AT697F/VxWorks 플랫폼에서 Lua 가상머신 기반의 OBCP 엔진 설계 및 구현

최종욱* 정회원, 박수현*

Design and Implementation of OBCP Engine based on Lua VM for AT697F/VxWorks Platform

Jong-Wook Choi*, Su-Hyun Park*

요 약

일반적으로 Operator on Board로 불리는 OBCP (On-Board Control Procedure)는 기존 탑재소프트웨어를 변경하지 않으면서 동적으로 지상 또는 온보드에서 명령과 로직이 포함된 특정 프로시저를 로딩, 언로딩 및 실행 할 수 있으며, OBCP를 통해 기존 위성의 제한된 자율성 및 강인성을 증대 시킬 수 있다. 탑재소프트웨어의 OBCP의 핵심은 OBCP 엔진이며, OBCP 엔진은 스크립트 기반의 프로시저를 해석 및 실행 할 수 있는 인터프리터 형태로 구현되어 있으며 내부적으로 가상머신을 가지고 있다. 탑재소프트웨어팀에서는 2010년부터 내부적으로 OBCP에 대해서 계속 연구를 수행하였으며 ERC32 프로세서 기반의 Java KVM, RTCS/C 및 KKOMA와 같은 자체 OBCP 엔진을 개발하였다. 최근에는 ESA OBCP 표준에 대한 연구를 계속 진행하고 있으며 LEON2-FT/AT697F 프로세서 기반에서 Lua와 MicroPython을 이용한 OBCP 엔진 연구를 진행하고 있다. 본 논문에서는 현재 가장 활발히 사용되고 있는 오픈소스 기반의 Lua를 탑재소프트웨어의 OBCP 엔진으로 사용하기 위하여 VxWorks 기반의 AT697F 프로세서에서의 설계 및 구현 방법에 대해서 기술하며, 시뮬레이터와 실제 하드웨어의 테스트 결과와 함께 성능 비교 분석을 수행한다.

Key Words : OBCP, OBCP engine, Lua, AT697F, VxWorks

ABSTRACT

The OBCP called 'operator on board' is that of a procedure to be executed on-board, which can be easily be loaded, executed, and also replaced, without modifying the remainder of the FSW. The use of OBCP enhances the on-board autonomy capabilities and increases the robustness to ground stations outages. The OBCP engine which is the core module of OBCP component in the FSW interprets and executes of the procedures based on script language written using a high-level language, possibly compiled, and it is relying on a virtual machine of the OBCP engine. FSW team in KARI has studied OBCP since 2010 as FSW team's internal projects, and made some OBCP engines such as Java KVM, RTCS/C and KKOMA on ERC32 processor target only for study. Recently we have been studying ESA's OBCP standard and implementing Lua and MicroPython on LEON2-FT/AT697F processor target as the OBCP engine. This paper presents the design and implementation of Lua for the OBCP engine on AT697F processor with VxWorks RTOS, and describes the evaluation result and performance of the OBCP engine.

I. 서 론

현재 ESA (European Space Agency)는 인공위성의 하드웨어, 인터페이스, 소프트웨어 및 시뮬레이터 등에 대한 전반적인 표준화를 위해 SAVOIR[1] (Space Avionic Open Interface aRchitecture)를 2007년부터 진행하여 2017년까지 최종 결과물과 표준문서를 공표할 예정이다. SAVOIR의 하위 그룹인 SAVOIR-FAIRE[2] (Fair Architecture and Interface Reference Elaboration)에서는 탑재소프트웨어를 위한 reference architecture[3]를 정의하고 다양한 소프트웨어 컴포넌트와 building block (BB)을 통한 표준방안을 제시하였다. 또한 SAVOIR-FAIRE와 SAVOIR-IMA[4] 결과물

*한국항공우주연구원 위성비행소프트웨어팀 (jwchoi@kari.re.kr, psh@kari.re.kr)
접수일자 : 2017년 9월 21일, 수정완료일자 : 2017년 9월 25일, 최종게재확정일자 : 2017년 9월 26일

을 이용하여 OSRA-SAVOIR[5] (On-board Software Reference Architecture)에서 최종 탑재소프트웨어 표준 아키텍처를 구현 및 제시할 계획이다. SAVOIR-FAIRE에서 또 하나의 중요한 소프트웨어 컴포넌트로 OBCP (On-Board Control Procedures)에 대한 표준[6]을 정의하였으며, ESA 협력업체인 GMV, SSF 등에서는 OBCP 표준에 따라 OBCP를 실제 구현 및 위성에 탑재를 준비하고 있다.

OBCP는 기존 위성의 제한된 자율성을 향상시키기 위하여 지상에서 명령과 로직이 포함된 프로시저를 전송하면 탑재소프트웨어 (FSW, Flight SoftWare)의 OBCP 컴포넌트에서 자동적으로 프로시저를 수행한다. FSW의 OBCP의 핵심은 OBCP 엔진이며, OBCP 엔진은 스크립트 기반의 프로시저를 해석 및 수행할 수 있는 인터프리터로 내부적으로 VM (Virtual Machine)을 가지고 있다. 본 논문에서는 현재 가장 활발히 사용되고 있는 오프소스 기반의 Lua[7]을 탑재소프트웨어의 OBCP 엔진으로 사용하기 위해 LEON2-FT/AT697F 프로세서 및 VxWorks RTOS 실행환경에서 Lua 기반의 OBCP 엔진의 설계 및 구현 방법에 기술하며, 실제 하드웨어에서 및 시뮬레이터에서의 테스트 결과와 함께 성능 비교 분석을 수행한다.

Ⅱ. On-Board Control Procedures

OBCP는 일반적으로 "Operator on Board"로 불리며 기존 탑재소프트웨어를 변경하지 않으면서 동적으로 지상 또는 온보드에서 명령과 로직이 포함된 특정 프로시저를 로딩, 언로딩 및 실행 할 수 있기 때문에 기존 위성의 제한된 자율성을 향상시킬 수 있으며 강인성을 증대할 수 있다.

1. OBCP 시스템

OBCP 시스템은 일반적으로 전통적인 MTL (Mission Time Line) 기반의 방식과 스크립트 기반의 OBCP로 분류할 수 있다. 첫번째 방식의 OBCP는 지상국과 교신이 되지 않는 시간동안 위성운영을 위하여 상대시간 명령 및 절대시간 명령들의 조합을 지상과 교신이 가능한 시간에 미리 위성으로 전송하여 저장한 뒤 실행시간이 도래한 명령들을 수행하는 MTL 방식이며, 이러한 접근방식은 미리 전송한 운영절차가 고정되어있어 예상하지 못한 상황에 대한 로직의 설계가 허용되지 않는 단점을 가지고 있다. 두번째 방식의 OBCP는 상위 레벨의 언어로 작성된 스크립트를 위성으로 전송하여 인터프리터 기반의 OBCP 엔진에서 수행하는 방식이다. 이 방식을 사용할 경우 특히 새로운 로직을 구현하기 위해 탑재소프트웨어를 변경하지 않으면서도 복잡한 제어 로직을 구동 할 수 있으며, 다양한 프로시저를 업로드 후 수행 할 수 있어 자율성을 높일 수 있는 장점을 가진다. 첫번째 방식의 경우 기존 한국항공우주연구원에서 개발 된 모든 위성에 적용된 반면 두번째 방식의 경우 지상국과 통신이 제한되는 달 탐사 및 심우주 탐사에서는 반드시 필요한 OBCP 개념이며 현재까지 항우연 위성에 적용된 경우가 없다.

OBCP 시스템은 그림 1과 같이 OBCP 준비환경 (OBCP Preparation Environment)과 OBCP 실행환경 (OBCP Execution Environment)으로 구성된다. 준비환경은 지상에 위치하며, OBCP 편집기, OBCP 컴파일러 및 검증환경을 포함한다. FSW OBCP 엔진의 경우 보통 인터프리터를 가진 VM으로 구성되며, OBCP 수행 중 오류가 발생하더라도 FSW에 오류가 전파되지 않게 설계된다.

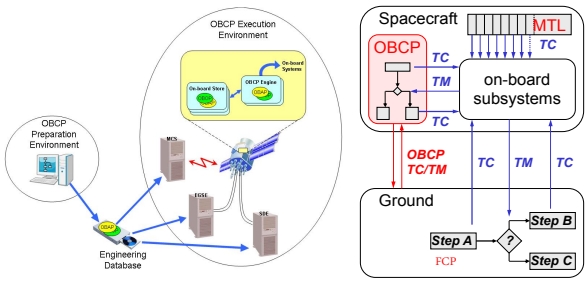

그림 1. The OBCP System

2. OBCP 적용사례 및 개발현황

ESA에서는 협력업체를 통해 다양한 방법으로 OBCP를 구현하였으며, OBCP 표준이 정립된 이후 이를 기반으로 재사용할 수 있으며 표준화 된 OBCP-BB를 새롭게 정의하고 프로타입을 개발하고 있다. 최초 OBCP 개념을 사용한 ESA 미션은 Eureca 위성이며 이후 Rosetta, Venus Express, Mars Express, SMART-1, Exomars, Herschel/Plank등에서 사용되었다. 항우연에서는 천리안 위성에 OBCP와 유사한 Airbus Eurostar 3000의 IP (Interpreted Program)를 사용하였다.

OBCP의 핵심인 OBCP 엔진의 경우 Airbus에서는 자체적으로 개발한 IP를 사용하고 있으며, Rosetta의 경우 SCL[8] (Spacecraft Control Language)을 사용하였으며, Herschel/Planck의 경우 새로운 OCL[9] (On-board Control Language)를 사용하였다. 이 경우 OBCP를 위한 새로운 언어를 새롭게 정의해야하고 OBCP 컴파일러를 처음부터 개발해야하기 때문에 OBCP 준비환경 구축하고 검증하는데 많은 시간과 노력이 소요된다. 하지만 Thales Alenia Space의 경우 새로운 OBCP 언어를 개발하는 대신 Java VM을 OBCP 엔진으로 개발하여 OBCP 준비환경을 위한 구축하는 시간과 노력을 최소화할 수 있었다. OBCP 표준에서는 OBCP language capabilities를 정의하였으며, SAVOIR-FAIRE를 통해 OBCP-BB 구현을 위해 연구가 진행되어, SSF에서는 Herschel/Planck에서 사용되었던 OCL을 확장하는 방향[10]으로 개발하고 있으며, GMV의 경우 Lua 인터프리터를 OBCP 엔진으로 개발[11]하고 있다. 그리고 ESA에서는 George Robotics를 통해 ARM 기반의 MicroPython을

LEON 플랫폼에서 사용 할 수 있도록 포팅[12]을 수행하였으며 탑재체 실행환경을 위한 OBCP-BB으로 준비하고 있다.

ESA에서는 현재 OBCP 수행 환경으로 아래 그림 2와 같이 non-IMA (Integrated Modular Avionics) 환경에서 Fault Containment와 동적 OBCP 업로드 기능을 포함한 수행 환경을 구축하였으며 추후 TSP (Time and Space Partition)를 지원하는 IMA 환경에서도 OBCP가 수행될 수 있는 연구를 최종 목표로 수행하고 있다.

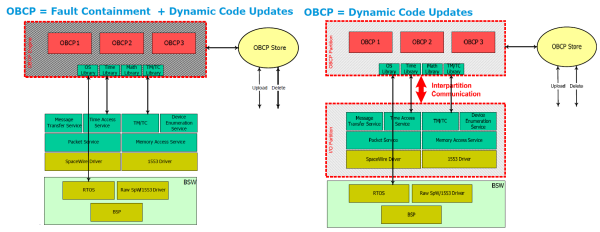

그림 2. OBCP Execution Environment in ESA

Ⅲ. OBCP 엔진을 위한 Lua 가상머신

1. Lua and Virtual Machine

Lua는 1993년 Pontifical Catholic 대학에서 개발되었으며, 경량형 명령형/절차적 언어로 확장언어로 쓰일 수 있는 스크립팅 언어를 주목적으로 설계되었다. Lua는 ANSI C로 작성되었고 메모리 사용률이 작기 때문에 다른 플랫폼으로 포팅이 용이하며, 상대적으로 간단한 C API를 가지고 있다. Lua는 레지스터 기반의 VM을 가지고 있으며 32bit 명령어 코드를 사용하며 자체적인 컴파일러를 가지고 있다. 현재 Lua 최신버전은 5.3.4를 제공하고 있으며 항우연에서는 기존 ERC32 프로세서를 위해 개발된 Lua 5.2.3을 기반으로 하여 AT697F 프로세서에도 동일한 버전을 사용하였다.

2. Lua Interpreter

Lua 는 인터프리터 언어로서 스크립트를 로드하고 실행하기 위한 command line interpreter를 라이브러리 형태로 제공한다. Lua는 ANSI C 로 작성되었으며, C 코드에서 Lua API를 호출해서 Lua interpreter[13]을 사용할 수 있다. Lua는 인터프리터 언어이지만, 미리 컴파일한 bytecode 형태로 로드할 수도 있다. 이 경우 바이너리 파일 방식으로 로드할 경우 텍스트 기반의 스크립트처럼 파싱 및 컴파일 하지 않기 때문에 더 빠른 시간 안에 Lua 스크립트를 로드하고 실행할 수 있다.

C 코드에서 Lua 스크립트를 실행하기 위해서는 먼저 Lua interpreter reference를 생성해야 한다. 이 단계에서는 lua_open() 함수를 호출함으로써 lua_State를 초기화한다. 이 단계를 여러 번 수행하게 되면, 여러 개의 Lua interpreter를 생성할 수 있다.

다음 단계에서는 Lua 스크립트에서 사용하는 라이브러리를 등록한다. Lua는 math, string 라이브러리와 같은 기본적인 라이브러리를 제공하는데, luaopen_math(), luaopen_string()과 같은 함수를 호출함으로써 math, string 라이브러리를 등록할 수 있다. Lua 스크립트에서는 사용자가 작성한 C 함수도 호출할 수 있다. Lua에서 호출할 C 함수는 미리 등록해야 하며, Lua 프로토콜에 따라 정의해야 한다. 즉, Lua API 함수를 이용해서 Lua로부터 파라미터를 받고, Lua로 결과를 반환해야 한다.

다음으로 Lua interpreter는 실행할 bytecode를 로드한다. 이 단계서는 미리 컴파일 된 bytecode를 로드할 수도 있고, Lua 스크립트 자체를 로드할 수도 있다. 스크립트를 로드한 경우, Lua lexer, Lua parser 및 Lua compiler를 사용해서 bytecode를 생성한다. 마지막으로 Lua interpreter는 루프를 돌면서 bytecode로부터 VM instruction을 하나씩 실행한다.

3. Lua Instructions

Lua VM 용 instruction[14]은 32bit로 구성되며, 대부분의 instruction 은 그림 3과 같이 레지스터 기반의 3-address format을 따른다. Opcode는 6비트로 구성되며, B와 C는 피연산자로서 레지스터 혹은 상수가 될 수 있다. A는 연산 결과를 저장하는 레지스터이다. 가령 a = a + 1 은 ADD x x y 로 변환되는데, x는 a 변수를 가지는 레지스터이고 y는 상수값 1을 나타낸다.

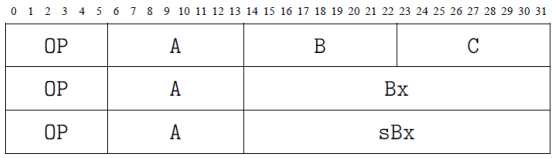

그림 3. Lua Instruction Format

Lua는 변수에 타입을 붙이는 것이 아니라 값 자체에 타입을 붙이는 dynamic-typed 언어이다. nil, boolean, number, string, table, function, userdata, thread의 8가지 타입을 제공한다. Nil은 값이 초기화가 되지 않은 경우, 즉 값이 없는 경우에 사용하는 타입이다. Number는 double-precision floating-point 로서 64비트이다. Table은 array 와 유사한데 index가 반드시 정수일 필요가 없다. 어떤 값으로도 index 할 수 있고, 어떤 값도 가질 수 있다. Function은 Lua function이거나 Lua 프로토콜에 따라 정의된 C function 이다. Userdata는 사용자가 정의한 메모리 블록에 대한 포인터이다. Thread는 동시에 실행되는 coroutine을 나타낸다. Thread를 이용하면 하나의 Lua interpreter에서 여러 개의 스크립트를 실행할 수 있고, 특정 스크립트의 실행을 멈추거나 (yield), 멈춘 부분부터 다시 실행을 시작 (resume)할 수 있다.

Lua 5.0 기준으로 35개의 instructions 이 존재한다. 대부분은 산술 연산, 함수 호출, 변수에 값을 설정하거나 값을 읽기 위한 instruction이다. 이 외에 if-then-else 문이나 loop 문과 같은 control structure를 구현하기 위한 jump instruction도 있다. 레지스터는 run-time stack을 사용하는데, 본질적으로는 배열이기 때문에 레지스터로의 접근은 빨리 이루어진다. 다만 Lua는 dynamic-typed 언어이기 때문에, 값을 읽거나 저장할 때 기본적으로 타입 정보도 함께 접근해야 한다. 값을 한번 복사하려면 8 byte의 값과 4 byte의 타입 정보를 함께 접근해야 하므로 머신 코드로는 3 또는 4 워드가 소요된다. 이 부분이 성능면에서 비용이 많이 발생하는 부분이다.

표 1은 Lua 스크립트를 bytecode로 변환한 예제이다. Lua interpreter는 왼쪽 열의 Lua script를 로드해서 오른쪽 열의 bytecode로 컴파일한 후, instruction 단위로 bytecode를 실행한다. R(x)는 x번째 레지스터를 의미하는데, a 변수는 R(0), b 변수는 R(1), m 변수는 R(2)에 저장된다.

표 1. Lua 스크립트와 bytecode 예제

Lua script	Bytecode

function max (a,b)
  local m = a
  if b > a then
  m = b
  end
  return m
end
	1 MOVE 2 0 0 ; R(2) = R(0)
	2 LT 0 0 1 ; R(0) < R(1)
	3 JMP 1 ; to 5 (4+1)
	4 MOVE 2 1 0 ; R(2) = R(1)
	5 RETURN 2 2 0 ; return R(2)
	6 RETURN 0 1 0 ; return

Ⅳ. Lua 기반의 OBCP 엔진 설계 및 구현

1. Implementation of Lua to AT697F/VxWorks

Lua가 ANSI C로 작성되었기 때문에 VxWorks 5.4에 구현하는 것은 큰 어려움이 없으나 위성에서 사용되는 실시간 운영체제인 VxWorks의 경우 파일 시스템을 포함하지 않기 때문에 Lua에서 사용되는 fopen(), fread(), fwrite()와 같은 라이브러리를 사용할 수 없으며 모두 UART 인터페이스를 통해 입출력을 수행해야 한다. Lua 스크립트의 수행시간을 측정하기 위해서는 os_clock()이라는 Lua API를 사용하는데, 이 함수를 VxWorks clock_gettime() 함수로 반드시 맵핑해 주어야 한다. 그리고 va_start(), va_arg() 및 va_end()를 위한 ANSI stdarg 라이브러리의 경우 SPARC으로 정확히 설정되어야 한다.

구현 과정을 통해 Lua에서 관련된 다수의 소스코드가 수정되었으며, Tornado에서 Lua shell 등 모든 기능을 포함하여 빌드 할 경우, 최적화 옵션을 사용하지 않았을 때 실행파일은 232.97Kbytes (text: 224.56Kbytes, data: 408bytes, bss: 8200bytes) 크기를 가지며 최적화 옵션을 레벨2로 설정하여 빌드하는 경우 143.86Kbytes (text: 135.45Kbytes, data: 408bytes, bss: 8200bytes)를 사용한다. 기존 Java KVM을 ERC32 프로세서에 포팅 했을 때 270Kbytes footprint를 가진 것을 고려할 때 Lua의 실행파일의 메모리 footprint가 작은 것을 알 수 있으며 위성용 컴퓨터의 메모리 용량을 고려할 경우 충분히 사용 가능하다.

현재 구현된 Lua 기반의 OBCP 엔진의 경우 Lua 자체의 검증을 위해 먼저 개발되었으며, 탑재소프트웨어와 연동되는 부분은 계속해서 개발 될 예정이다. 탑재소프트웨어의 연동은 Lua의 O/S 라이브러리 API에 구현 할 계획이며 이를 기반으로 차세대 위성에서는 OBCP 엔진으로 활용할 계획이다.

2. Test Lua on AT697F/VxWorks

구현된 Lua 테스트를 위해 AT697F 에뮬레이터인 laysim-at697기반의 K6-FSS에서 테스트를 수행하였다. AT697F UART B가 Lua shell로 사용되며 UART A의 경우 VxWorks Target Shell로 사용된다. K6-FSS에서 동적으로 Lua 모듈을 다운로드하고 VxWorks task로 수행하면 Lua shell이 구동하는 것을 확인할 수 있다. Lua 태스크는 VxWorks에서 우선순위 100으로 동작하고 있으며 태스크 스택으로 2560bytes를 사용한다. Lua shell에서 dofile() 명령을 통해 Lua 스크립트 파일이나 Lua 컴파일을 통해 생성된 bytecode을 로딩 할 수 있으며, Lua shell에서 다양한 명령을 수행 할 경우 정상적으로 동작하는 것을 확인할 수 있다.

그림 4는 텍스트 기반의 Lua 스크립트를 AT697F UART B를 통해서 로딩 및 실행한 결과를 보여준다. 텍스트 기반의 Lua 스크립트가 로딩 되면 레지스터 기반의 Lua VM에서 수행 될 수 있도록 실시간으로 스크립트는 파싱 및 컴파일 되어 bytecode로 변환되며 최종적으로 Lua 스택에 적재되어 실행할 수 있게 된다. Lua 스크립트를 미리 호스트 컴퓨터에서 Lua 컴파일러 (luac)를 이용하여 해당 스크립트를 컴파일 할 경우 bytecode로 생성된다. 이 경우 생성된 bytecode가 x86 호스트 컴퓨터의 little-endian 형태로 생성되기 때문에 big endian기반의 AT697F와 호환되지 않는 문제가 발생한다. 이를 위해 자체적으로 개발한 변환 툴을 이용하여 bytecode를 big-endian으로 변환해야 하며, bytecode를 로딩 할 경우 별도의 파싱 및 컴파일 과정 없이 bytecode 체크만을 수행하기 때문에 스크립트 로딩시간 보다 짧은 시간이 소요된다.

그림 5는 factorial을 계산하는 Lua 스크립트를 luac를 통해 bytecode로 생성한 뒤 AT697F를 위해 big-endian 포맷으로 변환된 결과이다. Bytecode list를 확인하면 첫번째 main은 3개의 instruction으로 구성되며 내부적으로 factorial이라는 함수를 하나 가지고 있다. factorial은 9개의 instruction으로 구성되며 하나의 parameter를 받아서 수행된다. AT697F를 위해 변환된 bytecode의 이진파일을 확인하면 0x00 ~ 0x11까지 18bytes LUA 헤더가 있으며, 0x11 ~ 0x43까지 main을 위한 코드가 있으며, 0x44 ~ 0x48은

factorial을 위한 코드가 있으며 나머지는 tail 코드이다.

현재 구현된 Lua VM의 경우 AT697F UART B를 통해서만 스크립트 및 bytecode를 로딩 할 수 있으며, 추후 위성의 원격명령 인터페이스를 통해 업로드 하는 절차를 확립할 예정이다.

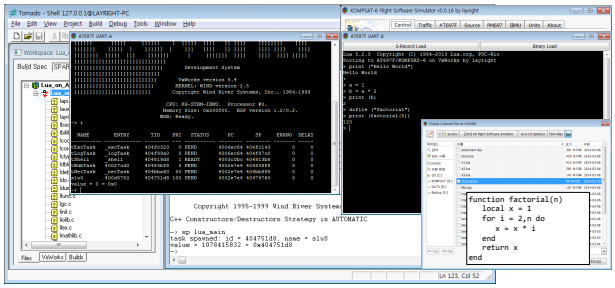

그림 4. Lua with VxWorks Test on K6-FSS

```
[Lua Script File]

function factorial(n)
   local x = 1
   for i = 2,n do
      x = x * i
   end
   return x
end
```

```
[Lua bytecode listing]

main <factorial.lua:0,0> (3 instructions at 00398EB8)
0+ params, 2 slots, 1 upvalue, 0 locals, 1 constant, 1 function
        1       [7]     CLOSURE         0 0     ; 00391528
        2       [1]     SETTABUP        0 -1 0  ; _ENV "factorial"
        3       [7]     RETURN          0 1

function <factorial.lua:1,7> (9 instructions at 00391528)
1 param, 6 slots, 0 upvalues, 6 locals, 2 constants, 0 functions
        1       [2]     LOADK           1 -1    ; 1
        2       [3]     LOADK           2 -2    ; 2
        3       [3]     MOVE            3 0
        4       [3]     LOADK           4 -1    ; 1
        5       [3]     FORPREP         2 1     ; to 7
        6       [4]     MUL             1 1 5
        7       [3]     FORLOOP         2 -2    ; to 6
        8       [6]     RETURN          1 2
        9       [7]     RETURN          0 1
```

```
[Lua bytecode for AT697F]

00000000 : 1B 4C 75 61 52 00 00 04 04 04 08 00 19 93 0D 0A
00000010 : 1A 0A 00 00 00 00 00 00 00 00 00 01 02 00 00 00
00000020 : 03 00 00 00 25 80 00 00 08 00 80 00 1F 00 00 00
00000030 : 01 04 00 00 00 0A 66 61 63 74 6F 72 69 61 6C 00
00000040 : 00 00 00 01 00 00 00 01 00 00 00 07 01 00 06 00
00000050 : 00 00 09 00 00 00 41 00 00 40 81 00 00 00 C0 00
00000060 : 00 01 01 80 00 00 A1 00 81 40 4F 7F FF 40 A0 01
00000070 : 00 00 5F 00 80 00 1F 00 00 00 02 03 3F F0 00 00
00000080 : 00 00 00 00 03 40 00 00 00 00 00 00 00 00 00 00
00000090 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000A0 : 00 00 00 00 00 00 00 00 01 01 00 00 00 00 00 00
000000B0 : 00 00 00 00 00 00 00 00 00 00 00 00
```

그림 5. Bytecode of Factorial Lua Script

3. Lua Error Handling and Exception

Lua에서 에러 및 오류처리는 Lua 자체적으로 처리되고 상위 응용프로그램에 에러 코드를 리턴한다. 상위 응용프로그램에서는 해당 에러 코드에 맞게 처리를 해야 한다. Lua shell이 사용되는 경우 자세한 에러 메시지가 출력되고, 다시 Lua shell은 정상동작을 하게 된다. 그리고 Lua에서는 오류처리를 위해 try()와 pcall()을 통해 exception을 처리 할 수 있다. 그림 6처럼 3개의 Lua 스크립트를 로딩하고 테스트를 수행하면 HI/LO 값이 로컬로 선언되었기 때문에 액세스 에러가 발생하여 자세한 에러정보가 리포팅 된다. 에러가 발생한 HI/LO 값을 전역변수로 변경 한 뒤 재수행하면, 정상적으로 exception없이 수행되는 것을 확인 할 수 있다.

추후 위성 OBCP 엔진으로 사용되기 위해서는 Lua의 exception handling 메커니즘에 대한 연구가 더 필요하며, 탑재소프트웨어와 연동되어서 동작할 때 발생할 수 있는 오류처리방안도 마련해야 한다.

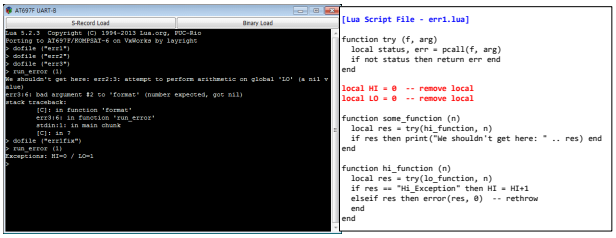

그림 6. Lua Exception Test

4. Experiment

Lua 성능 측정 사이트에서 제공되는 Ackermann's Function, Array Access, Matrix Multiplication, Fibonacci Number 4가지의 예제를 통해 Lua 인터프리터의 성능을 스크립트, bytecode 기준으로 AT697F/VxWorks 환경에서 측정하였다. 비교 대상으로 C 코드로 구현된 동일한 테스트를 최적화 옵션 O0, O2를 각각 적용하여 수행하였다.

그림 6과 같이 Lua 스크립트를 로딩 및 실행한 전체 시간과 bytecode를 로딩 및 실행한 전체 시간을 비교하면, bytecode로 로딩 및 실행한 결과가 평균 5%정도 빠른 것을 확인 할 수 있다. 이것은 앞에서 언급한 것처럼 텍스트 기반의 Lua 스크립트를 로딩 할 경우 실시간으로 파싱 및 컴파일 되는 시간이 bytecode를 로딩 하는 것보다 약 5배 느리기 때문이다. 실제 수행시간만을 고려할 경우 스크립트나 bytecode 모두 오차범위 내에서 수행시간이 동일한 것을 확인 할 수 있다. 즉 bytecode 변환 이후에 수행시간은 동일하며, 이것은 추후 탑재소프트웨어에서 OBCP 프로시저 로딩

Benchmark	Execution	Lua Scipt Execution Total Time (ms)						Lua Bytecode Execution Total Time (ms)						C Code Execution Time (ms)		Peformance Comparison (Lua Script vs		Peformance Slow Down (Lua Bytecode vs C)	
		Optimization Level 0			Optimization Level 2			Optimization Level 0			Optimization Level 2								
		Load Time	Execution Time	Total Time	Load Time	Execution Time	Total Time	Load Time	Execution Time	Total Time	Load Time	Execution Time	Total Time	-O0	-O2	S/B (O0)	S/B (O2)	Lua/C (O0)	Lua/C (O2)
Ackerman	Ack(3,5)	6.363	1037.826	1044.189	4.529	492.777	497.305	1.417	1037.82	1039.346	0.963	492.818	493.781	49.99	33.33	1.005	1.007	20.791	14.815
Arrary Access	Arry3(100)	7.569	1892.798	1900.368	5.15	898.304	903.455	1.403	1892.618	1894.022	0.983	898.102	899.085	66.66	16.67	1.003	1.005	28.413	53.934
Matrix	Matrix(50)	5.892	30836.538	30842.43	4.371	15120.823	15125.194	1.444	30651.479	30652.924	1.105	15009.785	15010.889	1420	520	1.006	1.008	21.587	28.867
Fibonacci	Fib(30)	5.13	60284.465	60289.595	3.742	28498.664	28502.397	1.343	60284.612	60285.954	0.958	28498.826	28499.783	1030	560	1.000	1.000	58.530	50.892

그림 6. Performance Comparison

을 위해 스크립트 기반을 사용할 지 bytecode 기반으로 선택할지에 대한 근거로 사용될 수 있으며 추후 OBCP 엔진에 대한 상세설계가 수행 될 때 반드시 고려할 항목이다.

Lua 인터프리터의 성능을 동등한 C 코드 수행시간과 비교할 경우 최소 20.79에서 최대 58.53배의 slow down이 발생한 것을 확인할 수 있다. 즉 Lua 스크립트를 이용 할 경우 대략 20배 이상 성능저하가 발생한다. 이 결과는 기존 Java KVM을 ERC32 프로세서에 포팅 했을 때 측정한 결과보다 월등히 좋은 결과이며 Lua의 동적 로딩 기능과 확장성을 고려할 경우 OBCP 엔진으로 문제가 없다고 판단된다.

V. 결 론

본 논문에서는 기존 위성의 제한된 자율성을 향상시키기 위해서 지상 또는 온보드에서 명령과 로직이 포함된 프로시저를 수행할 수 있는 OBCP 개념에 대해서 설명하였으며, Lua 5.2.3 VM을 AT697F 프로세서 기반의 VxWorks 5.4에 구현 방법과 테스트 결과를 제시하였다. Lua가 가지고 있는 동적 로딩 및 수행, 그리고 에러/오류 처리방식이 OBCP 엔진으로 사용하기에 문제가 없다고 현재 판단된다. 추후 달 탐사 및 심우주 탐사를 위한 OBCP 엔진으로 활용되기 위해서는 탑재소프트웨어와 연동 방안이 먼저 연구가 되어야 되며, 또한 Lua가 사용하는 스택영역에 대한 모니터링 기능과 함께 exception handling 메커니즘에 대한 연구가 계속 진행되어야 한다. 그리고 실제 탑재소프트웨어에 탑재하기 위해서는 Lua VM에 대한 연구가 추가적으로 수행될 필요가 있다.

참 고 문 헌

[1] Jean-Loup TERRAILLON, "SAVOIR Status/Reference Architecture", ESA Workshop in Avionics Data, Control and Software Systems (ADCSS), 2016.

[2] Jean-Loup TERAILLON, "SAVOIR-FAIRE Status and Perspective", ADCSS, 2010.

[3] Maria Hernek, "ESA - Execution Platform", DAta Systems In Aerospace (DASIA), 2011.

[4] M. Hiller, "Integrated Modular Avionics : SAVOIR-IMA status and progress", ADCSS, 2012.

[5] Andreas Jung, "Software Reference Architecture - Presentation of the OSRA Specification", ADCSS, 2014.

[6] ESA, "ECSS-E-ST-70-01C, Spacecraft On-Board Control Procedures", 2010.

[7] Lua, http://www.lua.org/

[8] G. M. Lautenschlager, "OBCPs - The Operator On-Board (The OBCP Concept used by ROSETTA)", DASIA, 2004.

[9] Massimo Ferraguto, "The On-Board Control Procedures Subsystem for the Herschel and Planck", Annual IEEE International Computer Software and Application Conference, 2009.

[10] Massimo Ferraguto, "Towards the Definition of ESA's Future OBCP Building Block", DASIA, 2012.

[11] A.I. Rodriiguez, "OBCP-BB GSTP Study - Requirements and Interface Definition for Future OBCP Building Block", ESTEC TEC-ED & SW Final Presentation Days, 2012.

[12] Damien G., David S., and Tiago J., "Porting of MicroPython to LEON Platforms", DASIA, 2016.

[13] M.S. Glsberg, Jim Bresler and Youngmin Cho, "The Lua Architecture", Advanced Topics in Software Engineering, 2006.

[14] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, "The Implementation of Lua 5.0", 11#7, 2005, pp.1159 - 1176.

저자

최 종 욱(Jong-Wook Choi) **정회원**

- 1999년 2월 : 경북대학교 전자공학 학사졸업
- 2001년 2월 : 경북대학교 전자공학 석사졸업
- 2016년 2월 : 충남대학교 컴퓨터공학과 박사 졸업
- 2000년 12월 ~ 현재 : 한국항공우주연구원 위성비행소프트웨어팀 책임연구원

<관심분야> : 시뮬레이터, 실시간운영체제

박 수 현(Su-Hyun Park)

- 2003년 2월 : 경북대학교 전산학과 학사졸업
- 2005년 2월 : 한국과학기술원 전산학과 석사졸업
- 2004년 12월 ~ 현재 : 한국항공우주연구원 위성비행소프트웨어팀 선임연구원

<관심분야> : 소프트웨어 공학, OBCP