PYNQ 프레임워크 심층 분석: HWH 파싱, C++ 추상화, 원격 통신 프로토콜

제 1부: HWH 메타데이터로부터의 동적 하드웨어 추상화

PYNQ 프레임워크의 생산성을 뒷받침하는 핵심 기술은 하드웨어 메타데이터로부터 소프트웨어 인터페이스를 동적으로 생성하는 메커니즘입니다. 이 기능은 복잡한 하드웨어 설계를 추상화하여 소프트웨어 개발자가 파이썬 객체처럼 다룰 수 있게 해줍니다. 본 분석에서는 AMD Vivado 설계 도구에서 생성된 파일이 PYNQ 환경 내에서 완전한 형태의 파이썬 객체 모델로 변환되는 과정을 추적합니다.

1.1 하드웨어 핸드오프(HWH) 파일의 중심성

메타데이터 파일은 생산적인 FPGA 프레임워크에서 필수불가결한 요소입니다. 이러한 파일이 없다면, 개발자는 메모리 주소와 같은 하드웨어 정보를 "매직 넘버(magic number)" 형태로 코드에 직접 하드코딩해야 합니다.[1] 이 방식은 오류 발생 가능성이 높고, 하드웨어 빌드가 변경될 때마다 코드를 수정해야 하므로 이식성이 떨어집니다. HWH 파일은 소프트웨어 스택이 인식하는 하드웨어 설계의 "단일 진실 공급원(single source of truth)" 역할을 수행하여 이러한 문제를 해결합니다.

PYNQ는 초기 버전에서 Vivado 빌드 과정의 일부였던 TCL(Tool Command Language) 파일을 메타데이터 소스로 사용했습니다.[1] 그러나 프레임워크가 발전함에 따라, Vivado가 구현(implementation) 단계에서 자동으로 생성하는 보다 구조화된 XML 기반 형식인 HWH 파일로 전환했습니다.[1] 이러한 변화는 더 강력하고 표준화된 메타데이터 수집 파이프라인으로의 전환을 의미하며, PYNQ 버전 2.6.0부터는 TCL 파싱 지원이 공식적으로 중단되었습니다.[3]

HWH 파일은 시스템에 대한 완전한 설명을 포함하는 XML 형식의 문서입니다. 여기에는 IP(Intellectual Property) 인스턴스, 해당 파라미터(VLNV - Vendor, Library, Name, Version), 메모리 맵 주소 세그먼트, 레지스터 정의, 인터럽트 연결 등 하드웨어 설계의 모든 세부 정보가

포함됩니다.[1] 이 파일은 사실상 프로그래머블 로직(PL) 설계의 디지털 청사진 역할을 합니다.

1.2 파싱 엔진: pynq.pl_server.hwh_parser 심층 분석

hwh_parser 모듈은 HWH 파일을 파이썬 네이티브 데이터 구조로 변환하는 핵심 엔진입니다.[5] 이 모듈의 구현을 분석해 보면, 파이썬의 표준 라이브러리인

xml.etree.ElementTree를 사용하여 HWH 파일의 복잡한 XML 트리를 탐색하는 것을 확인할 수 있습니다.[6]

소스 코드 분석을 통해 파싱 과정을 추적할 수 있습니다. 프로세스는 사용자가 오버레이를 로드할 때 .bit 파일에 해당하는 .hwh 파일의 이름을 찾는 get_hwh_name() 함수 호출로 시작됩니다.[5] 이후, 핵심 파싱 로직은 XML 요소를 순회하며 모듈, 메모리 인터페이스, 주소 블록, 레지스터 등을 식별합니다. 이 과정에서 각 IP의 메모리 맵을 결정하기 위해

C_BASEADDR 및 C_HIGHADDR와 같은 중요한 속성을 추출합니다.

이 파서는 계층적 블록이나 부분 재구성(Partial Reconfiguration, PR) 영역과 같은 복잡한 설계 구조도 처리할 수 있도록 설계되었습니다. 예를 들어, PR 블랙박스와 같이 불완전한 정보가 포함된 경우, 파서는 특정 가정을 기반으로 메모리 인터페이스 ID를 사용하여 IP 인스턴스를 구별합니다.[6] 이는 파서가 단순하지 않은 고급 하드웨어 설계를 처리할 수 있는 견고성을 갖추고 있음을 보여줍니다. HWH 파일의 파싱은 PYNQ가 하드웨어의 물리적 구현을 소프트웨어적 표현으로 변환하는 첫 번째이자 가장 중요한 단계입니다. 이 번역 과정이 없다면,

overlay.ip_name과 같은 PYNQ의 직관적인 구문은 불가능하며, 사용자는 주소를 직접 찾아 하드코딩하는 수동적이고 오류가 발생하기 쉬운 방식으로 회귀할 수밖에 없습니다.[1] 따라서

hwh_parser.py는 PYNQ 생태계의 하드웨어 언어(HWH)와 소프트웨어 언어(파이썬 객체) 사이의 간극을 메우는 결정적인 번역기, 즉 "로제타석"과 같은 역할을 수행합니다.

1.3 메타데이터에서 메모리로: ip_dict와 동적 드라이버 바인딩

HWH 파서의 주요 출력물은 ip_dict라는 파이썬 딕셔너리입니다. 이 딕셔너리는 오버레이 내에서 프로세서 시스템(PS)이 접근할 수 있는 모든 IP의 중앙 레지스트리 역할을 합니다.[1] PYNQ 문서에 따르면,

ip_dict의 각 항목은 IP의 이름을 키(key)로 가지며, 값(value)으로는 해당 IP의 물리적

주소(phys_addr), 주소 범위(addr_range), IP 유형(type), 구성 파라미터(config), 그리고 상태(state)를 포함하는 또 다른 딕셔너리를 가집니다.[8]

PYNQ의 핵심적인 기능은 이 ip_dict를 통해 동적으로 객체를 생성하는 것입니다. 사용자가 파이썬 코드에서 ol = Overlay('base.bit')와 같이 Overlay 객체를 인스턴스화하면, 내부적으로 HWH 파서가 실행되어 ip_dict가 채워집니다.[1] 그 후,

Overlay 클래스는 ip_dict의 키에 해당하는 속성들을 동적으로 노출합니다. 예를 들어, ip_dict에 'my_axi_gpio_0'라는 키가 있다면, 사용자는 ol.my_axi_gpio_0을 통해 해당 IP에 즉시 접근할 수 있습니다.

이 메커니즘은 적절한 드라이버를 바인딩하는 데에도 사용됩니다. 만약 특정 파이썬 드라이버 클래스가 ip_dict에서 발견된 IP 유형과 일치하는 bindto 속성을 가지고 있다면, 해당 드라이버의 인스턴스가 생성됩니다. 일치하는 특정 드라이버가 없는 경우에는 일반적인 DefaultIP 드라이버가 사용되며, 이는 기본적인 메모리 맵 입출력(MMIO) 읽기/쓰기 기능을 제공합니다.[8] 이 우아한 메커니즘은 확장 가능한 드라이버 모델을 가능하게 하여 PYNQ의 유연성을 크게 향상시킵니다. 다음 표는 HWH 파일의 XML 요소가

ip_dict의 키와 값으로 어떻게 매핑되는지를 요약한 것입니다.

표 **1: HWH** 요소와 **ip_dict** 키 매핑

HWH XML 요소/속성 (간소화된 경로)	ip_dict 키	설명
IP_INSTANCE/VLNV	type	IP 코어를 식별하는 Vendor-Library-Name-Version 문자열입니다.
ADDRBUS_SLAVE/C_BASEADDR	phys_addr	IP의 메모리 맵 영역의 시작 물리 주소입니다.
ADDRBUS_SLAVE/C_HIGHADDR	(addr_range 계산에 사용)	IP의 메모리 맵 영역의 끝 물리 주소입니다.
REG/NAME	config['registers'][reg_name]	IP 내의 특정 레지스터 이름입니다.
REG/OFFSET	config['registers'][reg_name]['address_offset']	IP의 기본 주소로부터의 레지스터 오프셋입니다.

GPIO_INSTANCE/PIN_NUMBER	gpio_dict	IP에 연결된 GPIO 핀에 대한 정보입니다.
IRQ_INSTANCE/PIN_NUMBER	interrupt_pins	IP에 연결된 인터럽트 라인에 대한 정보입니다.

1.4 실제적 고려사항 및 실패 지점

HWH 파싱 기반 접근 방식은 매우 강력하지만, 몇 가지 실제적인 제약과 잠재적인 실패 지점을 내포하고 있습니다. 가장 큰 약점 중 하나는 HWH 파일 형식과 파서 간의 강한 결합(tight coupling)입니다. PYNQ 커뮤니티 포럼의 논의를 살펴보면, 사용자들이 PYNQ가 검증된 Vivado 버전보다 최신 버전을 사용할 때 파싱 오류(AttributeError: 'NoneType' object has no attribute 'get')를 겪는 사례가 다수 보고되었습니다.[12] 이는 Vivado 릴리스 간에 HWH 파일의 XML 스키마가 안정적으로 유지된다는 보장이 없기 때문입니다.

이러한 의존성은 PYNQ 개발자와 사용자 모두에게 "호환성 쳇바퀴(compatibility treadmill)"를 만듭니다. PYNQ 팀은 새로운 Vivado 버전을 지원하기 위해 지속적으로 파서를 업데이트해야 하며, 이는 새로운 PYNQ 릴리스로 이어집니다.[3] 사용자 입장에서는 중요한 버그 수정이나 성능 향상을 제공할 수 있는 최신 Vivado 도구를 항상 사용할 수 없으며, 자신이 사용하는 PYNQ 릴리스와 호환되는 특정 버전의 도구 체인을 사용해야만 합니다.[14] 이는 개발 과정에서 상당한 마찰을 유발하는 실질적인 제약 조건입니다.

부분 재구성(PR)과 관련된 문제 또한 보고되었습니다. 파서가 PR을 처리하기 위한 로직을 포함하고 있음에도 불구하고, 재구성 가능한 파티션 내의 IP에 대한 레지스터 맵이 HWH 파일에 올바르게 생성되지 않아 PYNQ가 해당 레지스터에 접근하지 못하는 경우가 발생합니다.[15] 이 경우 사용자는 자체 드라이버를 작성하거나 HWH 파일을 수동으로 수정하는 등의 해결 방법을 찾아야 하며, 이는 프레임워크의 주요 목표인 생산성을 저해합니다.

마지막으로, 파서는 암묵적으로 특정 설계 규칙을 강제합니다. 예를 들어, 인터럽트는 AXI 인터럽트 컨트롤러를 통해 라우팅되어야 하며, PS에 직접 연결된 인터럽트가 포함된 설계는 파싱에 실패할 수 있습니다.[13] 마찬가지로, PS의 주소 공간에 연결되지 않은 IP는

ip_dict에 포함되지 않으므로 PYNQ를 통해 접근할 수 없습니다.[13]

제 **2**부**: PYNQ.cpp** 하드웨어 추상화 계층

PYNQ 생태계는 파이썬의 생산성을 넘어 고성능 C++ 백엔드로의 확장을 모색하고 있습니다. PYNQ.cpp는 이러한 움직임의 핵심으로, PYNQ의 하드웨어 추상화 개념을 C++ 환경으로 가져와 성능이 중요한 애플리케이션과 새로운 원격 아키텍처를 지원합니다.

2.1 개발 동기 및 아키텍처적 위치

PYNQ.cpp의 개발 동기는 두 가지 주요 요구 사항에서 비롯되었습니다. 첫째, 특정 고속 데이터 전송 시나리오에서 순수 파이썬 드라이버의 성능이 병목 현상을 일으킬 수 있습니다.[17] C++로 구현된 저수준 라이브러리는 이러한 성능 한계를 극복할 수 있는 해결책을 제공합니다. 둘째, 그리고 더 전략적으로,

PYNQ.cpp는 새로운 PYNQ.remote 아키텍처를 위한 필수적인 온-타겟(on-target) C++ 백엔드로 개발되었습니다. 이를 통해 타겟 디바이스의 소프트웨어 설치 공간을 최소화하면서도 고성능을 유지할 수 있습니다.[18]

PYNQ는 이전부터 CFFI(C Foreign Function Interface)를 사용하여 파이썬에서 C 공유 라이브러리를 호출하거나 [17], Pybind11을 사용하는 등 다양한 방식으로 C/C++ 통합을 지원해왔습니다.[3]

PYNQ.cpp는 이러한 기능을 PYNQ 고유의 개념을 반영하는 독립적인 C++ 라이브러리로 공식화한 것입니다.[19] 이 라이브러리는

PYNQ.remote와 독립적으로 사용될 수 있어, 개발자가 저수준 C++ 애플리케이션을 직접 제작하는 것을 가능하게 합니다.[19]

이러한 아키텍처적 변화는 PYNQ를 단순한 독립형 프로토타이핑 환경에서 확장 가능한 분산 엣지 컴퓨팅 프레임워크로 전환시키는 전략적 전환을 의미합니다. 기존의 PYNQ 모델은 개발 환경(Jupyter), 파이썬 런타임, 리눅스 OS가 모두 Zynq 보드에 통합된 모놀리식 구조였습니다.[21]

PYNQ.remote는 호스트와 타겟을 분리하고 [18],

PYNQ.cpp는 타겟에서 고성능, 저용량 C++ 엔진을 제공합니다. 이 새로운 아키텍처는 강력한 호스트 머신이 여러 개의 경량 PYNQ 타겟을 조율하는 "하이브리드 워크플로우(hybrid workflows)"를 가능하게 하여 [18], 분산 센싱, 추론, 실시간 제어와 같은 실제 제품 배포 시나리오에 더 적합한 모델을 제시합니다.

2.2 핵심 API 및 구현

PYNQ.cpp API는 하드웨어와 상호작용하기 위한 핵심 클래스들로 구성되어 있습니다.[23] 소스 코드 분석을 통해 다음과 같은 주요 기능을 담당하는 클래스를 식별할 수 있습니다.

- 디바이스 관리: 디바이스 컨텍스트를 열고 닫는 등 하드웨어 리소스에 대한 접근을 관리합니다.
- 메모리 맵 입출력 (MMIO): 특정 물리 메모리 주소에 데이터를 읽고 쓰는 함수를 제공합니다. 이는 대부분의 PL IP 코어를 제어하는 근본적인 메커니즘입니다.
- 버퍼 관리: 시스템의 메모리 할당자와 상호작용하여 물리적으로 연속적인 메모리 버퍼를 제공합니다. 이는 직접 메모리 접근(DMA) 작업에 필수적입니다.

이 라이브러리는 PYNQ의 친숙한 개념들을 C++ 개발자에게 제공하도록 설계되었습니다.[19] 예를 들어,

PYNQ.cpp의 C++ mmio_write 함수는 파이썬의 mmio.write() 메서드와 개념적으로 동일한 역할을 수행하며, 이를 통해 개발자는 기존 PYNQ 지식을 C++ 환경에서도 활용할 수 있습니다.

2.3 Xilinx 런타임(XRT)과의 상호작용

XRT(Xilinx Runtime)는 엣지 디바이스부터 데이터센터의 Alveo 가속기 카드에 이르기까지, Xilinx 플랫폼을 관리하고 상호작용하기 위한 표준화된 API를 제공하는 통합 런타임입니다.[24] PYNQ는 기존의 자체 메모리 관리자인

xlnk를 더 이상 사용하지 않고 XRT를 채택했습니다.[3]

PYNQ.cpp는 XRT 위에 구축된 고수준 추상화 계층으로 작동합니다. 즉, PYNQ.cpp는 메모리 할당이나 디바이스 핸들 관리와 같은 저수준 작업을 위해 내부적으로 XRT API를 호출합니다. 이에 대한 명확한 증거는 문서에 제공된 예제 빌드 명령어에서 찾을 수 있습니다. 이 명령어는 XRT 헤더(-I$XRT_INCLUDE)를 명시적으로 포함하고 XRT 라이브러리(-L$XRT_LIB)에 대해 링크합니다.[25] 이는

PYNQ.cpp와 XRT 간의 명확한 의존성과 계층적 아키텍처를 보여줍니다.

XRT를 PYNQ.cpp의 기본 런타임으로 채택한 것은 Xilinx 소프트웨어 스택의 통합을 의미하는 중요한 결정입니다. 이는 엣지 중심의 PYNQ 생태계와 데이터센터 중심의 Alveo 생태계를 기술적으로 정렬시킵니다. XRT는 Vitis 및 Alveo 플랫폼의 기반 기술이므로 [24], PYNQ가 XRT에 의존하게 함으로써 Xilinx는 보다 일관된 개발자 경험을 창출하고 있습니다. 이는 XRT API 수준에서 작성된 기술과 일부 코드가 엣지(Zynq/Kria)와 클라우드(Alveo) 플랫폼 간에 이식

가능해질 수 있음을 시사하며, 개발자에게는 더 강력하고 통일된 생태계를 제공하는 전략적 이점을 가집니다.

2.4 개발자 워크플로우: PYNQ.cpp 애플리케이션 빌드 및 링크

PYNQ.cpp를 사용하려는 개발자를 위한 실용적인 가이드는 다음과 같습니다. 개발 환경 설정에는 PYNQ.cpp와 XRT 모두에 대한 헤더 경로를 포함하고, 필요한 공유 라이브러리에 대해 링크하는 과정이 포함됩니다.

문서에 제공된 g++ 명령어는 구체적인 템플릿으로 활용될 수 있습니다.[25]

C++

```
g++ -std=c++17 -I<path_to_pynq_cpp_include> -I/opt/xilinx/xrt/include -L/opt/xilinx/xrt/lib -o my_app my_app.cpp -lpynqcpp -lxrt_coreutil
```

이 명령어의 각 구성 요소는 다음과 같은 역할을 합니다.

- g++ -std=c++17: C++17 표준을 사용하여 코드를 컴파일합니다.
- -I<path_to_pynq_cpp_include>: PYNQ.cpp 라이브러리의 헤더 파일이 위치한 디렉토리를 포함합니다.
- -I/opt/xilinx/xrt/include: XRT의 헤더 파일 경로를 지정합니다.
- -L/opt/xilinx/xrt/lib: XRT 공유 라이브러리가 위치한 디렉토리를 지정합니다.
- -o my_app my_app.cpp: 소스 파일 my_app.cpp를 컴파일하여 my_app이라는 실행 파일을 생성합니다.
- -lpynqcpp -lxrt_coreutil: PYNQ.cpp 라이브러리(libpynqcpp.so)와 XRT 코어 유틸리티 라이브러리(libxrt_coreutil.so)를 링크합니다.

개발자는 이 명령어를 자신의 프로젝트에 맞게 경로와 파일명을 수정하여 사용할 수 있습니다.

제 3부: PYNQ.remote 통신 프로토콜

PYNQ.remote는 PYNQ의 실행 모델을 근본적으로 바꾸는 확장 기능으로, 네트워크를 통해 PYNQ 디바이스를 원격으로 제어할 수 있게 합니다. 이 기능의 핵심에는 고성능 원격 프로시저

호출(RPC) 프레임워크인 gRPC가 있습니다. 본 파트에서는 PYNQ.remote의 통신 프로토콜을 심층적으로 분석하고, gRPC의 선택 이유와 API 계약 구조를 살펴봅니다.

3.1 아키텍처 청사진: gRPC 기반 클라이언트-서버 모델

합니다. 호스트에서 실행되는 파이썬 코드는 클라이언트 역할을 하며, 네트워크를 통해 타겟 PYNQ 디바이스에서 실행되는 서버와 통신합니다.[18]

이 시스템은 다음과 같은 네 가지 주요 구성 요소로 이루어집니다.

1. 호스트 클라이언트: 원격 디바이스에 연결하는 로직이 포함된 pynq 파이썬 라이브러리. 사용자의 파이썬 스크립트는 이곳에서 실행됩니다.
2. 네트워크 프로토콜: 고성능 오픈소스 RPC 프레임워크인 gRPC(gRPC Remote Procedure Calls).[18]
3. 타겟 서버: PYNQ 보드의 최소화된 Petalinux 이미지에서 실행되는 gRPC 서버 애플리케이션.
4. 타겟 백엔드: gRPC 서버로부터 요청을 받아 실제 하드웨어 작업을 실행하는 PYNQ.cpp 라이브러리.[18]

이 아키텍처는 타겟 디바이스의 소프트웨어 설치 공간을 획기적으로 줄여줍니다. 전체 PYNQ 이미지가 약 7GB에 달하는 반면, PYNQ.remote 이미지는 200MB 미만으로, 리소스가 제한된 애플리케이션에 매우 적합합니다.[18] 또한, 복잡한 연산은 호스트에서 처리하고 특화된 작업만 FPGA로 오프로드하는 효율적인 작업 분배를 가능하게 합니다.[18]

REST/JSON API와 같은 더 간단한 대안 대신 gRPC를 선택한 것은 성능, 유형 안전성(type safety), 그리고 장기적인 생태계 확장성을 우선시한 의도적인 아키텍처 결정입니다. 프로토콜 버퍼(Protobuf)의 바이너리 직렬화는 텍스트 기반 JSON보다 더 작고 파싱이 빨라 네트워크 지연 시간을 줄여줍니다.[26]

.proto 계약의 강력한 유형 지정은 느슨한 유형의 JSON에서 흔히 발생하는 런타임 오류를 방지합니다. 무엇보다도, gRPC의 교차 언어 코드 생성 기능 [27]과 API 확장에 대한 명시적 지원 [18]은 PYNQ 디바이스가 다양한 호스트 언어에 의해 제어되고 다국어(polyglot) 시스템에 통합될 수 있는 미래 지향적인 프레임워크를 구축하려는 장기적인 비전을 시사합니다.

3.2 서비스 계약: PYNQ .proto 파일 분석

gRPC는 프로토콜 버퍼(Protobuf)를 인터페이스 정의 언어(IDL)로 사용합니다.[27] 클라이언트와

서버 간의 API 계약은 언어에 구애받지 않는

.proto 파일에 정의됩니다.

PYNQ 저장소의 소스 코드에서 직접 .proto 파일을 찾을 수는 없지만, PYNQ의 핵심 기능을 기반으로 그 구조를 추론할 수 있습니다. PynqRemote라는 이름의 서비스는 다음과 같은 핵심 PYNQ 작업을 위한 RPC 메서드를 포함할 것으로 예상됩니다.

- rpc DownloadBitstream(BitstreamRequest) returns (BitstreamResponse) {}
- rpc ReadRegister(ReadRequest) returns (ReadResponse) {}
- rpc WriteRegister(WriteRequest) returns (WriteResponse) {}
- rpc AllocateBuffer(AllocateRequest) returns (AllocateResponse) {}
- rpc FreeBuffer(BufferHandle) returns (FreeResponse) {}
- rpc TransferToDevice(stream DataChunk) returns (BufferHandle) {}
- rpc TransferFromDevice(BufferHandle) returns (stream DataChunk) {}

이러한 RPC 메서드에서 사용되는 메시지 구조 역시 proto3 구문을 사용하여 정의할 수 있습니다.[26] 예를 들어,

WriteRequest 메시지는 IP 식별자, 주소 오프셋, 그리고 쓸 데이터를 필드로 포함할 것입니다. 아래 표는 추론된 gRPC 서비스의 구체적인 정의를 보여줍니다.

표 **2: PYNQ.remote gRPC** 서비스 정의 **(**추론**)**

RPC 메서드 이름	요청 메시지	응답 메시지	설명
WriteRegister	WriteRequest {string ip_name; uint32 offset; uint32 data;}	StatusResponse {bool success;}	특정 IP의 레지스터에 32비트 값을 씁니다.
ReadRegister	ReadRequest {string ip_name; uint32 offset;}	ReadResponse {uint32 data;}	특정 IP의 레지스터에서 32비트 값을 읽습니다.
Allocate	AllocateRequest {uint64 size; bool contiguous;}	BufferResponse {uint64 handle; uint64 phys_addr;}	타겟 디바이스에 메모리 버퍼를 할당합니다.
TransferToDevice	stream DataChunk {uint64 handle;	StatusResponse {bool success;}	호스트에서 타겟의 사전 할당된 버퍼로 데이터를

	bytes data;}		스트리밍합니다.
TransferFromDevice	BufferRequest {uint64 handle;}	stream DataChunk {bytes data;}	타겟의 버퍼에서 호스트로 데이터를 스트리밍합니다.

3.3 전체 트랜잭션 추적**:** 파이썬 호출부터 하드웨어 실행까지

overlay.my_ip.write(0x10, 0xFF)와 같은 간단한 원격 작업을 추적하여 전체 통신 과정을 이해할 수 있습니다.

1. 파이썬 호출 **(**호스트**):** 사용자의 파이썬 코드가 DefaultIP 객체의 write 메서드를 호출합니다.
2. 원격 디바이스 스텁**:** 이 호출은 gRPC 클라이언트 스텁 역할을 하는 RemoteDevice 클래스에 의해 가로채집니다.
3. 직렬화**:** gRPC 클라이언트 라이브러리는 {ip_name: "my_ip", offset: 0x10, data: 0xFF}와 같은 WriteRequest Protobuf 메시지를 생성하고 이를 압축된 바이너리 형식으로 직렬화합니다.
4. 네트워크 전송**:** 직렬화된 메시지는 네트워크를 통해 PYNQ 보드의 gRPC 서버로 전송됩니다.
5. 역직렬화 **(**타겟**):** gRPC 서버는 바이너리 데이터를 수신하고 이를 다시 WriteRequest 객체로 역직렬화합니다.
6. 백엔드 실행**:** WriteRegister RPC 메서드를 처리하는 서버 핸들러는 역직렬화된 파라미터를 사용하여 PYNQ.cpp 라이브러리의 해당 함수를 호출합니다.
7. 하드웨어 접근**:** PYNQ.cpp는 my_ip의 기본 주소에 오프셋 0x10을 더한 물리 주소에 실제 mmio_write 작업을 수행합니다.
8. 응답**:** {success: true}와 같은 WriteResponse 메시지가 생성되고, 직렬화되어 호스트로 다시 전송되어 RPC 호출이 완료됩니다.

이러한 PYNQ.remote 아키텍처는 시스템을 고전적인 제어 평면(control plane)과 데이터 평면(data plane)으로 효과적으로 분할합니다. 호스트 측 파이썬 환경은 비트스트림 다운로드, DMA 전송 설정, 복잡한 알고리즘 실행 등 상위 수준의 로직, 구성 및 조정을 담당하는 "제어 평면" 역할을 합니다. 반면, 타겟의 PYNQ.cpp와 FPGA 하드웨어 자체는 특정 작업의 고속, 저지연 실행을 담당하는 "데이터 평면"을 형성합니다. 이러한 분리는 고성능 네트워킹 및 컴퓨팅에서 표준적인 패턴으로, PYNQ를 단순한 모놀리식 임베디드 시스템을 넘어 더 복잡하고 분산된 컴퓨팅 토폴로지의 구성 요소로 격상시켜 훨씬 더 강력하고 확장 가능한 패러다임을 제공합니다.

3.4 확장성 및 사용자 정의

PYNQ 문서에서 강조하는 주요 기능 중 하나는 사용자가 자신만의 프로토콜 버퍼 정의를 사용하여 원격 API를 확장할 수 있다는 점입니다.[18]

이를 위한 개발자 워크플로우는 다음과 같습니다. 먼저, 개발자는 새로운 .proto 파일에 자신만의 서비스와 메시지를 정의합니다. 그런 다음 protoc 컴파일러를 사용하여 C++ 서버 측 스텁과 파이썬 클라이언트 측 스텁을 생성합니다. 마지막으로, 타겟에서 PYNQ.cpp를 사용하여 새로운 서비스 로직을 구현하고, 호스트 측 파이썬 애플리케이션에서 새로운 RPC를 호출합니다. 이 과정은 애플리케이션별 원격 기능을 추가하는 강력하고 구조화된 방법을 제공하여 PYNQ.remote의 활용 가능성을 크게 넓혀줍니다.

결론

본 보고서는 PYNQ 프레임워크의 세 가지 핵심 구성 요소인 동적 하드웨어 추상화, C++ 구현, 그리고 원격 통신 프로토콜에 대한 심층적인 분석을 제공했다.

첫째, HWH 파일 파싱 메커니즘은 PYNQ 생산성의 근간을 이룬다. hwh_parser는 Vivado가 생성한 하드웨어의 디지털 청사진을 파이썬 객체 모델로 자동 변환함으로써, 개발자가 복잡한 하드웨어 주소나 레지스터 오프셋을 직접 다룰 필요 없이 overlay.ip_name과 같은 직관적인 구문을 통해 PL과 상호작용할 수 있게 한다. 그러나 이 방식은 Vivado 도구 체인 버전과의 강한 결합이라는 내재적 한계를 가지며, 이는 사용자에게 호환성 문제를 야기할 수 있다.

둘째, PYNQ.cpp의 도입은 PYNQ가 단순한 프로토타이핑 도구를 넘어 성능 중심의 배포 가능한 프레임워크로 진화하고 있음을 보여준다. XRT를 기반으로 구축된 이 C++ 하드웨어 추상화 계층은 고성능이 요구되는 애플리케이션을 지원할 뿐만 아니라, PYNQ.remote 아키텍처의 핵심 백엔드 역할을 수행한다. 이는 Xilinx의 엣지 및 데이터센터 소프트웨어 스택을 통합하려는 더 큰 전략의 일환으로 해석될 수 있다.

셋째, gRPC를 기반으로 하는 PYNQ.remote는 PYNQ의 실행 모델을 근본적으로 변화시킨다. 호스트와 타겟을 분리하는 클라이언트-서버 아키텍처는 타겟 디바이스의 소프트웨어 설치 공간을 크게 줄이고, 제어 평면과 데이터 평면을 분리하여 보다 정교하고 확장 가능한 분산 시스템 설계를 가능하게 한다. gRPC와 프로토콜 버퍼의 채택은 성능, 유형 안전성, 그리고 장기적인 확장성을 고려한 전략적 선택이다.

종합적으로, PYNQ 프레임워크는 HWH 파싱을 통한 동적 추상화에서 시작하여, PYNQ.cpp와 PYNQ.remote를 통해 고성능, 저용량, 분산형 엣지 컴퓨팅 플랫폼으로 발전하고 있다. 이러한

진화는 소프트웨어 개발자가 FPGA의 성능을 보다 쉽고 효율적으로 활용할 수 있도록 하는 PYNQ의 핵심 철학을 유지하면서, 동시에 실제 산업 현장에서의 배포 및 확장 요구 사항을 충족시키기 위한 전략적 방향성을 명확히 보여준다.

참고 자료

1. What's in My Bitstream - A Pythonic Approach to Discovering FPGA Contents - Learn - PYNQ, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/whats-in-my-bitstream-a-pythonic-approach-to-discovering-fpga-contents/653
2. Python Productivity for Zynq - GitHub, 9월 16, 2025에 액세스, https://raw.githubusercontent.com/Xilinx/PYNQ_Workshop/master/01_PYNQ_Workshop_introduction.pdf
3. Releases · Xilinx/PYNQ - GitHub, 9월 16, 2025에 액세스, https://github.com/Xilinx/PYNQ/releases
4. PYNQ: advanced example on GitHub - MakarenaLabs, 9월 16, 2025에 액세스, https://www.makarenalabs.com/advanced-pynq-example-on-github/
5. pynq.pl_server.hwh_parser Module, 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v2.5.1/pynq_package/pynq.pl_server/pynq.pl_server.hwh_parser.html
6. pynq/pl_server/hwh_parser.py · image_v2.5.1 - GitLab at Illinois, 9월 16, 2025에 액세스, https://gitlab-03.engr.illinois.edu/paulrr2/PYNQ/-/blob/image_v2.5.1/pynq/pl_server/hwh_parser.py?ref_type=heads
7. Overlay — Python productivity for Zynq (Pynq) - Read the Docs, 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v2.6.1/pynq_libraries/overlay.html
8. pynq.overlay Module — Python productivity for Zynq (Pynq) v1.0 - Read the Docs, 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v2.1/pynq_package/pynq.overlay.html
9. Tutorial: Using a new hardware design with PYNQ (AXI GPIO) - Learn, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/tutorial-using-a-new-hardware-design-with-pynq-axi-gpio/146
10. Create a custom PYNQ overlay for PYNQ-Z1 - FPGA Developer, 9월 16, 2025에 액세스, https://www.fpgadeveloper.com/2018/03/create-a-custom-pynq-overlay-for-pynq-z1.html/
11. Blog #3: Third Eye for Blind - How to Make and Modify a PYNQ Overlay, 9월 16, 2025에 액세스, https://community.element14.com/challenges-projects/design-challenges/eye-on-intelligence-challenge/b/blog/posts/blog-3-third-eye-for-blind---how-to-make-and-modify-a-pynq-overlay
12. Problem with .hwh parsing - Support - PYNQ, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/problem-with-hwh-parsing/2179
13. Parse hwh file failed - Support - PYNQ, 9월 16, 2025에 액세스,

https://discuss.pynq.io/t/parse-hwh-file-failed/2975
14. PortNotFound when downloading/parsing 2023.2 hwh, design ok in 22.1 - Support - PYNQ, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/portnotfound-when-downloading-parsing-2023-2-hwh-design-ok-in-22-1/6472
15. PYNQ, Partial Reconfiguration, Part 2 - DJ Park, 9월 16, 2025에 액세스, https://dj-park.github.io/posts/2022/1/PYNQ_PR_2/
16. Parse hwh file failed - #5 by marioruiz - Support - PYNQ, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/parse-hwh-file-failed/2975/5
17. Python-C Integration - Read the Docs, 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v2.3/overlay_design_methodology/python-c_integration.html
18. PYNQ.remote — Python productivity for Zynq (Pynq) - Read the Docs, 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/pynq_remote.html
19. PYNQ.cpp — Python productivity for Zynq (Pynq), 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/latest/pynq_remote/cpp_index.html
20. Dev flow to combine C++ and Python drivers - Support - PYNQ, 9월 16, 2025에 액세스, https://discuss.pynq.io/t/dev-flow-to-combine-c-and-python-drivers/3904
21. PYNQ | Python Productivity for AMD Adaptive Computing platforms, 9월 16, 2025에 액세스, http://www.pynq.io/
22. PYNQ-Z1 Reference Manual - Digilent, 9월 16, 2025에 액세스, https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual
23. PYNQ.cpp C++ API — Python productivity for Zynq (Pynq) - Read the Docs, 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/pynq_remote/pynq_cpp/cpp_api.html
24. pynq - PyPI, 9월 16, 2025에 액세스, https://pypi.org/project/pynq/
25. How to build projects with PYNQ.cpp — Python productivity for Zynq (Pynq) - Read the Docs, 9월 16, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/pynq_remote/pynq_cpp/cpp_install.html
26. Grpc Basics: Creating a protobuf file (.proto) | by Rahul Gubiligari - Medium, 9월 16, 2025에 액세스, https://medium.com/@roystatham3003/grpc-basics-creating-a-protobuf-file-proto-a80f02e0143b
27. Introduction to gRPC, 9월 16, 2025에 액세스, https://grpc.io/docs/what-is-grpc/introduction/
28. Proto.Remote - Proto.Actor, 9월 16, 2025에 액세스, https://proto.actor/docs/remote/
29. Overview | Protocol Buffers Documentation, 9월 16, 2025에 액세스, https://protobuf.dev/overview/
30. Use gRPC and Python for Remote Procedural Calls | Linode Docs, 9월 16, 2025에 액세스, https://www.linode.com/docs/guides/using-grpc-for-remote-procedural-calls/
31. Language Guide (proto 3) | Protocol Buffers Documentation, 9월 16, 2025에 액세스, https://protobuf.dev/programming-guides/proto3/