J. Korean Soc. Aeronaut. Space Sci. 48(7), 547-554(2020)
DOI:https://doi.org/10.5139/JKSAS.2020.48.7.547
ISSN 1225-1348(print), 2287-6871(online)

항공기 건전성 관리시스템용 원자성 위배 자율 수리 소프트웨어 기법

최으뜸[1], 이동수[2], 전용기[3], 이성진[4]

On-the-fly Atomicity Violation Repairing Technique for Airborne Health Management Systems

Eu-Teum Choi[1], Dong-Su Lee[2], Yong-Kee Jun[3] and Seongjin Lee[4]
Department of Aerospace Software Engineering, Gyeongsang National University[1,3,4]
Korea Aerospace Industries, LTD.[2]

ABSTRACT

Airborne health management system prevents functional failure caused by errors or faults in the airborne software. On-the-fly repairing atomicity violations (AV) in an ARINC-653 concurrent software is critical for guaranteeing correctness of execution of the software. This paper proposes Repairing-AV which efficiently repairs atomicity violations. The Repairing-AV can diagnose and prevent an error on-the-fly by utilizing the training results of the software and controls access to the shared variable of the thread where the error occurred. The evaluation of the Repairing-AV measures the time overhead by applying the previous work and the Repairing-AV to five synthesis programs containing the atomicity violation. As the result of evaluation, the Repairing-AV constantly shows about 1.4x time overhead regardless of count of shared variable access.

초 록

항공기 건전성 관리시스템은 항공기 소프트웨어에서 발생한 오류 또는 결함으로 인해 항공기의 기능이 실패되는 것을 방지한다. ARINC-653의 병행프로그램에서 발생하는 원자성 위배의 자율 수리는 프로그램의 정상적인 실행을 보장하기 때문에 중요하다. 본 논문은 프로그램 실험 결과를 활용하여 수행 중에 원자성 위배를 예측하고 주요 관련 접근 사건을 지연시켜 수리하는 기법인 Repairing-AV를 제시한다. 실세계 소프트웨어에서 발생한 5가지 원자성 위배 패턴을 포함하는 합성 프로그램에 기존 기법과 Repairing-AV을 적용하여 수리 시간 오버헤드를 비교하였다. 실험 결과 Repairing-AV는 공유변수 접근 횟수와 관계없이 평균 1.4배의 일정한 시간 오버헤드를 가짐을 확인하였다.

Key Words : Health Management System(건전성 관리시스템), Airborne Software(항공기 소프트웨어), Atomicity Violations(원자성 위배), On-the-fly Repairing(자율적 수리)

Ⅰ. 서 론

항공전자 기술의 발달로 인해 항공기의 기능을 소프트웨어로 구현하는 비중이 급격하게 증가하고 있다. 소프트웨어로 구현한 기능의 비중이 1960년에 생산된 F-4의 경우 8%에 불과하였지만 2007년에 생산된 F-35의 경우 약 90%의 기능이 소프트웨어로 구현되었다[1]. 항공기에서 소프트웨어로 구현된 기능의 비중 증가는 소프트웨어의 적용 범위, 규모 및 복잡도를 증가시키기 때문에 항공기 시스템에서 발생 가

† Received : March 26, 2020 Revised : June 15, 2020 Accepted : June 16, 2020
[1] Ph.D. Student, [2] Engineer, [3] Professor, [4] Assistant Professor
[3,4] Corresponding author, E-mail : jun@gnu.ac.kr, insight@gnu.ac.kr, [4] ORCID 0000-0003-0760-1880
© 2020 The Korean Society for Aeronautical and Space Sciences

능한 잠재적인 오류도 크게 증가시킨다.

소프트웨어 오류 중 동시성 오류[2-8]는 재현이 어려워 디버깅(debugging)[4-8]이 힘든 오류로 잘 알려져 있다. 실세계(real world) 응용 소프트웨어에서 발생한 동시성 오류 중 70% 이상이 원자성 위배이다[3]. 프로그램의 일부 영역이 원자적으로 실행되도록 프로그래머가 의도적으로 설정한 원자성 영역 내에서 변수의 값이 의도와 다르게 변하는 경우를 원자성 위배라고 한다[3]. 프로그램의 모든 실행 상황을 고려하기 어렵기 때문에 코드에는 늘 원자성 위배의 위험이 잠재한다.

항공기 시스템은 오류를 관리하기 위해 건전성 관리 시스템(Health Management System, HMS)[9-15]을 적용하고 있다. HMS는 결함을 감시, 진단, 격리, 수리의 대상으로 삼아 항공기 시스템에서 발생한 결함이 전체 또는 주변 시스템에 영향을 미치지 않게 지속적으로 관리하는 것을 목적으로 한다[15]. 항공기 소프트웨어에서 원자성 위배의 건전성을 관리하는 연구에는 Ha et al.[16]과 Tchamgoue et al.[17]이 있다. 이 두 연구는 ARINC-653 소프트웨어[15]에서 발생하는 접근 사건을 감시하고 스레드의 락 사용 규칙 위배를 검사하여 원자성 위배의 발생을 진단(diagnosis)한다. 원자성 위배가 발생하면 lock 삽입[16] 또는 지연(stalling)[17] 방법으로 조치(treatment)한다.

이 두 논문 모두 원자성 위배 진단을 위해 접근 사건의 수행 시마다 생성 또는 유지되는 접근 역사를 실행 중 접근 정보와 비교하여 원자성 위배를 진단하는 프로토콜[4]을 사용한다. 하지만, 이 프로토콜은 원자성 위배 발생 진단을 접근 사건 수행 때마다 O(T)의 시간복잡도를 가진다. 여기서 T는 최대병렬성으로 프로그램이 동시에 수행 가능한 스레드의 최대 개수를 의미한다. 고성능 항공기의 소프트웨어 복잡도가 높아지는 추세로 볼 때 O(T) 시간의 진단 방식은 실시간 시스템에서 사용하기에는 효율적이지 못하다.

본 연구에서는 사전시험을 통해 생성된 올바른 인터리빙 정보를 기반으로 실행 중에 발생하는 원자성 위배를 자율적으로 수리하는 도구인 Repairing-AV (Repairing Atomicity Violations)을 제시한다. Repairing-AV의 구조는 각 스레드에 공유 자원의 접근을 감시하여 원자성 위배를 진단하는 모듈과 지연 방법으로 인터리빙이 올바로 되도록 조치하는 모듈로 구성된다. 원자성 위배 진단 모듈은 실행 중 접근 정보와 사전시험한 접근 정보를 이용하여 두 정보가 다르면 원자성 위배로 진단한다. 원자성 위배 수리 모듈은 조건 변수(condition variable)[17]을 사용한다. 원자성 위배가 진단된 스레드의 이벤트를 wait()를 사용하여 지연시키고, 올바른 순서가 진단되면 signal()을 사용하여 다시 실행시킨다.

본 논문의 2장은 항공기 건전성 관리시스템, 원자성 위배, 그리고 원자성 위배의 건전성 관리를 설명한다. 3장은 제안하는 기법의 구조와 각 모듈의 상세 설계를 설명한다. 4장은 Repairing-AV의 구현 이슈를 설명하고 실험을 통해 기법의 성능을 분석한다. 5장에서 논문의 결론을 제시한다.

II. 연구 배경

이 장에서는 항공기의 건전성 관리시스템과 동시성 오류의 한 종류인 원자성 위배에서 설명한다. 그리고, 항공기의 건전성 관리시스템에서 원자성 위배를 수리하는 기존의 연구를 소개하고 문제점을 분석한다.

2.1 항공기의 건전성 관리시스템

항공기의 건전성 관리 시스템[9-15]은 항공기 시스템 기능의 실패를 방지하는 것이 목적이다. 운용 중에 발생하는 오류는 감시(monitoring), 진단(diagnosis), 및 조치(treatment)의 대상이며 HMS는 오류를 관리한다. 감시 단계에서는 시스템의 수행 중에 발생하는 오류 및 관련 이벤트의 수행 정보를 수집한다. 진단 단계에서는 감시를 통해 수집된 정보를 이용하여 발생한 오류의 유형을 분석하거나 잠재적 오류의 발생을 예측한다[15].

HMS는 온라인 또는 오프라인 방법으로 발생한 오류를 조치한다[9-15]. 온라인 방법은 프로그램 실행 중에 발생한 오류로 인한 영향을 줄이기 위해 두 가지 복구 방법을 통해 오류가 없는 프로그램 수행으로 복구한다. 복구 방법은 backward recovery와 forward recovery로 구분된다. Backward recovery는 오류가 발생한 시점 전 마지막으로 시스템 결함 없이 동작한 상태가 저장된 checkpoint에서 다시 실행하는 방법이다. 이 방법은 오류를 완벽히 수리할 수 있지만, 시간 및 공간 오버헤드가 매우 큰 단점이 있다. Forward recovery는 오류가 발생하기 전에 동기화를 삽입하거나 지연(stalling)을 통해 조치한다. 따라서, backward recovery보다 낮은 시간 및 공간 복구 오버헤드를 가진다. 오프라인 방법은 프로그램 실행 중에 오류 정보를 저장하여 코드 상에서 결함을 제거한다.

추가적으로 HMS는 탐지된 결함이 전체 시스템 또는 다른 하위 시스템에 전파되는 것을 방지하기 위해 격리해서 시스템의 안전성을 높인다. 건전성 관리 시스템은 개별 컴포넌트에서 전체 시스템에 이르는 다양한 수준의 건전성을 관리해야 한다. 이를 위해 ARINC 653 기반의 항공전자 시스템은 파티션, 모듈, 시스템 수준에서 특정 오류 조건에 매핑된 콜백 기능을 사용하여 복구 작업을 지원하는 건전성 모니터링 서비스를 제공한다.

2.2 원자성 위배

원자성 위배[2-8]는 병렬 프로그램(parallel program)에서 개발자가 원자적으로 실행되도록 의도한 원자성 영역(atomic region)이 의도와 다르게 실행되는 오류이다. 공유 변수에 접근하는 두 개 이상의 스레드가 적절한 동기화 없이 적어도 한 번 이상의 쓰기 명령을 실행하면 원자성 위배가 발생할 수 있다[2, 3]. Fig. 1은 Apache의 메모리 캐시 모듈의 코드 중 일부로 원자성 위배가 발생한 실제 사례이다.

decrement_refcount()는 서버가 메모리 풀을 정리할 때 호출하는 함수로 객체의 참조 카운터가 0이 되면 cleanup_cache_object()를 호출하여 캐시 객체를 정리(clean up)한다. 이때, apr_atomic_dec()는 객체의 참조 카운터를 감소시키고 변경된 카운터를 리턴한다. 개발자는 하나의 스레드 상에서 원자적인 실행을 의도하였다. 하지만 Fig. 1과 같은 수행으로 인해 원자성 위배가 발생할 수 있다. 만약 refcount가 2인 경우 decrement_refcount()가 두 번 실행되어 refcount은 0이 된다. 그 후 thread 2의 cleanup_cache_object()가 정상적으로 객체를 정리한다. 하지만 thread 1의 cleanup_cache_object()가 정리된 객체를 다시 정리하게 되어 NULL point exception이 발생한다.

병렬 스레드 또는 동시성 프로그램의 개발에 대한 충분한 지식과 이해가 있는 개발자라 하더라도 프로그램의 모든 실행 상황을 개발자가 고려할 수 없기 때문에 코드에 잠재적 원자성 위배가 존재할 수 있다. 콜롬비아 우주 왕복선과 NASA의 Advanced Fighter Technology Integration (AFTI)-F16의 사례를 보면 원자성 위배와 유사한 경합 상태(race conditions)로 인해 이륙이 지연되었다[18]. Apache, Mysql 등과 같이 활용도가 매우 높은 오픈소스 프로그램에서도 원자성 위배[3]는 해결하기 어려운 문제를 만들어 낸다.

일반적으로, 소프트웨어의 결함을 제거하기 위한 소프트웨어 검증 및 확인 과정에서 먼저 정적·동적 분석 도구를 사용하여 소스 코드의 결함 위치를 탐지한다. 하지만, 원자성 위배와 같은 동시성 오류를 탐지 및 디버깅을 하는 것은 전문적인 지식과 관련 도구의 부족으로 인해 매우 어렵고 귀찮은 일로 잘 알려져 있다. 일반적인 순차 프로그램에서 발생하는

Thread 1	Thread 2

decrement_refcount(...)
{
  apr_atomic_dec(**&obj->refcount**);
 	
  if(!**obj->refcount**){
  cleanup_cache_object(obj);
  }
}
	decrement_refcount(...)
	{
	
	  apr_atomic_dec(**&obj->refcount**);
	  if(!**obj->refcount**){
	  cleanup_cache_object(obj);
	
	  }
	}

Fig. 1. An Example of Atomicity Violations (Bug Report #21287) in Apache [25]

오류(memory leak 등)와 병렬 프로그램에서 발생하는 동시성 오류를 비교했을 때, 동시성 오류를 수정하기 위해 17% 이상의 추가적인 개발 노력(추가 개발자 투입 등)이 필요하다[3]. 또한, 동시성 오류를 수정하기 위해서는 일반적인 순차 프로그램에서 발생하는 오류에 비해 46% 더 많은 파일에 영향이 있었고, 72% 이상 더 많은 패치가 생성되었다. 그럼에도 불구하고, 동시성 오류를 수정한 코드 중 39%는 부정확한 수정으로 판명되었다[3].

2.3 원자성 위배의 자율 수리 기법

원자성 위배의 자율 수리 기법은 진단 방법에 따라 탐지 프로토콜을 적용한 기법, 올바른 인터리빙 기반으로 진단하는 기법으로 구분할 수 있다. 최근 연구들은 사전시험을 통해 얻은 올바른 인터리빙을 기반으로 원자성 위배를 수리하는 기법이 연구되고 있으며 진단의 정확도가 다른 기법보다 높다.

ARINC 653 기반의 항공기 소프트웨어를 위한 원자성 위배의 건전성 관리 도구[18,19]는 스레드의 접근 사건 정보를 감시하여 록킹 규칙(locking principle)의 위배를 감시한다. 오류 진단과 원자성 위배를 해결하기 위해 락을 삽입하거나 스레드의 접근을 지연시키는 방법을 사용한다. 또 다른 연구에서는 원자성 위배의 탐지를 위해 접근 사건의 병렬성 및 순서 관계 비교를 위한 Labeling 기법[5, 6]을 사용한다. 그리고, 스레드의 록킹 메커니즘과 공유변수의 접근 사건 정보를 비교하기 위해 접근 역사 관리프로토콜[4,6]을 적용한다.

임의의 두 스레드 T_i, T_j가 있을 때 Label을 사용하여 스레드 및 접근 사건의 동시성 관계를 식별하는 예를 들어보자. 각 스레드가 사용하는 영역의 시작과 끝에 각각 정수 α와 β를 할당한다. T_i의 영역 $<\alpha_i, \beta_i>$와 T_j의 영역 $<\alpha_j, \beta_j>$가 겹치는지를 비교하여 스레드의 유일성과 동시성을 비교 및 관리한다. 스레드의 동시성은 다음과 같은 기준으로 판별한다.

- 두 스레드 T_i와 T_j의 영역이 만약 $\alpha_i<\beta_j$ 그리고 $\alpha_j>\beta_i$이면, T_i와 T_j는 동시성 관계이다.
- 아니면, T_i 는 T_j 순서 관계이다.

원자성 위배 탐지 프로토콜은 동시성 오류가 존재한다면 적어도 하나의 결함은 반드시 탐지한다. 이를 위해 접근 사건마다 <label, locks>의 쌍을 접근 역사에 기록하고 발생한 접근 사건의 유형을 분류한다. 접근 사건의 유형은 접근 사건 종류와 원자성 영역 내에 있는지에 따라 Read, Write, CS (Critical Section)-Read, CS (Critical Section)-Write로 구분한다.

진단을 위해 접근 사건의 유형에 따라 비교하는 접근 사건 대상을 규정하는 정책(policy)을 사용한다. Read 명령이 들어오면 접근 역사에 기록된 Write와 CS-Write 집합과 비교하여 원자성 위배를 진단한다. Write는 접근 역사에 기록된 Read, Write, CS-Read,

CS-Write 집합과 비교하고, CS-Read는 Write와 CS-Write 집합과, CS-Write는 Read와 Write 집합과 비교하여 원자성 위배를 진단한다.

기존 원자성 위배 진단 방법은 접근 사건이 발생할 때마다 유지해야 하는 접근 역사의 시간 및 공간 복잡도는 최대병렬성 T에 의존하는 O(T)가 된다. 최근 멀티스레드를 사용하는 실세계 프로그램은 최대 100만 개의 스레드를 사용하고 있으며[19-21], 항공기의 항법 소프트웨어는 10억 개 규모의 명령어를 사용하고 있다. 이러한 이유로 복잡도가 최대 병렬성에 의존하는 건전성 관리 기반 자율 수리 도구는 소프트웨어 복잡도가 높고 실시간성 보장이 중요한 항공기 소프트웨어에 적용하는 것은 비효율적이다.

III. Repairing Atomicity Violations

이 장에서는 사전시험을 통해 생성된 올바른 인터리빙 정보를 기반으로 원자성 위배를 자율적으로 수리하는 Repairing Atomicity Violations (Repairing-AV)을 소개한다. 먼저 Repairing-AV의 구조를 소개하고 두 핵심 모듈인 진단과 조치 엔진을 설명한다.

Repairing-AV는 사전 시험한 접근 정보를 통해 프로그램 실행 중에 자율적으로 원자성 위배를 진단하는 AI (Anticipate Invariant)[22]에 기반하고 있으며 입력과 출력정보는 다음과 같다.

- 입력: 실행 중 접근 정보, 사전 시험한 접근 정보
- 출력: 스레드 멈춤 또는 스레드 재개

입력에서 실행 중 접근 정보는 공유변수의 접근 사건 및 주소와 스레드로 구성된다. 사전 시험한 접근 정보는 정적 명령어(S_x)의 집합으로 구성된다. 수행 중 수집된 동적 명령어(D_x)를 기반으로 다음 조건을 만족하는 정적 명령어를 수집한다.

- D_x의 접근과 동일한 주소를 가진 동적 명령어
- D_x가 속하지 않은 다른 스레드에 접근하는 동적 명령어
- D_x의 수행 직전에 접근하는 동적 명령어

Repairing-AV의 구조는 Fig. 2와 같다. 각 스레드에 공유 자원의 접근을 감시하여 원자성 위배의 발생을 진단하는 Diagnosis Engine(AV-DE)과 지연(stall)을 삽입하여 인터리빙이 올바로 되도록 조치하는 Treatment Engine (AV-TE)으로 구성된다. Repairing-AV는 오류를 수리하기 위한 세 가지의 실행 경로를 가진다.

(1) 멈춤 수행 경로: ARINC-653 응용프로그램 → AV-DE → pthread TM
(2) 재개 수행 경로: ARINC-653 응용프로그램 → AV-DE → ARINC-653 HM → AV-TE → pthread TM
(3) 수리 활동 없음

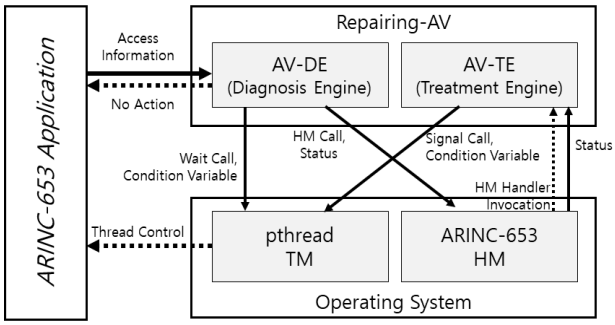

Fig. 2. Architecture of Repairing-AV

멈춤 수행 경로는 오류가 진단되면 ARINC-653 응용프로그램에서 오류가 발생한 스레드를 멈춘다. 재개 수행 경로는 오류 때문에 멈춘 스레드가 다른 스레드가 수행되어도 정상적으로 수행할 수 있다고 진단되면 멈춘 스레드를 재개시킨다. 마지막으로 (3)번 경로는 정상적인 수행으로 AV-DE에서 진단만 수행되고 ARINC- 653 응용프로그램의 수행에 영향을 주지 않는다.

Diagnosis Engine (AV-DE)은 사전 시험 단계에서 생성된 프로그램 시험 결과와 수행 정보를 비교하여 원자성 위배를 진단하고 ARINC-653 HM에 오류를 보고한다. AV-DE 알고리즘은 Fig. 3에, 오류 진단을 위한 AI 알고리즘은 Fig. 4에 나타내었다. AV_DE는 실행 중 접근 정보와 사전 시험한 접근 정보를 입력받아 AI를 호출한다. AI는 두 접근 정보를 비교하여 원자성 위배를 진단하고 결과를 리턴한다. 여기서 sanitized-bset.bin에는 사전 시험한 접근 정보가 저장되어 있다. 진단의 결과가 오류로 판정되었다면 AV-DE는 ARINC-653 HM을 호출하여 원자성 위배가 발생하였다고 알려준다.

```
AV-DE algorithm:
procedure AV-DE(access_info)
  bool isAV = call AI(access_info)
  if isAV = true then
    call RAISE_APPLICATION_ERROR()
  end if
end procedure
```

Fig. 3. The Algorithm of AV-DE

```
AI algorithm:
procedure AI(access_info)
  array test_info = getFile(sanitized-bset.bin);
  while test_info[i] != empty do
    if test_info[i] != access_info then
      return true
    end if
  increase i
  end while
end procedure
```

Fig. 4. The Algorithm of AI

```
AV-TE algorithm:
procedure AV-TE()
  if treatment_state == “normal” then
    call pthread_cond_wait();
    treatment_state = “wait”;
  else
    call pthread_cond_signal();
    treatment_state = “normal”;
  end if
end procedure
```

Fig. 5. The Algorithm of AV-TE

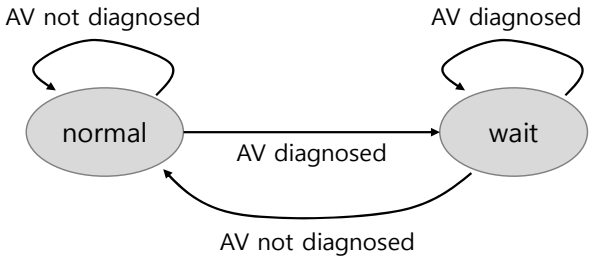

Fig. 6. The State Machine for AV-TE

AV-DE에서 원자성 위배가 진단되었으면 Treatment Engine (AV-TE)는 조건 변수를 이용하여 해당 스레드를 멈추었다가 이후에 진단 모듈이 정상 실행으로 진단하면 멈춘 스레드를 재개시킨다. AV-TE의 알고리즘은 Fig. 5에 나타내었다. AV-TE는 현재의 수리 상태를 확인하기 위해 상태 머신(state machine)을 이용한다. 상태 머신은 Fig. 6과 같이 동작한다. 최초 프로그램이 실행되었을 때의 상태는 normal이다. 수행 중에 AV-DE가 원자성 위배를 진단하여 AV-TE가 wait()를 수행하면 wait 상태로 전이한다. Wait 상태에서 AV-DE가 오류를 진단하더라도 AV-TE는 조치를 시도하지 않는다. 그리고, AV-DE가 원자성 위배가 없다고(수리되었음) 진단하면, AV-DE는 signal()을 통해 멈춘 스레드를 재개시키고 normal 상태로 전이한다.

ARINC-653 응용프로그램에서 원자성 위배가 발생하지 않으면 Repairing-AV는 어떤 조치도 하지 않는다. 그리고, 원자성 위배가 발생하면 ARINC-653 운영체제가 오류에 영향받을 수 있는 파티션 또는 프로그램들을 격리할 수 있도록 ARINC-653 HM에 오류를 보고한다. 그 후, AV-TE가 스레드를 제어하여 안전하게 원자성 위배를 수리한다.

Ⅳ. 구현 및 실험

이 장에서는 Repairing-AV의 구현과 성능 평가를 위한 실험 환경과 실험의 결과를 설명한다. 구현은 운영 환경, 개발 환경, 소스 코드, 설정으로 설명한다. 구현된 Repairing-AV의 성능을 평가하기 위한 시험프로그램과 실험의 결과를 설명한다.

4.1 구현

운영 환경과 개발 환경: 하드웨어는 Intel Xeon E5-2650 2.30GHz CPU, 64GB 메모리를 사용했다. Ubuntu 14.04 LTS-64bit 운영체제에 실시간 커널인 Linux 4.14.139-rt66을 적용하여 ARINC-653 시뮬레이터인 SIMA (simulated integrated modular avionics)를 설치하였다. 개발 환경은 Microsoft Visual Studio Code 1.37.1 개발 도구, GCC 5.4.0 컴파일러, LLVM 6.0.0을 이용하여 코드를 삽입하였다.

시험 프로그램: 구현된 Repairing-AV가 정상적으로 동작하는지 확인하기 위해 원자성 위배를 포함한 시험 프로그램 Fig. 7(a)와 같이 구현하였다. 이 시험 프로그램의 공유변수의 초깃값은 0이다. 프로그램의 실행 결과로 2를 기대했지만, Fig. 7(b)에 나타난 것처럼 1을 결과로 얻었다. 수행 순서를 확인해보면 thread 1의 읽기 접근 다음 thread 1의 쓰기 접근이 실행되어야 하는데 thread 2의 읽기 접근이 실행되어 원자성 위배가 발생함을 확인할 수 있다.

Repairing-AV의 검증: 시험 프로그램에 Repairing-AV를 적용하여 사전 시험 단계와 운용 단계로 수행하였다. 먼저, 시험 프로그램의 사전 시험을 통해 접근 정보를 생성하였다. Fig. 8은 생성된 사전 시험한 접근 정보를 보여준다. Fig. 8에서 (a)는 정적 명령어의 아이디, (b)는 정적 명령어가 수행했던 이전 접근의 횟수, 그리고 (c)는 이전 접근의 아이디의 목록을 나타낸다.

```
35  int sv = 0;
36
37  void *thread1(void *v) {
38      int rv = sv; //read
39      sv = rv + 1; //write
40      return NULL;
41  }
        …
43  void repairing_server()
44  {
        …
56      pthread_mutex_lock(&mlock);
57      rv = sv; //read
58      sv = rv + 1; //write
59      pthread_mutex_unlock(&mlock);
        …
```

```
init sv = 0
thread1 Read (23606)
thread2 Read (23607)
thread1 Write (23606)
thread2 Write (23607)
Result: sv = 1
```

(a) Code (b) Result

Fig. 7. Information of Testing Program

	(a)	(b)	(c)
1	11	2	0 20
2	12	2	0 20
3	17	2	0 12
4	20	2	0 12
5	24	2	12 20

(a) Access ID
(b) Number of Previous Accesses
(c) Previous Access IDs

Fig. 8. Tested Access Information

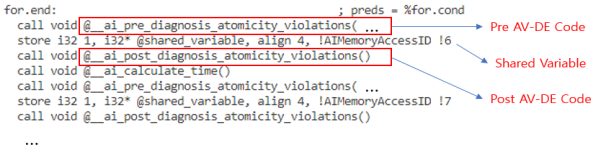

Fig. 9. The Results of Instrumentation

```
init sv = 0
call preAV-DE (24123)
thread1 Read (24123)
call postAV-DE (24123)
                                call preAV-DE (24124)
                                call ARINC-653 HM (24124)
                                call AV-TE: wait (24124)
call preAV-DE (24123)
thread1 Write (24123)
call postAV-DE (24123)
call ARINC-653 HM (24123)
call AV-TE: signal (24123)
                                thread2 Read (24124)
                                call postAV-DE (24124)
                                call preAV-DE (24124)
                                thread2 Write (24124)
                                call postAV-DE (24124)
Result: sv = 2
```

Fig. 10. The Result on Repairing-AV

시험 프로그램에 Repairing-AV가 올바르게 적용되었는지 바이너리 코드를 확인한 결과는 Fig. 9와 같다. 공유변수가 실행되기 전후로 AV-DE 코드가 정상적으로 삽입되었다. 공유변수가 실행되기 전의 AV-DE 코드는 원자성 위배가 진단되었을 때 실행을 멈추기 위한 wait() 코드이고, 후의 AV-DE 코드는 원자성 위배가 수리되면 스레드를 재개하기 위한 signal() 코드이다.

사전 시험 단계에서 사전 시험한 접근 정보가 생성되고, Repairing-AV를 정상적으로 삽입한 후 운용 단계를 진행하였다. Fig. 10은 자율 수리를 위해 수행된 모듈과 공유변수의 순서를 나타내었다. 실행 결과를 보면 값은 정상적으로 2가 되었다. Repairing-AV에 의해 프로그램 수행 중에 발생한 원자성 위배가 수리되었다.

4.2 실험

Repairing-AV를 평가하기 위해 실세계 소프트웨어에서 발생할 수 있는 5가지 원자성 위배 패턴을 포함하는 합성 프로그램에 기존 연구와 Repairing-AV를 적용하였다. 수리 도구가 적용되지 않은 합성 프로그램의 수행 시간과 두 가지 도구의 수행 시간을 비교하여 효율성을 나타내었다.

합성 프로그램은 실세계 버그 리포트에 보고된 5가지 원자성 위배 패턴을 이용하였다. Table 1은 MySQL[23], Mozilla[24], Apache[25]의 버그 리포트에 보고된 각 패턴별 오류 번호를 나타낸다. 패턴에서 R은 읽기 사건, W는 쓰기 사건을 의미한다. "["와 "]"은 lock과 같은 동기화를 의미하며, 이 괄호 사이의 사건들은 원자적 실행이 보장돼야 한다. "‖"는 동시성 실행을 나타낸다.

Table 1. Patterns of Atomicity Violations

Pattern	Report of Real-World Applications		
	MySQL[23]	Mozilla[24]	Apache[25]
$T_i^{[R-R]} \parallel T_j^{W}$	#644, #3596, #12228	#341323, #224911	–
$T_i^{[W-W]} \parallel T_j^{R}$	#791, #12848, #19938	#52111, #73761, #62269	–
$T_i^{[R-W]} \parallel T_j^{W}$	–	–	–
$T_i^{[W-R]} \parallel T_j^{W}$	#128486	–	–
$T_i^{[R-W]} \parallel T_j^{R-W}$	#56324, #59464	#342577, #270689, #225525	#48735, #21287

Figure 7(a)로 예를 들어보자. 메인 스레드인 repairing_server()는 lock 동기화를 가지고 읽기 접근과 쓰기 접근하고 있으므로 $T_i^{[R-W]}$로 표현할 수 있다. 또 다른 스레드 함수 thread1()은 동기화 없이 읽기 접근과 쓰기 접근하고 있으므로 T_j^{R-W}로 표현할 수 있다. 이 두 스레드는 동시에 실행할 수 있으므로 $T_i^{[R-W]} \parallel T_j^{R-W}$와 같이 표현된다.

합성 프로그램의 구현은 pthread를 사용한 병행 프로그램으로 두 개의 스레드로 구현하였다. 그리고 내포 병렬성과 잠금은 제외하였으며, 단일 공유변수에 대한 접근 사건만 고려하였다. 원자성 위배의 유형은 Table 1에서 설명된 5가지 원자성 위배 패턴만을 고려하였다. 마지막으로 프로그램 수행 중에 최소 10,000에서 최대 1,500,000번 fork()와 join()이 발생하도록 하였다. Fig. 11은 SIMA에서 수행한 합성 프로그램의 실행화면을 나타낸다.

구현된 합성 프로그램을 이용하여 두 가지로 시간 오버헤드를 측정하였다. 첫 번째는 프로그램 내 fork/join의 반복 횟수에 따른 시간 오버헤드의 증가량을 측정하였다. 반복 횟수는 최소 10,000번부터 최대 1,500,000번이 되도록 하였다. 두 번째로 5가지 원자

```
<10 - one                                          <<20 - posix
<--------------------------------------------------<<------------
<                                                  << POSIX
<                                                  << POSIX
< with connected ports: 0                          << POSIX
< mos: 18288                                       << POSIX
< pos: 18292                                       << POSIX
< cmd: 0                                           << POSIX
< err: 0                                           << POSIX
< The process for repairing_server is create       << POSIX
< d: 0000000378100000                              << POSIX
< The process for repairing_server is starte       << POSIX
< d: 0000000378300000                              << POSIX
< ctl_demo_init end point: 0000000378300000        << POSIX
< Error handler is created: 0000000378300000       << POSIX
<                                                  << POSIX
< ctl_err_init end point: 0000000378400000         << POSIX
< start R-R|W synthetic program!!value of sh       << POSIX
< ared variable: 1                                 << POSIX
< execution time: 9.711859                         << POSIX
```

Fig. 11. Execution of Synthetic Program in SIMA

성 위배 패턴을 적용한 합성 프로그램에 기존 연구와 Repairing-AV를 적용하여 시간 오버헤드를 측정하였다. 반복 횟수는 1,000,000번으로 고정하여 총 10번 수행하여 평균값을 산출하였다.

4.3 결과 분석 및 토의

반복 횟수에 따른 시간 오버헤드의 증가량 실험의 결과는 Fig. 12와 같다. Fig. 12에서 합성 프로그램만 실행했을 때 수행 시간이 반복 횟수가 10,000번일 때 0.12초에서 1,000,000일 때 14.7초로 증가하였다. 프로그램 내 반복 횟수가 증가할수록 수행 시간이 증가하는 것은 자연스러운 현상이다. 기존 연구가 적용된 합성 프로그램에서 수행 시간을 합성 프로그램만 실행했을 때와 비교해보면 1,000,000번에서 251%, 1,500,000번에서 275%로 수행 시간이 2.5배 이상 증가함을 확인할 수 있다. 이 결과로 반복 횟수가 증가할수록 시간 오버헤드가 지속적으로 증가한다는 것을 알 수 있다. Repairing-AV가 적용된 합성 프로그램에서 수행 시간을 합성 프로그램만 실행했을 때와 비교해보면 1,000,000번에서 144%, 1,500,000번에서 145%로 수행 시간이 약 1.4배 정도로 유지됨을 확인할 수 있다. 이 결과로 Repairing-AV는 항공기 소프트웨어에서 공유변수 접근 횟수와 관계없이 약 1.4배의 일정한 시간 오버헤드를 가지는 것을 알 수 있다.

5가지 합성 프로그램 패턴에 대한 시간 오버헤드 실험의 결과는 Fig. 13과 같다. Fig. 13에서 합성 프로그램만 실행했을 때 평균 실행 시간은 약 10초가 걸렸다. 기존 연구의 실행 시간은 약 25초 내외로 약 2.5배의 시간 오버헤드가 발생하였다. 각 패턴 중 $T_i^{[R-W]} \parallel T_j^{R-W}$ 패턴에서 가장 높은 수행 시간을 보였다. 이 결과는 $T_i^{[R-W]} \parallel T_j^{R-W}$ 패턴의 접근 사건 횟수가 다른 패턴들보다 반복 횟수만큼 많아지기 때문에 추가된 접근 사건 정보를 생성하고 유지하는 데 소비된 시간 오버헤드이다. Repairing-AV는 약 14초~15초로 수행 시간이 걸렸다. 따라서 시간 오버헤드는 기준 프로그램 대비 약 1.45배이다.

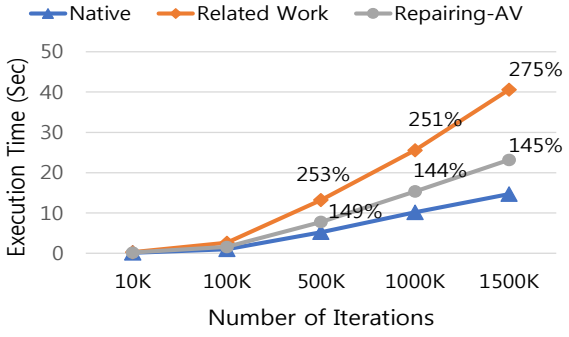

Fig. 12. Result on Iteration Scale of $T_i^{[R-W]} \parallel T_j^{R-W}$ Synthetic Programs

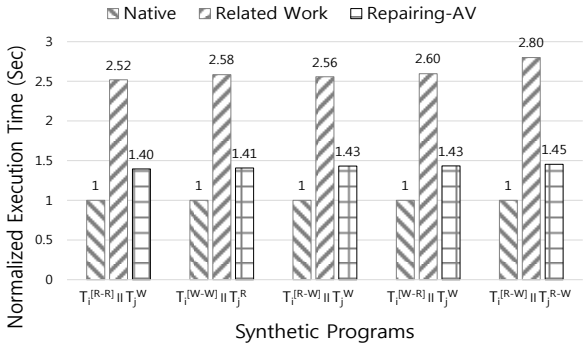

Fig. 13. Result on Type of Synthetic Programs

두 실험 결과를 분석한 결과, Repairing-AV는 접근 사건 수행에 따른 O(1)의 복잡도를 갖는다. Fig. 3과 Fig. 5와 같이 진단 및 조치 알고리즘은 조건문만으로 구성되어 있다. 그리고 Fig. 5와 같이 AI 알고리즘은 저장된 올바른 인터리빙 비교만 수행한다. 그러므로 Repairing-AV는 접근 사건이 증가하여도 수리 오버헤드가 일정하다. 시간에 따른 공유변수 접근이 증가한다고 가정을 한다면, 항공기의 최장 운용 시간이 약 16시간임을 고려해볼 때 기존 연구는 접근 사건 수행 횟수에 따른 시간 오버헤드가 지속적으로 증가하여 항공기 시스템에 적용하기 부적합함을 알 수 있다. 반면, Repairing-AV는 일정한 시간 오버헤드를 가지므로 수리에 필요한 시간 오버헤드를 고려한다면 항공기 시스템에 충분히 적용될 수 있다.

V. 결 론

ARINC-653의 병행 프로그램에서 발생하는 원자성 위배를 항공기 건전성 관리시스템이 자율 수리하는 것은 프로그램의 정상적인 실행을 보장하기 위해 중요하다. 기존 연구의 경우 접근 사건 수행 때마다 생성 및 유지되는 접근 역사를 실행 중 접근 정보와 비교하여 원자성 위배를 진단한다. 이로 인해 접근 사건이 발생할 때마다 진단하기 위한 시간 오버헤드가 발생하게 된다.

본 논문은 프로그램 실험 결과를 활용하여 수행 중에 오류를 예측하고 주요 관련 접근 사건을 지연(stalling)시키는 기법인 Repairing-AV를 제시하였다. Repairing-AV는 공유변수 접근 횟수와 관계없이 평균 1.4배의 일정한 시간 오버헤드를 가짐을 보였다. 따라서 항공기 시스템 설계 시 수리에 필요한 오버헤드를 고려한다면 충분히 항공기 시스템에 적용할 수 있다. 향후 연구로 진단 정확성이 사전 수행 정보에 의존적이기 때문에 사전수행 정보의 정확성을 분석해야 한다.

후 기

이 논문은 2020년도 정부(교육부) 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업(No. NRF-2018R1D1A3B07041838), 산업통상자원부 재원으로 한

국산업기술평가관리원(KEIT)의 지원을 받아 수행된 항공우주부품기술개발사업-수출유망부품및핵심기술개발사업(과제명 : 중소형 항공기급 개방형 항공전자 시스템 아키텍쳐 및 소프트웨어 개발 / 과제고유번호 : 20005378), 그리고 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구(No. 2019R1G1A1100455)의 결과입니다.

References

1) Merendino, T., Latimer, IV, D. T., Hammons, C. B., Falkenthal, D., Capell, P. and Firesmith, D. G., *The Method Framework for Engineering System Architectures*, CRC Press, 2008.

2) Netzer, R. H. and Miller, B. P., "What Are Race Conditions?," *ACM Letters on Programming Languages and Systems (LOPLAS)*, March 1992, pp. 74~88.

3) Lu, S., Park, S. Y., Seo, E. S. and Zhou, Y., "Learning from Mistakes A Comprehensive Study on Real World Concurrency Bug Characteristics," *ACM SIGOPS Operating Systems Review*, March 2008, pp. 329~339.

4) Dinning, A. and Schonberg, E., "Detecting Access Anomalies in Programs with Critical Sections," *Proceedings of the 1991 ACM/ONR workshop on Parallel and Distributed Debugging*, December 1991, pp. 85~96.

5) Jun, Y.-K. and Koh, K., "On-the-fly Detection of Access Anomalies in Nested Parallel Loops," *Proceedings of the 1993 ACM/ONR workshop on Parallel and Distributed Debugging*, December 1993, pp. 107~117.

6) Ha, O.-K., Kuh, I.-B., Tchamgoue, G. M. and Jun, Y.-K., "On-the-fly Detection of Data Races in OpenMP Programs," *Proceedings of the 10th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2012)*, July 2012., pp. 1~10.

7) Ratanaworabhan, P., Burtscher, M., Kirovski, D., Zorn, B., Nagpal, R. and Pattabiraman, K., "Detecting and tolerating asymmetric races," *ACM SIGPLAN Notices*, February 2009, pp. 173~184.

8) Lucia, B. and Ceze, L., "Cooperative empirical failure avoidance for multithreaded programs," *ACM Special Interest Group on Programming Languages Notices*, March 2013, pp. 39~50.

9) Mahadevan, N., Dubey, A. and Karsai, G., "Application of software health management techniques," *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, May 2011, pp. 1~10.

10) Srivastava, A. N. and Schumann, J., "The case for software health management," *2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology*, September 2011 , pp. 3~9.

11) Goldberg, A. and Horvath, G., "Software Fault Protection with ARINC 653," *2007 IEEE Aerospace Conference*, 2007 IEEE, March 2007, pp. 1~11.

12) Ofsthun, S., "Integrated vehicle health management for aerospace platforms," *IEEE instrumentation & measurement magazine*, Vol. 5, No 3, September 2002, pp. 21~24.

13) Scandura J. P., "7. Vehicle Health Management Systems," *Digital avionics handbook*, John Wiley & Sons, 2015, pp. 1~24.

14) Pullum, L. L., *Software fault tolerance techniques and implementation*, Artech House, 2001, pp. 1~53.

15) Airlines electronic engineering committee (AEEC)., *Avionics application software standard interface – ARINC Specification 653 - Part 1. (supplement 2 - required services)*, ARINC Inc. 2015.

16) Ha, O. K., Tchamgoue, G. M., Suh, J. B. and Jun, Y. K., "On-the-fly healing of race conditions in ARINC-653 flight software," *Digital Avionics Systems Conference (DASC)*, 2010 IEEE/AIAA 29th, October 2010, pp. 5.A.6-1~5.A.6-11.

17) Tchamgoue, G. M., Ha, O. K., Kim, K. H. and Jun, Y. K., "A framework for on-the-fly race healing in ARINC-653 applications," *International Journal of Hybrid Information Technology*, SERSC, April 2011, pp. 1~12.

18) United State Department of Defense, "Appendix E. Generic Software Safety Requirements and Guidelines," *Joint Software Systems Safety Engineering Handbook*, August 2010, pp. E-15~E-18.

19) Dang, H.-V., Snir, M. and Gropp, W., "Towards millions of communicating threads," *Proceedings of the 23rd European MPI Users' Group Meeting*, September 2016, pp. 1~14.

20) Ha, O.-K. and Jun, Y.-K., "An Efficient Algorithm for On-the-Fly Data Race Detection Using an Epoch-Based Technique," *Scientific Programming*, Vol. 2015, 205827, 2015, pp. 1~14

21) Sridharan, S., Gupta, G. and Sohi, G. S., "Adaptive, Efficient, Parallel Execution of Parallel Programs," *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2014, pp. 169~180.

22) Zhang, M., Wu, Y., Lu, S., Qi, S., Ren, J. and Zheng, W., "A Lightweight System for Detecting and Tolerating Concurrency Bugs," *IEEE Transactions on Software Engineering*, October 2016, pp. 899~917.

23) Oracle Corporation and/or its affiliates, MySQL Bugs. Available: http://bugs.mysql.com/, 2020.

24) Mozilla and Individual Contributors, https://bugzilla.mozilla.org/, 2020.

25) The Apache Software Foundation, https://httpd.apache.org/bug_report.html, 2020.