針對既有 x86-64 架構的 Linux 和 Windows 應用程式的快速 RISC-V 動態二進位轉譯器

An efficient RISC-V dynamic binary translator designed for executing existing x86-64 architecture Linux and Windows applications.

指導教授：涂嘉恆
專題成員：邱繼寬
開發工具：gcc, github, ninja, x11
測試環境：debian linux 13

作品摘要：

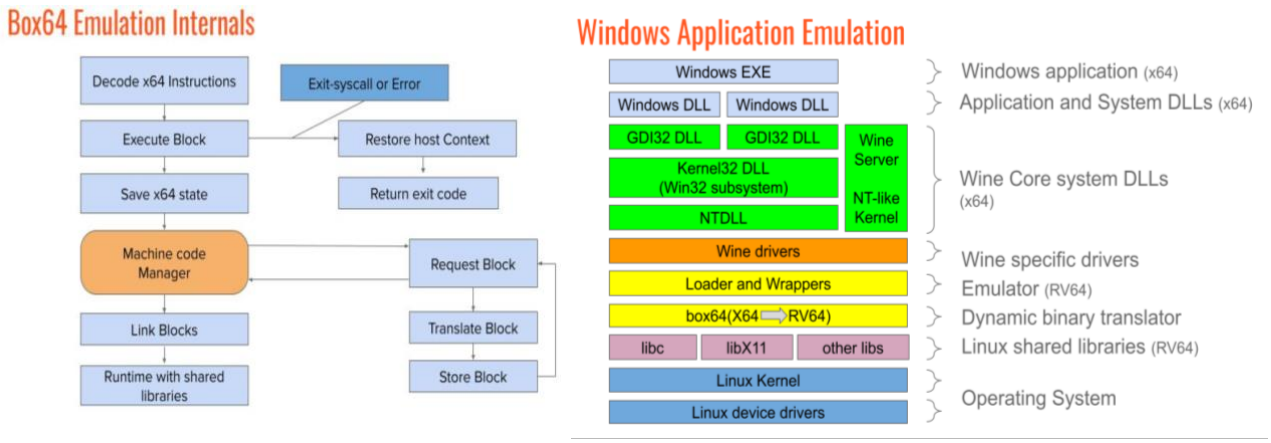
本專案著眼於 RISC-V 處理器架構上模擬既有運行於 x86-64 處理器架構的 Windows 和 Linux 應用程式，且無需修改原始程式碼或重新編譯程式。同時，專案確保動態轉譯效能達到可接受的水準。鑒於相關硬體資源通常有限，本專案特別著重於動態連結函式庫層級的置換，以顯著降低動態編譯器的運算成本。本專案不僅使 x86-64 與 RISC-V 之間的動態二進位轉譯成為可行，還針對模擬器的需求，調整 GNU/Linux 執行環境，以便進一步提升整體模擬過程的可用性和效率。

目前運作於 RISC-V 的軟體以開放原始碼為主，扣除技術社群擁抱 RISC-V 發展的積極因素，其實也說明 RISC-V 生態系統的不足，特別是既有 x86 世界豐富應用程式無法原封不動地運作在 RISC-V，為此，我們提出高速的動態二進位轉譯機制，允許既有 x86 應用程式可達到約原生速度一半的狀況運作在 RISC-V 處理器，支援 Linux 和 Windows 應用程式。這樣的概念類似 Apple 公司當年從 PowerPC 移轉到 Intel，隨後又從 Intel 架構移轉到自行設計的 Arm 晶片而開發的 Rosetta 轉譯器，但我們提出的設計方案完全建構在開放原始碼的軟體架構，利用客製化的動態二進位轉譯器，在 RISC-V 處理器上運作既有的 x86 Linux 和 Windows 應用程式，從而有效延展 RISC-V 的軟體生態。

box64 是本專案選定的模擬器程式碼基礎，藉由 dynarec (dynamic recompiler) 動態二進位轉譯的手法，來加速在 RISC-V 處理器架構模擬 x86-64 程式的過程，box64 的程式碼簡潔且有效，本專案修正 box64 原有動態二進位轉譯和執行環境的缺失，使其在 RISC-V 的效率和可用性獲得顯著提升。執行的最初階段，box64 會從給定的 x86-64 可執行檔中解析出關鍵資訊，例如 ELF 標頭檔描述的程式碼起始地址、程式碼和資料區段的描述，和可執行檔所需要的動態連結函式庫等等。對於每個動態連結函式庫，box64 首先嘗試載入已註冊的原生 (即 RV64 處理器架構) 函式庫，若原生函式庫尚未註冊或無法識別，則 box64 會嘗試找到一個可由自身進行 x86-64 模擬的動態連結函式庫。倘若動態連結函式庫是被 box64 進行動態二進位轉譯而模擬的，則 box64 會將其視作 x86-64 可執行檔，進行 x86-64 指令解碼、動態二進位轉譯，隨後執行。反之，若一個動態連結函式庫不用經由動態二進位轉譯，就意味著 box64 可將其轉包 (wrapping) 到預先描述且存在的原生動態連結函式庫，此舉不僅可大幅降低動態二進位轉譯的成本，也使得在標的 (target) 端環境中，不用完整保存大量 x86-64 動態連結函式庫，只要儲存原生動態連結函式庫沒有涵蓋或 (符號和介面) 不相容的 x86-64 動態連結函式庫，最終提高儲存空間的利用效率，且相較於 QEMU 一類的模擬器，可用更低的主記憶體去模擬相對複雜的 x86-64 程式，值得留意的是，應用程式本身不用作任何修改，直接下載到 RISC-V 環境，佐以本專案調整過的 box64 即可進行模擬。

當 box64 遇到之前尚未執行過的 x86-64 指令時，它不會單純地一次模擬一條指令，而是先構建一個 dyna 區塊 (即用以二進位轉譯的區塊單位)，並將指令存於其中，然後才運行該 dyna 區塊，此舉可避免已模擬過的 x86-64 指令再次被模擬。本專案考慮到 RISC-V 處理器環境的主記憶體容量往往有限，引入額外的記憶體管理機制 (memory pool)，抽換已不再使用的 dyna 區塊，並控管整體可用的記憶體容量，讓 x86-64 程式的模擬過程中，儘量降低模擬器自身的記憶體開銷。不同於 QEMU 一類的模擬器，box64 看待函式地址並非固定數值，而是允許動態連結器 (dynamic linker，而在 ELF 規格的術語則是 ELF interpreter) 針對標的環境的原生動態連結函式庫的存在，予以調整載入 ELF 的符號地址，也就是說，box64 會改寫原生函式的地址，指向模擬器內部的結構和上述的記憶體管理模組，於是建構出模擬器本體和動態編譯器模組皆可識別的「橋樑」，也就是介於模擬環境和原生執行環境之間的連接機制，該手法也是 box64 得以用精簡的程式碼超越 QEMU 一類複雜模擬器的模擬速度的關鍵考量。

當 x86-64 程式即將要模擬時，box64 會載入自身以及所需要的動態連結函式庫，後者又可區分為二種：原生函式庫和模擬函式庫。box64 載入二者的機制不同，使用原生動態連結函式庫的方式是，藉由退出 x86-64 指令集模擬，並切換到 RV64 原生執行環境，藉由 glibc (或 musl libc) 提供的動態連結器，將原生動態連結函式庫載入到主記憶體，且 Linux 會用其虛擬記憶體來管理這些函式庫，為了加速載入動態連結函式庫的速度，本專案調整 box64 程式碼，善用 mmap 系統呼叫以減少模擬 x86-64 程式早期階段的大量 I/O 操作。以下為本專案的 x86-64 對 RV64 的動態二進位轉譯的流程圖：



在由 box64 載入的 ELF 檔案裡頭，只要是動態連結的 ELF 檔案，就存在名為「重新定位」(relocation) 的標誌，允許動態連接器將任何地址填充到需要的符號所在，當需要重新定位到仍在 x86-64 架構中的某些程式碼時，box64 會將 ld.so 一類的 Linux 動態連接器 (由 glibc 或 musl libc 提供) 執行重新定位。不過，當重新定位到原生動態連結函式庫所在地址時，box64

會將該地址指向到自身可識別的特別 x86-64 指令序列，後者其實是例外處理常式的 x86-64 指令碼 0xCC (即 INT3，在 x86 架構中作為開發者除錯和追蹤使用)，隨後是足以讓 box64 識別的簽章符號 (字元 'S' 和 'C')，緊接著存放二個供模擬器運用的指標。一旦 x86-64 程式所有內容成功載入，模擬器的主迴圈將開始運作。

展現模擬器效能和涵蓋程度最佳的案例，莫過於 Microsoft Windows 程式的模擬，為此，我們借助另一個開放原始碼專案 WINE，在 Linux 之上提供 Windows 相容層，允許原有使用 x64 指令集的 Windows 程式在完全不作任何修改的狀況下，藉由 box64 從 x64 動態轉譯為 RV64，並且運用原生的函式庫和服務，例如 X Window System，予以展現 Windows 程式。儘管受限於 WINE 專案的涵蓋，實際上能夠模擬的 Windows 程式有限，但這個途徑說明我們的動態二進位轉譯的確可用。

針對 box64 現有的 JIT 編譯器，我們縮減記憶體開銷、提出 peephole optimization, constant propagation, constant folding 等改進。針對常見的校能評比程式，例如 primes、AES 和 SHA256 等測試項目，由大量的位元及邏輯運算組成，box64 的動態二進位轉譯可將原本的 x86-64 指令改為在 RISC-V 指令，幾乎是一對一轉換，於是儘管 box64 起始和及時編譯器存在一定的執行開銷，效能評比結果顯示僅有約一半的效能折損，而且當長時間執行時，效能的差異則不顯著。從 miniz 、whetstonez 和 dhrystone 測試項目中，可見到目前 box64 對於存在大量分支指令的模擬，依舊有很大的進步空間，在極端狀況下，會有 30 倍的效能折損，這也是本專題著重的議題，可帶來相較 QEMU 有更好的效能表現。