Review Questions and Solutions
Dynamic IoT Applications and Isomorphic IoT Systems Using WebAssembly

Uddhav P. Gautam, Ram Mude
upgautam@vt.edu, ramm@vt.edu

Case Study Information

Case Study: Kuribayashi, K., Miyake, Y., Rikitake, K., Tanaka, K., & Shinoda, Y. (2023). Dynamic IoT Applications and Isomorphic IoT Systems Using WebAssembly. 2023 IEEE 9th World Forum on Internet of Things (WF-IoT).

Key Focus Areas: Rapid IoT device updates, multi-platform development, WebAssembly-based isomorphic systems, machine learning model deployment, performance evaluation.

Question 1: Technical Architecture

Question: Explain how the proposed dynamic IoT application architecture enables faster updates compared to traditional firmware updates. What are the key components and how do they interact?

Context: With the proliferation of IoT devices, developers need to respond to diverse and rapidly changing user requirements quickly. Traditional firmware updates require complete system restarts, causing significant downtime.

Solution:

The dynamic IoT application architecture achieves faster updates through a two-part design that separates stable core functionality from dynamic application logic:

Key Components:

1. **Core Application (Elixir):** Handles system interfaces, hardware access, and stable functionality
2. **Wasmtube Bridge:** Custom library that enables communication between Elixir and Wasm modules
3. **Wasm Runtime:** Executes the dynamic application logic in an isolated sandbox
4. **Dynamic Wasm Binary:** Contains the updatable application behavior

How They Interact:

- The core Elixir application launches Wasmtube, which initializes the Wasm runtime
- Wasmtube creates a sandboxed environment with dedicated linear memory for the Wasm binary
- When updates are needed, developers deploy a new Wasm binary to the file system
- Linux inotify API notifies Wasmtube of the file change

- Wasmtube discards the old sandbox and initializes a new one with the updated binary
- The core application continues running without interruption

Speed Improvement:

- Traditional firmware updates: 10+ seconds (full OS reboot required)
- Wasm-based updates: 1.4 seconds (only Wasm runtime restart)
- **Result:** 7x faster update times

This architecture enables rapid response to changing user requirements while maintaining system stability through the separation of concerns.

Question 2: Performance Analysis

Question: The study reports a 200-microsecond overhead for Wasm function calls compared to native Elixir execution. Analyze the significance of this overhead and explain why it's considered acceptable for the proposed use cases.

Context: A quantitative evaluation was performed to confirm the effectiveness of the proposed method. The overhead measurement is crucial for understanding the practical viability of WebAssembly in IoT systems.

Solution:

Overhead Analysis:

- Native Elixir execution: 31.01 μs average
- Wasm function call: 252.48 μs average
- **Overhead:** $\sim$221 μs (approximately 200 μs as reported)

Why This Overhead is Acceptable:

1. **Constant Overhead:** The 200μs overhead is nearly constant regardless of the complexity of the Wasm function. This means:
   - For simple operations: overhead dominates (221μs vs 31μs = 7x slower)
   - For complex operations: overhead becomes negligible (221μs vs 3000ms = 0.007%)
2. **Real-World Context:** The study demonstrates this with machine learning inference:
   - MobileNetV2 inference: 588.69ms on Raspberry Pi 4
   - Overhead percentage: 221μs / 588,690μs = 0.037%
   - **Negligible impact** on total processing time
3. **IoT Application Characteristics:** IoT devices typically perform:
   - Sensor data collection (seconds)
   - Data processing (milliseconds to seconds)
   - Network communication (milliseconds)

- The 200μs overhead is insignificant compared to these operations

4. **Trade-off Justification:** The overhead enables:
    - Dynamic updates without system downtime
    - Cross-platform code reuse
    - Improved development efficiency
    - Better maintainability

Conclusion: The 200μs overhead is acceptable because it enables significant architectural benefits while being negligible for the complex, time-consuming operations typical in IoT applications, especially machine learning inference tasks.

Question 3: Isomorphic IoT Systems

Question: How does the isomorphic IoT system concept address the challenges of multi-platform IoT development?

Context: The study proposes isomorphic IoT systems that utilize the same Wasm binary across different layers (device, edge, cloud) of IoT infrastructure, enabling a common codebase for improved development efficiency and maintainability.

Solution: The isomorphic IoT system concept addresses multi-platform challenges by:

1. **Code Reuse:** Same Wasm binary runs on different architectures (ARM, x86/64, arm64)
2. **Reduced Complexity:** Developers maintain one codebase instead of multiple platform-specific versions
3. **Consistent Behavior:** Identical logic across device, edge, and cloud layers
4. **Simplified Deployment:** Single binary deployment process for all platforms
5. **Easier Maintenance:** Bug fixes and updates applied once, deployed everywhere

Question 4: Security Implications

Question: What are the security implications of using WebAssembly in IoT systems?

Context: WebAssembly provides a sandboxed execution environment with linear memory model, but IoT systems have unique security requirements that must be carefully considered.

Solution: Security implications include:

Benefits:

- Sandboxed execution environment
- Memory safety through linear memory model
- Code integrity verification
- Isolated execution prevents system compromise

Challenges:

- Core application still handles sensitive system operations
- Need secure channels for Wasm binary distribution
- Runtime security depends on Wasm runtime implementation
- Potential for malicious Wasm binaries if not properly verified

Question 5: Technology Comparison

Question: The study proposes WebAssembly as a solution for platform-agnostic IoT development. Compare WebAssembly with other platform-agnostic technologies like Docker and eBPF. Why did the researchers choose WebAssembly over these alternatives?

Context: The research addresses the need for platform-agnostic solutions in IoT development, where the same codebase must run across diverse hardware architectures and computing environments (device, edge, cloud).

Solution:

While Docker and eBPF are indeed platform-agnostic technologies, they have significant limitations for IoT applications that make WebAssembly a superior choice for the specific requirements addressed in this research.

Comparison Analysis:

Feature	WebAssembly	Docker	eBPF
Memory Overhead	∼1-5MB	50-200MB	∼512KB
Update Time	1.4s	1-5s	N/A
Platform Support	Universal	Linux/Windows	Linux only
Security Model	Sandboxed	Containerized	Kernel space
Programming Support	Multiple languages	Any language	C-like only
ML Model Support	Full support	Full support	Limited
IoT Suitability	Excellent	Poor	Limited

Why WebAssembly Over Docker:
Docker Limitations:

- **Resource Overhead:** Docker containers require full OS virtualization with 50-200MB RAM overhead, unsuitable for resource-constrained IoT devices
- **Security Concerns:** Docker daemon runs with elevated privileges, creating larger attack surface
- **Update Mechanism:** Still requires container restart, doesn't solve the partial update problem
- **Embedded Constraints:** Not suitable for embedded/constrained environments

Why WebAssembly Over eBPF:
eBPF Limitations:

- **Kernel Dependency:** eBPF programs run in kernel space, requiring specific kernel support and versions
- **Programming Constraints:** Limited to C-like programming with restricted instruction set
- **Platform Support:** Primarily Linux-focused with limited support on embedded systems
- **Application Logic:** Cannot run complex ML models or business logic due to memory limitations (4KB stack, 512KB program size)

WebAssembly Advantages for IoT:
1. Lightweight Runtime:

- Minimal overhead: $\sim 200\mu$s function call overhead

- Small runtime footprint: ∼1-5MB
- No OS virtualization required

2. True Platform Agnosticism:

- Runs on ARM, x86/64, RISC-V, etc.
- Same binary works across device, edge, cloud layers
- No kernel dependencies

3. Dynamic Updates:

- Can replace Wasm binary without system restart
- Only restart Wasm runtime (1.4s vs 10s+ for full reboot)
- Enables true "hot swapping" of application logic

4. Rich Programming Support:

- Supports multiple languages (Rust, C++, AssemblyScript)
- Can run complex ML models (ResNet-50, MobileNetV2)
- Full application logic, not just kernel operations

5. Security Model:

- Sandboxed execution environment
- Linear memory model prevents buffer overflows
- No direct system access (controlled by host)

Conclusion:

The researchers chose WebAssembly because it uniquely provides all the requirements for modern IoT systems:

- Partial updates without system restart
- Cross-platform deployment (ARM device $\rightarrow$ x86 cloud $\rightarrow$ ARM edge)
- ML model execution (complex algorithms)
- Minimal resource usage (embedded constraints)
- Security isolation (untrusted code execution)

WebAssembly is the only solution that provides true platform agnosticism, lightweight execution, dynamic updates, and rich programming support - all essential for the IoT use cases described in the research. While Docker and eBPF have their strengths, they are more limited in their applicability to the specific IoT requirements addressed in this study.