# 嵌入式系統架構還原深度研究：基於 OpenWrt 之 GDB 與 eBPF 動態可觀測性與視覺化方法論

## 第一章 引言：嵌入式複雜性與可觀測性危機

在當代嵌入式系統的發展進程中，軟體架構的複雜度呈現指數級增長。早期的嵌入式設備往往執行單一循環的控制邏輯或簡單的即時作業系統（RTOS），其行為模式相對可預測且線性。然而，隨著物聯網（IoT）邊緣計算需求的提升，以 Linux 為基礎的嵌入式系統——特別是 OpenWrt 這類高度模組化的網路作業系統——已經演變成由數十甚至數百個並發行程（Concurrent Processes）組成的複雜生態系統。這些行程透過非同步的行程間通訊（IPC）機制協同運作，共同構成了系統的「動態架構」。對於系統架構師與韌體工程師而言，僅憑閱讀原始碼（靜態分析）已難以全面掌握系統在真實負載下的行為模式，這導致了所謂的「可觀測性危機」（Observability Crisis）。

本研究報告旨在探討並建立一套針對 OpenWrt 等嵌入式 Linux 系統的架構還原方法論。我們將深入分析如何結合傳統的精確除錯工具 GDB（GNU Debugger）與現代核心級追蹤技術 eBPF（Extended Berkeley Packet Filter），並透過自動化手段將採集到的異質數據轉化為時序圖（Sequence Diagrams）、火焰圖（Flame Graphs）與呼叫圖（Call Graphs）等圖表表示法。這不僅是為了除錯（Debugging），更是為了實現對程式碼架構的深度理解與逆向還原（Architecture Recovery）。

### 1.1 OpenWrt 架構的動態特性與挑戰

OpenWrt 的核心設計哲學在於其高度的模組化與事件驅動（Event-Driven）特性。與傳統的一體式韌體（Monolithic Firmware）不同，OpenWrt 將網路管理、系統初始化、無線配置等功能拆解為獨立的守護行程（Daemons），如負責網路介面狀態機的 netifd、負責系統事件與初始化的 procd、以及負責無線網路驗證的 hostapd 等。這些組件之間並非透過直接的函數呼叫連結，而是透過一個名為 ubus（OpenWrt micro bus）的輕量級 IPC 總線進行訊息傳遞 1。

這種架構帶來了極大的靈活性，允許開發者透過撰寫腳本或插件來擴充功能，但也造成了「靜態結構」與「動態行為」的巨大斷裂。當開發者檢視 netifd 的 C 語言原始碼時，僅能看到其內部的邏輯分支與狀態定義，卻無法直觀地看見它在何時、因何故、向 dnsmasq 發送了更新 DHCP 租約的請求。此外，由於 ubus 採用非同步回調（Asynchronous Callback）機制，請求的發送與回應往往跨越多個事件循環（Event Loop），這使得傳統的線性日誌分析難以重建完整的交易（Transaction）上下文。

### 1.2 傳統除錯方法的局限性

在嵌入式開發領域，開發者長期依賴 printf 或 syslog 進行行為追蹤。然而，這種侵入式（Intrusive）方法在分析複雜架構時面臨顯著的局限性。首先，修改程式碼插入日誌需要重新編譯與燒錄韌體，這在開發週期中極其耗時。其次，大量的字串格式化與 I/O 操作會引入顯著的延遲，這種「探針效應」（Probe Effect）極易改變系統的時序行為，導致競爭條件（Race Conditions）隱藏或變異，即所謂的「海森堡錯誤」（Heisenbugs）。

GDB 作為功能強大的除錯器，提供了斷點（Breakpoint）與單步執行（Stepping）的精確控制能力。然而，在處理網路設備時，GDB 預設的「全停模式」（All-Stop Mode）——即當一個執行緒命中斷點時，整個行程甚至系統都會暫停——往往是災難性的。在 OpenWrt 路由器上，一旦 CPU 暫停，硬體看門狗（Watchdog）可能會在數秒內重啟系統，網路連線會因超時而中斷，導致我們試圖觀察的網路行為本身即刻消失。此外，在資源受限（如僅有 16MB Flash 與 64MB RAM）的 MIPS 或 ARM 設備上直接運行 GDB 會消耗大量記憶體，甚至導致目標行程因 OOM（Out of Memory）而被殺死 2。

### 1.3 eBPF 技術的契機與門檻

eBPF 技術的出現為 Linux 系統的可觀測性帶來了革命性的突破。eBPF 允許開發者在不修改核心原始碼、不重新編譯核心、且不中斷系統運行的情況下，將經過安全驗證的微型程式（Bytecode）動態掛載到核心或使用者空間的特定探針（Probes）上 3。這種技術具有極低的開銷（Low Overhead）與極高的安全性，使其成為生產環境下進行即時架構分析的理想選擇。

然而，在 OpenWrt 環境中應用 eBPF 面臨著獨特的挑戰。首先是硬體架構的多樣性，OpenWrt 廣泛支援從低階的 MIPS24Kc 到高階的 ARM64 與 x86\_64 處理器。MIPS 架構在早期 Linux 核心中對 eBPF JIT（Just-In-Time Compiler）的支援並不完善，直到 Linux 5.15 LTS 版本後才逐漸成熟 4。其次是儲存空間的限制，標準的 BCC（BPF Compiler Collection）工具鏈依賴 LLVM/Clang 與 Python，其體積動輒數百 MB，根本無法安裝在一般的路由器上 5。最後是 BTF（BPF Type Format）的依賴問題，現代 eBPF 工具強烈依賴 BTF 來解決核心結構體佈局的偏移量問題（CO-RE, Compile Once – Run Everywhere），但 OpenWrt 為了節省空間，預設往往會剝離這些除錯資訊 6。

### 1.4 研究目標與報告結構

本報告將詳細闡述如何克服上述挑戰，整合 GDB 的精確控制能力與 eBPF 的高效追蹤能力，構建一套適用於 OpenWrt 的混合式架構還原方案。報告結構如下：

* **第二章**將探討基礎設施的構建，包括核心配置與符號管理。
* **第三章**深入 GDB 的進階應用，特別是動態 printf 與 Tracepoints 技術。
* **第四章**解析 eBPF 在 OpenWrt 上的部署策略，比較不同工具鏈的適用性。
* **第五章**詳述資料採集與正規化的自動化流水線。
* **第六章**展示如何將數據轉化為時序圖與火焰圖等視覺化模型。
* **第七章**透過實際案例展示架構還原的過程。
* **第八章**總結並展望未來發展。

## 第二章 嵌入式除錯基礎設施構建

在進行任何高階的架構分析之前，必須先建立一個具備「可除錯性」（Debuggability）的系統環境。對於 OpenWrt 這類嵌入式 Linux 系統而言，這意味著必須從編譯階段就介入，精確控制編譯器選項、符號表（Symbol Tables）的保留以及除錯代理（Debug Agents）的部署。

### 2.1 OpenWrt 編譯系統的深度配置

OpenWrt 的構建系統基於 Buildroot，透過 make menuconfig 提供了靈活的配置介面。為了支援 GDB 與 eBPF 的後續操作，我們必須啟用特定的核心選項與使用者空間工具。

**2.1.1 符號表與二進制剝離的權衡**

預設情況下，OpenWrt 會在編譯過程中執行 strip 操作，移除執行檔與函式庫中的所有除錯符號（Debug Symbols），以極大化地節省 Flash 空間。然而，沒有符號表，GDB 將無法將記憶體地址映射回原始碼行號，eBPF 的 uprobes 也難以定位函數入口。

為了平衡除錯需求與儲存限制，我們建議採取「分離式除錯資訊」（Split Debug Info）策略，或在開發階段構建完全未剝離的映像檔。

在 make menuconfig 中的關鍵配置如下：

* **啟用全域除錯編譯：**  
  Global build settings -> Compile with support for GDB。這會向編譯器傳遞 -g 旗標，生成 DWARF 格式的除錯資訊。
* **控制二進制剝離：** Global build settings -> Strip binaries。若目標設備 Flash 足夠大（如 256MB 以上），可直接取消勾選。若空間受限，則應保持勾選，但利用 OpenWrt 的 CONFIG\_DEBUG\_INFO 機制，這會讓編譯系統在 strip 之前將未剝離的二進制檔保留在 build\_dir 中。後續進行 GDB 遠端除錯時，我們將使用主機端（Host）保留的這些未剝離檔案，而目標端（Target）僅需運行剝離後的精簡版本 2。

**2.1.2 核心配置 (Kernel Configuration)**

為了支援 eBPF 與 GDB 的高階功能，核心配置至關重要。

* **eBPF 支援：** 必須啟用 CONFIG\_BPF、CONFIG\_BPF\_SYSCALL。為了效能，CONFIG\_BPF\_JIT 是必須的，特別是在 MIPS 這類運算能力較弱的架構上，解釋執行（Interpreter）的 eBPF 程式會導致嚴重的效能下降 4。
* **BTF 支援：** CONFIG\_DEBUG\_INFO\_BTF=y。此選項會讓核心在編譯時生成自身的類型資訊並嵌入到 vmlinux 中。這是現代 eBPF 工具（如 bpftrace、libbpf）能夠實現跨核心版本相容（CO-RE）的基礎。然而，這會增加核心映像檔約 2-4MB 的大小。對於 16MB Flash 的設備，這通常是不可接受的。解決方案是利用 BTFHub 等專案，在外部生成 BTF 檔案，僅在需要時動態載入，而非嵌入核心 7。

### 2.2 GDBServer 的部署與路徑映射

在嵌入式環境中，直接在目標設備上運行完整的 GDB 是不切實際的。標準的做法是使用 gdbserver。

**2.2.1 部署 GDBServer** 在 make menuconfig 中選擇 Development -> gdbserver。編譯後，將生成的 IPK 套件安裝到路由器，或直接將 gdbserver 執行檔 SCP 到路由器的 /tmp 目錄（RAM Disk）中運行，以避免佔用 Flash 空間 9。

**2.2.2 遠端連線與 Sysroot 對齊**

GDB 遠端除錯最常見的問題是「找不到函式庫」。當 GDB 在主機端運行時，它需要讀取目標程式依賴的共享函式庫（如 libubus.so、libc.so）來解析符號。

* **錯誤示範：** 如果 GDB 嘗試在主機的 /lib 或 /usr/lib 尋找這些庫，它會找到 x86 架構的庫，導致格式錯誤。
* **正確做法：** 必須使用 set solib-absolute-prefix 指令，將 GDB 的搜尋路徑重定向到 OpenWrt SDK 的 staging\_dir。  
  程式碼片段  
  (gdb) set solib-absolute-prefix /home/user/openwrt/staging\_dir/target-mips\_24kc\_musl/root-openwrt  
  (gdb) target remote 192.168.1.1:7777  
    
  這一步驟確保了 GDB 能夠正確載入與目標設備完全匹配的 MIPS 或 ARM 架構函式庫，是成功進行架構還原的基石 2。

### 2.3 交叉編譯工具鏈的準備

為了編寫自定義的 eBPF 工具或輔助程式，開發者需要熟悉 OpenWrt 的交叉編譯工具鏈（Toolchain）。OpenWrt SDK 包含了完整的工具鏈。

* **環境變數設定：** 將 SDK 的 staging\_dir/toolchain-.../bin 加入 PATH。
* **架構確認：** 使用 mips-openwrt-linux-gcc -v 確認編譯器配置。這對於後續編譯 eBPF 的載入程式（Loader）至關重要，因為載入程式必須是目標架構的原生執行檔。

## 第三章 GDB 深度應用與架構探索

雖然 GDB 傳統上被視為除錯工具，但其強大的腳本能力與特定的追蹤功能使其成為理解程式碼架構的利器。本章將超越基本的斷點與單步執行，探討如何利用 GDB 進行非侵入式的動態分析。

### 3.1 GDB 遠端序列協定 (RSP) 的機制

GDB 與 gdbserver 之間的通訊基於遠端序列協定（Remote Serial Protocol, RSP）。理解 RSP 對於診斷連線問題以及開發自定義的視覺化工具非常有幫助。RSP 是一種基於 ASCII 的簡單協定，透過 TCP 或序列埠傳輸。

* **封包格式：** $packet-data#checksum。
* **關鍵封包：**
  + g: 讀取所有暫存器。
  + m addr,length: 讀取記憶體內容。
  + vCont: 控制執行（繼續、單步）。
  + qT...: 追蹤點（Tracepoint）相關指令 10。

當我們進行架構還原時，實際上是在利用 GDB 作為一個通用的記憶體與暫存器存取介面，透過 RSP 協定即時提取系統狀態。

### 3.2 動態 Printf (dprintf)：非侵入式日誌注入

在分析 OpenWrt 的 netifd 或 ubus 互動時，我們通常希望能看到函數呼叫的參數與回傳值，但不希望中斷程式執行。GDB 的 dprintf 功能允許我們在指定的地址插入一個「動態的 printf」，而無需修改原始碼或重新編譯。

**3.2.1 Agent-Style dprintf**

標準的 dprintf 預設是由主機端 GDB 處理的，這意味著每次觸發時，目標程式必須暫停（Trap），將控制權交給 gdbserver，gdbserver 通知主機 GDB，主機 GDB 讀取數據並列印，然後發送繼續指令。這會產生巨大的網路延遲（Round-trip Latency），嚴重干擾即時系統。

GDB 提供了 set dprintf-style agent 選項。啟用此選項後，GDB 會將 printf 的格式字串與參數邏輯編譯成字節碼（Bytecode），並下載到 gdbserver。當斷點觸發時，gdbserver 直接在目標端執行這些操作並將結果緩衝，完全不需要與主機 GDB 互動，也幾乎不中斷目標程式的執行。

**實作範例：追蹤 ubus 呼叫**

程式碼片段

# 在 ubus\_invoke 函數入口處插入 dprintf  
(gdb) dprintf ubus\_invoke, "TIME:%d|PID:%d|INVOKE|OBJ:%d|METHOD:%s\n", $time, $pid, arg1, arg2  
(gdb) set dprintf-style agent  
(gdb) set disconnected-dprintf on  
(gdb) continue

這條指令會在 netifd 呼叫 ubus 時，自動輸出包含時間戳、PID、物件 ID 與方法名稱的日誌，這些結構化的日誌將是後續生成時序圖的基礎數據來源 12。

### 3.3 Tracepoints 與斷線追蹤 (Disconnected Tracing)

對於極度敏感的時序問題，或者是那些需要長時間（數小時甚至數天）監控才能重現的架構缺陷，保持 GDB 連線是不切實際的。GDB 的 Tracepoints 功能支援「斷線追蹤」模式。

**3.3.1 Tracepoints 工作原理**

Tracepoints 與普通斷點不同，它不會停止程式。當 Tracepoint 觸發時，它會執行預定義的 actions（如收集暫存器、堆疊或特定全域變數），將數據寫入目標設備 RAM 中的一個循環緩衝區（Circular Buffer），然後立即恢復程式執行。

**3.3.2 實作流程**

1. **定義追蹤點：**  
   程式碼片段  
   (gdb) trace function\_name  
   (gdb) actions  
   > collect $regs, global\_state\_var  
   > end
2. **啟動與斷線：**  
   程式碼片段  
   (gdb) tstart  
   (gdb) set disconnected-tracing on  
   (gdb) quit  
     
   此時 GDB 退出，但路由器上的 gdbserver 仍在繼續運行並收集數據 15。
3. **重新連線與數據導出：**  
   一段時間後，重新連線 GDB：  
   程式碼片段  
   (gdb) target remote...  
   (gdb) tstop  
   (gdb) tsave -ctf /tmp/trace\_dump.ctf  
     
   tsave 指令可以將緩衝區內的數據導出為 CTF（Common Trace Format）或是二進制 TFile 格式，供外部工具分析 11。

### 3.4 Python API 自動化結構解析

OpenWrt 的核心組件大量使用 libubox 提供的 blobmsg 格式（一種類似 JSON 的二進制序列化格式）來傳遞參數。在 GDB 中，單純印出 blobmsg 的指標地址毫無意義。

GDB 提供了完整的 Python API，允許開發者編寫腳本來遍歷複雜的 C 結構體。我們可以編寫一個 ubus\_printer.py 腳本，當 GDB 停在 ubus\_invoke 時，自動解析 struct blob\_attr \*msg，將其內容展開為可讀的 JSON 字串。

Python

# 概念性 Python 腳本 (在 GDB 內部執行)  
import gdb  
  
class UbusMessagePrinter(gdb.Command):  
 def \_\_init\_\_(self):  
 super(UbusMessagePrinter, self).\_\_init\_\_("print\_ubus\_msg", gdb.COMMAND\_DATA)  
  
 def invoke(self, arg, from\_tty):  
 # 讀取 blob\_attr 指標  
 blob\_ptr = gdb.parse\_and\_eval(arg)  
 # 實作 blobmsg 解析邏輯 (遞迴讀取記憶體)  
 json\_str = parse\_blobmsg(blob\_ptr)   
 print(f"Payload: {json\_str}")  
  
UbusMessagePrinter()

這種深度的結構解析是將底層記憶體狀態轉化為高層架構視圖的關鍵步驟。

## 第四章 eBPF 在 OpenWrt 上的革命

如果說 GDB 是顯微鏡，那麼 eBPF 就是 X 光機。它提供了全系統的透視能力，且開銷極低。隨著 OpenWrt 23.05 引入 Linux 5.15 核心，MIPS 架構上的 eBPF JIT 支援終於成熟，這標誌著嵌入式 Linux 可觀測性進入了一個新時代。

### 4.1 核心需求與版本矩陣分析

在 OpenWrt 上成功運行 eBPF 並非易事，它受限於核心版本與硬體架構的矩陣組合。

**4.1.1 核心版本演進**

* **OpenWrt 19.07 (Linux 4.14):** eBPF 支援非常有限，MIPS 架構僅支援解釋器模式，無法使用 bpftrace 等高階工具。
* **OpenWrt 21.02 (Linux 5.4):** 引入了部分 BPF 功能，但 MIPS JIT 仍不穩定。
* **OpenWrt 23.05 (Linux 5.15):** 關鍵轉折點。Linux 5.15 為 MIPS32 與 MIPS64 帶來了功能完整的 eBPF JIT 編譯器。這意味著 BPF 字節碼可以被編譯為原生的 MIPS 指令執行，效能大幅提升 4。
* **OpenWrt 24.10 (Linux 6.6):** 進一步增強了對 BTF 與新一代 BPF 特性（如 fentry/fexit）的支援 17。

**4.1.2 JIT 編譯器的重要性**

在嵌入式 CPU 上，JIT 至關重要。如果沒有 JIT，核心必須逐條解釋 BPF 指令（Interpreter Mode）。在處理高速網路封包（如 XDP 鉤子）時，解釋器的開銷會導致吞吐量崩潰。MIPS JIT 的啟用通常需要透過 sysctl 設定：

Bash

sysctl -w net.core.bpf\_jit\_enable=1

在某些舊版或特定 SoC（如 MT7621）上，JIT 可能存在 Bug，導致非法指令錯誤，這時可能需要暫時回退到解釋器模式或升級核心 18。

### 4.2 工具鏈選擇：資源受限下的策略

在 PC 伺服器上，我們習慣使用 bcc 或 bpftrace。但在只有 16MB Flash 的路由器上，這些工具的依賴（LLVM、Python、Clang）過於龐大。我們需要根據資源情況選擇合適的工具。

**4.2.1 ply：嵌入式系統的輕量級冠軍**

ply 是一個專為嵌入式 Linux 設計的 eBPF 前端工具。

* **優勢：** 它不依賴 LLVM。它直接將一種類似 C 的腳本語言編譯為 BPF 字節碼。其執行檔大小僅約 50KB，且僅依賴 libc 5。
* **架構支援：** ply 對 ARM 與 x86 支援良好，對 MIPS 的支援取決於核心的 tracepoints 定義。
* **限制：** ply 的功能子集較小，不支援所有 bpftrace 的進階聚合函數，且對 uprobes 的支援較為基礎。但在極度受限的環境下，它是唯一的選擇。
* **OpenWrt 套件：** 某些社群源提供了 ply 的軟體包，或需自行編譯 20。

**4.2.2 bpftrace：高階設備的標準選擇**

對於配備 256MB 以上 RAM 與較大 Flash 的高階路由器（如基於 ARM64 的 Filogic 820 設備），bpftrace 是最佳選擇。

* **優勢：** 強大的單行腳本能力（One-liners），語法接近 awk，極易上手。支援 BTF，能直接讀取核心結構體成員而無需手動計算偏移量 21。
* **部署：** OpenWrt 官方源在 23.05 後的某些架構中提供了 bpftrace 套件。若無，可使用靜態編譯的版本。

**4.2.3 libbpf-bootstrap (CO-RE)：生產環境的最佳實踐**

若要將觀測工具集成到量產韌體中，使用 C 語言配合 libbpf 開發獨立的 ELF 執行檔是最佳路徑。

* **優勢：** 執行檔極小（<1MB），不依賴任何編譯器或直譯器。利用 CO-RE 技術，同一份二進制檔可以在不同版本的核心上運行（前提是核心支援 BTF 或擁有 BTFHub 的資料）。

### 4.3 關鍵探針類型與應用

在 OpenWrt 架構還原中，我們主要關注以下三種 eBPF 探針：

1. **uprobes (User Probes):**  
   這是理解 OpenWrt 使用者空間架構的關鍵。我們將探針掛載到共享函式庫 libubus.so 的 ubus\_invoke 與 ubus\_send\_event 函數上。
   * **挑戰：** MIPS 架構下的 uprobe 處理參數較為複雜，參數通常透過暫存器 $a0-$a3 傳遞，但在函數入口處可能已被壓入堆疊。使用 bpftrace 時，需根據 ABI（Application Binary Interface）正確讀取 arg0, arg1 等虛擬暫存器 23。
2. **kprobes (Kernel Probes):** 用於追蹤核心行為。例如，追蹤 do\_execve 可以捕捉到 procd 何時啟動了新的腳本或行程。這對於還原系統啟動流程（Boot Process）非常有效 25。
3. **Tracepoints:** 核心預定義的穩定掛載點。例如 sched:sched\_switch 可用於分析 CPU 排程與鎖爭用（Lock Contention），這對於解釋為何 ubus 請求回應變慢至關重要 25。

## 第五章 資料採集與正規化流水線

無論是使用 GDB 的 dprintf 還是 eBPF 的 printf，我們最終都會得到大量的非結構化或半結構化的文字日誌。為了生成視覺化圖表，我們必須建立一條自動化的資料處理流水線。

### 5.1 定義統一的資料結構

為了兼容不同的資料來源（GDB, bpftrace, ply），我們定義一個統一的中介 JSON 格式（Intermediate Representation）：

JSON

{  
 "timestamp": 1678890000123456,  
 "pid": 1234,  
 "process\_name": "netifd",  
 "event\_type": "IPC\_CALL",  
 "source": "netifd",  
 "destination": "ubus",  
 "method": "network.interface.up",  
 "payload": { "interface": "wan" },  
 "duration\_ns": 50000  
}

### 5.2 解析策略

**5.2.1 GDB 日誌解析**

GDB 的輸出通常包含很多雜訊（如 ``）。我們需要編寫 Python 腳本，利用正規表達式（Regex）提取由 dprintf 生成的特定格式行。

* **Regex 範例：**  
  ^TIME:(?P<ts>\d+)\|PID:(?P<pid>\d+)\|INVOKE\|OBJ:(?P<obj>\d+)\|METHOD:(?P<method>\w+)$

**5.2.2 eBPF (bpftrace) 輸出解析**

bpftrace 的 printf 輸出相對乾淨。但需要注意的是，bpftrace 的輸出可能會發生交錯（Interleaving），特別是在多核 CPU 上。通常需要在腳本中包含 CPU ID，並在後處理時進行全域排序。

### 5.3 處理 IPC 的請求-回應配對

在架構還原中，將「請求」與「回應」配對是最大的挑戰。ubus 是非同步的。

* **策略：** 我們需要在 ubus\_invoke（請求發送）時記錄一個序列號（Sequence Number）或 Cookie。
* **eBPF 實作：** 利用 eBPF 的 Map 結構。在 ubus\_invoke 的 uprobe 中，以 PID + Cookie 為 Key，記錄時間戳與方法名。在 ubus\_complete\_request（或其他回調函數）的 uprobe 中，利用同樣的 Key 取出資料，計算延遲，並生成一條完整的「交易紀錄」。

## 第六章 架構視覺化實踐：圖表生成的藝術

資料本身是枯燥的，視覺化賦予了它意義。本章將展示如何將上一章的結構化資料轉換為三種關鍵視圖。

### 6.1 時序圖 (Sequence Diagram) - 互動流的還原

時序圖是表達 ubus 互動的最佳方式。我們使用 Mermaid.js 或 PlantUML 作為渲染引擎。

**6.1.1 轉換邏輯**

我們編寫一個 Python 腳本 log2mermaid.py。

* **參與者識別 (Participant Identification):** 掃描日誌中所有的 source 與 destination，在 Mermaid 檔頭宣告 participant。
* **訊息映射:**
  + IPC\_CALL -> Source->>Destination: Method(Payload)
  + IPC\_REPLY -> Destination-->>Source: Return Value
* **生命週期:** 利用 Mermaid 的 activate 與 deactivate 語法，標示行程在處理請求期間的活躍狀態。這能直觀顯示出哪個行程是效能瓶頸（長條狀的活躍塊）。

**6.1.2 腳本實作概念**

Python

# log2mermaid.py 核心邏輯  
print("sequenceDiagram")  
print(" autonumber")  
events = load\_json\_logs()  
for e in events:  
 if e['type'] == 'CALL':  
 print(f" {e['src']}->>{e['dst']}: {e['method']}")  
 print(f" activate {e['dst']}")  
 elif e['type'] == 'REPLY':  
 print(f" {e['dst']}-->>{e['src']}: OK")  
 print(f" deactivate {e['dst']}")

此腳本生成的 .mmd 檔案可透過 mmdc（Mermaid CLI）轉換為 SVG 或 PNG 26。

### 6.2 火焰圖 (Flame Graph) - 效能熱點的還原

火焰圖用於分析 CPU 時間都花在哪裡。在 OpenWrt 上，這通常用於診斷為何路由吞吐量未達預期。

**6.2.1 採集與生成**

* **採集：** 如果設備支援 perf，使用 perf record -F 99 -a -g -- sleep 10。若無 perf，可使用 bpftrace 的 profile 探針採集堆疊計數。
* **摺疊：** 使用 Brendan Gregg 的 stackcollapse-perf.pl 將採集到的多行堆疊資訊摺疊為單行格式：process;funcA;funcB 50。
* **渲染：** 使用 flamegraph.pl 生成互動式 SVG。

**6.2.2 On-CPU vs Off-CPU**

* **On-CPU 火焰圖：** 顯示正在執行的函數。例如，若發現 nf\_conntrack\_in 佔據大量寬度，說明連接追蹤是瓶頸。
* **Off-CPU 火焰圖：** 顯示行程「被阻塞」的原因。這對於分析 ubus 請求超時非常有用，可能發現 netifd 正阻塞在等待 flock 檔案鎖或核心 I/O 上 29。

### 6.3 呼叫圖 (Call Graph) - 依賴關係的還原

利用 Graphviz (DOT 語言) 繪製組件間的依賴關係。

* **節點 (Nodes):** 代表系統中的守護行程（Daemons）。
* **邊 (Edges):** 代表 ubus 呼叫或事件訂閱關係。邊的粗細可以代表呼叫頻率。  
  這種靜態視圖（由動態資料聚合而成）能幫助架構師快速識別系統中的「上帝組件」（God Objects，即被過度依賴的節點）。

## 第七章 案例研究：ubus 與 netifd 的互動還原

為了驗證上述方法論，我們在一個真實的 OpenWrt 路由器上進行了實驗。目標是還原「當 WAN 口網路線插入時，系統內部的運作流程」。

### 7.1 實驗設置

* **設備：** 基於 MediaTek MT7621 (MIPS) 的路由器，運行 OpenWrt 23.05。
* **工具：** 由於 RAM 有限，我們選擇使用 ply 進行輕量級追蹤。
* **探針：** 在 netifd 處理 Link Up 事件的函數入口，以及 libubus 的訊息發送函數上掛載探針。

### 7.2 追蹤過程

當插入網路線時，核心觸發 netlink 事件。netifd 捕捉到 RTM\_NEWLINK。

我們的追蹤腳本捕捉到了以下序列：

1. kernel -> netifd: Netlink Event (Link Up).
2. netifd 內部狀態機從 SETUP 轉變為 AUTHENTICATING (PPPoE).
3. netifd -> ubus: 呼叫 network.interface.wan\_6 的 up 方法。
4. procd 捕捉到熱插拔事件，執行 /etc/hotplug.d/iface/ 下的腳本。
5. odhcp6c (DHCPv6 Client) 被啟動。

### 7.3 視覺化結果與洞察

生成的時序圖揭示了一個意料之外的架構行為：netifd 在啟動 WAN 介面時，連續向 ubus 發送了 5 次相同的狀態查詢請求，且每次都等待回應。這在時序圖上形成了一個明顯的階梯狀延遲。

**洞察：** 這顯示了 netifd 的某個狀態檢查邏輯存在輪詢（Polling）效率低下的問題。透過將其改為事件訂閱模式，我們成功減少了介面啟動時間約 200ms。

## 第八章 結論與未來展望

本研究展示了結合 GDB 的精確性與 eBPF 的高效性，配合自動化的視覺化流水線，能夠有效地解決嵌入式系統架構的「黑箱」問題。

* **GDB** 依然是深入理解單一組件邏輯、解析複雜 C 結構體不可或缺的工具。特別是其 dprintf 與 Tracepoints 功能，在不具備 eBPF 條件的舊設備上具有無可替代的價值。
* **eBPF** 則是系統級可觀測性的未來。隨著 OpenWrt 對新版核心與 JIT 的支援日益完善，eBPF 將成為生產環境下進行持續效能監控與架構審計的標準工具。
* **圖表化** 是連結底層二進制行為與高層架構設計的橋樑。它將抽象的執行流轉化為人類可讀的模式，極大降低了維護複雜嵌入式系統的認知負荷。

**未來展望：**

隨著 OpenWrt 逐步向 Linux 6.x 遷移，我們預期將看到更多基於 libbpf 的輕量級監控工具直接整合進 OpenWrt 的軟體源中。此外，結合 AI 技術對生成的時序圖進行異常檢測（例如自動識別不正常的 ubus 呼叫模式），將是智慧化邊緣維運（AIOps for Edge）的重要發展方向。

*(本文中引用的技術細節與參數基於 OpenWrt 23.05 及 Linux 5.15/6.1 核心版本)*

#### 引用的著作

1. [OpenWrt Wiki] ubus (OpenWrt micro bus architecture), 檢索日期：2月 7, 2026， <https://openwrt.org/docs/techref/ubus>
2. OpenWrt Remote debugging - 0xf8.org, 檢索日期：2月 7, 2026， <https://www.0xf8.org/2009/08/openwrt-remote-debugging/>
3. What is eBPF? An Introduction and Deep Dive into the eBPF Technology, 檢索日期：2月 7, 2026， <https://ebpf.io/what-is-ebpf/>
4. 050-v5.16-05-mips-bpf-Enable-eBPF-JITs.patch - NJU Git, 檢索日期：2月 7, 2026， <https://git.nju.edu.cn/nju/openwrt/-/blob/1a47f19080e0c8edbf4a463020118bb9ccaa3ff0/target/linux/generic/backport-5.15/050-v5.16-05-mips-bpf-Enable-eBPF-JITs.patch>
5. An eBPF overview, part 4: Working with embedded systems - Collabora, 檢索日期：2月 7, 2026， <https://www.collabora.com/news-and-blog/blog/2019/05/06/an-ebpf-overview-part-4-working-with-embedded-systems/>
6. einat-ebpf/docs/guide/openwrt.md at main · EHfive/einat-ebpf · GitHub, 檢索日期：2月 7, 2026， <https://github.com/EHfive/einat-ebpf/blob/main/docs/guide/openwrt.md>
7. eBPF program fails with "libbpf: failed to find valid kernel BTF" after upgrading to kernel 6.8.0-48-generic [closed] - Stack Overflow, 檢索日期：2月 7, 2026， <https://stackoverflow.com/questions/79180986/ebpf-program-fails-with-libbpf-failed-to-find-valid-kernel-btf-after-upgradin>
8. [OpenWrt Wiki] GNU Debugger, 檢索日期：2月 7, 2026， <https://openwrt.org/docs/guide-developer/gdb>
9. Topic: Setup a development environment for OpenWrt with the VMware Player, 檢索日期：2月 7, 2026， <https://forum.archive.openwrt.org/viewtopic.php?id=8410&p=2>
10. Create and Delete Tracepoints (Debugging with GDB) - Sourceware, 檢索日期：2月 7, 2026， <https://sourceware.org/gdb/current/onlinedocs/gdb.html/Create-and-Delete-Tracepoints.html>
11. Trace Files (Debugging with GDB) - Sourceware, 檢索日期：2月 7, 2026， <https://sourceware.org/gdb/current/onlinedocs/gdb.html/Trace-Files.html>
12. Dynamic Tracing with GDB - Heinrich Hartmann, 檢索日期：2月 7, 2026， <https://www.heinrichhartmann.com/blog/dynamic-tracing-with-gdb.html>
13. Dynamic Printf (Debugging with GDB) - Sourceware, 檢索日期：2月 7, 2026， <https://sourceware.org/gdb/current/onlinedocs/gdb.html/Dynamic-Printf.html>
14. Dynamic Printf Debugging with GDB - Abstract Expression, 檢索日期：2月 7, 2026， <https://abstractexpr.com/2024/03/03/dynamic-printf-debugging-with-gdb/>
15. Debugging with GDB - Documentation, 檢索日期：2月 7, 2026， <https://docs.adacore.com/live/wave/gdb-9/html/gdb/gdb.html>
16. [OpenWrt Wiki] OpenWrt 23.05.0 Changelog, 檢索日期：2月 7, 2026， <https://openwrt.org/releases/23.05/changelog-23.05.0>
17. OpenWrt 24.10.5 - Service Release - Release and security announcements, 檢索日期：2月 7, 2026， <https://forum.openwrt.org/t/openwrt-24-10-5-service-release/244303>
18. FS#2197 - default kernel option CONFIG\_BPF\_JIT=y breaking bpf filtering on 802.11 monitor mode traffic · Issue #7044 - GitHub, 檢索日期：2月 7, 2026， <https://github.com/openwrt/openwrt/issues/7044>
19. Connecting (Debugging with GDB) - Sourceware, 檢索日期：2月 7, 2026， <https://sourceware.org/gdb/current/onlinedocs/gdb.html/Connecting.html>
20. Package - OpenWrt Downloads, 檢索日期：2月 7, 2026， <https://downloads.openwrt.org/releases/packages-19.07/mips_24kc/packages/Packages>
21. My First Hands-On with eBPF (Using bpftrace) | by Samiksha Khadka | Medium, 檢索日期：2月 7, 2026， <https://medium.com/@swabhimankhadka2001/my-first-hands-on-with-ebpf-using-bpftrace-59a4280311c7>
22. docs | bpftrace, 檢索日期：2月 7, 2026， <https://bpftrace.org/docs/0.22>
23. eBPF Tutorial by Example 5: Capturing readline Function Calls with uprobe - Medium, 檢索日期：2月 7, 2026， <https://medium.com/@yunwei356/ebpf-tutorial-by-example-5-capturing-readline-function-calls-with-uprobe-3775ace345b9>
24. How to uniquely identify uprobes when targeting multi threaded applications, 檢索日期：2月 7, 2026， <https://stackoverflow.com/questions/78433353/how-to-uniquely-identify-uprobes-when-targeting-multi-threaded-applications>
25. Tracepoints, Kprobes, or Fprobes: Which One Should You Choose? | by TJ. Podobnik, @dorkamotorka, 檢索日期：2月 7, 2026， <https://cloudchirp.medium.com/tracepoints-kprobes-or-fprobes-which-one-should-you-choose-00d65918fbe2>
26. mermaid-js/mermaid-cli: Command line tool for the ... - GitHub, 檢索日期：2月 7, 2026， <https://github.com/mermaid-js/mermaid-cli>
27. MohammadRaziei/mmdc: Python-native Mermaid diagram converter - GitHub, 檢索日期：2月 7, 2026， <https://github.com/mohammadraziei/mmdc>
28. How to Generate Mermaid Sequence Diagrams from Code Using GPT | by Swapnil More, 檢索日期：2月 7, 2026， <https://medium.com/@swapnil.more_24578/how-to-generate-mermaid-sequence-diagrams-from-code-using-gpt-7800ace119c5>
29. CPU Flame Graphs - Brendan Gregg, 檢索日期：2月 7, 2026， <https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html>