소프트웨어정의 전장시스템을 위한 RUST언어기반 하이퍼바이저 소프트웨어

서 상 범[1].문 태 현[1].정 지 원[1]

㈜페르세우스[1].

)

Hypervisor Software Written in RUST For E/E Systems of Software-Defined Vehicle

Sang-bum Suh[1] · Taehyun Moon[1] · Jiwon Jeong[1]

Perseus Co., Ltd[1]

Abstract : RUST 언어로 작성된 PERSEUS사의 PEGASUS 하이퍼바이저 소프트웨어는 Micro-controller Unit(MCU)/ Central-processor Unit(CPU) SoC 기반 임베디드 하드웨어 시스템을 위한 가볍고 고성능의 하이퍼바이저입니다. 다가오는 Software-Defined Vehicle(SDV)의 폭넓은 커버리지를 위해 차량의 전장시스템(E/E System) CPU위에서부터 MCU위에서까지 하이퍼바이저 소프트웨어가 어디에서나 동작하게 됩니다. 또한 전장시스템이 분산컴퓨팅 구조로부터 중앙집중식 컴퓨팅 구조로 옮아감에 따라, CPU와 MCU가 보다 큰 연산능력을 보유한 멀티코아화가 되어가고 있습니다. 이와 같은 멀티코아화에 따라 멀티스레드 병렬 프로그램 동작환경에서 메모리자원의 접근에 대한 concurrency 이슈는 더욱 부각됩니다. 이에따라 병렬 데이터 처리시의 안전한 메모리 접근성을 보장하는 RUST언어가 점차 각광을 받을 것이라 예측합니다. 10여년전에 태어난 RUST언어는 역사는 비교적 짧지만, 멀티코아기반 병렬 컴퓨팅의 도입증가로 시스템소프트웨어에까지 활용이 급증될 수 있습니다.

본 논문에서는 RUST언어기반으로 개발된 전장시스템향 하이퍼바이저 소프트웨어와 RUST언어를 소개하며, 또한 NXP, STMicro사등 주요 자동차향 반도체 벤더의 MCU System-on Chip(SoC)에서 상기의 하이퍼바이저 동작시 수집한 성능 데이터와, 하이퍼바이저 소프트웨어 동작에 필요한 임베디드 컴퓨팅 하드웨어의 요구사양을 소개합니다.
본 논문에서는 또한 기존의 싱글코아 MCU SoC위에서 동작하였던 실시간 운영체제(Real-time OS)와 AutoSAR Classic등이 동일한 Instruction Set Architecture(ISA)를 갖는 멀티코아 MCU SoC 동작시 잠재하는 소프트웨어 구조적 이슈와 해결 방법을 소개합니다.

Key words : 하이퍼바이저, 소프트웨어정의자동차, Hypervisor, SDV, 전장시스템 소프트웨어, MCU

1. 서 론

차량내에서의 서비스의 증가와 자율주행등 스마트한 주행기능이 도입됨에따라 차량 전장시스템의 데이터 처리양이 급격히 증가하는 추세에 있다. 이에따라 연산능력이 크게 증가한 고성능 차량 SoC(High Performance Vehicle Computer SoC)가 출현하고 있으며, 또한 차량의 전장시스템이 분산 컴퓨팅 구조에서 중앙집중형 컴퓨팅구조로 변화하고 있다. 이를 위하여 전장시스템에 사용되던 CPU뿐만아니라 MCU까지 멀티코아를 기본구조로 하여 연산 능력이 대폭 증가한 SoC가 출시되고 있다. 더욱이 SoC동작 clock이 크게 증가하여, 기존의 싱글코아 대비 멀티코아의 개별 코아 연산능력이 기존 싱글코아보다 몇 배로 큰 상태이다.

반면 기존에 전장시스템의 주행기능을 위하여 싱글코아 CPU/MCU SoC에서 동작되던 다양한 종류의 운영체제 소프트웨어를, 연산능력이 대폭으로 증가한 멀티코아 SoC상으로 옮겨 동작시키면 SoC 연산 자원이 크게 남게되어 비효율적이 된다. 이때 만약 주행기능으로 사용된 다수의 운영체제를 하나의 운영체제로 통일하여, 다수의 운영체제 위에서 동작하던 여러 종류의 응용 소프트웨어를 통합하여 새로 개발한다면, ISO26262 기능안전 ASIL 인증을 획득한 소프트웨어의 재인증을 처음부터 다시 준비하는 것과 같은 큰 overhead가 발생하게 된다. 더욱이 ASIL-D인증을 받으려면, 소프트웨어 unit test의 검증과 통합 소프트웨어 시스템의 test시나리오와 coverage가 기하급수적으로 증가할 수 있다. 또한 소프트웨어 공학연구를 보면 하나의 시스템의 소프트웨어 LOC(Line of Codes)가 증가하면, 소프트웨어 결함/버그의 발생 확률이 비례하여 증가한다.[1)]

이러한 상황을 고려하면, 기존에 싱글코아에서 사용된 운영체제의 재사용을 고려하는 것이 기능안전면에서 유리하게 된다. 또한 고성능 SoC위에서 주요 기능을 작은 소프트웨어 크기의 단위로 분리하여 다수의 독립된 운영체제별로 개발하는 전략적 방법이 기능안전과 소프트웨어의 결함을 줄이는 데 유리할 수 있다.

이를 구현하는 방법론에는 크게 두가지가 있다: 1) CPU/MCU의 코아별(물리적 코아의 affinity number)로 운영체제를 한 개씩 동작하게 하는 방법; 2) 하이퍼바이저를 CPU/MCU SoC에 설치하고, 실제코아를 가상으로 여러 개를 생성한 가상의 코아위에서 운영체제를 동작시키는 방법.

서두에 언급하였듯이, 이미 멀티코아의 개별 코아 연산능력이 기존의 싱글코아보다 몇 배로 커진 상태이기에 1)번의 방법은 SoC자원을 비효율적으로 사용하게 될 수 있다. 본 논문에서는 이러한 멀티코아의 증대된 연산능력을 가상의 코아를 생성하여 코아별로도 여러 개의 운영체제를 동시 동작시키거나, 또는 여러 개의 코아를 통합하여 하나의 운영체제에 할당하고 동작시키는 것이 가능한 하이퍼바이저를 사용하는 방안에 대하여 논하려한다. 또한 하이퍼바이저를 활용하면, 운영체제 시스템의 보안을 크게 증강시키는 데 유리하다.[2)]

ARM CPU하이퍼바이저는 기존에 개발된 사례와 역사적으로 성숙되 있기에[3)], 본 논문에서는 최근 3년 사이에 출시된 ARM Cortex-R52 멀티코아기반 MCU하이퍼바이저에 초점을 둔다.

본 논문의 타겟 시스템은 6개의 OS가 임베디드 MCU 하드웨어 보드위에서 PEGASUS 하이퍼바이저 소프트웨어위에서 동시에 실행되게 구성되어 있다. 각 OS당 기본 하드웨어 리소스 할당과 가상 머신(Virtual Machine: VM) 구성은 PEGASUS 하이퍼바이저의 매니페스트화일 및 비쥬얼툴을 통하여 쉽게 동적으로 수행된다. PEGASUS 하이퍼바이저 소프트웨어는 베어 메탈(Bare Metal) 하드웨어에서 실행되는 Type-1 전가상화(Full Virtualization) 방식이며, PEGASUS 하이퍼바이저는 Renesas Electronics, NXP, Infineon Technologies, STMicro, NVIDIA, Samsung Electronics, Rasberry Pi 등과 같은 다양한 자동차 SoC 공급업체의 ARM Cortex-R52 MCU/ Cortex-A CPU 프로세서 아키텍처 외에도 RISC-V에서 동작한다. PEGASUS 하이퍼바이저가 지원하는 OS는 실시간 OS에서 범용 OS까지 다양하며, 이러한 OS는 AutoSAR Classic/Adaptative, FreeRTOS, Zephyr, Linux, Android 등으로 구성된다.

2. 임베디드 하이퍼바이저와 RUST 언어

2.1 CPU/MCU기반 임베디드 하이퍼바이저

2.2.1 하이퍼바이저 구조

하이퍼바이저는 가상화 이론을 구현한 소프트웨어이며, 기본 역할은 실제 하드웨어(예: CPU/MCU, Memory, I/O)를 가상의 하드웨어로 생성하고 그 자원을 관리한다. 또한 (하이퍼바이저 위에서 동작하는) 게스트 운영체제(Guest OS)에게 생성한 가상 하드웨어 자원을 할당하고, 게스트 운영체제와 가상 하드웨어자원을 스케줄링하며 관리한다.

본 논문에서는 ARM 하드웨어를 기반으로 CPU/MCU, Memory, I/O등을 가상화하며, 하드웨어 바로 위에서 동작하는 Type-1 하이퍼바이저를 기준으로 설명한다.

ARM의 V8 구조는 CPU와 MCU별로 기능이 다르게 설계되었으며, 각각 Cortex-A (CPU)코아와 Cortex-R52 (MCU)코아가 있다. ARM CPU와 MCU의 주된 차이는 Memory를 관리하는 기능에 있다. CPU에는 Memory Management Unit(MMU)가 있으며, MCU에는 MMU가 없고 Memory Protection Unit(MPU)가 존재한다. 이를 운영체제가 동작하는 측면으로 보면, MMU가 없는 MCU위에서는, 대용량 DRAM을 요구하고 관리하는 Linux와 같은 범용운영체제가 동작할 수 없다. 다른 한편으로 보면, MCU는 컴퓨팅과 데이터의 실시간성 처리에 유리하여, 차량의 주행시스템에 사용되는 실시간 운영체제가 동작한다. 이와 같은 ARM 하드웨어의 MMU와 MPU 구조에 맞게 CPU하이퍼바이저와 MCU하이퍼바이저가 개발되어야 하며, 세계적으로 현재 한 두 회사만 그 모든 하이퍼바이저를 제공한다. 본 논문에서는 CPU하이퍼바이저(Fig. 1)와 MCU하이퍼바이저(Fig. 2)를 모두 보유한 PERSEUS사의 "PEGASUS" 하이퍼바이저를 사용하여 기술한다.

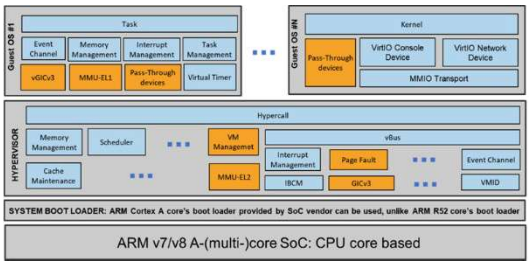

Fig. 1 하이퍼바이저 시스템: ARM Cortex-A CPU향

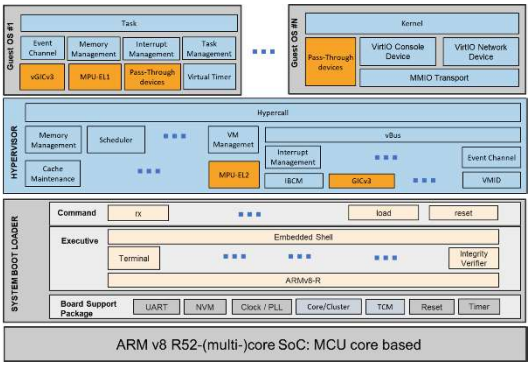

Fig. 2 하이퍼바이저 시스템: ARM Cortex-R52 MCU향

2.2.2 CPU및 MCU 구조와 가상화 방법

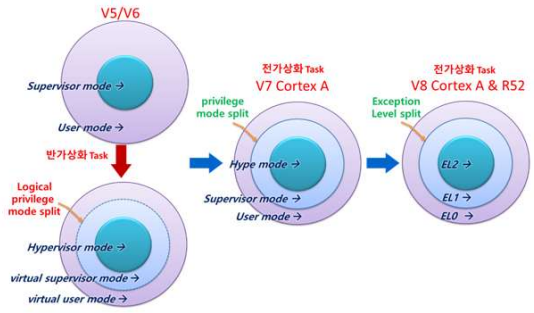

Fig. 3 ARM V5/V6/V7/V8 구조 진화

Fig. 3은 ARM 구조의 진화를 보여주며, V6 구조까지는 가상화를 위한 CPU/MCU 모드가 없었기에, 하이퍼바이저 위에서 동작하는 운영체제의 커널이 Fig. 3의 virtual supervisor mode(실제는 CPU의 user mode)에서 동작할 수 있게 커널의 소스코드를

수정하는 반가상화(para-virtualization) 작업이 필요하였다. 이후 CPU는 ARM V7구조부터, MCU는 V8구조부터 하이퍼바이저를 위한 EL2 mode가 설계되어, 운영체제 소스코드를 수정없이 하이퍼바이저위에서 동작할 수 있는 전가상화(full-virtualization)가 사용되고 있어 다양한 운영체제를 수정없이 양산시 하이퍼바이저 시스템에 적용할 수 있게 되었다. 본 논문은 전가상화를 기반으로 개발된 시스템을 기반으로 기술한다.

2.2.3 멀티코아 SoC향 시스템SW 코딩을 위한 RUST

CPU와 MCU가 멀티코아로 개발되고, RAM 메모리 시스템은 공유되는 컴퓨터 구조와 멀티스레드 프로그램 동작환경이 보편화되는 추세를 고려할 때, 병렬 컴퓨팅에서의 메모리 데이터 처리의 안전성을 보장하는 것은 기본이 된다. 기존의 방식처럼 C/C++언어를 사용하면 데이터 안전성 보장을 개발자의 구현능력에 맡기게 되고, 이로인한 개발자 에러유발 확률은 늘 존재할 가능성이 있다. 그러므로 언어와 컴파일러 수준에서 데이터 처리를 안전하게 보장할 수 있는 RUST의 사용은 필수불가결한 것 이다.

PERSEUS사는 PEGASUS 하이퍼바이저를 세계최초로 RUST 언어로 개발하였고, RUST 기반 하이퍼바이저 시스템의 시연을 ARM 하드웨어위에서 2017년 9월 독일의 IAA 프랑크푸르트 모터쇼에서 진행 하였다.

하이퍼바이저 개발시 사용한 RUST 주요 경험예를 발췌하여 다음과 같이 요약하였다.

1) RUST와 C/C++의 메모리 및 스레드 안전성 비교: 시스템 소프트웨어 개발 관점

메모리 안정성은 현대 시스템 소프트웨어 개발에서 가장 중요한 이슈 중 하나이다. Microsoft 보안 대응 센터(MSRC)의 연구에 따르면, 자사에서 보고된 치명적 보안 취약점의 약 70%가 메모리 안전성 결함에서 비롯되었으며, 그 상당수가 C/C++로 작성된 코드에서 기인하였다. 이는 시스템 수준 프로그래밍에서 메모리 안전성 확보가 얼마나 중요한 문제인지를 여실히 보여준다.

하지만, 전통적인 시스템 프로그래밍 언어는 일반적으로 '제어력(control)'과 '안정성(safety)'을 동시에 만족시키기 어려운 구조적 한계를 갖고 있다. C와 C++은 개발자에게 직접 메모리와 하드웨어를 다룰 수 있는 높은 수준의 제어력을 부여하지만, 그만큼 메모리 안전성과 동시성 오류에 대한 보장은 거의 전적으로 개발자의 책임에 맡겨진다. 이로 인해 성능과 유연성은 확보할 수 있지만, 치명적인 메모리 오류와 보안 취약점을 야기할 위험도 함께 존재한다. 이러한 한계는 안전성과 제어력을 모두 요구하는 현대 시스템 소프트웨어 개발 환경에서, 새로운 언어 설계 패러다임의 필요성을 제기하게 되었다.

이러한 배경에서 등장한 RUST는 메모리 안전성과 동시성 안전성을 컴파일 타임에 보장하면서도, 네이티브 수준의 성능과 시스템 자원에 대한 제어력을 동시에 제공하는 언어로 주목받고 있다. 본 논문은 시스템 소프트웨어 개발 관점에서 C/C++과 RUST를 메모리 안전성, 스레드 안전성, 제어력의 측면에서 비교하고, 주요 언어 메커니즘과 실사용 사례를 중심으로 그 차이점을 분석한다.

2) Control vs. Safety: 시스템 프로그래밍의 트레이드오프

시스템 소프트웨어는 커널, 부트로더, 드라이버, 펌웨어, 하이퍼바이저 등 하드웨어에 가까운 위치에서 실행된다. 이 분야에서 성능과 제어력(Control)은 필수 요건이다. 그렇기에 개발의 자유도가 높은 C/C++를 주로 사용해오고 있다. 그러나 C/C++는 자유도가 높아진 만큼 메모리 시스템의 안전성 완벽히 보장하지 않는다.

3) 메모리 안전성: C/C++ vs. RUST

C와 C++은 포인터 연산, 수동 메모리 할당/해제, 타입 캐스팅 등 개발자에게 극단적인 수준의 제어력(Control)을 허용한다. 하지만 이는 다음과 같은

위험을 동반한다:

- Use-After-Free
- Buffer Overflow
- Double Free
- Dangling Pointer
- Null Pointer Dereference

이러한 버그는 런타임에 발견되며, 재현이 어렵고 보안상 치명적일 수 있다. 더불어 C/C++은 하드웨어 직접 접근, MMIO, 인라인 어셈블리 등을 포함해 거의 무제한에 가까운 저수준 제어를 허용하지만, 이러한 제어력은 곧 메모리 안정성의 손실로 이어질 가능성이 크다.

반면 RUST는 소유권(Ownership), 빌림(Borrowing), 수명(Lifetime)에 기반한 정적 타입 시스템과 빌림 검사기(Borrow Checker)를 통해 대부분의 메모리 및 동시성 오류를 컴파일 타임에 사전 차단한다. 이로 인해 런타임 검사나 가비지 컬렉터 없이도 안전성을 확보할 수 있으며, 필요 시 `Result`, `Option` 등의 명시적 에러 처리와 패닉 제어 메커니즘을 통해 런타임에서도 예외 상황을 안전하게 제어할 수 있다.

RUST의 메모리 안전성 보장 기법:
- 소유권(Ownership): 자원의 유일 소유자 개념
- 빌림(Borrowing): 불변/가변 참조 제약
- 수명(Lifetime): 참조 유효기간 추론 및 검증

4) Ownership과 Borrowing

RUST 언어에서 소유권과 참조는 ==메모리 관리를 위한 핵심 개념==이다. 소유권은 변수에 특정 데이터의 소유권을 부여하여 변수의 유효 범위를 명확히 하며 참조는 다른 변수가 해당 데이터를 잠시 빌려 쓰는 것을 허용한다. 이렇게 함으로써 메모리 중복 해제(double free), dangling reference, use-after-free 같은 오류를 원천적으로 차단한다.

5) Ownership
변수는 하나의 소유자만 가질 수 있으며, 소유자가 스코프(scope)를 벗어나면 값은 자동으로 메모리에서 해제된다. 그리고 소유권은 변수 이동(move)을 통해 다른 변수로 이전할 수 있다.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // s1의 소유권이 s2로 이동
    // println!("{}", s1); // 오류: s1은 더 이상 유효하지 않음
}
```

위 코드에서 s1의 값은 s2로 이동되었기 때문에, s1은 더 이상 유효하지 않다.

6) Borrowing
빌림(borrow)은 함수에 데이터를 전달할 때 소유권을 넘기지 않고 참조만을 전달하는 것을 의미한다. RUST에서는 다음과 같이 두 가지 타입의 빌림이 있다.

- `&T`: 불변 참조. 여러 개 가능.
- `&mut T`: 가변 참조. 단 하나만 가능.

```
fn borrow_example() {
    let mut data = 5;
    let r1 = &data;
    let r2 = &data;
    // let r3 = &mut data; // 컴파일 오류: 불변 참조가 끝나기 전에는 가변 참조 불가
    println!("{} {}", r1, r2);
}
```

이러한 제약은 빌림 검사기(Borrow Checker)에 의해 컴파일 타임에 검증된다.

7) Lifetime

RUST는 참조의 안전한 사용을 보장하기 위해 lifetime(수명)을 도입한다. Lifetime은 변수 혹은 참

조의 '유효 범위(이하 스코프)' 라고 생각할 수 있다. RUST의 모든 변수와 참조는 각자의 범위를 가진다. 범위를 벗어난 위치에서의 접근은 컴파일 타임에 오류를 일으킨다. 아래의 코드를 컴파일 하면 컴파일 오류가 발생한다. 즉, s1이 s2를 참조하지만, s2는 스코프를 벗어나면 사라지기 때문에 s1는 잘못된 메모를 참조하게 된다. RUST는 이와같은 참조에 대해 라이프타임을 계산하여 컴파일 타임에 dangling reference 발생을 차단한다.

```
fn main() {
    let s1;
    {
     let s2 = "hello".to_string();
     s1 = &s2;
    }
 println!("{}", s1);
}
```

8) 저수준 하드웨어 접근 제어

RUST는 기본적으로 안전한 추상화를 제공하지만, 시스템 프로그래밍에서 요구되는 저수준 제어를 완전히 배제하지 않는다. 이를 위해 RUST는 `unsafe` 블록을 도입하여, 다음과 같은 경우에 선택적으로 제어권을 부여한다:

- 외부 FFI 연동 (C API)
- MMIO 접근
- 성능 최적화 루틴
- 커널/드라이버 개발 시 필수적인 저수준 연산

이러한 `unsafe`는 명시적으로 나타나기 때문에, 코드 리뷰와 정적 분석 도구가 추적 가능하다. 예를 들어 MMIO(Memory-Mapped I/O)를 통해 하드웨어 레지스터에 직접 접근해야 하는 경우, 포인터를 사용한 접근이 필수적이며 이는 안전하지 않은 연산으로 분류된다. 이처럼 시스템 프로그래밍에서 불가피한 저수준 연산을 수행할 때, unsafe는 필요한 제어력을 부여하면서도 해당 코드 범위를 명확히 제한할 수 있도록 한다.

```
unsafe {
    let reg_ptr = 0x4000_0000 as *mut u32;
    *reg_ptr = 0xDEADBEEF;
}
```

RUST는 이처럼 기본은 안전성을 유지하되, 시스템 수준 제어가 필요한 경우에만 제한적으로 이를 허용함으로써, 제어력과 안전성 사이의 균형을 지향한다.

9) 스레드 안전성

멀티스레드 환경에서의 안전성은 개발자가 직접 `mutex`, `atomic`, `memory fence` 등을 통해 확보해야 하며, 작은 실수도 race condition으로 이어질 수 있다.

반면, RUST는 `Send`, `Sync` 트레잇, 그리고 빌림 규칙을 통해 데이터 경쟁을 원천적으로 방지한다. 예를 들어 `&mut T`는 해당 데이터에 대한 단일 가변 접근만을 허용하므로, 두 스레드가 동시에 같은 데이터를 가변으로 접근하는 것은 컴파일 타임에 차단된다.

```
use std::thread;
use std::sync::{Arc, Mutex};

fn main() {
    let data = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let data = Arc::clone(&data);
        handles.push(thread::spawn(move || {
            let mut num = data.lock().unwrap();
            *num += 1;
        }));
    }

    for handle in handles {
```

```
        handle.join().unwrap();
    }

    println!("Result: {}", *data.lock().unwrap());
}
```

Table 1 C/C++과 RUST의 비교

항목	C/C++	RUST
메모리 안전성	개발자 책임, 런타임 오류	컴파일타임 검증, 안전 기본값
스레드 안전성	개발자 책임, race condition 위험	타입시스템 기반 안전 보장
성능	스크립트언어대비 매우 높음	동등
저수준 하드웨어접근	개발자 책임	안전한 추상화 기본
런타임 의존성	없음	없음 (GC 미사용, zero-cost abstraction)

시스템 소프트웨어 개발에서 control과 safety는 오랜 시간 상충하는 목표였다. C/C++은 강력한 제어력을 바탕으로 수십 년간 시스템 프로그래밍의 표준이 되었지만, 메모리 오류와 동시성 오류에 대한 구조적 취약점을 극복하지 못했다. 반면, RUST는 소유권과 빌림을 기반으로 하는 정적 분석을 통한 안전성 보장과 선택적 `unsafe` 사용을 통해 control과 safety를 조화롭게 결합하려는 시도를 현실화하였다.

실제 산업계에서도 RUST는 보안 민감 시스템, 임베디드 장치, 커널 모듈, 가상화 플랫폼 등에서 점진적으로 도입되고 있으며, 기존 C/C++ 코드베이스와의 점진적 통합이 가능하다는 점에서 전환 비용도 상대적으로 낮다. RUST는 시스템 프로그래밍 언어의 진화를 대표하는 사례로, 향후 보안성과 신뢰성이 요구되는 컴퓨팅 환경에서 핵심적인 역할을 수행할 것으로 전망된다.

3. 하이퍼바이저 타겟 시스템 구성

타겟 하드웨어 시스템은 동일한 MCU인 ARM Cortex-R52 코아를 탑재한, STMicro사와 NXP사가 시판하는 평가 보드를 사용하였다. STMicro사의 평가 보드는 (STMicro SR6x7탑재) Stellar, 그리고 NXP사의 평가 보드는 (NXP S32E 탑재) GreenBox3이다.

보드의 기본 clock은 300MHz이며, GreenBox3는 가변적으로 800MHz까지 증가를 시키는 것이 가능하다. 성능의 측정을 위하여는 동일한 상기의 MCU향 PEGASUS하이퍼바이저를 평가 보드에 설치하고, 게스트 운영체제로는 동일한 FreeRTOS가 사용되었다. 하이퍼바이저 시스템 전체 블록 다이어그램 구조는 Fig. 2와 같다.

PEGASUS 하이퍼바이저는 ASPICE를 준수하여 개발되고, ISO26262 기능안전 ASIL 수준을 만족하게 개발되었다.

4. 성능 평가

성능은 다음의 항목에 대하여 측정하였으며, 하드웨어 시스템에 공통적으로 발생되는 고유의 오버헤드라고 볼 수 있으며, 또한 하이퍼바이저는 Switching 오버헤드를 증가시키지 않았다.

-. Mode Switching Overhead: (게스트운영체제 동작하는) 가상머신(VM)과 하이퍼바이저모드간 전환 시간
-. Context Switching Overhead: vMCU 컨텍스트 전환 시간
-. Interrupt Virtualization Overhead: 물리인터럽트 수신 후 가상머신에 전달하기까지 걸린 시간

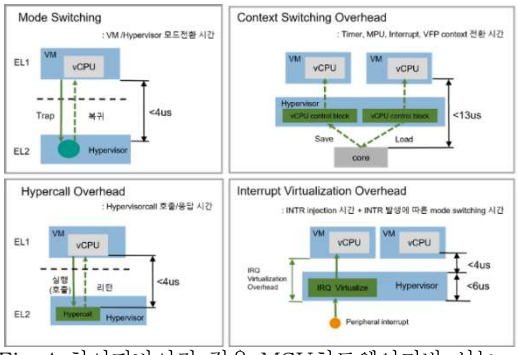

Fig. 4 하이퍼바이저 적용 MCU하드웨어기반 성능

상기의 Fig. 4의 결과는 STMicro Stellar 평가보드와 NXP GreenBox3 평가보드로부터 측정한 결과이며, 동일한 성능 결과값을 얻었다. 이때 평가보드에는 clock을 300MHz로 동일하게 설정하였다.

상기의 측정 외에 NXP GreenBox3는 clock를 800MHz로 올릴 수 있어서, clock을 증가하고 측정한 결과를 Table 2에 정리하였다.

Table 2 하이퍼바이저사양 및 MCU성능비교

Binary Size	Less than 128KB (Position Independent Support) Less than 80KB (Static Linking Only)
Processor Requirement	ARMv8 Cortex-R52 with VT Ext.
Memory Requirement: DRAM or SRAM	8K(Heap) + # of vMCUs * 4KB(Stack)
Exception Mode Switching Overhead	Less than 0.5us @ 800MHz, NXP S32E2xx Less than 4us @ 300MHz, STMicro SR6x7
VM-to-VM Context Switching Overhead	Less than 4us @ 800MHz, NXP S32E2xx Less than 13us @ 300MHz, STMicro SR6x7
Virtual Interrupt Latency	Less than 1.5us @ 800MHz, NXP S32E2xx Less than 6us @ 300MHz, STMicro SR6x7
Device Driver Model	VirtIO + Pass-through + Emulated
Dynamic VM Loading	O
Hypercalls for Guest OS	VM/vMCU Lifecycle, Resource(VirtIO) Management
Scheduling	Round-robin, Priority-based, EDF/RM
Language	RUST

Table 2에는 하이퍼바이저의 바이너리 크기, 하이퍼바이저가 동작하는 데 필요한 DRAM/SRAM 크기(Memory Footprint)를 정리하였다. PEGASUS 하이퍼바이저는 pass-through 모드와 VirtIO 인터페이스로 게스트 운영체제의 IO 디바이스드라이버 모델을 제공한다. 또한 Flash Memory에서 하이퍼바이저 코드를 동작시키는 XiP(Execute In Place) 모드가 제공된다.

5. 결론

하이퍼바이저를 ARM Cortex-R52 MCU 멀티코아 하드웨어에서 동작 시키고 성능을 측정하였다. 결과적으로 하드웨어 고유의 Switching 오버헤드 등에 영향을 주지 않았다. 참고적으로 전원을 끈 상태에서 전원을 켰을 때 측정한 콜드 부팅 시간은 수 ms정도로 빠르게 기동하였다. 하이퍼바이저를 적용하여 기존의(Legacy) 운영체제를 재사용하여 동작시키는 때를 포함하여 여러 운영체제를 하이퍼바이저위에서 동작시키는 것은 성능적으로 이슈가 없다.

또한 병렬 컴퓨팅 시대에 데이터 메모리 시스템의 안전성을 보장하는 RUST언어를 하이퍼바이저를 개발한 경험지식을 공유하였다.

참고적으로 기존의 운영체제(예: 실시간 운영체제, AutoSAR Classic)를 멀티코아 또는 ARM V7/V8기반의 최신 MCU구조 SoC에 적용시 가이드를 다음과 같이 정리하였다. 기존의 운영체제들은 EL1및 EL2 Exception mode를 구분하지 못하는 이슈가 있으니, 최신 멀티코아 MCU SoC에 적용시 이를 확인하는 것이 필요하며, 또한 AutoSAR Classic은 20년전에 개발된 사양으로 개발된 것이 많아, 멀티코아 MCU에 적용시 운영체제별로 하드웨어 clock을 초기화하는 이슈를 해결하며 적용하는 것이 필요하다.

본 연구에서는 위와 같은 이슈를 다 해결하고 상기와 같은 가이드를 공유한다.

References

1) Programming Languages and Safety-Related Systems, Les Hatton, 14th International Conference on Computer Safety 1995, Reliability and Security, Lecture Notes in Computer Science Volume 976, pp. 39-46, Springer-Verlag

2) Fine-grained I/O access control of the mobile devices based on the Xen architecture, Sung-Min Lee, Sang-Bum Suh, et. al, Proceedings of the 15th annual international conference on Mobile computing and networking (MobiCom '09)

3) Secure Architecture and Implementation of Xen on ARM for Mobile Devices, Sang-bum Suh, Xen Summit North America at IBM TJ Watson, April 17 2007

Author biography

1) 서상범
PhD, Computer Science,
University of Cambridge
현) PERSEUS 대표이사
전)삼성전자 상무이사

2) 문태현
연세대학교 전자공학 박사
현) PERSEUS SW연구소장

3) 정지원
국민대학교 소프트웨어학과
현) PERSEUS 대구 SW
R&D센터장