Universidade Estadual de Londrina

GUILHERME AKIRA DEMENECH MORI

REDUÇÃO DO *SPILL CODE* NO ALGORITMO *LINEAR SCAN*

LONDRINA
2023

GUILHERME AKIRA DEMENECH MORI

REDUÇÃO DO *SPILL CODE* NO ALGORITMO *LINEAR SCAN*

Versão Preliminar de Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Wesley Attrot

LONDRINA

2023

GUILHERME AKIRA DEMENECH MORI

REDUÇÃO DO *SPILL CODE* NO ALGORITMO *LINEAR SCAN*

Versão Preliminar de Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Orientador: Prof. Dr. Wesley Attrot
Universidade Estadual de Londrina

Prof. Dr. Segundo Membro da Banca
Universidade/Instituição do Segundo Membro da Banca – Sigla instituição

---

Prof. Dr. Terceiro Membro da Banca
Universidade/Instituição do Terceiro Membro da Banca – Sigla instituição

---

Prof. Ms. Quarto Membro da Banca
Universidade/Instituição do Quarto Membro da Banca – Sigla instituição

Londrina, 11 de dezembro de 2023.

Mostre que os habitantes deste planeta podem cavar um túnel reto passando pelo centro do planeta, começando e terminando em terra seca (suponha que sua tecnologia está suficientemente desenvolvida).

[1]

MORI, G. A. D.. **Redução do *Spill Code* no Algoritmo *Linear Scan***. 2023. 40f. Trabalho de Conclusão de Curso – Versão Preliminar (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2023.

RESUMO

O presente trabalho propõe melhorar a geração de código de *spill* do alocador de registradores *linear scan* implementando nele as técnicas de *interference region spilling* e *live range splitting*. É muito importante que a perda de desempenho do alocador seja balanceada pela melhoria do código gerado. Os resultados do *linear scan* tradicional e aprimorado serão comparados experimentalmente entre si e com outras ferramentas de construção de compiladores. A avaliação considerará os tempos de compilação e execução dos *benchmarks*, bem como a quantidade de instruções `load` e `store` adicionadas.

Palavras-chave: Latex. Template ABNT-DC-UEL. Editoração de texto.

MORI, G. A. D.. **Title of the Work**. 2023. 40p. Final Project – Draft Version (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2023.

ABSTRACT

This project seeks to enhance the spill code generation of the linear scan register allocator by implementing interference region spilling and live range splitting techniques. The allocator's performance loss should be balanced by the improvement of the generated code. The traditional and enhanced linear scan approaches will have their results experimentally compared, as well as with other compiler construction tools. The evaluation will consider compilation and execution times of benchmarks, along with the quantity of added `load` and `store` instructions.

Keywords: Latex. ABNT-DC-UEL template. Text editoration.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

RAM	*Random Access Memory*
JIT	*Just-in-time*

SUMÁRIO

1 INTRODUÇÃO

Compiladores são programas que aceitam código em uma linguagem-fonte de alto nível[1] e o traduzem no código equivalente em uma linguagem-objeto na qual possa ser executada por computadores [5, 3]. Compiladores são estruturados como uma sequência de fases (pelo menos quatro) que analisam o programa e o codificam para o formato desejado [6]. A Figura 1 apresenta essa sequência: as quatro fases necessárias (com o contorno contínuo) são as análises léxica, sintática e semântica no *front end* e a geração de código no *back end*. Estão tracejadas as fases que nem sempre são desenvolvidas: as otimizações de código e a geração de código intermediário.

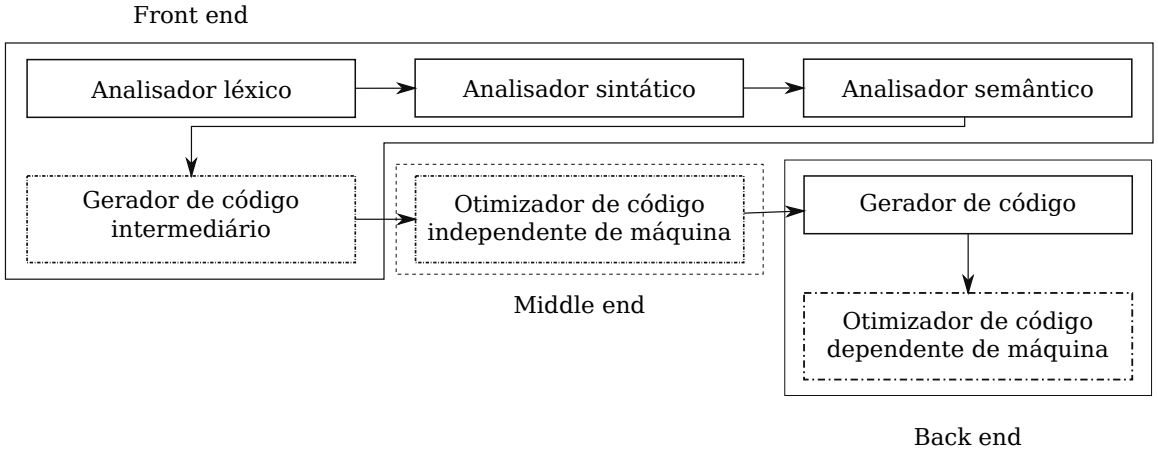

Figura 1 – Fases da compilação

Fonte: autoria própria.

As fases de análise (*front end*) são projetadas para a linguagem do código-fonte e as fases de síntese (*back end*) atendem a arquitetura alvo. A organização do compilador nessas partes permite que sejam combinados o *front end* de qualquer linguagem com o *back end* para qualquer arquitetura. Assim, novas linguagens ou arquiteturas podem se beneficiar de técnicas de otimização já implementadas [3].

O presente trabalho se debruça especificamente sobre a alocação e a atribuição de registradores, realizadas pelo alocador durante a fase de geração de código da máquina alvo.

Na alocação de registradores são escolhidas quais variáveis utilizarão registradores a cada momento e na atribuição são escolhidos os registradores que serão utilizados por cada variável. A atribuição pode ser resolvida em tempo polinomial, porém a alocação

[1] Linguagens de alto nível permitem notações mais abstratas como as utilizadas em ambiente matemático e científico, como Álgebra [2]. Geralmente capazes de gerar código de baixo nível tão eficiente quanto (ou melhor) que humanos, os compiladores foram importantes para a adoção dessas linguagens, como C, Java e Python, no lugar das de baixo nível, como Assembly e linguagens de máquina [3, 4].

ótima é NP-completa[2], sendo sensível às restrições específicas de *hardware* ou sistema operacional [3, 8].

A utilização de registradores para armazenar os valores das variáveis permite alto desempenho, contudo registradores são recursos caros e limitados. A Figura 2 ilustra a hierarquia de memória na forma de uma pirâmide na qual os tipos de memória mais velozes e mais próximos do processador são posicionados nos níveis superiores e no topo estão os registradores. Em função dessa organização hierárquica, os programas podem utilizar a pouca memória mais próxima do processador para os dados mais usados em um certo momento e deixar no armazenamento maior o que será usado mais tarde ou com menos frequência [9]. Assim, a alocação e a atribuição são processos muito importantes para a qualidade do código gerado, contribuindo para melhorar o desempenho dentro das restrições impostas pela hierarquia de memória.

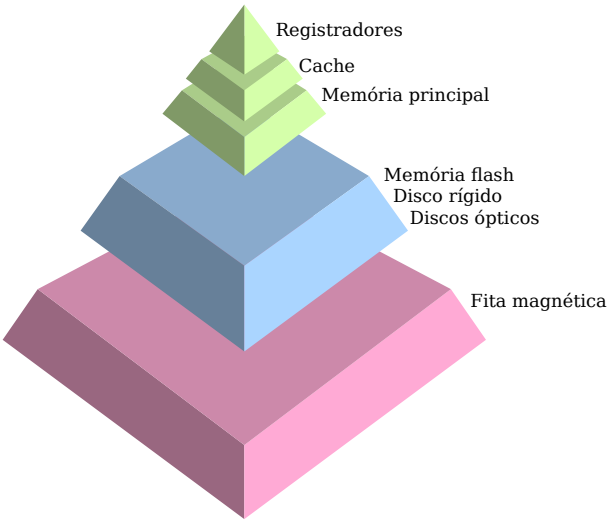

Figura 2 – Pirâmide da hierarquia de memória

Fonte: autoria própria.

Com a grande quantidade de variáveis necessárias pela maior parte dos programas, não há registradores suficientes para dedicar a todas elas [3]. O alocador considera, então, quando os valores das variáveis são úteis e quando não serão mais utilizados (ou seja, em quais instruções estão vivas ou não), para que poucos registradores possam atender diversas variáveis sem conflitos ou perdas. Contudo, quando não for encontrado registrador disponível para alguma variável, recorre-se à memória principal, mais distante do processador e mais lenta. A retirada de variáveis dos registradores e seu armazenamento e carregamento em memória RAM denomina-se *spilling*[3] [9].

A geração de *spill code*, isto é, a adição de instruções `store` e `load` ao código, permite que variáveis tenham seus valores preservados e possam ser acessadas mesmo

[2] Para os problemas não-determinísticos de tempo polinomial completo (NP-completos) são conhecidas somente soluções com pelo menos tempo exponencial [7].

[3] Usualmente traduzido como derramamento.

quando todos os registradores já estejam sendo utilizados. Embora seja mais simples alocar todas as variáveis na memória principal e carregá-las no mínimo de registradores necessário para a execução das instruções, o acesso à memória RAM é muito mais custoso (em tempo e energia) do que o acesso aos registradores, muito mais próximos do processador.

Em geral, os registradores são alocados e atribuídos como um problema de coloração de grafos [3]: variáveis são nós que devem ter cores (registradores) atribuídas de forma que nós vizinhos (conectados por uma aresta) não compartilhem da mesma cor. As arestas chama-se também de interferências. São conectados os nós de variáveis que, em algum momento, estejam vivas simultaneamente. Como dois corpos no espaço, variáveis não podem ocupar o mesmo registrador ao mesmo tempo.

É pouco complexo o processo de identificar quais variáveis estão vivas em cada instrução, bem como o de verificar quais estão vivas ao mesmo tempo [10]. Essas informações permitem conhecer os *live ranges*[4] das variáveis e construir o grafo de interferência. Quando a coloração desse grafo falha (faltam cores/registradores) é preciso gerar *spill* e repetir a computação de interferências e do grafo. O código gerado pode ser muito otimizado, mas depende desse alto custo de alocação. Quando baixo tempo de compilação é necessário, técnicas mais rápidas também são necessárias, como na compilação *just-in-time* (JIT)[5] [12].

A compilação JIT pode se privilegiar de informações do código de alto nível e de tendências conhecidas somente durante a execução, o que possibilita otimização e geração de código específicas para o ambiente de execução [12]. A oposição desses benefícios às limitações de tempo e processamento impostas aos compiladores JIT, que não podem causar ao programa mais atraso do que aceleração, exige deles bastante balanceamento entre rapidez e qualidade do código gerado [12].

Várias técnicas podem ser aplicadas para reduzir o custo da alocação de registradores, mas estas devem gerar código de qualidade suficiente para compensar o processamento adicional sobre o programa interpretado. Estratégias simples demais (porém sem tanta melhoria), como fixar os registradores disponíveis somente para as variáveis mais usadas, ou complexas demais (mesmo que com grande melhora), como coloração de grafos, podem não se adequar aos requisitos de sistemas JIT. Uma conhecida técnica de alocação apropriada para compiladores JIT é chamada *linear scan* [10]. Esse algoritmo não demanda tanto processamento e pode rapidamente gerar código satisfatório para acelerar a execução do programa, compensando seu próprio custo computacional.

A técnica de alocação *linear scan* [10] assume uma ordenação das instruções e

[4] Tempos de vida.

[5] Também chamada de compilação dinâmica, a compilação JIT ocorre durante a execução, com benefícios da compilação estática (que ocorre separadamente, antes do início do programa) e da interpretação [11].

simplifica a representação de *live range* em *live interval*, desconsiderando trechos intermediários em que a variável não está viva e extrapolando o todo como um intervalo ininterrupto. Os registradores são alocados aos *live intervals* com um percorrimento simples desses intervalos. Quando todos os registradores estão sendo utilizados e um novo *live interval* começar, este novo ou algum dos demais deverá sofrer *spill*. Mantendo a compilação rápida, essa alocação busca também desempenho aceitável para o código gerado: mesmo que não faça as escolhas mais otimizadas para geração de código de *spill*, a aplicação da técnica *linear scan* reduz significativamente o tempo de compilação se comparada à coloração de grafos [10].

A política de *spill* apresentada pelo algoritmo *linear scan* tradicional [10], chamada de *spill everywhere*, escolhe um *live interval* para ser inteiramente movido para a memória principal. Cada uso da variável será precedido de uma instrução `load` e cada definição será seguida de `store` [13, 12]. Essa abordagem para geração de *spill code* muito simples pode ser aprimorada.

Das diversas técnicas para redução de *spill code* serão destacadas duas: *interference region spilling* [14] e *live range splitting* [15]. Ambas observam como as interferências entre variáveis se faz presente no código para, em face da decisão de *spill*, reduzir os acessos a memória adicionados e seu impacto. Em várias ocasiões, essas técnicas contornam interferências com menos código de *spill*.

Interference region spilling busca mover variáveis para memória principal somente na região de interferência delas, quando o *spill* realmente é necessário. Assim, quando for possível alocar parte do *live range* em registradores, essa técnica insere menos operações `load` e `store` que *spill everywhere* [14].

Live range splitting tem o mesmo objetivo, buscando mover variáveis para a memória por mais tempo, quando elas não estiverem em uso e registradores forem necessários para outras operações. Essa técnica corta o tempo de vida em que os dados ainda aguardam para liberar registradores para tempos de vida mais curtos [15].

Assim, este trabalho busca reduzir a geração de *spill code* na alocação de registradores *linear scan* pela sua integração com as técnicas de *interference region spilling* [14] e *live range splitting* [15]. Pretende-se, assim, obter um alocador rápido com uma política de *spill* melhor que o algoritmo tradicional, com a abordagem *spill everywhere*.

A melhoria de desempenho visada será avaliada experimentalmente pela comparação de tempo de execução dos *benchmarks* compilados, de inserção de instruções de acesso a memória neles e de tempo de compilação. Procura-se baixo tempo de compilação, menos inserção de instruções de *spill* e menor tempo de execução com o aprimoramento do alocador, comparando-o com o algoritmo *linear scan* tradicional e outras ferramentas de construção de compiladores disponíveis.

A seguir, no Capítulo 2, são abordadas as técnicas empregadas no desenvolvimento de alocadores de registradores, sendo o *linear scan* destacado no Capítulo 3. O Capítulo 4 detalha as políticas de geração de código de *spill* e o Capítulo 5 implementadas neste trabalho. No Capítulo 6 serão apresentados os próximos passos da pesquisa.

2 ALOCAÇÃO DE REGISTRADORES

A importância e complexidade da alocação de registradores faz com que ela costume ser implementada como uma etapa separada dentre as várias tarefas do gerador de código [12]. Geralmente é abordada como um problema de coloração de grafos ou de empacotamento e demanda muitas heurísticas para obter soluções satisfatórias em tempo aceitavelmente curto [6]. Os alocadores por coloração de grafos são diversos, havendo muitas melhorias nas heurísticas e políticas de *spill* desde a primeira proposta de Chaitin [16, 12].

O método de coloração de grafos consiste em construir o grafo de interferência com base nos *live ranges* das variáveis, simplificá-lo ao máximo e tentar colorir os nós (ou seja, atribuir registradores às variáveis). Se não há cores suficientes para todos os nós, heurísticas devem indicar qual *live range* é menos custoso de sofrer *spill* (podendo ser adotadas várias políticas diferentes para isso) e então ele deve ser armazenado e carregado da memória principal. Havendo *spill*, deve-se recomputar o *live range* fragmentado da variável e reconstruir o grafo de interferência (já que os pequenos trechos em que a variável é utilizada em operações e atualizada podem interferir com outros *live ranges*) e recomeçar o processo. Quando não houver mais necessidade de *spill*, a coloração será bem sucedida e é encerrada [12]. A estrutura do alocador por coloração de grafos está na Figura 3.

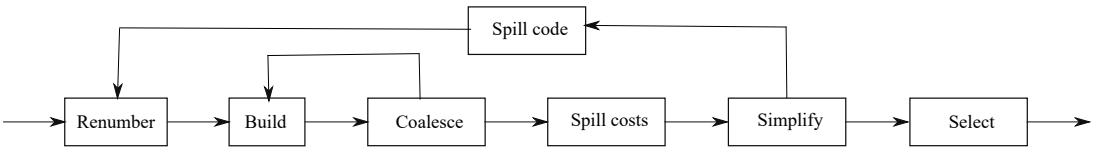

Figura 3 – Alocador proposto por Chaitin [16]

Fonte: adaptado de Briggs et al. [17].

Renumber é uma renomeação aplicada aos *live ranges* para que eles sejam os mais adequados. Os novos *live ranges* serão compostos por definições e pelos usos que elas conseguem alcançar [17]. Caso, no código original, haja uma variável com algumas definições e usos isolados, esse trecho seria um *live range* separado, permitindo que ele receba outra cor, caso seja preciso.

Build refere-se à construção do grafo propriamente dito. É recomendada o uso simultâneo da matriz e da lista de adjacências, conferindo-se rapidamente na matriz a existência de aresta (interferência) entre dois vértices e sendo simples o percorrimento das arestas do grafo [16, 17].

Coalesce é uma das várias estratégias que alocadores por coloração podem empregar para simplificar o grafo de interferências. Quando o valor de uma variável é copiado

para outra e essas variáveis não têm interferência, pode-se remover a operação de cópia e juntá-las em uma só. No grafo, os dois nós são substituídos por um, com a união de todas as interferências deles. Essa estratégia reduz a quantidade de instruções `move`, contudo, o novo nó pode ter mais restrições e exigir mais cores. Para isso, o *coalescing* conservativo somente irá juntar variáveis se o nó unido resultante tiver menos adjacências com nós de grau significativo[1] do que registradores disponíveis. Assim, esse nó adicionado não irá alterar a colorabilidade do grafo, pois poderá ser retirado na simplificação [18, 17].

São calculadas estimativas dos custos de *spill* na etapa de *spill costs* para caso seja necessária decisão de *spill*. Os custos estimados levam em consideração definições e usos de cada variável, com peso 10 vezes maior para cada nível de encadeamento de laços de repetição.

Simplify irá remover os vértices e empilhá-los. Busca-se sempre o nó com o menor grau (menos arestas) dentre os nós com grau menor que a quantidade de registradores disponíveis. Esses são garantidamente coloríveis, afinal, mesmo que todos os nós com quem interferir tenham cores diferentes, ainda haverá cor disponível.

Se, em algum momento, todos os nós tenham grau maior ou igual a quantidade de registradores disponíveis, será preciso uma decisão de potencial *spill* com base nos custos calculados anteriormente. Assim, o *spill code* é gerado nos locais necessários para permitir que o grafo seja reconstruído. Se a simplificação conseguir remover todos os vértices, o grafo poderá ser colorido com os registradores disponíveis [17].

Select seleciona o registrador (a cor) de cada *live range* desempilhado conforme a simplificação (de trás para frente) [17].

Mesmo que a alocação por coloração de grafos gere código de alta qualidade, computacionalmente ela é muito cara, criando demanda para métodos mais simples e rápidos para quando a qualidade do código gerado puder ser reduzida em prol de limitações computacionais mais rígidas [10].

Um algoritmo de alocação com baixo custo computacional é dedicar cada registrador disponível para uma variável e as restantes mover para a memória principal. Pode-se contar os usos e definições das variáveis com uma passagem pelo código intermediário e atribuir as mais utilizadas para os registradores.

Outro algoritmo leve e rápido é o *linear scan* [10], que será apresentado a seguir e terá o foco do presente trabalho. Ele não é baseado em coloração de grafos e, assim, busca acelerar a alocação. Com isso, sua aplicação é direcionadas para tarefas em que tempo é crítico e que pode-se sacrificar em parte a qualidade da alocação.

[1] Nós com mais adjacências do que registradores disponíveis.

3 *LINEAR SCAN*

Proposto por Poletto et al. [19], o algoritmo *linear scan* escaneia os *live intervals* e aloca os registradores em tempo linear. Seu objetivo é o equilíbrio entre alocar rapidamente os registradores e gerar código de bom desempenho.

Primeiramente, devem ser consideradas as instruções em uma ordenação arbitrária. O algoritmo não está restrito a um método específico, mas a ordem das instruções escolhida reflete na qualidade da alocação [10]. Os autores utilizam a ordem de acesso de uma busca em profundidade na estrutura do programa, mas comentam que a ordem em que estão apresentadas as instruções na representação intermediária do programa gera código de qualidade muito próxima de usar a busca em profundidade [10].

Após ser fixada uma ordenação, são identificados os *live intervals* das variáveis. O *live interval* é definido por conter todas as instruções nas quais a sua variável esteja viva. Podem ser indicados o início e o fim pelos números i e j da sua primeira e última instrução, de acordo com a ordem adotada. Dessa maneira, para toda instrução k em que a variável estiver viva, $i \leq k \leq j$ [10].

Vários intervalos contém todas as instruções nas quais uma variável está viva, mas o intervalo mais adequado contém o mínimo possível de instruções. Se o intervalo for maior do que o necessário, o algoritmo poderá ter mais dificuldade para alocar os registradores. De fato, o programa inteiro é um *live interval* válido para todas as variáveis, mas não adiciona nenhuma informação útil para uma política de alocação [10]. O menor intervalo começa com a primeira instrução i em que a variável em questão esteja viva, ou seja, $i \leq k$ para toda instrução k em que a variável esteja viva, de acordo com a ordenação escolhida. Da mesma forma, esse intervalo mínimo termina com a última instrução j em que a variável esteja viva: $j \geq k$ para todo k em que ela esteja viva. Esse intervalo, o mais adequado, não pode ser reduzido mais sem deixar de conter todo o tempo de vida da variável (e, portando, violar sua própria definição).

Evidentemente, podem haver instruções no intervalo que a variável não esteja viva. Isso pode ocorrer entre um uso e uma definição da variável ou por causa da ordenação das instruções.

A identificação dos *live intervals* exige somente uma leitura do código intermediário. Nenhum dos dois métodos de ordenação comentados demanda muita computação: a apresentação das instruções no programa é a ordem fornecida pelo próprio código intermediário e o acesso em profundidade percorre cada instrução somente uma vez.

Com a informação dos *live intervals*, o *linear scan* então os ordena por início crescente ou por final decrescente. O algoritmo passa pelos intervalos nessa ordem, con-

siderando os intervalos ativos a cada ponto. Na Figura 4 estão ilustrados intervalos de variáveis. Os pontos enumerados foram utilizados pelo critério de ordenação.

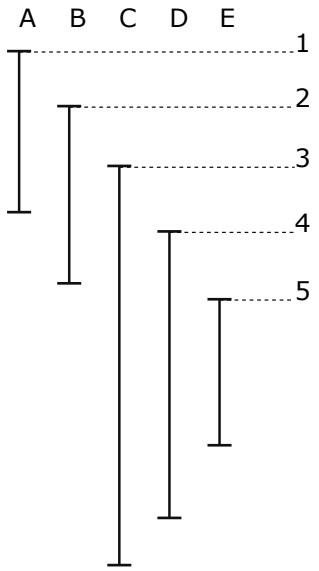

Figura 4 – *Live intervals* das variáveis A a E com os pontos de início enumerados em ordem crescente.

Fonte: adaptado de Poletto e Sarkar [10].

Durante essa passagem pelos intervalos são realizadas as decisões de *spill*. É utilizada uma lista de intervalos ativos para se identificar quando o *spill* é necessário. Nessa lista é adicionado cada *live interval* que iniciar e retirados os que terminarem. Ela só se altera nos pontos extremos dos intervalos, quando variáveis começam a exigir memória ou deixam de utilizar registradores. Assim o *linear scan* passa pelos pontos extremos[1] dos *live intervals*, adicionando os intervalos que se iniciam e retirando os que terminaram antes.

No momento em que a quantidade de intervalos ativos superar a quantidade de registradores disponíveis, pelo menos um intervalo deverá sofrer *spill*. A heurística descrita por Poletto e Sarkar [10] escolhe o intervalo que termina por último. Essa heurística muito simples permite uma rápida decisão de *spill*[2] e busca evitar que mais decisões sejam necessárias, tentando reduzir a quantidade de intervalos movidos para memória. O *spill* de menos intervalos não significa necessariamente que esses intervalos tivessem o menor custo de *spill*.

Variáveis movidas para a memória sofrem *spill everywhere* [10]. Elas residem na memória principal durante todo o *live interval*, devendo ser carregadas para todos os usos

[1] Que tiverem sido utilizados pelo critério de ordenação: os pontos de início crescentes ou os pontos finais decrescentes.

[2] Como a lista de intervalos ativos está ordenada por final, é facilitada a remoção de intervalos que já terminaram e a escolha do *live interval* mais longevo. Para a primeira, pode-se parar a busca por intervalos terminados ao se encontrar um que ainda estiver ativo. Para a segunda, é necessária somente a comparação entre o último intervalo da lista cheia e o novo intervalo que tentou-se inserir nela.

e armazenadas novamente a cada redefinição.

No exemplo da Figura 4, com somente dois registradores disponíveis, os seguintes passos ocorreriam (a lista após cada passo é apresentada entre $\langle$ e $\rangle$):

1. A é adicionado à lista de intervalos ativos: $\langle A \rangle$ (Figura 5);

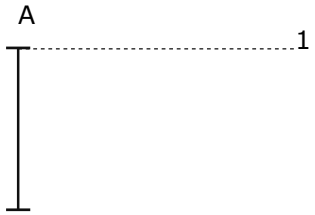

Figura 5 – *Live interval* A ativo na lista.

Fonte: adaptado de Poletto e Sarkar [10].

2. B é adicionado à lista: $\langle A, B \rangle$ (Figura 6);

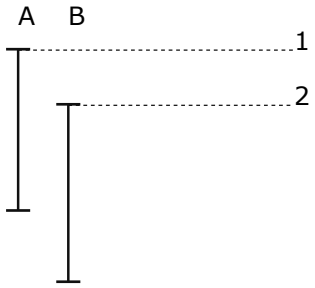

Figura 6 – *Live intervals* A e B ativos na lista.

Fonte: adaptado de Poletto e Sarkar [10].

3. Uma decisão de *spill* é necessária. C termina por último, então sofre *spill* e não entra na lista: $\langle A, B \rangle$ (Figura 7);

4. A é removido da lista e D é adicionado: $\langle B, D \rangle$ (Figura 8);

5. B é removido da lista e E é adicionado: $\langle E, D \rangle$ (Figura 9).

A escolha do *live interval* C para *spill* talvez não seja ideal. É possível que a variável C seja usada e definida várias vezes dentro de laços de repetição, tornando seu *spilling* mais custoso. Também é possível que esse longo intervalo represente uma variável que tenha poucos usos e definições. A heurística que selecionou C não leva em consideração o custo de *spill*.

Pode-se observar que o maior comprimento que a lista de intervalos ativos poderia ter assumido é 3, mas a limitação de 2 registradores não permitiu a inserção de C, que sofreu *spill*. Mesmo que C tivesse sido inserido na lista, nos passos seguintes as inserções

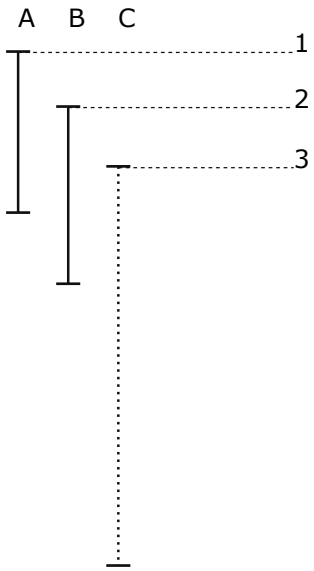

Figura 7 – *Live intervals* A e B ativos na lista, C sofreu *spill*.

Fonte: adaptado de Poletto e Sarkar [10].

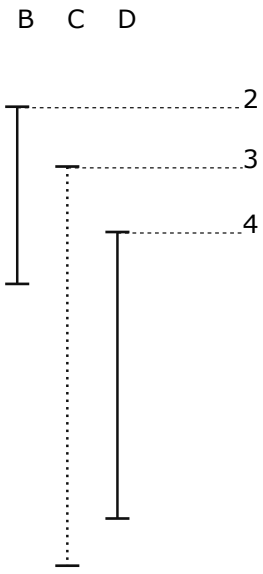

Figura 8 – *Live intervals* B e D ativos na lista, C sofreu *spill*.

Fonte: adaptado de Poletto e Sarkar [10].

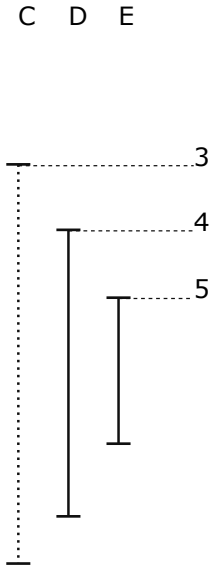

Figura 9 – *Live intervals* D e E ativos na lista, C sofreu *spill*.

Fonte: adaptado de Poletto e Sarkar [10].

são acompanhadas de remoções, o que não aumentaria a lista além disso. Então, ao aplicar o *linear scan* com três registradores, não haverá necessidade de *spilling*:

1. A é adicionado à lista de intervalos ativos: $\langle A \rangle$ (Figura 5);
2. B é adicionado à lista: $\langle A, B \rangle$ (Figura 6);
3. C é adicionado à lista: $\langle A, B, C \rangle$ (Figura 10);

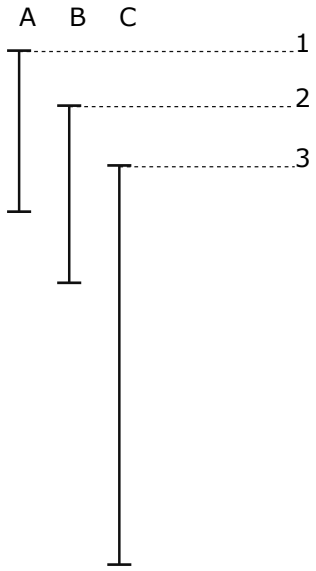

Figura 10 – *Live intervals* A, B e C ativos na lista.

Fonte: adaptado de Poletto e Sarkar [10].

4. A é removido da lista e D é adicionado: $\langle B, D, C \rangle$ (Figura 11);

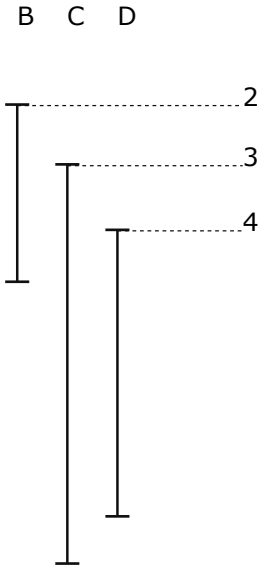

Figura 11 – *Live intervals* B, C e D ativos na lista.

Fonte: adaptado de Poletto e Sarkar [10].

5. B é removido da lista e E é adicionado: $\langle E, D, C \rangle$ (Figura 12).

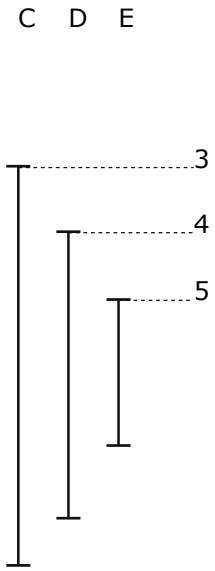

Figura 12 – *Live intervals* C, D e E ativos na lista.

Fonte: adaptado de Poletto e Sarkar [10].

Há três momentos em que três intervalos estão ativos simultaneamente: entre o começo de C e o final de A, entre o começo de D e o final de B e durante todo o intervalo E. Somente C está ativo nesses três momentos. Assim, no exemplo com somente dois registradores, se C não fosse escolhido para *spill*, seria necessário *spilling* de mais do que um intervalo (como A e D ou B e E). É possível que o custo total de *spill* desses intervalos seja menor que o custo de C.

Independente da heurística utilizada, os intervalos escolhidos para *spill* seriam inteiramente movidos para a memória principal. Em toda a sua duração, usos e definições precisariam acessar memória. Esse *spill everywhere* seria necessário somente por causa de alguns momentos de maior interferência. No Capítulo 4 serão abordadas técnicas para reduzir a quantidade de acessos necessários.

3.1 Exemplos de alocação por *linear scan*

Os códigos curtos a seguir serão utilizados em exemplos para ilustrar a aplicação do algoritmo *linear scan*. A Figura 13 representa um algoritmo com cinco variáveis para serem alocadas. As linhas estão enumeradas à esquerda e, entre `{` e `}` estão listadas as variáveis ativas depois de cada instrução. Os valores das variáveis `a` e `b` são parâmetros.

```
                                                { a b }
 0              if (a > b) goto A_maior         { a b }
 1              c = b                          { a b c }
 2              goto C_maior                   { a b c }
 3  A_maior:    c = a                          { a b c }
 4  C_maior:    d = c * a                      { a b d }
 5  A_positivo: d = d * b                      { a b d }
 6              a = a - 1                      { a b d }
 7              if (a > 0) goto A_positivo     { a b d }
 8              e = 2                           { d e }
 9  D_maior:    d = d / e                       { d e }
10              e = e + 1                       { d e }
11              if (d > e) goto D_maior         { d e }
```

Figura 13 – Primeiro algoritmo de exemplo: as variáveis entre chaves do lado direito estão vivas após a instrução.

Fonte: autoria própria.

Ordenação

Há várias possibilidades para a escolha da ordem para linearizar o código. No caso do algoritmo da Figura 13, a busca em profundidade[3] resulta na mesma ordem em que as instruções já estão apresentadas no código.

Se o algoritmo tiver suas linhas reordenadas, mas a sequência de execução mantida pela adição de instruções `goto`, a busca em profundidade obteria essa mesma ordem apresentada na Figura 13.

[3] Visitando primeiro as ramificações dos casos positivos das estruturas condicionais.

Live intervals

Para identificar os *live intervals* é necessário identificar quais variáveis estão vivas a cada instante. Na Figura 14 estão marcados em vermelho os intervalos em que as variáveis estão vivas.

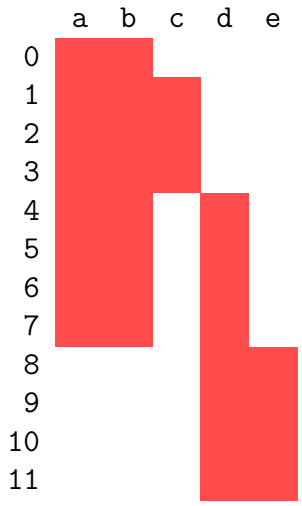

Figura 14 – *Live intervals* do primeiro algoritmo de exemplo

Fonte: autoria própria.

A identificação dos menores *live intervals* possíveis evita *spill* desnecessário (deixando de reservar trechos de código para uma variável morta). Ajustar *live intervals* depende de conhecer a primeira e a última instruções em que a variável esteja viva. O menor intervalo possível incluirá somente elas e as variáveis entre elas. A Figura 14 já representa os menores intervalos válidos para as variáveis de `a` a `e`:

- `a`: $[0, 7]$
- `b`: $[0, 7]$
- `c`: $[1, 3]$
- `d`: $[4, 11]$
- `e`: $[8, 11]$

Passagem

A passagem que o *linear scan* realiza pelos pontos extremos dos *live intervals* pode ser feita em ordem crescente de início ou decrescente de final.

Considerando apenas 2 registradores e os pontos de início, a alocação exige *spill*:

1. `a` é adicionado à lista de intervalos ativos: `{ a }`;

2. `b` é adicionado à lista: `{ a b }`;
3. Uma decisão de *spill* é necessária. `b` termina por último, então sofre *spill* e é removido da lista. *c* é adicionado à lista: `{ c a }`;
4. `c` é removido e `d` é adicionado à lista: `{ a d }`;
5. `a` é removido e `e` é adicionado à lista: `{ d e }`.

Considerando 3 registradores e os pontos de início, a alocação será dada por:

1. `a` é adicionado à lista de intervalos ativos: `{ a }`;
2. `b` é adicionado à lista: `{ a b }`;
3. `c` é adicionado à lista: `{ c a b }`;
4. `c` é removido e `d` é adicionado à lista: `{ a b d }`;
5. `a` e `b` são removidos e `e` é adicionado à lista: `{ d e }`.

Em nenhum momento houve mais do que 3 registradores na lista de intervalos ativos, então não houve nenhum *spill*.

O mesmo resultado é atingido se forem considerados os pontos de fim, ordenando a lista de intervalos por ponto de início decrescente:

1. `d` é adicionado à lista de intervalos ativos: `{ d }`;
2. `e` é adicionado à lista: `{ e d }`;
3. `e` é removido e `a` é adicionado à lista: `{ d a }`;
4. `b` é adicionado à lista: `{ d a b }`;
5. `d` é removido e `c` é adicionado à lista: `{ c a b }`.

Variação do primeiro algoritmo com mais interferência

A Figura 15 Altera a última condição no algoritmo da Figura 13 para usar novamente a variável `c`.

O novo *live interval* da variável `c` interfere agora com `d` e `e`, como pode ser observado na Figura 16.

Assim, a passagem (pelos pontos iniciais crescentes) exigiria uma escolha de *spill*:

```
                                                  { a b }
 0              if (a > b) goto A_maior           { a b }
 1              c = b                            { a b c }
 2              goto C_maior                     { a b c }
 3  A_maior:    c = a                            { a b c }
 4  C_maior:    d = c * a                       { a b c d }
 5  A_positivo: d = d * b                       { a b c d }
 6              a = a - 1                       { a b c d }
 7              if (a > 0) goto A_positivo      { a b c d }
 8              e = 2                            { c d e }
 9  D_maior:    d = d / e                        { c d e }
10              e = e + 1                        { c d e }
11              if (c < d) goto D_maior          { c d e }
```

Figura 15 – Segundo algoritmo de exemplo: as variáveis entre chaves do lado direito estão vivas após a instrução.

Fonte: autoria própria.

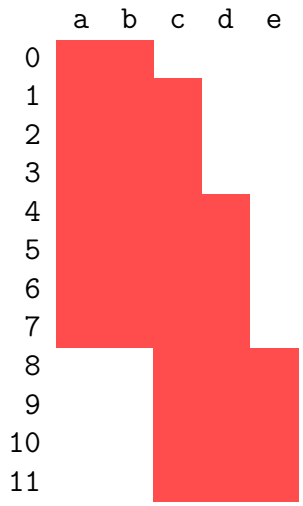

Figura 16 – *Live intervals* do segundo algoritmo de exemplo

Fonte: autoria própria.

1. `a` é adicionado à lista de intervalos ativos: `{ a }`;
2. `b` é adicionado à lista: `{ a b }`;
3. `c` é adicionado à lista: `{ a b c }`;
4. Uma decisão de *spill* é necessária. `c` termina por último, então sofre *spill* e é removido da lista. `d` é adicionado à lista: `{ a b d }`;
5. `a` e `b` são removidos e `e` é adicionado à lista: `{ d e }`.

Como `c` e `d` terminam ao mesmo tempo, a decisão de *spill* poderia ter sido tomada diferentemente, já que a heurística não define critério de desempate. Mas, como não

começam ao mesmo tempo, a passagem considerando os pontos finais decrescentes teria decisão definitiva:

1. `d` é adicionado à lista de intervalos ativos: `{ d }`;
2. `e` é adicionado à lista: `{ e d }`;
3. Uma decisão de *spill* é necessária. `c` começa primeiro, então sofre *spill* e não é adicionado à lista: `{ e d }`;
4. `e` é removido e `a` é adicionado à lista: `{ d a }`;
5. `b` é adicionado à lista: `{ d a b }`.

4 GERAÇÃO DE *SPILL CODE*

Ao identificar que os registradores disponíveis não serão suficientes para todas as variáveis, o alocador deve escolher quais delas (e em quais partes do código) deverão ser armazenadas na memória principal e carregadas dela.

A política de *spill everywhere* proposta para o alocador de Chaitin [16] é bastante simples: para uma variável que sofreu *spill* é adicionado um `load` antes de todo uso e um `store` depois de toda definição. As figuras 17 e 18 exemplificam, respectivamente, da aplicação dessa política com 2 registradores disponíveis nos *live ranges* de `b` e de `a`, alterando o código da Figura 13. Pela proposta do algoritmo *linear scan* [10], tanto `a` quando `b` poderiam ser escolhidos para *spill*, já que ambas têm a mesma duração.

```
                                          { a b }
 0              store b                    { a }
 1              load b                    { a b }
 2              if (a > b) goto A_maior    { a }
 3              load b                    { a b }
 4              c = b                     { a c }
 5              goto C_maior              { a c }
 6  A_maior:    c = a                     { a c }
 7  C_maior:    d = c * a                 { a d }
 8  A_positivo: load b                   { a b d }
 9              d = d * b                 { a d }
10              a = a - 1                 { a d }
11              if (a > 0) goto A_positivo { a d }
12              e = 2                     { d e }
13  D_maior:    d = d / e                 { d e }
14              e = e + 1                 { d e }
15              if (d > e) goto D_maior   { d e }
```

Figura 17 – *Spill everywhere* de `b` no primeiro algoritmo de exemplo.

Fonte: autoria própria.

Como foi apresentado pela Figura 3, o alocador de Chaitin [16] estima custos de *spill* antes de decidir quais *live ranges* sofrerão *spill*. Esses custos calculados são a soma da quantidade de usos e definições com pesos individuais, totalizando o tempo de execução adicional causado pelo *spill* [16]. Cada instrução terá como peso a estimativa de repetições $10^d c$, onde c é o custo dessa instrução individual na arquitetura-alvo e d é a quantidade de laços de repetição que a incluem (o nível de encadeamento) [17]. Com essa heurística é possível orientar o *spill* para as variáveis com menos acessos previstos para uma execução. Dividindo-se o custo projetado pela quantidade de interferências (grau do vértice), se tem a heurística da Equação 4.2.

```
                                                { a b }
 0              store a                          { b }
 1              load a                          { a b }
 2              if (a > b) goto A_maior          { b }
 3              c = b                           { b c }
 4              goto C_maior                    { b c }
 5              load a                         { a b c }
 6  A_maior:    c = a                           { b c }
 7  C_maior:    load a                         { a b c }
 8              d = c * a                       { b d }
 9  A_positivo: d = d * b                       { b d }
10              load a                         { a b d }
11              a = a - 1                      { a b d }
12              store a                         { b d }
13              load a                         { a b d }
14              if (a > 0) goto A_positivo      { b d }
15              e = 2                           { d e }
16  D_maior:    d = d / e                       { d e }
17              e = e + 1                       { d e }
18              if (d > e) goto D_maior         { d e }
```

Figura 18 – *Spill everywhere* de `a` no primeiro algoritmo de exemplo.

Fonte: autoria própria.

$$\text{custo}(v) = \sum_{i \in \text{instruções em que } v \text{ está viva}} 10^{d_i} c_i \tag{4.1}$$

$$h_0(v) = \text{custo}(v)/\text{grau}(v) \tag{4.2}$$

Essa divisão dá valor a vértices que, se fossem removidos, reduziriam mais o grau outros vértices.

Muitas pesquisas foram desenvolvidas a respeito dessa escolha e de como alterar o código do programa. Muitas melhorias à política de *spill everywhere* foram propostas, como as diferentes heurísticas e a limpeza (abordagem *spill almost everywhere*) de Bernstein et al. [13] e a rematerialização de valores cuja recomputação é mais barata que a transferências em memória [17].

Bernstein et al. [13] apresenta heurísticas para estimar os custos de *spill* com menos incertezas. São utilizadas as definições de largura(i), a quantidade de variáveis vivas na instrução i e area(i) (Equação 4.3), o custo considerando a largura.

$$\text{area}(v) = \sum_{i \in \text{instruções em que } v \text{ está viva}} \text{largura}(i) 10^{d_i} c_i \tag{4.3}$$

Assim, as heurísticas de Bernstein são 4.4, 4.5 e 4.6, sendo variáveis escolhidas para *spill* as melhores das três heurísticas:

$$h_1(v) = \text{custo}(v)/\text{grau}(v)^2 \tag{4.4}$$

$$h_2(v) = \text{custo}(v)/(\text{area}(v)\text{grau}(v)) \tag{4.5}$$

$$h_3(v) = \text{custo}(v)/(\text{area}(v)\text{grau}(v)^2) \tag{4.6}$$

Outra proposta de Bernstein et al. é remover instruções desnecessárias depois da geração do código de *spill*. Essa técnica, denominada *spill almost everywhere*, remove parte das operações custosas de *spill* que permaneceriam se dependesse de *spill everywhere* [13].

Rematerialização é uma técnica para evitar *spill* que, quando é possível e menos custoso, redefine a variável pelo mesmo valor quando ele for acessível, como uma atribuição de constante literal ou um cálculo cujos valores ainda são garantidamente os mesmos. Ela depende de acompanhamento atento às definições que atingem cada uso e se os valores das quais elas dependem ainda estão disponíveis (como outras variáveis que ainda não mudaram ou constantes) [20].

Além dessas técnicas mencionadas, há políticas que precisam de pouco processamento para reduzir significativamente a geração de código de *spill*. As duas técnicas escolhidas serão apresentadas no próximo capítulo.

5 DAS TÉCNICAS DE REDUÇÃO DE *SPILL CODE* ESCOLHIDAS

Devido a sua simplicidade e compatibilidade com *linear scan*, foram selecionadas para este trabalho as políticas para reduzir *spilling* de *interference region spilling* e *live range splitting*, descritas a seguir. Ambas evitam *spill everywhere* restringindo a maneira de gerar *spill code* somente quando ele é necessário. As informações que elas utilizam já estão presentes nas análises do *linear scan* ou são facilmente computáveis. Essas técnicas não dependem do método de coloração de grafos, podendo ser aplicadas em outros tipos de alocadores como, neste trabalho, no *linear scan*.

Elas são relacionadas entre si pela ideia de gerar código de *spill* mais preciso, somente onde é necessário. Ambas adotam a estratégia *spill everywhere* quando não conseguem se provar benéficas (comparando seus custos) Não são técnicas antagônicas e podem ser utilizadas conjuntamente, escolhendo a menos custosa das duas, ou mesmo a própria *spill everywhere*.

5.1 *Interference region spilling*

A técnica de *interference region spilling* [14] consiste em observar, entre dois *live ranges* que se interferem, em qual parte do programa estão ativos simultaneamente. Além de identificar a interferência entre dois *live ranges*, identifica-se também a região de interferência. A eliminação da interferência entre os *live ranges* será possível pelo *spill* parcial de um deles, sendo necessário que seja movido para a memória somente nessa região de interferência. A Figura 19 mostra a diferença entre *spill everywhere* (no centro) e *interference region spill* (à direita).

A técnica de *spill everywhere* não utiliza a informação da região de interferência. Se houver a necessidade para *spill*, um *live range* inteiro sofrerá *spill*, mesmo que sua única interferência problemática fosse com um *live range* que compartilha a atividade em um pequeno trecho do programa. Na Figura 19, a região de interferência entre A e B é somente o final de A e o começo de B, partes menores que as regiões em que estão ativos sozinhos.

Para realizar *spill*, são inseridas na região de interferência as mesmas instruções `load` e `store` que seriam inseridas por *spill everywhere* e, além dela, seriam evitadas várias inserções. Assim, essa técnica geralmente insere menos `load` e `store` que *spill everywhere*. Quando a região de interferência contém um *live range* inteiro, o resultado de *interference region spilling* será exatamente o mesmo de *spill everywhere*.

Existe a possibilidade do custo de *interference region spilling* superar o custo de

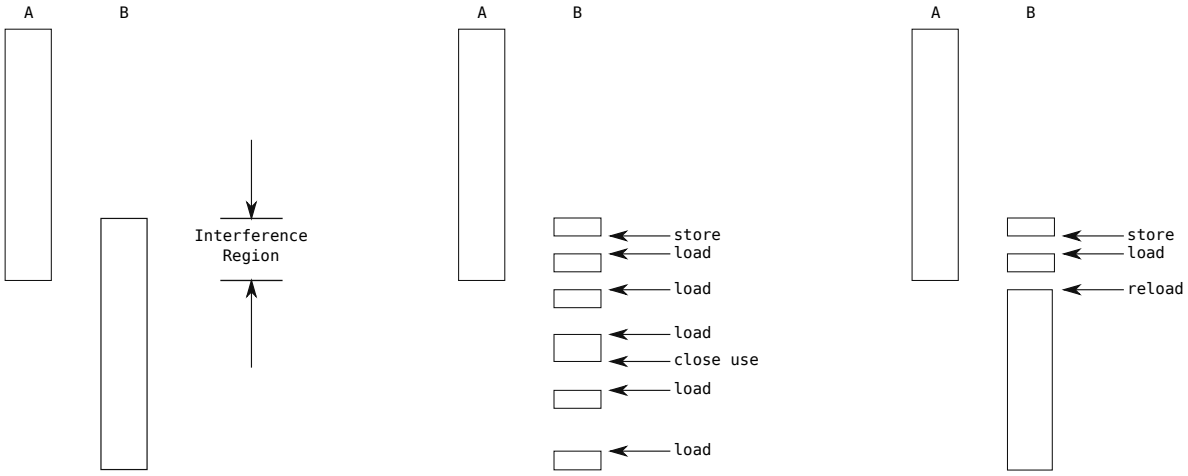

Figura 19 – À esquerda, os *live ranges* das variáveis A e B, identificada a região de interferência. Ao centro, B sofre *spill everywhere*. À direita, B sofre *spill* somente na região de interferência.

Fonte: Bergner, Dahl, Engebretsen e O'Keefe [14].

spill everywhere por causa dos usos da variável depois do fim da região de interferência. Se for necessário recarregar a variável para ser utilizada após a região de *spill*, é possível que sejam inseridas instruções `load`[1] que talvez não seriam inseridas por *spill everywhere*. Para evitar que esses carregamentos adicionais sejam piores que o *spill* completo, o alocador escolhe a abordagem de *spill everywhere* quando os custos de *interference region spilling* forem maiores.

5.1.1 Exemplos

A aplicação de *interference region spilling* no primeiro código de exemplo (Figura 13) adiaria que as instruções `store` e `load` somente algumas instruções, já que `c` surge logo após o começo. Na Figura 20, `b` é utilizado na linha 1 tal qual no código original, porém logo depois já é necessário armazenar em memória. Como o desvio condicional exigirá uso de `b`, o `store` é adiantado para o começo. Analogamente, é aplicada *interference region spilling* em `a` na Figura 21.

5.2 *Live range splitting*

A técnica de *live range splitting* [15] propõe evitar passivamente a geração de *spill*. São calculados os custos de dividir *live ranges* quando estes forem selecionados para *spill*, verificando se a inserção de `load` e `store` nos pontos de *split* não custarão mais que *spill everywhere*.

Se for benéfica, a divisão permitirá que diferentes partes sejam alocadas para diferentes registradores disponíveis ou que *live ranges* antes conflitantes possam utilizar o

[1] Os autores se referem a elas como `reload`, como na Figura 19.

```
                                          { a b }
 0              store b                   { a b }
 1              if (a > b) goto A_maior   { a b }
 3              load b                    { a b }
 4              c = b                     { a c }
 5              goto C_maior              { a c }
 6  A_maior:    c = a                     { a c }
 7  C_maior:    d = c * a                 { a d }
 8  A_positivo: load b                   { a b d }
 9              d = d * b                 { a d }
10              a = a - 1                 { a d }
11              if (a > 0) goto A_positivo { a d }
12              e = 2                     { d e }
13  D_maior:    d = d / e                 { d e }
14              e = e + 1                 { d e }
15              if (d > e) goto D_maior   { d e }
```

Figura 20 – *Interference region spilling* de `b` no primeiro algoritmo de exemplo.

Fonte: autoria própria.

```
                                          { a b }
 0              store a                   { a b }
 1              if (a > b) goto A_maior    { b }
 2              c = b                     { b c }
 3              goto C_maior              { b c }
 4  A_maior:    load a                   { a b c }
 5              c = a                     { b c }
 6              load a                   { a b c }
 7  C_maior:    d = c * a                 { b d }
 8  A_positivo: d = d * b                 { b d }
 9              load a                   { a b d }
10              a = a - 1                { a b d }
11              store a                   { b d }
12              load a                   { a b d }
13              if (a > 0) goto A_positivo { b d }
14              e = 2                     { d e }
15  D_maior:    d = d / e                 { d e }
16              e = e + 1                 { d e }
17              if (d > e) goto D_maior   { d e }
```

Figura 21 – *Interference region spilling* de `a` no primeiro algoritmo de exemplo.

Fonte: autoria própria.

mesmo registrador. Para isso, é preciso que os pontos de *split* removam interferências ou as dividam entre as partes, que podem ser então coloridas (ou sofrer *spill*) separadamente. A proposta de *live range splitting* também sugere que haja benefícios na sua combinação com *interference region spilling*.

Quando um *live range* está inteiramente contido dentro de outro e for necessário *spill*, qualquer tentativa de aplicar *interference region spilling* no *live range* interno resultará no mesmo código de *spill everywhere*. Porém, quando os *live ranges* têm uma intersecção em somente parte deles, como na Figura 19, *interference region spilling* poderá resolver o problema facilmente.

Live range splitting, por outro lado, consegue dividir o *live range* externo ao redor do interno se não houver nenhuma operação do maior enquanto o menor está vivo. Mas se ambos os *live ranges* tiverem usos ou definições na região de interferência, *live range splitting* não poderá ajudar em nada.

Dessa forma, embora não possam atender a todas as situações em que *spill code* será necessário, juntas essas técnicas conseguem atender uma grande variedade.

No fluxograma à esquerda, a Figura 22 mostra o resultado do *splitting* da variável l_1 ao redor de l_2. Tanto a definição quanto os usos de l_2 ocorrem entre a definição e o uso de l_1, ou seja, o *live range* de l_2 está contido no *live range* de l_1, mas não o contrário (como mostra o grafo de contenção do lado direito da Figura 22). Assim, se l_1 for armazenado antes de se definir l_2 e depois for recarregado depois do fim do uso de l_2, ambos podem utilizar um único registrador, sem causar *spill* dentro de nenhum *loop*.

O grafo de contenção é um detalhamento do grafo de interferências. Um nó x está conectado com y, ou seja, $x \rightarrow y$, se durante o *live range* de y houver usos ou definições de x. Se dois nós estão conectados nos dois sentidos, não haverá como aplicar *live range splitting*.

5.2.1 Exemplos

Considerando o primeiro algoritmo de exemplo (Figura 13), o grafo de contenção da Figura 23 demonstra que não é possível aplicar *live range splitting*, já que todos os intervalos que se interferem estão mutuamente contidos um no outro (ambos possuem usos e definições durante o outro).

A Figura 24 apresenta um código em que é possível (e benéfico) aplicar *live range splitting*. Como nos exemplos anteriores, `a` é um parâmetro e já se inicia vivo.

Pode-se observar que, enquanto `b` está vivo, não há nenhuma instrução que utilize ou defina `a`. Assim, como ilustrado pela Figura 25, `b` está contido em `a`, mas não o contrário.

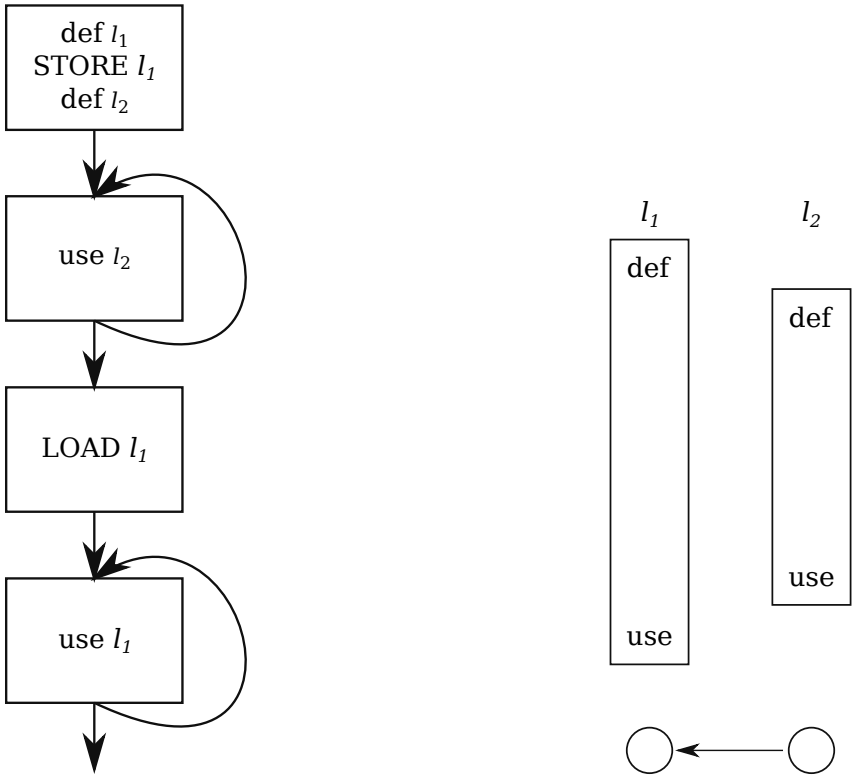

Figura 22 – *Splitting* de l_1 ao redor de l_2 e seus respectivos *live ranges*.

Fonte: adaptado de Cooper e Simpson [15].

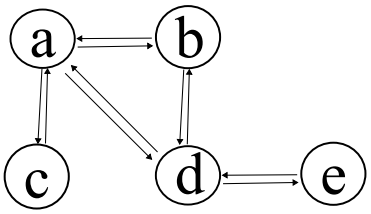

Figura 23 – Grafo de contenção para o primeiro algoritmo de exemplo.

Fonte: autoria própria.

```
                                          { a }
 0              b = a                     { a b }
 1              c = 1                    { a b c }
 2  B_positivo: if (b <= 0) goto B_zero   { a b c }
 3              c = c * b                 { a b c }
 4              b = b - 1                 { a b c }
 5              goto B_positivo           { a b c }
 6  B_zero:     if (a <= 0) goto A_zero    { a c }
 7              c = c - a                  { a c }
 8              a = a - 1                  { a c }
 9              goto B_zero                { a c }
10  A_zero:                                 { c }
```

Figura 24 – Algoritmo para exemplo de *live range splitting*.

Fonte: autoria própria.

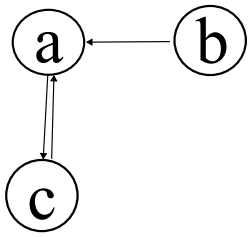

Figura 25 – Grafo de contenção para o exemplo de *live range splitting*.

Fonte: autoria própria.

Se houver somente 2 registradores, seria preciso *spill* de pelo menos um *live range* desse exemplo. O *spill* de qualquer uma das variáveis seria um problema, já que todas são utilizadas dentro de laços de repetição. Para evitar que seja adicionado *spill code* dentro de laços, pode-se aplicar *live range splitting*, obtendo o código da Figura

```
                                          { a }
 0              b = a                     { a b }
 1              store a                    { b }
 2              c = 1                     { b c }
 3  B_positivo: if (b <= 0) goto B_zero   { b c }
 4              c = c * b                 { b c }
 5              b = b - 1                 { b c }
 6              goto B_positivo           { b c }
 7  B_zero:     load a                    { a c }
 8  A_positivo: if (a <= 0) goto A_zero   { a c }
 9              c = c - a                 { a c }
10              a = a - 1                 { a c }
11              goto A_positivo           { a c }
12  A_zero:                                { c }
```

Figura 26 – Código de exemplo com *splitting* de `a` ao redor de `b` e `c`.

Fonte: autoria própria.

5.3 Implementação em um alocador *linear scan*

As técnicas de *interference region spilling* e *live range splitting* foram propostas para alocadores por coloração de grafos, mas não dependem intrinsecamente do algoritmo de coloração para poderem ser implementadas.

Como os *live intervals* são simplificações de *live ranges* (podem ser interpretados como um *live range* em que a variável está viva continuamente do início ao fim), a alocação *linear scan* oferece um cenário mais simples para o funcionamento das técnicas abordadas neste Capítulo.

O grafo de contenção e a região de interferência podem ser identificados em *live intervals* da mesma maneira que em *live ranges*.

A região de interferência entre dois intervalos pode ser mais facilmente identificada como o subintervalo que se inicia no último início e que termina com o primeiro término. Quando intervalos não interferem, o primeiro término antecede o último início. Se um intervalo estiver contido inteiramente dentro do outro, o início e o fim da região de interferência coincidirão com ele.

6 PRÓXIMAS ETAPAS

As próximas tarefas deste trabalho são:

1. Implementação do alocador *linear scan* tradicional;
2. Implementação de *interference region spilling*;
3. Implementação de *live range splitting*;
4. Testes de *benchmark* dos alocadores;
5. Análise dos resultados;
6. Escrita do trabalho final.

Tabela 1 – Cronograma de Execução

Atividade	**Mês**							
	01	02	03	04	05	06	07	08
Levantamento bibliográfico	X	X	X					
Implementação do alocador *linear scan*				X	X	X		
Implementação de *interference region spilling*					X	X		
Implementação de *live range splitting*					X	X		
Testes de *benchmark* dos alocadores						X	X	
Análise dos resultados							X	X
Escrita do trabalho			X	X	X	X	X	X

REFERÊNCIAS

[1] FOMIN, D.; GENKIN, S.; ITENBERG, I. *Círculos Matemáticos: A Experiência Russa*. 1. ed. Rio de Janeiro: IMPA, 2012.

[2] LOUDEN, K. C.; LAMBERT, K. A. *Programming Languages: Principles and Practice*. 3. ed. Boston: Cengage Learning, 2011.

[3] AHO, A. V. et al. (Ed.). *Compilers: principles, techniques, & tools*. 2. ed. Boston: Pearson/Addison Wesley, 2007. OCLC: ocm70775643. ISBN 9780321486813.

[4] SCOTT, M. L. *Programming Language Pragmatics*. 4. ed. Waltham: Morgan Kaufmann, 2016.

[5] NETO, J. J. *Introdução à compilação*. Rio de Janeiro: LTC, 1987.

[6] MUCHNICK, S. S. *Advanced compiler design and implementation*. San Francisco: Morgan Kaufmann Publishers, 1997.

[7] AHO, A. V.; SETHI, R.; ULLMAN, J. D. (Ed.). *Compiladores: princípios, técnicas e ferramentas*. Rio de Janeiro: LTC, 1995.

[8] COOPER, K. D.; TORCZON, L. *Construindo Compiladores*. 2. ed. Rio de Janeiro: Elsevier, 2014.

[9] PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: the hardware/software interface*. 3. ed. Amsterdam, Heidelberg: Elsevier, Morgan Kaufmann, 2005. ISBN 9781558606043 9780120884339.

[10] POLETTO, M.; SARKAR, V. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM New York, NY, USA, v. 21, n. 5, p. 895–913, 1999.

[11] AYCOCK, J. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 35, n. 2, p. 97–113, 2003.

[12] COOPER, K. D.; TORCZON, L. *Engeneering a Compiler*. 3. ed. Cambridge, MA: Elsevier, Morgan Kaufmann, 2023.

[13] BERNSTEIN, D. et al. Spill code minimization techniques for optimizing compliers. *ACM SIGPLAN Notices*, ACM, New York, v. 24, n. 7, p. 258–263, 1989.

[14] BERGNER, P. et al. Spill code minimization via interference region spilling. *ACM SIGPLAN Notices*, ACM, New York, v. 32, n. 5, p. 287–295, 1997.

[15] COOPER, K. D.; SIMPSON, L. T. Live range splitting in a graph coloring register allocator. In: SPRINGER. *International Conference on Compiler Construction*. Berlin, Heidelberg, 1998. p. 174–187.

[16] CHAITIN, G. J. Register allocation & spilling via graph coloring. *ACM Sigplan Notices*, ACM, New York, v. 17, n. 6, p. 98–101, 1982.

[17] BRIGGS, P.; COOPER, K. D.; TORCZON, L. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, New York, v. 16, n. 3, p. 428–455, 1994.

[18] APPEL, A. W. *Modern Compiler Implementation in Java*. 2. ed. New York: Cambridge University Press, 2002.

[19] POLETTO, M.; ENGLER, D. R.; KAASHOEK, M. F. tcc: A system for fast, flexible, and high-level dynamic code generation. *ACM SIGPLAN Notices*, ACM, New York, NY, USA, v. 32, n. 5, p. 109–121, 1997.

[20] BRIGGS, P.; COOPER, K. D.; TORCZON, L. Rematerialization. In: ACM. *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. [S.l.], 1992. p. 311–321.