Chuyên môn hóa loại Just-in-time dựa trên dấu vết cho động Ngôn ngữ

Andreas Gal □ +, Brendan Eich □ , Mike Shaver □ , David Anderson □ , David Mandelin □ ,
Mohammad R. Haghighat $, Blake Kaplan □ , Graydon Hoare □ , Boris Zbarsky □ , Jason Orendorff □ ,
Jesse Ruderman □ , Edwin Smith #, Rick Reitmaier #, Michael Bebenita +, Mason Chang + #, Michael Franz +

Tổng công ty Mozilla □
{gal, brendan, shaver, danderson, dmandelin, mrbkap, graydon, bz, jorendorff, jruderman} @ mozilla.com

Adobe Corporation #
{edwsmith, rreitmai} @ adobe.com

Tập đoàn Intel $
{mohammad.r.haghighat}@intel.com

Đại học California, Irvine +
{mbebenit, changm, franz} @ uci.edu

trừu tượng

Các ngôn ngữ động như JavaScript khó nhập hơn so với các ngôn ngữ được nhập tĩnh. Vì không có thông tin loại cụ thể có sẵn, các trình biên dịch truyền thống cần phát ra mã chung có thể xử lý tất cả các kết hợp kiểu có thể có trong thời gian chạy. Chúng tôi trình bày một kỹ thuật biên dịch al ternative cho các ngôn ngữ được nhập động xác định các dấu vết vòng lặp được thực thi thường xuyên tại thời điểm chạy và sau đó tạo mã máy một cách nhanh chóng chuyên dành cho các loại động ac tual xảy ra trên mỗi đường dẫn qua vòng lặp. Của chúng tôi phương pháp cung cấp sự chuyên môn hóa kiểu liên thủ tục rẻ và một cách thanh lịch và hiệu quả để biên dịch từng bước một cách lười biếng khám phá các đường dẫn thay thế được yêu thích thông qua các vòng lặp lồng nhau. Chúng tôi đã thực hiện một trình biên dịch động cho JavaScript dựa trên kỹ thuật của chúng tôi và chúng tôi đã đo tốc độ gấp 10 lần trở lên cho một số điểm chuẩn nhất định các chương trình.

Bộ mô tả danh mục và chủ đề D.3.4 [Lập trình Lan guages]: Bộ xử lý - Bộ biên dịch tăng dần, tạo mã.

Thuật ngữ chung Thiết kế, Thử nghiệm, Đo lường, Perfor mance.

Từ khóa JavaScript, biên dịch đúng lúc, cây dấu vết.

1. Giới thiệu

Các ngôn ngữ động như JavaScript, Python và Ruby, là những ngôn ngữ nổi bật vì chúng rất dễ hiểu, dễ tiếp cận đối với những người không phải là chuyên gia và làm cho triển khai dễ dàng như phân phối tệp nguồn. Chúng được sử dụng cho các tập lệnh nhỏ cũng như cho các ứng dụng phức tạp. JavaScript, dành cho ví dụ, là tiêu chuẩn thực tế cho lập trình web phía máy khách và được sử dụng cho logic ứng dụng về năng suất dựa trên trình duyệt các ứng dụng như Google Mail, Google Docs và Zimbra Col labration Suite. Trong miền này, để cung cấp cho người dùng linh hoạt trải nghiệm và kích hoạt thế hệ ứng dụng mới, điện thoại ảo phải cung cấp thời gian khởi động thấp và hiệu suất cao.

Các trình biên dịch cho các ngôn ngữ được nhập kiểu tĩnh dựa vào thông tin kiểu để tạo ra mã máy hiệu quả. Trong một ngôn ngữ về đồ thị chuyên nghiệp được nhập động, chẳng hạn như JavaScript, các loại biểu thức có thể thay đổi trong thời gian chạy. Điều này có nghĩa là trình biên dịch không còn có thể dễ dàng chuyển đổi các hoạt động thành các lệnh máy hoạt động trên một loại cụ thể. Không có thông tin loại chính xác, trình biên dịch phải phát ra mã máy tổng quát chậm hơn để có thể đối phó với tất cả các tổ hợp kiểu tiềm năng. Mặc dù suy ra kiểu tĩnh thời gian biên dịch có thể thu thập thông tin kiểu để tạo mã máy tối ưu hóa, nhưng phân tích tĩnh truyền thống rất tốn kém và do đó không phù hợp với môi trường tương tác cao của một trình duyệt web.

Chúng tôi trình bày một kỹ thuật biên dịch dựa trên dấu vết cho động các ngôn ngữ điều hòa tốc độ biên dịch với mỗi hình thức xuất sắc của mã máy được tạo. Hệ thống của chúng tôi sử dụng phương pháp thực thi chế độ hỗn hợp: hệ thống bắt đầu chạy JavaScript trong trình thông dịch bytecode khởi động nhanh. Khi chương trình chạy, hệ thống xác định các chuỗi bytecode nóng (thường xuyên được thực thi), các bản ghi và biên dịch chúng thành mã gốc nhanh. Chúng tôi gọi một chuỗi các hướng dẫn như vậy là một dấu vết.

Không giống như các trình biên dịch động dựa trên phương pháp, trình biên dịch com động của chúng tôi hoạt động ở mức độ chi tiết của các vòng riêng lẻ. Thiết kế này sự lựa chọn dựa trên kỳ vọng rằng các chương trình dành phần lớn thời gian của họ trong các vòng lặp nóng. Ngay cả trong các ngôn ngữ được nhập động, chúng tôi mong đợi các vòng lặp nóng chủ yếu là loại ổn định, có nghĩa là các loại giá trị là bất biến. (12) Ví dụ, chúng tôi mong đợi các coun vòng lặp bắt đầu dưới dạng số nguyên vẫn là số nguyên cho tất cả các lần lặp. Khi nào cả hai kỳ vọng này đều giữ nguyên, một trình biên dịch dựa trên dấu vết có thể bao gồm thực thi chương trình với một số lượng nhỏ các dấu vết được biên dịch hiệu quả, chuyên biệt về kiểu.

Mỗi dấu vết đã biên dịch bao gồm một đường dẫn thông qua chương trình với một ánh xạ các giá trị thành các loại. Khi máy ảo thực thi một biên dịch theo dõi, nó không thể đảm bảo rằng con đường tương tự sẽ được theo sau hoặc các kiểu tương tự sẽ xảy ra trong các lần lặp vòng lặp tiếp theo.

PLDI'09, ngày 15-20 tháng 6 năm 2009, Dublin, Ireland.

Do đó, việc ghi lại và biên dịch một dấu vết suy đoán rằng đư ờng dẫn và cách nhập sẽ chính xác như trong quá trình ghi cho các lần lặp tiếp theo của vòng lặp.

Mọi dấu vết đư ợc biên dịch đều chứa tất cả các biện pháp bảo vệ (kiểm tra) cần thiết để xác thực suy đoán. Nếu một trong các biện pháp bảo vệ không thành công (nếu luồng điều khiển khác hoặc giá trị của một kiểu khác đư ợc tạo ra), dấu vết sẽ thoát ra. Nếu một lối ra trở nên nóng, máy ảo có thể ghi lại một dấu vết nhánh bắt đầu từ lối ra để che đi đư ờng dẫn mới. Bằng cách này, VM ghi lại một cây theo dõi bao gồm tất cả các đư ờng dẫn nóng thông qua vòng lặp.

Các vòng lặp lồng nhau có thể khó tối ư u hóa cho việc theo dõi các máy ảo. Trong quá trình triển khai gần đây, các vòng bên trong sẽ trở nên nóng trư ớc tiên và máy ảo sẽ bắt đầu theo dõi ở đó. Khi vòng lặp bên trong thoát ra, máy ảo sẽ phát hiện ra rằng một nhánh khác đã đư ợc sử dụng. Máy ảo sẽ cố gắng ghi lại một dấu vết nhánh và nhận thấy rằng dấu vết không đến đư ợc tiêu đề vòng lặp bên trong mà là tiêu đề vòng lặp bên ngoài. Tại thời điểm này, VM có thể tiếp tục theo dõi cho đến khi nó tiếp cận tiêu đề vòng lặp bên trong một lần nữa, do đó truy tìm vòng lặp bên ngoài bên trong cây theo dõi cho vòng lặp bên trong. Như ng điều này đòi hỏi phải truy tìm một bản sao của vòng lặp bên ngoài cho mọi lối ra bên và kết hợp loại trong vòng lặp bên trong. Về bản chất, đây là một dạng sao chép đuôi ngoài ý muốn, dễ làm tràn bộ nhớ đệm mã. Ngoài ra, máy ảo có thể đơn giản ngừng theo dõi và từ bỏ việc theo dõi các vòng bên ngoài.

Chúng tôi giải quyết vấn đề vòng lặp lồng nhau bằng cách ghi lại các cây dấu vết lồng nhau. Hệ thống của chúng tôi theo dõi vòng lặp bên trong chính xác như phiên bản ban đầu. Hệ thống ngừng mở rộng cây bên trong khi nó đạt đến vòng ngoài, như ng sau đó nó bắt đầu một dấu vết mới ở tiêu đề vòng ngoài. Khi vòng lặp bên ngoài đến tiêu đề vòng lặp bên trong, hệ thống sẽ cố gắng gọi cây theo dõi cho vòng lặp bên trong. Nếu lệnh gọi thành công, VM ghi lại lệnh gọi đến cây bên trong như một phần của dấu vết bên ngoài và kết thúc dấu vết bên ngoài như bình thư ờng. Bằng cách này, hệ thống của chúng tôi có thể theo dõi bất kỳ số lư ợng vòng lặp nào đư ợc lồng vào bất kỳ độ sâu nào mà không gây ra trùng lặp đuôi quá mức.

Các kỹ thuật này cho phép một máy ảo dịch động một gam chuyên nghiệp sang các cây dấu vết chuyên biệt, lồng nhau. Bởi vì các dấu vết có thể vư ợt qua ranh giới gọi hàm, các kỹ thuật của chúng tôi cũng đạt đư ợc các giới hạn riêng của nội tuyến. Bởi vì các dấu vết không có tham gia luồng kiểm soát nội bộ, chúng có thể đư ợc tối ư u hóa theo thời gian tuyến tính bằng một trình biên dịch đơn giản (10). Do đó, máy ảo theo dõi của chúng tôi thực hiện một cách hiệu quả cùng một loại op timizations yêu cầu phân tích liên thủ tục trong một cài đặt tối ư u hóa tĩnh. Điều này làm cho việc theo dõi trở thành một công cụ hấp dẫn và hiệu quả để nhập mã lệnh gọi hàm nhiều hàm phức tạp, thậm chí phức tạp.

Chúng tôi đã triển khai các kỹ thuật này cho JavaScript hiện có trong trình thông dịch, SpiderMonkey. Chúng tôi gọi kết quả là VM Trace Monkey theo dõi. TraceMonkey hỗ trợ tất cả các tính năng JavaScript của Spi derMonkey, với tốc độ tăng gấp 2 lần-20 lần cho các chư ơng trình có thể theo dõi.

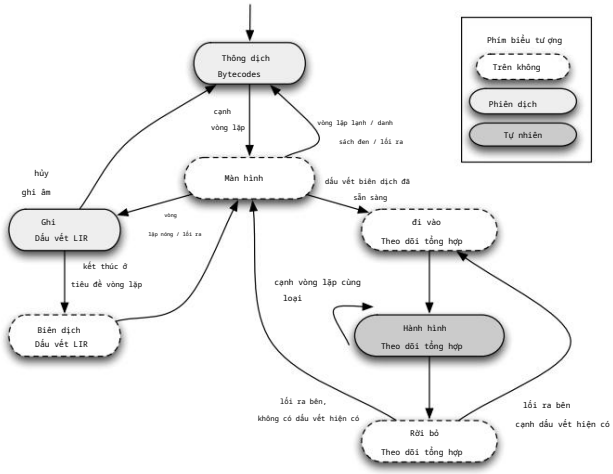
Bài báo này có những đóng góp sau:

- Chúng tôi giải thích một thuật toán để tạo động các cây dấu vết để bao phủ một chư ơng trình, biểu diễn các vòng lặp lồng nhau dư ới dạng cây dấu vết lồng nhau.
- Chúng tôi giải thích cách tạo một cách suy đoán mã chuyên biệt hiệu quả cho các dấu vết từ các chư ơng trình ngôn ngữ động.
- Chúng tôi xác thực các kỹ thuật theo dõi của mình trong một triển khai dựa trên trình thông dịch JavaScript SpiderMonkey, đạt đư ợc tốc độ gấp 2 lần-20 lần trên nhiều chư ơng trình.

Phần còn lại của bài viết này đư ợc tổ chức như sau. Phần 3 là tổng quan chung về biên dịch dựa trên cây dấu vết mà chúng tôi sử dụng để giới hạn và biên dịch các vùng mã đư ợc thực thi thư ờng xuyên. Trong Phần 4, chúng tôi mô tả cách tiếp cận của chúng tôi để bao phủ các vòng lồng nhau bằng cách sử dụng một số cây dấu vết riêng lẻ. Trong Phần 5, chúng tôi mô tả phư ơng pháp tiếp cận chuyên môn hóa kiểu suy đoán dựa trên biên dịch theo dõi mà chúng tôi sử dụng để tạo mã máy hiệu quả từ các dấu vết bytecode đư ợc ghi lại.
Việc triển khai trình biên dịch chuyên về kiểu động cho JavaScript của chúng tôi đư ợc mô tả trong Phần 6. Công việc liên quan đư ợc thảo luận trong Phần 8. Trong Phần 7, chúng tôi đánh giá trình biên dịch động của chúng tôi dựa trên một tập hợp các điểm chuẩn của ngành. Bài báo kết thúc với kết luận trong Phần 9 và triển vọng về công việc trong tư ơng lai đư ợc trình bày trong Phần 10.

```
1 for (var i = 2; i <100; ++ i) {2 if (! Primes [i])
3 continue; 4 for (var k = i + i; i <100; k + = i)
số nguyên tố [k] = false; 5 6}
```

Hình 1. Chư ơng trình mẫu: sàng của Eratosthenes. số nguyên tố đư ợc khởi tạo thành một mảng gồm 100 giá trị sai khi nhập vào đoạn mã này.



Hình 2. Máy trạng thái mô tả các hoạt động chính của Trace Monkey và các điều kiện gây ra chuyển đổi sang hoạt động mới. Trong hộp tối, TM thực thi JS dư ới dạng dấu vết đã biên dịch. Trong các hộp màu xám nhạt, TM thực thi JS trong trình thông dịch tiêu chuẩn. Hộp màu trắng ở trên cao. Do đó, để tối đa hóa hiệu suất, chúng ta cần tối đa hóa thời gian ở ô tối nhất và giảm thiểu thời gian ở ô trắng. Trư ờng hợp tốt nhất là một vòng lặp trong đó các kiểu ở cạnh vòng lặp giống với các kiểu trên mục nhập – sau đó TM có thể ở trong mã gốc cho đến khi hoàn tất vòng lặp.

2. Tổng quan: Chạy theo dõi ví dụ Phần này cung

cấp tổng quan về hệ thống của chúng tôi bằng cách mô tả cách TraceMonkey thực thi một chư ơng trình ví dụ. Chư ơng trình ví dụ, đư ợc hiển thị trong Hình 1, tính toán 100 số nguyên tố đầu tiên với các vòng lặp lồng nhau. Phần tư ờng thuật nên đư ợc đọc cùng với Hình 2, mô tả các hoạt động mà TraceMonkey thực hiện và khi nó chuyển đổi giữa các vòng lặp.

TraceMonkey luôn bắt đầu thực hiện một chư ơng trình trong trình thông dịch mã byte. Mỗi cạnh trở lại của vòng lặp là một điểm theo dõi tiềm năng.
Khi trình thông dịch vư ợt qua một cạnh của vòng lặp, TraceMonkey sẽ gọi trình giám sát theo dõi, có thể quyết định ghi lại hoặc thực thi một dấu vết gốc. Khi bắt đầu thực thi, vẫn chư a có dấu vết đã biên dịch nào, vì vậy trình theo dõi theo dõi đếm số lần mỗi cạnh quay lại của vòng lặp đư ợc thực hiện cho đến khi một vòng lặp trở nên nóng, hiện tại sau 2 lần giao nhau. Lư u ý rằng cách các vòng lặp của chúng ta đư ợc biên dịch, cạnh của vòng lặp đư ợc vư ợt qua trư ớc khi vào vòng lặp, vì vậy việc vư ợt qua lần thứ hai xảy ra ngay sau lần lặp đầu tiên.

Đây là chuỗi sự kiện đư ợc chia nhỏ bằng cách lặp lại vòng lặp bên ngoài:

```
v0: = ld trạng thái [748]            // tải các số nguyên tố từ bản ghi kích hoạt theo dõi
         st sp [0], v0 v1:           // lưu trữ các số nguyên tố vào ngăn xếp trình thông dịch
= ld state [764] v2: = i2f           // tải k từ bản ghi kích hoạt theo dõi
(v1) st sp [8], v1 st sp             // chuyển đổi k từ int thành double
         [16], 0 v3: = ld v0         // lưu trữ k vào ngăn xếp trình thông dịch
         [4] v4: = and v3,           // lưu trữ false vào ngăn xếp trình thông dịch
-4 v5: = eq v4, Mảng                 // tải lớp từ cho số nguyên tố
                                     // che dấu lớp đối tượng thẻ cho các số nguyên tố
                                     // kiểm tra xem số nguyên tố có phải là một mảng hay không
         xf v5                       // lối ra bên nếu v5 là sai
v6: = js_Array_set (v0, v2, false) // gọi hàm để thiết lập phần tử mảng
v7: = eq v6, 0 // lối                // kiểm tra giá trị trả về từ cuộc gọi
         xt v7       ra bên nếu js_Array_set trả về false.
```

Hình 3. Đoạn mã LIR cho chương trình mẫu. Đây là LIR được ghi lại cho dòng 5 của chương trình mẫu trong Hình 1. Mã hóa LIR ngữ nghĩa ở dạng SSA bằng cách sử dụng các biến tạm thời. LIR cũng mã hóa tất cả các lưu trữ mà trình thông dịch sẽ thực hiện đối với ngăn xếp dữ liệu của nó. Đôi khi các cửa hàng này có thể được tối ưu hóa đi vì các vị trí ngăn xếp chỉ tồn tại trên các lối ra cho trình thông dịch. Cuối cùng, LIR ghi lại những người bảo vệ và các lối ra bên cạnh để xác minh các giả định được thực hiện trong bản ghi này: rằng số nguyên tố là một mảng và lệnh gọi thiết lập phần tử của nó thành công.

```
mov edx, ebx (748) mov               // tải các số nguyên tố từ bản ghi kích hoạt theo dõi
edi (0), edx mov esi, ebx            // (*) lưu trữ các số nguyên tố vào ngăn xếp trình thông dịch
(764) mov edi (8), esi               // tải k từ bản ghi kích hoạt theo dõi
mov edi (16), 0 mov eax,             // (*) lưu trữ k vào ngăn xếp trình thông dịch
edx (4) và eax, -4 cmp               // (*) lưu trữ false vào ngăn xếp trình thông dịch
eax, Array jne side_exit_1           // (*) tải từ lớp đối tượng cho các số nguyên tố
sub esp, 8 push false                // (*) che dấu lớp đối tượng cho các số nguyên tố
push esi call js_Array_set           // (*) kiểm tra xem số nguyên tố có phải là một mảng hay không
add esp, 8 mov ecx, eax              // (*) lối ra bên nếu số nguyên tố không phải là một mảng
test eax, eax je side_exit_2         // ngăn xếp ngăn xếp cho quy ước căn chỉnh cuộc gọi
                                     // đẩy đối số cuối cùng cho cuộc gọi
                                     // đẩy đối số đầu tiên cho cuộc gọi
                                     // gọi hàm để thiết lập phần tử mảng
                                     // dọn dẹp không gian ngăn xếp thừa
                                     // (*) được tạo bởi trình cấp phát thanh ghi
                                     // (*) giá trị trả về kiểm tra của js_Array_set
                                     // thoát bên (*) nếu cuộc gọi không thành công
...
side_exit_1:
mov ecx, ebp (-4) mov                // khôi phục lại ecx
esp, ebp jmp epilog                  // khôi phục lại esp
                                     // chuyển đến câu lệnh ret
```

Hình 4. Đoạn mã x86 cho chương trình mẫu. Đây là mã x86 được biên dịch từ đoạn mã LIR trong Hình 3. Hầu hết các lệnh biên dịch LIR cho một lệnh x86 duy nhất. Các hướng dẫn được đánh dấu bằng (*) sẽ bị bỏ qua bởi một trình biên dịch lý tưởng hóa biết rằng không bên nào thoát sẽ bao giờ được thực hiện. 17 hướng dẫn do trình biên dịch tạo ra so sánh thuận lợi với hơn 100 hướng dẫn mà trình thông dịch sẽ thực thi cho cùng một đoạn mã, bao gồm 4 bước nhảy gián tiếp.

i = 2. Đây là lần lặp đầu tiên của vòng lặp ngoài. Vòng lặp trên dòng 4-5 trở nên nóng trong lần lặp thứ hai, vì vậy TraceMonkey sử dụng chế độ ghi trên dòng 4. Ở chế độ ghi, TraceMonkey ghi lại mã dọc theo dấu vết trong một biểu diễn ate trung gian trình biên dịch cấp thấp mà chúng tôi gọi là LIR. Dấu vết LIR mã hóa tất cả các hoạt động được thực hiện và các loại của tất cả các toán hạng. Dấu vết LIR cũng mã hóa bảo vệ, là các kiểm tra xác minh rằng luồng điều khiển và các loại giống với các loại được quan sát thấy trong quá trình ghi lại dấu vết. Do đó, trong các lần thực thi sau này, nếu và chỉ khi tất cả các lính canh được thông qua, dấu vết có ngữ nghĩa chương trình bắt buộc.

TraceMonkey dừng ghi khi quá trình thực thi quay trở lại tiêu đề vòng lặp hoặc thoát khỏi vòng lặp. Trong trường hợp này, việc thực thi trả về tiêu đề vòng lặp trên dòng 4.

Sau khi ghi xong, TraceMonkey biên dịch dấu vết thành mã gốc sử dụng thông tin loại được ghi lại để tối ưu hóa. Kết quả là một đoạn mã gốc có thể được nhập nếu PC thông dịch và các loại giá trị khớp với những giá trị được quan sát khi ghi dấu vết đã được bắt đầu. Dấu vết đầu tiên trong ví dụ của chúng tôi, T45, bao gồm các dòng 4 và 5. Dấu vết này có thể được nhập nếu PC ở dòng 4, i và k là các số nguyên và các số nguyên tố là một đối tượng. Sau khi biên dịch T45, TraceMonkey quay trở lại trình thông dịch và lặp lại dòng 1.

i = 3. Bây giờ tiêu đề vòng lặp tại dòng 1 đã trở nên nóng, vì vậy Trace Monkey bắt đầu ghi. Khi ghi đến dòng 4, Trace Monkey quan sát thấy rằng nó đã đạt đến tiêu đề vòng lặp bên trong mà nó đã sẵn sàng có một dấu vết đã biên dịch, vì vậy TraceMonkey cố gắng lồng vòng lặp bên trong bên trong dấu vết hiện tại. Bước đầu tiên là gọi nội theo dõi như một chương trình con. Điều này thực hiện vòng lặp trên dòng 4 để hoàn thành và sau đó quay trở lại máy ghi. TraceMonkey xác minh rằng cuộc gọi đã thành công và sau đó ghi lại lệnh gọi đến dấu vết bên trong như một phần của dấu vết hiện tại. Quá trình ghi tiếp tục cho đến khi quá trình thực thi đạt đến dòng 1, và tại thời điểm đó TraceMonkey kết thúc và biên dịch một dấu vết cho vòng ngoài, T16.

i = 4. Trong lần lặp này, TraceMonkey gọi T16. Vì i = 4 nên câu lệnh if trên dòng 2 được sử dụng. Nhánh này không được lấy theo dấu vết ban đầu, vì vậy điều này khiến T16 không có ngư ời bảo vệ và đi theo lối ra bên lề.
Lối ra chư a nóng, vì vậy TraceMonkey quay trở lại trình thông dịch, trình thông dịch thực thi câu lệnh continue. i = 5. TraceMonkey gọi T16, lần lư ợt gọi theo dõi lồng nhau T45. T16 lặp lại tiêu đề của chính nó, bắt đầu lần lặp tiếp theo mà không bao giờ quay lại màn hình. i = 6. Trong lần lặp lại này, lối ra bên trên dòng 2 đư ợc thực hiện lại. Lần này, lối ra bên trở nên nóng, do đó, một dấu vết T23,1 đư ợc ghi lại bao phủ dòng 3 và quay trở lại tiêu đề vòng lặp. Do đó, phần cuối của T23,1 nhảy trực tiếp đến phần đầu của T16. Lối ra bên đư ợc vá để trong các lần lặp lại trong tư ơng lai, nó sẽ chuyển trực tiếp đến T23,1.

Tại thời điểm này, TraceMonkey đã biên dịch đủ dấu vết để bao phủ toàn bộ cấu trúc vòng lặp lồng nhau, vì vậy phần còn lại của chư ơng trình chạy hoàn toàn dư ới dạng mã gốc.

3. Theo dõi cây

Trong phần này, chúng tôi mô tả dấu vết, cây dấu vết và cách chúng đư ợc hình thành tại thời điểm hoạt động. Mặc dù các kỹ thuật của chúng tôi áp dụng cho bất kỳ trình thông dịch ngôn ngữ động nào, chúng tôi sẽ mô tả chúng với giả định là trình thông dịch bytecode để giữ cho việc giải thích đơn giản.

3.1 Dấu vết

Dấu vết chỉ đơn giản là một đư ờng dẫn chư ơng trình, có thể vư ợt qua ranh giới lệnh gọi hàm. TraceMonkey tập trung vào các dấu vết của vòng lặp, bắt nguồn từ một cạnh của vòng lặp và đại diện cho một lần lặp duy nhất thông qua vòng lặp đư ợc liên kết.

Tư ơng tự như một khối cơ bản mở rộng, một dấu vết chỉ đư ợc nhập ở trên cùng, như ng có thể có nhiều lối ra. Ngư ợc lại với một khối cơ bản mở rộng, một dấu vết có thể chứa các nút tham gia. Tuy nhiên, vì một dấu vết luôn đi theo một con đư ờng duy nhất thông qua chư ơng trình gốc, các nút tham gia không thể nhận biết đư ợc như vậy trong một dấu vết và có một nút tiền nhiệm duy nhất giống như các nút thông thư ờng.

Dấu vết đư ợc đánh máy là dấu vết đư ợc chú thích với một kiểu cho mọi biến (bao gồm cả thời gian tạm thời) trên dấu vết. Một dấu vết đư ợc đánh máy cũng có một bản đồ kiểu mục nhập cung cấp các kiểu cần thiết cho các biến đư ợc sử dụng trên dấu vết trư ớc khi chúng đư ợc xác định. Ví dụ: một dấu vết có thể có một ánh xạ kiểu (x: int, b: boolean), nghĩa là dấu vết chỉ có thể đư ợc nhập vào nếu giá trị của biến x thuộc kiểu int và giá trị của b là kiểu boolean. Bản đồ loại mục nhập giống như chữ ký của một hàm.

Trong bài báo này, chúng tôi chỉ thảo luận về các dấu vết vòng lặp đư ợc đánh máy, và chúng tôi sẽ gọi chúng đơn giản là "dấu vết". Thuộc tính quan trọng của dấu vết vòng lặp đư ợc định kiểu là chúng có thể đư ợc biên dịch thành mã máy hiệu quả bằng cách sử dụng các kỹ thuật tư ơng tự đư ợc sử dụng cho các ngôn ngữ đư ợc định kiểu.

Trong TraceMonkey, dấu vết đư ợc ghi lại trong SSA LIR có hư ơng vị theo vết (đại diện trung gian cấp thấp). Trong SSA có hư ơng vị theo dõi (hoặc TSSA), các nút phi chỉ xuất hiện tại điểm vào, điểm này đạt đư ợc cả khi vào và qua các cạnh vòng lặp. Các nguyên thủy LIR quan trọng là các giá trị không đổi, tải và lư u trữ bộ nhớ (theo địa chỉ và độ lệch), toán tử số nguyên, toán tử dấu phẩy động, lời gọi hàm và các lần thoát có điều kiện. Các chuyển đổi kiểu, chẳng hạn như số nguyên thành nhân đôi, đư ợc biểu thị bằng các lệnh gọi hàm. Điều này làm cho LIR đư ợc TraceMonkey sử dụng độc lập với hệ thống kiểu cụ thể và các quy tắc chuyển đổi kiểu của ngôn ngữ nguồn. Các hoạt động LIR đủ chung để trình biên dịch phụ trợ là ngôn ngữ độc lập.

Hình 3 cho thấy một ví dụ về dấu vết LIR.

Trình thông dịch Bytecode thư ờng biểu diễn các giá trị trong một cấu trúc dữ liệu phức tạp khác nhau (ví dụ: bảng băm) ở định dạng đóng hộp (tức là với các bit thẻ loại đính kèm). Vì một dấu vết nhằm thể hiện mã hiệu quả giúp loại bỏ tất cả sự phức tạp đó, hoạt động theo dõi của chúng tôi đã dựa trên các giá trị không đư ợc đóng hộp trong các biến và mảng đơn giản càng nhiều càng tốt.

Một dấu vết ghi lại tất cả các giá trị trung gian của nó trong một vùng ghi kích hoạt nhỏ. Để thực hiện các truy cập biến nhanh khi theo dõi, dấu vết cũng nhập các biến cục bộ và toàn cục bằng cách mở hộp chúng và sao chép chúng vào bản ghi kích hoạt của nó. Do đó, dấu vết có thể đọc và ghi các biến này với các tải đơn giản và lư u trữ từ bản ghi kích hoạt gốc, độc lập với cơ chế quyền anh đư ợc trình thông dịch sử dụng. Khi dấu vết thoát ra, VM đóng hộp các giá trị từ vị trí lư u trữ gốc này và sao chép chúng trở lại trình thông dịch cấu trúc.

Đối với mọi nhánh luồng điều khiển trong chư ơng trình nguồn, bộ ghi tạo ra các lệnh LIR lối ra có điều kiện. Các chỉ dẫn này thoát ra khỏi dấu vết nếu luồng kiểm soát đư ợc yêu cầu khác với luồng điều khiển khi ghi dấu vết, đảm bảo rằng các hư ớng dẫn theo dõi chỉ đư ợc chạy nếu chúng đư ợc cho là như vậy. Chúng tôi gọi những hư ớng dẫn này là hư ớng dẫn bảo vệ.

Hầu hết các dấu vết của chúng tôi đại diện cho các vòng lặp và kết thúc bằng lệnh LIR vòng lặp đặc biệt. Đây chỉ là một nhánh vô điều kiện ở phía trên cùng của dấu vết. Những dấu vết như vậy chỉ trở lại thông qua lính canh.

Bây giờ, chúng tôi mô tả các tối ư u hóa chính đư ợc thực hiện như một phần của quá trình ghi LIR. Tất cả các tối ư u hóa này làm giảm các cấu trúc ngôn ngữ động phức tạp thành các cấu trúc đư ợc đánh máy đơn giản bằng cách phân tích giọng nói cho dấu vết hiện tại. Mỗi tối ư u hóa yêu cầu bảo vệ trong các cấu trúc để xác minh các giả định của họ về trạng thái và thoát khỏi dấu vết nếu cần thiết.

Loại chuyên môn hóa.

Tất cả các nguyên hàm của LIR đều áp dụng cho các toán hạng của các kiểu cụ thể. Do đó, các dấu vết LIR nhất thiết phải chuyên biệt về loại và một trình biên dịch có thể dễ dàng tạo ra một bản dịch mà không cần gửi loại. Một trình thông dịch bytecode điển hình mang các bit thẻ cùng với mỗi giá trị và để thực hiện bất kỳ thao tác nào, phải kiểm tra các bit thẻ, điều phối động, che dấu các bit thẻ để khôi phục giá trị không đư ợc gắn thẻ, thực hiện thao tác và sau đó áp dụng lại thẻ. LIR bỏ qua mọi thứ ngoại trừ chính hoạt động.

Một vấn đề tiềm ẩn là một số hoạt động có thể tạo ra các giá trị kiểu không thể đoán trư ớc. Ví dụ: đọc một thuộc tính từ một đối tư ợng có thể mang lại giá trị thuộc bất kỳ kiểu nào, không nhất thiết là kiểu đư ợc quan sát trong quá trình ghi. Máy ghi phát ra lệnh bảo vệ thoát có điều kiện nếu hoạt động mang lại giá trị thuộc loại khác với giá trị đư ợc thấy trong quá trình ghi. Các hư ớng dẫn bảo vệ này đảm bảo rằng miễn là việc thực thi đư ợc theo dõi, các loại giá trị khớp với các giá trị của dấu vết đã nhập. Khi máy ảo quan sát một lối ra bên dọc theo kiểu bảo vệ như vậy, một dấu vết đư ợc đánh mới sẽ đư ợc ghi lại bắt nguồn từ vị trí lối ra bên, ghi lại kiểu thao tác mới đư ợc đề cập.

Chuyên môn hóa biểu diễn: các đối tư ợng. Trong JavaScript, ngữ nghĩa tra cứu tên rất phức tạp và có khả năng đắt tiền vì chúng bao gồm các tính năng như kế thừa đối tư ợng và đánh giá. Để đánh giá một biểu thức đọc thuộc tính đối tư ợng như ox, trình thông dịch phải tìm kiếm bản đồ thuộc tính của o và tất cả các nguyên mẫu và cha mẹ của nó.
Bản đồ thuộc tính có thể đư ợc triển khai với các cấu trúc dữ liệu khác nhau (ví dụ: bảng băm cho mỗi đối tư ợng hoặc bảng băm dùng chung), do đó, quá trình tìm kiếm cũng phải điều chỉnh biểu diễn của từng đối tư ợng đư ợc tìm thấy trong quá trình tìm kiếm. TraceMonkey có thể chỉ cần quan sát kết quả của quá trình tìm kiếm và ghi lại LIR đơn giản nhất có thể để truy cập giá trị thuộc tính. Ví dụ: tìm kiếm có thể tìm thấy giá trị của ox trong nguyên mẫu của o, sử dụng tation đại diện bảng băm đư ợc chia sẻ đặt x vào vị trí 2 của vectơ thuộc tính. Sau đó, bản ghi có thể tạo ra LIR đọc ox chỉ với hai hoặc ba lần tải: một để lấy nguyên mẫu, có thể một để lấy vectơ giá trị thuộc tính và một tải nữa để lấy vị trí 2 từ vectơ. Đây là một sự đơn giản hóa và tăng tốc đáng kể so với mã thông dịch ban đầu. Mối quan hệ kế thừa và biểu diễn đối tư ợng có thể thay đổi trong quá trình thực thi, vì vậy mã đơn giản hóa yêu cầu các lệnh bảo vệ để đảm bảo biểu diễn đối tư ợng giống nhau. Trong TraceMonkey, đại diện của đối tư ợng

các lần gửi lại được gán một khóa số nguyên được gọi là hình dạng đối tượng. Vì vậy, bảo vệ là một kiểm tra bình đẳng đơn giản trên hình dạng đối tượng.

Chuyên môn hóa biểu diễn: số. JavaScript không có kiểu số nguyên, chỉ có kiểu Số là tập hợp các số con trỏ động 64-bit IEEE 754 (“nhân đôi”). Nhưng nhiều toán tử JavaScript, đặc biệt là truy cập mảng cụ thể và toán tử bitwise, thực sự hoạt động trên số nguyên, vì vậy trước tiên họ chuyển đổi số thành số nguyên, sau đó chuyển đổi bất kỳ kết quả số nguyên nào trở lại thành kép.1 Rõ ràng, một máy ảo JavaScript muốn nhanh phải tìm cách hoạt động trực tiếp trên số nguyên và tránh những chuyển đổi này.

Trong TraceMonkey, chúng tôi hỗ trợ hai biểu diễn cho số: số nguyên và số đôi. Trình thông dịch sử dụng các biểu diễn số nguyên nhiều nhất có thể, chuyển đổi cho các kết quả chỉ có thể được biểu diễn dưới dạng nhân đôi. Khi bắt đầu theo dõi, một số giá trị có thể được nhập và biểu diễn dưới dạng số nguyên. Một số hoạt động trên số nguyên yêu cầu bảo vệ. Ví dụ, thêm hai số nguyên có thể tạo ra một giá trị quá lớn cho biểu diễn số nguyên.

Hàm nội tuyến. Dấu vết LIR có thể vượt qua ranh giới chức năng theo cả hai hướng, đạt được nội tuyến chức năng. Hướng dẫn di chuyển cần được ghi lại để nhập và thoát hàm để sao chép các đối số vào và trả về giá trị. Các câu lệnh di chuyển này sau đó được trình biên dịch tối ưu hóa bằng cách sử dụng truyền bản sao. Để có thể quay trở lại trình thông dịch, dấu vết cũng phải tạo ra LIR để ghi lại rằng một khung cuộc gọi đã được nhập và thoát. LIR vào và ra khung lưu vừa đủ thông tin để cho phép khôi phục ngăn xếp cuộc gọi intepreter sau này và đơn giản hơn nhiều so với mã cuộc gọi tiêu chuẩn của trình thông dịch. Nếu hàm đang được nhập không phải là hằng số (trong JavaScript bao gồm bất kỳ lệnh gọi nào theo tên hàm), bộ ghi cũng phải phát ra LIR để bảo vệ rằng hàm đó là giống nhau.

Bảo vệ và lối ra bên. Mỗi tối ưu hóa được mô tả ở trên yêu cầu một hoặc nhiều người bảo vệ để xác minh các giả định được thực hiện khi thực hiện tối ưu hóa. Một người bảo vệ chỉ là một nhóm các lệnh LIR thực hiện kiểm tra và lối ra có điều kiện. Lối ra rẽ nhánh đến một lối ra bên cạnh, một đoạn LIR nhỏ ngoài dấu vết trả về một con trỏ đến một cấu trúc mô tả lý do cho lối ra cùng với PC thông dịch tại điểm thoát và bất kỳ dữ liệu nào khác cần thiết để khôi phục cấu trúc trạng thái của trình thông dịch .

Hủy bỏ. Một số cấu trúc khó ghi lại trong dấu vết LIR.
Ví dụ, eval hoặc các lệnh gọi đến các chức năng bên ngoài có thể thay đổi trạng thái chương trình theo những cách không thể đoán trước, khiến trình dò tìm khó biết được bản đồ loại hiện tại để tiếp tục truy tìm.
Việc triển khai theo dõi cũng có thể có bất kỳ số limi nào khác, ví dụ: thiết bị bộ nhớ nhỏ có thể giới hạn độ dài của dấu vết.
Khi bất kỳ tình huống nào xảy ra khiến việc triển khai không thể tiếp tục ghi lại dấu vết, việc triển khai sẽ hủy bỏ việc nhập hồ sơ theo dõi và quay trở lại theo dõi dấu vết.

3.2 Theo dõi cây

Đặc biệt là các vòng lặp đơn giản, cụ thể là những vòng mà luồng điều khiển, kiểu giá trị, biểu diễn giá trị và các hàm nội tuyến đều bất biến, có thể được biểu diễn bằng một dấu vết duy nhất. Nhưng hầu hết các vòng lặp đều có ít nhất một số biến thể, và vì vậy chương trình sẽ thực hiện các lối thoát phụ khỏi dấu vết chính. Khi một lối ra bên trở nên nóng, TraceMonkey bắt đầu một dấu vết nhánh mới từ điểm đó và vá lối ra bên để chuyển trực tiếp đến dấu vết đó. Bằng cách này, một dấu vết mở rộng theo yêu cầu thành một cây dấu vết một lối vào, nhiều lối ra.

Phần này giải thích cách cây dấu vết được hình thành trong quá trình thực thi. Mục đích là để tạo ra các cây theo dõi trong quá trình thực thi bao gồm tất cả các đường dẫn nóng của chương trình.

Khởi đầu một cái cây. Cây cối luôn bắt đầu ở các đầu đường vòng, bởi vì chúng là một nơi tự nhiên để tìm kiếm những con đường nóng. Trong TraceMonkey, các tiêu đề vòng lặp rất dễ phát hiện – trình biên dịch mã bytecode đảm bảo rằng một mã byte là một tiêu đề vòng lặp vì nó là mục tiêu của một nhánh lùi.
TraceMonkey bắt đầu một cây khi một tiêu đề vòng lặp nhất định đã bị cắt exe một số lần nhất định (2 trong lần triển khai hiện tại).
Khởi động một cây chỉ có nghĩa là bắt đầu ghi lại dấu vết cho điểm hiện tại và nhập bản đồ và đánh dấu dấu vết là gốc của cây. Mỗi cây được liên kết với một tiêu đề vòng lặp và bản đồ kiểu, vì vậy có thể có một số cây cho một tiêu đề vòng lặp nhất định.

Kết thúc vòng lặp. Việc ghi lại dấu vết có thể kết thúc theo nhiều cách.

Lý tưởng nhất là dấu vết đến tiêu đề vòng lặp nơi nó bắt đầu với cùng loại bản đồ như khi nhập. Đây được gọi là phép lặp vòng lặp kiểu ổn định. Trong trường hợp này, phần cuối của dấu vết có thể nhảy ngay sang phần đầu, vì tất cả các biểu diễn giá trị đều chính xác khi cần thiết để nhập dấu vết. Bước nhảy thậm chí có thể bỏ qua mã thông thường sẽ sao chép trạng thái ở cuối dấu vết và sao chép lại vào bản ghi kích hoạt dấu vết để nhập dấu vết.

Trong một số trường hợp nhất định, dấu vết có thể đến tiêu đề vòng lặp với một bản đồ loại khác. Tình huống này đôi khi được quan sát cho lần lặp đầu tiên của một vòng lặp. Một số biến bên trong vòng lặp ban đầu có thể không được xác định, trước khi chúng được đặt thành một kiểu cụ thể trong lần lặp lại vòng lặp đầu tiên. Khi ghi lại một lần lặp lại như vậy, bộ ghi không thể liên kết dấu vết trở lại tiêu đề vòng lặp của chính nó vì nó không ổn định về kiểu.
Thay vào đó, quá trình lặp được kết thúc bằng một lối ra bên sẽ luôn không thành công và quay trở lại trình thông dịch. Đồng thời, một dấu vết mới được ghi lại với bản đồ kiểu mới. Mỗi khi một dấu vết không ổn định kiểu bổ sung được thêm vào một khu vực, bản đồ kiểu lối ra của nó sẽ được so sánh với bản đồ nhập của tất cả các dấu vết hiện có trong trường hợp chúng bổ sung cho nhau. Với cách tiếp cận này, chúng tôi có thể bao gồm các lặp lại vòng lặp kiểu không ổn định miễn là cuối cùng chúng tạo thành một trạng thái cân bằng ổn định.

Cuối cùng, dấu vết có thể thoát khỏi vòng lặp trước khi đến tiêu đề vòng lặp, ví dụ: vì việc thực thi đạt đến câu lệnh break hoặc return. Trong trường hợp này, VM chỉ đơn giản là kết thúc quá trình theo dõi bằng một lối ra vào màn hình theo dõi.

Như đã đề cập trước đó, chúng tôi có thể suy đoán chọn gửi lại một số giá trị được đánh số nhất định dưới dạng số nguyên đang theo dõi. Chúng tôi làm như vậy khi chúng tôi quan sát thấy các biến kiểu Số chứa một giá trị số nguyên tại mục nhập theo dõi. Nếu trong quá trình ghi theo dõi, biến được gán một giá trị không phải là số nguyên mà không rõ ràng, chúng ta phải mở rộng kiểu của biến thành nhân đôi. Do đó, dấu vết được ghi lại trở nên không ổn định về kiểu vì nó bắt đầu bằng giá trị số nguyên nhưng kết thúc bằng giá trị kép. Điều này đại diện cho một suy đoán sai, vì khi nhập dấu vết, chúng tôi chuyên biệt giá trị kiểu Số thành một số nguyên, giả sử rằng tại cạnh vòng lặp, chúng tôi sẽ lại tìm thấy một giá trị nguyên trong biến, cho phép chúng tôi đóng vòng lặp. Để tránh các lỗi suy đoán trong tương lai liên quan đến biến này và để có được dấu vết ổn định kiểu, chúng tôi lưu ý thực tế rằng biến được đề cập như đã được quan sát đôi khi giữ các giá trị không phải số nguyên trong cấu trúc dữ liệu tư vấn mà chúng tôi gọi là “tiên tri”.

Khi biên dịch các vòng lặp, chúng tôi tham khảo ý kiến của nhà tiên tri trước khi đặc biệt hóa các giá trị vào số nguyên. Việc suy đoán đối với các số nguyên chỉ được thực hiện nếu nhà tiên tri không biết thông tin bất lợi nào về biến cụ thể đó. Bất cứ khi nào chúng tôi vô tình biên dịch một vòng lặp không ổn định về kiểu do có thể suy đoán sai về một biến thể được nhập Số, chúng tôi ngay lập tức kích hoạt việc ghi lại một dấu vết mới, dựa trên thông tin tiên tri được cập nhật hiện sẽ bắt đầu bằng giá trị dou ble và do đó trở thành loại ổn định.

Kéo dài một cái cây. Các lối ra bên cạnh dẫn đến các đường dẫn khác nhau qua vòng lặp, hoặc các đường dẫn có các kiểu hoặc biểu diễn khác nhau. Do đó, để bao phủ hoàn toàn vòng lặp, máy ảo phải ghi lại các dấu vết bắt đầu từ tất cả các lối ra bên. Các dấu vết này được ghi lại giống như dấu vết gốc: có một bộ đếm cho mỗi lối ra bên và khi bộ đếm đạt đến ngưỡng độ nóng, quá trình ghi sẽ bắt đầu. Quá trình ghi dừng chính xác như đối với dấu vết gốc, sử dụng tiêu đề vòng lặp của dấu vết gốc làm mục tiêu để tiếp cận.

[1] Mảng thực sự tệ hơn thế này: nếu giá trị chỉ mục là một số, nó phải được chuyển đổi từ kép sang chuỗi cho toán tử truy cập thuộc tính và sau đó thành số nguyên bên trong để triển khai mảng.

Việc triển khai của chúng tôi không mở rộng ở tất cả các lối ra bên. Nó chỉ mở rộng nếu lối ra bên dành cho nhánh luồng điều khiển và chỉ khi lối ra bên không rời khỏi vòng lặp. Đặc biệt, chúng tôi không muốn mở rộng một cây dấu vết dọc theo một con đư ờng dẫn đến một vòng lặp bên ngoài, bởi vì chúng tôi muốn che những con đư ờng như vậy trong một cây bên ngoài thông qua việc lồng vào cây.

3.3 Danh sách đen Đôi

khi, một chư ơng trình đi theo một đư ờng dẫn mà không thể đư ợc biên dịch thành một dấu vết, thư ờng là do những hạn chế trong việc triển khai.
TraceMonkey hiện không hỗ trợ ghi lại việc ném và bắt các trư ờng hợp ngoại lệ tùy ý. Sự đánh đổi thiết kế này đã đư ợc chọn, bởi vì các trư ờng hợp ngoại lệ thư ờng rất hiếm trong JavaScript. Tuy nhiên, nếu một chư ơng trình chọn sử dụng nhiều ngoại lệ, chúng tôi sẽ đột nhiên phải chịu phí thời gian chạy đáng phạt nếu chúng tôi liên tục cố gắng ghi lại một dấu vết cho đư ờng dẫn này và liên tục không thực hiện đư ợc, vì chúng tôi hủy truy tìm mỗi khi chúng tôi quan sát thấy một ngoại lệ đư ợc ném ra.

Do đó, nếu một vòng lặp nóng chứa các dấu vết luôn bị lỗi, máy ảo có thể chạy chậm hơn nhiều so với trình thông dịch cơ sở: máy ảo liên tục dành thời gian để ghi lại dấu vết, như ng không bao giờ có thể chạy bất kỳ dấu vết nào. Để tránh vấn đề này, bất cứ khi nào VM chuẩn bị bắt đầu theo dõi, nó phải cố gắng dự đoán xem nó có kết thúc quá trình theo dõi hay không.

Thuật toán dự đoán của chúng tôi dựa trên các dấu vết trong danh sách đen đã đư ợc thử và không thành công. Khi máy ảo không hoàn thành một lần bắt đầu theo dõi tại một điểm nhất định, máy ảo sẽ ghi lại rằng lỗi đã xảy ra. VM cũng thiết lập một bộ đếm để nó sẽ không cố gắng ghi lại một dấu vết bắt đầu từ thời điểm đó cho đến khi nó đư ợc thông qua một vài lần nữa (32 trong phần đề cập của chúng tôi). Bộ đếm dự phòng này đư a ra các điều kiện tạm thời ngăn cản việc truy tìm có cơ hội kết thúc. Ví dụ, một vòng lặp có thể hoạt động khác trong khi khởi động so với trong quá trình thực thi ở trạng thái ổn định của nó. Sau một số lỗi nhất định (2 trong quá trình triển khai của chúng tôi), VM đánh dấu phân đoạn là danh sách đen, có nghĩa là VM sẽ không bao giờ bắt đầu ghi lại vào thời điểm đó.

Sau khi thực hiện chiến lư ợc cơ bản này, chúng tôi nhận thấy rằng đối với các vòng lặp nhỏ bị đư a vào danh sách đen, hệ thống có thể dành một khoảng thời gian đáng kể chỉ để tìm đoạn vòng lặp và xác định rằng nó đã đư ợc đư a vào danh sách đen. Bây giờ chúng tôi tránh vấn đề đó bằng cách vá mã bytecode. Chúng tôi xác định thêm một mã bytecode no-op chỉ ra một tiêu đề vòng lặp. VM gọi vào theo dõi giám sát mỗi khi inter preter thực hiện no-op tiêu đề vòng lặp. Để đư a vào danh sách đen một phân đoạn, chúng tôi chỉ cần thay thế no-op của tiêu đề vòng lặp bằng no-op thông thư ờng. Do đó, thông dịch viên sẽ không bao giờ gọi vào màn hình theo dõi nữa.

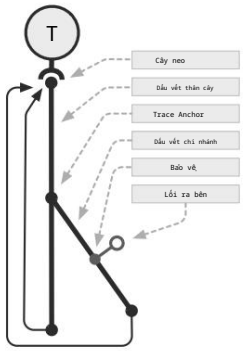
Có một sự cố liên quan mà chúng tôi chư a giải quyết đư ợc, sự cố này xảy ra khi vòng lặp đáp ứng tất cả các điều kiện sau:

- Máy ảo có thể tạo ít nhất một dấu vết gốc cho vòng lặp.
- Có ít nhất một lối ra bên nóng mà máy ảo không thể hoàn thành một dấu vết.
- Thân vòng ngắn.

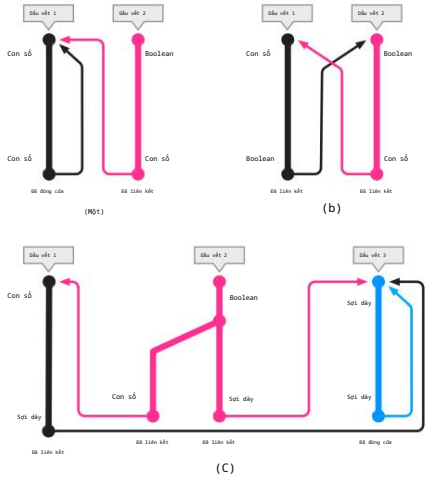
Trong trư ờng hợp này, máy ảo sẽ liên tục chuyển tiêu đề vòng lặp, tìm kiếm dấu vết, tìm thấy nó, thực thi nó và quay trở lại trình thông dịch.
Với phần thân vòng lặp ngắn, chi phí tìm và gọi dấu vết là cao và khiến hiệu suất thậm chí còn chậm hơn so với trình thông dịch cơ bản. Cho đến nay, trong tình huống này, chúng tôi đã cải thiện việc triển khai để VM có thể hoàn thành việc theo dõi chi nhánh.
Như ng khó có thể đảm bảo rằng tình trạng này sẽ không bao giờ xảy ra.
Trong tư ơng lai, tình huống này có thể tránh đư ợc bằng cách phát hiện và đư a vào danh sách đen các vòng lặp mà cuộc gọi theo dõi trung bình thực hiện một vài mã byte trư ớc khi quay trở lại trình thông dịch.

4. Sự hình thành cây theo dõi lồng nhau

Hình 7 cho thấy biên dịch cây theo dõi cơ bản (11) đư ợc áp dụng cho một vòng lặp lồng nhau trong đó vòng lặp bên trong chứa hai đư ờng dẫn. Thông thư ờng, vòng lặp bên trong (với tiêu đề tại i2) trở nên nóng trư ớc tiên và một cây theo dõi đư ợc bắt nguồn từ thời điểm đó. Ví dụ, dấu vết đư ợc ghi lại đầu tiên có thể là một chu kỳ



Hình 5. Cây có hai vết, một vết thân và một vết cành. Dấu vết thân cây chứa một lớp bảo vệ mà dấu vết nhánh đã đư ợc gắn vào. Dấu vết nhánh chứa một bộ phận bảo vệ có thể bị lỗi và kích hoạt lối ra bên. Cả thân cây và dấu vết nhánh đều quay trở lại mỏ neo của cây, đó là điểm bắt đầu của cây dấu vết.

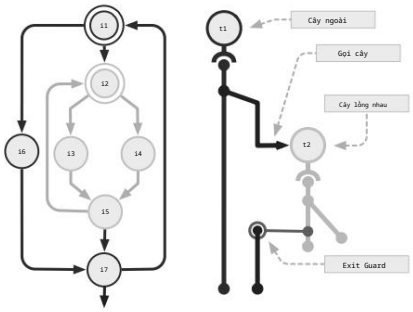


Hình 6. Chúng tôi xử lý các vòng lặp kiểu không ổn định bằng cách cho phép biên dịch các dấu vết không thể lặp lại chính chúng do khớp sai kiểu. Khi các dấu vết như vậy tích tụ, chúng tôi cố gắng kết nối các cạnh vòng lặp của chúng để tạo thành các nhóm cây dấu vết có thể thực thi mà không cần phải thoát ra bên cạnh trình thông dịch để bao gồm các trư ờng hợp loại lẻ. Điều này cực kỳ quan trọng đối với các cây dấu vết lồng nhau trong đó cây bên ngoài cố gắng gọi cây bên trong (hoặc trong trư ờng hợp này là rừng cây bên trong), vì các vòng bên trong thư ờng có giá trị không xác định ban đầu thay đổi kiểu thành giá trị cụ thể sau lần lặp đầu tiên .

thông qua vòng lặp bên trong, {i2, i3, i5, α}. Biểu tư ợng α đư ợc sử dụng để chỉ ra rằng dấu vết lặp lại neo cây.

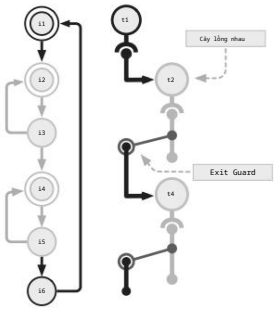
Khi việc thực thi rời khỏi vòng lặp bên trong, thiết kế cơ bản có hai lựa chọn. Đầu tiên, hệ thống có thể ngừng truy tìm và từ bỏ việc biên dịch vòng lặp bên ngoài, rõ ràng là một giải pháp không mong muốn. Sự lựa chọn khác là tiếp tục truy tìm, biên dịch các dấu vết cho vòng lặp bên ngoài bên trong cây dấu vết của vòng lặp bên trong.

Ví dụ: chư ơng trình có thể thoát ở i5 và ghi lại dấu vết nhánh kết hợp vòng lặp ngoài: {i5, i7, i1, i6, i7, i1, α}.
Sau đó, chư ơng trình có thể lấy nhánh khác tại i2 và sau đó thoát ra, ghi lại dấu vết nhánh khác kết hợp vòng lặp bên ngoài: {i2, i4, i5, i7, i1, i6, i7, i1, α}. Do đó, vòng lặp bên ngoài đư ợc ghi lại và biên dịch hai lần, và cả hai bản sao phải đư ợc giữ lại trong bộ nhớ đệm theo dõi.



(Một) (b)

Hình 7. Đồ thị luồng điều khiển của một vòng lặp lồng nhau với câu lệnh if bên trong vòng lặp bên trong nhất (a). Một cây bên trong nắm bắt vòng lặp bên trong và đư ợc lồng vào bên trong một cây bên ngoài mà nó "gọi" cây bên trong.
Cây bên trong quay trở lại cây bên ngoài khi nó thoát ra dọc theo trình bảo vệ điều kiện vòng lặp của nó (b).



Hình 8. Đồ thị luồng điều khiển của một vòng lặp với hai vòng lặp lồng nhau (bên trái) và cấu hình cây theo dõi lồng nhau của nó (bên phải). Cây bên ngoài gọi hai cây dấu vết lồng nhau bên trong và đặt lính canh tại các vị trí lối ra bên cạnh của chúng.

Nói chung, nếu các vòng lặp đư ợc lồng vào nhau đến độ sâu k và mỗi vòng lặp có n đư ờng đi (tính theo giá trị trung bình hình học), thì chiến lư ợc na¨ ıve này mang lại n^k dấu vết, có thể dễ dàng lấp đầy bộ nhớ đệm dấu vết.

Để thực thi các chư ơng trình với các vòng lồng nhau một cách hiệu quả, hệ thống theo dõi cần có kỹ thuật để bao phủ các vòng lồng nhau bằng mã gốc mà không bị trùng lặp dấu vết theo cấp số nhân.

4.1 Thuật toán lồng ghép Thông

tin quan trọng là nếu mỗi vòng lặp đư ợc biểu diễn bằng cây dấu vết riêng của nó, thì mã cho mỗi vòng lặp chỉ có thể đư ợc chứa trong cây của chính nó và các đư ờng dẫn vòng ngoài sẽ không bị trùng lặp. Một thực tế quan trọng khác là chúng tôi không truy tìm các mã byte tùy ý có thể có đồ thị luồng điều khiển không thể thu đư ợc, mà là các mã byte do trình biên dịch tạo ra cho một ngôn ngữ có luồng điều khiển có cấu trúc. Do đó, với hai cạnh vòng lặp, hệ thống có thể dễ dàng xác định xem chúng có đư ợc lồng vào nhau hay không và đâu là vòng lặp bên trong. Sử dụng kiến thức này, hệ thống có thể biên dịch các vòng lặp bên trong và bên ngoài một cách riêng biệt, và làm cho các dấu vết của vòng lặp bên ngoài đư ợc gọi là cây dấu vết của vòng lặp bên trong.

Thuật toán xây dựng cây dấu vết lồng nhau như sau. Chúng tôi bắt đầu truy tìm tại các tiêu đề vòng lặp chính xác như trong hệ thống truy tìm cơ bản.
Khi chúng tôi thoát khỏi một vòng lặp (đư ợc phát hiện bằng cách so sánh PC thông dịch với phạm vi đư ợc cung cấp bởi cạnh vòng lặp), chúng tôi dừng theo dõi. Bư ớc quan trọng của thuật toán xảy ra khi chúng ta đang ghi lại một dấu vết cho vòng lặp LR (R cho vòng lặp đang đư ợc ghi lại) và chúng tôi đến tiêu đề của một vòng lặp khác LO (O cho vòng lặp khác). Lư u ý rằng LO phải là một vòng lặp bên trong của LR vì chúng ta dừng dấu vết khi chúng ta thoát khỏi một vòng lặp.

- Nếu LO có cây dấu vết đã biên dịch phù hợp với kiểu, chúng ta gọi LO là cây dấu vết lồng nhau. Nếu cuộc gọi thành công, thì chúng tôi ghi lại cuộc gọi trong dấu vết cho LR. Trên các lần thực thi trong tư ơng lai, dấu vết cho LR sẽ gọi trực tiếp dấu vết bên trong. • Nếu LO chư a có cây theo dõi đã biên dịch phù hợp với kiểu, chúng ta

phải lấy nó trư ớc khi có thể tiếp tục. Để làm điều này, chúng tôi chỉ cần hủy ghi lại dấu vết đầu tiên. Bộ theo dõi sẽ thấy tiêu đề vòng lặp bên trong và ngay lập tức sẽ bắt đầu ghi lại vòng lặp bên trong.

Nếu tất cả các vòng lặp trong một tổ là kiểu ổn định, thì lồng ghép vòng lặp sẽ không tạo ra sự trùng lặp. Ngư ợc lại, nếu các vòng đư ợc lồng vào nhau ở độ sâu k và mỗi vòng lặp đư ợc nhập với m bản đồ kiểu khác nhau (trung bình hình học), sau đó chúng tôi biên dịch O (mk) bản sao của vòng lặp trong cùng. Miễn là m gần bằng 1, các cây dấu vết thu đư ợc sẽ có thể xử lý đư ợc.

Một chi tiết quan trọng là lệnh gọi đến cây dấu vết bên trong phải hoạt động giống như một địa chỉ gọi hàm: nó phải quay trở lại cùng một điểm mọi lúc.
Mục tiêu của việc làm tổ là làm cho các vòng bên trong và bên ngoài độc lập; do đó khi cây bên trong đư ợc gọi, nó phải thoát đến cùng một điểm trong cây bên ngoài mọi lúc với cùng một loại bản đồ. Bởi vì chúng tôi thực sự không thể đảm bảo tài sản này, chúng tôi phải bảo vệ nó sau cuộc gọi, và thoát ra bên cạnh nếu tài sản không đư ợc giữ. Một lý do phổ biến khiến cây bên trong không trở lại cùng một điểm là nếu cây bên trong có một lối ra bên mới mà nó chư a bao giờ biên dịch một dấu vết. Tại thời điểm này, PC thông dịch đang ở trong cây bên trong, vì vậy chúng tôi không thể tiếp tục ghi hoặc thực hiện cây bên ngoài.

Nếu điều này xảy ra trong quá trình ghi, chúng tôi sẽ loại bỏ dấu vết bên ngoài, để tạo cơ hội cho cây bên trong phát triển xong. Sau đó, một quá trình thực thi cây bên ngoài trong tư ơng lai sẽ có thể kết thúc đúng cách và ghi lại một lệnh gọi đến cây bên trong. Nếu một lối ra phía cây bên trong xảy ra trong quá trình thực hiện một dấu vết đã biên dịch cho cây bên ngoài, chúng ta chỉ cần thoát khỏi dấu vết bên ngoài và bắt đầu ghi lại một nhánh mới trong cây bên trong.

4.2 Danh sách đen có lồng ghép Thuật toán

đư a vào danh sách đen cần sửa đổi để hoạt động tốt với lồng ghép. Vấn đề là các dấu vết vòng lặp bên ngoài thư ờng bị loại bỏ trong quá trình khởi động (vì cây bên trong không có sẵn hoặc có một lối ra bên), điều này sẽ dẫn đến việc chúng nhanh chóng bị đư a vào danh sách đen của thuật toán cơ bản.

Quan sát chính là khi một dấu vết bên ngoài bị hủy bỏ do cây bên trong chư a sẵn sàng, đây có thể là một điều kiện tạm thời.
Vì vậy, chúng ta không nên tính những lần hủy bỏ như vậy vào danh sách đen miễn là chúng ta có thể tạo thêm dấu vết cho cây bên trong.

Trong cách triển khai của chúng tôi, khi một cây bên ngoài ngừng hoạt động trên cây bên trong, chúng tôi tăng bộ đếm danh sách đen của cây bên ngoài như bình thư ờng và lùi lại việc biên dịch nó. Khi cây bên trong kết thúc một dấu vết, chúng tôi giảm bộ đếm danh sách đen trên vòng lặp bên ngoài, "tha thứ" cho vòng lặp bên ngoài vì đã hủy bỏ trư ớc đó. Chúng tôi cũng hoàn tác việc lùi lại để cây bên ngoài có thể bắt đầu ngay lập tức cố gắng biên dịch vào lần tiếp theo chúng tôi tiếp cận nó.

5. Tối ư u hóa cây theo dõi Phần này giải thích

cách một dấu vết đã ghi đư ợc dịch thành một dấu vết mã máy đư ợc tối ư u hóa. Hệ thống con biên dịch theo dõi, NANOJIT, tách biệt với VM và có thể đư ợc sử dụng cho các ứng dụng khác.

[2] Thay vì hủy bỏ bản ghi bên ngoài, về cơ bản, chúng tôi có thể chỉ tạm dừng bản ghi, như ng điều đó sẽ yêu cầu việc triển khai có thể ghi lại nhiều dấu vết đồng thời, làm phức tạp việc triển khai, trong khi chỉ lư u một vài lần lặp lại trong trình thông dịch.

5.1 Tối ư u hóa Bởi vì

các dấu vết ở dạng SSA và không có điểm nối hoặc nút φ, các tối ư u hóa nhất định rất dễ thực hiện. Để có đư ợc hiệu suất khởi động tốt, các tối ư u hóa phải chạy nhanh chóng, vì vậy chúng tôi đã chọn một nhóm tối ư u hóa nhỏ. Chúng tôi đã triển khai các tối ư u hóa dư ới dạng bộ lọc pipelined để chúng có thể đư ợc bật và tắt độc lập, như ng tất cả đều chạy chỉ trong hai vòng lặp đi qua dấu vết: một tiến và một lùi.

Mỗi khi bộ ghi theo dõi phát ra lệnh LIR, cấu trúc trong ngay lập tức đư ợc chuyển tới bộ lọc đầu tiên trong đư ờng ống chuyển tiếp. Do đó, tối ư u hóa bộ lọc chuyển tiếp đư ợc thực hiện khi dấu vết đư ợc ghi lại. Mỗi bộ lọc có thể chuyển từng lệnh cho bộ lọc tiếp theo không thay đổi, viết một lệnh khác cho bộ lọc tiếp theo hoặc không viết lệnh nào cả. Ví dụ: bộ lọc gấp không đổi có thể thay thế lệnh nhân như v13 : = mul3, 1000 bằng lệnh hằng v13 = 3000.

Chúng tôi hiện đang áp dụng bốn bộ lọc chuyển tiếp:

- Trên các ISA không có lệnh dấu phẩy động, bộ lọc soft-float sẽ chuyển đổi các lệnh LIR dấu phẩy động thành chuỗi lệnh số nguyên.
- CSE (loại bỏ biểu thức con không đổi), • đơn giản hóa biểu thức, bao gồm việc gấp không đổi và một số đặc điểm đại số (ví dụ: a - a = 0), và
- đơn giản hóa biểu thức ngữ nghĩa cụ thể của ngôn ngữ nguồn, chủ yếu là các định dạng đại số cho phép thay thế DOUBLE bằng INT. Ví dụ: LIR chuyển đổi INT thành DOUBLE và sau đó quay lại sẽ bị bộ lọc này loại bỏ.

Khi quá trình ghi dấu vết hoàn tất, nanojit sẽ chạy các bộ lọc tối ư u hóa ngư ợc. Chúng đư ợc sử dụng để tối ư u hóa yêu cầu phân tích chư ơng trình ngư ợc. Khi chạy các bộ lọc ngư ợc, nanojit đọc một lệnh LIR tại một thời điểm và các lần đọc đư ợc chuyển qua đư ờng ống.

Chúng tôi hiện áp dụng ba bộ lọc ngư ợc:

- Loại bỏ kho lư u trữ ngăn xếp dữ liệu đã chết. Dấu vết LIR mã hóa nhiều cửa hàng đến các vị trí trong ngăn xếp thông dịch viên. Như ng những giá trị này không bao giờ đư ợc đọc lại trư ớc khi thoát ra khỏi dấu vết (bởi trình thông dịch hoặc một dấu vết khác). Do đó, các kho lư u trữ vào ngăn xếp bị ghi đè trư ớc khi lối ra tiếp theo bị chết. Các cửa hàng ở vị trí nằm ngoài đầu ngăn xếp thông dịch viên tại các lối ra trong tư ơng lai cũng đã chết. • Loại bỏ ngăn xếp cuộc gọi đã chết. Đây là cách tối ư u hóa tư ơng tự như trên, ngoại trừ áp dụng cho ngăn xếp cuộc gọi của trình thông dịch đư ợc sử dụng cho nội tuyến lệnh gọi hàm.
- Loại bỏ mã chết. Điều này giúp loại bỏ bất kỳ hoạt động nào lư u trữ đến một giá trị không bao giờ đư ợc sử dụng.

Sau khi một lệnh LIR đư ợc đọc thành công (“kéo”) từ đư ờng ống bộ lọc ngư ợc, trình tạo mã của nanojit sẽ phát ra (các) lệnh máy gốc cho nó.

5.2 Phân bổ thanh ghi Chúng tôi

sử dụng một trình cấp phát thanh ghi tham lam đơn giản để thực hiện một lần chuyển lùi duy nhất qua dấu vết (nó đư ợc tích hợp với erator gen mã). Vào thời điểm trình cấp phát đạt đến một lệnh như v3 = add v1, v2, nó đã gán một thanh ghi cho v3. Nếu v1 và v2 chư a đư ợc chỉ định các thanh ghi, bộ cấp phát sẽ chỉ định một thanh ghi miễn phí cho mỗi thanh ghi. Nếu không có thanh ghi miễn phí, một giá trị đư ợc chọn để làm tràn. Chúng tôi sử dụng phư ơng pháp phân tích lớp để chọn giá trị mang của thanh ghi “cũ nhất” (6).

Heuristic coi tập R của các giá trị v trong các thanh ghi ngay sau lệnh hiện tại để làm tràn. Gọi vm là lệnh cuối cùng trư ớc dòng điện mà mỗi v đư ợc tham chiếu. Sau đó

Thẻ JS	Loại xx1 số	Mô tả con trỏ
000	đối tư ợng 010	biểu diễn số nguyên 31-bit tới con
số	100 chuỗi 110	trỏ xử lý JSObject để con trỏ xử
boolean	null hoặc	lý kép tới liệt kê xử lý JSString
không	xác định	cho null, undefined, true, false

Hình 9. Các giá trị đư ợc gắn thẻ trong trình thông dịch SpiderMonkey JS. Kiểm tra thẻ, mở hộp (trích xuất giá trị không đư ợc gắn thẻ) và quyền anh (tạo giá trị đư ợc gắn thẻ) là những chi phí đáng kể. Tránh những chi phí này là lợi ích chính của việc truy tìm nguồn gốc.

heuristic chọn v với vm tối thiểu. Động lực là điều này giải phóng một đăng ký càng lâu càng tốt nếu chỉ có một lần đổ.

Nếu chúng ta cần đổ giá trị so với tại thời điểm này, chúng ta tạo mã khôi phục ngay sau mã cho lệnh hiện tại. Mã tràn tư ơng ứng đư ợc tạo ngay sau điểm cuối cùng mà vs đư ợc sử dụng. Thanh ghi đã đư ợc gán cho vs đư ợc đánh dấu miễn phí cho mã trư ớc đó, vì thanh ghi đó bây giờ có thể đư ợc sử dụng tự do mà không ảnh hư ởng đến mã sau

6. Triển khai Để chứng minh tính

hiệu quả của phư ơng pháp tiếp cận của chúng tôi, chúng tôi đã thiết lập một trình biên dịch động dựa trên dấu vết cho Máy ảo JavaScript SpiderMonkey (4). SpiderMonkey là máy ảo JavaScript đư ợc nhúng trong trình duyệt web nguồn mở Firefox của Mozilla (2), đư ợc hơn 200 triệu ngư ời dùng trên toàn thế giới sử dụng. Cốt lõi của SpiderMonkey là một trình thông dịch mã bytecode đư ợc triển khai bằng C ++.

Trong SpiderMonkey, tất cả các giá trị JavaScript đư ợc thể hiện bằng kiểu jsval. Jsval là từ máy trong đó tối đa 3 bit ít quan trọng nhất là thẻ loại và các bit còn lại là dữ liệu.
Xem Hình 6 để biết thêm chi tiết. Tất cả các con trỏ chứa trong khoảng thời gian đều trỏ đến các khối do GC điều khiển đư ợc căn chỉnh trên các ranh giới 8 byte.

Giá trị đối tư ợng JavaScript là ánh xạ của các tên thuộc tính có giá trị chuỗi thành các giá trị tùy ý. Chúng đư ợc thể hiện theo một trong hai cách trong SpiderMonkey. Hầu hết các đối tư ợng đư ợc đại diện bởi một mô tả tural struc đư ợc chia sẻ, đư ợc gọi là hình dạng đối tư ợng, ánh xạ tên thuộc tính với các chỉ mục mảng bằng cách sử dụng bảng băm. Đối tư ợng lư u trữ một con trỏ đến hình dạng và mảng các giá trị thuộc tính của chính nó. Các đối tư ợng có bộ tên thuộc tính lớn, duy nhất sẽ lư u trữ các thuộc tính của chúng trực tiếp trong bảng băm.

Ngư ời thu gom rác là một ngư ời thu gom đánh dấu và quét chính xác, không thế hệ, dừng lại ở thế giới.

Trong phần còn lại của phần này, chúng tôi thảo luận về các lĩnh vực chính của việc triển khai khóa TraceMon.

6.1 Gọi dấu vết đã biên dịch Các dấu

vết đã biên dịch đư ợc lư u trữ trong bộ nhớ cache dấu vết, đư ợc lập chỉ mục bởi PC intepreter và bản đồ loại. Các dấu vết đư ợc biên dịch để chúng có thể đư ợc gọi là các hàm bằng cách sử dụng các quy ư ớc gọi gốc tiêu chuẩn (ví dụ: FASTCALL trên x86).

Trình thông dịch phải nhấn vào một cạnh vòng lặp và đi vào màn hình để gọi một dấu vết gốc lần đầu tiên. Màn hình tính toán bản đồ loại hiện tại, kiểm tra bộ nhớ cache theo dõi để tìm dấu vết cho PC hiện tại và bản đồ loại, và nếu nó tìm thấy, thực hiện theo dõi.

Để thực hiện theo dõi, trình giám sát phải xây dựng một bản ghi kích hoạt theo dõi chứa các biến cục bộ và toàn cục đã nhập, không gian ngăn xếp tạm thời và không gian cho các đối số cho các lệnh gọi nguyên bản. Các giá trị cục bộ và toàn cục sau đó đư ợc sao chép từ trạng thái thông dịch sang bản ghi kích hoạt theo dõi. Sau đó, dấu vết đư ợc gọi giống như một con trỏ hàm C bình thư ờng.

Khi một cuộc gọi theo dõi trở lại, màn hình sẽ khôi phục trình thông dịch tình trạng. Đầu tiên, màn hình kiểm tra lý do thoát ra dấu vết và áp dụng danh sách đen nếu cần. Sau đó, nó bật lên hoặc tổng hợp các khung ngăn xếp cuộc gọi JavaScript khác nhau khi cần thiết. Cuối cùng, nó sao chép đã nhập các biến trở lại từ bản ghi kích hoạt theo dõi về trạng thái trình diễn dịch.

Ít nhất là trong việc triển khai hiện tại, các bư ớc này có chi phí thời gian chạy không đáng kể, do đó, giảm thiểu số lư ợng chuyển đổi trình thông dịch để theo dõi và từ theo dõi sang phiên dịch là điều cần thiết cho perfor mance. (xem thêm Phần 3.3). Các thí nghiệm của chúng tôi (xem Hình 12) cho thấy rằng đối với các chư ơng trình, chúng tôi có thể theo dõi tốt các chuyển đổi như vậy không thư ờng xuyên và do đó không đóng góp đáng kể vào tổng số thời gian chạy. Trong một số chư ơng trình, nơi hệ thống bị ngăn chặn ghi lại dấu vết nhánh cho các lối ra bên nóng bằng cách hủy bỏ, chi phí này có thể tăng lên đến 10% tổng thời gian thực hiện.

6.2 Khâu theo vết

Việc chuyển đổi từ một dấu vết sang một dấu vết nhánh ở một lối ra bên cạnh tránh chi phí gọi dấu vết từ màn hình, trong một tính năng đư ợc gọi là dấu vết đư ờng khâu. Tại một lối ra bên, dấu vết thoát chỉ cần ghi trực tiếp các giá trị đư ợc đăng ký mang trở lại bản ghi kích hoạt theo dõi của nó. Trong hợp đồng của chúng tôi, các bản đồ loại giống hệt nhau mang lại bản ghi kích hoạt giống hệt nhau bố cục, do đó, bản ghi kích hoạt theo dõi có thể đư ợc sử dụng lại ngay lập tức bằng dấu vết nhánh.

Trong các chư ơng trình có cây dấu vết nhánh với dấu vết nhỏ, dấu vết khâu có một chi phí đáng chú ý. Mặc dù ghi vào bộ nhớ và sau đó đọc lại sẽ sớm đư ợc kỳ vọng là sẽ có L1 cao tỷ lệ truy cập bộ nhớ cache, đối với các dấu vết nhỏ, số lư ợng lệnh tăng lên đã một chi phí đáng chú ý. Ngoài ra, nếu ghi và đọc rất gần nhau trong luồng hư ớng dẫn động, chúng tôi đã tìm thấy rằng hiện tại bộ xử lý x86 thư ờng phải chịu hình phạt từ 6 chu kỳ trở lên (ví dụ: nếu các hư ớng dẫn sử dụng các thanh ghi cơ sở khác nhau với các giá trị bằng nhau, bộ xử lý có thể không phát hiện đư ợc rằng các địa chỉ giống nhau ngay lập tức).

Giải pháp thay thế là biên dịch lại toàn bộ cây dấu vết, do đó đạt đư ợc phân bổ thanh ghi liên vết (10). Bất lợi là việc biên dịch lại cây đó mất thời gian bậc hai theo số vết. Chúng tôi tin rằng chi phí biên dịch lại cây dấu vết mỗi lần một nhánh đư ợc thêm vào sẽ bị cấm. Vấn đề đó có thể là giảm nhẹ bằng cách biên dịch lại chỉ ở một số điểm nhất định hoặc chỉ cho rất cây nóng, ổn định.

Trong tư ơng lai, phần cứng đa lõi dự kiến sẽ trở nên phổ biến, làm cho việc biên soạn lại cây nền trở nên hấp dẫn. Trong một dự án đư ợc kiểm soát chặt chẽ (13), việc biên dịch lại nền đã mang lại tốc độ lên đến 1,25 lần trên điểm chuẩn có nhiều dấu vết nhánh. Chúng tôi lên kế hoạch để áp dụng kỹ thuật này cho TraceMonkey như công việc trong tư ơng lai.

6.3 Ghi lại dấu vết

Công việc của bộ ghi theo dõi là phát ra LIR với ngữ nghĩa giống hệt nhau tới dấu vết bytecode của trình thông dịch hiện đang chạy. Một sự cố vấn tốt cần có tác động thấp đến trình thông dịch không truy tìm theo mỗi hình thức và là một cách thuận tiện cho những ngư ời thực hiện để duy trì sự tư ơng đư ơng giữa các phiên bản.

Trong quá trình triển khai của chúng tôi, sự sửa đổi trực tiếp duy nhất đối với inter preter là một cuộc gọi đến trình giám sát theo dõi ở các cạnh của vòng lặp. Trong tiêu chuẩn của chúng tôi kết quả (xem Hình 12) tổng thời gian sử dụng màn hình (cho tất cả hoạt động) thư ờng nhỏ hơn 5%, vì vậy chúng tôi coi trình thông dịch đáp ứng yêu cầu tác động. Việc tăng số lần truy cập vòng lặp là rất tốn kém bởi vì nó yêu cầu chúng tôi tìm kiếm vòng lặp trong bộ nhớ cache theo dõi, như ng chúng tôi đã điều chỉnh vòng lặp của mình để trở nên nóng hơn và theo dõi rất nhanh (ở lần lặp thứ hai). Việc triển khai bộ đếm lư ợt truy cập có thể là đư ợc cải thiện, điều này có thể mang lại cho chúng ta một sự gia tăng nhỏ về tổng thể tích cũng như linh hoạt hơn với các ngư ỡng độ nóng điều chỉnh. Khi một vòng lặp đư ợc đư a vào danh sách đen, chúng tôi không bao giờ gọi vào theo dõi theo dõi cho vòng lặp đó (xem Phần 3.3).

Việc ghi âm đư ợc kích hoạt bởi một hoán đổi con trỏ đặt bảng điều phối của inter preter để gọi một quy trình "ngắt" duy nhất cho mã bytecode của ev ery. Quy trình ngắt đầu tiên gọi một mã bytecode cụ thể ghi chép thư ờng xuyên. Sau đó, nó sẽ tắt ghi âm nếu cần (ví dụ: dấu vết kết thúc). Cuối cùng, nó chuyển sang phần thực thi mã byte thông dịch viên tiêu chuẩn. Một số mã byte có tác dụng trên bản đồ loại không thể đư ợc dự đoán trư ớc khi thực thi bytecode (ví dụ: gọi ing String.charCodeAt, trả về một số nguyên hoặc NaN nếu đối số chỉ mục nằm ngoài phạm vi). Đối với những điều này, chúng tôi sắp xếp để preter inter gọi lại vào bộ ghi sau khi thực hiện bytecode. Vì những móc như vậy tư ơng đối hiếm, chúng tôi nhúng chúng trực tiếp vào trình thông dịch, với một kiểm tra thời gian chạy bổ sung để xem liệu một máy ghi âm hiện đang hoạt động.

Trong khi việc tách trình thông dịch khỏi trình ghi làm giảm độ phức tạp của mã riêng biệt, nó cũng yêu cầu triển khai cẩn thận và thử nghiệm rộng rãi để đạt đư ợc sự tư ơng đư ơng về ngữ nghĩa.

Trong một số trư ờng hợp, việc đạt đư ợc sự tư ơng đư ơng này là khó khăn vì Spi derMonkey tuân theo một thiết kế theo mã phân tử (fat-bytecode), đư ợc phát hiện là có lợi cho hiệu suất thông dịch viên thuần túy.

Trong các thiết kế mã byte béo, các mã byte riêng lẻ có thể triển khai xử lý phức tạp (ví dụ: getprop bytecode, mã này cung cấp quyền truy cập giá trị thuộc tính JavaScript đầy đủ, bao gồm cả các trư ờng hợp đặc biệt để truy cập mảng đư ợc lư u trong bộ nhớ cache và dày đặc).

Các mã byte béo có hai ư u điểm: ít mã byte hơn có nghĩa là chi phí điều phối thấp hơn và triển khai bytecode lớn hơn mang lại trình biên dịch nhiều cơ hội hơn để tối ư u hóa trình thông dịch.

Các mã byte béo là một vấn đề đối với TraceMonkey vì chúng yêu cầu đầu ghi thực hiện lại cùng một logic trư ờng hợp đặc biệt theo cách tư ơng tự. Ngoài ra, các lợi thế bị giảm vì (a) chi phí vận chuyển đư ợc loại bỏ hoàn toàn trong các dấu vết tổng hợp, (b) dấu vết chỉ chứa một trư ờng hợp đặc biệt, không phải là trư ờng hợp lớn của trình thông dịch đoạn mã và (c) TraceMonkey dành ít thời gian hơn để chạy thông dịch viên cơ sở.

Một cách chúng tôi đã giảm thiểu những vấn đề này là bằng cách triển khai một số mã byte phức tạp trong máy ghi dư ới dạng chuỗi các mã đơn giản mã bytecodes. Diễn đạt ngữ nghĩa gốc theo cách này không quá khó và việc ghi lại các mã byte đơn giản dễ dàng hơn nhiều. Điều này cho phép chúng tôi giữ lại những ư u điểm của mã byte béo trong khi tránh một số vấn đề của họ để ghi lại dấu vết. Điều này đặc biệt hiệu quả đối với các mã byte béo sẽ đệ quy trở lại trình thông dịch, chẳng hạn như để chuyển đổi một đối tư ợng thành một giá trị nguyên thủy bằng cách gọi một trên đối tư ợng, vì nó cho phép chúng ta nội tuyến cuộc gọi hàm này.

Điều quan trọng cần lư u ý là chúng tôi chỉ chia các mã opcodes béo thành các mã op mỏng hơn trong quá trình ghi. Khi chạy thuần túy diễn giải (tức là mã đã đư ợc đư a vào danh sách đen), trình thông dịch trực tiếp và thực thi hiệu quả các mã opcodes béo.

6.4 Dự bị

SpiderMonkey, giống như nhiều máy ảo khác, cần truy cập trư ớc chư ơng trình ngư ời dùng định kỳ. Những lý do chính là để ngăn chặn vòng lặp vô hạn kịch bản khóa hệ thống máy chủ lư u trữ và lên lịch GC.

Trong trình thông dịch, điều này đã đư ợc thực hiện bằng cách đặt cờ "trư ớc hết ngay bây giờ" đư ợc kiểm tra trên mỗi lần nhảy lùi. Cái này chiến lư ợc đư ợc chuyển vào TraceMonkey: VM chèn một bảo vệ vào cờ ư u tiên ở mọi cạnh vòng lặp. Chúng tôi đo đư ợc ít hơn một Tăng 1% thời gian chạy trên hầu hết các điểm chuẩn cho bảo vệ bổ sung này. Trong thực tế, chi phí chỉ có thể phát hiện đư ợc đối với các chư ơng trình có thời gian rất ngắn các vòng lặp.

Chúng tôi đã thử nghiệm và từ chối một giải pháp tránh đư ợc các lính canh bằng cách biên dịch cạnh vòng lặp như một bư ớc nhảy vô điều kiện và vá mục tiêu nhảy sang một quy trình thoát khi yêu cầu ư u tiên. Giải pháp này có thể làm cho trư ờng hợp bình thư ờng nhanh hơn một chút, như ng sau đó preemption trở nên rất chậm. Việc thực hiện cũng rất phức tạp, đặc biệt là cố gắng khởi động lại quá trình thực thi sau khi preemption.

6.5 Gọi các chức năng bên ngoài Giống như

hầu hết các trình thông dịch, SpiderMonkey có một chức năng ngoại lai (FFI) cho phép nó gọi nội trang C và các chức năng của hệ thống máy chủ (ví dụ: điều khiển trình duyệt web và truy cập DOM). FFI có một chữ ký stan dard cho các hàm có thể gọi JS, đối số chính của nó là một mảng các giá trị đư ợc đóng hộp. Các hàm bên ngoài đư ợc gọi thông qua FFI tư ơng tác với trạng thái chư ơng trình thông qua một API thông dịch (ví dụ: để đọc một thuộc tính từ một đối số). Ngoài ra còn có một số nội trang trư ớc đó không sử dụng FFI, như ng tư ơng tác với trạng thái chư ơng trình theo cách tư ơng tự, chẳng hạn như hàm CallIteratorNext đư ợc sử dụng với các đối tư ợng trình vòng lặp. TraceMonkey phải hỗ trợ FFI này để tăng tốc mã tư ơng tác với hệ thống máy chủ bên trong các vòng lặp nóng.

Việc gọi các chức năng bên ngoài từ TraceMonkey có khả năng gây khó khăn vì dấu vết không cập nhật trạng thái trình thông dịch cho đến khi thoát nhập. Đặc biệt, các hàm bên ngoài có thể cần ngăn xếp cuộc gọi hoặc các biến toàn cục, như ng chúng có thể đã lỗi thời.

Đối với vấn đề ngăn xếp cuộc gọi lỗi thời, chúng tôi đã cấu trúc lại một số hàm triển khai API trình thông dịch để hiện thực hóa lại ngăn xếp cuộc gọi thông dịch theo yêu cầu.

Chúng tôi đã phát triển một phân tích tĩnh C ++ và chú thích một số hàm preter để xác minh rằng ngăn xếp cuộc gọi đư ợc làm mới tại bất kỳ thời điểm nào nó cần đư ợc sử dụng. Để truy cập ngăn xếp cuộc gọi, một hàm phải đư ợc chú thích là FORCESSTACK hoặc RE QUIRESSTACK. Các chú thích này cũng đư ợc yêu cầu để gọi các hàm REQUIRESSTACK , đư ợc cho là để truy cập tạm thời ngăn xếp cuộc gọi. FORCESSTACK là một chú thích đáng tin cậy, chỉ áp dụng cho 5 hàm, có nghĩa là hàm làm mới ngăn xếp cuộc gọi.

REQUIRESSTACK là một chú thích không đáng tin cậy có nghĩa là chức năng chỉ có thể đư ợc gọi nếu ngăn xếp cuộc gọi đã đư ợc làm mới.

Tư ơng tự như vậy, chúng tôi phát hiện khi các hàm máy chủ cố gắng đọc hoặc ghi trực tiếp các biến toàn cục và buộc dấu vết hiện đang chạy sang một bên thoát. Điều này là cần thiết vì chúng tôi lư u vào bộ nhớ cache và bỏ hộp các biến toàn cục vào bản ghi kích hoạt trong quá trình thực thi theo dõi.

Vì cả truy cập ngăn xếp cuộc gọi và truy cập biến toàn cục hiếm khi đư ợc thực hiện bởi các hàm máy chủ, nên hiệu suất không bị ảnh hư ởng đáng kể bởi các cơ chế an toàn này.

Một vấn đề khác là các hàm bên ngoài có thể nhập lại inter preter bằng cách gọi các tập lệnh, do đó lại có thể muốn truy cập ngăn xếp cuộc gọi hoặc các biến toàn cục. Để giải quyết vấn đề này, chúng tôi đã đặt máy ảo đặt cờ bất cứ khi nào trình thông dịch đư ợc nhập lại trong khi một dấu vết chất đống com đang chạy.

Mọi lệnh gọi đến một hàm bên ngoài sau đó sẽ kiểm tra cờ này và thoát khỏi dấu vết ngay lập tức sau khi quay lại từ lệnh gọi hàm bên ngoài nếu nó đư ợc thiết lập. Có nhiều chức năng bên ngoài hiếm khi hoặc không bao giờ nhập lại, và chúng có thể đư ợc gọi mà không có vấn đề gì, và sẽ chỉ thoát ra khỏi dấu vết khi cần thiết.

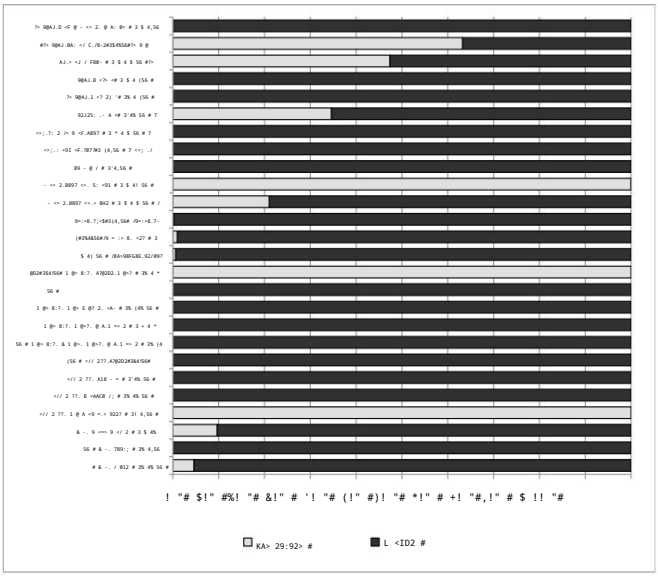
Yêu cầu mảng giá trị đóng hộp của FFI có chi phí hiệu suất, vì vậy chúng tôi đã xác định một FFI mới cho phép các hàm C đư ợc ký hiệu với các loại đối số của chúng để trình đánh dấu có thể gọi chúng trực tiếp mà không cần chuyển đổi đối số không cần thiết.

Hiện tại, chúng tôi không hỗ trợ gọi các hàm ghi đè thuộc tính gốc và đặt các hàm ghi đè hoặc các hàm DOM trực tiếp từ dấu vết. Hỗ trợ là công việc có kế hoạch trong tư ơng lai.

6.6 Tính đúng đắn

Trong quá trình phát triển, chúng tôi có quyền truy cập vào các bộ thử nghiệm JavaScript hiện có, như ng hầu hết chúng không đư ợc thiết kế với các máy ảo theo dõi và chứa ít vòng lặp.

Một công cụ đã giúp chúng tôi rất nhiều là trình thử nghiệm JavaScript fuzz của Mozilla, JSFUNFUZZ, tạo ra các chư ơng trình JavaScript ngẫu nhiên bằng cách lồng các phần tử ngôn ngữ ngẫu nhiên. Chúng tôi đã sửa đổi JSFUNFUZZ để tạo các vòng lặp và cũng để kiểm tra các cấu trúc nhất định nặng nề hơn mà chúng tôi nghi ngờ sẽ tiết lộ sai sót trong quá trình triển khai của chúng tôi. Đối với bài kiểm tra, chúng tôi nghi ngờ có lỗi trong việc xử lý kiểu không ổn định của TraceMonkey



Hình 11. Các phân đoạn của mã byte động đư ợc thực thi bởi inter preter và trên các dấu vết gốc. Tốc độ tăng tốc so với trình thông dịch đư ợc hiển thị trong dấu ngoặc đơn bên cạnh mỗi bài kiểm tra. Phần mã byte exe bị cắt trong khi ghi quá nhỏ để có thể nhìn thấy trong hình này, ngoại trừ crypto-md5, trong đó hoàn toàn 3% số byte đư ợc thực thi trong khi ghi. Trong hầu hết các bài kiểm tra, hầu hết tất cả các mã bytecodes đều đư ợc exe cắt bởi các dấu vết đã biên dịch. Ba trong số các điểm chuẩn hoàn toàn không đư ợc theo dõi và chạy trong trình thông dịch.

các vòng lặp và mã phân nhánh nhiều, và một trình kiểm tra lông tơ chuyên dụng trong chứng thư đã tiết lộ một số hồi quy mà sau đó chúng tôi đã sửa chữa.

7. Đánh giá

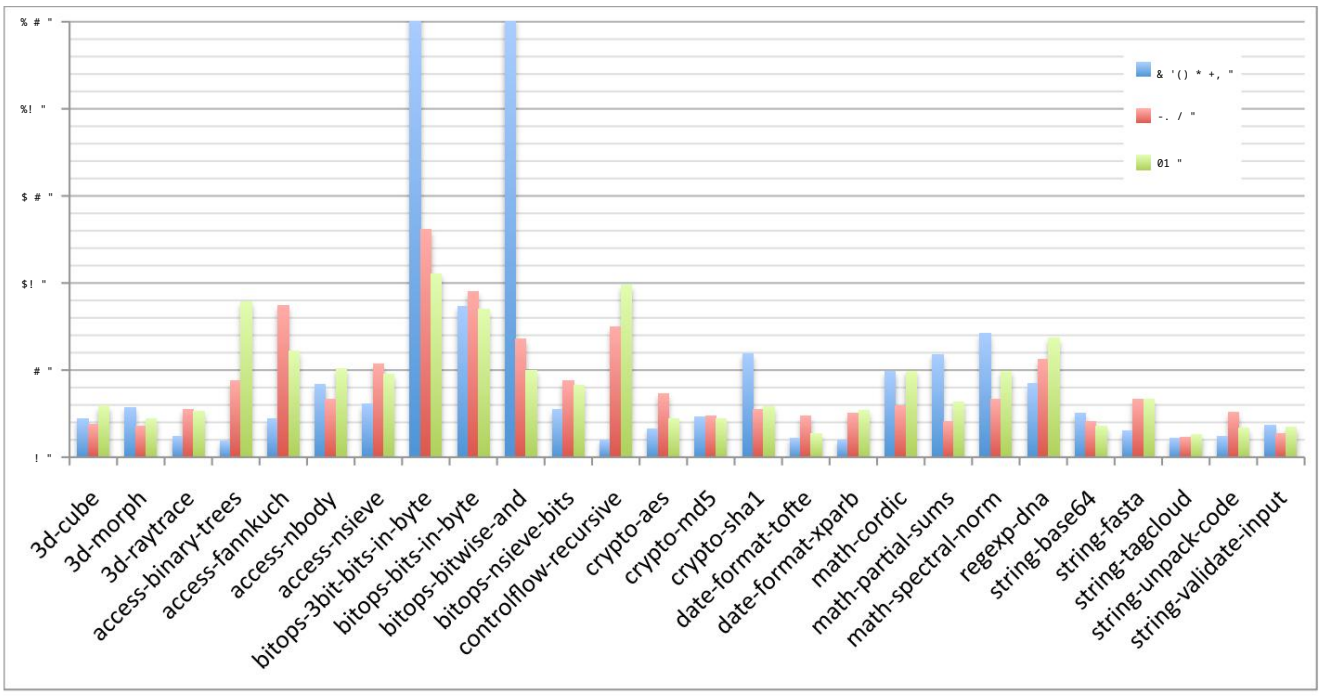
Chúng tôi đã đánh giá việc triển khai theo dõi JavaScript của mình bằng cách sử dụng Sun Spider, bộ tiêu chuẩn JavaScript tiêu chuẩn của ngành. SunSpi der bao gồm 26 lần chạy ngắn (dư ới 250ms, trung bình 26ms)
Các chư ơng trình JavaScript. Điều này hoàn toàn trái ngư ợc với các bộ điểm chuẩn như SpecJVM98 (3) đư ợc sử dụng để đánh giá máy ảo Java trên máy tính để bàn và máy chủ. Nhiều chư ơng trình trong các điểm chuẩn đó sử dụng tập dữ liệu lớn và thực thi trong vài phút. Các chư ơng trình SunSpider thực hiện nhiều tác vụ khác nhau, chủ yếu là kết xuất 3D, xử lý bit-bashing, mã hóa mật mã, hạt nhân toán học và xử lý chuỗi.

Tất cả các thử nghiệm đều đư ợc thực hiện trên MacBook Pro với 2.2 Bộ vi xử lý Core 2 GHz và RAM 2 GB chạy MacOS 10.5.

Kết quả điểm chuẩn. Câu hỏi chính là liệu các chư ơng trình có chạy nhanh hơn với tính năng theo dõi hay không. Đối với điều này, chúng tôi đã chạy trình điều khiển thử nghiệm SunSpider tiêu chuẩn, khởi động trình thông dịch JavaScript, tải và chạy mỗi chư ơng trình một lần để khởi động, sau đó tải và chạy mỗi chư ơng trình 10 lần và báo cáo thời gian trung bình đư ợc thực hiện bởi mỗi chư ơng trình. Chúng tôi đã chạy 4 cấu hình ent khác nhau để so sánh: (a) SpiderMonkey, trình thông dịch đư ờng cơ sở, (b) TraceMonkey, (d) SquirrelFish Extreme (SFX), trình thông dịch JavaScript theo chuỗi cuộc gọi đư ợc sử dụng trong WebKit của Apple và (e) V8, phư ơng pháp biên dịch máy ảo JavaScript từ Google.

Hình 10 cho thấy tốc độ tư ơng đối đạt đư ợc bằng cách truy tìm, SFX và V8 so với đư ờng cơ sở (SpiderMonkey). Theo dõi đạt đư ợc tốc độ tốt nhất trong các điểm chuẩn nặng về số nguyên, tốc độ tăng gấp 25 lần trên bitops-bitwise-and.

TraceMonkey là máy ảo nhanh nhất trên 9 trong số 26 điểm chuẩn (3d-morph, bitops-3bit-bit-in-byte, bitops-bitwise và, crypto-sha1, math-cordic, math-một phần-tổng, toán phổ-tiêu chuẩn, string-base64, string-validate-input).



Hình 10. Tăng tốc so với trình thông dịch JavaScript cơ bản (SpiderMonkey) cho trình biên dịch JIT dựa trên dấu vết của chúng tôi, trình thông dịch phân luồng nội tuyến SquirrelFish Extreme của Apple và trình biên dịch V8 JS của Google. Hệ thống của chúng tôi tạo mã đặc biệt hiệu quả cho các chương trình được hưởng lợi nhiều nhất từ chuyên môn hóa kiểu, bao gồm các chương trình SunSpider Benchmark thực hiện thao tác bit. Chúng tôi nhập chuyên biệt mã được đề cập để sử dụng số học số nguyên, điều này cải thiện đáng kể hiệu suất. Đối với một trong những chương trình điểm chuẩn, chúng tôi thực thi nhanh hơn 25 lần so với trình thông dịch SpiderMonkey và nhanh hơn gần 5 lần so với V8 và SFX. Đối với một số lượng lớn các điểm chuẩn, cả ba máy ảo đều cho kết quả tương tự. Chúng tôi thực hiện tồi tệ nhất trên các chương trình điểm chuẩn mà chúng tôi không theo dõi và thay vào đó rơi vào trình thông dịch. Điều này bao gồm điểm chuẩn đệ quy truy cập-cây nhị phân và điều khiển-luồng-đệ quy, mà chúng tôi hiện không tạo bất kỳ mã gốc nào.

Đặc biệt, các điểm chuẩn của bitops là các chương trình ngắn mà mỗi lần tạo thành nhiều phép toán bit, do đó, TraceMonkey có thể bao gồm 1 hoặc 2 dấu vết cho chương trình hoạt động trên số nguyên. Khóa TraceMon chạy tất cả các chương trình khác trong bộ này gần như hoàn toàn dưới dạng mã gốc. regexp-dna bị chi phối bởi đối sánh biểu thức chính quy, được triển khai trong cả 3 máy ảo bởi một trình biên dịch biểu thức chính quy đặc biệt. Do đó, hiệu suất trên điểm chuẩn này có rất ít liên quan đến cách tiếp cận tổng hợp dấu vết được thảo luận trong bài báo này.

Tốc độ nhỏ hơn của TraceMonkey trên các điểm chuẩn khác có thể do một số nguyên nhân cụ thể:

- Việc triển khai hiện không theo dõi đệ quy, vì vậy TraceMonkey đạt được tốc độ tăng nhỏ hoặc không tăng tốc trên các điểm chuẩn sử dụng đệ quy rộng rãi: 3d-cube, 3d raytrace, access-binary-tree, string-tagcloud và controlflow-recursive.

- Việc triển khai hiện không theo dõi eval và một số chức năng khác được thực hiện trong C. Vì tofte định dạng ngày và ngày-định dạng-xparb sử dụng các chức năng như vậy trong các vòng lặp chính của chúng, nên chúng tôi không theo dõi chúng.

- Việc triển khai hiện không theo dõi thông qua các hoạt động thay thế biểu thức chính quy. Hàm thay thế có thể được thông qua một đối tượng hàm được sử dụng để tính toán văn bản thay thế.
  Việc triển khai của chúng tôi hiện không theo dõi các chức năng được gọi là các chức năng thay thế. Thời gian chạy của string-unpack-code bị chi phối bởi một cuộc gọi thay thế như vậy.

- Hai chương trình theo dõi tốt, nhưng có thời gian biên dịch lâu. access-nbody tạo thành một số lượng lớn các dấu vết (81). crypto-md5 tạo thành một dấu vết rất dài. Chúng tôi hy vọng sẽ cải thiện hiệu suất trên các chương trình này bằng cách cải thiện tốc độ biên dịch của nano jit.

- Một số chương trình theo dõi rất tốt và tăng tốc so với trình thông dịch, nhưng không nhanh bằng SFX và / hoặc V8, cụ thể là bitops-bit-in-byte, bitops-nsieve-bit, access fannkuch, access-nsieve, và tiền điện tử. Lý do không rõ ràng, nhưng tất cả các chương trình này đều có các vòng lặp lồng nhau với các phần tử nhỏ, vì vậy chúng tôi nghi ngờ rằng việc triển khai có chi phí tương đối cao cho việc gọi các dấu vết lồng nhau. string-fasta theo dõi tốt, nhưng thời gian chạy của nó bị chi phối bởi các nội trang xử lý chuỗi, không bị ảnh hưởng bởi việc theo dõi và dường như kém hiệu quả hơn trong SpiderMonkey so với hai máy ảo khác.

Chỉ số hiệu suất chi tiết. Trong Hình 11, chúng tôi chỉ ra một phần nhỏ các hướng dẫn được thông dịch và một phần các lệnh exe được cắt thành mã gốc. Hình này cho thấy rằng đối với nhiều chương trình, chúng ta có thể thực thi hầu như tất cả các mã nguyên bản.

Hình 12 chia tổng thời gian thực thi thành bốn thời gian hoạt động: giải thích các mã byte trong khi không ghi, ghi lại dấu vết (bao gồm cả thời gian thực hiện để diễn giải dấu vết đã ghi), biên dịch dấu vết thành mã gốc và thực thi dấu vết mã gốc.

Các chỉ số chi tiết này cho phép chúng tôi ước tính các thông số cho một mô hình đơn giản về hiệu suất theo dõi. Những ước tính này nên được coi là rất thô, vì các giá trị được quan sát trên các điểm chuẩn riêng lẻ có độ lệch chuẩn lớn (theo thứ tự của

	Dấu vết cây lặp lại	Bỏ qua cây / Dấu vết vòng / Dấu vết cây / Tăng tốc vòng lặp							
3d-cube 3d-	25	27	29	3	0	1,1	1,1	1,2	2,20x
morph 3d-	5	8	8	2	0	1,6	1,0	1,6	2,86x
raytrace	10	25	100	10	1	2,5	4,0	10,0	1,18x
access-binary-tree access-	0	0	0	5	0	-	-	-	0,93x
fannkuch access-nbody	10	34	57	24	0	3,4	1,7	5,7	2,20x
access-nsieve bitops-3bit-	8	16	18	5	0	2,0	1,1	2,3	4,19x
bit-in-byte bitops-bit-	3	6	8	3	0	2,0	1,3	2,7	3.05x
in-byte bitops-bitwise-and	2	2	2	0	0	1,0	1,0	1,0	25,47x
bitops-nsieve-bits controlflow	3	3	4	1	0	1,0	1,3	1,3	8,67x
-recursive crypto-aes crypto-	1			0	0	1,0	1,0	1,0	25,20x
md5 crypto-sha1 date-format-	3	1	1	0	0	1,0 1,0	1,7	1,7	2,75x
tofte date-format-xparb math-	0	3	5	1	0	-	-	-	0,98x
cordic math-một phần-tính	50	0	0	19	0	1,4	1,1	1,6	1,64x
toán toán-phổ-chỉ tiêu regexp-	4			0	0	1,0	1,3	1,3	2,30x
dna string-base64 string-	5		78	0	0	1,0	2,0	2,0	5,95x
fasta string-tagcloud string-	3		5	7	0	1,0	1,3	1,3	1,07x
unpack- chuỗi mã xác thực-đầu	3		10	3	0	1,0	3,7	3,7	0,98x
vào	2		4	1	0	2,0	1,3	2,5	4,92x
	2			1	0	2,0	1,0	2,0	5,90x
	15	72	11	0	0	1,3	1,0	1,3	7,12x
	2	4	5	0	0	1,0	1,0	1,0	4,21x
	3	5	4	0	0	1,7	1,4	2,3	2,53x
	5	3	20	6	0	2,2	1,4	3,0	1,49x
	3	3	2	5	0	2,0	1,0	2,0	1,09x
	4	4	7	0	0	1,0	9,3	9,3	1,20x
	6	4 20 2 5 11 6 4 10 6 37 13		1	0	1,7	1,3	2,2	1,86x

Hình 13. Số liệu thống kê ghi lại dấu vết chi tiết cho bộ tiêu chuẩn SunSpider.

bần tiện). Chúng tôi loại trừ regexp-dna khỏi các tính toán sau, bởi vì phần lớn thời gian của nó được dành cho trình so khớp biểu thức chính quy, có nhiều đặc điểm hiệu suất khác với các chương trình khác. (Lưu ý rằng điều này chỉ tạo ra sự khác biệt về 10% trong kết quả.) Chia tổng thời gian thực thi trong bộ xử lý chu kỳ đồng hồ theo số lượng mã byte được thực thi trong cơ sở trình thông dịch cho thấy rằng trung bình, một bytecode thực thi trong khoảng 35 chu kỳ. Các dấu vết gốc mất khoảng 9 chu kỳ cho mỗi byte, một 3,9 lần tăng tốc độ qua trình thông dịch.

Sử dụng các phép tính tương tự, chúng tôi thấy rằng việc ghi lại dấu vết cần khoảng 3800 chu kỳ mỗi bytecode và biên dịch 3150 chu kỳ mỗi mã bytecode. Do đó, trong quá trình ghi và biên dịch, máy ảo chạy ở 1/200 tốc độ của trình thông dịch. Bởi vì nó tốn 6950 chu kỳ để biên dịch một mã bytecode và chúng tôi lưu 26 chu kỳ mỗi lần mã đó chạy tự nhiên, chúng tôi hòa vốn sau khi chạy một dấu vết 270 lần.

Các máy ảo khác mà chúng tôi so sánh đạt được tốc độ tổng thể 3,0 lần so với trình thông dịch cơ sở của chúng tôi. Bản địa ước tính của chúng tôi tốc độ mã 3,9x tốt hơn đáng kể. Điều này cho thấy rằng các kỹ thuật biên dịch của chúng tôi có thể tạo ra mã gốc hiệu quả hơn hơn bất kỳ máy ảo JavaScript hiện tại nào khác.

Những ước tính này cũng chỉ ra rằng hiệu suất khởi động của chúng tôi có thể về cơ bản sẽ tốt hơn nếu chúng tôi cải thiện tốc độ ghi lại dấu vết và biên dịch. Thời gian chậm lại ước tính 200 lần để ghi và quá trình biên dịch rất khó và có thể bị ảnh hưởng bởi các yếu tố khởi động trong trình thông dịch (ví dụ: bộ nhớ đệm chưa nóng lên trong ghi âm). Một quan sát hỗ trợ phỏng đoán này là trong trình đánh dấu, mã byte được thông dịch mất khoảng 180 chu kỳ để chạy. Còn, ghi âm và biên dịch rõ ràng là cả hai đều đắt tiền và tốt hơn triển khai, có thể bao gồm cả thiết kế lại phần tóm tắt LIR cú pháp hoặc mã hóa, sẽ cải thiện hiệu suất khởi động.

Kết quả hoạt động của chúng tôi xác nhận rằng loại chuyên môn hóa bằng cách sử dụng cây dấu vết cải thiện đáng kể hiệu suất. Chúng ta có thể vượt trội so với trình biên dịch JavaScript hiện có nhanh nhất (V8) và Trình thông dịch chuỗi nội tuyến JavaScript khả dụng nhanh nhất (SFX) trên 9 trong tổng số 26 điểm chuẩn.

8. Công việc liên quan

Tối ưu hóa theo dõi cho các ngôn ngữ động. Khu vực gần nhất của công việc liên quan là áp dụng tối ưu hóa theo dõi cho loại chuyên môn hóa ngôn ngữ động. Công việc hiện tại chia sẻ ý tưởng tạo ra loại mã chuyên biệt một cách cụ thể với các bộ phận bảo vệ cùng với trình thông dịch dấu vết.

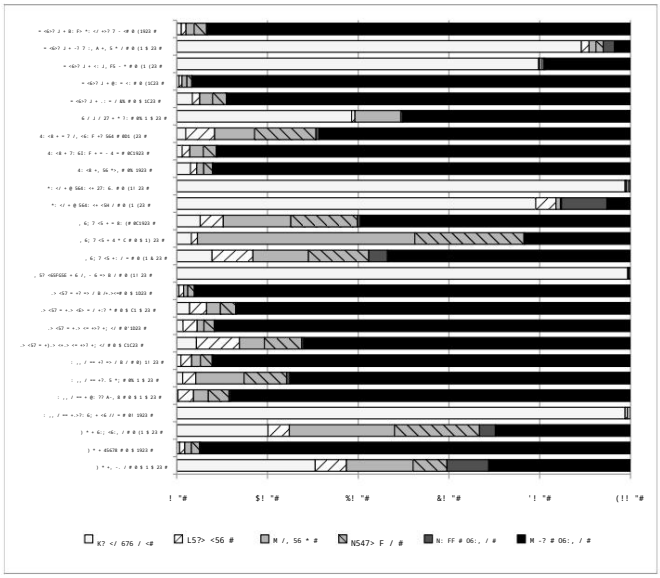
Theo hiểu biết của chúng tôi, Psyco của Rigo (16) là tác phẩm duy nhất được xuất bản trình biên dịch theo dõi chuyên về kiểu cho một ngôn ngữ động (Python). Psyco không cố gắng xác định các vòng lặp nóng hoặc các lệnh gọi hàm nội tuyến. Thay vào đó, Psyco biến đổi các vòng lặp thành đệ quy lẫn nhau trước khi chạy và theo dõi tất cả các hoạt động.

Pall's LuaJIT là một máy ảo Lua đang được phát triển sử dụng các ý tưởng thử nghiệm theo dõi com. (1). Không có ấn phẩm nào về LuaJIT nhưng cre ator đã cho chúng tôi biết rằng LuaJIT có thiết kế tương tự với hệ thống của chúng tôi, nhưng sẽ sử dụng kiểu suy đoán ít tích cực hơn (ví dụ: sử dụng biểu diễn dấu phẩy động cho tất cả các giá trị số) và không tạo ra dấu vết lồng nhau cho các vòng lặp lồng nhau.

Tối ưu hóa dấu vết chung. Tối ưu hóa dấu vết chung có một lịch sử lâu hơn đã xử lý hầu hết là mã gốc và đã nhập ngôn ngữ như Java. Do đó, các hệ thống này đã tập trung ít hơn vào loại chuyên môn hóa và hơn thế nữa về các tối ưu hóa khác.

Dynamo (7) của Bala và cộng sự, đã giới thiệu tính năng theo dõi mã gốc như một thay thế cho tối ưu hóa hướng dẫn hồ sơ (PGO). Một mục tiêu chính là thực hiện PGO trực tuyến để hồ sơ cụ thể cho việc thực hiện hiện tại. Dynamo đã sử dụng tiêu đề vòng lặp làm ứng cử viên nóng dấu vết, nhưng không cố gắng tạo dấu vết vòng lặp một cách cụ thể.

Cây dấu vết ban đầu được đề xuất bởi Gal et al. (11) trong ngữ cảnh của Java, một ngôn ngữ được định kiểu tĩnh. Các cây dấu vết của chúng xoay chiều các bộ phận được sắp xếp thẳng hàng của các vòng bên ngoài trong các vòng bên trong (bởi vì



Hình 12. Một phần thời gian dành cho các hoạt động chính của VM. Các
speedup so với trình thông dịch được hiển thị trong dấu ngoặc đơn bên cạnh mỗi bài kiểm tra.
Hầu hết các chương trình mà VM dành phần lớn thời gian để chạy mã gốc đều có tốc
độ tốt. Ghi âm và biên soạn
chi phí có thể là đáng kể; tăng tốc các phần của quá trình thử nghiệm sẽ cải thiện
hiệu suất của SunSpider.

vòng bên trong trở nên nóng trước), dẫn đến phần đuôi trùng lặp lớn hơn nhiều.

YETI, từ Zaleski et al. (19) theo dõi kiểu Dynamo được áp dụng
sang Java để đạt được nội tuyến, loại bỏ bước nhảy gián tiếp,
và các tối ưu hóa khác. Trọng tâm chính của họ là thiết kế một
trình thông dịch có thể dễ dàng được thiết kế lại dần dần như một lần theo dõi
Máy ảo.

Suganuma và cộng sự. (18) tổng hợp dựa trên khu vực được mô tả (RBC),
một người thân của truy tìm. Vùng là một chương trình con đáng để tối ưu hóa
có thể bao gồm các tập con của bất kỳ số lượng phương thức nào. Do đó, com piler
có tính linh hoạt hơn và có khả năng tạo mã tốt hơn,
như ng hệ thống biên dịch và biên dịch tương ứng nhiều hơn
phức tạp.

Nhập chuyên môn hóa cho các ngôn ngữ động. Những người triển khai động vật lan
từ lâu đã nhận ra tầm quan trọng của loại
chuyên môn hóa cho hiệu suất. Hầu hết các công việc trước đây đều tập trung vào
thay vì dấu vết.

Chambers et. al (9) đi tiên phong trong ý tưởng biên dịch nhiều
phiên bản của một thủ tục chuyên biệt cho các loại đầu vào trong Lan guage Self.
Trong một lần triển khai, họ đã tạo ra một
phương thức trực tuyến mỗi khi một phương thức được gọi với các kiểu đầu vào mới.
Trong một cách khác, họ đã sử dụng phân tích tĩnh toàn bộ chương trình ngoại tuyến để
suy ra kiểu đầu vào và kiểu máy thu không đổi tại các điểm gọi. Thật thú vị, hai kỹ
thuật tạo ra hiệu suất gần như giống nhau.

Salib (17) đã thiết kế một thuật toán suy luận kiểu cho dựa trên Python
trên Thuật toán sản phẩm Descartes và sử dụng kết quả để đánh dấu đặc biệt trên các
loại và dịch chương trình sang C ++.

McCloskey (14) có công việc đang được tiến hành dựa trên suy luận kiểu độc lập
với ngôn ngữ được sử dụng để tạo ra C hiệu quả
triển khai các chương trình JavaScript và Python.

Tạo mã gốc bởi thông dịch viên. Thiết kế inter preter truyền thống là một máy
ảo thực thi trực tiếp các AST hoặc
mã bytecodes giống mã máy. Các nhà nghiên cứu đã chỉ ra cách gen
tạo mã gốc với cấu trúc gần giống nhau như ng hiệu suất tốt hơn
mance.

Luồng cuộc gọi, còn được gọi là luồng ngữ cảnh (8), biên dịch
các phương pháp bằng cách tạo một lệnh gọi riêng cho một trình thông dịch
phương pháp cho từng mã byte thông dịch viên. Một cặp cuộc gọi-trả lại đã được
được chứng minh là một cơ chế điều phối tiềm năng hiệu quả hơn nhiều
so với các bước nhảy gián tiếp được sử dụng trong trình thông dịch bytecode tiêu chuẩn.

Phân luồng nội tuyến (15) sao chép các đoạn mã gốc của trình thông dịch
triển khai các mã bytecodes được yêu cầu vào một bộ đệm mã riêng,
do đó hoạt động như một trình biên dịch JIT đơn giản cho mỗi phương thức giúp loại bỏ
chi phí vận chuyển.

Luồng cuộc gọi và luồng nội tuyến đều không thực hiện kiểu ký tự đặc biệt.

Apple SquirrelFish Extreme (5) là một triển khai JavaScript dựa trên phân luồng
cuộc gọi với phân luồng nội tuyến có chọn lọc. Được kết hợp với kỹ thuật thông
dịch hiệu quả, các kỹ thuật phân luồng này đã mang lại cho SFX hiệu suất tuyệt vời
trên các điểm chuẩn Sun Spider tiêu chuẩn.

V8 của Google là một triển khai JavaScript chủ yếu dựa trên
về phân luồng nội tuyến, với phân luồng cuộc gọi chỉ dành cho rất phức tạp
các hoạt động.

9. Kết luận

Bài báo này mô tả cách chạy các ngôn ngữ động một cách hiệu quả bằng cách
ghi lại các dấu vết nóng và tạo mã gốc chuyên biệt về loại.
Kỹ thuật của chúng tôi tập trung vào các vòng lặp nội tuyến tích cực và cho mỗi
vòng lặp, nó tạo ra một cây các dấu vết mã gốc đại diện cho
các đường dẫn và kiểu giá trị thông qua vòng lặp được quan sát tại thời điểm chạy. chúng tôi
đã giải thích cách xác định các mối quan hệ lồng vào vòng lặp và tạo
dấu vết lồng nhau để tránh trùng lặp mã quá mức do
đến nhiều con đường thông qua một tổ vòng lặp. Chúng tôi đã mô tả loại của chúng tôi
thuật toán chuyên môn hóa. Chúng tôi cũng đã mô tả trình biên dịch theo dõi của chúng tôi,
dịch một dấu vết từ một đại diện trung gian sang
mã gốc được tối ưu hóa trong hai lần chuyển tuyến tính.

Kết quả thử nghiệm của chúng tôi cho thấy rằng trong thực tế, các vòng lặp thường
được nhập chỉ với một số kết hợp khác nhau của các loại giá trị
của các biến. Do đó, một số lượng nhỏ dấu vết trên mỗi vòng lặp là đủ
để chạy một chương trình hiệu quả. Các thử nghiệm của chúng tôi cũng cho thấy rằng trên
các chương trình có thể truy tìm, chúng tôi đạt được tốc độ từ 2 lần đến 20 lần.

10. Công việc trong tương lai

Công việc đang được tiến hành trong một số lĩnh vực để cải thiện hơn nữa
hiệu suất của trình biên dịch JavaScript dựa trên dấu vết của chúng tôi. Hiện tại chúng tôi
không theo dõi qua các lệnh gọi hàm đệ quy, như ng có kế hoạch thêm
hỗ trợ cho khả năng này trong thời gian tới. Chúng tôi cũng đang khám phá
áp dụng công việc hiện có về biên dịch lại cây trong bối cảnh
của trình biên dịch động được trình bày để giảm thiểu việc tạm dừng JIT
thời gian và có được những gì tốt nhất của cả hai thế giới, khâu cây cũng nhanh chóng
vì chất lượng mã được cải thiện do biên dịch lại cây.

Chúng tôi cũng có kế hoạch bổ sung hỗ trợ truy tìm trên các thay thế ex pression
thông thường bằng cách sử dụng các hàm lambda, các ứng dụng hàm và đánh giá biểu
thức bằng eval. Tất cả những ngôn ngữ này
các cấu trúc hiện đang được thực thi thông qua diễn giải, điều này giới hạn
hiệu suất của chúng tôi đối với các ứng dụng sử dụng các tính năng đó.

Sự nhìn nhận

Các phần của nỗ lực này đã được tài trợ bởi Khoa học Quốc gia
Tổ chức dưới sự tài trợ CNS-0615443 và CNS-0627747,
như của Chương trình MICRO California và nhà tài trợ công nghiệp Sun
Hệ thống vi mô thuộc Dự án số 07-127.

không được hiểu là nhất thiết phải đại diện cho các quan điểm, chính sách hoặc xác nhận chính thức, được thể hiện hoặc ngụ ý, của Na tional Science Foundation (NSF), bất kỳ cơ quan nào khác của Chính phủ Hoa Kỳ hoặc bất kỳ công ty nào được đề cập ở trên.

Ngươi giới thiệu

[1] Lộ trình LuaJIT 2008 - http://lua-users.org/lists/lua-l/2008- 02 / msg00051.html.

[2] Mozilla - trình duyệt web Firefox và ứng dụng email Thunderbird - http://www.mozilla.com.

[3] SPECJVM98 - http://www.spec.org/jvm98/.

[4] SpiderMonkey (JavaScript-C) http:// Động cơ - www.mozilla.org/js/spidermonkey/.

[5] Surfin 'Safari - Lưu trữ blog - Thông báo SquirrelFish Extreme - http:// webkit.org/blog/214/introductioning-squirrelfish-extreme/.

[6] A. Aho, R. Sethi, J. Ullman, và M. Lam. Ngươi biên dịch: Nguyên tắc, kỹ thuật và công cụ, 2006.

[7] V. Bala, E. Duesterwald và S. Banerjia. Dynamo: Một hệ thống tối ưu hóa động minh bạch. Trong Kỷ yếu của Hội nghị ACM SIGPLAN về Thiết kế và Triển khai Ngôn ngữ Lập trình, trang 1–12. ACM Press, 2000.

[8] M. Berndl, B. Vitale, M. Zaleski và A. Brown. Phân luồng theo ngữ cảnh: Kỹ thuật điều phối linh hoạt và hiệu quả cho máy ảo trong trình phiên dịch. Trong Tạo mã và Tối ưu hóa, 2005. CGO 2005. Hội nghị chuyên đề quốc tế trên, trang 15–26, 2005.

[9] C. Chambers và D. Ungar. Tùy chỉnh: Tối ưu hóa Công nghệ Trình biên dịch cho SELF, một Ngôn ngữ vẽ đồ họa chuyên nghiệp định hướng đối tượng O được đánh máy động. Trong Kỷ yếu của Hội nghị ACM SIGPLAN 1989 về Thiết kế và Triển khai Ngôn ngữ Lập trình, trang 146–160. ACM New York, NY, Hoa Kỳ, 1989.

[10] A. Gal. Xác minh và biên dịch mã Bytecode hiệu quả trong một luận văn về máy ảo. Luận án Tiến sĩ, Đại học California, Irvine, 2006.

[11] A. Gal, CW Probst, và M. Franz. HotpathVM: Một trình biên dịch JIT hiệu quả cho các thiết bị hạn chế tài nguyên. Trong Kỷ yếu của Hội nghị Quốc tế về Môi trường Thực thi Ảo, trang 144–153. ACM Press, 2006.

[12] C. Garrett, J. Dean, D. Grove, và C. Chambers. Đo lường và Ứng dụng của phân phối lớp máy thu động. Năm 1994.

[13] J. Ha, MR Haghighat, S. Cong, và KS McKinley. Trình biên dịch chỉ trong thời gian dựa trên dấu vết đồng thời cho javascript. Khoa Khoa học Máy tính, Đại học Texas tại Austin, TR-09-06, 2009.

[14] B. McCloskey. Sự giao tiếp cá nhân.

[15] I. Piumarta và F. Riccardi. Tối ưu hóa mã luồng trực tiếp bằng cách nội tuyến chọn lọc. Trong Kỷ yếu của hội nghị ACM SIGPLAN 1998 về thiết kế và triển khai ngôn ngữ lập trình, trang 291– 300. ACM New York, NY, USA, 1998.

[16] A. Rigo. Chuyên môn đúng lúc dựa trên đại diện và Nguyên mẫu Psyco cho Python. Trong PEPM, 2004.

[17] M. Salib. Starkiller: Một Trình biên dịch và Inferencer Loại Tĩnh cho Python. Trong Luận văn Thạc sĩ, 2004.

[18] T. Suganuma, T. Yasue và T. Nakatani. Một kỹ thuật tổng hợp dựa trên khu vực cho trình biên dịch động. Giao dịch ACM trên Hệ thống và Ngôn ngữ vẽ đồ thị chuyên nghiệp (TOPLAS), 28 (1): 134–174, 2006.

[19] M. Zaleski, AD Brown, và K. Stoodley. YETI: Trình thông dịch dấu vết mở rộng gradualLY. Trong Kỷ yếu của Hội nghị Quốc tế về Môi trường Thực thi Ảo, trang 83–93. ACM Press, 2007.