 디바이스 드라이버인 다중 섹터 읽기와 같은 4개의 실제 프로그램을 대상으로 테스팅을 수행해 기존 CREST에 비해 이 심볼릭 배열 인덱스 참조 연산 지원이 더 높은 커버리지를 달성함을 보여준다. 또한 최적화 기법들을 적용하였을 때 다중 섹터 읽기에서 메모리 사용량 및 수행 시간이 각각 기존의 8.60%, 23.66%로 감소하였고, 이를 통해 최적화 기법을 적용하였을 때 메모리 및 수행시간 측면에서 더 효율적으로 concolic 테스팅이 실행됨을 보여준다.

핵심낱말 Concolic 테스팅, 심볼릭 실행, 심볼릭 배열 인덱스 참조 연산, 소프트웨어 자동화 테스팅

Abstract

Concolic testing technique which dynamically generates test cases using symbolic execution and concrete inputs is widely used in many industry field. One of the main obstacles to improve the coverage of concolic testing is value-concretization of symbolic variables. The concretization occurs when the execution path includes operations which are not supported by concolic testing tool, and it may fail to cover expected execution path.

To overcome this limitation, I have implemented symbolic array index operations in concolic testing tool CREST to achieve high branch coverage. However, symbolic executions with symbolic array index operations consume large amount of memory. I have proposed two optimization techniques to reduce the size of memory used for the operations. The first technique is to remove redundant snapshots generated on every symbolic array index dereference operation. The second technique is to store only updated memory elements on a new snapshot compared to the previous snapshot. The experiment with four real-world programs including multi-sector read show that the CREST which support symbolic array index operations supports high branch coverage than CREST. Also, the optimization techniques consume only 8.60% of the memory use and 23.66% of the total execution time of CREST without optimizing technique on multi-sector read. The result shows that the concolic testing technique using symbolic array index operations with the optimization techniques can achieve high branch coverage with efficient memory use and execution time.

Keywords Concolic Testing, Symbolic Execution, Symbolic Array Index Operation, Automated Software Testing

차 례

표 차례

그림 차례

제 1 장 머리말

소프트웨어에 존재하는 다양한 종류의 에러들을 프로그램의 실행을 통해 찾아내고 고치는 것의 중요성은 꾸준히 제기되었다. 하지만 프로그램 실행을 위한 테스트 입력 값을 임의로 생성하거나 개발자들이 지정한 값에 대해서만 테스트를 실행하는 방법은 그 한계가 명확하고, 따라서 심볼릭 실행(symbolic execution)을 이용하여 테스트 입력 값을 자동으로 생성하는 동적 테스트 생성 기법(dynamic test generation)들이 앞선 여러 연구들에서 소개되었다[1-8].

이러한 동적 테스트 생성 기법들 중 널리 사용되는 concolic 테스팅 기법은 실제 산업 분야에서도 활발하게 사용되고 있기 때문에[9-12], 이 테스팅 기법을 개선 및 보완하는 것 또한 중요한 연구 주제이다. 이러한 테스팅 기법의 주요한 목표 중 하나는 적절한 테스트 입력 값을 생성하여 높은 커버리지를 달성하여 프로그램의 신뢰성을 높이는 것이기 때문에, 테스팅 기법을 개선하여 동일한 프로그램에 대해 더 높은 커버리지를 달성하는 것 또한 중요하다.

더 높은 커버리지를 달성하도록 concolic 테스팅 기법을 개선하는 주요한 방법으로는 테스팅 도구가 지원하지 않는 연산을 지원하여 심볼릭 변수의 값 실제화를 막는 것이 존재한다. 테스팅 도구가 지원하지 않는 연산 중에는 심볼릭 배열 인덱스 참조 연산이 있는데, 배열의 값을 참조하기 위해 사용되는 인덱스가 심볼릭 변수일 때 배열의 다른 값을 참조한 결과를 심볼릭 실행에서 저장하여 다음 입력 값을 생성할 때 사용하기 위한 연산이다. (2.2절 참조)

이 연산은 주로 문자열을 처리하거나 디스크 I/O를 처리하기 위한 코드에서 많이 사용되는데, 한 배열에 존재하는 문자나 데이터 값 중 어떤 값을 참조했느냐에 따라 프로그램 실행 경로가 달라지도록 구현될 수 있기 때문이다. 예를 들어, 삼성전자의 플래시 메모리의 디바이스 드라이버 중 하나인 다중 섹터 읽기(MSR) 프로그램은 배열의 어떤 위치에 값이 존재하는지에 따라 프로그램 실행 경로가 바뀐다[9]. 이러한 프로그램을 테스팅하기 위해서는 심볼릭 배열 인덱스 참조 연산이 필요하다.

1.1 기존 접근의 문제점

기존 연구[13]에서 정의된 심볼릭 배열 인덱스 참조 연산은 연산이 수행될 때마다 참조되는 배열의 심볼릭 메모리 전체를 복사하고 저장해 이를 다음 입력 값 생성을 위한 경로 제약조건 생성시 사용하고 있다. 하지만 한 배열의 심볼릭 메모리는 실제 값이 저장된 메모리보다 더 많은 공간을 사용할 수 있고, 매 참조 연산마다 이 심볼릭 메모리를 복사하여 저장하는 것은 여러 개의 중복되는 정보들이 따로 저장되도록 만들기도 한다. 이 문제 때문에 발생하는 메모리 낭비는 실제 심볼릭 실행에 필요한 최소한의 메모리보다 기하급수적으로 클 수 있고, 이는 대상 프로그

램이 커질 때 심볼릭 배열 인덱스 참조 연산을 사용할 수 없도록 만드는 원인이 된다. 또한, 불필요한 메모리 복사 및 저장 과정이 반복되면서 메모리의 낭비 뿐만 아니라 concolic 테스팅을 수행하는 시간이 길어지는 현상까지 불러오기도 한다. 따라서 심볼릭 배열 인덱스 참조 연산 수행 시 사용되는 메모리를 최적화하는 기법을 적용하여 메모리 및 수행 시간 효율성을 증가시킬 필요가 있다. (3.1절 참조)

1.2 제안하는 접근 방식

매 참조 연산마다 배열에 대한 심볼릭 메모리를 복사하여 저장하는 방식은 배열이 업데이트 되지 않고 참조만 여러 번 실행되었을 때 완전히 동일한 심볼릭 메모리가 2개 이상 존재하도록 만든다. 이렇게 참조 연산마다 복사되는 심볼릭 메모리를 스냅샷이라고 하는데, 이 스냅샷은 한 번 생성되면 심볼릭 경로 제약 조건과 새로운 입력 값을 생성하는 일련의 과정에서 변경 되지 않는다. 따라서 매번 참조 연산 시마다 심볼릭 메모리를 복사하는 대신 동일한 스냅샷들, 즉 하나의 배열에 대한 심볼릭 메모리의 업데이트와 업데이트 사이에 존재하는 모든 참조 연산에서 생성된 스냅샷을 제거하는 최적화 기법 1을 적용한다. (3.2절 참조)

동일한 스냅샷이 중복 생성되는 것을 최적화 기법 1을 통해 방지하였지만, 하나의 배열에 대한 심볼릭 메모리가 업데이트 될 때 배열에 존재하는 모든 원소가 변경되는 것은 아니기 때문에 새로 생성된 스냅샷과 직전에 생성된 스냅샷은 심볼릭 메모리의 원소들 중 대부분이 동일할 가능성이 높다. 기존 방식에서 각 스냅샷들은 심볼릭 메모리의 모든 원소들을 갖고 있기 때문에, 최적화 기법 2를 적용하여 각 스냅샷이 심볼릭 메모리의 모든 원소를 갖는 대신 이전 스냅샷에서 변경된 원소들만을 새로운 스냅샷에 저장하고, 심볼릭 경로 제약 조건 생성시에는 이전 스냅샷을 참조한다. 이 과정을 통해 중복되는 스냅샷이나 심볼릭 메모리의 원소들이 생성되지 않도록 하고, 심볼릭 배열 인덱스 참조 연산의 메모리 사용량을 최적화한다. (3.3절 참조)

1.3 논제 및 기여

이 논문의 논제(thesis statement)는 다음과 같다:

> Concolic 테스팅에 메모리 최적화한 심볼릭 배열 인덱스 참조 연산을 추가하면
> 더 높은 분기 커버리지를 달성할 수 있다.

또한 이 논문은 다음과 같은 기여를 한다:

- 오픈 소스 concolic 테스팅 도구인 CREST[14]에 심볼릭 배열 인덱스 참조 연산을 구현하고 메모리 사용량 최적화 기법들을 적용함.

- 심볼릭 배열 인덱스 참조 연산이 적용된 도구를 커버리지 측정 실험을 실제 프로그램에 대해 수행하여 더 높은 분기 커버리지를 달성하는데 심볼릭 배열 인덱스 참조 연산이 유효함을 보여줌.
- 메모리 최적화 기법을 적용한 도구를 실제 프로그램에 적용해 짧은 수행시간 내에 더 적은 메모리 사용량으로 (e.g. MSR의 평균 기준 각각 8.60%, 23.66%로 감소) 심볼릭 배열 인덱스 참조 연산이 수행됨을 보여줌.

1.4 논문의 전체 구성

이하에 기술되는 논문의 내용은 다음과 같이 구성되어 있다.

제 2 장에서는 concolic 테스팅과 심볼릭 배열 인덱스 참조 연산에 대한 배경을 설명한다. 먼저 concolic 테스팅의 개념과 알고리즘에 대해 설명하고 기존 concolic 테스팅의 한계점인 심볼릭 변수의 값 실제화를 예제와 함께 설명한 후 이를 해결하기 위한 심볼릭 배열 인덱스 참조 연산을 개념과 구현, 예시로 나누어 설명한다.

제 3 장에서는 기존 심볼릭 배열 인덱스 참조 연산의 문제점인 메모리 낭비를 예제를 통해 설명하고 이를 해결하기 위한 최적화 기법 1, 2에 대해 기술한다.

제 4 장에서는 심볼릭 배열 인덱스 참조 연산이 유효한지, 최적화 기법 1, 2을 적용하였을 때 메모리 사용량과 수행 시간이 얼마나 감소하는지를 예제 프로그램과 실제 프로그램들을 통해 보여준다.

제 5 장에서는 사례 연구로 실제 프로그램인 다중 섹터 읽기에 심볼릭 배열 인덱스 참조 연산과 최적화 기법들을 통해 concolic 테스팅을 수행하고 그 때의 분기 커버리지와 메모리 사용량, 수행 시간에 대해 알아본다.

제 6 장에서는 이 논문의 결론과 향후 연구에 대해 기술한다.

제 2 장 연구 배경

이 장에서는 이 연구를 수행한 배경에 대해 기술한다. 여기서는 제안한 기법의 기본이 되는 concolic 테스팅 기법과 최적화 기법이 적용되지 않은 심볼릭 메모리 참조 연산의 이론과 구현 방식에 대해 설명한다.

2.1 Concolic 테스팅 기법

Concolic (CONCrete + symbOLIC) 테스팅 기법 [15][16]은 프로그램이 일부 구체적인 입력 값에 대해 수행되는 동안, 동적인 심볼릭 실행을 동시에 수행한다. 전통적인 심볼릭 실행과 달리, concolic 테스팅은 프로그램이 실행되는 동안 전반적인 구체적 변수 상태를 유지해야 하므로, concolic 테스팅은 주어진 실제 입력 값(concrete value)을 받아 테스트 대상 프로그램을 실행하고, 실제 실행 과정에서 생성된 프로그램의 실행 경로로부터 심볼릭 경로 제약 조건을 생성한다 [25]. 하나의 심볼릭 경로 제약 조건은 특정 경로를 방문할 수 있도록 하는 입력 값을 생성할 수 있는 조건이다. Concolic 테스팅에서, 테스트 대상 프로그램 실행 시 사용된 입력 값은 이 조건식에서 실제 값 대신 심볼릭 변수로 대체 된다. 이렇게 생성된 심볼릭 경로 제약 조건의 일부에 부정 기호(negation)을 추가하는 과정을 반복하여 테스팅을 수행하는데, 이 과정은 실행 가능한 경로를 모두 실행했거나, 사용자가 지정한 테스트 케이스 개수만큼 테스트 케이스를 생성할 때까지 반복된다.

2.1.1 Concolic 테스팅 알고리즘

Concolic 테스팅을 수행하기 위한 입력에는 분석할 대상 프로그램, 종료 조건, 분석 프로그램 실행을 위한 테스트 드라이버가 있다. Concolic 테스팅의 결과로는 테스트 케이스들과 생성된 입력 값들로 실행한 테스트 실행 결과인 분기 커버리지 등이 있다. Concolic 테스팅의 수행은 일반적으로 다음과 같은 절차를 통해 진행된다.

1. 심볼릭 변수로 설정할 입력 변수들을 선택한다.
2. 심볼릭 경로 조건을 저장하기 위해 분석 대상 프로그램에 정적으로 탐침을 삽입한다.
3. 대상 프로그램 분석을 시작하기 위해 초기 입력 값을 선택하고 탐침이 삽입된 프로그램을 실행한다.
4. 탐침이 삽입된 프로그램을 실행하여 심볼릭 경로 제약 조건을 얻는다.
5. 다음 심볼릭 경로 제약 조건을 생성하기 위해 앞서 얻은 제약 조건의 분기 중 하나를 부정(negation)한다.

6. 앞서 얻은 경로 제약조건을 부정하여 생성한 새로운 경로 제약조건을 만족하는 새로운 입력 값을 찾는다.
7. 주어진 종료 조건을 만족하면 concolic 테스팅을 종료하고, 그렇지 않으면 3번으로 돌아가 새로운 입력 값을 생성하는 과정을 반복한다.

1번째 단계에서 사용자는 어떤 입력 값을 심볼릭 변수로 선언할 지 결정해야 한다. 사용자는 concolic 테스팅 도구에서 제공하는 API 중 심볼릭 변수 선언 함수를 통해 입력 값을 심볼릭 변수로 만들 수 있고, 이 과정은 분석할 대상 프로그램 실행 직전이나 테스트 드라이버 함수에서 수행한다.

2번째 단계에서 concolic 테스팅 도구는 대상 프로그램이 실행될 때 지나가는 경로와 이 경로를 수행하며 생성되는 경로 조건을 얻기 위해 테스트 대상 프로그램과 테스트 드라이버에 탐침을 삽입한다. 탐침은 프로그램의 각 문장을 분석하기 위해 정적으로 삽입된 외부 함수이다.

3번째 단계에서 concolic 테스팅 도구는 앞서 변경된 테스트 드라이버와 대상 프로그램을 실행하게 되는데, 입력 값을 지정해준 경우 해당 입력 값으로 대상 프로그램이 실행되고, 그렇지 않은 경우 랜덤하게 선택된 입력 값으로 프로그램을 실행한다. 또한 앞서 대상 프로그램을 실행하여 새로운 입력 값을 얻었다면 그 값을 사용해 프로그램을 실행한다.

4번째 단계에서 삽입된 탐침들이 호출됨에 따라 심볼릭 경로 제약 조건이 생성된다. 탐침은 대상 프로그램을 동적으로 분석하면서 각 표현식과 분기문 정보를 수집해 심볼릭 경로 제약 조건을 생성하는데, 이 프로그램 실행을 통한 조건 생성을 심볼릭 실행(symbolic execution)이라고 한다.

5번째 단계에서는 4번째 단계에서 얻은 심볼릭 경로 제약 조건 중 하나의 분기를 부정한다. 만약 부정할 수 있는 분기가 여러 개인 경우, 어떤 분기를 부정할 지를 결정하기 위해 사용되는 탐색 전략에 따라 하나를 선택해 부정한다. 예를 들어, 깊이 우선 탐색(DFS) 전략은 생성된 분기의 마지막에 있는 분기를 부정한다.

6번째 단계에서는 SMT solver[17]를 이용해 5번째 단계에서 부정한 경로 제약 조건을 풀고 새로운 입력 값을 찾는다. SMT solver에는 Z3[18], Yices[19], MathSAT[20]와 같은 툴이 존재한다. 만약 SMT 실행 결과가 UNSAT(만족하는 해가 없음)인 경우, 5번으로 돌아가 다른 분기를 부정하고 다시 6번째 단계를 수행한다.

7번째 단계에서는 종료 조건을 만족했는지 확인하는데, 이는 유저가 지정한 조건 (e.g. 최대 생성 테스트 케이스의 수, 최대 수행 시간, 분기 커버리지 등)을 만족하는 것을 말하고, 조건이 만족되지 않았다면 계속해서 테스팅을 수행하게 된다.

```
01:  #include <crest.h>
02:  int main(){
03:      int x, y;
04:      CREST_int(x); CREST_int(y);
05:      testme(x, y);
06:  }
07:  void testme(int x, int y) {
08:      int z = 2*y;
09:      if (z == x){
10:          if(x > y+5)
11:              Error();
12:      }
13:  }
```

그림 2.1 Concolic 테스팅을 설명하기 위한 프로그램 예시

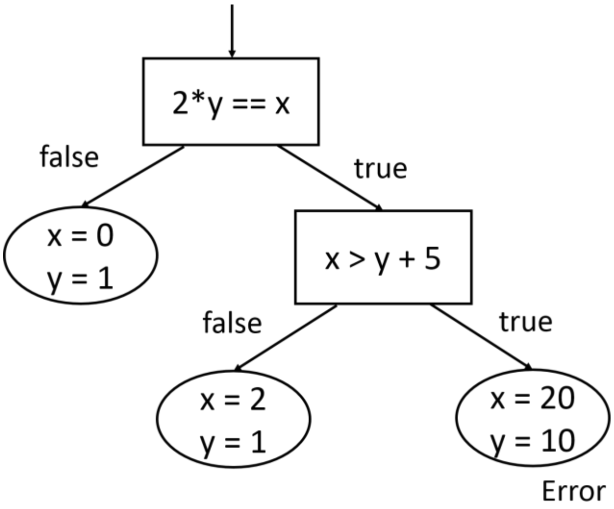

그림 2.2 testme() 함수의 실행 트리

2.1.2 Concolic 테스팅 실행 예시

그림 2.1의 예제에서 concolic 테스팅은 임의의 입력 값으로 {x=0, y=1}을 생성한다고 가정하였을 때, testme() 함수에 대해 구체적 실행과 심볼릭 실행을 동시에 수행한다. 이 때, 구체적 실행은 9번째 줄에서 ‘else’ 분기를 선택하게 된다. 동시에 심볼릭 실행은 x!=2*y의 경로 조건을 생성한다. Concolic 테스팅은 경로 조건에 접합된 부분 (conjunct)을 부정한 (negate), x=2*y을 해석하여 테스트 입력 값 {x=2, y=1}을 얻는다.

이렇게 생성된 새로운 입력 값은 프로그램이 다른 실행 경로를 따라 실행되도록 만든다. Concolic 테스팅은 새롭게 생성된 입력 값에 대해 구체적 실행과 심볼릭 실행을 되풀이하여 기존의 실행 경로와 다른 경로를 선택하게 된다. 즉, 그림 1의 9번째 줄 분기문에서 이전과 달리 ‘then’ 갈래를 선택하게 된다. 이전 단계의 실행처럼, concolic 테스팅은 구체적 실행과 더불어 심볼릭 실행을 수행하고 경로 조건으로 {x=2*y}∧{x<=y+5}을 생성한다. 그 후 concolic 테스팅은

```
01:  #include <crest.h>
02:  int main(){
03:      int a[4] = {1,2,3,4};
04:      CREST_int(x);
05:      if (a[x] == 2){
06:         Error();
07:      }
08:      return 0;
09:  }
```

그림 2.3 심볼릭 변수의 값 실제화 문제 설명을 위한 예시

이전에 실행되지 않은 실행 경로를 따라 프로그램이 수행되도록 새로운 테스트 입력 값을 생성해 낼 것이다. 이는 경로 조건의 끝에 접합된 {x<=y+5}을 부정한 결과, 즉, {x=2*y}∧{x>y+5}을 해석하여, 새로운 테스트 입력 값 {x=20, y=10}를 구함으로써 이루어진다. 새로운 테스트 입력 값은 10번째 줄의 ERROR 문에 도달하도록 한다. 그 후 concolic 테스팅은 프로그램의 모든 실행 경로가 검색되었음을 알리고, 테스트 입력 값 생성을 종료한다. 이렇게 실행된 testme() 함수의 실행 트리는 그림 2.1와 같다.

2.1.3 Concolic 테스팅의 한계점

- 심볼릭 변수의 값 실제화의 문제점

심볼릭 변수의 값 실제화는 다양한 경로를 실행시키고 분기 커버리지를 높이는데 있어 한계점이 될 수 있다. Concolic 테스팅에서는 실제 값으로 대상 프로그램을 실행하기 때문에 외부 환경 변수가 있거나 도구가 지원하지 않는 복잡한 연산이 있거나, 소스 코드가 존재하지 않는 함수를 실행한 경우 심볼릭 변수를 실제 값으로 대체하는 것으로 불완전한 경로 제약 조건을 생성한다. 이 경우 생성된 경로 제약 조건은 실제 값이 포함되어 있기 때문에 분기 조건에 심볼릭 변수가 없다고 판단하여 부정을 수행한 제약조건을 생성하지 않고, 이 때문에 실제 대상 프로그램에서는 도달할 수 있으나 concolic 테스팅 도구에서는 도달하지 못한다고 결과를 내기도 한다.

- CREST에서 심볼릭 변수의 값 실제화의 문제 예시

그림 2.3은 Concolic 테스팅 도구 중 하나인 CREST에서 심볼릭 변수의 값 실제화로 인해 커버 가능한 분기 커버에 실패하는 프로그램의 예시이다. 심볼릭 변수 x의 최초 입력 값이 0이라고 하자. 5번째 줄을 수행할 때 'else' 분기를 실행하고 a[x] != 2라는 경로 조건이 생성된다. 하지만 CREST는 a[x]라는 심볼릭 배열 참조 연산을 지원하지 않기 때문에 a[x] 대신 0으로 값 실제화를 수행하고, 생성된 심볼릭 경로 제약 조건 0 != 2는 부정을 수행하여 0 == 2가 되어도 심볼릭 변수가 포함되지 않았기 때문에 UNSAT이 나오게 된다. 때문에 가능한 모든 경로를 수행했다고 보고하고 종료하게 된다. 심볼릭 배열 참조 연산으로 인한 문제를 해결하려 한 다른 테스팅 도구

```
01:  #include <crest.h>
02:  int main(){
03:      int x, y;
04:      CREST_int(x); CREST_int(y);
05:      foo(x, y);
06:  }
07:  void foo(int x, int y) {
08:      int a[4];
09:      a[0] = x; a[1] = 0;
10:      a[2] = 1; a[3] = 2;
11:      if(a[x] == a[y] + 2){
12:         Error();
13:      }
14:  }
```

그림 2.4 심볼릭 배열 인덱스 참조 연산의 필요성 설명을 위한 예시

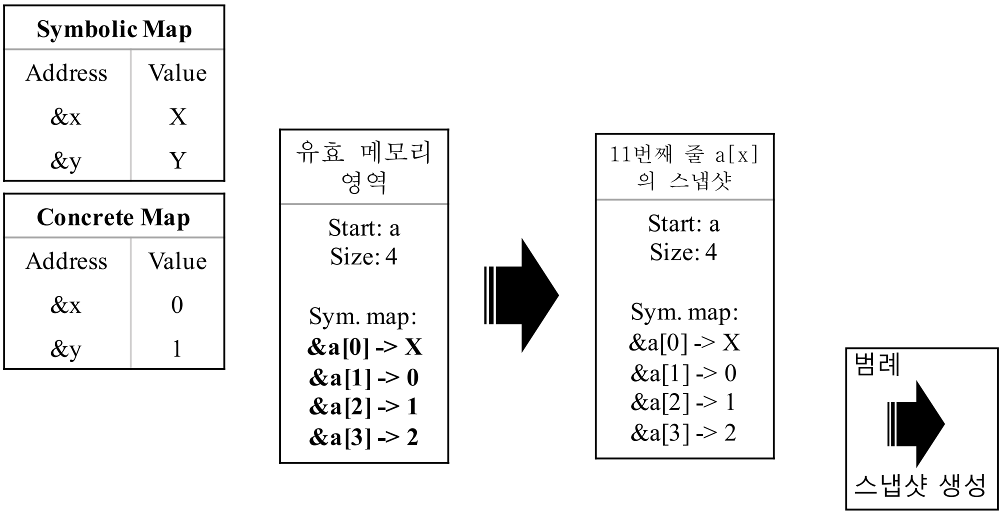

그림 2.5 그림 2.4 11번째 줄에서의 유효 메모리 영역과 스냅샷 예시

들에 대해 2.3절에서 비교한다.

2.2 심볼릭 배열 인덱스 참조 연산

심볼릭 배열 인덱스는 배열을 참조하기 위한 인덱스가 실제 값이 아닌 심볼릭 입력에 종속된 인덱스를 말한다. 이 심볼릭 배열 인덱스의 참조 연산이란 배열 내부의 값을 얻기 위해 심볼릭 배열 인덱스가 배열의 인덱스로 사용되어 참조(dereference, deref)가 일어나는 경우를 말한다.

2.2.1 심볼릭 배열 인덱스 참조 연산의 필요성

그림 2.4는 심볼릭 배열 인덱스 참조 연산을 지원하지 않아 심볼릭 변수가 실제 값으로 대체됨에 따라 발생하는 문제점을 설명하기 위한 예제 코드이다. 이 예제 코드에서는 foo 함수를 테

스트하기 위해 concolic 테스팅 도구인 CREST가 실행되도록 테스트 드라이버인 main 함수가 작성되어 있고, foo 함수의 파라메터인 x, y가 심볼릭 변수로 세팅되어 있다.

이 프로그램을 테스트하기 위해 최초 실행할 때 입력 값이 {x=0, y=1}이라고 가정하자. 탐침이 삽입된 프로그램이 수행됨에 따라 CREST는 11번째 줄에 대한 경로 제약 조건을 생성할 것이다. 이 때, 심볼릭 배열 인덱스 참조 연산이 지원되지 않으므로 a[x]와 a[y]의 인덱스 x와 y는 실제 값 0과 1로 대체된다. 따라서 11번째 줄에 대한 경로 제약 조건은 a[0] == a[1] + 2가 되고, a[0]와 a[1]은 각각 심볼릭 변수 x와 상수 0이기 때문에 x == 0 + 2라는 조건이 생성된다. 첫 번째 프로그램 수행 시 x == 0 + 2가 거짓이었으므로 이 조건을 참으로 하는 값인 {x=2, y=1}이 SMT solver를 통해 생성되고, 다음 입력 값으로 사용된다. 하지만 이 입력 값에 대한 11번째 줄의 조건은 1 == 0 + 2로 여전히 12번째 줄을 실행하지 못한다. 12번째 줄을 실행하기 위해서는 심볼릭 배열 인덱스 참조 연산을 통해 a[x]가 나타낼 수 있는 값이 인덱스 x의 값에 따라 x, 0, 1, 2가 가능하고, a[y]가 나타낼 수 있는 값은 인덱스 y의 값에 따라 x, 0, 1, 2가 가능하다는 정보가 필요하다.

이와 같이 심볼릭 배열 인덱스 참조 연산을 지원하지 않는 경우 실제로 가능한 프로그램의 실행 경로를 수행하지 못해 발생 가능한 버그를 놓칠 수 있고, 동일한 실행 경로를 수행하는 테스트 케이스를 중복 생성하게 된다.

2.2.2 심볼릭 배열 인덱스 참조 연산을 위한 메모리 모델

- 메모리 모델의 정의

메모리 모델은 심볼릭 배열 인덱스 참조 연산을 구현하기 위해 각 배열의 심볼릭 메모리를 기존의 심볼릭 메모리와 별도의 메모리로 구분하여 인덱스가 배열 내의 유효한 영역만을 가리킬 수 있도록 정의한 것으로 메모리 모델은 다음과 같이 구성되어 있다[13].

유효 메모리 영역(valid memory region): 한 배열의 시작점과 그 배열의 크기, 그리고 배열 내부를 참조할 수 있는 각각의 인덱스 값과 그 인덱스로 배열을 참조한 결과를 맵으로 나타낸 것

스냅샷(snapshot): 참조 연산 수행시 참조되는 배열의 유효 메모리 영역의 상태를 복사한 것

- 심볼릭 배열 인덱스 참조 연산의 구성 요소

심볼릭 배열 인덱스 참조 연산은 2가지로 구분할 수 있다. 첫번째는 기존 예제에서 설명된 것과 같이 분기문의 조건이나 할당문의 오른쪽 등에 나타나는 참조 연산(deref)이다. 심볼릭 인덱스로 배열을 참조 시에는 인덱스의 값에 따라 다른 참조 값이 나타나는데, 이 정보를 심볼릭 경로 제약 조건에 추가해야 한다. 두번째는 인덱스가 심볼릭이고 배열을 참조하였을 때 참조된 위치에 값을 할당하는 쓰기(write) 연산이다. 쓰기 연산은 심볼릭 인덱스의 값에 따라 할당 하고자 하는 값이 할당되는 위치가 달라지므로, 이 정보까지 심볼릭 경로 제약 조건에 추가할 수 있

어야 concolic 테스팅 도구에서 심볼릭 배열 인덱스 참조 연산을 지원한다고 할 수 있다.

이러한 심볼릭 경로 제약 조건을 생성하기 위해서는 먼저 심볼릭 인덱스로 배열을 참조할 때 인덱스가 가질 수 있는 값의 범위를 알아야 한다. 인덱스가 가질 수 있는 실제 값은 인덱스 변수가 가질 수 있는 값 전체이지만, 심볼릭 인덱스로 배열을 참조하였을 때 세그먼테이션(segmentation) 결함이 나타나는 값보다 실제로 유효한 결과를 나타낼 수 있는 값으로 제한하여 심볼릭 배열 인덱스 참조 연산을 실용적으로 수행할 수 있도록 한다. 세그먼테이션 결함이 나타나는 경우를 concolic 테스팅을 통해 찾기 위해서는 심볼릭 배열 인덱스 참조 연산을 사용할 필요 없이 참조 연산이 발생하는 구문 직전에 assert 함수로 배열이 선언된 크기를 인덱스의 값이 초과했는지 여부를 체크함으로써 실현 가능하다.

따라서 심볼릭 인덱스가 배열을 참조하였을 때 유효한 결과를 나타낼 수 있는 값의 범위와 그 인덱스 값으로 배열을 참조한 결과를 나타낸 유효 메모리 영역이 필요하다. 좀 더 구체적으로는, 배열이 시작하는 지점의 주소 값(start)과 해당 배열의 크기(size), 그리고 각 인덱스를 참조하였을 때의 심볼릭 값을 매핑(mapping)한 심볼릭 맵(symbolic map)으로 구성된다.

또한 배열의 참조 값은 프로그램이 실행됨에 따라 계속해서 변경될 수 있지만 심볼릭 인덱스로 참조 연산을 수행할 때 필요한 배열의 메모리 상태는 참조 연산이 일어나는 시점의 상태이기 때문에 경로 제약 조건을 생성하기 위해 대상 프로그램 실행 중 탐침에 의해 생성되는 심볼릭 표현식에서 참조 연산은 참조 연산 수행 시 참조되는 배열의 유효 메모리 영역을 복사한 스냅샷을 가지고 있어야 한다.

그림 2.5는 그림 2.4의 11번째 줄 a[x]에서 참조 연산을 수행할 때의 유효 메모리 영역과 참조 연산 수행을 통해 생성된 스냅샷을 나타낸 것이다. 심볼릭 맵에서 굵은 글씨로 써진 원소들은 업데이트 된 원소를 말한다. 유효 메모리 영역이 8번째 줄의 배열 선언문에 의해 생성되고, 9번째 줄과 10번째 줄에 의해 유효 메모리 영역의 심볼릭 맵이 업데이트 되어 그림에 존재하는 상태가 된다. 11번째 줄의 a[x] 참조 연산이 발생할 때 현재 유효 메모리 영역의 상태를 그대로 복사하여 스냅샷을 생성하고, 심볼릭 경로 제약 조건의 참조 연산이 이 스냅샷을 갖는(has a)다.

2.2.3 메모리 모델의 구현

심볼릭 배열 인덱스 연산을 위한 메모리 모델을 CREST-BV[11]를 기반으로 구현한다. 그림 2.6은 CREST-DEREF의 구조를 도식화한 것으로 CREST-BV의 구조와 유사하다. CREST-BV는 분석 대상 프로그램에 탐침을 삽입해주는 프론트엔드(front-end), 프로그램이 실행됨에 따라 심볼릭 경로 제약 조건을 생성해주는 미들엔드(middle-end), 심볼릭 경로 제약 조건을 SMT solver가 풀고 다음 입력을 생성할 수 있도록 SMT formula로 만들어주는 백엔드(back-end)가 존재한다.

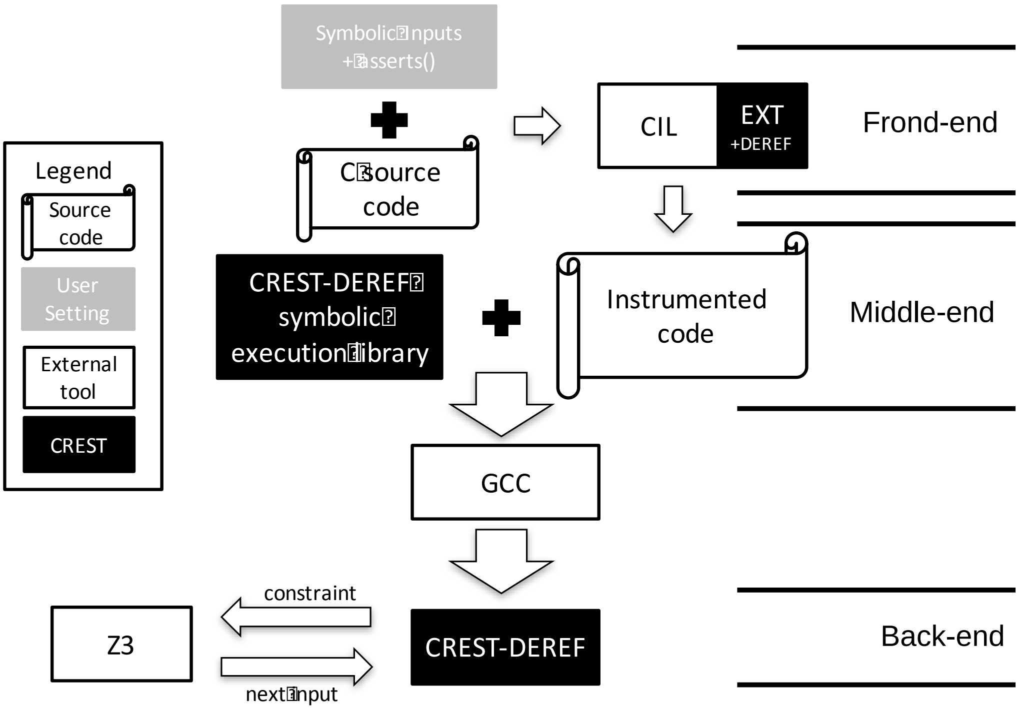

그림 2.6 CREST-DEREF의 구조도

프론트엔드는 CIL (C Intermediate Language)[21][22]을 이용해 구현되며, 프로그램 실행시 정보를 얻을 수 있도록 탐침을 삽입한다. 기존 CREST-BV의 프론트엔드에서는 심볼릭 인덱스로 배열을 참조하여 값을 읽거나 저장하는 것과 배열이 아닌 심볼릭 변수에서 값을 읽거나 저장하는 것의 차이가 없다. 하지만 인덱스가 심볼릭 변수로 값이 바뀔수 있도록 변경되기 때문에 배열을 참조하여 값을 읽는 것을 참조(deref), 쓰는 것을 쓰기(write)로 탐침을 변경해 삽입한다. 이 참조와 쓰기 탐침에서는 전체 심볼릭 맵에 있는 값을 읽거나 쓰는 대신 해당 배열의 유효 메모리 영역에 존재하는 심볼릭 맵을 사용한다. 또한 한 메모리를 심볼릭 배열 인덱스 연산을 수행할 대상으로 만들기 위해 정적으로 배열이 선언된 시점이나 동적으로 메모리 할당을 실행할 때 이 메모리에 대한 유효 메모리 영역을 생성할 수 있도록 탐침을 삽입한다.

미들엔드는 프로그램에 삽입된 탐침이 실행됨에 따라 동적으로 심볼릭 경로 제약 조건을 생성하는데, 참조 또는 쓰기 연산이 실행된 경우 유효 메모리 영역을 계속해서 업데이트 한다. 이 유효 메모리 영역은 유효 메모리 영역 생성을 위해 프론트엔드에서 삽입한 탐침 내에서 생성된다. 정적으로 선언되어 스택에 저장되는 배열에 대한 유효 메모리 영역은 배열이 선언된 함수가 종료될 때 소멸된다. 또한 동적으로 생성되어 힙에 저장되는 배열은 메모리 해제 함수를 호출하거나 전체 프로그램이 종료되기 직전에 소멸되며, 그 이전까지는 쓰기 연산에서 해당 유효 메모리 영역이 업데이트 되거나, 참조 연산에서 스냅샷을 생성한다.

백엔드에서는 생성된 심볼릭 경로 제약 조건을 SMT 표현식(formula)으로 만들고 SMT solving

을 진행하는데, 메모리 모델을 SMT 표현식으로 만들기 위해서 CREST-BV가 사용하는 SMT solver인 Z3에 존재하는 배열 이론(array theory)[23]을 사용한다. 배열 이론은 저장(store)과 선택(select) 함수로 구성되어 있는데, 하나의 배열에 인덱스와 값의 쌍을 저장하는 것을 저장이라 하고, 배열에 저장되어 있는 인덱스와 값의 쌍 중 하나의 인덱스를 통해 값을 꺼내는 것을 선택이라 한다. 저장과 선택을 논리식으로 표기하는 방법은 다음과 같다:

저장(store): 배열 array에서 i번째 index에 값 e를 쓰는 것을 array{i <- e}라 정의한다.

선택(select): 배열 array에서 i번째 index에 존재하는 값을 읽는 것을 array[i]라 정의한다.

또한, 배열을 참조하는 심볼릭 인덱스의 값이 배열이 선언된 범위를 벗어나면 안되기 때문에, SMT 표현식에 start address <= start address + index < start address + size 조건을 논리곱(conjunction)한다. 이렇게 생성된 표현식을 SMT solver가 풀었을 때 UNSAT이 아니면 새로운 입력 값을 받아 그 값으로 프로그램을 다시 실행한다.

위에서 설명한 구현 방식과 논리 표기 방식을 통해 그림 2.4의 예시 프로그램에 심볼릭 배열 인덱스 참조 연산이 적용 되었을 때 생성되는 조건식을 다음 절에서 설명한다.

2.2.4 심볼릭 배열 인덱스 참조 연산의 적용 예시

앞서 심볼릭 배열 인덱스 참조 연산의 필요성을 설명하기 위해 사용된 그림 2.4를 통해 이 연산이 적용된 예시를 설명한다. 앞선 예시와 동일하게 concolic 테스팅에서 최초 프로그램 실행 시 입력 값을 {x=0, y=1}이라 가정하자. 탐침이 삽입된 프로그램이 수행됨에 따라 CREST는 11번째 줄에 대한 경로 제약 조건을 생성한다. 이 때, 심볼릭 배열 인덱스 참조 연산이 지원되기 때문에 a[x]와 a[y]의 인덱스 x와 y가 실제 값으로 대체되지 않고 심볼릭 경로 제약 조건을 생성한다. x와 y는 int a[4]로 선언된 배열 a를 참조하였기 때문에 가질 수 있는 값은 각각 a <= x < a + 4, a <= y < a + 4이다. 또한 참조 시점의 배열의 상태를 배열 이론으로 표현하면 a{0 <- x}{1 <- 0}{2 <- 1}{3 <- 2}가 된다. 그 다음 분기문에 대한 조건 a[x] == a[y] + 2가 추가되고, 이를 모두 논리곱(conjunction)하여 제약조건을 생성한다.

이 조건들로 구성된 경로 제약 조건을 SMT solver로 풀면, {x=3, y=1}이 결과 값으로 나오고 이는 Error()가 포함된 12번째 줄을 방문하는 실행경로를 출력하는 입력 값으로 성공적으로 대상 프로그램에 존재하는 모든 구문을 수행하고 검증할 수 있다.

표 2.1 Concolic 테스팅 도구들의 심볼릭 배열 인덱스 참조 연산 적용 여부 비교

	연산 지원 여부	테스팅 대상 언어	Open Source	Public Access
CUTE	지원안함	C	X	X
DART	지원안함	C	X	X
EGT	지원안함	C	X	X
CREST	지원안함	C	O	O
jCUTE	지원안함	Java	O	O
EXE	부분적 지원	C	X	X
KLEE	지원?	C	O	O
PEX	지원?	.Net	X	O
SAGE'	지원	x86 binary	X	X
CREST-DEREF	지원	C	X	X

2.3 관련 연구

표 2.1은 다른 동적 테스팅 도구들이 심볼릭 배열 인덱스 참조 연산을 어떻게 다루었는지, 그리고 각각이 어떤 테스팅 대상 언어를 사용하고 공개되고 있는지를 비교 설명하기 위한 표이다. CUTE[1], DART[2], CREST[14]는 C 프로그램 언어를 대상으로 concolic 테스팅을 수행하기 위한 도구이다. DART는 랜덤하게 생성된 입력 값으로 테스트를 수행하고, 이를 통해 심볼릭 제약식을 생성한다. DART는 배열을 인덱스로 참조할 때 인덱스 값을 실제화하기 때문에 심볼릭 배열 인덱스 참조 연산을 전혀 지원하지 않는다.

CUTE에서는 배열 이론을 이용해 심볼릭 배열 인덱스 참조 연산을 심볼릭 실행으로 구현하는 경우 경로 제약 조건을 해결하는 비용이 너무 커지기 때문에 심볼릭하게 지정하지 않고, 대신 배열 내부의 각각의 값만을 심볼릭 변수로 둔다. CUTE는 현재 더 이상 public access가 불가능하다. jCUTE[4]는 java 프로그램을 대상으로 concolic 테스팅을 수행하기 위해 CUTE를 재 구현 한 것으로, 심볼릭 배열 인덱스 참조 연산은 CUTE와 동일하게 지원하지 않지만 오픈 소스로 소스가 공개되어 있다.

EXE[5]의 전신이 되는 concolic 테스팅 도구인 EGT[3] 역시 심볼릭 배열 인덱스 참조 연산을 지원하지 않는다. 반면 EXE는 1차원 배열에 대해 심볼릭 배열 인덱스 참조 연산을 지원하고 있다. 또한, 심볼릭 배열 인덱스 참조 연산 사용시 경로 제약 조건을 해결하는 시간이 길어지는데, 이 문제를 해결하기 위해 SAT solver인 STP[24]에서 제약 조건 개선 기법을 수행하였다. 하지만 다차원 배열에 대해 심볼릭 배열 인덱스 참조 연산을 지원하지 못하는 것은 한계점으로 남아있

다.

KLEE[6]는 심볼릭 배열 인덱스 참조 연산을 지원하고 있지만, 쓰기 연산이 포함된 프로그램의 경우 테스트 케이스 수가 기하급수적으로 늘어나면서 프로그램 검증이 어려워지기도 한다. 예를 들어, 제 5 장 실험에서 사용하는 Sort (5, 5)예제 프로그램의 경우 KLEE 실행 시 24시간 이상이 소요되어 테스팅을 수행할 수 없었다.

PEX[7]는 Microsoft에서 개발하고 .Net을 대상으로 하는 concolic 테스팅 도구로, 심볼릭 배열 인덱스 참조 연산의 참조(dereference)는 지원하지만, 쓰기 연산을 지원하는지 여부는 기술되지 않았다. 이외에 concolic 테스팅 도구들을 비교 분석한 논문들[25][26]을 통해 오픈 소스이고 널리 사용되는 concolic 테스팅 도구인 CREST를 확장하여 심볼릭 메모리 인덱스 참조 연산을 사용하는 것이 적합하다고 판단하였다.

SAGE’[13]는 심볼릭 배열 인덱스 참조 연산이 구현된 SAGE[8]를 말하는 것으로, 이 논문에서 concolic 테스팅을 위한 심볼릭 배열 인덱스 참조 연산을 처음 제시하였다. 하지만 참조 연산에 대해 최적화를 진행하지 않아 메모리 사용량이 기하급수적으로 늘어날 수 있고, 오픈 소스가 아니어서 실제 산업에 적용하여 사용할 수도 없다.

- 실제 프로그램에 심볼릭 배열 인덱스 참조 연산을 적용한 연구들

실제 프로그램을 대상으로 concolic 테스팅을 수행할 때 이러한 심볼릭 배열 인덱스 참조 연산의 필요성이 대두되는데, M. Kim의 논문[9]에서는 심볼릭 배열 인덱스 참조 연산을 지원하지 않아 concolic 테스팅을 위한 심볼릭 변수 설정 시 switch-case문으로 모든 가능한 값을 전개하여 MSR의 테스팅을 수행하였다.

B. Elkarablieh의 논문[13]에서는 C++ packet decoder에서 발생하는 버그를 심볼릭 배열 인덱스 참조 연산을 통해 찾았으며, J. Yang의 논문[27]에서도 EXE에서 구현된 심볼릭 배열 인덱스 참조 연산을 이용해 리눅스 파일 시스템 중 하나인 EXT2(second extended filesystem)에서 발생하는 버그를 검출하였다. 이와 같이 심볼릭 배열 인덱스 참조 연산 없이는 테스팅 수행을 위한 심볼릭 설정 과정이 상대적으로 복잡해지고, 실제 프로그램에서 발생하는 버그 중 일부를 심볼릭 배열 인덱스 참조 연산을 통해서 잡을 수 있기 때문에 이 연산을 CREST에서 지원하여 실제 프로그램에 적용하는 것이 중요하다.

```
01:  #include <crest.h>
02:  int main(){
03:     int x, y;
04:     CREST_int(x); CREST_int(y);
05:     bar(x, y);
06:  }
07:  void bar(int x, int y) {
08:     int a[4] = {0};
09:     a[0] = x; a[1] = 0;
10:     a[2] = 0; a[3] = 2;
11:     if(a[x] == a[y] + 2){
12:        a[2] = 2;
13:        if(a[x] == a[y])
14:           Error()
15:     }
16:  }
```

그림 3.1 심볼릭 배열 인덱스 참조 연산의 문제점 설명을 위한 예시

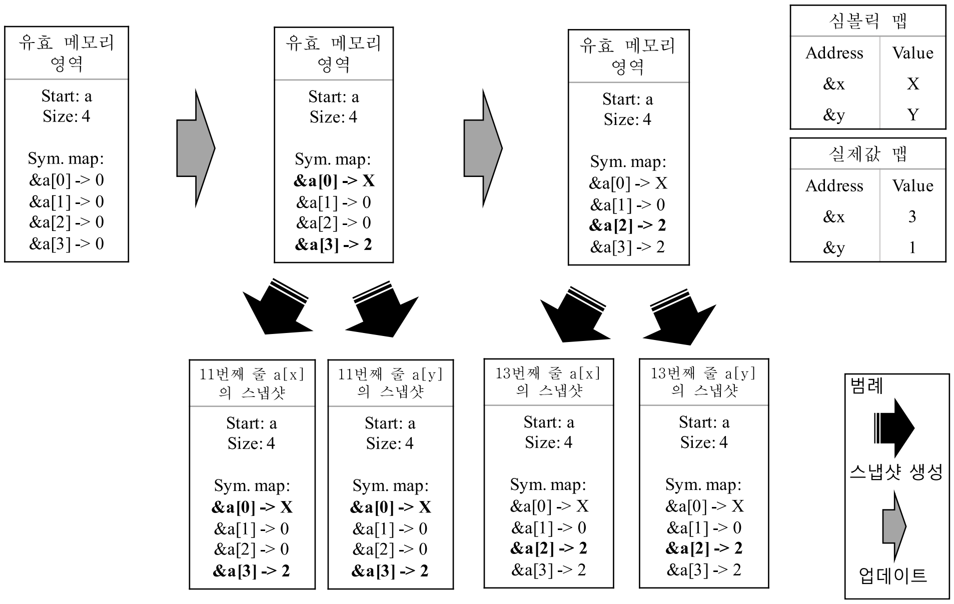

그림 3.2 기존 심볼릭 배열 참조 연산의 스냅샷 생성 과정

제 3 장 심볼릭 배열 인덱스 참조 연산의 최적화

3.1 기존 배열 인덱스 참조 연산의 문제점

앞서 설명된 심볼릭 배열 인덱스 참조 연산의 메모리 모델은 메모리 사용량이 지속적으로 증가한다는 문제점이 존재한다. 이는 메모리 모델이 참조와 쓰기 연산이 여러 번 수행되더라도 올바른 심볼릭 경로 제약 조건을 생성해주고 새로운 입력 값을 만들 수 있게 해주지만, 심볼릭 배열에 접근하는 참조 연산을 수행할 때마다 매번 스냅샷을 생성하기 때문이다.

```
01:  #include <crest.h>
02:  int main(){
03:      int x, y;
04:      CREST_int(x); CREST_int(y);
05:      bar_exp(x, y);
06:  }
07:  void bar_exp(int x, int y) {
08:      int a[4] = {0};
09:      a[0] = x;
10:      a[1] = 0;
11:      a[2] = a[x];
12:      a[3] = a[x];
13:      if(a[x] == a[y] + 2)
14:          Error()
15:  }
```

그림 3.3 쓰기 연산으로 스냅샷 생성이 많아지는 예시 프로그램

그림 3.1은 심볼릭 배열 참조 연산이 사용되는 프로그램을 테스트하기 위해 테스트 드라이버와 심볼릭 변수 설정을 한 예시 프로그램이다. 이 프로그램을 입력 값 {x=3, y=1}로 concolic 테스팅을 시작했다고 가정하자. 이 코드에서는 참조 연산이 11번째 줄과 13번째 줄에서 모두 4번 발생하고 9 ~ 10번째 줄과 12번째 줄에서 해당 배열 a를 쓰기 연산으로 업데이트한다. 참조 연산이 4번 발생하였기 때문에 스냅샷 또한 4개가 생성되고, 이때의 유효 메모리 영역의 상태 변화 과정과 각 스냅샷들의 상태는 그림 3.2와 같다. 11번째 줄에서 참조 연산에 의해 생성되는 2개의 스냅샷은 두 연산 사이에 배열이 업데이트 되지 않았기 때문에 완전히 동일한 심볼릭 맵을 각각 가지고 있다. 또한, 13번째 줄에서 참조 연산에 의해 생성되는 2개의 스냅샷도 12번째 줄의 쓰기 연산에 의해 11번째 줄의 스냅샷과는 심볼릭 맵의 형태가 다르지만 두 개의 스냅샷은 동일한 심볼릭 맵을 가지고 있다.

위 예시에서 생성된 스냅샷 중 2개의 스냅샷은 다른 스냅샷과 심볼릭 맵이 동일하기 때문에 모두 저장하는 것은 메모리 낭비로 이어진다. 또한 동일하지 않은 11번째 줄의 스냅샷과 13번째 줄의 스냅샷도 하나의 원소를 제외하고는 동일한데, 중복되는 원소들을 각 스냅샷에서 모두 가지고 있는 것 또한 메모리 낭비로 이어질 수 있다.

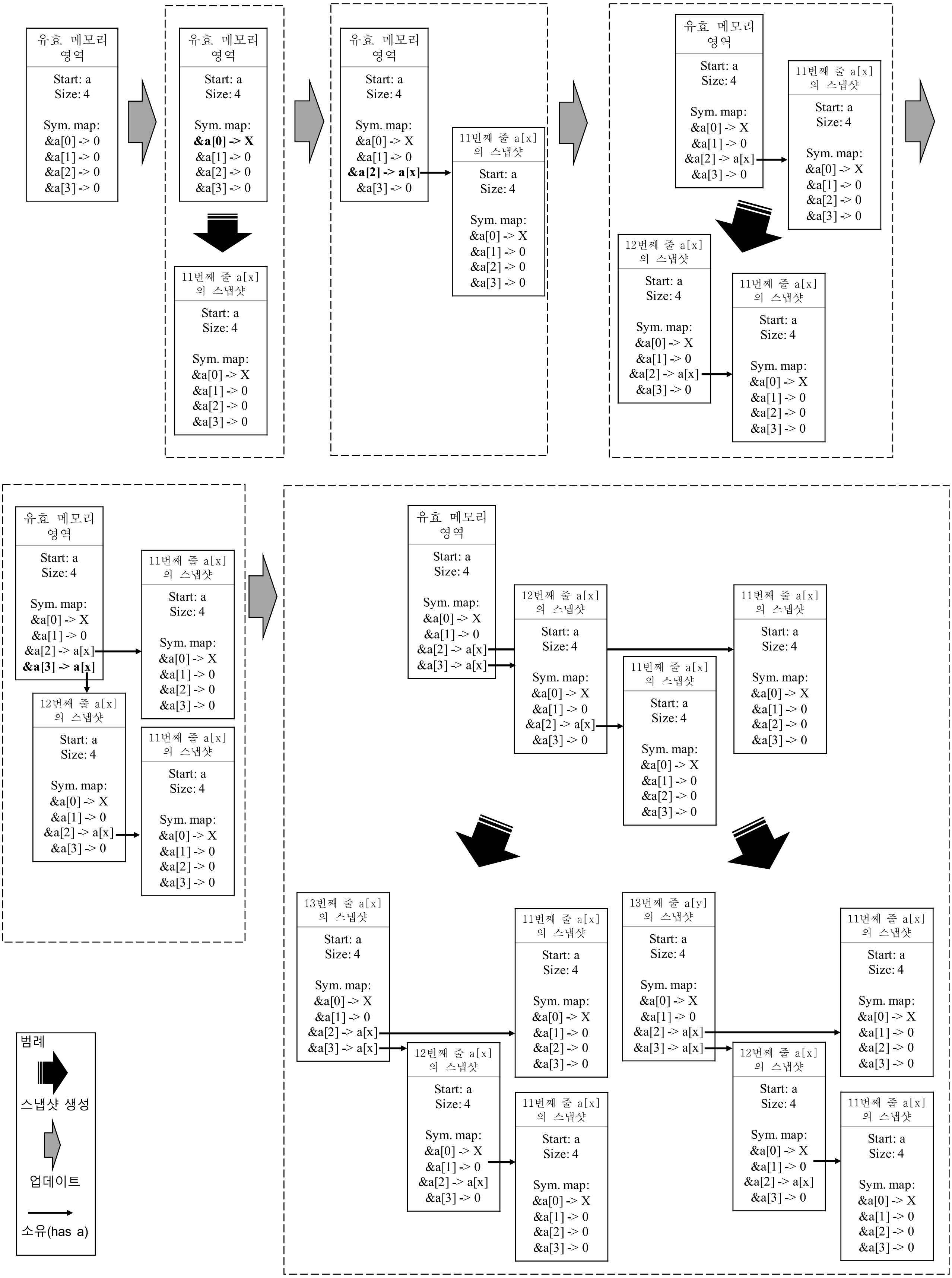

그림 3.4 예제 그림 3.3 프로그램의 스냅샷 생성 과정

그림 3.3은 기존 심볼릭 배열 참조 연산의 메모리 사용량이 기하급수적으로 늘어날 수 있음을 보여주기 위한 예시 프로그램이다. 이 프로그램 역시 입력 값 {x=3, y=1}로 concolic 테스팅을 시작했다고 가정하자. 이 프로그램의 11번째 줄과 12번째 줄에서, 배열 a에 대해 심볼릭 인덱스 참조 연산이 발생한 후 쓰기 연산으로 참조 연산 결과를 할당한다. 연산 결과가 할당될 때, 참조 연산에서 생성된 스냅샷 또한 배열의 유효 메모리 영역에 함께 저장되는데, 이 일련의 과정을 그림으로 나타내면 그림 3.4와 같다. 심볼릭 맵에서 굵은 글씨로 써진 원소들은 업데이트 된 원소를 말한다. 11번째 줄의 참조 연산 a[x]에서 생성된 스냅샷이 쓰기 연산 a[2]에 의해 유효 메모리 영역에 저장되고, 이 유효 메모리 영역을 12번째 줄의 참조 연산 a[x]에서 복사해 스냅샷으로 생성하고 쓰기 연산 a[3]에 의해 유효 메모리 영역에 할당되면서 13번째 줄의 참조연산 a[x]와 a[y]에서 생성된 스냅샷은 배열 a의 4배 크기가 된다.

이와 같이 심볼릭 배열 인덱스 참조 연산과 쓰기 연산이 반복되면서 메모리 사용량이 기하급수적으로 증가하는 문제를 해결하기 위해, 아래에서 설명할 최적화 기법 1: 중복 스냅샷의 생성 방지 기법과 최적화 기법 2: 유효 메모리 영역의 변경점 저장 기법을 적용하고 적용 전후의 메모리 사용량을 실험을 통해 비교 분석한다.

3.2 최적화 기법 1: 중복 스냅샷의 생성 방지

앞서 설명한 기존 심볼릭 배열 인덱스 참조 연산의 문제점인 메모리 사용량의 증가를 해결하기 위해 동일한 스냅샷이 중복으로 생성 되는 것을 방지하는 최적화 기법 1을 적용한다. 동일한 스냅샷은 하나의 유효 메모리 영역의 상태에 대해 두 개 이상의 스냅샷을 생성할 때 나타난다. 스냅샷 생성은 참조 연산 시 발생하고, 유효 메모리 영역의 상태가 업데이트 되는 것은 쓰기 연산이 해당 메모리 영역에 사용될 때이다. 따라서 한 유효 메모리 영역에서 쓰기 연산 사이에 일어나는 모든 참조 연산에서 생성되는 스냅샷은 동일한 심볼릭 맵을 가지고 있는 스냅샷이다.

이러한 가정을 바탕으로 스냅샷을 생성하는 시점을 매 참조 연산이 발생하는 시점에서 한 유효 메모리 영역에 대해 쓰기 연산이 발생한 이후 첫 번째 참조 연산이 발생하는 시점으로 변경한다. 쓰기 연산 이후 첫 번째 참조 연산부터 다음 쓰기 연산이 발생하기 전까지 수행되는 모든 참조 연산에서는 가장 최근에 생성된 스냅샷을 가리키는 것(points to)으로 매번 생성해 갖던(has a) 스냅샷을 대체한다. 이 과정은 하나의 유효 메모리 영역에 한정된 것이기 때문에, 각각의 유효 메모리 영역은 독립적으로 이 과정을 수행하여 메모리 중복을 방지한다.

위 과정을 CREST에 구현하기 위해서는 CREST-DEREF의 미들엔드와 백엔드를 재 구현하여야 한다. 먼저 각 유효 메모리 영역이 업데이트 되었는지를 확인하고 일련의 스냅샷들을 관리할 수 있어야 한다. CREST는 백엔드에서 대상 프로그램을 호출하여 미들엔드가 실행되기 때문에 미들엔드

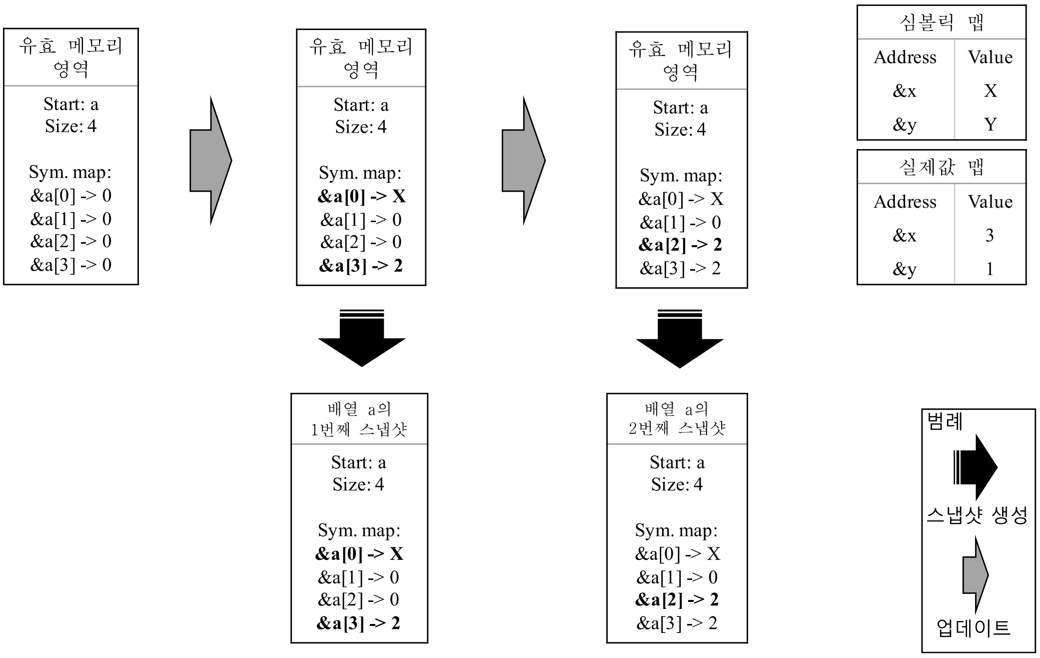

그림 3.5 최적화 기법 1 적용 시 스냅샷 생성 과정

와 백엔드의 메모리가 개별적으로 분리되어 있는데, 이 때문에 참조 연산에서 스냅샷을 포인터로 가리키면 백엔드에서 SMT 표현식을 심볼릭 경로 제약 조건으로부터 만들 때 참조 연산에 해당하는 스냅샷을 찾을 수 없게 된다. 따라서 포인터의 주소값 대신 어떤 유효 메모리 영역의 몇 번째 스냅샷인지를 참조 연산에서 저장하여 SMT 표현식 생성시 사용한다.

백엔드에서 SMT 표현식을 생성할 때, 동일한 스냅샷에 대한 표현식을 매번 다시 생성하는 것은 낭비이기 때문에 미들엔드에서 스냅샷을 한번만 생성한 것과 같이 하나의 스냅샷에 대한 SMT 표현식은 하나만 생성한다. 이 생성된 표현식을 하나의 새로운 심볼릭 변수와 동등하다고 두고 (let) 각 참조 연산에서 이 스냅샷에 대한 SMT 표현식으로 사용한다.

그림 3.3의 예시 프로그램에 최적화 기법 1을 적용했을 때 유효 메모리 영역의 상태와 생성되는 스냅샷을 그림 3.5에서 나타낸다. 그림을 보면 이 프로그램의 실행에서 생성되는 스냅샷은 2개이다. 11번째 줄의 참조 연산 `a[x]`와 `a[y]`는 모두 1번째 스냅샷을 가리키고, 13번째 줄의 참조 연산 `a[x]`와 `a[y]`는 2번째 스냅샷을 가리킨다. 이를 통해 그림 3.3의 예제 프로그램에서 최적화 기법을 하지 않은 CREST-DEREF에 비해 절반의 스냅샷이 줄어든 것을 확인할 수 있다.

3.3 최적화 기법 2: 유효 메모리 영역의 변경점 저장

최적화 기법 1을 적용한 후 중복되는 스냅샷이 여러 개 생성되어 메모리를 낭비하지는 않는다. 하지만 심볼릭 맵 전체가 변경된 것이 아니라 심볼릭 맵의 일부만 업데이트 되었을 때 유효 메모리 영역의 심볼릭 맵 전체를 복사해 스냅샷으로 만드는 것은 메모리 낭비로 이어질 수 있다.

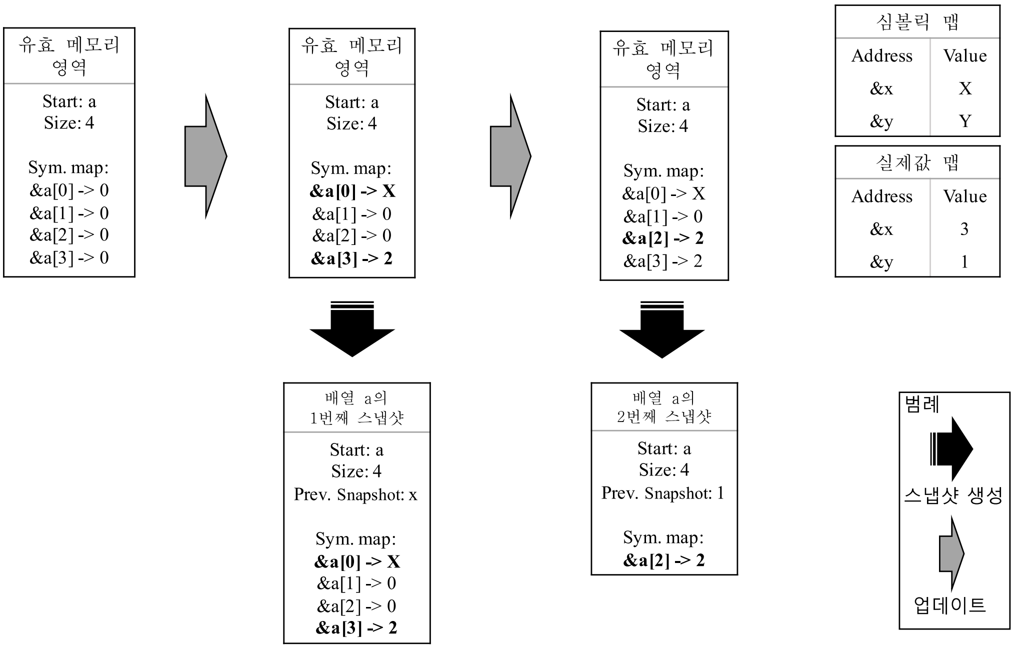

그림 3.6 최적화 기법 2 추가 적용 시 스냅샷 생성 과정

기본적으로 하나의 쓰기 연산은 유효 메모리 영역 내에 존재하는 심볼릭 맵의 원소 하나만을 업데이트한다. 매 참조 연산과 참조 연산 사이에 쓰기 연산을 통해 심볼릭 맵의 모든 원소를 업데이트 하는 경우보다는 일부 영역만을 업데이트하고 참조 연산을 수행하는 경우가 많기 때문에, 이전에 생성된 스냅샷과 새로 생성되는 스냅샷에 중복되는 심볼릭 맵의 원소가 존재할 가능성이 높고, 이 원소들을 하나의 스냅샷에만 저장하여 메모리 낭비를 줄일 수 있다. 구체적으로는 유효 메모리 영역으로부터 스냅샷을 생성할 때, 이전에 생성된 스냅샷에서 변경된 심볼릭 맵의 원소만을 복사하여 새로운 스냅샷에 저장한다. 하나의 유효 메모리 영역에 대해 생성된 스냅샷의 순서를 저장하였기 때문에, 이전에 생성된 스냅샷을 현재 스냅샷이 참조할 수 있고 이를 통해 현재 스냅샷의 심볼릭 맵에 존재하지 않는 원소를 이전 스냅샷으로부터 참조하여 사용할 수 있다. 이 방법을 유효 메모리 영역의 변경점 저장 기법이라고 한다.

최적화 기법 2를 구현하기 위해서는 한 배열에 대한 유효 메모리 영역에서 어떤 원소들이 생성된 스냅샷과 스냅샷 사이에 변경되었는지 알아야 한다. 하나의 스냅샷이 생성된 후 발생하는 업데이트를 유효 메모리 영역에 바로 반영하는 대신, 하나의 배열에 대한 유효 메모리 영역과 스냅샷을 관리하는 객체가 변경된 내역들을 저장한다. 이후 다음 참조 연산이 발생할 때 변경된 내역들을 스냅샷으로 생성하고, 유효 메모리 영역에 이 변경 내역들을 반영한다. 또한, 최적화 기법 2를 구현하기 위해서 모든 스냅샷은 이전에 생성된 스냅샷을 참조할 수 있어야 한다. 최적화 기법 1을 적용할 때 모든 스냅샷에 인덱스를 부여하였기 때문에 직전 스냅샷은 별도의 구현 없이 참조 가능하다.

그림 3.5의 예시 프로그램에 최적화 기법 2를 추가 적용했을 때 유효 메모리 영역의 상태와

생성되는 스냅샷을 그림 3.6에서 나타낸다. 최적화 기법 1을 적용한 결과와 같이 이 프로그램의 실행에서 생성되는 스냅샷은 2개이다. 하지만 최적화 기법 1을 적용했을 때와는 다르게 각 스냅샷은 이전 스냅샷의 번호를 저장하고 있다. 또한 13번째 줄의 참조 연산 `a[x]`와 `a[y]`를 위해 생성된 2번째 스냅샷은 인덱스 값이 2일때의 값만을 저장하고 있다. 이는 1번째 스냅샷이 생성된 시점에서 2번째 스냅샷이 생성될 때까지 유효 메모리 영역에 인덱스가 2일때의 참조 값만 업데이트되었기 때문이다. 최적화 기법 2를 추가 적용했을 때 2번째 스냅샷의 심볼릭 맵의 크기가 최적화 기법 1을 적용했을 때의 심볼릭 맵의 크기보다 작기 때문에 메모리 낭비가 줄어든다.

제 4 장 실험

4.1 실험 설정

CREST에 구현된 메모리 참조 연산의 유효성과 효율성을 측정하기 위해 다음과 같은 4가지 연구 질문을 작성하여 실험을 설계하였다.

연구 질문 1. 유효성: *심볼릭 배열 인덱스 참조 연산을 적용하였을 때 분기 커버리지가 얼마나 향상되는가?*

심볼릭 메모리 참조 연산을 지원하는 도구가 지원하지 않는 도구에 비해 얼마나 더 유효한지 확인하기 위해 각 실험 대상 프로그램에 대해 연산을 지원하는 도구(CREST-DEREF)와 지원하지 않는 도구(CREST-BV)를 각각 실행하였을 때 생성되는 테스트 케이스의 개수와 분기 커버리지를 측정하였다. 테스트 케이스의 개수는 올바른 테스트 케이스가 생성되었다고 가정할 때 경로 커버리지와 동일하다.

연구 질문 2. 최적화 기법 1의 메모리 효율성: *최적화 기법 1은 CREST의 메모리 사용량을 얼마나 감소 시키는가?*

심볼릭 메모리 참조 연산에 최적화 기법 1인 심볼릭 메모리의 중복 생성 방지 기법을 적용하였을 때 메모리 사용량 측면에서 최적화 기법을 적용하지 않았을 때와 비교해 얼마나 효율적으로 도구가 실행되는지를 확인하고자 한다. 따라서 각 실험 대상 프로그램에 대해 최적화 기법을 적용하지 않은 도구(DEREF-NoOPT)와 최적화 기법 1을 적용한 도구(DEREF-OPT1)를 각각 실행하였을 때 사용된 최대 메모리를 측정하고, 그 차이를 효율성((DEREF-NoOPT 메모리 사용량 / DEREF-OPT1 메모리 사용량) * 100%)으로 나타내었다.

연구 질문 3. 최적화 기법 1의 수행 시간 효율성: *최적화 기법 1은 전체 테스팅 수행 시간을 얼마나 감소 시키는가?*

연구 질문 2와 마찬가지로, 심볼릭 메모리 참조 연산에 최적화 기법 1을 적용하였을 때 최적화 기법을 적용하지 않았을 때보다 수행시간이 얼마나 단축되는지를 확인하고자 한다. 따라서 각 실험 대상 프로그램에 대해 DEREF-NoOPT와 DEREF-OPT1을 각각 실행하였을 때 사용된 분석 대상 프로그램 실행 시간을 포함한 전체 도구 실행 시간을 측정하고, 실행 시간의 비율 차이를 시간 효율성((DEREF-NoOPT 실행 시간 / DEREF-OPT1 실행 시간) * 100%)으로 나타내었다.

연구 질문 4. 최적화 기법 2의 효율성: *최적화 기법 2를 추가 적용하면 메모리 사용량과 수행 시간을 얼마나 더 감소 시키는가?*

최적화 기법 1을 적용한 심볼릭 메모리 참조 연산에 최적화 기법 2인 심볼릭 메모리의 중복 데이터 방지 기법을 추가 적용하였을 때 최적화 기법 1만 적용했을 때에 비해 메모리 사용량과 수행 시간 측면에서 얼마나 더 효율적으로 도구가 실행되는지를 확인하고자 한다. 따라서 각 실

표 4.1 분석 대상 프로그램에 대한 설명

대상 프로그램	라인수	분기문의 수	분류
Sort	36	10, 20, 또는 30	Synthesized Code
parse_tilde	309	96	GNU glibc-2.23의 word-expansion
parse_qtd_backslach	199	32	GNU glibc-2.23의 word-expansion
__hsearch_r	241	30	GNU glibc-2.23의 hash table

```
#define ARRSIZE 5
#define ITER# 5
int swap(int* a, int* b){
    int tmp = *a; *a = *b; *b = tmp;
}
int main() {
    int a,b;
    int array[ARRSIZE]={0,1,2,3,...};
    // Ascending order array
    for(int i=0;i<ITER#;i++){
       CREST_int(a);
       CREST_int(b);
       swap(&array[a], &array[b]);
    }
    if(array[0] > array[1] &&
       array[1] > array[2] &&...){
       printf("ASSERT!!\n");
       //Descending order Array
    }
    return 0;
}
```

그림 4.1 Sort 예제 프로그램의 소스 코드

험 대상 프로그램에 대해 최적화 기법 1을 적용한 도구와 최적화 기법 2를 추가 적용한 도구(DEREF-OPT1&2)를 각각 실행하였을 때 사용된 최대 메모리와 대상 프로그램 실행 시간을 포함한 전체 도구 실행 시간을 측정하고, 그 차이를 시간 효율성과 메모리 효율성(%)으로 나타내었다.

위 연구 질문들에 대답하기 위해서, 1개의 생성된 예제 프로그램과 3개의 실제 프로그램을 실험 대상으로 두고 실험을 진행하였다. 다음 절에서 이 실험에 대한 세부 사항을 기술한다.

4.1.1 실험 대상 프로그램

실험을 수행하기에 앞서 1개의 생성된 예제 프로그램 Sort와 3개의 실제 프로그램 parse_tilde, parse_qtd_backslash, __hsearch_r에 대해 설명한다.

표 4.1은 위 실험 대상 프로그램에 대해 라인 수, 분기문의 수, 분류를 각각 나타낸 것이다.

그림 4.1은 생성된 예제 프로그램 Sort의 코드이다. 이 프로그램은 정해진 크기의 오름차순 정렬된 배열에 대해 배열의 인덱스 두 개를 심볼릭 변수로 두고 해당 인덱스의 값을 교환하는 작업을 정해진 반복 횟수만큼 수행해 내림차순 배열로 바꾸는 프로그램이다. 이 프로그램 내부에는 배열이 정렬되었는지 확인하기 위한 오라클이 존재하는데, 이 때문에 분기문의 수가 배열의 크기에 따라 달라진다. 15번째 줄에 존재하는 분기문은 1개이지만, && 조건으로 배열의 모든 원소들이 정렬되었는지를 차례로 확인한다. CREST에서는 && 조건을 개별적인 분기로 처리하기 때문에 배열 원소의 개수의 2배만큼의 분기문이 존재하게 된다.

3개의 실제 프로그램 중 2개의 프로그램은 GNU 프로젝트가 유닉스/리눅스 시스템에서 필요한 C 표준 라이브러리를 구현한 GNU glibc version 2.23[28]에서 사용된다. 이 중 2개의 프로그램은 posix 규격을 따르는 wordexp에 사용되는 parse_tilde 함수와 parse_qtd_backslash 함수이다. parse_tilde 함수는 물결표(~)를 홈 디렉토리 문자열로 바꾸어주는 기능을 수행한다. parse_qtd_backslash 함수는 따옴표 내부의 백슬래시(back-slash) 문자와 그 다음 문자를 따옴표가 없을 때 표기하려던 문자로 바꾸는 기능을 수행한다. 예를 들어, 따옴표 내부의 \$나 \"를 $와 "로 바꾸어준다. 나머지 하나의 프로그램은 해쉬 함수 구현에 사용되는 __hsearch_r 함수이다. 이 함수는 키(key)와 값(value)을 가진 구조체들의 배열인 해쉬 테이블(hash table)에서 키 값에 일치하는 값을 찾아 반환해준다. 이 프로그램들은 공통적으로 배열을 사용하고 있고 인덱스 값에 따라 실행 경로가 달라진다.

4.1.2 심볼릭 변수 설정

앞서 설명한 3개의 실제 프로그램에 Concolic 테스팅을 수행하기 위하여 테스트 드라이버를 생성하고 심볼릭 변수 설정을 진행하였다.

- parse_tilde 함수

이 함수를 테스팅 하기 위한 심볼릭 변수 설정을 나타내기 위해 다음과 같이 함수의 선언 구문을 기재하였다: `static int parse_tilde(char **word, size_t *word_length, size_t *max_length, const char *words, size_t *offset, size_t wordc)`. 여기서 `**word`는 반환되는 문자열이고 `*words`가 물결표(~)를 찾기 위해 구문 분석을 진행하는 문자열이다. 이 두 문자열은 심볼릭 배열의 인덱스 참조 연산이 수행되는 경우 새로운 분기를 커버할 수 있도록 실제 값 `"0$`\"\\:/\t'"`으로 고정하였고, 나머지 문자열 정보인 `*word_length`, `*max_length`, `*offset`, `wordc`를 심볼릭 변수로 설정하였다.

- parse_qtd_backslash 함수

Word-expansion의 parse_qtd_backslash 함수의 선언 구문은 다음과 같다: `static int parse_qtd_backslash(char **word, size_t *word_length, size_t *max_length,` const char *words, size_t *offset). 이는 parse_tilde 함수와 마지막 파라메터인 wordc를 제외하고 동일한데, 심볼릭 변수 설정 역시 wordc를 제외하고 동일하게 진행하였다.

- __hsearch_r 함수

Hash table의 __hsearch_r 함수는 다음과 같이 선언되어 있다: int __hsearch_r (ENTRY item, ACTION action, ENTRY **retval, struct hsearch_data *htab). 여기서 ENTRY는 char* key와 void* data로 구성된 구조체이고, ACTION은 {FIND, ENTER}의 값을 갖는 enum 타입이다. hsearch_data는 ENTRY 배열과 그 배열의 크기를 가지고 있는 구조체이다. 이 함수의 파라메터 구조체에 존재하는 모든 원시 자료형(primitive type) 변수를 심볼릭 변수로 지정하였다. 단, htab의 ENTRY 배열의 크기를 심볼릭 변수로 설정하면 테스트 케이스가 무한하게 생성되므로 10으로 고정하였다.

4.1.3 실험 변인 설정

앞서 설정한 연구 질문들에 답하기 위하여 실험에서는 기본적으로 2개의 독립 변인(IV)을 조작하였다.

IV1: 심볼릭 배열 인덱스 참조 연산의 적용 유무

연구 질문 1에 대해 답하기 위해서 심볼릭 배열 인덱스 참조 연산을 적용한 Concolic 테스팅 도구인 CREST-DEREF와 심볼릭 배열 인덱스 참조 연산이 적용되지 않은 Concolic 테스팅 도구인 CREST-BV[11]를 비교 실험한다. CREST-DEREF는 CREST-BV와 마찬가지로 bitwise 연산을 지원하고 추가적으로 배열 인덱스 참조 연산을 지원한다. 또한, CREST-DEREF는 CREST-BV와 동일하게 Z3 SMT solver를 사용한다.

IV2: 최적화 기법 1과 2의 적용 유무

연구 질문 2, 3, 4에 답하기 위해서 CREST-DEREF에 최적화 기법 1과 2를 적용하였을 때와 적용하지 않았을 때를 비교 실험한다. 최적화 기법 2는 스냅샷의 변경점 만을 저장하고 나머지 정보는 이전 스냅샷을 참고하는 방식이기 때문에 최적화 기법 1을 적용하지 않고 적용할 수 없다. 따라서 최적화 기법이 없는 CREST(NoOPT), 최적화 기법 1이 적용된 CREST(OPT1), 최적화 기법 1,2가 모두 적용된 CREST(OPT1&2) 3가지로 나뉜다.

또한, Sort 프로그램에 대해서는 메모리 사용량 관점에서 CREST-DEREF의 확장성을 알아보기 위해 2개의 독립 변인을 추가하였다.

IV3-Sort: 배열의 크기(ARRSIZE)

배열 인덱스 참조 연산이 수행될 때, 해당 배열의 크기가 커질 때 메모리 사용량 및 수행 시간의 차이가 최적화 기법 유무에 따라 얼마나 차이가 나는지 확인할 수 있도록 추가하였다.

IV4-Sort: 배열 인덱스 참조 연산을 수행하는 횟수(ITER#)

배열 인덱스 참조 연산의 참조와 쓰기가 수행되는 횟수가 증가할 때 메모리 사용량 및 수행 시간의 차이가 최적화 기법 유무에 따라 얼마나 차이가 나는지 확인할 수 있도록 추가하였다.

최적화 기법이 적용된 CREST-DEREF의 유효성과 효율성을 측정하기 위해서, 공통적으로 다음과 같은 4개의 종속 변인(DV)을 선택하였다.

DV1: 생성된 테스트 케이스의 수

DV2: 커버한 분기문의 수

CREST-BV와 CREST-DEREF를 비교 실험할 때 각각의 도구가 대상 프로그램의 분기문 중 커버한 개수를 측정한다. 전체 분기문 및 커버한 분기문의 수는 CREST 도구 자체에서 측정한 것을 기준으로 한다. CREST에서 분기 커버리지 측정시, CIL에서 분기문 내의 &&, ||연산을 별도의 분기로 만들기 때문에 실제 분기문의 개수보다 더 많은 수의 분기로 측정될 수 있다.

DV3: 최대 메모리 사용량

CREST-DEREF에 최적화 기법들을 적용하여 대상 프로그램에 대해 도구를 실행하였을 때 도구가 사용한 최대 메모리(Peak Resident Set Size)를 측정한다. 이 값은 Linux에서 프로세스가 수행 중일 때 존재하는 /proc/[pid]/status에 존재하는데, 프로세스가 끝나기 직전에 이 파일의 VmHWM (Peak RSS)이 해당 프로세스가 실행되는 동안 가장 많은 메모리를 사용한 시점의 메모리 값이다. 이 값을 측정하기 위해 CREST가 종료되기 직전에 pid 값을 이용해 자신의 VmHWM를 출력하고, 이를 사용한다.

DV4: 수행 시간

CREST-DEREF에 최적화 기법들을 적용하여 대상 프로그램에 대해 도구를 실행하였을 때 대상 프로그램 수행 시간을 포함한 전체 도구 실행 시간을 측정한다. 도구 실행 시 time 명령어를 추가하고 도구 실행 종료 시 출력된 실제(real) 수행 시간을 사용한다.

4.1.4 실험 환경

모든 실험은 Intel Core i5-3570K 3.40Ghz 프로세서와 8GB 메모리에서 실행되었고, CREST 컴파일에 사용된 gcc (g++)버전은 4.9.2, 생성된 경로 제약 조건을 풀기 위해 사용된 SMT solver는 Z3로 4.4.1버전을 사용한다. 도구 실행 시 사용된 탐색 기법은 깊이 우선 탐색(depth first search, DFS)이고, 모든 실험에서 종료 조건은 다음 세 가지 조건 중 하나를 만족하는 것이다:

1. 가능한 모든 경로를 다 탐색하였다.
2. 메모리 사용량이 4GB를 초과하여 out of memory (OOM) 에러로 도구 또는 분석 대상 프로그램이 비정상 종료된다.

표 4.2 CREST-BV와 CREST-DEREF에서 생성된 테스트 입력 값 및 분기 수 비교

Target	CREST-BV		CREST-DEREF	
	# of TC	커버한 분기 수 /전체 분기 수	# of TC	커버한 분기 수 /전체 분기 수
Sort (5, 5)	1	3/10 (30.00%)	5	10/10 (100.0%)
Sort (5, 10)	1	3/10 (30.00%)	5	10/10 (100.0%)
Sort (5, 15)	1	3/10 (30.00%)	5	10/10 (100.0%)
Sort (10, 5)	1	3/20 (15.00%)	10	20/20 (100.0%)
Sort (10, 10)	1	3/20 (15.00%)	10	20/20 (100.0%)
Sort (10, 15)	1	3/20 (15.00%)	10	20/20 (100.0%)
Sort (15, 5)	1	3/30 (10.00%)	11	23/30 (76.67%)[1]
Sort (15, 10)	1	3/30 (10.00%)	15	30/30 (100.0%)
Sort (15, 15)	1	3/30 (10.00%)	13(N/A)	27/30 (90.00%)
parse_tilde	5	25/96 (26.04%)	41	49/96 (51.04%)
parse_qtd_backslash	2	18/32 (56.25%)	11	27/32 (84.38%)
__hsearch_r	39	23/30 (76.66%)	112	24/30 (80.00%)

3. 대상 프로그램 실행을 포함한 전체 도구 실행 시간이 24시간 (MSR의 경우 48시간)을 초과하였다.

4.2 실험 결과

4.2.1 연구 질문1. 심볼릭 메모리 참조 연산 지원을 통한 분기 커버리지 향상

표 4.2는 각 대상 프로그램에 대해 CREST-BV와 CREST-DEREF를 실행하였을 때 생성된 테스트 케이스 수 (# of TC)와 커버한 분기문의 수, 그리고 전체 분기문 수에 대해 커버한 분기문의 수의 비율(%)인 분기 커버리지를 나타낸 것이다. Sort 예제의 이름에 적혀있는 (x, y)는 각각 ARRSIZE와 ITER#를 나타낸 것이다. 모든 결과에서 CREST-DEREF가 CREST-BV보다 많은 테스트 케이스를 생성하고 분기 커버리지도 더 높게 나온 것으로 나타난다.

Sort 예제의 경우 배열이 내림 차순으로 정렬되었는지 확인하기 위한 조건문을 참으로 만들기 위해서는 심볼릭 배열 참조 연산이 필요하기 때문에 CREST-BV에서는 최초 수행 이후 다음 테

---

[1] Swap을 5번 수행하는 것으로는 원소가 15개인 배열을 완전히 정렬할 수 없음.

```
static int parse_qtd_backslash(char **word, size_t *word_length,
         size_t *max_length, const char *words, size_t *offset){
    size_t i;
    ...
  switch(words[1+ *offset]){
    case '0': return WRDE_SYNTAX;
    case '\n':
      ++(*offset);
      break;
    case '”':
    case '\\':
      *word = w_addchar (*word, word_length,
                  max_length, words[1 + *offset]);
  }...
}
```

그림 4.2 parse_qtd_backslash에서 심볼릭 배열 인덱스 참조 연산 사용 예시

표 4.3 최적화 기법 적용 유무에 따른 메모리 사용량 비교

Target	최대 메모리 사용량 (MB)			메모리 효율성 (×100)	
	NoOPT (A)	OPT1 (B)	OPT1&2 (C)	(B)/(A)	(C)/(B)
Sort (5, 5)	67.60	15.55	15.27	23.00%	98.20%
Sort (5, 10)	OOM	19.70	18.70	-	94.92%
Sort (5, 15)	OOM	35.06	32.87	-	93.75%
Sort (10, 5)	194.48	19.64	19.28	10.10%	98.17%
Sort (10, 10)	OOM	33.96	30.52	-	89.87%
Sort (10, 15)	OOM	70.34	64.58	-	91.81%
Sort (15, 5)	N/A(>24h)	47.49	46.32	-	97.54%
Sort (15, 10)	OOM	103.64	100.29	-	96.77%
Sort (15, 15)	OOM	N/A(>24h)	N/A(>24h)	-	-
parse_tilde	23.24	22.50	22.36	96.82%	99.38%
parse_qtd_backslash	21.84	21.73	21.70	99.50%	99.86%
__hsearch_r	34.18	33.78	33.50	98.83%	99.17%

스트 케이스를 생성하는데 실패하였다.

위 실험에서 CREST-DEREF가 CREST-BV보다 더 높은 커버리지를 달성하는 것을 보임으로써 심볼릭 배열 인덱스 참조 연산이 필요한 대상 프로그램에 대해 CREST-DEREF가 이 연산을 적절히 수행하여 더 높은 커버리지를 달성할 수 있음을 보였다.

그림 4.2는 parse_qtd_backslash에서 심볼릭 배열 인덱스 참조 연산이 사용되는 구문의 예시

표 4.4 최적화 기법 적용 유무에 따른 테스팅 시간 비교

Target	수행 시간 (s)			시간 효율성 (×100)	
	NoOPT (A)	OPT1 (B)	OPT1&2 (C)	(B)/(A)	(C)/(B)
Sort (5, 5)	3.35	2.35	1.50	70.15%	63.83%
Sort (5, 10)	OOM(>11.40)	26.37	18.94	-	71.82%
Sort (5, 15)	OOM(>10.84)	230.28	97.73	-	42.44%
Sort (10, 5)	1,597.76	122.12	105.96	7.64%	86.77%
Sort (10, 10)	OOM(>11.47)	3,262.27	2,949.15	-	90.40%
Sort (10, 15)	OOM(>10.63)	13,087.43	12,102.00	-	92.47%
Sort (15, 5)	N/A(>24h)	2,875.21	2,551.46	-	88.74%
Sort (15, 10)	OOM(>10.69)	79,651.43	63,517.21	-	79.74%
Sort (15, 15)	OOM(>10.64)	N/A(>24h)	N/A(>24h)	-	-
parse_tilde	2.28	1.66	1.53	72.81%	92.17%
parse_qtd_backslash	0.49	0.49	0.46	100.00%	93.88%
__hsearch_r	51.01	29.23	29.18	57.30%	99.83%

이다. 5번째 줄의 switch문은 문자열인 words의 특정 offset에 있는 값에 따라 각각 다른 case 분기로 이동하게 만드는 구문이다. 예를 들어 newline (₩n)인 경우 7번째 줄부터 수행하고, 역슬래쉬인 경우 11번째 줄부터 수행된다. 문자열은 실제 값으로 고정되어 있기 때문에 concolic 테스팅에서 커버리지가 높아지기 위해서는 배열의 어떤 위치에 아직 수행하지 않은 경로를 수행하도록 하는 값이 있는지 알아야 하고, 이를 심볼릭 배열 인덱스 참조 연산을 통해 알 수 있다.

4.2.2 연구 질문2. 최적화 기법 1의 메모리 효율성

표 4.3은 각 대상 프로그램에 대해 최적화 기법을 적용하지 않았거나, 최적화 기법 1만 적용하였거나, 최적화 기법 1,2를 모두 적용해 실행했을 때 테스팅 도구의 최대 메모리 사용량을 나타낸 것이다. 표 4.2와 마찬가지로 Sort 예제 프로그램에서 (x, y)는 ARRSIZE와 ITER#를 나타낸다.

최적화 기법을 적용하지 않았을 때 Sort 예제 프로그램을 테스팅한 경우, ITER#이 10 이상일 때 out of memory(OOM) 발생으로 도구가 비정상 종료 되는 것을 확인할 수 있다. ITER#의 개수가 증가함에 따라 심볼릭 배열 인덱스 참조 연산의 쓰기와 읽기 횟수가 증가하였고, 이 연산이 반복 수행되면서 메모리 사용량이 그림 3.3에서 보여준 예시와 같이 기하급수적으로 증가한다. 반면 ITER#가 10인 Sort 예제 프로그램을 최적화 기법 1이 적용된 도구로 테스팅하면, 메모리 사용량

이 소폭(e.g. (10, 5)일 때 19.28MB에서 (10, 10)일 때 30.52MB) 증가하였다. 또한, Sort (10, 5)일 때 메모리 사용량을 비교하면 194.48MB와 19.64MB로 약 10배가량 메모리 사용량이 차이 나는 것을 확인할 수 있다.

3개의 실제 프로그램에서 메모리 사용량을 비교하면 최적화 기법을 적용하지 않을 때보다 최적화 기법 1을 적용하였을 때 더 적은 메모리를 사용하지만, 이 실제 프로그램들을 분석하는데 사용된 메모리가 적기 때문에 기법 적용 유무에 따른 메모리 사용량 차이도 작다 (e.g. parse_tilde 프로그램에서 각각 23.24MB, 22.50MB로 4%이내의 차이를 보임).

4.2.3 연구 질문3. 최적화 기법 1의 시간 효율성

표 4.4는 각 대상 프로그램에 대해 최적화 기법을 적용하지 않았거나, 최적화 기법 1만 적용하였거나, 최적화 기법 1,2를 모두 적용해 실행했을 때 대상 프로그램의 실행 시간을 포함하여 테스팅 도구가 테스팅을 수행하는데 걸린 전체 실행 시간을 나타낸 것이다.

Sort 예제 프로그램에서 최적화 기법을 적용하지 않은 CREST-DEREF는 ITER#를 증가시켰을 때 메모리 사용량이 크게 늘어나 OOM이 발생하여 실행 시간 측정이 불가능하였다. 반면 ARRSIZE가 늘어남에 따라 실행시간이 크게 늘어난 것을 볼 수 있다. 특히, 최적화 기법을 적용하지 않았을 때는 Sort (5, 5)와 Sort (10, 5)에서 각각 3.35s와 1,597.76s로 수행시간 차이가 약 477배 났으나 최적화 기법을 적용하였을 때는 2.35s와 122.12s로 수행 시간 차이가 약 51배였다. 이를 통해 최적화 기법을 적용하였을 때 수행 시간이 더 빠르면서 배열의 크기 변화에 따른 수행 시간 차이의 폭도 더 적은 것을 확인할 수 있다.

3개의 실제 프로그램에서 걸린 테스팅 시간을 비교하면 최적화 기법을 적용하지 않을 때보다 최적화 기법 1을 적용하였을 때 더 빨라짐을 확인할 수 있었다 (e.g. __hsearch_r 프로그램에서 각각 51.01s, 29.23s로 최적화 기법을 적용한 후 수행시간이 57.30%로 단축됨).

4.2.4 연구 질문4. 최적화 기법 2의 메모리 및 시간 효율성

표 4.3과 표 4.4에서 OPT1과 OPT1&2를 비교하여 최적화 기법 2를 추가 적용하였을 때 효율성이 얼마나 증가하는지를 확인한다. 최대 메모리 사용량 관점에서는 Sort (10, 10) 예제 프로그램을 제외하고는 큰 차이를 보이지 않았고, 이 프로그램 도한 OPT1&2가 OPT1의 약 89.87%의 메모리를 사용하여 테스팅을 수행하였다. 수행 시간의 경우 수행시간이 짧은 경우 더 큰 효율성을 보여주었다. 또한, 최적화 기법 2를 추가 적용하였을 때 커버리지가 최적화 기법을 적용하지 않았거나, 최적화 기법 1만 적용하였을 때와 동등하고, 수행 시간과 메모리 사용량 측면에서는 항상 더 효율적이었기 때문에 최적화 기법 2를 추가 적용하는 것이 좋다고 결론 내릴 수 있다.

```
// Read the len amount of sectors starting from startLUN’th LU
// and store the data into buf
MSR(byte *buf, int startLUN, int len) {
  curLUN = startLUN;
  numScts = len;
  pBuf = buf;

  while(numScts > 0) {
    readScts = # of sectors to read in the current LU
    numScts -= readScts;
    while(readScts > 0) {
      curPU = LU[curLUN]->firstPU;
      while(curPU != NULL) {
        while(...) {
          conScts = # of consecutive Pses to read in curPU
          offset = the starting offset of these consecutive Pses
        }
        // Copy data in the corresponding physical sectors
        // into the read buffer
        BML_READ(pBuf, curPU, offset, conScts);
        readScts = readScts - conScts;
        pBuf = buf + conScts;
        curPU = curPU->next;
      }
    }
    curLUN++;
  }
  // check the requirement property
  for(i=0;i<len;i++) { assert (buf[i]==LS[i]); }
}
```

그림 5.1 다중 섹터 읽기의 의사 코드[2]

제 5 장 사례 연구: 다중 섹터 읽기(MSR)

4번째 실제 프로그램은 다중 섹터 읽기(Multi-sector read, MSR)로 삼성전자 OneNAND 플래시의 device driver 중 하나로 플래시 메모리의 읽기 속도를 높이기 위해 여러 개의 인접한 물리 섹터를 한번에 읽는 기능을 수행한다[9]. 이 프로그램은 157줄로 구성되어 있고 분기문의 수는 24개이다. 아래 절에서 MSR에 대해 자세히 기술한다.

---

[2] 이 의사 코드는 [9]에서 차용된 것임.

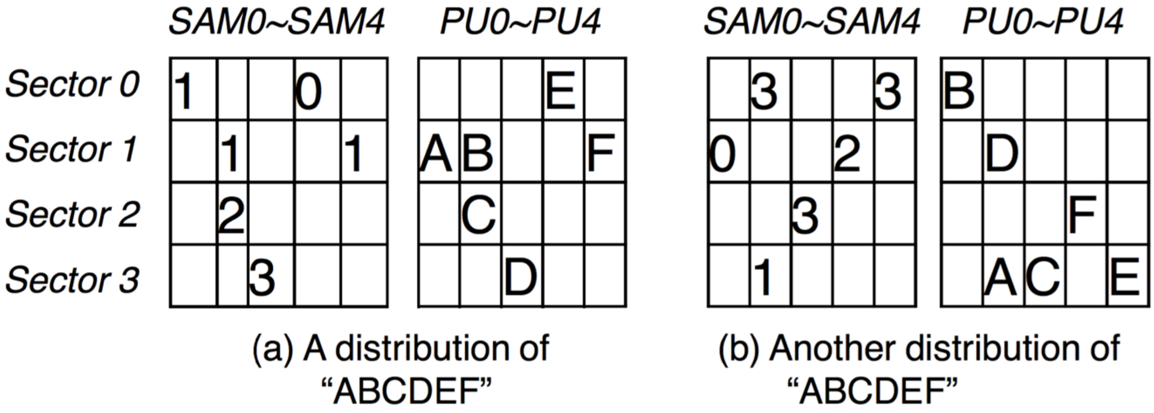

그림 5.2 논리 섹터 ABCDEF의 서로 다른 두 물리 섹터 분포[3]

5.1 프로그램 설명

MSR은 OneNAND 플래시 메모리 디바이스 드라이버의 핵심 구성 요소로 플래시 메모리의 읽기 성능을 향상시키기 위해 연속된 물리섹터를 가능한 최대로 한번에 읽는 기능을 수행하는 함수이다. 플래시 메모리 하드웨어의 특성 상 사용자가 보는 논리 섹터와 실제 플래시 메모리의 물리 섹터가 서로 구분되어 있으며 플래시 메모리 디바이스 드라이버가 논리 섹터와 물리 섹터의 복잡한 매핑을 담당한다. 이와 같은 복잡한 논리-물리 섹터 매핑 때문에 MSR은 복잡한 자료구조를 순회하면서 플래시 메모리 섹터를 읽어야 한다. 그렇기 때문에 MSR은 4단계의 중첩된 반복 구조를 갖고 있다. MSR의 수행이 끝났을 때 읽기 버퍼(read buffer)에 플래쉬 메모리에 저장된 데이터가 논리 섹터 순서대로 올바르게 저장되어 있어야 한다.

그림 5.1은 다중 섹터 읽기 동작의 의사 코드를 나타낸다. 실제 구현은 157줄로 구성된 하나의 함수이며 다음 세 파라미터를 입력을 받는다.

(1) `buf`: 플래시 메모리에서 데이터를 읽어 저장할 메모리 공간

(2) `startLUN`: 읽기 동작을 시작할 논리 유닛 주소

(3) `len`: `startLUN` 논리 유닛으로 부터 읽을 데이터의 크기

가장 바깥쪽의 반복문(8-27번 행)은 주어진 크기 `len`만큼의 데이터를 읽을 때 까지 논리 유닛을 순회한다. 두 번째 바깥쪽의 반복문(11-25번 행) 현재 논리 유닛의 논리 섹터를 완전히 읽을 때 까지 반복하여 데이터를 읽는다. 세 번째 반복문(13-24번 행) 현재 논리 유닛에 매핑된 물리 유닛들을 순회하고 가장 안 쪽의 반복문은(14-17번 행) 얼마나 많은 논리 섹터가 연속된 물리 섹터에 매핑되는지 체크하여 `conScts` 변수 및 `offset` 변수를 설정한다. `conScts`와 `offset`

---

[3] 이 그림은 [9]에서 차용된 것임.

변수는 20번째 행에서 BML_READ함수의 파라메터로 사용되어 실제 물리 섹터로부터 연속된 데이터를 읽는 역할을 한다. 다중 섹터 읽기 동작이 끝났을 때 읽기 버퍼 buf에 플래시 메모리에 저장된 데이터가 논리 섹터 순서대로 올바르게 저장되어 있어야 한다.

그림 5.2는 동일한 논리 섹터 “ABCDEF”가 물리 유닛 0~4번에 걸쳐 분포되어 있는 서로 다른 두 경우를 나타낸다. 여기서 플래시 메모리의 물리 유닛 0~2번은 논리 유닛 0번에 물리유닛 3~4는 논리 유닛 1번에 매핑되어 있다고 가정하자. Sector Allocation Map(SAM)은 물리 섹터와 논리 섹터의 매핑 정보를 담고 있는 자료 구조이다. 예를 들어 그림 5.2의 (a)에서 SAM[0].sector[0]의 값이 1인 것이 의미하는 것은 논리 유닛 0번의 논리 섹터 0을 읽기 위해서는 먼저 물리 유닛 0번의(SAM[0]의 sector[0]이 설정되어 있기 때문) 물리 섹터 1(SAM[0].sector[0]의 값)의 값이 설정되어 있다는 의미이다. (a)의 경우는 ABCDEF가 물리 유닛과 물리 섹터의 순서에 따라 분포되어 있기 때문에 물리 유닛과 물리 섹터를 순서대로 읽기만 해도 논리 섹터를 올바르게 읽을 수 있다. 그러나 (b)의 경우 논리 섹터의 순서와 물리 유닛 및 물리 섹터의 순서가 서로 다르기 때문에 (a)와 같이 순서대로 읽는 것으로는 논리 섹터를 올바르게 읽을 수 없다.

5.2 테스트 드라이버 및 심볼릭 변수 설정

MSR이 올바르게 동작하는지 검증하기 위해서는 MSR의 동작 환경을 설정해야 한다. MSR은 논리 섹터가 물리 유닛에 임의로 분포되어 있고 각 논리 섹터가 어디에 저장되어 있는지 SAM을 통해 정확하게 알 수 있다고 가정하고 동작한다. 하지만 하드웨어 구조 및 논리 섹터 - 물리 섹터 사상 관계로 인해 다음과 같은 조건을 항상 만족해야 한다.

- 하나의 물리 유닛은 둘 이상의 논리 유닛과 사상될 수 없다.
- 만약 i번째 논리 섹터가 j번째 물리 유닛의 k번째 물리 섹터에 저장되었다면 j번째 물리 유닛의 SAM의 (i mod m) 번째 오프셋은 물리 섹터 k를 올바르게 가리켜야 한다. 여기서 m 은 한 유닛이 가질 수 있는 최대 섹터 크기로 본 검증에서는 4로 가정한다.
- i번째 논리 섹터가 저장된 물리 섹터 번호는 $\left\lfloor \frac{i}{m} \right\rfloor$ 번째 논리 유닛을 구성하는 물리 유닛들의 SAM 테이블의 (i mod m)번째 오프셋 가운데 하나에만 저장되어 있어야 한다.

IV3-MSR: 물리 유닛(PU)
IV4-MSR: 논리 섹터(LS)

MSR은 LS와 PUN의 크기로 각각 5 ≤ LS ≤ 8, 4 ≤ PU ≤ 8를 가질 수 있다. 즉, LS와 PUN의 값에 따라 그림 5.2에 나타난 배열의 크기가 커지고 모든 경우를 테스트하기 위해서는 더 많은 테스트 케이스를 생성하게 된다. 따라서, LS와 PU의 값을 조절하여 생성해야 할 테스트 케이스

```
typedef struct SAMtype{
    unsigned char offset[SECT_PER_U];
}SAM_type;
typedef struct PUtype{
    unsigned char sect[SECT_PER_U];
}PU_type;
const char *LS0=”abcd”, *LS1=”efgh”;
void MSR_TEST(){
    SAM_type SAM[NUM_PUN]; PU_type PU[NUM_PUN]; ...
    for (i=0; i< NUM_LS; i++){
        unsigned char idxPU, idxSect;
        CREST_unsigned_char(idxPU);
        CREST_unsigned_char(idxSect);
        PU[idxPU].sect[idxSect]= LS0[i];
        SAM[idxPU].offset[i]= idxSect;
    }
    ... MSR(...);
}
```

그림 5.3 MSR의 심볼릭 변수 설정 의사 코드

수가 증가함에 따라 CREST-DEREF가 얼마나 메모리 및 실행 시간이 증가하는지, 그리고 최적화 기법을 적용하였을 때 얼마나 차이가 나는지 확인한다.

그림 5.3은 MSR을 테스팅하기 위해 심볼릭 변수를 설정한 의사 코드를 나타낸다. LS (논리적 섹터)가 SAM과 PU (물리적 유닛)에 삽입되는 위치를 idxPU와 idxSect 변수를 통해 지정하는데, 이 변수들이 배열의 인덱스로 사용된다. 이 변수들의 값에 따라 프로그램 실행이 달라지기 때문에 이 인덱스들을 심볼릭 변수로 지정한다.

실험 환경과 독립 변수, 종속 변수는 제 4 장의 실험과 동일하게 진행하게 진행하였고, 깊이 우선 탐색(DFS) 전략을 사용하여 모든 경로를 탐색할 수 있도록 하였다.

5.3 실험 결과

표 5.1은 CREST-DEREF가 주어진 PU 개수와 LS의 개수 (MSR (PU#, LS#))로 MSR을 실행하였을 때 생성된 테스트 케이스의 수를 나타낸 것이다. CREST-DEREF는 심볼릭 배열 참조 연산을 지원하기 때문에 MSR 함수에서 나타날 수 있는 모든 경로를 실행하였다. MSR 함수는 모든 입력 값에 대해 도달 불가능한 분기는 존재하지 않으므로 표에서 정상 종료로 나타난 PU 개수와 LS 개수에 따른 실행에서 100%의 분기 커버리지(24개)를 달성하였다. 표 5.1은 CREST-DEREF로 MSR을 실행하였을 때 생성된 테스트 케이스의 수를 나타낸 것으로 DFS 전략을 사용해 concolic 테스팅을 수행하였기 때문에 모든 테스트 케이스는 각각 서로 다른 실행 경로를 수행한 것이다. PU와 LS의 합이 15이상인 대상 프로그램은 CREST-DEREF로 테스팅을 진행하였을 때 48시간 이상 수행이 계속되었기 때문에 모든 테스트 케이스를 생성하지 못했다.

표 5.1 CREST-DEREF에서 생성된 테스트 입력 값 및 분기 수 비교

Target	CREST-DEREF	
	# of TC	커버한 분기 수 /전체 분기 수
MSR (4, 5)	386	24/24 (100.0%)
MSR (4, 6)	856	24/24 (100.0%)
MSR (4, 7)	2,184	24/24 (100.0%)
MSR (4, 8)	6,056	24/24 (100.0%)
MSR (5, 5)	1,136	24/24 (100.0%)
MSR (5, 6)	3,029	24/24 (100.0%)
MSR (5, 7)	8,928	24/24 (100.0%)
MSR (5, 8)	28,827	24/24 (100.0%)
MSR (6, 5)	2,984	24/24 (100.0%)
MSR (6, 6)	8,870	24/24 (100.0%)
MSR (6, 7)	29,476	24/24 (100.0%)
MSR (6, 8)	108,258	24/24 (100.0%)
MSR (7, 5)	6,890	24/24 (100.0%)
MSR (7, 6)	22,495	24/24 (100.0%)
MSR (7, 7)	83,064	24/24 (100.0%)
MSR (7, 8)	N/A(>48h)	23/24 (95.83%)
MSR (8, 5)	14,380	24/24 (100.0%)
MSR (8, 6)	51,072	24/24 (100.0%)
MSR (8, 7)	N/A(>48h)	24/24 (100.0%)
MSR (8, 8)	N/A(>48h)	23/24 (95.83%)

반면 CREST-BV를 실행하였을 때 생성된 모든 테스트 케이스들은 MSR 함수에 진입하는데 실패하여 커버리지가 0%로 나타났는데, 이는 테스트 드라이버에서 심볼릭 변수를 이용해 생성된 SAM과 PU가 함수 입력값으로 필요한 제약 조건에 부합하지 않았기 때문이다.

표 5.2 최적화 기법 적용 유무에 따라 MSR 테스팅시 메모리 사용량 비교

Target	최대 메모리 사용량 (MB)			메모리 효율성 (×100)		
	NoOPT (A)	OPT1 (B)	OPT1&2 (C)	(B)/(A)	(C)/(A)	(C)/(B)
MSR (4, 5)	120.09	24.00	20.84	19.99%	17.35%	86.83%
MSR (4, 6)	366.07	41.26	32.29	11.27%	8.82%	78.26%
MSR (4, 7)	1,624.08	102.70	76.72	6.32%	4.72%	74.70%
MSR (4, 8)	OOM	270.02	181.18	-	-	67.10%
MSR (5, 5)	399.78	48.01	37.25	12.01%	9.32%	77.59%
MSR (5, 6)	1,572.50	117.78	82.95	7.49%	5.28%	70.43%
MSR (5, 7)	OOM	365.74	246.08	-	-	67.28%
MSR (5, 8)	OOM	1,265.57	808.84	-	-	63.91%
MSR (6, 5)	1,092.48	122.63	91.09	11.22%	8.34%	74.28%
MSR (6, 6)	OOM	341.74	231.52	-	-	67.75%
MSR (6, 7)	OOM	1,226.57	789.98	-	-	64.41%
MSR (6, 8)	OOM	OOM	3,110.28	-	-	-
MSR (7, 5)	2,861.59	259.36	182.54	9.06%	6.38%	70.38%
MSR (7, 6)	OOM	878.25	570.49	-	-	64.96%
MSR (7, 7)	OOM	3,624.90	2,269.57	-	-	62.61%
MSR (7, 8)	OOM	N/A(>48h)	N/A(>48h)	-	-	-
MSR (8, 5)	OOM	556.47	373.64	-	-	67.14%
MSR (8, 6)	OOM	2,153.49	1,357.32	-	-	63.03%
MSR (8, 7)	OOM	N/A(>48h)	N/A(>48h)	-	-	-
MSR (8, 8)	OOM	N/A(>48h)	N/A(>48h)	-	-	-
평균				11.05%	8.60%	70.04%

표 5.2는 MSR을 테스팅하였을 때 최적화 기법 적용 유무에 따른 메모리 사용량을 비교한 것이다. OPT1은 NoOPT가 OOM으로 테스팅에 실패한 MSR 프로그램들을 성공적으로 테스팅할 수 있었고, NoOPT에서 실행하는데 성공한 MSR (5, 6)의 경우 메모리 사용량이 10배 이상 차이나는 것을 확인할 수 있었다. 또한, 최적화 기법 2를 추가 적용하였을 때, 메모리 사용량이 적게는 13.17% (MSR 4,5인 경우)에서 많게는 37.39%까지 감소한 것을 확인할 수 있었다. 특히, PU와 LS가 증가함에 따라 메모리 사용량이 커질수록 메모리 효율성이 증가하였고, OPT1에서 OOM으로 인해 MSR (6, 8)을 테스팅하는데 실패하였지만 OPT1&2에서는 모든 경로를 탐방하고 100% 분기 커버리지를

표 5.3 최적화 기법 적용 유무에 따라 MSR의 테스팅 시간 비교

Target	수행 시간 (s)			시간 효율성 (×100)		
	NoOPT (A)	OPT1 (B)	OPT1&2 (C)	(B)/(A)	(C)/(A)	(C)/(B)
MSR (4, 5)	470.42	117.17	106.90	24.91%	22.72%	91.23%
MSR (4, 6)	1,872.44	355.90	337.98	19.01%	18.05%	94.96%
MSR (4, 7)	7,803.80	1,190.67	1,180.99	15.26%	15.13%	99.19%
MSR (4, 8)	OOM(>10,980)	5,863.68	5,729.05	-	-	97.70%
MSR (5, 5)	2,037.77	464.07	443.75	22.77%	21.78%	95.62%
MSR (5, 6)	8,434.68	1,583.84	1,570.56	18.78%	18.62%	99.16%
MSR (5, 7)	OOM(>12,352)	7,898.21	7,708.58	-	-	97.60%
MSR (5, 8)	OOM(>11,349)	36,798.50	36,401.72	-	-	98.92%
MSR (6, 5)	3,931.45	1,405.49	1,397.52	35.75%	35.55%	99.43%
MSR (6, 6)	OOM(>13,776)	5,759.01	5,676.14	-	-	98.56%
MSR (6, 7)	OOM(>12,644)	28,217.04	27,794.56	-	-	98.50%
MSR (6, 8)	OOM(>11,513)	OOM(>145,579)	159,772.06	-	-	-
MSR (7, 5)	11,309.79	3,863.78	3,822.12	34.16%	33.79%	98.92%
MSR (7, 6)	OOM(>14,367)	17,648.59	17,406.64	-	-	98.63%
MSR (7, 7)	OOM(>13,081)	106,405.74	105,092.09	-	-	98.77%
MSR (7, 8)	OOM(>12,072)	N/A(>48h)	N/A(>48h)	-	-	-
MSR (8, 5)	OOM(>17,530)	8,680.21	8,531.76	-	-	98.29%
MSR (8, 6)	OOM(>14,312s)	43,360.44	42,422.89	-	-	97.84%
MSR (8, 7)	OOM(>13,678s)	N/A(>48h)	N/A(>48h)	-	-	-
MSR (8, 8)	OOM(>12,001s)	N/A(>48h)	N/A(>48h)	-	-	-
평균				24.38%	23.66%	97.71%

달성하는데 성공하였다.

표 5.3은 MSR을 테스팅하였을 때 최적화 기법 적용 유무에 따른 대상 프로그램 실행 시간을 포함한 전체 수행 시간을 비교한 것이다. NoOPT와 OPT1을 비교하면 OPT1이 최저 34.16% (MSR 7, 5), 최대 15.26% (MSR 4, 7)로 수행시간이 단축된 것을 확인할 수 있다. 또한, LS의 값이 커질수록 전체 실행 시간이 커짐과 동시에 OPT1을 적용하였을 때 시간 효율성도 증가한 것을 확인할 수 있다. OPT1과 OPT1&2를 비교하면 OPT1&2가 모든 경우에 대해 수행 시간이 단축되었지만 최저 99.43% (MSR 6, 5)에서 최대 91.23% (MSR 4, 5)로 큰 차이를 보이지는 않았다.

앞서 제 4장에서 진행한 실험의 결과와 마찬가지로 최적화 기법 1과 최적화 기법 2를 순차 적용해 감에 따라 커버리지 및 생성되는 테스트 케이스의 수는 동일하면서 메모리 사용량은 감소하고, 전체 수행 시간도 더 빨라지는 것을 확인하였다. 특히, 이 메모리 사용량 감소로 PU와 LS의 값이 커짐에 따라 OOM 발생으로 테스팅을 수행하지 못하던 MSR 프로그램들 중 다수를 테스팅 할 수 있게 되었다.

제 6 장 맺음말

6.1 요약

이 연구에서는 concolic 테스팅에서 배열의 인덱스가 심볼릭 변수로 사용될 때 발생하는 값 실제화로 인한 분기 커버리지 저하를 해결하고자 하였다. 이를 위해 오픈소스 concolic 테스팅 도구인 CREST가 지원하지 않는 심볼릭 메모리 인덱스 참조 연산을 구현하고, 이 연산으로 테스팅 시 발생하는 메모리 낭비를 막기 위해 참조 연산 수행 시 복사되는 스냅샷들 중 중복된 것들을 제거하고, 이전 스냅샷과 비교해 변경된 원소만 복사 및 저장하는 최적화 기법을 적용하였다. 실험에서 4개의 실제 프로그램을 대상으로 concolic 테스팅을 수행하여 기존 CREST보다 더 높은 커버리지를 달성하고, 최적화 기법을 적용하였을 때 더 빠른 시간 내에 더 적은 메모리를 사용하면서 동일한 커버리지를 달성함을 보였다. 특히, 삼성 NAND 플래시의 디바이스 드라이버 중 하나인 MSR에서 이 최적화 기법을 적용하였을 때 평균 8.60%만큼의 메모리를 사용하고 23.66%의 수행 시간으로 테스팅함을 확인하여 최적화 기법을 적용한 심볼릭 메모리 인덱스 참조 연산이 최적화 기법 미 적용시보다 더 효율적으로 실행됨을 확인하였다.

6.2 향후 연구

심볼릭 메모리 인덱스 참조 연산에 최적화 기법을 적용하여 메모리 사용량이 감소하였지만, 규모가 큰 프로그램(e.g. grep)을 테스팅 하기에는 여전히 메모리 사용량이 많았다. 따라서 규모가 큰 프로그램에서 심볼릭 메모리 인덱스 참조 연산이 적용된 CREST를 사용하기 위한 방법들을 향후 연구로 기술한다.

6.2.1 심볼릭 배열 인덱스 참조 연산 효율성 증가

앞선 최적화 기법을 사용하면 하나의 경로 제약 조건에서 생성되는 중복 스냅샷들을 제거할 수 있다. 하지만, 테스트 케이스를 생성함에 따라 누적되는 경로 제약 조건들 간에 존재하는 중복된 요소들은 제거하지 못한다. 테스트 케이스를 생성하던 중 OOM이 발생하는 이유는 경로 제약 조건들이 누적되면서 메모리 사용량이 점점 늘어나기 때문이므로 경로 제약 조건들 간에 존재하는 중복을 제거하여 메모리 사용량 측면에서 효율성을 증가시킬 것이다.

6.2.2 심볼릭 배열 인덱스 참조 연산의 사용자 선택

규모가 큰 프로그램(e.g. grep)에서 심볼릭 배열 인덱스 참조 연산을 사용할 때 메모리 사용

량이 커지는 원인으로 프로그램 내에 존재하는 배열들 중 심볼릭 배열 인덱스 참조 연산을 적용할 필요가 없는 배열까지 심볼릭 배열로 두는 것이 있다. 이것은 concolic 테스팅을 통한 동적 실행으로는 미래에 특정 배열에 심볼릭 배열 인덱스 참조 연산이 사용될지 알 수 없기 때문으로 사용자가 심볼릭 변수를 지정해주는 것과 같이 심볼릭 배열 인덱스 참조 연산을 사용할 배열을 지정하여 더 큰 규모의 프로그램에서 이 연산을 사용할 수 있도록 할 것이다.

참 고 문 헌

[1] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, *Proceedings of the Joint 10th European Software Engineering Conference, ESEC, and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE-13*, 2005, pp. 263–272.

[2] P. Godefroid, N. Karlund, and K. Sen. DART: Directed automated random testing. *In Proc. of ACM SIGPLAN on Programming Language Design and Implementation (PLDI)*, 2005.

[3] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. *In Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, August 2005

[4] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. *In Proc. of Int. Conf. on Computer Aided Verification (CAV)*, 2006.

[5] C. Cadar, V. Ganesh, et al. EXE: a system for automatically generating inputs of death using symbolic execution, *Proceedings of the ACM Conference on Computer and Communications Security*, 2006.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *In Proc. of USENIX Conf. on Operating System Design and Implementation (OSDI)*, 2008.

[7] N. Tillmann, and J. Halleux. Pex-white box test generation for .NET, *Proceedings of 2nd International Conference on Tests and Proofs*, 2008, pp. 134–153.

[8] P. Godefroid, M. Levin, and D. Molnar. SAGE: whitebox fuzzing for security testing. *Queue 10.1* (2012): 20.

[9] M. Kim, Y. Kim, and Y. Choi. Concolic testing of the multi-sector read operation for flash storage platform software. *Formal Aspects of Computing (FAC)*, 24(2), 2012.

[10] M. Kim, Y. Kim, and Y. Jang. Industrial application of concolic testing on embedded software: Case studies. *In Proc. of Int. Conf. on Software Testing, Verification and Validation (ICST)*, 2012.

[11] Y. Kim, M. Kim, Y. Kim, and Y. Jang. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. *In Proc. of Int. Conf. on International Conference on Software Engineering (ICSE) SEIP track*, 2012.

[12] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim. Automated unit testing of large industrial embedded software using concolic testing. *In Proc. of Int. Conf. on Automated Software Engineering (ASE) experience track*, 2013.

[13] B. Elkarablieh, P. Godefroid, and M. Levin, Precise pointer reasoning for dynamic test generation, *Proceedings of International Conference on Software Testing and Analysis*, 2009, pp. 129–140.

[14] J. Burnim. CREST - automatic test generation tool for C. `https://github.com/jburnim/crest`

[15] K. Sen. Concolic testing. *In Proc. of Int. Conf. on Automated Software Engineering (ASE)*, 2007.

[16] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing, *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[17] L. Moura and N. Bjorner. Z3: An efficient SMT solver. *In Proc. of Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[18] Z3, an SMT solver. `https://github.com/Z3Prover/z3`

[19] Yices, an SMT solver. `http://yices.csl.sri.com/`

[20] MathSAT, an SMT solver. `http://mathsat.fbk.eu/`

[21] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: intermediate language and tools for analysis and transformation of C programs, *Proceedings of Conference on Compiler Construction*, 2002, pp. 213–228.

[22] CIL, C intermediate language. `https://people.eecs.berkeley.edu/~necula/cil/`

[23] D. Kroening, and O. Strichman. Decision Procedures. Springer, Book ch 7, p. 171-179

[24] V. Ganesh, Vijay, and D. Dill. A decision procedure for bit-vectors and arrays. *Computer Aided Verification. Springer Berlin Heidelberg*, 2007.

[25] C. Cader, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice - preliminary assessment. *In Proc. of Int. Conf. on Software Engineering (ICSE)*, 2011.

[26] T. Chen, X. Zhang, et al. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems 29.7*, 2013. 1758-1773.

[27] J. Yang, C. Sar, et al. Automatically generating malicious disks using symbolic execution. *In: Security and Privacy, 2006 IEEE Symposium on. IEEE*, 2006. p. 15 pp.-257.

[28] GNU glibc version 2.23. `https://www.gnu.org/s/libc/`

사 사

석사 과정 동안 저의 연구를 도와주시고 응원해주신 모든 분들께 이 지면을 빌어 감사의 말씀을 전하고자 합니다. 먼저 오늘의 제가 있을 수 있도록 사랑으로 키워 주신 가족들에게 감사드립니다. 매일같이 전화 주셔서 안부를 물어보시고 걱정해주신 부모님의 말씀과 항상 뒤에서 응원해준 누나의 격려가 힘들 때 큰 힘이 되었습니다.

저에게 많은 가르침을 주신 김문주 교수님께 감사 드립니다. 항상 논리적인 사고를 할 수 있도록 지도해주시고, 항상 객관적인 판단을 내려주시며 때로는 쓴 소리 마다 않고 해주시고, 저의 반복되는 실수에도 포기하지 않고 끝까지 지도해주신 교수님의 인내심과 배려 덕분에 일취월장할 수 있었습니다.

저의 연구에 많은 도움을 주신 김윤호 박사과정에게 감사합니다. 제가 여러 가지 어려움에 부딪힐 때마다 정확하고 올바른 방향으로 해결할 수 있도록 도움의 손길을 주시고, 이해하기 쉬운 발표 자료를 만들 수 있도록 항상 좋은 피드백을 주신 덕분에 매주 제 능력보다 한 발짝 더 빠르게 나아갈 수 있었습니다. Concolic 테스팅에 대해서도 많은 것들을 배울 수 있었습니다. 또한 연구실 옆자리에서 아낌없이 조언해주신 홍신 박사님, 앞서 제가 가야 할 길이 어떤 것인지를 알려준 박용배, 곽태훈 졸업생께도 감사 드립니다. 연구실에서 함께 공부하였던 전이루, 한충우, 김현우, 양웅규 석사과정에게도 감사 드립니다. 지칠 때 함께 한 이야기들이 힘을 내는데 많은 도움을 주었습니다.

2년간 함께 석사 생활을 한 동기들에게 감사 드립니다. 동기들을 통해서 전혀 다른 관점으로 문제를 바라볼 수 있었고, 동기들의 연구 분야를 보며 전산학 전반에 대해 더욱 깊은 관심을 가질 수 있었습니다. 특히 힘든 시간 함께 하면서도 항상 대화 상대가 되어주고 많은 도움을 준 이원영 형, 박준영 형, 이근홍, 서민관 형, 한영현, 권영기에게 감사 드립니다.

마지막으로 제 투정에도 항상 격려하고 기운을 북돋아 준 ZeroPage 분들과 대학교 및 고등학교 친구들, 어려운 시간 견딜 수 있게 항상 응원해준 이준희에게도 감사 드리고, 여기에 적지 못한 많은 분들께도 감사 드립니다.

여러분께서 도와주신 만큼, 저의 이 작은 결실이 다른 사람들에게 조금이나마 도움이 되기를 기원합니다.

약 력

이 름: 김 태 진
생 년 월 일: 1992년 7월 31일
주 소: 대전시 유성구 궁동 406-15번지
메 일 주 소: taejin@kaist.ac.kr

학 력

2008. 3. – 2011. 2.	양정고등학교
2011. 3. – 2014. 8.	중앙대학교 컴퓨터공학부 학사 (B.S.)
2014. 9. – 2016. 8.	KAIST 전산학부 석사 (M.S.)

경 력 및 수 상 이 력

2015. 1. – 2015. 12.	KAIST 전산학부 석사 대표
2015. 12. 19	한국정보과학회 동계학술대회 우수논문상

학 회 활 동

1. **T. Kim**, M.Kim, H.Lee, H.Jang, M.Park, Detecting Integer Promotion Bugs with Embedded Software using Static Analysis Technique, *Korean Institute of Information Scientists and Engineers*, page. 467-469, Dec 17-19, 2016. (Best paper award)

연 구 업 적

1. Y.Kim, **T.Kim**, M.Kim, H.Lee, H.Jang, M.Park, Effective Integer Promotion Bug Detection Technique for Embedded Software, *Journal of KIISE: Software and Applications*, Vol 40, Num 6, Jun, 2016.