# HỌC VIỆN KỸ THUẬT MẬT MÃ



**KHOA CÔNG NGHỆ THÔNG TIN**

BÁO CÁO MÔN HỌC

# TỐI ƯU PHẦN MỀM DI ĐỘNG

**Đề tài:**

# NGHIÊN CỨU BÀI BÁO ANDROID MEMORY

**OPTIMIZATION**

**Giảng viên hướng dẫn: ThS. Lê Bá Cường**

**Sinh viên thực hiện: Trương Quang Nghĩa - CT040335**

**Nguyễn Phúc Sơn - CT040343 Hồ Minh Thông - CT040346**

**Nhóm 9**

*Hà Nội, 2023*

# MỤC LỤC

[LỜI MỞ ĐẦU 4](#_TOC_250035)

[CHƯƠNG I: GIỚI THIỆU VỀ TỐI ƯU HÓA BỘ NHỚ 5](#_TOC_250034)

[CHƯƠNG II: CÁC VẤN ĐỀ VỀ TỐI ƯU HÓA BỘ NHỚ 7](#_TOC_250033)

* 1. [Memory Leaks (Rò rỉ bộ nhớ) 7](#_TOC_250032)
  2. [Quản lý rò rỉ bộ nhớ 8](#_TOC_250031)
     1. [Ưu tiên Trường Hợp Thử Nghiệm 8](#_TOC_250030)
     2. [Tạo Thử Nghiệm để Phát Hiện Rò Rỉ 8](#_TOC_250029)
  3. [Quản lý bộ đệm GPU 8](#_TOC_250028)
  4. [Tránh trùng lặp bộ nhớ 9](#_TOC_250027)
  5. [Quản lý ứng dụng nền linh hoạt 10](#_TOC_250026)
  6. [Áp dụng những tối ưu hóa nhỏ 11](#_TOC_250025)
  7. [Phát hiện việc lập trình kém 11](#_TOC_250024)
  8. [Phân vùng bộ nhớ 12](#_TOC_250023)
  9. [Xử lý bộ nhớ bằng cách sử dụng Logs 12](#_TOC_250022)
  10. [Phát hiện Anti-Patterns 13](#_TOC_250021)
  11. [Giảm thiểu quá trình lão hóa phần mềm 13](#_TOC_250020)
  12. [Non-Blocking Garbage Collector 14](#_TOC_250019)
  13. [Sử dụng bộ nhớ Non-Volatile 14](#_TOC_250018)

[CHƯƠNG III: GIẢI PHÁP TỐI ƯU HÓA BỘ NHỚ 16](#_TOC_250017)

[Đề xuất giải pháp 16](#_TOC_250016)

* 1. [Sử dụng các services một cách tiết kiệm 20](#_TOC_250015)
  2. [Sử dụng các vùng chứa dữ liệu được tối ưu hóa 20](#_TOC_250014)
  3. [Cẩn thận với các code abstractions 21](#_TOC_250013)
  4. [Sử dụng protobufs lite cho dữ liệu được tuần tự hóa 21](#_TOC_250012)
  5. [Tránh xáo trộn bộ nhớ 21](#_TOC_250011)
  6. [Xóa các tài nguyên và thư viện sử dụng nhiều bộ nhớ 22](#_TOC_250010)
  7. [Giảm kích thước APK tổng thể 23](#_TOC_250009)
  8. [Dependency injection frameworks 23](#_TOC_250008)
  9. [Hãy cẩn thận khi sử dụng các thư viện bên ngoài 23](#_TOC_250007)

[CHƯƠNG IV: TIẾN HÀNH THỰC NGHIỆM 25](#_TOC_250006)

* 1. [Sử dụng tham chiếu yếu 26](#_TOC_250005)
  2. [Sử dụng ProGuard 30](#_TOC_250004)
  3. [Sử dụng cache 30](#_TOC_250003)
  4. [Sử dụng ARC 31](#_TOC_250002)
  5. [Tối ưu hóa ảnh 37](#_TOC_250001)

[KẾT LUẬN 39](#_TOC_250000)

**DANH MỤC ẢNH**

Hình 1: Cách hoạt động của ARC 16

Hình 2: Android ImageView loading 17

Hình 3: iOS ImageView loading 17

Hình 4: Tạo động một Đối tượng tham chiếu trong iOS 18

Hình 5: Android Button Registering and Unregistering event 18

Hình 6: iOS Button Registering event 19

Hình 7: Memory Profiler 25

Hình 8: LeakCanary 25

Hình 9: Kích thước của ứng dụng trước và sau khi dùng Proguard 30

Hình 10: Code tích hợp cache vào ứng dụng phát video 30

Hình 11: Kết quả khi sử dụng cache 31

Hình 12: Glide 38

# LỜI MỞ ĐẦU

## Về đề tài

Trong bối cảnh hiện nay, khi công nghệ di động phát triển với tốc độ nhanh chóng, hệ điều hành Android đóng vai trò quan trọng trong việc đáp ứng nhu cầu ngày càng cao của người sử dụng. Với sự phổ biến của các ứng dụng di động và đa dạng của các thiết bị Android, việc tối ưu hóa bộ nhớ trở thành một thách thức đối với những nhà phát triển và người quản lý hệ thống.

Phần mềm di động hiện nay được phát triển cho hai nền tảng chính là Android và iOS, nhưng với lợi thế là hệ điều hành mở Android đã vươn lên là một hệ điều hành được sử dụng phổ biến nhất cho smartphone.

Với tiềm năng và sự phát triển mạnh thì việc các sản phẩm phần mềm cũng ngày càng cạnh tranh hơn, từ đó chất lượng các phần mềm dần được nâng cao và cải thiện hơn về hiệu năng, độ tin cậy, tính an toàn,… Bản báo cáo với nhiệm vụ nghiên cứu về tối ưu hóa bộ nhớ trên nền tảng Android, nhằm cải thiện hiệu suất và trải nghiệm người dùng. Mục tiêu của đề tài báo cáo là cung cấp cái nhìn tổng quát về tối ưu hóa bộ nhớ, giới thiệu các kỹ thuật tối ưu hóa bộ nhớ, giới thiệu một số phần mềm giúp tối ưu hóa bộ nhớ nhằm tạo ra một ứng dụng tối ưu tốt về bộ nhớ.

## Cấu trúc báo cáo

Ngoài phần mở đầu và kết luận, bản báo cáo gồm 4 phần:

* Chương 1: Giới thiệu về tối ưu hóa bộ nhớ
* Chương 2: Các vấn đề về tối ưu hóa bộ nhớ
* Chương 3: Giải pháp tối ưu hóa bộ nhớ
* Chương 4: Thực nghiệm

Cuối cùng, nhóm chúng em xin gửi lời cảm ơn sâu sắc tới thầy. Người đã dành nhiều thời gian của mình hướng dẫn tận tình, chỉnh sửa và tạo điều kiện để nhóm có thể hoàn thiện đề tài một cách tốt nhất. Chúng em xin chân thành cảm ơn!

# CHƯƠNG I: GIỚI THIỆU VỀ TỐI ƯU HÓA BỘ NHỚ

Ngày nay, điện thoại thông minh đã trở thành một nhu cầu thiết yếu đối với tất cả mọi người. Hầu hết chúng ta đều phụ thuộc quá nhiều vào điện thoại thông minh để hoàn thành công việc hàng ngày. Ở một số khía cạnh, chúng đã thay thế được máy tính của chúng ta. Cùng với thời gian, điện thoại thông minh ngày càng phát triển và các thiết bị ngày càng trở nên mạnh mẽ.

**Tính linh hoạt trong so sánh**

Chỉ số AI Jaccard không giới hạn trong việc so sánh từng từ riêng lẻ. Nó cũng có thể được áp dụng cho các đơn vị ngôn ngữ phức tạp hơn, chẳng hạn như câu hoặc đoạn văn. Tính linh hoạt này làm cho nó trở thành một công cụ có giá trị trong các nhiệm vụ NLP liên quan đến việc phân tích và so sánh dữ liệu văn bản.

**Xử lý khối lượng dữ liệu lớn**

Một trong những ưu điểm chính của việc sử dụng Chỉ số AI Jaccard trong NLP là khả năng xử lý khối lượng lớn dữ liệu một cách hiệu quả. Các phương pháp so sánh văn bản truyền thống, chẳng hạn như so khớp chuỗi hoặc tương tự cosine, có thể tốn kém về mặt tính toán và tốn thời gian. Ngược lại, Chỉ số Jaccard cung cấp một cách đơn giản và nhanh chóng để đo lường mức độ tương tự, khiến chỉ số này trở nên lý tưởng để xử lý các tập dữ liệu lớn.

Hơn nữa, Chỉ số AI Jaccard không phụ thuộc vào ngôn ngữ, nghĩa là nó có thể được áp dụng cho văn bản bằng bất kỳ ngôn ngữ nào. Điều này làm cho nó trở thành một công cụ linh hoạt dành cho các nhà nghiên cứu và thực hành NLP làm việc với dữ liệu đa ngôn ngữ.

**Phát hiện đạo văn**

Một ứng dụng quan trọng khác của Chỉ số AI Jaccard trong NLP là trong lĩnh vực phát hiện đạo văn. Bằng cách so sánh sự giống nhau giữa một văn bản nhất định và cơ sở dữ liệu của các văn bản hiện có, các thuật toán có thể xác định các trường hợp đạo văn tiềm ẩn. Điều này đặc biệt hữu ích trong môi trường học thuật và xuất bản, nơi tính độc đáo và ghi công là tối quan trọng.

**Kết luận**

Tóm lại, Chỉ số AI Jaccard là một công cụ có giá trị trong Xử lý ngôn ngữ tự nhiên. Khả năng đo lường sự giống nhau giữa các bộ dữ liệu, bất kể ngôn ngữ hay độ phức tạp, khiến nó trở thành một thành phần thiết yếu trong các nhiệm vụ NLP khác nhau. Từ phân loại văn bản đến phát hiện đạo văn, Jaccard Index cung cấp một cách nhanh chóng và hiệu quả để phân tích và so sánh dữ liệu văn bản. Khi NLP tiếp tục phát triển, Chỉ số AI Jaccard chắc chắn sẽ đóng một vai trò quan trọng trong việc định hình tương lai của công nghệ xử lý ngôn ngữ.

Android là hệ điều hành điện thoại thông minh được sử dụng phổ biến nhất hiện nay vì nó là mã nguồn mở và các nhà sản xuất có thể dễ dàng tích hợp nó vào phần cứng của họ, giúp cho các thiết bị Android có giá thành rẻ hơn đối thủ cạnh tranh là iOS. Ngoài những mặt tích cực, nó cũng có những mặt tiêu cực. Một trong những vấn đề lớn mà người dùng Android phải đối mặt là gặp các sự cố không mong muốn làm chậm thiết bị của họ. Những vấn đề như vậy chủ yếu gây ra khi thiết bị hết bộ nhớ. Các nhà sản xuất điện thoại tiếp tục tăng bộ nhớ chính để bù đắp nhưng đây không phải là giải pháp hiệu quả nhất.

Hiện tại, Hệ điều hành Android chạy trên cấu trúc bộ nhớ được biết đến là Bộ thu gom rác (Garbage Collector). Bộ thu gom rác là một công cụ Quản lý Bộ nhớ của Java do Android được xây dựng trên Máy ảo Java (JVM). Bộ thu gom rác theo dõi và xác định các đối tượng đã chết và giải phóng không gian khi chúng không còn cần thiết nữa. Nó có thể giải phóng bộ nhớ theo hai cách cơ bản. Đầu tiên là bằng cách kiểm tra định kỳ các đối tượng đã chết và giải phóng bộ nhớ của chúng, thứ hai, ngay lập tức giải phóng bộ nhớ khi có một phân bổ nào đó lớn hơn bộ nhớ trống có sẵn. Hãy giả sử chúng ta có một điện thoại thông minh Android với bộ nhớ chính là 1 GB. Bây giờ, có ba đối tượng là a, b và c, và khi chúng được tạo ra, bộ nhớ được cấp phát cho mỗi đối tượng theo nhu cầu của chúng. Khi những đối tượng này bị hủy sau khi phục vụ mục đích của chúng, chúng được đánh dấu là đã chết nhưng bộ nhớ vẫn chưa được giải phóng hoàn toàn. Trong tình huống này, nếu một đối tượng mới cần được cấp phát bộ nhớ và kích thước phân bổ của nó lớn hơn bộ nhớ trống có sẵn, thì sẽ xảy ra sự cố. Điều này xảy ra do Bộ thu gom rác chưa giải phóng không gian đã chết qua chu kỳ định kỳ của nó cho đến bây giờ. Khi nó giải phóng

không gian đã chết ngay sau khi nhận được một đối tượng mới có yêu cầu cao hơn, điều này sẽ gây ra một trục trặc và trải nghiệm người dùng kém chất lượng. Hơn nữa, JVM sao chép đối tượng từ một nơi sang nơi khác trong quá trình Thu gom rác và không ghi đè lên dữ liệu đã được thu gom cũ. Điều này có thể dẫn đến rò rỉ thông tin và thông tin cá nhân có thể bị đánh cắp.

Để giải quyết những vấn đề này và cải thiện quản lý bộ nhớ, đã có nhiều giải pháp được đề xuất bằng cách giữ bộ nhớ phụ trống bằng cách sử dụng GPU Buffers, Quản lý Ứng dụng Nền Linh hoạt, áp dụng các tối ưu hóa nhỏ, tránh Sự trùng lặp Bộ nhớ, phát hiện các thói quen lập trình xấu và khắc phục chúng v.v. Tất cả những kỹ thuật này đều cung cấp các tối ưu hóa trong hệ thống quản lý bộ nhớ hiện tại, đó là Bộ thu gom rác. Tuy nhiên, vấn đề chính vẫn tồn tại, đó là làm thế nào Bộ thu gom rác giải phóng bộ nhớ của các đối tượng đã chết.

Mục đích của nghiên cứu là đề xuất một phương pháp hoàn toàn khác để giải quyết vấn đề này. Việc sử dụng Đếm Tham Chiếu Tự động (Automatic Reference Counting - ARC) thay vì Bộ Thu Gom Rác trong thiết bị Android có thể là một hướng tiếp cận hướng tới việc cải thiện các vấn đề về bộ nhớ.

# CHƯƠNG II: CÁC VẤN ĐỀ VỀ TỐI ƯU HÓA BỘ NHỚ

Tất cả các kỹ thuật được đề xuất cho đến nay đều xoay quanh Bộ Thu Gom Rác. Do đó, trước hết, các vấn đề liên quan đến bộ nhớ Android được thảo luận do đó Android thể hiện hiệu suất kém. Sau đó, các kỹ thuật đề xuất để cải thiện những vấn đề đó sẽ được thảo luận chi tiết.

## Memory Leaks (Rò rỉ bộ nhớ)

Android chạy trên một nhân Linux được tùy chỉnh, điều này điều khiển phần cứng như Bộ nhớ và CPU và cho phép giao tiếp giữa phần cứng và người dùng. Hai thành phần chính của ứng dụng Android là Activities và Fragments. Mã nguồn không tối ưu cho những thành phần này, thường là quản lý không hiệu quả tài nguyên, có thể dẫn đến nhiều rò rỉ bộ nhớ và do đó gây ra vấn đề về bộ nhớ. Chi tiết về những thành phần này được thảo luận dưới đây:

* Activities (Hoạt động) là các thành phần cơ bản của bất kỳ Hệ điều hành Android nào. Chúng thực sự đại diện cho một Bộ điều khiển Xem. Mỗi lần một chế độ xem mới được hiển thị, đó là một hoạt động mới, và khi chế độ xem đó bị đóng, hoạt động cũng bị đóng.
* Fragments (Đoạn mã) là các thành phần của Giao diện người dùng. Chúng là các thành phần có thể tái sử dụng có thể được sử dụng trong nhiều hoạt động khác nhau. Tuổi thọ của chúng phụ thuộc vào tuổi thọ của một hoạt động. Khi một hoạt động được tạo, các đoạn mã cũng được tạo. Khi một hoạt động bị hủy, các đoạn mã cũng bị hủy.

Hầu hết các rò rỉ bộ nhớ xảy ra trong mã nguồn không tối ưu của Activities và Fragments, chẳng hạn như quên tái chế các thể hiện bitmap, hủy đăng ký sự kiện xử lý click, đóng các thể hiện con trỏ sau khi truy cập cơ sở dữ liệu và tham chiếu đối tượng từ các lớp tĩnh hoặc đánh dấu đối tượng làm tĩnh.

## Quản lý rò rỉ bộ nhớ

Đã có nhiều phương pháp được đề xuất để giảm và loại bỏ rò rỉ bộ nhớ như sử dụng LeakDAF, tạo các trường hợp thử nghiệm để xác định và sửa chữa rò rỉ bộ nhớ bằng cách tuân theo các hướng dẫn lập trình cụ thể.

LeakDAF sử dụng kỹ thuật thử nghiệm UI để chạy ứng dụng và phân tích các tệp bảng xếp hạng bộ nhớ để xác định các hoạt động và đoạn mã đã rò rỉ.

Phương pháp tạo trường hợp thử nghiệm để xác định và sửa chữa rò rỉ bộ nhớ đã được các nhà nghiên cứu đề xuất theo hai cách riêng biệt như sau:

## Ưu tiên Trường Hợp Thử Nghiệm

Sử dụng một phương pháp để ưu tiên các trường hợp thử nghiệm và chạy những trường hợp đó theo một thứ tự cụ thể thay vì chạy tất cả các trường hợp thử nghiệm vì chúng có thể tốn kém và dẫn đến tăng cường tải trên CPU. Ưu tiên của các trường hợp thử nghiệm được xác định bằng cách triển khai các thuật toán học máy dự đoán độ chính xác của một trường hợp thử nghiệm để xác định một rò rỉ bộ nhớ.

## Tạo Thử Nghiệm để Phát Hiện Rò Rỉ

Đề xuất một kỹ thuật xác định một chuỗi sự kiện GUI tự nhiên như khởi chạy và đóng ứng dụng. Lặp lại các sự kiện như vậy không nên tăng sử dụng bộ nhớ nếu không có rò rỉ bộ nhớ. Nếu có rò rỉ, bộ nhớ sẽ tiếp tục tăng lên.

## Quản lý bộ đệm GPU

GPU (Đơn vị Xử lý Đồ họa) tương tự như CPU nhưng được sử dụng để xử lý đồ họa. Theo thời gian, ứng dụng và trò chơi ngày càng phức tạp và yêu cầu xử lý đồ họa mạnh mẽ, vì vậy cần có một GPU riêng biệt. Thiết bị di động được trang bị GPU giống như máy tính để bàn, nhưng có sự khác biệt lớn giữa các GPU của chúng. GPU dành cho máy tính để bàn có bộ nhớ riêng biệt, nhưng do kích thước nhỏ và tính di động của thiết bị di động, GPU của điện thoại thông minh không có bộ nhớ riêng mà chia sẻ bộ nhớ chính. Việc chia sẻ này giảm bớt bộ nhớ chính có sẵn.

Android lưu trữ ứng dụng để chúng có thể được khởi chạy nhanh chóng lần sau khi chúng được gọi. Khi được lưu trữ, bộ đệm GPU cũng được lưu trữ vì ứng dụng chứa đồ họa. Khi ứng dụng bị kết thúc, dữ liệu đã được lưu trữ của nó vẫn còn trong bộ nhớ chính, do đó làm tăng sử dụng bộ nhớ và giảm bớt bộ nhớ có sẵn.

Một trong những giải pháp hiện tại là nén bộ nhớ của GPU khi ứng dụng được đưa vào nền và giải nén khi quay lại mặt trước. Bộ nhớ nén sẽ chiếm rất ít không gian và sẽ cải thiện hiệu suất tổng của thiết bị. Tuy nhiên, quá trình nén và giải nén này sẽ tăng thời gian truy cập, làm cho giải pháp này ít tối ưu hóa hơn.

## Tránh trùng lặp bộ nhớ

Trùng lặp bộ nhớ là quá trình trong đó các trang bộ nhớ giống nhau được đặt trong bộ nhớ chính nhiều lần hơn một lần. Bằng cách sử dụng kỹ thuật trùng lặp này, khi Bộ Thu Gom Rác giải phóng không gian, nó sử dụng rất nhiều chu kỳ CPU, gây giật và đôi khi thậm chí là crash ứng dụng. Do đó, tránh trùng lặp trang là cần thiết để Bộ Thu Gom Rác không phá hủy nội dung giống nhau nhiều lần. Để khắc phục những vấn đề như vậy, Android đã giới thiệu một số cơ chế như zRAM và Kernel Same-Page Merging (KSM). KSM hợp nhất các trang bộ nhớ giống nhau thành một. zRAM sử dụng một khu vực được cấp phát đặc biệt trong bộ nhớ chính được biết đến là khu vực đổi, nơi nó nén các trang đã lưu trữ cần được đổi chỗ. Cả hai cơ chế này giảm sử dụng bộ nhớ nhưng tiêu tốn số lượng lớn chu kỳ CPU và năng lượng.

Memscope là một công cụ có thể được sử dụng để tránh những vấn đề như vậy. Nó lấy bản chụp bộ nhớ ở các khoảng thời gian cố định. Nó xác định các khung trùng lặp có thể tồn tại trong một khoảng thời gian cụ thể và làm thế nào chúng sẽ thay đổi qua vòng đời của ứng dụng. Nó phân tích các bản chụp và xác định các khung có khả năng trùng lặp và tập trung vào những khung đó để tránh trùng lặp.

Một Phương pháp khác hiện đang được sử dụng để giảm và loại bỏ Trùng Lặp Bộ Nhớ là Trùng Lặp Bộ Nhớ Chọn Lọc. Thay vì quét toàn bộ bộ nhớ, nó quét một số trang bộ nhớ cụ thể để giảm sử dụng CPU và năng lượng. Cơ chế này quét các trang bộ nhớ của các ứng dụng chỉ một lần trong nền cho đến khi chúng được đưa lên phần trước vì khả năng rất ít khi kích thước bộ nhớ sẽ thay đổi cho các ứng dụng đang chạy ở nền.

## Quản lý ứng dụng nền linh hoạt

Hệ điều hành Android duy trì các ứng dụng được sử dụng gần đây trong bộ nhớ cache thông qua hệ thống Tang để tăng tốc độ nạp lại, giảm thời gian khởi chạy và tiêu thụ năng lượng. Trạng thái của một ứng dụng khác nhau khi nó được khởi chạy và khi nó bị dừng. Trong Hệ điều hành Android, quá trình Zygote tích hợp sẵn đóng vai trò khởi chạy ứng dụng, cũng như nhận và phản hồi sự kiện của người dùng. Quá trình này cũng được sử dụng để quản lý bộ nhớ trong Android. Khi thiết bị trải qua giai đoạn bộ nhớ thấp, Low Memory Killer (LMK) bắt đầu hoạt động và kết thúc ứng dụng ít được sử dụng gần đây nhất (LRU). Nếu ứng dụng cần bị kết thúc có kích thước lớn, việc giải phóng bộ nhớ sẽ mất thời gian khi nó được khởi chạy lần tiếp theo. Một kỹ thuật khác là Out of Memory Killer (OOMK), giết chết các ứng dụng có ưu tiên thấp, giết chết nhiều ứng dụng cùng một lúc để giải phóng bộ nhớ, điều này có thể ảnh hưởng xấu đối với các ứng dụng này. Tóm lại, những cơ chế này gặp phải những vấn đề cơ bản sau:

1. Ứng dụng sử dụng bộ nhớ cao được ưu tiên để kết thúc.
2. Thời gian khởi chạy ứng dụng không được xem xét khi chọn người chết.
3. Việc thu hồi bộ nhớ là một quy trình theo yêu cầu và phải đưa ra quyết định nhanh chóng, vì vậy một thuật toán tối ưu hóa có thể dẫn đến việc kết thúc các ứng dụng không đúng.

Dịch vụ Quản lý Hoạt động (AMS) có trách nhiệm xử lý yêu cầu của người dùng. Nó duy trì các ứng dụng trong bộ nhớ cache dựa trên ưu tiên, được xác định theo thứ tự sau:

1. Ứng dụng đang chạy ở phía trước.
2. Một quy trình được ràng buộc với ứng dụng đang chạy ở phía trước.
3. Một quy trình được ràng buộc với ứng dụng đang chạy ở phía sau.
4. Các quy trình ẩn có trên thiết bị nhưng không hiển thị.
5. Các ứng dụng cung cấp nội dung như lịch và email.
6. Các quy trình trống rỗng được lưu trữ để tăng tốc độ nạp lại.

Các chính sách lưu trữ hiện tại có phần tĩnh lặp và không thay đổi tùy thuộc vào sở thích sử dụng ứng dụng của người dùng. Các nhà nghiên cứu đã đề xuất một giải pháp tốt

hơn, lấy into account tính tái sử dụng của ứng dụng để xác định số lượng ứng dụng động cần được lưu trữ. Kỹ thuật này cho phép quản lý bộ nhớ chính một cách hiệu quả để có cách lưu trữ tốt hơn và do đó cải thiện hiệu suất tổng thể của thiết bị.

Một đề xuất là sử dụng một công cụ được gọi là SmartLMK. Đây là một quy trình động, về cơ bản là một ứng dụng bộ nhớ thấp đang chạy ở nền. Nó theo dõi thời gian khởi chạy của ứng dụng, dữ liệu sử dụng và các đặc điểm khác. Sử dụng các thống kê này, nó tính toán một phạt thời gian và sử dụng nó để kết thúc một ứng dụng.

## Áp dụng những tối ưu hóa nhỏ

Tối ưu hóa cần được áp dụng cho ứng dụng di động vì chúng thường xuyên gặp vấn đề về hiệu suất. Micro-tối ưu hóa nên được áp dụng ở đầu vòng đời của ứng dụng vì dễ tối ưu hóa khi cấu trúc đang được viết so với khi nó đã được viết hoàn toàn. Theo các kết quả nghiên cứu, việc loại bỏ biến và phương thức riêng tư không sử dụng cải thiện hiệu suất vì nó giảm bớt kích thước bộ nhớ.

Thường thì các nhà phát triển không áp dụng micro-tối ưu hóa vì:

1. Hầu hết họ không biết về điều này.
2. Họ nghĩ rằng ứng dụng của họ quá nhỏ để áp dụng bất kỳ tối ưu hóa nào.
3. Họ không nghĩ rằng việc dành thời gian cho micro-tối ưu hóa là đáng giá.
4. Họ không tin rằng micro-tối ưu hóa sẽ giúp ích cho ứng dụng của họ.

Các kết quả được thu được thông qua phân tích tĩnh được thực hiện bằng cách sử dụng các công cụ như FindBugs, PMD và LINT cung cấp cảnh báo để cải thiện mã nguồn. Một trong những tối ưu hóa hữu ích nhất được tìm thấy là loại bỏ mã không sử dụng. Nhưng các công cụ và kỹ thuật như vậy có thể dễ mắc lỗi và không thể tin tưởng hoàn toàn.

## Phát hiện việc lập trình kém

Một giải pháp khác được các nhà nghiên cứu đề xuất là công cụ CheckDroid được sử dụng để xác định thực hành lập trình kém. Việc sửa chữa những điều này có thể cải thiện hiệu suất ứng dụng và bộ nhớ tổng thể của hệ điều hành Android. Một công cụ như vậy là cần thiết vì thực hành kém thường không được xác định và báo cáo bởi môi trường

phát triển tích hợp (IDE). Hầu hết các công cụ và kỹ thuật được áp dụng để phát hiện rò rỉ bộ nhớ và hiệu suất, nhưng ít chú ý đến thực hành lập trình kém.

Một số đề xuất về hiệu suất sau khi phân tích là:

1. Logging chi tiết không nên được giữ lại trong ứng dụng hoạt động.
2. Các công việc chạy lâu dài nên được chia thành các luồng con.
3. Những luồng con này nên có ưu tiên thấp hơn so với luồng chính.

Một số tối ưu hóa chính được đề xuất là:

1. Tham chiếu đến ngữ cảnh không nên được lưu trữ trong biến tĩnh.
2. Các luồng được tạo ra nên được hủy bỏ khi không cần thiết.

## Phân vùng bộ nhớ

Một đề xuất khác là phân vùng bộ nhớ chính vì Android sử dụng LMK và OOMK để giải phóng bộ nhớ bằng cách kết thúc các quy trình nạn nhân, điều này không hiệu quả lắm. Vì vậy, đề xuất một kỹ thuật phân vùng bộ nhớ chia bộ nhớ thành hai nút ảo.

1. Nút ảo 0 được sử dụng cho các ứng dụng đáng tin cậy.
2. Nút ảo 1 được sử dụng cho các ứng dụng không đáng tin cậy.

Nếu bộ nhớ cạn kiệt từ một nút, chỉ bộ nhớ của nút đó sẽ được giải phóng. Thông thường, các ứng dụng không đáng tin cậy chiếm nhiều bộ nhớ hơn so với các ứng dụng đáng tin cậy. Bằng cách tuân thủ phương pháp này, bộ nhớ có thể được tiết kiệm đến một mức độ từ việc cạn kiệt.

## Xử lý bộ nhớ bằng cách sử dụng Logs

Kỹ thuật này đề xuất sử dụng các logs được tạo ra để xác định tương tác của người dùng và thời gian mà họ sử dụng các ứng dụng để động động thay đổi ưu tiên của các ứng dụng. Điều này sẽ cho phép giữ các ứng dụng ưu tiên cao trong bộ nhớ cache và chỉ kết thúc các ứng dụng ưu tiên thấp khi có vấn đề về bộ nhớ thấp.

## Phát hiện Anti-Patterns

Trong việc phát triển ứng dụng một cách nhanh chóng, các nhà phát triển hiện nay thường lạc quan khỏi các mô hình lập trình được biết đến là các mẫu chống, dẫn đến thiết kế kém và do đó hiệu suất kém của ứng dụng. Paprika là một công cụ được đề xuất để phân tích mã nguồn và xác định các mẫu chống, đồng thời đề xuất giải pháp để sửa chúng. OOP (Lập trình hướng đối tượng) là khối xây dựng cơ bản của việc phát triển bất kỳ loại ứng dụng nào, nó cung cấp khả năng tái sử dụng và các chức năng khác mà trước đây là không thể.

Các file apk của Android chứa file .dex chứa các lớp Java đã được biên dịch. Android chạy trên Máy ảo Darvik và bytecode của nó khác biệt so với Java. Có nhiều công cụ có sẵn để đảo ngược các file .dex.

Paprika trước tiên trích xuất siêu dữ liệu từ apk như tên ứng dụng, định danh gói và đánh giá của người dùng, sau đó các đối tượng mã nguồn như lớp, phương thức và tên biến cũng được trích xuất. Bằng cách sử dụng các đối tượng đã được trích xuất, một mô hình mã nguồn được tính toán dưới dạng đồ thị với các giá trị raw. Mô hình này được lưu trữ trong một cơ sở dữ liệu đồ thị và sau đó cơ sở dữ liệu này được truy vấn để phát hiện các mẫu chống. Những mẫu chống này khi được sửa chữa có thể giải phóng một lượng lớn bộ nhớ.

## Giảm thiểu quá trình lão hóa phần mềm

Lão hóa phần mềm là quá trình mà hiệu suất của hệ điều hành và ứng dụng giảm đi theo thời gian. Vấn đề chủ yếu là rò rỉ bộ nhớ. Khi lão hóa, bộ nhớ trống giảm đi nên ít ứng dụng được lưu trữ và khi ứng dụng mới được khởi chạy, bộ nhớ cần được giải phóng sẽ tốn CPU và pin. Để phát hiện và xác định lão hóa, cần giữ các thống kê sử dụng tài nguyên. Để điều tra rò rỉ bộ nhớ, thiết bị cần được kiểm tra dưới điều kiện nghiêm trọng nơi nó có nguy cơ gặp sự cố. Đối với điều này, Exerciser Monkey được sử dụng bởi các nhà nghiên cứu, mô phỏng sự kiện chạm, nhấp chuột và các sự kiện Android phổ biến khác. Các bài kiểm tra được chia thành các bài kiểm tra phụ, với mỗi bài kiểm tra phụ phụ thuộc vào kết quả của bài kiểm tra trước đó. Nếu phát hiện lão hóa, bài kiểm tra tiếp theo sẽ được thực hiện trong thời gian dài hơn.

Những bài kiểm tra này giúp xác định phạm vi của rò rỉ bộ nhớ trong các ứng dụng khác nhau.

## Non-Blocking Garbage Collector

Bộ thu gom rác (Garbage Collector) hoạt động dựa trên nguyên tắc "ngừng thực thi hiện tại, đánh dấu các đối tượng để xóa và làm sạch chúng khỏi bộ nhớ," điều này có thể dẫn đến các hành vi không phản ứng và thậm chí là sự cố. Để khắc phục những vấn đề này, các nhà nghiên cứu đã đề xuất một Bộ thu gom rác thời gian thực có hai đặc điểm sau đây:

1. GC này sẽ hoạt động một cách tăng dần với giai đoạn chặn ngắn.
2. Tốc độ của GC phải phù hợp với rác được tạo ra bởi hệ điều hành để tránh tình trạng Hết Bộ Nhớ.

Điều này sẽ cho phép giải phóng bộ nhớ nhanh chóng so với Bộ thu gom rác không chặn và sẽ cải thiện hiệu suất tổng thể của hệ điều hành Android.

## Sử dụng bộ nhớ Non-Volatile

Theo một nghiên cứu, hầu hết người dùng có các phiên ứng dụng ít hơn 10 giây và, trong trường hợp như vậy, việc giảm thời gian tải ứng dụng là cần thiết. Nhiều giải pháp đã được đề xuất để cải thiện thời gian khởi chạy ứng dụng nhưng hầu hết đều không phải là dựa trên phần cứng. Giải pháp được đề xuất bởi các nghiên cứu là sử dụng Bộ Nhớ Không Biến Động (NVM), cụ thể là Bộ Nhớ Thay Đổi Pha (PCM) như là bản sao lưu của bộ nhớ chính. Bộ Nhớ NVM phổ biến vì nó tiêu thụ ít năng lượng. PCM nhanh, rất hiệu quả năng lượng cho các hoạt động đọc nhưng tiêu thụ nhiều năng lượng trong các hoạt động ghi và rất chậm. Do đó, một giải pháp kết hợp DRAM-PCM được đề xuất để cải thiện thời gian khởi chạy ứng dụng.

Một nghiên cứu đã được tiến hành để phân tích ứng dụng dựa trên bộ nhớ và phát hiện rằng ứng dụng thuộc hai loại hạng mục:

Ứng dụng ổn định, nơi bộ nhớ tăng lên ban đầu trong 10 giây đầu tiên rồi ổn định.

Ứng dụng không ổn định, nơi bộ nhớ tiếp tục tăng lên theo thời gian và không bao giờ ổn định.

Giữ cả hai loại ứng dụng này trong tâm trí, giải pháp đề xuất là sử dụng NVM như bản sao lưu của bộ nhớ chính. Khi Bộ thu gom rác (GC) bắt đầu và loại bỏ dữ liệu khỏi bộ nhớ chính, dữ liệu được chuyển giao sang NVM. Các khu vực cụ thể được chỉ định trong NVM để lưu trữ dữ liệu từ các ứng dụng sử dụng thường xuyên và thư viện chia sẻ phổ biến. Phương pháp này nhằm mục đích cải thiện thời gian khởi chạy, ngay cả khi các ứng dụng không được lưu trữ trong bộ nhớ chính, nhưng chúng sẽ có mặt trong bộ nhớ dự phòng.

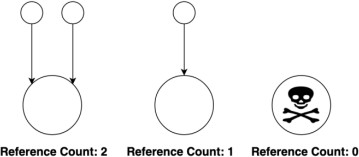
Trong khi những giải pháp này tập trung vào việc tối ưu hóa các hệ thống bộ nhớ và quản lý bộ nhớ hiện có trong Android, không có giải pháp nào cung cấp một giải pháp toàn diện và rõ ràng để giải quyết những vấn đề này.

# CHƯƠNG III: GIẢI PHÁP TỐI ƯU HÓA BỘ NHỚ

## Đề xuất giải pháp

Đề xuất một phương pháp hoàn toàn khác nhau để thay thế hệ thống Garbage Collector của Android bằng hệ thống Automatic Reference Counting (ARC) của iOS.

Đề xuất một phương pháp hoàn toàn khác để thay thế hệ thống Garbage Collector của Android bằng hệ thống Automatic Reference Counting (ARC) của iOS. Trong hệ thống ARC, đối tượng sẽ tự động giải phóng bộ nhớ khi không còn đối tượng nào giữ tham chiếu đến nó, và do đó, giải phóng bộ nhớ một cách tự động. Cơ chế này của ARC được minh họa trong hình 1.



Hình 1: Cách hoạt động của ARC

Sự khác biệt lớn nhất giữa ARC và Garbage Collector là ARC giải phóng bộ nhớ ngay lập tức khi đối tượng được giải phóng. Trong khi đó, Garbage Collector đánh dấu các đối tượng đã giải phóng và làm sạch bộ nhớ sau các khoảng thời gian đều đặn hoặc khi hệ điều hành cạn kiệt bộ nhớ.



Hình 2: Android ImageView loading

Giờ hãy thảo luận về một số ví dụ mã nguồn về quản lý bộ nhớ trong Garbage Collector và ARC để làm rõ sự khác biệt giữa hai phương pháp này. Đối với các ví dụ mã nguồn, ARC được sử dụng trong ngôn ngữ Swift trên Xcode.



Hình 3: iOS ImageView loading

IDE và Bộ Thu Gom Rác được sử dụng trong Ngôn Ngữ Java trên Android Studio IDE. Trước hết, cả trong iOS (ARC) và Android (Bộ Thu Gom Rác), hãy gán tham chiếu hình ảnh cho đối tượng ImageView được tạo trong các tệp tài nguyên giao diện người dùng tương ứng của họ trong các hình 2 và 3.

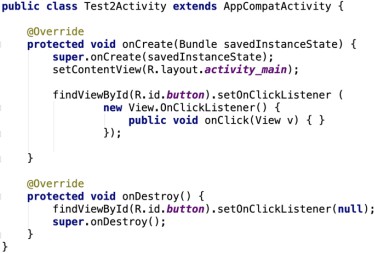
Trong hình 2 và 3, các đối tượng ImageView với một số hình ảnh thử nghiệm trong cả hai hệ thống bộ nhớ được khởi tạo. Android tái sử dụng tham chiếu imageView của imageResource khi lớp cha onDestroy được gọi và cũng đặt tham chiếu imageView về null để tránh gây ra rò rỉ bộ nhớ. Điều này có nghĩa là ngay cả khi bạn không tạo đối tượng tham chiếu một cách động, bạn vẫn cần gán giá trị null để đánh dấu nó có thể gỡ bỏ được cho Bộ Thu Gom Rác. Trong khi đó, iOS không cần thực hiện bất kỳ điều gì tương tự.

Khi lớp cha bị hủy, referenceCount của đối tượng myImageView giảm và đến khi nó đạt đến số không, nó sẽ bị hủy và bộ nhớ sẽ được giải phóng tự động. Ở đây, một ví dụ khác được hiển thị trong hình 4.



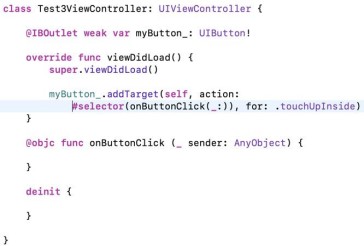
Hình 4: Tạo động một Đối tượng tham chiếu trong iOS

Trong Hình 4, một đối tượng tham chiếu (secondImageView kiểu UIImageView) được tạo động và thêm vào view cha. Khi view cha bị hủy, referenceCount của secondImageView trở thành số không và tự động bị hủy. Điều này cho thấy rằng ngay cả đối với các đối tượng được tạo động, không cần phải gán tham chiếu null khi view cha bị giải phóng. Điều này mang lại hiệu suất lớn so với Bộ Thu Gom Rác, đối với nó, bạn cần gán null cho tham chiếu để đánh dấu chúng để được thu gom rác.



Hình 5: Android Button Registering and Unregistering event

Bây giờ, hãy xem xét một ví dụ phổ biến khác về việc đăng ký sự kiện cho nút trong cả hai mô hình bộ nhớ, như được thể hiện trong các hình 5 và 6.



Hình 6: iOS Button Registering event

Trong trường hợp của Android (hình 5), sau khi đăng ký sự kiện nhấn nút, việc hủy đăng ký là bắt buộc, nếu không sẽ gây rò rỉ bộ nhớ. Tuy nhiên, không cần điều này trong iOS. Trong iOS (hình 6), khi lớp cha bị hủy, referenceCount của đối tượng myButton\_ giảm và nó đạt đến số không, do đó nó bị hủy và bộ nhớ được giải phóng tự động.

Sau khi đi qua các ví dụ, rõ ràng rằng Garbage Collector có khả năng gây rò rỉ bộ nhớ hơn trong khi ARC xử lý bộ nhớ một cách tự động và hiệu quả hơn. Tóm tắt về các ưu điểm chính của ARC so với Garbage Collection như sau:

1. ARC giải phóng bộ nhớ ngay lập tức khi đối tượng bị hủy, khác với phương pháp quét và đánh dấu của Garbage Collector.
2. Không cần phải đánh dấu biến là null hoặc tái sử dụng, và sự kiện nhấn không cần được loại bỏ để tránh rò rỉ bộ nhớ.
3. Đối tượng được tự động hủy khi ra khỏi phạm vi.

Chính vì những sự thật này mà các thiết bị iOS hoạt động hiệu quả hơn so với các đối tác Android của chúng và vì lý do này, các thiết bị iOS sở hữu khoảng 1/3 bộ nhớ ít hơn so với một thiết bị Android có hiệu suất tương tự.

Chỉ số AI Jaccard là một công cụ quan trọng trong lĩnh vực Xử lý ngôn ngữ tự nhiên (NLP). Nó đóng một vai trò quan trọng trong việc tìm hiểu sự tương đồng và khác biệt giữa hai bộ dữ liệu. Chỉ số này, được đặt theo tên của nhà toán học người Pháp Paul Jaccard, đo lường sự giống nhau giữa hai tập hợp bằng cách tính tỷ số giữa giao điểm của các tập hợp và hợp của các tập hợp.

**Phân loại và truy xuất**

Trong NLP, Chỉ số Jaccard AI đặc biệt hữu ích trong các tác vụ như phân loại văn bản, truy xuất thông tin và phân tích độ tương tự của tài liệu. Bằng cách so sánh sự giống nhau giữa các văn bản hoặc tài liệu khác nhau, thuật toán NLP có thể phân loại và truy xuất thông tin liên quan một cách hiệu quả.

Để hiểu cách hoạt động của Chỉ số AI Jaccard, hãy xem xét một ví dụ. Giả sử chúng ta có hai bộ từ: Bộ A chứa các từ “táo”, “chuối” và “cam”, trong khi Bộ B chứa các từ “chuối”, “nho” và “kiwi”. Để tính Chỉ số Jaccard, chúng ta tìm giao điểm của các bộ (trong trường hợp này là từ “chuối”) và chia nó cho hợp của các bộ (bao gồm tất cả các từ duy nhất từ cả hai bộ). Trong ví dụ này, Chỉ số Jaccard sẽ là 1/5 hoặc 0,2.