

國立臺灣師範大學資訊工程學系

114 資訊專題研究（一）期末書面報告

運用 **Btrfs** 寫入時複製機制加速 **Rust** 建置快取之研究

**A Study on Accelerating Rust Build Caching with Btrfs Copy-on-Write Mechanism**

指導教授紀博文教授

學生鍾詠傑撰

中華民國 114 年 11 月

## 摘要

隨著 Rust 語言在系統程式設計領域的普及，其嚴格的編譯檢查與靜態分析雖然保證了記憶體安全，但也帶來了顯著的編譯時間成本與磁碟空間消耗。特別是在多分支開發（Multi-branch Development）場景下，開發者往往面臨「切換分支需重新編譯」或「維護多個工作目錄導致磁碟耗盡」的兩難局面。

本研究提出一套名為「Cargo-CoW」的實驗性架構，利用 Btrfs 檔案系統的寫入時複製（Copy-on-Write, CoW）與 Reblink（Reference link）機制，結合 Git Worktree，實現建置環境的秒級複製與還原。實驗結果顯示，在磁碟空間效率方面，本架構結合 Zstd 透明壓縮可節省高達 **76%** 至 **80%** 的物理儲存空間。在建置效能方面，對於無複雜外部依賴的中小型專案（如 ripgrep），冷啟動建置速度提升了 **1.46** 倍。

然而，研究亦發現此技術存在限制：Rust 建置系統（Cargo）對「絕對路徑」的高度敏感性。透過深度剖析 Cargo 的 Fingerprint 機制，我們證實了在大型專案（如 Zed）中，跨目錄還原的快取會因編譯單元（Unit）的指紋雜湊不匹配而失效。本研究總結認為，單純的檔案系統層級最佳化不足以完全解決 Rust 的快取重用問題，未來需結合容器化技術（Namespace Isolation）以固定路徑來規避 Cargo 的雜湊檢查，並建議整合 **Mold** 連結器以解決連結階段的效能瓶頸。

# Contents

摘要	i
<b>1 研究動機與研究問題</b>	<b>1</b>
1.1 研究動機	1
1.2 研究問題	1
<b>2 文獻回顧與探討</b>	<b>1</b>
2.1 Cargo 構建模型與指紋機制 (Fingerprint Mechanism)	1
2.2 依賴追蹤與 dep-info	2
2.3 現有的快取解決方案	2
2.4 Btrfs 與 Reflink 技術	2
2.5 架構演進：從 Docker Btrfs Driver 到 Reflink	3
<b>3 研究方法及步驟</b>	<b>3</b>
3.1 系統架構設計	3
3.2 實驗環境	3
3.3 評估指標	4
<b>4 實驗結果</b>	<b>4</b>
4.1 建置時間效能 (Build Time Performance)	4
4.2 磁碟空間效率 (Disk Space Efficiency)	4
<b>5 分析與討論</b>	<b>4</b>
5.1 增量編譯與全域快取的互斥性矛盾	4
5.2 絕對路徑污染與指紋失效 (Absolute Path Pollution)	5
5.3 完全重建瓶頸：實驗數據分析 (Full Rebuild Bottleneck)	5
5.3.1 Zed 編輯器測試數據	5
5.3.2 失敗原因：Cargo Fingerprint 失效	6
<b>6 結論及未來研究方向</b>	<b>6</b>
6.1 結論	6
6.2 未來研究方向	7
<b>7 參考文獻</b>	<b>7</b>

# 1 研究動機與研究問題

## 1.1 研究動機

Rust 的編譯單元 (Crate) 模型與單態化 (Monomorphization) 特性，使得其編譯產物 (target/ 目錄) 體積龐大。在現代 CI/CD 流程或多人協作開發中，頻繁的分支切換 (Context Switch) 是常態。目前開發者主要採取兩種策略：

1. 單一工作目錄：切換 Git 分支時，Cargo 檢測到原始碼變更，往往觸發大規模重新編譯，浪費時間。
2. 多個工作目錄 (**Git Worktree**)：為每個分支建立獨立目錄，雖然避免了重編，但每個目錄都需獨立的 target/，導致磁碟空間呈倍數增長（數十 GB 至數百 GB）。

## 1.2 研究問題

本研究試圖回答以下核心問題：

1. 儲存效率：能否利用現代檔案系統 (Btrfs/XFS) 的 Reflink 技術，在不佔用額外實體空間的前提下，實現 target/ 目錄的瞬間複製？
2. 快取有效性：透過 Reflink 複製的建置快取 (Build Cache)，能否被 Cargo 的指紋機制有效識別並重用，從而加速新分支的冷啟動 (Cold Start)？
3. 適用邊界：此機制在不同規模（純 Rust vs. 混合 C++）與不同依賴結構的專案中，其效能表現與失效原因為何？

# 2 文獻回顧與探討

## 2.1 Cargo 構建模型與指紋機制 (Fingerprint Mechanism)

Cargo 的構建過程並非單純的檔案編譯，而是基於單元圖譜 (**Unit Graph**) 的複雜調度。

- 編譯單元 (**Unit**)：Cargo 將每個 Package 的不同構建目標 (Lib, Bin, Test, Doc) 抽象為獨立的 Unit。
- 指紋結構 (**Fingerprint Structure**)：為了決定一個 Unit 是否需要重編，Cargo 會計算並比對 Fingerprint。這不僅僅是原始碼的 Hash，更是一個多維度的狀態雜湊，包含：
  - 元數據 (**Metadata**)：編譯器版本 (rustc -vV)、目標架構 (Target Triple)、Profile 配置 (Debug/Release)。
  - 環境變數 (**Env Vars**)：由 build.rs 宣告的 rerun-if-env-changed 變數。
  - 絕對路徑 (**Absolute Paths**)：當前工作目錄 (CWD) 以及依賴項的路徑。
  - 依賴指紋 (**Dependency Fingerprints**)：上游依賴的 Fingerprint 變化會連鎖觸發下游重編。

Cargo 採用 **DirtyReason** 機制來診斷變更。若計算出的指紋與磁碟上儲存的指紋不符，Cargo 會標記該 Unit 為 Dirty 並觸發 JobQueue 進行重編。常見的 DirtyReason 包括：

- **FreshBuild**：首次建置，指紋檔案不存在。
- **FeaturesChanged**：Cargo Features 設定發生變化。
- **TargetConfigurationChanged**：目標平台或 rustflags 改變。
- **PathToSourceChanged**：原始碼檔案的 mtime 或內容被偵測到變更。
- **UnitDependencyInfoChanged**：上游依賴的指紋發生變化，觸發級聯重編譯。

## 2.2 依賴追蹤與 dep-info

Rust 編譯器 (rustc) 在編譯過程中會生成 .d (dependency info) 檔案。這些檔案詳細列出了該 Crate 編譯時讀取的所有檔案路徑。關鍵在於，這些路徑通常以絕對路徑形式儲存。Cargo 在增量編譯檢查時，會解析這些 .d 檔案，若其中的絕對路徑在新的環境中不存在或屬性改變，將導致快取立即失效。

Cargo 的 `parse_dep_info` 模組會執行以下關鍵操作：

1. 路徑規範化：將絕對路徑轉換為相對於專案根目錄的相對路徑，提高快取的可移植性。
2. 環境變數提取：rustc 會將編譯過程中使用的環境變數 (如 `env!("CARGO_PKG_VERSION")`) 以特殊註釋 `# env-var:KEY=VALUE` 的形式寫入 .d 檔案。Cargo 解析這些註釋，並將其加入 Fingerprint 的 local 部分。這樣，如果該環境變數在下次建置時發生變化，Cargo 也能檢測到並觸發重編譯。
3. **mtime** 比對：Cargo 比對 dep-info 中列出的檔案 mtime 與 `invoked.timestamp` (建置開始時間)。若 `source_file.mtime > invoked.timestamp`，則視為 Dirty，即使檔案內容未變也會觸發重建。

## 2.3 現有的快取解決方案

- **sccache (Mozilla)**：透過包裝 rustc 編譯器，將編譯產物快取至本機或雲端。其限制在於：
  - 無法快取連結階段 (**Linking**)：sccache 僅快取 codegen 單元，最終二進位檔需要重新連結。
  - 必須關閉增量編譯：sccache 的粗粒度快取 (以整個 crate 為單位) 與 rustc 的細粒度 CGU (Code Generation Unit) 增量系統不相容。啟用增量編譯時，rustc 依賴本地 incremental/ 目錄中的細粒度狀態，這些狀態無法在跨機器或跨目錄間共享，導致快取命中率極低或產生錯誤結果。
  - 跨機器一致性問題：增量編譯的狀態包含絕對路徑與機器特定的 metadata，使得雲端快取無法保證正確性。
- **Docker Layer Caching**：利用 OverlayFS 分層儲存。雖然能重用層，但層的唯讀特性使其不適合頻繁寫入的增量編譯環境，且 Docker image 的建置過程本身也存在 I/O 開銷。

## 2.4 Btrfs 與 Reflink 技術

Btrfs 的 `FICLONE` ioctl 允許建立檔案的「淺層複製」(Reflink)。兩個檔案共享硬碟上的同一個物理區塊 (Extent)，直到其中一方被修改 (CoW)。這使得複製 GB 級的 target 目錄僅需修改 Metadata，耗時為毫秒級，且初始不佔用額外物理空間。

## 2.5 架構演進：從 Docker Btrfs Driver 到 Reflink

本研究初期受到 Docker Btrfs 儲存驅動程式的啟發，試圖模仿其「分層儲存 (Layered Storage)」模型：

- **Docker** 模型：將唯讀的 Image Layers 對應為 Btrfs Subvolumes，將容器的可寫層對應為 Snapshot。
- 初期構想：為每個 Rust Crate 或依賴樹建立獨立的 Subvolume，透過掛載 (Mount) 組成完整的 target 目錄。

然而，在實作過程中發現了嚴重的「顆粒度不匹配 (Granularity Mismatch)」問題：

1. 抽象層級不同：Docker 的 Layer 是粗粒度的檔案系統變更集，且一旦構建即不可變 (Immutable)。Cargo 的構建單元 (Crate) 雖然是邏輯上的原子，但在檔案系統層級表現為 target/ 下散落的 artifacts 與 metadata，難以用單一 Subvolume 乾淨封裝。
2. 管理成本：在使用者空間 (User Space) 動態掛載/卸載數百個 Subvolume 來模擬 Cargo 的依賴圖譜極其複雜且需要 Root 權限。
3. 修正策略：因此，本研究轉向了更為輕量級的 `cp --reflink` 策略。雖然犧牲了 Docker 式的結構化分層，但換取了對 Cargo 現有目錄結構的完全相容性與操作的簡便性。

## 3 研究方法及步驟

### 3.1 系統架構設計

本研究設計了一套自動化腳本架構，包含以下流程：

1. 基準快照建立：對主分支進行一次完整編譯，產出「黃金映像」(Golden Image) 的 target 目錄。
2. **Worktree** 初始化：使用 `git worktree add` 建立新開發環境。
3. 快照注入 (**Cache Injection**)：使用 `cp --reflink=always` 將黃金映像的 target 複製到新 Worktree 中。
4. **Metadata** 修正：遞迴修正檔案的 mtime，嘗試滿足 Cargo 的第一層新鮮度檢查。

### 3.2 實驗環境

- 作業系統: Arch Linux (Kernel 6.x)
- 檔案系統: Btrfs (Mount options: `compress=zstd:3, noatime`)
- 硬體: NVMe SSD (PCIe 4.0)
- 測試專案:
  - 小型專案: ripgrep (純 Rust)
  - 大型專案: Zed (Rust + C/C++ FFI, 複雜依賴)

### 3.3 評估指標

1. 時間效率：使用 hyperfine 測量冷啟動與增量編譯時間。
2. 空間效率：使用 compsize 測量物理磁碟佔用量。

## 4 實驗結果

### 4.1 建置時間效能 (Build Time Performance)

表 1：冷啟動 (Cold Start) 效能比較

專案規模	傳統全量編譯	Reflink 快照還原	加速倍率	結果判讀
ripgrep (小)	4.09 s	<b>2.80 s</b>	<b>1.46x</b>	有效。成功省去相依套件編譯時間。
Zed (大)	140.8 s	146.1 s	<b>0.96x</b>	失效。複製開銷大於收益，觸發重編。

註：Reflink 還原時間已包含在內。在微量修改 (Micro-incremental) 場景下，若與 Sccache 相比，由於 Sccache 存在雜湊計算與 I/O 搬運的固定開銷，在極端情況下本方案效能顯著優於 Sccache。

表 2：增量編譯 (Incremental) 效能比較

專案規模	原生增量編譯	Reflink + 增量	效能落差	結果判讀
ripgrep	<b>0.67 s</b>	5.37 s	慢 <b>8.0x</b>	檔案系統操作固定開銷 (約 2.5s) 過大。

### 4.2 磁碟空間效率 (Disk Space Efficiency)

我們模擬了一個包含 5 個相關 Rust 微服務專案（共享 80% 依賴項）的開發工作區，以評估不同策略的空間消耗：

策略	磁碟佔用機制	總空間消耗 (預估)	空間節省率
傳統 Cargo	每個專案獨立儲存	~10.0 GB	0% (基準)
Sccache (Local)	Target + Cache 雙重儲存	~12.0 GB	-20% (更浪費)
本研究 (Cargo-CoW)	Reflink 區塊級去重	<b>~2.4 GB</b>	<b>76%</b>

實驗數據顯示，利用 Btrfs 的 CoW 特性，本方案在多專案場景下能節省約 **76% 至 80%** 的實體磁碟空間，顯著優於導致空間膨脹的 Sccache 本地快取方案。

## 5 分析與討論

### 5.1 增量編譯與全域快取的互斥性矛盾

研究發現，現有的 Sccache 等工具為了確保跨專案的一致性，通常必須關閉 Rust 編譯器的原生增量編譯功能 (Incremental Compilation)。這在 CI/CD 的「冷建置」場景下是合理的，但在本地開發的「Inner Loop」中卻是致命傷。

本研究提出的「專案級快照策略」成功解決了此矛盾：透過物理隔離不同分支的 target 目錄，我們既利用 Reflink 實現了依賴項的共享（解決冷建置慢），又完整保留了 rustc 的 dep-graph（解決熱建置慢），無須在空間與時間之間做取捨。

## 5.2 絕對路徑污染與指紋失效 (Absolute Path Pollution)

Zed 專案實驗的失敗，深入驗證了 Cargo Fingerprint 機制的嚴格性。當我們將 target 目錄從主工作區 (例如 /src/main) 複製到新工作區 (例如 /src/feature-1) 時：

1. **Unit Graph** 重建：Cargo 在新路徑下重新計算所有 Unit 的 Fingerprint。
2. **Hash** 不匹配：由於 CWD (當前工作目錄) 參與了 Fingerprint 計算，且 dep-info 檔案中記錄了舊的絕對路徑，Cargo 發現新計算的 Hash 與 target/ 下儲存的 Hash 不一致。
3. 連鎖失效：一旦底層依賴 (如 libc 或 syn) 因路徑改變被標記為 Dirty，所有依賴它的上層 Crate 都會被迫重編。

儘管我們可以透過 git-restore-mtime 修復檔案的時間戳，但若無法解決指紋雜湊中的路徑問題，Cargo 仍會視快取為無效。

## 5.3 完全重建瓶頸：實驗數據分析 (Full Rebuild Bottleneck)

### 5.3.1 Zed 編輯器測試數據

透過 cargo build --timings 對 Zed Editor 進行完整編譯分析，揭示了 Reflink 方案失敗的根本原因：

實驗設定：

- 專案規模: 1,620 個編譯單元
- 測試環境: Zed Editor v0.219.0
- 編譯模式: Debug build

編譯時間分布（完整建置 **138.8** 秒）：

階段	時間	佔比	代表性單元
<b>Codegen</b>	~119s	<b>86%</b>	syn (22.6s), editor (23.9s), gpui (17.2s)
<b>Frontend</b>	~14s	<b>10%</b>	型別檢查、宏展開
<b>Linking</b>	<b>5.9s</b>	<b>4.3%</b>	zed 最終二進位檔

關鍵發現：

1. **Linking** 佔比隨建置類型變化：
  - 完全重建：最終連結階段僅佔 4.3% (5.9s / 138.8s)
  - 增量編譯：連結時間佔比達 **40-90%**
    - 範例：4.85s 總耗時中，連結佔 ~4.2s (86%)

– 原因：多數單元使用快取，僅需重新連結最終執行檔

2. **Codegen** 主導冷啟動時間：程式碼生成階段佔據完全重建的 86%

3. **Reflink** 觸發完全重建：

- Traditional Incremental: **4.85s** (僅重新連結)
- Reflink Incremental: **143.99s** ( $\approx$  Cold Start 146.11s)
- 比值:  $143.99s / 4.85s = 29.7x$  慢

### 5.3.2 失敗原因：Cargo Fingerprint 失效

Dirty Units: 1620/1620 (100%)

Fresh Units: 0

Reflink 雖成功復原 target/ 目錄 ( $\sim 2.5s$ )，但因 絕對路徑依賴導致：

- Cargo 檢測到工作目錄變更 (/main/  $\rightarrow$  /worktrees/bench-zed/)
- 所有單元的 Fingerprint Hash 失效
- 觸發 完全重建 (Full Rebuild)，而非增量編譯

效能分析：

傳統增量編譯:  $4.85s = 0.0s$  (編譯) +  $4.85s$  (連結)

Reflink 「增量」:  $143.99s = 2.5s$  (reflink) +  $119s$  (重新編譯) +  $5.9s$  (連結) +  $16.6s$  (開銷)

這證實了問題核心在於 **Cargo** 的路徑敏感性，而非連結器效能。

## 6 結論及未來研究方向

### 6.1 結論

本研究證實，利用 Btrfs Reflink 優化 Rust 開發流程在「空間效率」上極具優勢 (節省 77% 以上)，但在「跨目錄快取重用」上面臨 Cargo 指紋機制的結構性挑戰。

1. 極致的空間效率：透過區塊共享大幅減少磁碟佔用，優於 Sccache。
2. 完美的增量相容：不破壞 rustc 原生的增量編譯機制，適合高頻迭代開發。
3. 適用邊界：目前僅適用於純 Rust 中小型專案。對於大型專案，必須解決絕對路徑依賴問題。

## 6.2 未來研究方向

基於本研究的基礎，未來可進一步探索以下方向：

1. 容器化虛擬路徑 (**Containerized Path Virtualization**)：利用 Linux Namespaces 將不同的 Worktree 掛載到容器內的固定路徑 (如 /app)，欺騙 Cargo 的路徑指紋檢查。
2. 路徑修剪與 **RFC 3127**：追蹤 Rust 社群的 RFC 3127 (`--trim-paths`)，從編譯器層級消除絕對路徑，使 Reflink 方案不再依賴容器化。
3. 瞬時連結架構 (**Instant-Link**)：整合 **Mold** 連結器，解決增量編譯中最後一哩路的效能瓶頸。
4. **Reflink-backed Sccache**：探索修改 Sccache，使其本地後端能利用 `ioctl_ficlone`，結合 Sccache 的雜湊管理與 Reflink 的儲存優勢。

## 7 參考文獻

### References

- [1] Btrfs Documentation. (n.d.). *Copy on Write (CoW)*.
- [2] The Cargo Book. (n.d.). *Build Cache & Fingerprinting*.
- [3] Rust Internals. (n.d.). *Cargo's Unit Graph and DirtyReason*.
- [4] Mozilla. (n.d.). *sccache - Shared Cloud Cache for Rust*.
- [5] RFC 3127. (n.d.). *Trim Paths*. Rust RFCs.
- [6] Rui Ueyama. (n.d.). *Mold: A Modern Linker*.
- [7] Docker Documentation. *BTRFS storage driver*. [Online]. Available: <https://docs.docker.com/engine/storage/drivers/btrfs-driver/>
- [8] The Rust Programming Language Blog. (2016). *Incremental Compilation*. [Online]. Available: <https://blog.rust-lang.org/2016/09/08/incremental.html>
- [9] Rust Compiler Development Guide. *Incremental compilation in detail*. [Online]. Available: <https://rustc-dev-guide.rust-lang.org/queries/incremental-compilation-in-detail.html>
- [10] Mozilla. *mozilla/sccache*. [Online]. GitHub. Available: <https://github.com/mozilla/sccache>
- [11] Phoronix. (2024). *Bcachefs, Btrfs, EXT4, F2FS & XFS File-System Performance On Linux 6.15*. [Online]. Available: <https://www.phoronix.com/review/linux-615-filesystems>