UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JOÃO FELIPE PORTO DE ALBUQUERQUE

ANÁLISE ESTÁTICA DE CÓDIGO PARA DETECÇÃO DE CERTAS CONDIÇÕES DE CORRIDA

RIO DE JANEIRO
2019

JOÃO FELIPE PORTO DE ALBUQUERQUE

ANÁLISE ESTÁTICA DE CÓDIGO PARA DETECÇÃO DE CERTAS CONDIÇÕES DE CORRIDA

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profa. Silvana Rossetto

RIO DE JANEIRO

2019

CIP - Catalogação na Publicação

A345a

Albuquerque, João Felipe Porto de
Análise estática de código para detecção de certas condições de corrida / João Felipe Porto de Albuquerque. -- Rio de Janeiro, 2019.
43 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2019.

1. Programação concorrente. 2. Análise estática de código. 3. Condições de corrida. 4. C/Pthread. I. Rossetto, Silvana, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

JOÃO FELIPE PORTO DE ALBUQUERQUE

ANÁLISE ESTÁTICA DE CÓDIGO PARA DETECÇÃO DE CERTAS CONDIÇÕES DE CORRIDA

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 26 de julho de 2019

BANCA EXAMINADORA:

Silvana Rossetto
D.Sc. (DCC-UFRJ)

Daniel Sadoc Menasche
D.Sc. (DCC-UFRJ)

Geraldo Zimbrão da Silva
D.Sc. (DCC-UFRJ)

AGRADECIMENTOS

Agradeço à minha família e à minha namorada, Nathalia, pelo apoio ao longo desta trajetória. Agradeço à UFRJ pelas experiências e aprendizados que pude ter ao longo dessa graduação. Agradeço aos professores do Departamento de Ciência da Computação por tudo que foi ensinado e aprendido durante esses anos, em especial a Silvana Rossetto, que me orientou no presente trabalho, pela paciência e pelos conselhos.

“ Race, what is that? Race is a competition,
somebody winning and somebody losing."

Beah Richards

RESUMO

A programação concorrente se mostra cada vez mais presente em diversas áreas da computação. Junto com o crescimento dessa área, a presença de ferramentas capazes de auxiliar no aprendizado e no desenvolvimento de aplicações concorrentes se mostra cada vez maior. O presente trabalho tem como objetivo criar uma ferramenta direcionada aos alunos da disciplina de Computação Concorrente (DCC/UFRJ), capaz de auxiliá-los na identificação de possíveis condições de corrida com variáveis globais e estáticas. A partir de uma análise estática do código fornecido pelo aluno, a ferramenta se propõe a identificar quais são as variáveis passíveis de sofrer uma condição de corrida e retornar essa informação ao aluno. Com esse conhecimento em mãos, espera-se que o aluno reveja o código com olhos mais críticos, prestando atenção especial às variáveis citadas, de forma a fortalecer os conhecimentos de lógica de programação de forma geral, e, em particular, dos mecanismos de sincronização usados na programação concorrente.

Palavras-chave: Programação Concorrente. Análise estática de código. Condições de corrida. C/Pthread.

ABSTRACT

Concurrent programming is increasingly present in several areas of computer science and software development. Along with the growth of this area, the presence of tools capable of assisting in both the learning and development of concurrent applications is more and more important. The present study aims to create a tool that is able to assist students of the discipline of Concurrent Computation (DCC / UFRJ) in identifying possible data races with global and static variables. From a static analysis of the code provided by the student, the tool strives to identify which variables can suffer a data race and provide this information to the student. With this knowledge in hand, the student is expected to review the code with more critical eyes, paying special attention to the variables mentioned by the program, in order to strengthen the student's knowledge of programming logic in general, and, in particular, synchronization mechanisms used in concurrent programming.

Keywords: Concurrent Programming. Static code analysis. Data races. C/Pthreads

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

GPU	Graphical Processing Unit
TPU	Tensor Processing Unit

LISTA DE SÍMBOLOS

π	Letra grega Pi

SUMÁRIO

1 INTRODUÇÃO

Concorrência é uma área que vem ganhando cada vez mais espaço na computação. Com o surgimento e popularização de processadores multi-core, GPUs (*Graphical Processing Units*), TPUs (*Tensor Processing Units*), entre outros, torna-se cada vez mais necessário escrever programas concorrentes (ao invés de programas sequenciais) para que o potencial dessas novas tecnologias seja de fato explorado. Por outro lado, existem aplicações que são naturalmente concorrentes, como por exemplo, aplicações gráficas. Nesse caso, a programação concorrente pode facilitar o trabalho do desenvolvedor.

De forma simples, um programa concorrente começa com um único fluxo de execução (thread), igual a um programa sequencial. A partir de certo ponto da execução do programa, um ou mais fluxos de execução independentes são criados, separando assim as tarefas do programa em fluxos que podem ser executados concorrentemente, cada um com seu contexto próprio de execução.

Quando existe necessidade de interação entre dois ou mais fluxos de execução de um programa concorrente, é preciso usar mecanismos de comunicação entre eles. Uma das soluções mais comuns para isso (no caso de linguagens que não são concorrentes, como é o caso de C/PThreads) é usar a memória compartilhada do programa para ler e escrever dados em variáveis de escopo global. Essa solução cria a necessidade de se usar mecanismos de sincronização de código por exclusão mútua para evitar que dois ou mais fluxos de execução tentem acessar a mesma variável global ao mesmo tempo, causando um problema conhecido como **condição de corrida (data races)**.

1.1 MOTIVAÇÃO

Especialmente durante seu aprendizado, a computação concorrente se mostra cheia de desafios. A dificuldade dos alunos para desenvolver o pensamento concorrente, e, ao mesmo tempo, lidar com as particularidades da sua programação faz com que muitos programas apresentem erros básicos de acesso não sincronizado a áreas de memória compartilhada por mais de um fluxo de execução (levando ao problema de condição de corrida).

Infelizmente, a maioria das ferramentas que permitem detecção de erros ou de falhas lógicas de programação foram pensados para programas sequenciais, e não se aplicam à lógica dos programas concorrentes. Fica, portanto, a cargo do desenvolvedor garantir que o programa não possui nenhum tipo de erro lógico ou falta de sincronização. Isso se mostra especialmente complicado quando o mesmo ainda não se habituou a encontrar e tratar esses erros.

Entretanto, já existem diversas iniciativas que procuram atacar este problema em

diferentes linguagens. Rust (DEVELOPERS, 2019) e Go (GOOGLE, 2019), por exemplo, são linguagens que possuem intrinsecamente alguns recursos para tratar condições de corrida. Rust elimina as condições de corrida em tempo de execução ao exigir que exista no máximo uma referência mutável para qualquer variável existente. Com isso, para que qualquer fluxo consiga modificar aquela variável é preciso que ela não tenha referência mutável aberta em nenhum outro fluxo (KLABNIK, 2018). Já Go possui um detector de condições de corrida que funciona em tempo de execução, podendo ser utilizado para rodar testes ou criar um executável com a verificação embutida e executá-lo diversas vezes para garantir a ausência da condição de corrida (GOOGLE, 2019).

Outras linguagens, como por exemplo Java, possuem ferramentas externas para auxiliar os desenvolvedores de programas concorrentes, como o RacerD, desenvolvido pelo Facebook como parte do projeto Infer. A ferramenta analisa estaticamente o código, não com o propósito de provar a ausência de condições de corrida, mas para buscar com alta confiança condições de corrida entre classes que deveriam ser seguras (indicadas pela anotação `@ThreadSafe`) (BLACKSHEAR et al., 2018).

A linguagem C também possui ferramentas de auxílio à programação concorrente, como por exemplo a ferramenta Helgrind. O algoritmo do Helgrind é conceitualmente muito simples e funciona de forma similar ao programa desenvolvido no presente trabalho. Ele monitora todos os acessos à memória. Se um determinado local é acessado em mais de um fluxo de execução, o Helgrind verifica a ordem de criação e junção dos fluxos para ver se o acesso pode acontecer ao mesmo tempo ou não (DEVELOPERS, c2019).

No curso de Ciência da Computação da UFRJ temos uma disciplina de introdução à programação concorrente. Nessa disciplina adota-se a linguagem C e a biblioteca PThreads. Passados alguns anos de oferta dessa disciplina, foi possível observar as dificuldades recorrentes e comuns dos alunos para detectar problemas básicos de condição de corrida em seus programas; e a carência de ferramentas de apoio para auxiliá-los.

1.2 OBJETIVO

O propósito deste trabalho é desenvolver uma ferramenta para detecção de condições de corrida em programas concorrentes escritos em C e com funções da biblioteca PThreads, com a finalidade de auxiliar o aprendizado da programação concorrente por alunos de graduação da UFRJ. Nesse sentido, a ferramenta implementada deverá apresentar os resultados ao aluno de forma a não apenas indicar possíveis condições de corrida, mas também auxiliá-lo no seu aprendizado, incentivando o pensamento crítico. Ao não apontar diretamente onde se encontra o problema, mas apenas indicar a presença do mesmo, espera-se que o aluno consiga ele mesmo analisar seu código e encontrar o problema apontado pela ferramenta.

1.3 METODOLOGIA

O projeto foi desenvolvido com uma metodologia similar às metodologias ágeis, criando pequenos pacotes incrementais da solução proposta. As principais etapas do trabalho incluíram as seguintes tarefas: (i) leitura do programa e identificação das variáveis globais e estáticas; (ii) criação de sumários para as funções mostrando onde as variáveis globais e estáticas são acessadas para leitura ou escrita; (iii) junção dos sumários para identificar possíveis condições de corrida e, por fim; (iv) avaliação da presença dos *locks* (mecanismo de sincronização dos diferentes fluxos de execução) para evitar condições de corrida no código.

Durante o desenvolvimento do trabalho e na etapa de avaliação foram usados exemplos de programas desenvolvidos por alunos da disciplina Computação Concorrente (DCC/UFRJ) para verificar se a ferramenta desenvolvida estava correta (de acordo com o escopo definido), bem como para avaliar o quão útil ela se mostrava para detectar condições de corrida em programas escritos por estudantes que estão aprendendo programação concorrente.

1.4 ORGANIZAÇÃO DO TEXTO

O restante deste texto está organizado da seguinte forma. No Capítulo 2, serão apresentados os conceitos básicos necessários para melhor entendimento do trabalho proposto. O Capítulo 3 traz a motivação para o problema tratado, bem como a proposta de solução para o mesmo. Já o Capítulo 4 apresenta as principais decisões de implementação da solução proposta e as ferramentas auxiliares que foram utilizadas nessa etapa. O Capitulo 5 apresenta a avaliação do trabalho. Por fim, o Capítulo 6 traz as conclusões do trabalho realizado e uma discussão sobre as possibilidades de extensão do mesmo.

2 CONCEITOS BÁSICOS

Neste capítulo, é feita uma breve introdução aos conceitos básicos de computação concorrente, seus benefícios e desafios de programação. Também é apresentado o mecanismo de análise estática de código que será usado neste trabalho.

2.1 COMPUTAÇÃO CONCORRENTE

A computação concorrente se baseia em dividir o programa em vários contextos que podem ser executados simultaneamente. Com isso, o programa pode obviamente ser executado mais rápido, caso ele possa utilizar mais de uma unidade de processamento. Vale notar que para muitos casos, não se trata apenas de melhorar o desempenho do programa, mas devido à própria lógica do programa, a separação em mais de um fluxo de execução facilita até mesmo a implementação do programa.

Um exemplo simples que pode ser usado para ilustrar o benefício de múltiplos fluxos de execução é o problema de soma de vetores. Como nesta operação o cálculo de cada elemento do vetor de saída não depende do cálculo dos demais elementos, é possível implementar um algoritmo concorrente onde cada elemento de saída (ou um grupo deles) é calculado por um fluxo de execução independente. Consequentemente, o tempo de execução da aplicação pode ser significantemente reduzido, caso a máquina disponibilize mais de uma unidade de execução.

Um outro exemplo que mostra o benefício no desempenho da computação concorrente são as aplicações gráficas. Justamente para lidar melhor com esse tipo de aplicação foram criadas as unidades de processamento gráficas (GPUs), as quais possuem milhares de unidades de processamento que são utilizadas simultaneamente para tratar de cada pixel transmitido do computador para o monitor. Por outro lado, um editor de texto com backup automático é um exemplo onde a estrutura lógica do programa sugere o desenvolvimento usando computação concorrente, uma vez que é mais fácil colocar o processo de backup em um fluxo separado do que junto com o resto da lógica do programa.

Existem ainda alguns padrões comuns na programação concorrente, como produtor/-consumidor e leitores/escritores, que pela própria lógica do programa se mostram mais simples de implementar de forma concorrente do que de forma sequencial. No padrão produtor/consumidor, diferentes fluxos de execução compartilham um espaço comum de memória (*e.g.* um buffer). Os Produtores geram e incluem novos objetos nesse espaço comum; enquanto os Consumidores consomem os objetos depositados nesse espaço comum, realizando algum processamento sobre os mesmos. No padrão leitores/escritores, onde similarmente os fluxos acessam um espaço de memória comum, os Escritores registram/escrevem dados no espaço comum (um arquivo, ou uma tabela por exemplo);

enquanto os Leitores acessam/leem os dados escritos e os utilizam para a realização da tarefa designada para eles no programa.

Um fator importante da computação concorrente, que se mostrou especialmente essencial nos últimos exemplos dados, é a presença de um espaço de memória comum entre os diferentes fluxos de execução. Esse espaço, que em geral é a memória global do programa, é essencial para comunicação e sincronização dos diferentes fluxos de execução. Portanto, sempre que um fluxo precisar ler alguma informação disponibilizada pelos outros fluxos ou então compartilhar alguma informação com os outros fluxos, ele acessa a memória compartilhada. Vale notar que, de forma geral, os vários fluxos precisam constantemente acessar a memória compartilhada, muitas vezes inclusive acessando junto com os outros fluxos.

Com, isso, surge um novo problema: condições de corrida. Este tipo de problema ocorre quando um fluxo tenta escrever em uma determinada posição da memória e outro tenta acessá-la ao mesmo tempo. Como a memória não pode ser de fato acessada simultaneamente, acaba que um fluxo acessa a memória antes do outro, e essa ordem não é definida pelo programa e pode, consequentemente, mudar em outras execuções do programa. Isso obviamente pode levar a inconsistências no programa em diferentes execuções e diversos comportamentos inesperados. Um caso especial é quando um fluxo é interrompido quando está no meio de uma operação de leitura ou alteração de uma variável (entre uma instrução de máquina e outra) e outro fluxo acessa essa mesma variável antes do outro fluxo conseguir salvar o resultado da sua operação. Condições de corrida são o problema central que este trabalho busca identificar para que o programador possa corrigí-las e evitar que as mesmas prossigam para a versão final da aplicação.

Uma das formas de evitar condições de corrida é a criação de seções críticas de código. Isto é, seções onde apenas um fluxo pode ser executado de cada vez. Com isso, as operações de leitura e escrita de variáveis compartilhadas são sempre concluídas antes que outro fluxo de execução consiga acessar essa mesma variável. O mecanismo presente na maioria das bibliotecas para criar essas sessões críticas são os *locks*.

Basicamente, quando um fluxo alcança uma seção crítica ele verifica se o lock responsável por aquela sessão está alocado. Caso esteja, o fluxo fica bloqueado até que o lock seja liberado. Caso o lock esteja liberado (ou após a sua liberação) o fluxo atual aloca aquele lock para garantir que ele executará a seção crítica sozinho. Uma vez que o fluxo conclua a sua seção crítica, ele libera o lock para que outros fluxos consigam acessar suas seções críticas. Vale notar, no entanto, que identificar essas seções críticas para saber onde a presença dos locks é realmente necessária, sem criar seções muito grandes e, consequentemente, perder a característica concorrente do código, nem sempre é fácil.

2.1.1 Biblioteca `pthread`

Para a linguagem C existem algumas bibliotecas disponíveis para desenvolvimento concorrente, como `OpenMP`, `MPI` e a biblioteca `pthread`, que foi escolhida para este projeto. Essa escolha se deu pelo fato da biblioteca possuir todos os mecanismos de programação concorrente que pretendemos explorar, além de ser a biblioteca usada na disciplina de Computação Concorrente, do curso de Ciência da Computação na UFRJ. Detalhamos a seguir algumas das funções que a biblioteca possui que são de interesse deste trabalho.

As funções `pthread_create` e `pthread_exit` são respectivamente responsáveis pela criação e finalização dos fluxos de execução. A função `pthread_join` permite esperar a finalização de um dado fluxo de execução antes de dar prosseguimento no contexto atual.

O código 2.1 mostra um exemplo básico de aplicação concorrente em C usando as funções da biblioteca `pthread`. Nele, o programa começa executando um fluxo único com a função `main`. Usando a função `pthread_create` o programa cria um novo fluxo que executa a função `thread`. A partir desse momento, o programa contém dois fluxos concorrentes, cada um responsável por escrever uma frase na saída padrão do programa. No caso do fluxo da `main`, após imprimir sua frase ele espera o outro fluxo acabar (com a chamada da função `pthread_join`) para posteriormente ele mesmo acabar com a função `pthread_exit`.

Código 2.1 – Exemplo de programa em C com mais de um fluxo de execução

```c
#include <stdio.h>
#include <pthread.h>

void* thread ( void* arg ) {
    printf("Hi from the thread\n");

    pthread_exit((void *)NULL);
}

int main ( void ) {
    pthread_t t;
    pthread_create(&t, NULL, thread, NULL);
    printf("Hello from main\n");
    pthread_join(t, NULL);

    pthread_exit((void *)NULL);
}
```

Para o mecanismo de `locks`, a biblioteca dispõe de quatro funções. `pthread_mutex_init` e `pthread_mutex_destroy` são as funções responsáveis por iniciar e destruir os locks. Já as funções `pthread_mutex_lock` e `pthread_mutex_unlock` são as funções responsáveis por fechar (alocar) e abrir (liberar) os locks, respectivamente.

O código 2.2 mostra um exemplo de aplicação que usa as operações de lock para criar seções críticas de código. Note que sem a presença das funções `pthread_mutex_lock` e `pthread_mutex_unlock` delimitando as seções críticas em cada função, a variável `var` estaria sujeita a condição de corrida.

Código 2.2 – Exemplo do uso de locks

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex;
int var = 0;

void* thread ( void* arg ) {
    pthread_mutex_lock(&mutex);
    var++;
    pthread_mutex_unlock(&mutex);

    pthread_exit((void *)NULL);
}

int main ( void ) {
    pthread_t t;

    pthread_mutex_init(&mutex, NULL);
    pthread_create(&t, NULL, thread, NULL);

    pthread_mutex_lock(&mutex);
    var++;
    pthread_mutex_unlock(&mutex);

    pthread_join(t, NULL);

    pthread_mutex_destroy(&mutex);
    printf("var = %d\n", var);

    pthread_exit((void *)NULL);
}
```

Existem alguns outros mecanismos disponíveis na biblioteca PThread que podem ser usados para lidar com os problemas de condições de corrida, como por exemplo `semáforos`. No entanto, apenas os recursos aqui apresentados serão considerados neste trabalho.

2.2 GRAFO DE FLUXO DE CONTROLE

Um dos mecanismos que compiladores usam internamente ao longo do processo de compilação é o grafo de fluxo de controle. Este grafo, como o nome indica, consegue

identificar todos os caminhos que o programa pode tomar, definindo vértices para linhas de código e arestas para diferentes fluxos dentro de cada função.

As informações contidas em um grafo de fluxo de controle são essenciais para o propósito deste trabalho, dado que ao longo do programa uma das maiores preocupações é justamente identificar as sequências possíveis de acesso a variáveis de escopo global e criação de diferentes fluxos de execução. Além disso, permitem que seja feita uma análise pontual em cada nó do grafo e uma análise completa do fluxo de execução como um todo.

Para ilustrar o conteúdo apresentado por um grafo de fluxo de controle, utilizamos o código 2.3 como exemplo. O grafo de fluxo de controle desse código foi criado usando o `gcc` e sua imagem foi gerada com auxílio da ferramenta WebGraphiz[1]. No grafo é possível observar a separação nos fluxos possíveis em cada função. Na função `thread`, a presença do comando condicional fica evidenciada na bifurcação do seu grafo (Fig. 1). Já na função `main` podemos observar a presença dos dois laços `for` que ficam encapsulados no grafo dentro de um subgrafo próprio (Fig. 2 e 3). Olhando com mais atenção para cada nó é fácil perceber acessos e atribuições às variáveis globais e estáticas, que mantém o nome (no exemplo, `var`) bem como as chamadas de função reconhecidas pelo padrão `call['função']`.

[1] http://www.webgraphviz.com/

Código 2.3 – Exemplo de programa concorrente com laços e condicionais

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex;
int var = 0;

void* thread ( void* arg ) {
    pthread_mutex_lock(&mutex);
    if (var%2)
        var++;
    else
        var += 2;
    pthread_mutex_unlock(&mutex);

    pthread_exit((void *)NULL);
}

int main ( void ) {
    pthread_t t[2]; int i;

    pthread_mutex_init(&mutex, NULL);
    for (i=0; i<2; i++)
        pthread_create(&(t[i]), NULL, thread, NULL);

    pthread_mutex_lock(&mutex);
    var++;
    pthread_mutex_unlock(&mutex);

    for (i=0; i<2; i++)
        pthread_join(t[i], NULL);

    pthread_mutex_destroy(&mutex);
    printf("var = %d\n", var);

    pthread_exit((void *)NULL);
}
```

Figura 1 – Grafo de fluxo de controle referente à função `thread` do código 2.3

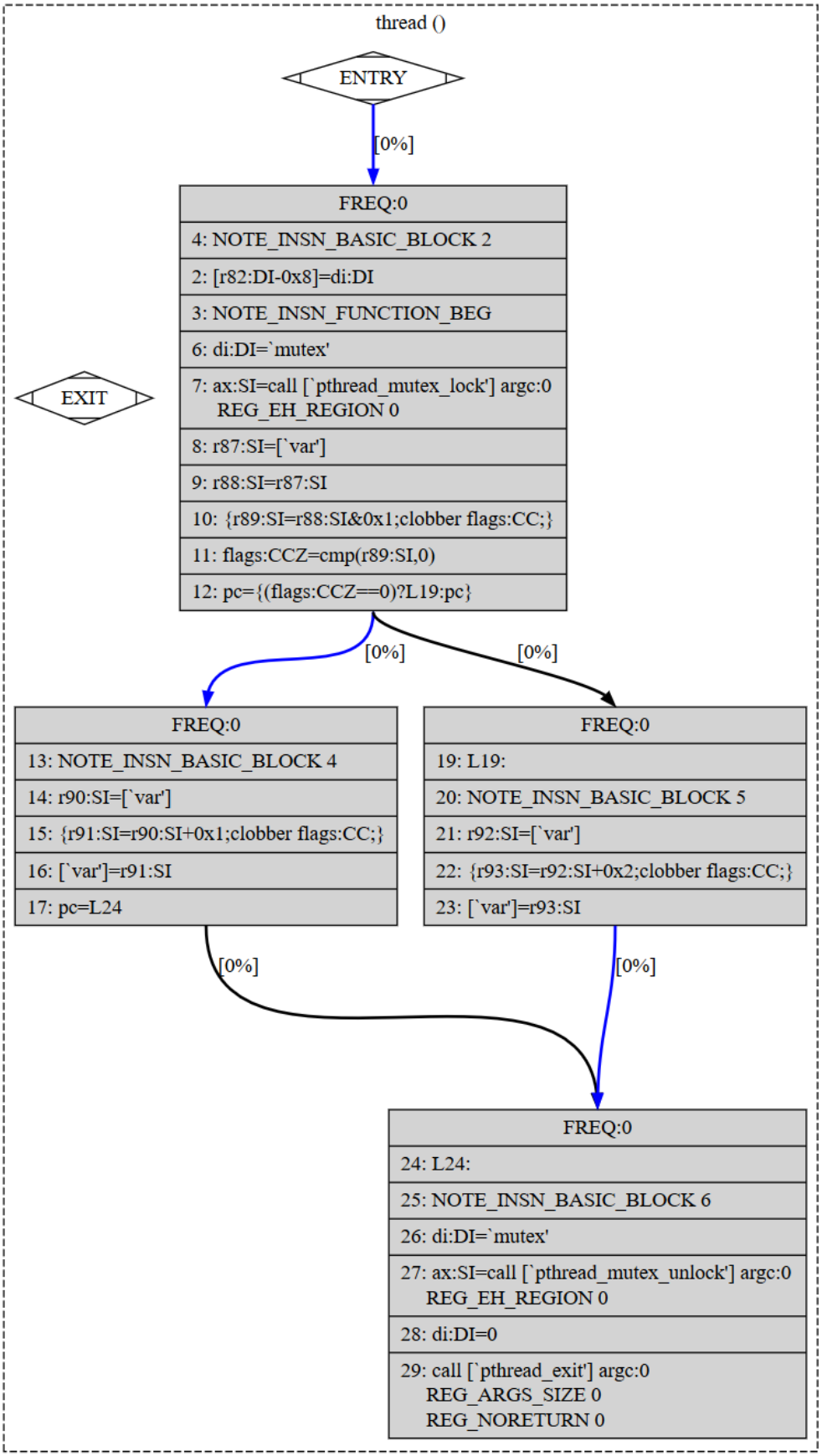


Figura 2 – Grafo de fluxo de controle referente à função `main` do código 2.3

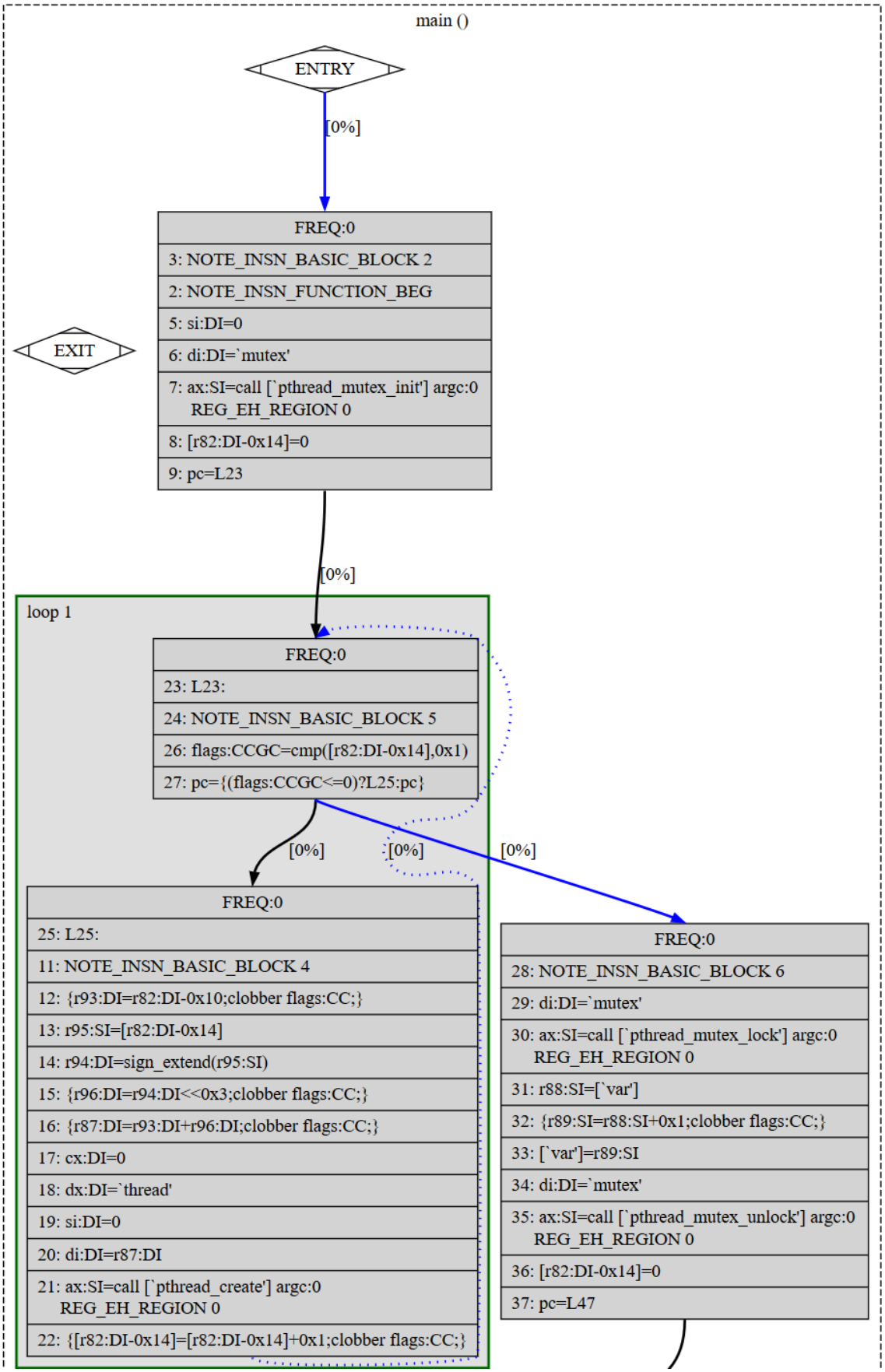
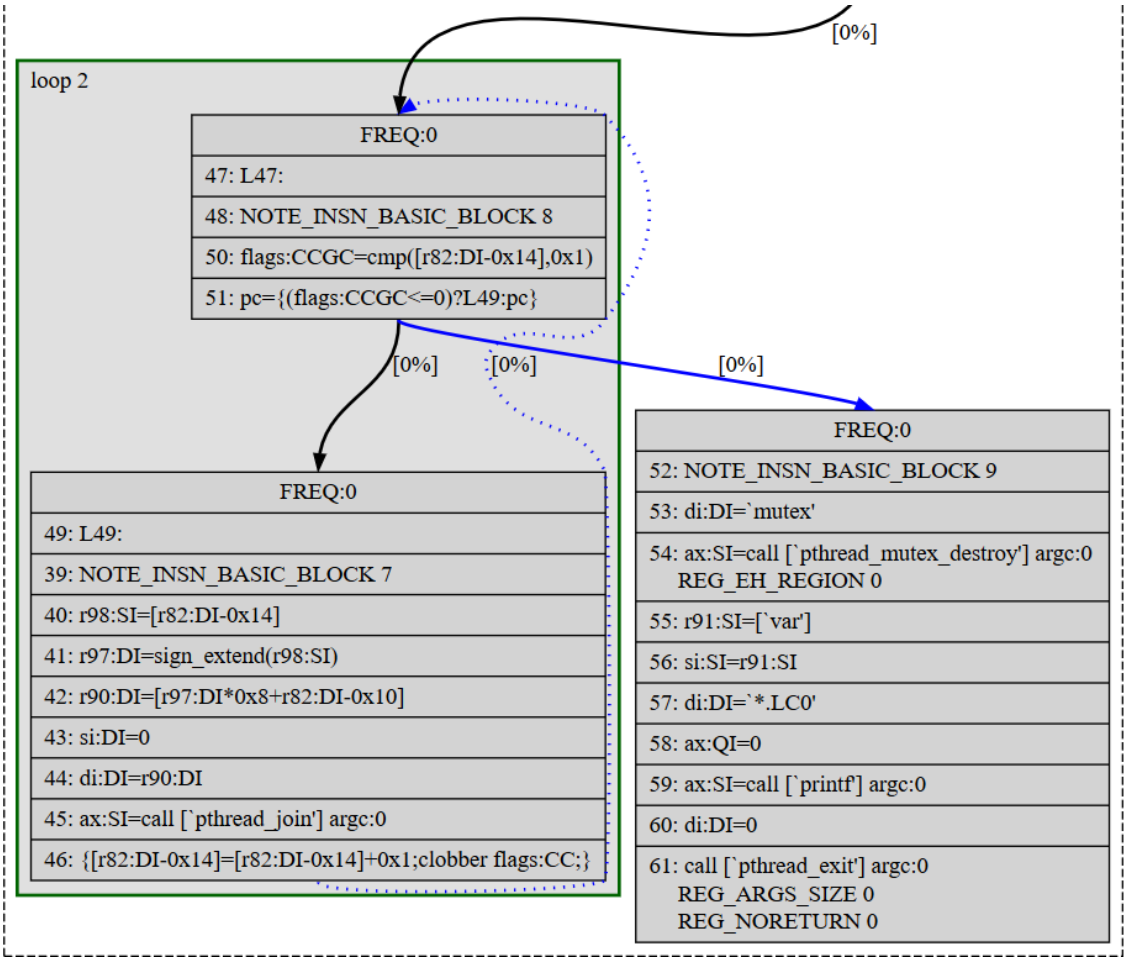


Figura 3 – Grafo de fluxo de controle referente à função `main` do código 2.3 (continuação)



3 FERRAMENTA PARA DETECÇÃO DE CONDIÇÕES DE CORRIDA EM CÓDIGOS C USANDO A BIBLIOTECA PTHREADS

A programação concorrente traz consigo vários desafios, especialmente para os inciantes, e há poucas ferramentas disponíveis para auxiliar os programadores a detectar possíveis falhas nos seus códigos. Neste capítulo, descrevemos algumas dessas ferramentas e apresentamos a proposta de uma nova ferramenta focada particularmente na detecção de condições de corrida para acesso a uma variável em programas concorrentes escritos na linguagem C com auxílio da biblioteca PThread, visando auxiliar estudantes que estão sendo introduzidos à programação concorrente.

3.1 FERRAMENTAS PARA DETECÇÃO DE CONDIÇÃO DE CORRIDA

Como dito no capítulo 1, existem ferramentas criadas para detectar e evitar condições de corrida em diferentes linguagens. Algumas linguagens possuem mecanismos embutidos nelas mesmas ou no compilador padrão para tratar condições de corrida. Em contraponto, outras linguagens necessitam de ferramentas externas para auxiliar nesse processo, como por exemplo a linguagem C.

Rust é um exemplo de linguagem que inclui mecanismos para evitar condições de corrida. Para isso, Rust trata as variáveis de uma forma especial. Toda variável é dada como imutável, a menos que ela seja explicitamente nomeada como mutável. Contudo apenas isso não seria suficiente para impedir as condições de corrida, sendo o mais importante que apenas uma referência mutável para uma variável possa existir em um dado momento. Isto é, quando um fluxo possui uma referência mutável e pode alterar o valor de uma variável, outro fluxo não deveria conseguir uma referência mutável para a mesma variável, impedindo assim a condição de corrida. Isto portanto torna identificável em tempo de compilação qualquer possibilidade de condição de corrida (KLABNIK, 2018).

Como outro exemplo, temos a linguagem Go que apresenta uma forma diferente de lidar com condições de corrida. Passando a opção `-race` para o compilador, a linguagem inclui um verificador de condições de corrida que pode alertar o programador, criar logs ou então parar completamente o programa ao identificar a ocorrência de uma condição de corrida (GOOGLE, 2019).

Já quando se trata de ferramentas externas utilizadas como auxiliares na detecção de condição de corrida, temos como exemplo o RacerD. Esta é uma ferramenta para detecção de erros de concorrência que é parte do projeto de análise estática desenvolvido pelo Facebook, denominado Infer. Feito para analisar grandes programas, o RacerD não se propõe a analisar o código completo. Ao invés disso, usa anotações no código Java para se guiar e saber quais classes analisar, ou então quais métodos não são motivo de preocupação, tais como métodos funcionais (BLACKSHEAR et al., 2018).

No caso de programas escritos na linguagem C, da qual o presente trabalho trata, a ferramenta Helgrind é uma das possibilidades de auxílio para detecção de condições de corrida de forma efetiva. Sendo parte da ferramenta Valgrind, uma framework de instrumentação para construção de ferramentas de análise dinâmica de código, Helgrind é capaz de analisar não apenas códigos na linguagem C, mas também em C++ e FORTRAN, contanto que usem a biblioteca PThreads. Analisando a ordem de criação dos diferentes fluxos de execução no programa e os acessos à memória feito por cada um dos fluxos, a ferramenta consegue identificar se é possível que dois acessos à mesma variável ocorram simultaneamente. Além de levar em conta a criação, finalização e junção dos diferentes fluxos no programa, Helgrind considera ainda os locks, variáveis usadas para notificação de eventos entre fluxos, locks de leitor/escritor, spinlocks, semáforos e barreiras (DEVELOPERS, c2019).

Apesar de bem completa, a ferramenta Helgrind não foi desenvolvida para ser usada por programadores iniciantes na área. Consequentemente, a quantidade de informação que ela retorna, bem como a forma com que essa informação é transmitida — usando posições de memória em hexadecimal e mostrando pilhas com várias funções das bibliotecas internas — pode se tornar um obstáculo no entendimento do problema para o aluno que tentar usar a ferramenta. Além disso, em diversos casos é necessária uma série de configurações e anotações por parte do programador para que a ferramenta funcione adequadamente, o que é mais um impeditivo no seu uso por parte de alunos que estejam cursando uma disciplina introdutória de computação concorrente.

3.2 FERRAMENTA PROPOSTA

Neste trabalho, propomos uma ferramenta de análise estática de código, para programas escritos na linguagem C e com funções da biblioteca `pthread`, com a finalidade de detectar possíveis condições de corrida no acesso a variáveis globais e estáticas. Essa ferramenta visa auxiliar, em particular, alunos que estão cursando disciplinas básicas de computação concorrente, chamando sua atenção para possíveis pontos de falha no código. Utilizando a análise estática é possível abranger as várias possibilidades de combinação dos diferentes fluxos sem depender de encontrar as diferentes permutações de execução dos comandos individuais.

Serviram de inspiração para este trabalho, em especial, as duas últimas ferramentas citadas na Seção 3.1: RacerD e Helgrind. Do RacerD, o conceito de sumarizar o conteúdo de funções para melhor tratar a relação entre funções, e assim identificar as possíveis condições de corrida, foi essencial para a criação da ferramenta. Já do Helgrind utilizou-se o que eles chamam de relação *happens-before*, que representa a forma de analisar a criação dos vários fluxos ao longo do programa para saber assim quais fluxos podem estar rodando simultaneamente e, consequentemente, causar uma condição de corrida.

A ferramenta pretende conseguir identificar 100% das condições de corrida que usam variáveis globais e estáticas, mas não é uma preocupação tão grande garantir que alguma situação em que a condição de corrida já foi tratada pela lógica do programa seja ignorada (falso positivo). Isto é, o programa evita ao máximo obter falsos negativos (falhar na detecção de situações que podem levar o programa a condições de corrida) e, quando necessário, opta por mostrar um falso positivo.

A decisão de tolerar falsos positivos se dá por dois motivos. Primeiro, ao longo do desenvolvimento do projeto percebemos que, para melhor tratar os casos onde o programa retorna um falso positivo e garantir a validade do resultado, a complexidade da solução aumentaria consideravelmente ou a garantia era simplesmente impossível, logo o resultado retornado sempre correria o risco de ser falso. Sendo assim, optou-se por permitir falsos positivos visto que, de certa forma, os falsos positivos têm um certo cunho didático. Isto é, não só os falsos positivos em si, mas o simples conhecimento de que o resultado apontado pode ser falso leva o aluno a precisar rever com olhos críticos os acessos às variáveis apontadas pelo programa, reforçando assim o conteúdo que ele precisa aprender.

Tendo o foco em alunos da disciplina Computação Concorrente (DCC/UFRJ), um curso introdutório na área, algumas concessões foram feitas ao delimitar o escopo da ferramenta. O programa assume que todas as funções a serem analisadas estão em um único arquivo, dado que a maioria dos trabalhos dos alunos da disciplina são feitos nesse formato. Consequentemente, o programa assume que nenhuma condição de corrida acontece com funções fora do arquivo analisado. Além disso, dos mecanismos conhecidos para sincronização de diferentes fluxos de execução, apenas *locks* foram considerados, visto que é o primeiro mecanismo apresentado na disciplina.

4 IMPLEMENTAÇÃO DA FERRAMENTA PROPOSTA

Neste capítulo, detalhamos o algoritmo utilizado para implementar a ferramenta de análise estática de programas concorrentes proposta, bem como as restrições de aplicação desse algoritmo e as ferramentas auxiliares usadas para o seu desenvolvimento.

Para alcançar o objetivo de apontar todas as condições de corrida que podem vir a ocorrer em um programa concorrente, é necessário primeiramente identificar todos os possíveis acessos a variáveis de escopo global, a partir das tarefas realizadas em cada fluxo de execução interno ao programa. Feito isso, o passo seguinte consiste em identificar em que pontos do programa novos fluxos de execução são criados. Apenas quando dois ou mais fluxos de execução acessam a mesma variável e um deles escreve nessa variável, entende-se que uma condição de corrida é possível.

4.1 ALGORITMO DE ANÁLISE ESTÁTICA DE PROGRAMAS CONCORRENTES

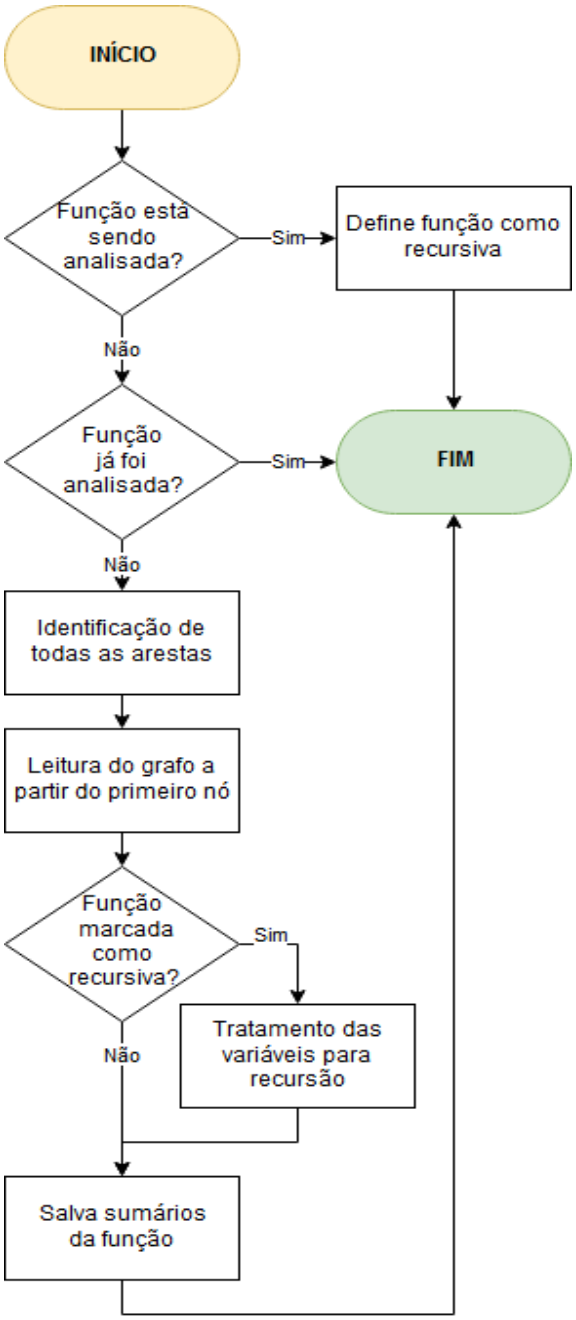
Para a ferramenta, cada função possui um sumário próprio contendo seu estado atual (se a função já foi analisada), as variáveis acessadas naquela função (no mesmo fluxo ou em fluxos criados por ela) que contém um sumário próprio, os locks que foram abertos ou fechados ao longo da função, bem como informação sobre a função ser recursiva ou não. O sumário das variáveis inclui se naquele escopo a variável foi acessada para leitura ou escrita, se foi acessada naquele ou em outro fluxo e se existe condição de corrida até o momento para aquela variável.

Portanto, dado o grafo de fluxo de controle do código a ser analisado, o programa começa a analisar a função `main` de acordo com o fluxograma na Figura 4. Ao final da execução da ferramenta, o resultado da análise estará contido no sumário da função `main` e as variáveis com possíveis condições de corrida estarão identificadas nesse sumário, uma vez que, para um programa padrão no escopo da ferramenta, tudo é criado pela `main`, então o seu sumário é representativo do código inteiro. Isso é verdade inclusive para as variáveis acessadas apenas em outros fluxos, pois como o fluxo foi criado dentro da `main`, ou por alguma função chamada por ela, o sumário da função que o novo fluxo usa acaba embutido no sumário da `main`, como explicado mais a frente.

Para a análise de uma função qualquer, dado o subgrafo correspondente, são necessários dois passos básicos: (i) ler e armazenar as arestas do seu grafo de forma mais amigável para o algoritmo, dado que o formato fornecido não provê uma forma de se obter as arestas saindo de um dado nó em tempo constante; e (ii) analisar o subgrafo em si, como representado pelo fluxograma na Figura 5.

Além dos dois passos citados no parágrafo anterior, podemos notar algumas condicionais na Figura 4 referentes ao estado atual da função. A primeira condicional é necessária

Figura 4 – Fluxograma de análise de função



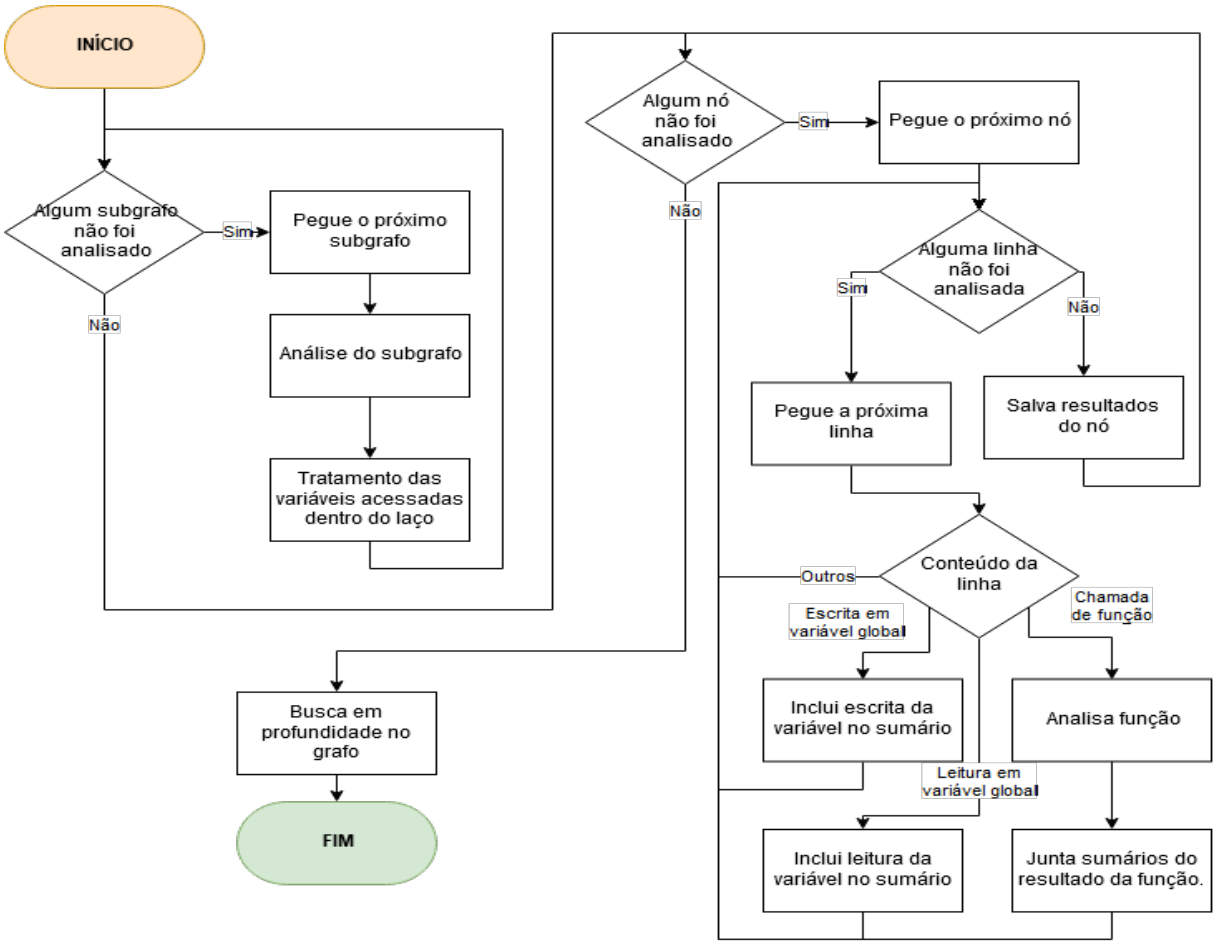
pois no caso de uma função recursiva, a análise padrão acabaria gerando uma dependência circular. Dessa forma, optou-se por marcar a função como recursiva e reavaliar a possibilidade de condição de corrida para as variáveis acessadas naquela função após a análise do seu grafo. Já a segunda verificação é apenas por motivo de desempenho, para evitar repetir a análise de uma função que já foi previamente analisada.

Uma vez que o grafo tenha sido analisado, a ferramenta verifica se a função é recursiva. Caso seja, é feita uma análise em cima de todas as variáveis acessadas ao longo da função. Essa avaliação é necessária pois, ainda que a análise do grafo não indique que uma determinada variável possa sofrer condição de corrida, é possível que dada a

recursividade da função a variável acabe tendo esse problema. Por exemplo, suponha que um acesso a uma determinada variável aconteça antes da criação de um fluxo que escreva nela. Contudo, como a função é recursiva esse acesso acontece novamente após a criação do fluxo, podendo assim acarretar numa condição de corrida.

Após isso, resta apenas salvar o sumário resultante da função na memória do programa e mostrar o resultado — no caso da função `main` — ou dar prosseguimento à análise da função que invocou a função que acabou de ser analisada.

Figura 5 – Fluxograma de análise de grafo



Quanto à leitura do grafo em si (Figura 5), três passos são importantes. Primeiramente, para cada grafo é feita a análise de todos os seus subgrafos de forma recursiva. Feito isso, todos os nós são analisados para se obter um sumário próprio, similar ao sumário da função completa, e posteriormente é feita uma busca em profundidade no grafo para juntar a informação em todos os nós e subgrafos no sumário da função.

Para formar o sumário em cada nó, o script lê as linhas de código na linguagem intermediária do compilador. Com isso, o programa consegue identificar três partes importantes: a leitura ou escrita de variáveis globais ou estáticas, bem como chamadas de funções e seus argumentos. Com isso é possível identificar para cada nó não apenas as lei-

turas e escritas feitas diretamente naquela função, mas também o impacto das funções ali chamadas, *i.e.*, é possível incorporar o sumário das funções chamadas naquele nó, sendo assim possível incluir as variáveis acessadas nelas como parte do nó.

Aqui vale notar que algumas funções citadas na seção 2.1.1 merecem atenção especial. Referente à identificação de chamadas de função, existem três comportamentos distintos possíveis. O primeiro refere-se às funções que controlam o estado atual dos locks naquele sumário, isto é, as funções `pthread_mutex_lock` e `pthread_mutex_unlock`. Isso é importante para, na ocorrência de acesso às variáveis globais ou estáticas, saber se elas estão sendo protegidas por algum lock e, se sim, quais locks são esses.

O segundo comportamento possível refere-se à função `pthread_create`. Quando uma chamada a essa função é reconhecida, o algoritmo busca identificar nos seus parâmetros qual a função do novo fluxo criado. Em seguida, gera o sumário dessa função (de forma similar ao processo aqui descrito para criar o sumário da função principal) e associa as variáveis globais presentes naquela função identificando-as como variáveis acessadas em paralelo naquele nó. Isto é, após analisar as variáveis que a função chamada em um novo fluxo de execução acessa, o programa guarda essa informação para poder identificar posteriormente se a mesma variável foi acessada no fluxo que criou o novo contexto.

O terceiro e último comportamento tratado para chamadas de funções lida com as outras funções em geral. O programa busca se o grafo gerado pelo compilador possui informação sobre aquela função, isto é, se a função foi criada no código sendo analisado. Sendo a função própria do código, o programa analisa então a função e gera o seu sumário como dito anteriormente. Com o sumário em mãos, ele inclui de forma sequencial o sumário naquele nó. Isto é, ele lê e inclui todos os acessos feitos na função chamada ao seu próprio sumário, sabendo que foram feitos depois dos acessos já verificados naquele nó.

Vale notar que toda vez que um novo acesso a uma determinada variável é identificado, verifica-se a presença de acessos a esta mesma variável em fluxos de execução previamente criados para verificar se aquele novo acesso pode acarretar numa condição de corrida. Isso é feito em ambos os casos: quando o acesso é feito diretamente no nó; ou a partir de outra função cujo sumário foi acrescentado ao nó. No último caso, essa verificação é feita para todas as variáveis acessadas na outra função, tanto os acessos no mesmo fluxo quanto os feitos em fluxos paralelos.

Uma vez que todos os nós tenham o seu sumário parcial, faz-se a busca em profundidade com o objetivo de juntar todos os sumários parciais e formar o sumário da função inteira. Porém, para fazer essa junção alguns cuidados são necessários.

Primeiramente, cada laço dentro da função forma um subgrafo próprio e é tratado separadamente. Assume-se que todo o conteúdo do laço é repetido pelo menos duas vezes, visto que de forma geral não é possível saber previamente a quantidade de iterações daquele laço. Portanto, o subgrafo do nó, tratado como função, é analisado. Gera-se um

sumário, e depois processa-se esse sumário para avaliar o impacto da repetição daquele laço, tais como criação de mais de um fluxo de execução, ou até mesmo o aparecimento de uma condição de corrida, de forma similar ao tratamento aplicado em funções recursivas. Este sumário do laço é então colocado como um nó no grafo da função para análise na busca.

Tendo tratado o caso específico dos laços, resta apenas lidar com a junção do fluxo da função. Para isso, é preciso considerar dois aspectos. Primeiro, a junção sequencial de dois nós. Esse processo é extremamente similar à inclusão do sumário de uma função em um nó. O outro caso é quando saindo de um nó, existe mais de uma aresta, como por exemplo no caso de uma condicional (if-else). Nesses casos, o programa avalia cada ramo separadamente e depois junta os dois de forma a considerar todas as variáveis acessadas e condições de corridas criadas nos ramos antes de juntar ao nó pai.

Ao final da execução, o script terá analisado todas as funções acessadas a partir da função principal e juntado todas as variáveis acessadas, bem como as condições de corrida que podem ocorrer no seu sumário. Portanto, para se obter o resultado final do programa, basta avaliar as condições de corrida que foram geradas pela função `main`.

4.2 RESTRIÇÕES DO ALGORITMO PROPOSTO

Para o funcionamento adequado do programa, supõe-se que o programa está contido em um único arquivo, uma vez que no método utilizado para se obter o grafo de fluxo de controle (descrito na Seção 4.3) apenas as funções de um arquivo fonte são incluídas no arquivo contendo o grafo. Consequentemente, qualquer condição de corrida que incluísse funções definidas em outros arquivos seriam ignoradas.

Além disso, como dito na Seção 3.2, o programa se propõe apenas a identificar os acessos diretos às variáveis de escopo global. Isto é, ele ignora o uso de variáveis do tipo ponteiro e se atém apenas a detectar leituras e escritas nos valores dessas variáveis e as condições de corrida daí provenientes.

4.3 FERRAMENTAS AUXILIARES UTILIZADAS

O algoritmo proposto, disponível no GitLab[1] foi implementado no formato de um script `python` (FOUNDATION, c2001). Foi escolhida a linguagem Python pela sua simplicidade e pelo fato dela oferecer várias ferramentas auxiliares, como uma biblioteca nativa para uso de expressões regulares — `re` — e estruturas de dados sofisticadas, tais como listas, dicionários e listas não ordenadas (`sets`), incluindo operadores unários para essas mesmas estruturas.

Como ferramentas externas foram utilizados o gcc (FOUNDATION, 2019) (versão >= 4.8.2, sendo que no trabalho foi utilizada a versão 5.4.0) sem nenhum tipo de otimização

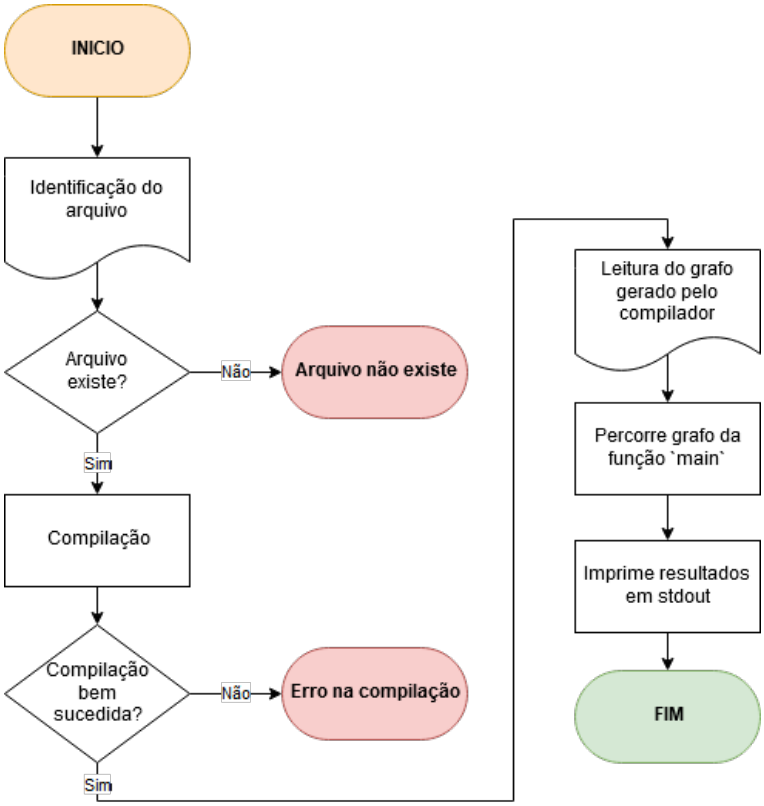
[1] https://gitlab.com/jfportalb/tcc

para gerar o grafo de fluxo de controle do programa, bem como uma biblioteca capaz de ler o formato gerado pelo gcc. No caso, foi escolhida a biblioteca `pydot` (PYDOT, 2018) (versão 1.4.1).

4.4 COMO USAR A FERRAMENTA PROPOSTA

Em um pouco mais de detalhe, a ferramenta funciona da seguinte forma. Dado um nome de arquivo válido, o script chama o compilador (gcc) passando o parâmetro `-fdump-rtl-expand-graph` para gerar o grafo de fluxo de controle. Uma vez que a compilação tenha sido bem sucedida, o script lê o arquivo com o grafo gerado pelo `gcc` com o auxílio da biblioteca `pydot` e o guarda em memória. A partir desse grafo, o programa então faz o processamento necessário e depois imprime os nomes das variáveis que podem sofrer condição de corrida. A Figura 6 ilustra a operação da ferramenta.

Figura 6 – Fluxograma do uso do programa



5 AVALIAÇÃO DA FERRAMENTA DESENVOLVIDA

Neste capítulo serão apresentados os testes feitos com a ferramenta desenvolvida no presente trabalho no intuito de analisar seu desempenho.

5.1 MÉTRICAS

No que concerne a aferição de desempenho da ferramenta, foram realizados testes objetivando a determinação de um conjunto de métricas. Entre as métricas coletadas estão:

- **verdadeiros positivos**: variáveis apontadas pelo programa que são de fato passíveis de condição de corrida;
- **verdadeiros negativos**: variáveis que não foram apontadas pelo programa e de fato não correm risco de condição de corrida;
- **falsos positivos**: variáveis apontadas pelo programa mas que estão protegidas pela lógica do código analisado; e
- **falsos negativos**: variáveis que podem ter condição de corrida, mas que não foram mostradas pelo programa.

5.2 CASOS DE TESTE

Para avaliar a eficiência da ferramenta desenvolvida, foram usadas duas bases de exemplos de códigos C/Pthreads. Uma criada no escopo deste trabalho, com casos de teste que procuravam cobrir todas as possíveis situações tratadas pelo algoritmo proposto; e a outra obtida de programas desenvolvidos por alunos da disciplina Computação Concorrente (DCC/UFRJ), com o objetivo de avaliar os resultados obtidos pela ferramenta a partir de programas desenvolvidos por alunos iniciantes na área de programação concorrente.

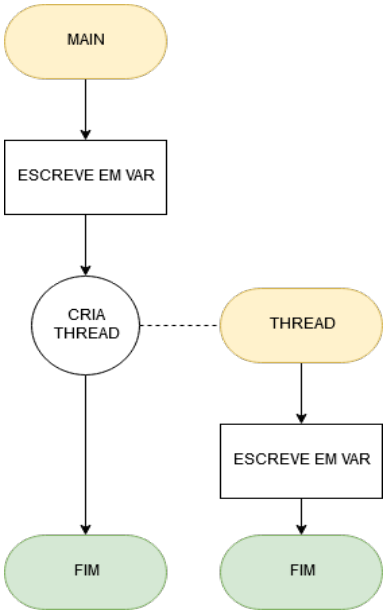
5.2.1 Base de testes da ferramenta

Com o intuito de testar um grande número de grafos e junções de fluxos distintas, foram criados diversos códigos simples com pequenas variações, tais como utilizar chamadas de função ou laços.

A maioria dos testes foi gerada a partir de um teste anterior acrescentando ou removendo algo. Nos exemplos, temos como base o Código 5.1 (fluxograma da Figura 7). Incluindo um acesso à variável `var` na função main após a criação de um novo fluxo de

execução, foi obtido o teste do Código 5.2 (fluxograma da Figura 8). Por fim, outra alteração foi incluir um novo fluxo de execução e criar o teste do Código 5.3 (fluxograma da Figura 9).

Figura 7 – Teste criando apenas um fluxo de execução adicional, sem condição de corrida



Código 5.1 – Teste criando apenas um fluxo sem condição de corrida

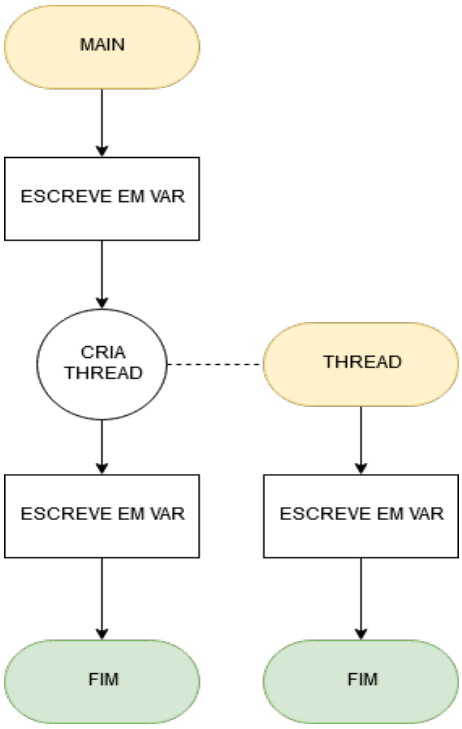
```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {int id;} thread_arg;
void *thread(void *vargp);
pthread_mutex_t mutex;
int var;

int main() {
    pthread_t tid[1]; thread_arg a[1];
    var = 0;
    a[0].id = 0;
    pthread_create(&(tid[0]), NULL, thread, (void *)&(a[0]));
    pthread_join(tid[0], NULL);
    pthread_exit((void *)NULL);
}

void *thread(void *vargp) {
    thread_arg *a = (thread_arg *) vargp;
    var = var + a->id + 1;
    pthread_exit((void *)NULL);
}
```

Figura 8 – Teste criando apenas um fluxo de execução adicional, com condição de corrida



Código 5.2 – Teste criando apenas um fluxo com condição de corrida

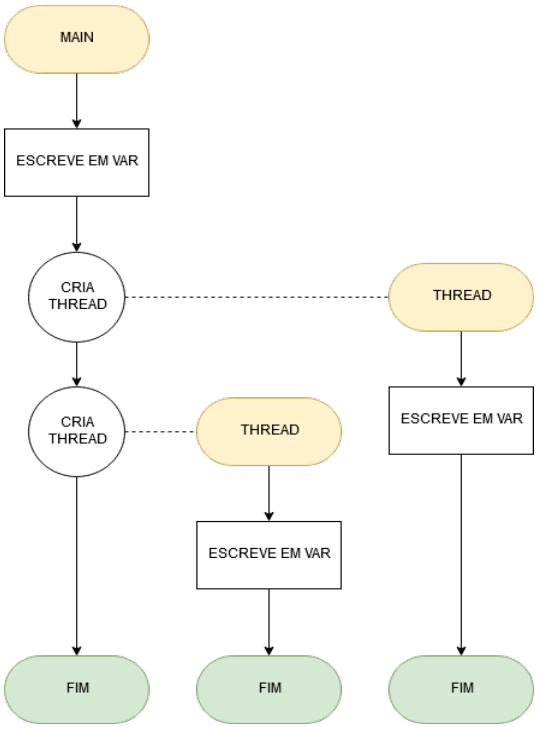
```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {
    int id;
} thread_arg;
void *thread(void *vargp);
pthread_mutex_t mutex;
int var;

int main() {
    pthread_t tid[1];
    thread_arg a[1];
    var = 0;
    pthread_create(&(tid[0]), NULL, thread, (void *)&(a[0]));
    var = 8;
    pthread_join(tid[0], NULL);
    pthread_exit((void *)NULL);
}

void *thread(void *vargp) {
    thread_arg *a = (thread_arg *) vargp;
    int g = var + a->id + 1;
    pthread_exit((void *)NULL);
}
```

Figura 9 – Teste criando dois fluxos de execução adicionais e gerando condição de corrida



Código 5.3 – Teste criando dois fluxos e gerando condição de corrida

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {int id;} thread_arg;
void *thread(void *vargp);
pthread_mutex_t mutex;
int var;

int main() {
    pthread_t tid[2]; thread_arg a[2];
    var = 0;
    a[0].id = 0; a[1].id = 1;
    pthread_create(&(tid[0]), NULL, thread, (void *)&(a[0]));
    pthread_create(&(tid[1]), NULL, thread, (void *)&(a[1]));
    pthread_join(tid[0], NULL); pthread_join(tid[1], NULL);
    pthread_exit((void *)NULL);
}

void *thread(void *vargp) {
    thread_arg *a = (thread_arg *) vargp;
    var = var + a->id + 1;
    pthread_exit((void *)NULL);
}
```

Da mesma forma, os testes foram expandidos para incluir locks, em um fluxo ou nos dois, antes ou depois da criação dos novos contextos. Apesar de não abranger todos os casos possíveis tratados pela ferramenta, as variações conseguiram incluir diversas possibilidades e variações nos grafos resultantes.

5.2.2 Base de testes de trabalhos de alunos da disciplina Computação Concorrente

Foram usados os resultados de duas atividades desenvolvidas na disciplina como base. Um deles tinha o objetivo de calcular o valor de π a partir de uma série infinita. O outro requeria a implementação de um conjunto de operações com matrizes de tamanho variável, tais como encontrar o maior valor presente na matriz ou calcular a soma de todos os seus valores.

No trabalho sobre o cálculo de π, a solução proposta era o cálculo da variável com o uso de um vetor auxiliar, onde cada fluxo de execução escrevia em uma posição exclusiva o resultado da sua soma parcial e na função *main* era feita a soma final. Sendo assim, o único exemplo que tinha a condição de corrida foi um em que o aluno implementou uma solução alternativa em que uma única variável global era acessada por todos os fluxos de execução. Já no segundo trabalho, com as operações de matrizes, o uso de variáveis globais que não eram ponteiros foi maior, visto que o objetivo do trabalho foi obter alguns valores absolutos a partir da matriz.

A partir desses códigos, foi feita uma pre-análise para identificar quais seriam as possíveis condições de corrida presentes. Em seguida, a ferramenta proposta foi executada para verificar se os resultados coincidiam com o esperado. Vale notar que foi analisado tanto quais seriam as condições de corrida realmente possíveis, quanto as que o programa consideraria possíveis, dadas as suas limitações.

5.3 RESULTADOS OBTIDOS

Foram realizados um total de 32 testes, sendo 21 da base da ferramenta (Tabela 1), 6 do trabalho do cálculo de π (Tabela 2) e 5 do trabalho com matrizes (Tabela 3). A Tabela 4 sumariza os resultados obtidos de ambas as bases de teste.

Em relação aos resultados, considerando as limitações da ferramenta, como era esperado, não foram encontradas discrepâncias. Já sem ignorar essas limitações, isto é, considerando os casos em que realmente a condição de corrida era esperada, 7 testes retornaram falsos positivos. Considerando apenas os trabalhos de alunos de Computação Concorrente, 36% retornaram falsos positivos. O número de variáveis que acabaram tendo falsos positivos foi considerável em especial nos testes com exemplos dos alunos. Isto ocorreu pois o trabalho que serviu de base para os testes pedia o retorno de várias variáveis.

Tabela 1 – Resultados dos testes criados para a base da ferramenta

Nº	VERDADEIROS POSITIVOS	VERDADEIROS NEGATIVOS	FALSOS POSITIVOS	FALSOS NEGATIVOS
01	0	0	1	0
02	0	1	0	0
03	0	1	0	0
04	1	0	0	0
05	0	1	0	0
06	1	0	0	0
07	0	1	0	0
08	1	0	0	0
09	1	0	0	0
10	1	0	0	0
11	1	0	0	0
12	1	0	0	0
13	1	1	0	0
14	0	2	0	0
15	2	0	0	0
16	1	1	0	0
17	1	0	0	0
18	0	0	0	0
19	1	0	0	0
20	0	0	1	0
21	0	0	1	0

Tabela 2 – Resultados dos testes do trabalho de cálculo de π

Nº	VERDADEIROS POSITIVOS	VERDADEIROS NEGATIVOS	FALSOS POSITIVOS	FALSOS NEGATIVOS
01	0	2	0	0
02	0	3	0	0
03	0	4	0	0
04	0	4	0	0
05	1	3	0	0
06	0	3	0	0

Os problemas encontrados nos resultados obtidos foram causados pelo programa desconsiderar a função `pthread_join`. Isto é, os falsos positivos apontados pelo programa foram causados pelo programa não identificar que os diferentes fluxos de execução em um dado momento se juntaram e, consequentemente, não é mais possível ter condições de corrida depois da junção.

No entanto, apesar desses falsos positivos, o programa se mostrou perfeitamente capaz de lidar com o escopo definido. Isto é, nenhum falso negativo foi detectado, mostrando que considerando apenas o valor de variáveis globais e estáticas, nenhuma possível condição de corrida foi ignorada.

Tabela 3 – Resultados dos testes dos trabalhos com matrizes

Nº	VERDADEIROS POSITIVOS	VERDADEIROS NEGATIVOS	FALSOS POSITIVOS	FALSOS NEGATIVOS
01	0	9	5	0
02	0	11	0	0
03	0	13	4	0
04	0	6	6	0
05	0	7	7	0

Tabela 4 – Sumário do resultado dos testes

TIPO DE TESTE	VERDADEIROS POSITIVOS	VERDADEIROS NEGATIVOS	FALSOS POSITIVOS	FALSOS NEGATIVOS
Criados	13	9	3	0
Cálculo de π	1	19	0	0
Matrizes	0	46	22	0

6 CONCLUSÃO

São notáveis a dificuldade recorrente de alunos de graduação da UFRJ que cursam a disciplina de Computação Concorrente em encontrar problemas simples de condição de corrida nos seus códigos, bem como a falta de ferramentas de apoio capaz de ajudá-los. O presente trabalho procurou desenvolver uma ferramenta capaz de detectar condições de corrida em programas concorrentes escritos em C e com funções da biblioteca PThreads — linguagem e biblioteca adotados na disciplina de Computação Concorrente (DCC/UFRJ).

Com o foco no aprendizado, a ferramenta desenvolvida ao longo do presente trabalho tem o propósito de encontrar condições de corrida envolvendo variáveis globais e estáticas através de análise estática nos códigos dos alunos, sendo o único mecanismo de sincronização considerado no desenvolvimento os locks. Além disso, um dos princípios pensados foi em não prover uma resposta completa ao aluno, para exigir que o mesmo, após executar o script, fosse obrigado a fazer uma análise crítica do seu programa e do resultado da ferramenta.

Ao longo do desenvolvimento do projeto, foram encontrados diversos desafios imprevistos. Apesar de muitos poderem ser resolvidos, alguns deles se mostraram mais complexos, incluindo o tratamento de variáveis do tipo ponteiro e das chamadas da função `pthread_join`.

A dificuldade em lidar com ponteiros se deu em grande parte como consequência da estratégia adotada para realizar a análise do código. Apesar de facilitar imensamente a identificação das variáveis globais e estáticas, o grafo gerado pelo compilador não possui nenhuma referência a variáveis locais ou posições de memória, o que dificulta o tratamento de ponteiros. Como consequência, justamente devido a dificuldade de identificar tanto ponteiros como variáveis locais, tornou-se inviável tratar a função `pthread_join` pois, para conseguir identificar qual o fluxo que a função está esperando, é necessário saber em qual variável foi armazenado o identificador do fluxo criado no método `pthread_create`.

Apesar dos percalços ao longo do desenvolvimento, o projeto obteve sucesso em não mostrar nenhum falso negativo, como foi apontado na Seção 5.3. Além disso, dado o objetivo principal da ferramenta — de auxiliar nas aulas da disciplina Computação Concorrente — os falsos positivos podem servir como incentivo para os alunos analisarem sempre a lógica do código e garantir que o acesso às variáveis esteja realmente protegido, fortalecendo assim o seu aprendizado sem depender exclusivamente da ferramenta.

6.1 DESENVOLVIMENTOS FUTUROS

Devido a limitações de tempo e/ou complexidade de outras funções, certas funcionalidades não foram implementadas. Consequentemente, o projeto ainda tem capacidade

de crescer. Entre essas funcionalidades estão: (i) inclusão de outros mecanismos de sincronização (`e.g.` semáforos e barreiras); (ii) tratamento da função `pthread_join` (ainda que para situações mais simples, como o uso de variáveis simples, ao invés de ponteiros ou vetores); (iii) tratamento de variáveis do tipo ponteiro.

REFERÊNCIAS

BLACKSHEAR, S. et al. Racerd: Compositional static race detection. **Proc. ACM Program. Lang.**, v. 2, n. 144, p. 28, 2018.

DEVELOPERS, T. R. P. **The Rustonomicon**. [s.n.], 2019. Acessado em 08 jul. 2019). Disponível em: <https://doc.rust-lang.org/nomicon/>.

DEVELOPERS, V. **Helgrind: a thread error detector**. [S.l.], c2019. (Acessado em 08 jul. 2019). Disponível em: <http://valgrind.org/docs/manual/hg-manual.html>.

FOUNDATION, I. F. S. **GCC, the GNU Compiler Collection**. [S.l.], 2019. (Acessado em 12 jul. 2019). Disponível em: <https://gcc.gnu.org/>.

FOUNDATION, P. S. **Python**. [S.l.], c2001. (Acessado em 12 jul. 2019). Disponível em: <https://www.python.org/>.

GOOGLE. **Data Race Detector**. [S.l.], 2019. (Acessado em 08 jul. 2019). Disponível em: <https://golang.org/doc/articles/race_detector.html>.

KLABNIK, C. N. S. **The Rust Programming Language**. 2018. ed. [s.n.], 2018. (Acessado em 08 jul. 2019). Disponível em: <https://doc.rust-lang.org/book>.

PYDOT. [S.l.], 2018. (Acessado em 12 jul. 2019). Disponível em: <https://pypi.org/project/pydot/>.