Universidade Estadual de Londrina

GUILHERME AKIRA DEMENECH MORI

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

**LONDRINA**

**2024**

GUILHERME AKIRA DEMENECH MORI

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

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**
**2024**

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

M854r Mori, Guilherme Akira Demenech .
REDUÇÃO DO SPILL CODE NO ALGORITMO LINEAR SCAN / Guilherme Akira Demenech Mori. - Londrina, 2024.
59 f. : il.

Orientador: Wesley Attrot.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Graduação em Ciência da Computação, 2024.
Inclui bibliografia.

1. Compiladores - TCC. 2. Alocação de Registradores - TCC. 3. Geração de Spill Code - TCC. I. Attrot, Wesley. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Graduação em Ciência da Computação. III. Título.

CDU 519

GUILHERME AKIRA DEMENECH MORI

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

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. Fábio Sakuray
Universidade Estadual de Londrina – UEL

Pedro Zaffalon da Silva
Universidade Estadual de Londrina – UEL

Londrina, 07 de maio de 2024.

*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***. 2024. 59f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2024.

# RESUMO

A alocação de registradores é um processo muito importante para compiladores de programas, porém algoritmos baseados em coloração de grafos são muito lentos, mesmo que gerem código de qualidade. O algoritmo *linear scan* é o alocador alternativo proposto para contextos em que a velocidade deva ser priorizada, como a compilação *just-in-time*. 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*. O *linear scan* tradicional e aprimorado foram comparados experimentalmente entre si e com outras ferramentas de construção de compiladores. A avaliação considerou a quantidade de instruções `load` e `store` adicionadas aos *benchmarks* compilados. A alocação do *linear scan* foi comparável ou melhor que o alocador *fast* disponível no LLVM e foi identificada redução de *spill* gerado com a aplicação da técnica de *live range splitting*. Foram, porém, obtidos poucos dados dos testes com os *benchmarks* SPEC CPU® 2017, pois pouco *spilling* foi necessário.

**Palavras-chave**: *Linear Scan*. *Interference Region Spilling*. *Live Range Splitting*. Redução de *Spill Code*.

MORI, G. A. D.. **Spill code minimization in the linear scan algorithm**. 2024. 59p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2024.

# ABSTRACT

Register allocation is a crucial process for program compilers. However graph-coloring-based algorithms, despite generating high-quality code, are very slow. The linear scan algorithm is an alternative allocator proposed for contexts where speed must be prioritized, such as just-in-time compilation. This work seeks to improve the spill code generation of the linear scan register allocator implementing the techniques of interference region spilling and live range splitting. The traditional and improved linear scan were experimentally compared with each other and other compiler construction tools. The evaluation considered the number of `load` and `store` instructions added to the compiled benchmarks. The linear scan allocation was comparable to or better than the LLVM fast allocator and a spill reduction was identified in the live range splitting technique application. However, few data was obtained from the SPEC CPU® 2017 benchmarks tests due to the low spilling required.

**Keywords**: Linear Scan. Interference Region Spilling. Live Range Splitting. Spill code minimization.

# LISTA DE ILUSTRAÇÕES

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| RAM | *Random Access Memory* |
| JIT | *Just-in-time* |
| LLVM IR | *LLVM intermediate representation* |
| LLVM MIR | *LLVM machine specific intermediate representation* |
| MIPS | *Microprocessor without interlocked pipeline stages* |

# LISTA DE ALGORITMOS

# SUMÁRIO

# 1 INTRODUÇÃO

Compiladores são programas que recebem como entrada código em uma linguagem-fonte de alto nível[1] e o traduzem no código equivalente em uma linguagem-objeto que possa ser executada por computadores [6, 4]. São estruturados como uma sequência de fases que analisam o programa e o traduzem para o formato desejado [7]. A Figura 1 apresenta as fases: as análises léxica, sintática e semântica e a geração de código intermediário são o *front end*, a otimização de código independente de máquina é sozinha o *middle end* e a geração de código de máquina e a otimização de código dependente de máquina são o *back end*. As fases contornadas por linhas contínuas são necessárias e o contorno tracejado indica as fases opcionais.

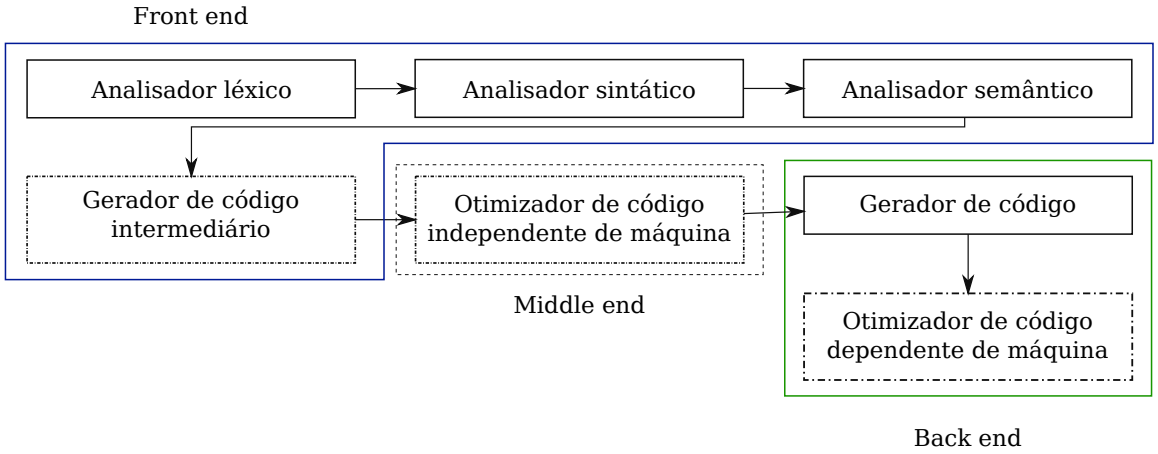


Figura 1 – Fases da compilação

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 [4].

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 ótima é NP-completa[2], sendo sensível às restrições específicas de *hardware* ou sistema operacional [4, 9].

[1] Linguagens de alto nível permitem notações mais abstratas como as utilizadas em ambiente matemático e científico, como Álgebra [3]. 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 [4, 5].

[2] Para os problemas não-determinísticos de tempo polinomial completo (NP-completos) são conhecidas somente soluções com pelo menos tempo exponencial [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 em que 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 [10]. 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.

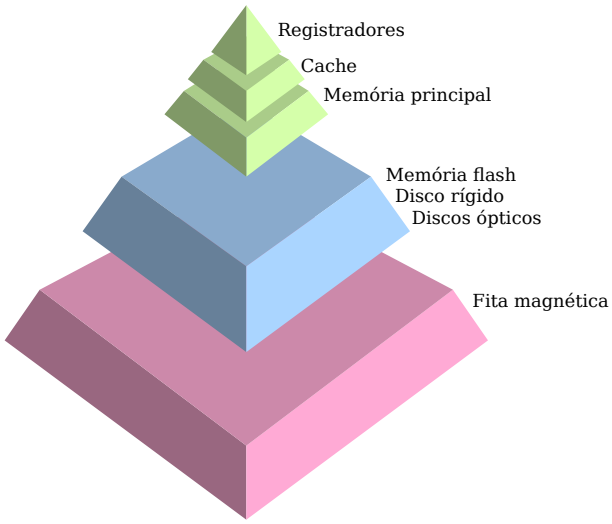


Figura 2 – Pirâmide da hierarquia de memória

Com a grande quantidade de variáveis necessárias pela maior parte dos programas, não há registradores suficientes para dedicar a todas elas [4]. 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] [10].

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 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.

[3] Usualmente traduzido como derramamento.

Em geral, os registradores são alocados e atribuídos como um problema de coloração de grafos [4]: 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 [11]. 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] [13].

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 [13]. 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 [13].

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* [11]. 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* [11] assume uma ordenação das instruções e 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 com-

[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 [12].

pilação 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 [11].

A política de *spill* apresentada pelo algoritmo *linear scan* tradicional [11], 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` [14, 13]. 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* [15] e *live range splitting* [16]. 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* [15].

*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 [16].

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* [15] e *live range splitting* [16]. 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 inserção de instruções de acesso a memória nos *benchmarks* compilados. Além das implementações do algoritmo *linear scan* convencional e aprimorado, também serão avaliados resultados de outros alocadores disponíveis.

O restante deste trabalho está organizado da seguinte forma: no Capítulo 2, são abordadas as técnicas empregadas no desenvolvimento de alocadores de registradores, sendo o *linear scan* destacado na Seção 2.2. O Capítulo 3 detalha as políticas de geração de código de *spill* e a Seção 3.1 aborta especificamente as implementadas neste trabalho. A implementação e os resultados de testes são discutidos no Capítulo 4.

# 2 ALOCAÇÃO DE REGISTRADORES

A importância e complexidade da alocação de registradores faz com que essa ação costume ser implementada como uma etapa separada dentre as várias tarefas do gerador de código [13]. Geralmente é abordada como um problema de coloração de grafos ou de empacotamento, modelos de grande complexidade que demandam muito tempo ou muitas heurísticas para obter soluções satisfatórias [7]. Os alocadores por coloração de grafos são diversos, havendo muitas melhorias nas heurísticas e políticas de *spill* desde a primeira proposta de Chaitin [2, 13], como a coloração baseada em prioridades [17, 18], a coloração otimista [19, 20], a coloração hierárquica [21] e as técnicas discutidas no Capítulo 3.

## 2.1 Coloração de Grafos

Na republicação do artigo original [2], Chaitin [22] atribui o sucesso da abordagem de coloração de grafos para alocação de registradores às ideias matemáticas simples que a baseiam. Ele defende que algoritmos devam ser baseados nessas ideias limpas e compreensíveis ao invés de serem desenvolvidos diretamente como soluções específicas[6]. Essa filosofia também é visível na construção de compiladores em partes dedicadas a problemas bem definidos (como análise sintática, otimização e geração de código), organizadas por propriedades formais e abstratas e não por atributos confusos e específicos do mundo real.

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 [13]. A estrutura do alocador por coloração de grafos está na Figura 3.

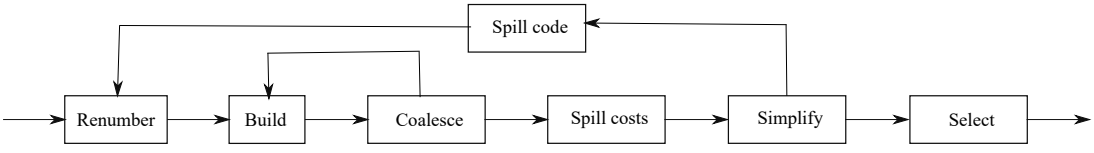


Figura 3 – Alocador proposto por Chaitin [2]

Fonte: adaptado de Briggs et al. [20].

[6] *ad hoc hacking*, nas palavras dele mesmo

*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 [20]. 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 [2, 20].

*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[7] do que registradores disponíveis. Assim, esse nó adicionado não irá alterar a colorabilidade do grafo, pois poderá ser retirado na simplificação [23, 20].

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 [20].

*Select* seleciona o registrador (a cor) de cada *live range* desempilhado conforme a simplificação (de trás para frente) [20].

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

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

computacionais mais rígidas [11].

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* [11], 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.

## 2.2 *Linear scan*

Proposto por Poletto et al. [24], 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 [11]. 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 da utilização de busca em profundidade [11].

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, $i \leq k \leq j$ [11].

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 [11]. 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).

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[8]. O algoritmo passa pelos intervalos nessa ordem, considerando os intervalos ativos a cada ponto. Nas Figuras 4 e 5 estão ilustrados intervalos de variáveis com os pontos enumerados na ordem em que serão visitados. O Algoritmo 1 tem variáveis cujos *live intervals* estão representados na Figura 4.

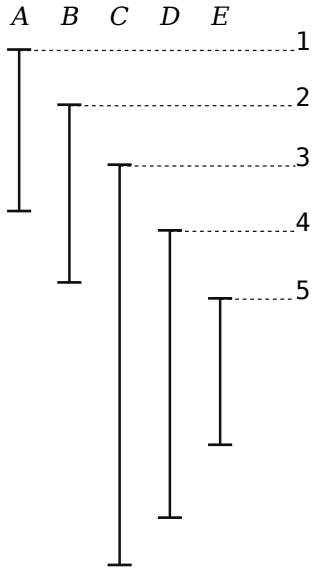


Figura 4 – *Live intervals* das variáveis $A$ a $E$ com os pontos de início enumerados em ordem crescente

Fonte: adaptado de Poletto e Sarkar [11].

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[9]

[8] Os finais dos intervalos em ordem decrescente (como na Figura 5) podem ser interpretados como pontos de parada da mesma forma que os pontos de início em ordem crescente. Muito embora as instruções estejam na sequência inversa (da ordenação arbitrária), o algoritmo *linear scan* é capaz de alocar da mesma maneira: a cada ponto de parada desconsidera os intervalos que não estão mais ativos dali em diante e tenta adicionar o novo intervalo, caso haja registrador disponível.

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

**Algoritmo 1:** Código de exemplo com as variáveis $A$ a $E$ com os *live intervals* da Figura 4

```
 0       A = 3                 { A }
 1       B = 2                { A B }
 2       C = 1               { A B C }
 3  f:   C = C * A           { A B C }
 4       A = A - 1           { A B C }
 5       if (A > 0) goto f   { A B C }
 6  g:   D = B * 2           { B C D }
 7       C = C + D           { B C D }
 8       B = B - 1           { B C D }
 9       if (B > 0) goto g   { B C D }
10       E = D + 1           { C D E }
11       C = C * E            { C D }
12  h:   C = C * D            { C D }
13       D = D - 2            { C D }
14       if (D > 0) goto h    { C D }
15       print C               { C }
16       return C
```

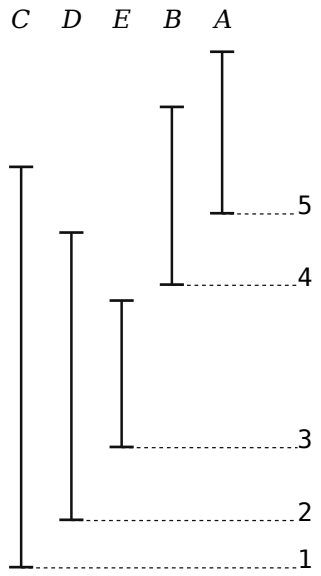


Figura 5 – Os *live intervals* da Figura 4 com os pontos de final enumerados em ordem decrescente

Fonte: adaptado de Poletto e Sarkar [11].

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 [11] escolhe o intervalo que termina por último. Essa heurística muito simples permite uma rápida decisão de *spill*[10] 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* [11]. Elas residem na memória principal durante todo o *live interval*, devendo ser carregadas para todos os usos e armazenadas novamente a cada redefinição.

A seguir estão os passos que ocorreriam no exemplo da Figura 4 com somente dois registradores disponíveis. A lista de intervalos ativos após cada passo é apresentada entre ⟨ e ⟩. Cada passo irá saltar para o início do próximo *live interval*, como está indicado pelos números `1` a `5` associados do lado direito.

1. $A$ é adicionado à lista de intervalos ativos: $\langle A \rangle$ (Figura 6);

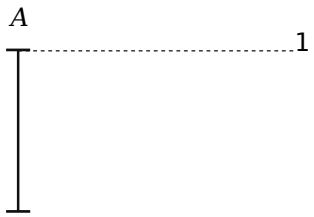


Figura 6 – *Live interval* $A$ ativo na lista

Fonte: adaptado de Poletto e Sarkar [11].

2. $B$ é adicionado à lista: $\langle A, B \rangle$ (Figura 7);

[10] 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.

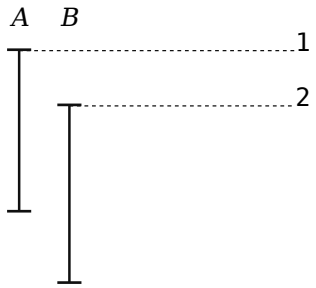


Figura 7 – *Live intervals* $A$ e $B$ ativos na lista

Fonte: adaptado de Poletto e Sarkar [11].

3. Uma decisão de *spill* é necessária.

   $C$ termina por último, então sofre *spill* e não entra na lista: $\langle A, B \rangle$ (Figura 8);

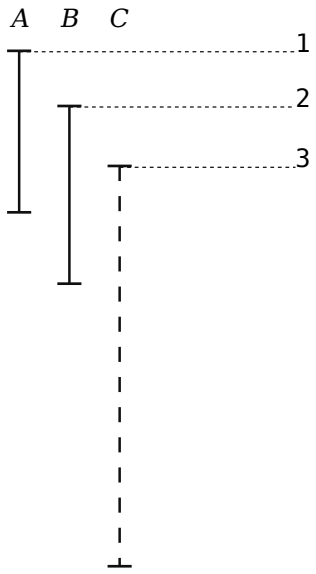


Figura 8 – *Live intervals* $A$ e $B$ ativos na lista, $C$ sofreu *spill*

Fonte: adaptado de Poletto e Sarkar [11].

4. $A$ é removido da lista e $D$ é adicionado: $\langle B, D \rangle$ (Figura 9);

5. $B$ é removido da lista e $E$ é adicionado: $\langle E, D \rangle$ (Figura 10).

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

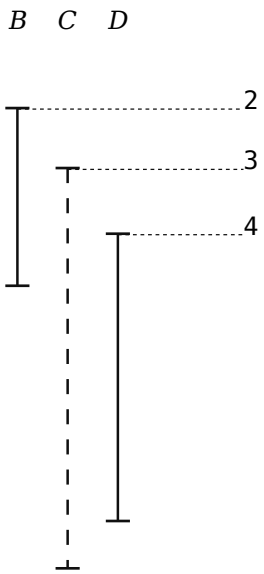


Figura 9 – *Live intervals* $B$ e $D$ ativos na lista, $C$ sofreu *spill*

Fonte: adaptado de Poletto e Sarkar [11].

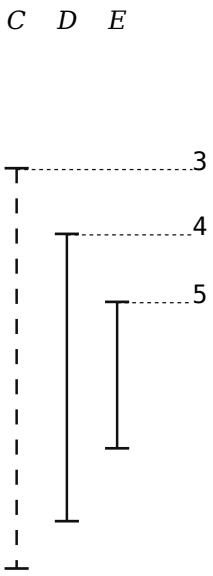


Figura 10 – *Live intervals* $D$ e $E$ ativos na lista, $C$ sofreu *spill*

Fonte: adaptado de Poletto e Sarkar [11].

*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 são precedidas por 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 6);
2. $B$ é adicionado à lista: $\langle A, B \rangle$ (Figura 7);
3. $C$ é adicionado à lista: $\langle A, B, C \rangle$ (Figura 11);

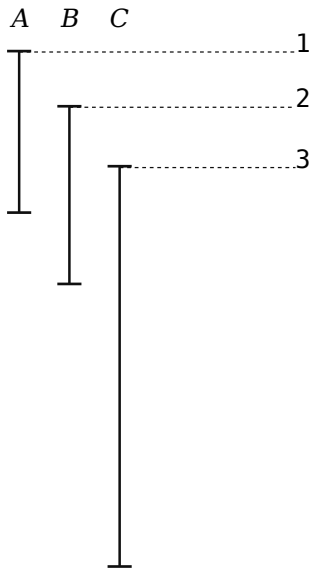


Figura 11 – *Live intervals* $A$, $B$ e $C$ ativos na lista

Fonte: adaptado de Poletto e Sarkar [11].

4. $A$ é removido da lista e $D$ é adicionado: $\langle B, D, C \rangle$ (Figura 12);
5. $B$ é removido da lista e $E$ é adicionado: $\langle E, D, C \rangle$ (Figura 13).

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 $C$.

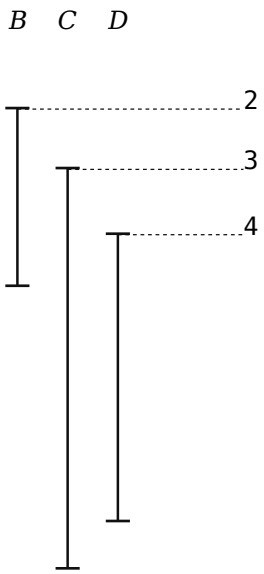


Figura 12 – *Live intervals* $B$, $C$ e $D$ ativos na lista

Fonte: adaptado de Poletto e Sarkar [11].

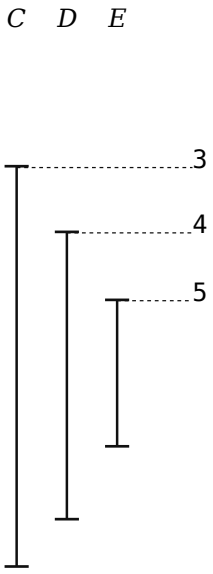


Figura 13 – *Live intervals* $C$, $D$ e $E$ ativos na lista

Fonte: adaptado de Poletto e Sarkar [11].

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 3 serão abordadas técnicas para reduzir a quantidade de acessos necessários.

### 2.2.1 Exemplos de alocação por *linear scan*

Os códigos curtos a seguir serão utilizados em exemplos para ilustrar a aplicação do algoritmo *linear scan*. O Algoritmo 2 tem 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.

**Algoritmo 2:** Exemplo de pseudocódigo

```
                                          { 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 }
```

#### Ordenação

Há várias possibilidades para a escolha da ordem para linearizar o código. No caso do Algoritmo 2, a busca em profundidade[11] resulta no Algoritmo 3.

#### *Live intervals*

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

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

[11] Visitando primeiro as ramificações dos casos negativos das estruturas condicionais.

representa os menores intervalos válidos para as variáveis de `a` a `e` (com as instruções na ordem do Algoritmo 2):

- `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 }`.

**Algoritmo 3:** Algoritmo 2 com as instruções ordenadas por busca em profundidade (explorando antes as ações para condições falsas)

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

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 }`.

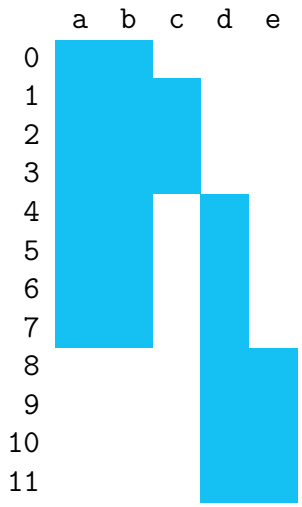


Figura 14 – *Live intervals* do primeiro algoritmo de exemplo

#### 2.2.1.1 Variação do primeiro algoritmo aumentando o número de interferências

No Algoritmo 4 foi alterada a última condição do Algoritmo 2 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 15.

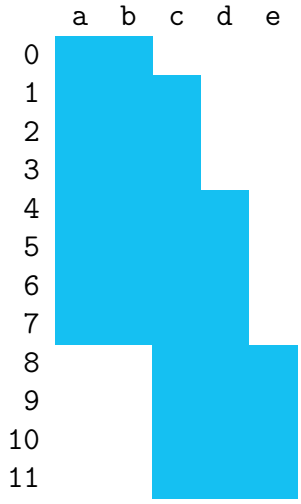


Figura 15 – *Live intervals* do segundo algoritmo de exemplo

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

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 }`;

**Algoritmo 4:** Alteração do Algoritmo 2 com a variável `c` viva até o final

```
                                                { 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 }
```

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 }`.

Ao sofrer *spill*, um *live interval* é dividido em vários intervalos menores, conectados entre si por acessos à memória. Eles deverão ser alocados (podendo ser utilizados registradores diferentes para cada um) e podem interferir com os outros intervalos. A depender da estratégia de *spilling*, esses intervalos variam de tamanho (desde somente 2 instruções até partes consideráveis do código). Assim, a decisão de *spill* e a inserção de *spill code* interfere significativamente na qualidade da alocação.

Nos próximos capítulos serão discutidas algumas estratégias de geração de código de *spill* e de reduzir sua quantidade e impacto na execução.

# 3 TÉCNICAS DE REDUÇÃO 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 [2] é bastante simples: para uma variável que sofreu *spill* é adicionado um `load` antes de todo uso e um `store` depois de toda definição. Os Algoritmos 5 e 6 exemplificam, respectivamente, a aplicação dessa política com 2 registradores disponíveis nos *live ranges* de `b` e de `a`, alterando o código do Algoritmo 2. Pela proposta do algoritmo *linear scan* [11], tanto `a` quando `b` poderiam ser escolhidos para *spill*, já que ambas têm a mesma duração.

**Algoritmo 5:** *Spill everywhere* de `b` no Algoritmo 2

```
                                          { 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 }
```

Como foi apresentado pela Figura 3, o alocador de Chaitin [2] 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* [2]. 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) [20]. 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 3.2.

$$\text{custo}(v) = \sum_{i \in \text{instruções em que } v \text{ está viva}} 10^{d_i} c_i \tag{3.1}$$

$$h_0(v) = \text{custo}(v)/\text{grau}(v) \tag{3.2}$$

Essa divisão dá valor menor a vértices que reduziriam mais o grau de outros vértices se fossem removidos, porém não desconsidera o custo dessa remoção (*spill*).

Palanciuc e Badea [25] comentam que, diferente da ampla aceitação da coloração de grafos como *O Método* para alocação de registradores, a inserção de *spill code* não tem nenhuma solução consensual. Ainda é um campo dominado por heurísticas e abordagens específicas. Muitas pesquisas foram desenvolvidas a respeito da escolha de *spill* e de como alterar o código do programa, com propostas diversificadas de melhorias à política de *spill everywhere*, como as diferentes heurísticas e a limpeza (abordagem *spill almost everywhere*) de Bernstein et al. [14], a rematerialização de valores cuja recomputação é mais barata que a transferências em memória [26, 20] e as técnicas aprofundadas no Capítulo 3.1.

Bernstein et al. [14] 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 3.3), soma discreta do produto da largura, do peso $10^d$ do nível de iteração $d$ e do custo $c$ de cada instrução do *live range*.

**Algoritmo 6:** *Spill everywhere* de `a` no Algoritmo 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 }
```

$$\text{area}(v) = \sum_{i \in \text{instruções em que } v \text{ está viva}} \text{largura}(i) 10^{d_i} c_i \tag{3.3}$$

Assim, as heurísticas de Bernstein são 3.4, 3.5 e 3.6, sendo variáveis escolhidas para *spill* as melhores das três heurísticas:

$$h_1(v) = \text{custo}(v)/\text{grau}(v)^2 \tag{3.4}$$

$$h_2(v) = \text{custo}(v)/(\text{area}(v)\text{grau}(v)) \tag{3.5}$$

$$h_3(v) = \text{custo}(v)/(\text{area}(v)\text{grau}(v)^2) \tag{3.6}$$

Outra proposta de Bernstein et al. [14] 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* [14].

Rematerialização é uma técnica para evitar *spill* que, quando possível e menos custoso, redefine a variável pelo mesmo valor quando este 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 constantes ou outras variáveis que ainda não mudaram) [26].

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 a seguir.

## 3.1 Técnicas de redução de *spill code* para aplicação no *linear scan*

Devido a sua simplicidade e compatibilidade com *linear scan*, foram selecionadas para este trabalho as políticas 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*. A escolha dessas técnicas se deu por essa complementariedade e pela compatibilidade com o *linear scan*: mesmo que suas propostas considerem alocadores por coloração de grafos, as informações necesssárias para seu funcionamento podem ser obtidas na representação de *live intervals* mais facilmente do que são obtidas de *live ranges*.

### 3.1.1 *Interference region spilling*

A técnica de *interference region spilling* [15] consiste em observar, entre dois *live ranges* que interferem entre si, 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. Estão na Figura 16 os *live ranges* das variáveis A e B, identificada a região de interferência (esquerda), aplicação de *spill everywhere* em B (centro) e *spill* de B somente na região de interferência (direita).

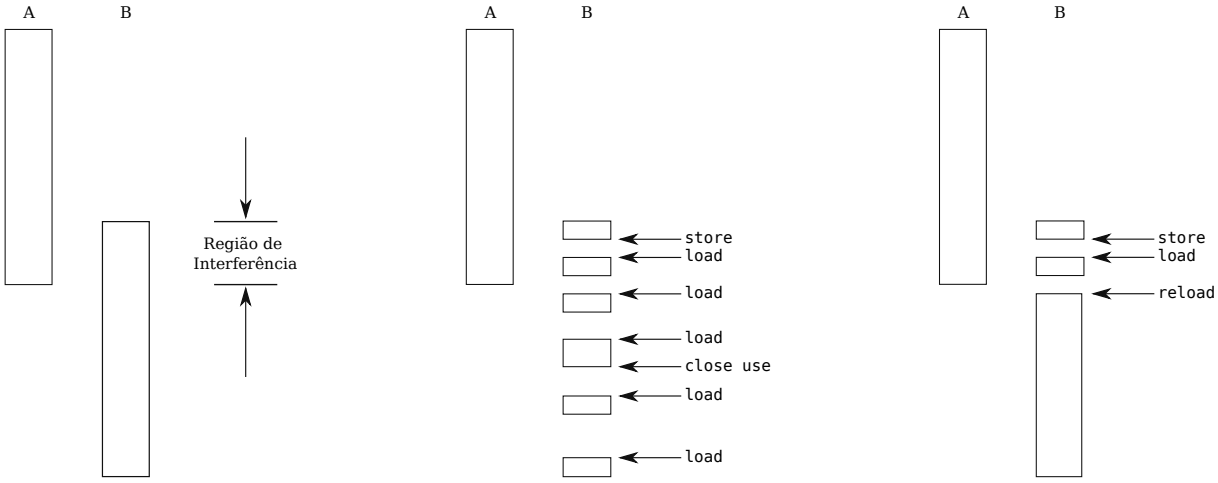


Figura 16 – Diferença entre *spill everywhere* (centro) e *interference region spill* (direita) quando a região de interferência (esquerda) é menor que ambos os *live ranges*

Fonte: Bergner, Dahl, Engebretsen e O'Keefe [15].

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 16, 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 *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`[12] 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.

#### 3.1.1.1 Exemplo de aplicação de *interference region spilling*

A aplicação de *interference region spilling* no primeiro código de exemplo (Algoritmo 2) adiaria que as instruções `store` e `load` somente algumas instruções, já que `c` surge logo após o começo. No Algoritmo 7, `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` no Algoritmo 8.

**Algoritmo 7:** *Interference region spilling* de `b` no Algoritmo 2

```
                                          { 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 }
```

[12] Os autores se referem a elas como `reload`, como na Figura 16.

### 3.1.2 *Live range splitting*

A técnica de *live range splitting* [16] 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[13], a divisão permitirá que diferentes partes sejam alocadas para diferentes registradores disponíveis ou que *live ranges* antes conflitantes possam utilizar o 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 16, *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

[13] Se não inserir mais acessos à memória que outras técnicas ou se os acessos não forem inseridos em posições que são executadas mais vezes.

**Algoritmo 8:** *Interference region spilling* de `a` no Algoritmo 2

```
                                          { 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 }
```

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 17 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 17). 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*.

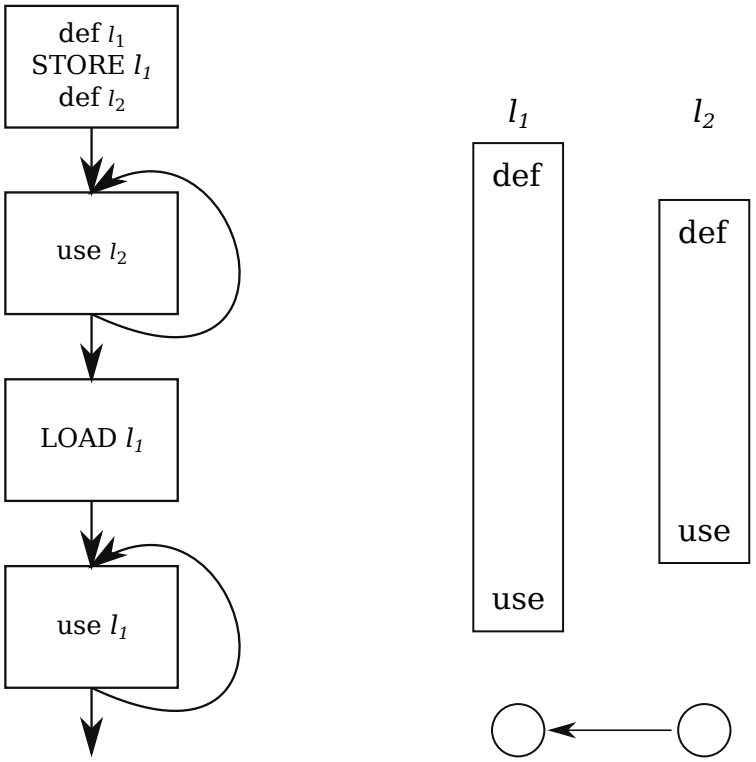


Figura 17 – *Splitting* de $l_1$ ao redor de $l_2$ e seus respectivos *live ranges*

Fonte: adaptado de Cooper e Simpson [16].

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*.

#### 3.1.2.1 Exemplo de aplicação de *live range splitting*

Considerando o Algoritmo 2, o grafo de contenção da Figura 18 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).

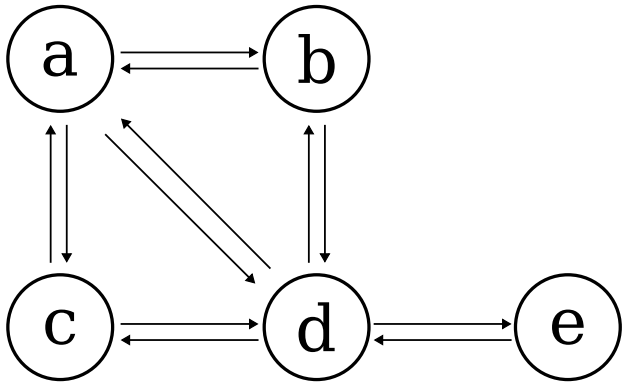


Figura 18 – Grafo de contenção para o Algoritmo 2

O Algoritmo 9 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.

**Algoritmo 9:** Exemplo para *live range splitting*

```
                                          { 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 }
```

Pode-se observar que, enquanto `b` está vivo, não há nenhuma instrução que utilize ou defina `a`. Assim, como ilustrado pela Figura 19, `b` está contido em `a`, mas não o contrário.

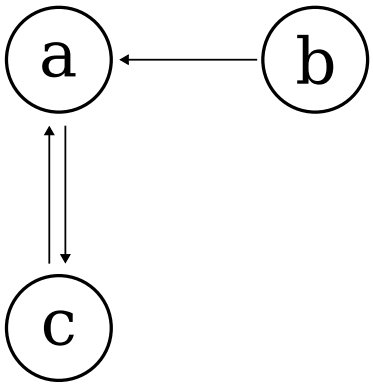


Figura 19 – Grafo de contenção para o exemplo de *live range splitting* (Algoritmo 9)

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 do Algoritmo 10.

**Algoritmo 10:** Algoritmo 9 com *splitting* de `a` ao redor de `b` e `c`

```
                                          { 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 }
```

# 4 EXPERIMENTOS COMPUTACIONAIS

Para a realização de experimentos computacionais foram implementados o algoritmo *linear scan* e as técnicas de *interference region spilling* e *live range splitting*. Embora elas tenham sido propostas para alocadores por coloração de grafos, 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) e a alocação *linear scan* oferece um cenário mais simples para o funcionamento delas. Além das informações obtidas pelo *linear scan* tradicional, são necessárias também análises para construir o grafo de contenção e identificar as regiões de interferência.

O grafo de contenção e a região de interferência podem ser identificados em *live intervals* da mesma maneira que em *live ranges*. Para o grafo de contenção deve-se identificar se há usos ou definições de um intervalo dentro do outro. Para que uma instrução esteja dentro de um intervalo basta compará-la com o início e o fim dele[14]. A região de interferência entre dois intervalos pode ser mais facilmente identificada como o intervalo comum que se inicia no último início e que termina com o primeiro término. Quando intervalos não têm interferência, 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 o início e o fim dele.

Assim é possível rapidamente identificar se há interferência (e qual sua região) ou se há contenção entre dois intervalos. Essa possibilidade de integração de *interference region spilling* e *live range splitting* ao *linear scan* motivou o presente trabalho.

Para a leitura de programas em C e C++ foi utilizado o *front end* do compilador LLVM Clang [27]. O projeto de código aberto LLVM [28] tem muitas ferramentas para construção de compiladores, desenvolvidas em bibliotecas bem estruturadas: as fases de compilação presentes na Figura 1 são todas *passes*, incluindo outras *passes* mais abstratas e muitos modelos para análises ou transformações específicas. As ferramentas do LLVM são disponibilizadas em binários separados, o que possibilita executá-las separadamente e escolher quais *passes* serão utilizadas.

Os códigos compilados pelo Clang foram convertidos na LLVM IR, o formato de representação intermediária do projeto. Essa representação intermediária foi transformada pelo opt [29] para retirar instruções `load` e `store` inseridas nos *live ranges* das variáveis antes do alocador[15]. Os arquivos de LLVM IR transformados foram então compilados de

[14] A representação do simplificada de *live range* como *live interval* com instruções ordenadas permite que essa comparação simples seja suficiente. Instruções dentro de um intervalo são posteriores ao início e anteriores ao fim, ou são a instrução inicial ou final.

[15] Ao gerar LLVM IR, o Clang insere operações de acesso à memória para armazenar e carregar valores

duas maneiras: foi diretamente invocado o llc [30] para geração de código de máquina, selecionando o alocador e o nível de otimização do teste; e, além disso, foi executado o llc para emissão da representação intermediária de linguagem de máquina (LLVM MIR) [31] antes da alocação começar. Os arquivos de LLVM MIR foram utilizados como entrada para a implementação do *linear scan*. A Figura 20 apresenta este processo.

## 4.1 Implementação

A primeira proposta do trabalho buscava implementação do alocador *linear scan* convencional e aprimorado dentro do código do LLVM, compilando-o como mais um dos alocadores disponíveis no llc. Contudo, não foi possível registrar corretamente no *pass manager* legado[16] a implementação de *MachineFunctionPass* e *RegAllocBase*[17].

Foi verificado se seria possível parar o llc antes da *pass* do alocador, ler e editar a LLVM MIR para alocar seus registradores virtuais e então finalizar a geração de código novamente no llc, fornecendo a MIR editada. Porém a alocação na MIR foi ignorada pelo llc, provavelmente refazendo as *passes* iniciais de *CodeGen* com base na cópia da IR que é necessária para referenciar inclusões de bibliotecas, globais ou constantes na MIR.

A presente versão do trabalho utiliza, assim, da MIR criada pelo llc antes da *pass* do alocador e realiza a alocação de registradores e emissão final do código de máquina.

A implementação foi feita em Python para a arquitetura alvo MIPS. Ela aceita execução do *linear scan* com somente a política de *spill everywhere* ou também podendo optar entre ela e *interference region spilling* e *live range splitting*, se gerarem menos *spill*.

A ordenação dos blocos básicos foi realizada por busca em profundidade, como recomendado pelos autores [11].

## 4.2 Resultados experimentais

Os *benchmarks* de C e C++ do SPEC CPU® 2017 [36] foram utilizados para os testes. Foi gerada MIR parando o llc antes dos alocadores nos níveis de otimização -O0 e -O2. Também foram comparados os resultados dos alocadores *fast*, *basic* e *greedy* do LLVM nos níveis de otimização -O2 e -O3 (e *fast* também foi executado com -O0).

de variáveis. Esse "*spill* prematuro" reduz a quantidade de *spill* criado pelos alocadores (próprios do LLVM ou desenvolvidos no trabalho), impedindo a sua adequada avaliação.

[16] Há alguns anos partes das bibliotecas do LLVM passaram a utilizar um novo *pass manager*. O opt, por exemplo, já usa esse novo formato, que permite inserir *passes* como bibliotecas dinâmicas separadas do código LLVM [32, 33]. O *pass manager* legado, porém, exige que as *passes* e todas as suas dependências sejam registradas nos cabeçalhos do próprio LLVM [34].

[17] A classe *RegAllocBase* não é claramente documentada (e duas implementações dela, o *RAFast* e *RABasic* sequer constam no *doxygen* [35]) e o registro de uma implementação sua exige alteração (de acordo com padrões de projeto e nomeação muito específicos) em várias classes e cabeçalhos do *back end* do LLVM.

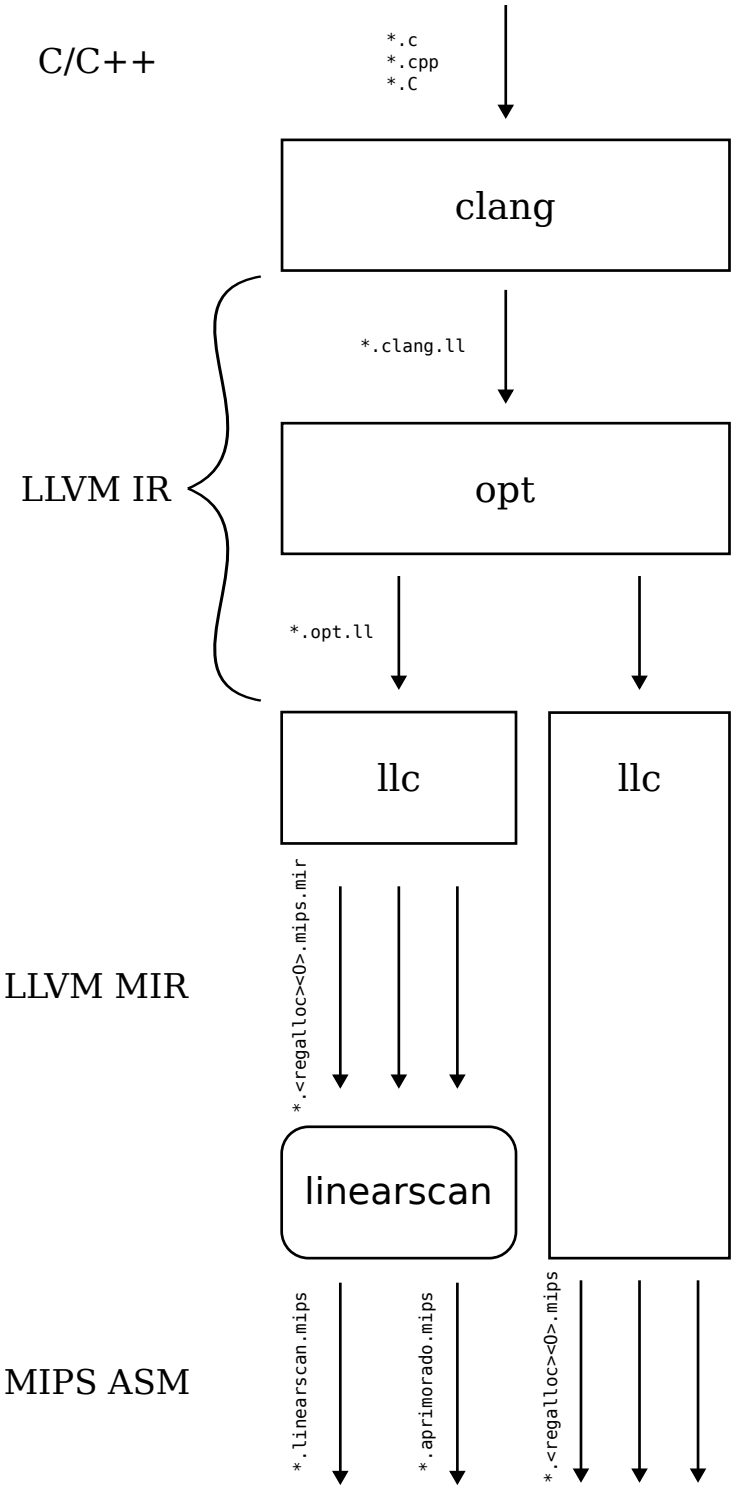


Figura 20 – Organização dos testes computacionais

Para a maioria dos *benchmarks*, os *live intervals* dos registradores virtuais eram pequenos e com poucas interferências, o que não exigiu nenhum *spill* do alocador. As Tabelas 1 e 2 exemplificam os resultados para os arquivos `specrand.c` e `stdio.c` (do *benchmark* `500.perlbench_r` compilado pelos alocadores *basic*, *fast* e *greedy* nos níveis de otimização 0, 2 e 3 (identificados na segunda coluna pelo nome do alocador e o seu nível de otimização). As linhas em que o alocador *linearscan* está presente indicam que o llc foi parado antes do alocador à direita, emitindo a MIR. Foram contadas as operações de `store` e `load` inseridas no código. Também foram contados os comentários de *spill* e *reload* adicionados a elas, quando o alocador considerava a instrução oriunda de *spilling*.

Tabela 1 – *Benchmark* `500.perlbench_r` `specrand.c`

| **Alocadores** | | **Store** | **Load** | **Spill** | **Reload** |
|---|---|---|---|---|---|
| | basic2 | 51 | 72 | 27 | 27 |
| | basic3 | 51 | 73 | 27 | 27 |
| | fast0 | 89 | 107 | 62 | 57 |
| | fast2 | 84 | 105 | 52 | 60 |
| | fast3 | 84 | 109 | 53 | 63 |
| | greedy2 | 51 | 72 | 27 | 27 |
| | greedy3 | 51 | 73 | 27 | 27 |
| linearscan | basic2 | 73 | 72 | 69 | 60 |
| linearscan | fast0 | 118 | 141 | 91 | 91 |
| linearscan | fast2 | 73 | 72 | 69 | 60 |
| linearscan | fast3 | 73 | 72 | 69 | 60 |

Tabela 2 – *Benchmark* `500.perlbench_r` `stdio.c`

| **Alocadores** | | **Store** | **Load** | **Spill** | **Reload** |
|---|---|---|---|---|---|
| | basic2 | 6 | 6 | 6 | 6 |
| | basic3 | 6 | 6 | 6 | 6 |
| | fast0 | 6 | 6 | 6 | 6 |
| | fast2 | 6 | 6 | 6 | 6 |
| | fast3 | 6 | 6 | 6 | 6 |
| | greedy2 | 6 | 6 | 6 | 6 |
| | greedy3 | 6 | 6 | 6 | 6 |
| linearscan | basic2 | 12 | 12 | 12 | 12 |
| linearscan | fast0 | 12 | 12 | 12 | 12 |
| linearscan | fast2 | 12 | 12 | 12 | 12 |
| linearscan | fast3 | 12 | 12 | 12 | 12 |

A alocação do *linear scan* para o arquivo `specrand.c` (Tabela 1) teve melhores resultados com o código otimizado em níveis -O2 e -O3, inclusive gerando a mesma quantidade de instruções `load` que os alocadores *basic* e *greedy*.

O arquivo `studio.c` (Tabela 2) é pequeno demais para diferenciar os níveis de otimização e os alocadores do LLVM entre si.

Os resultados do *linearscan* com a MIR gerada antes da execução do *fast* e do *basic* com -O2[18] são os mesmos, então para os outros *benchmarks* somente foi parado o alocador *fast*. As Figuras 21 a 30 exibem a quantidade de instruções de acesso à memória gerados por cada alocador (para os *benchmarks* em que não houve necessidade de *spill*).

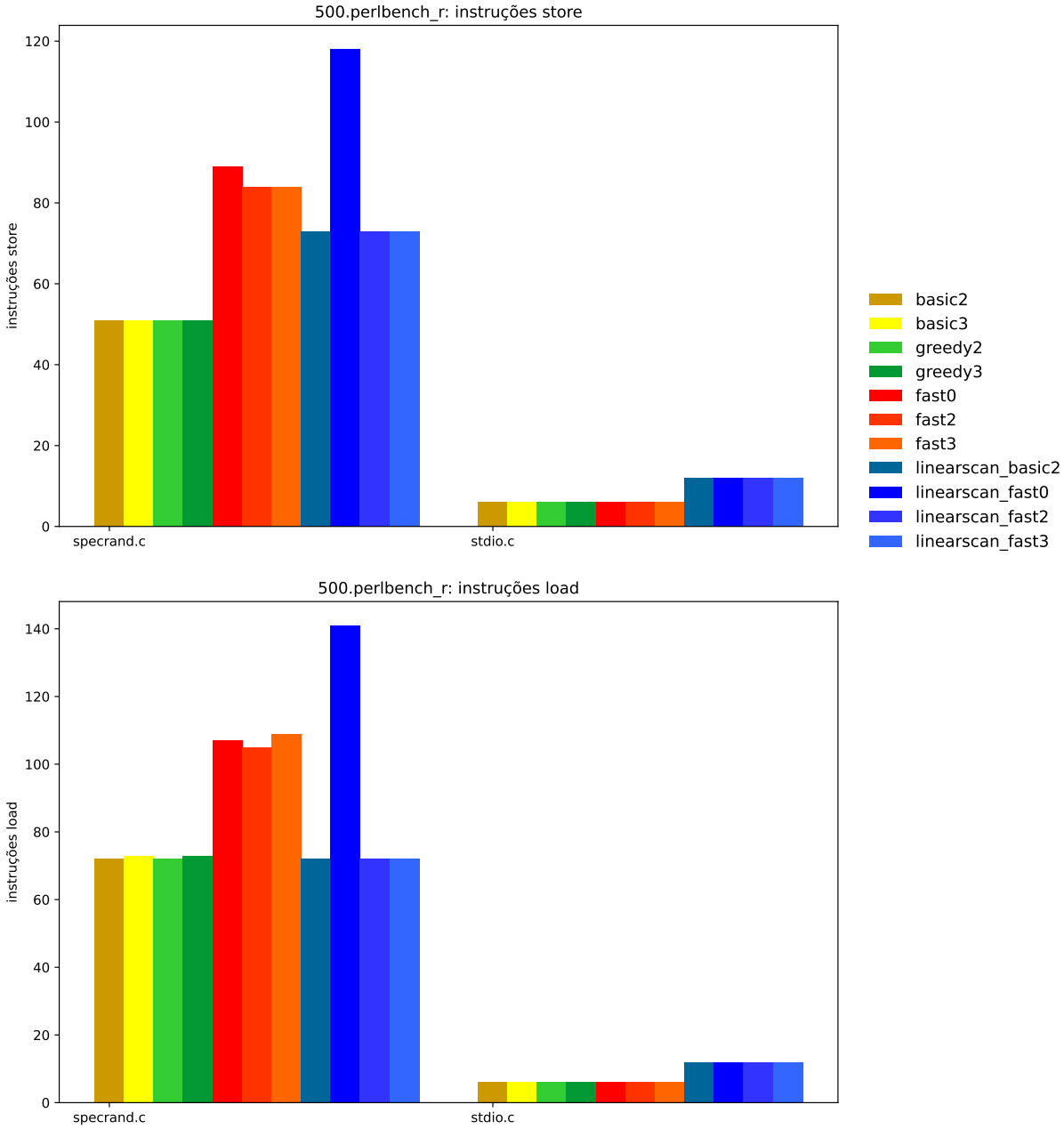


Figura 21 – *Benchmark* `500.perlbench_r`

Houve poucos *benchmarks* que exigiram *spill* do *linear scan*. Estes estão nas Tabelas 3 a 5, incluindo o alocador *aprimorado* (*linear scan* com as técnicas de minimização

[18] Linhas com alocador *linearscan fast2* e *basic2*

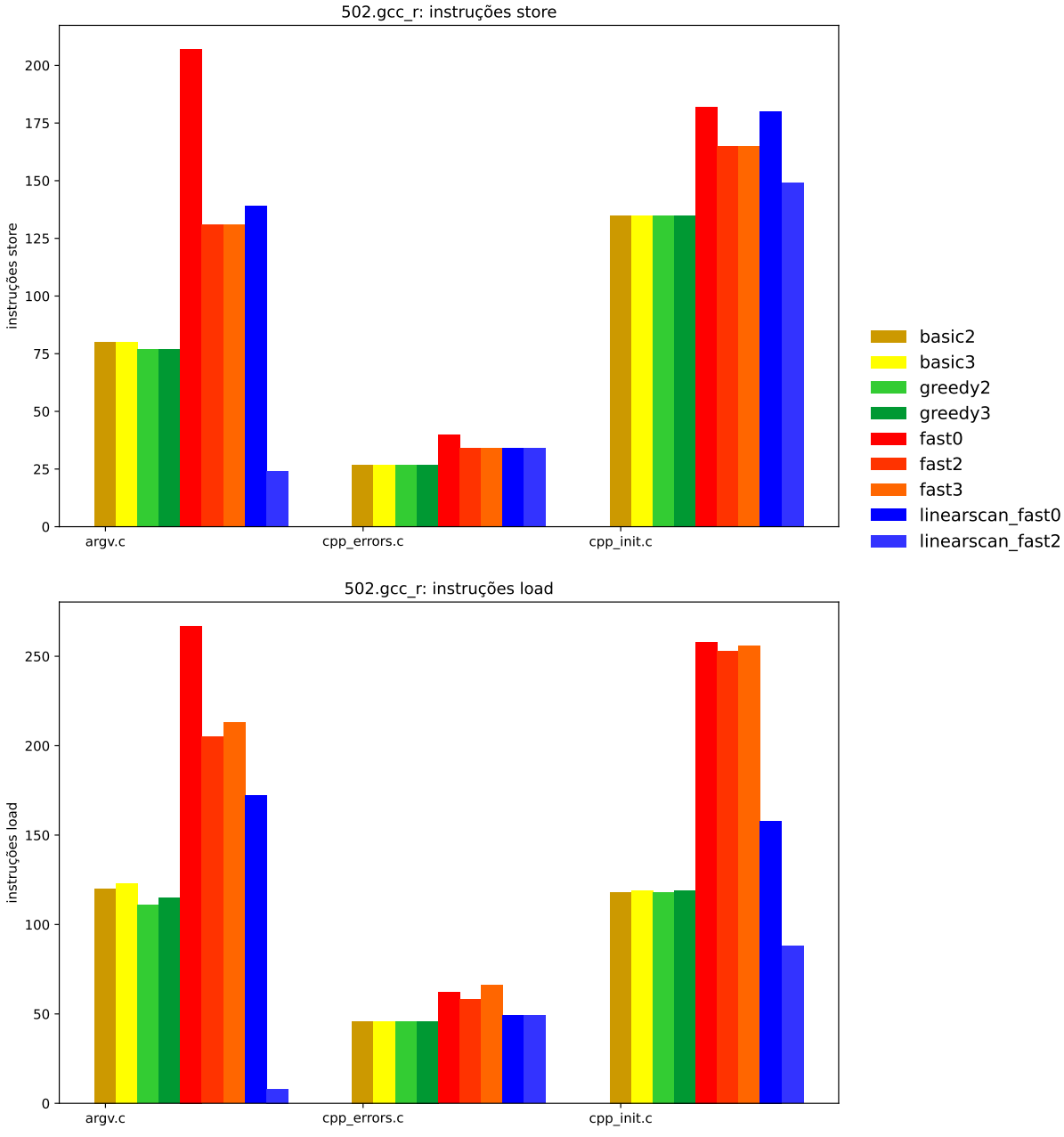


Figura 22 – *Benchmark* `502.gcc_r`

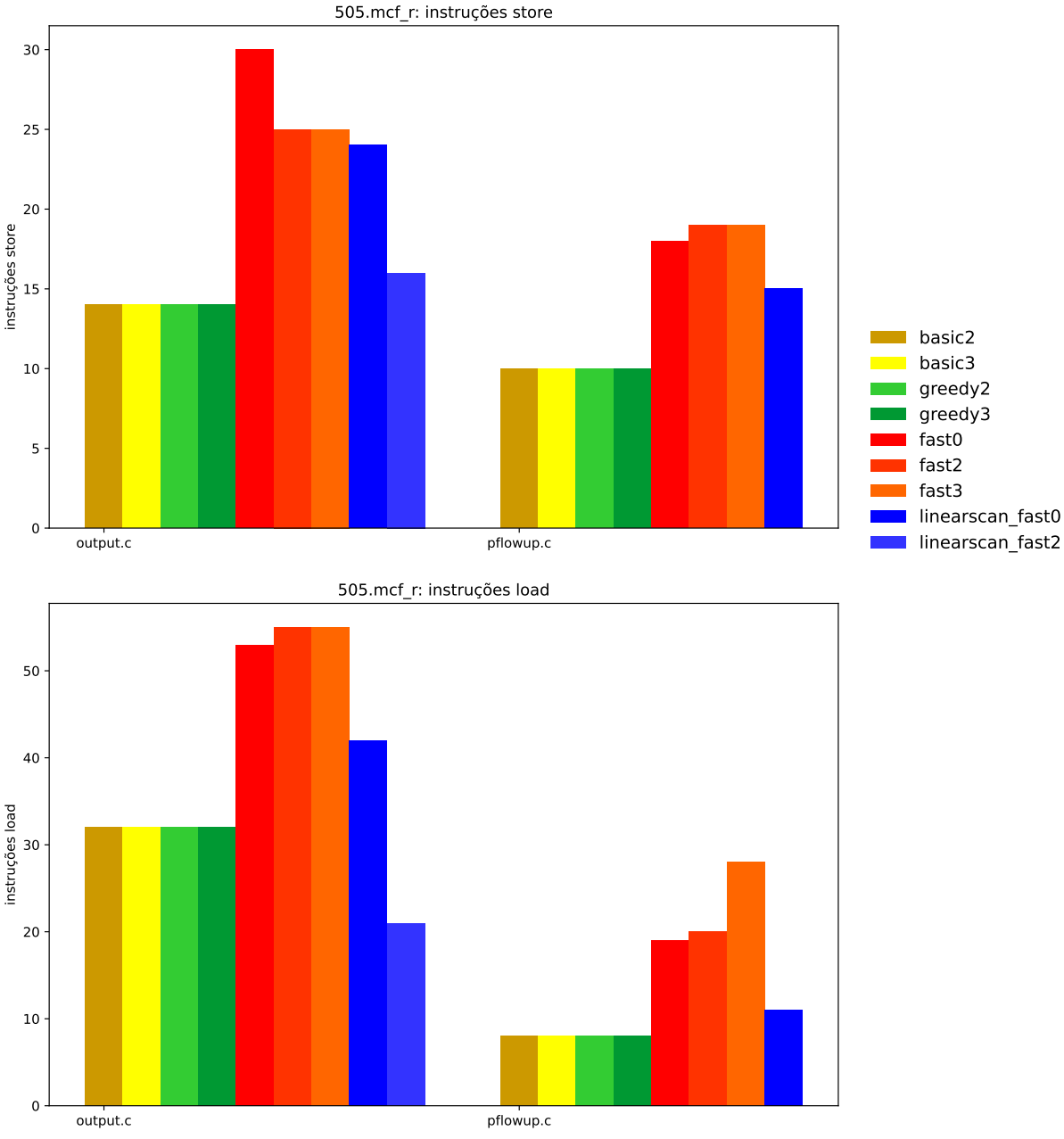


Figura 23 – *Benchmark* `505.mcf_r`

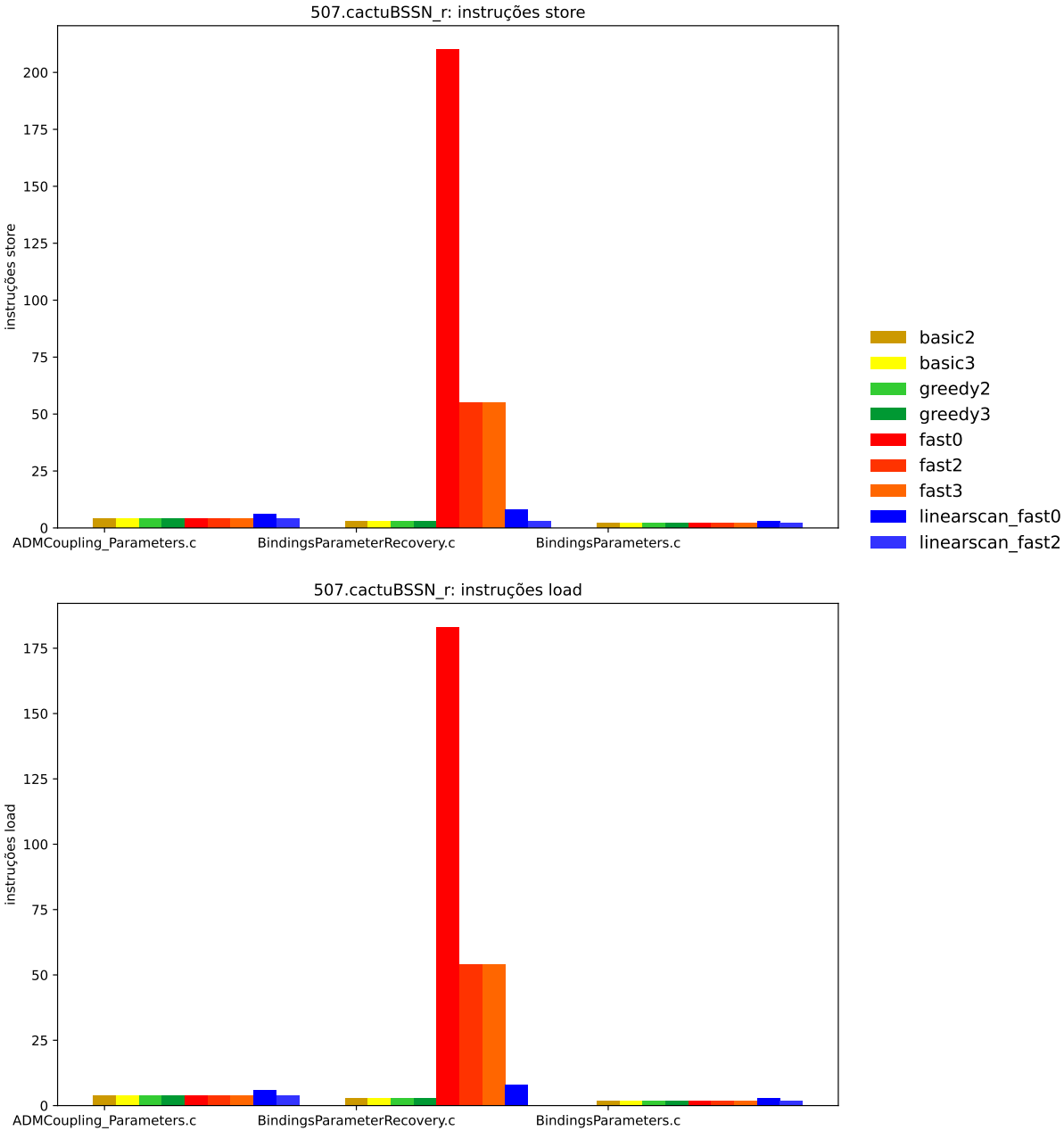


Figura 24 – *Benchmark* `507.cactuBSSN_r`

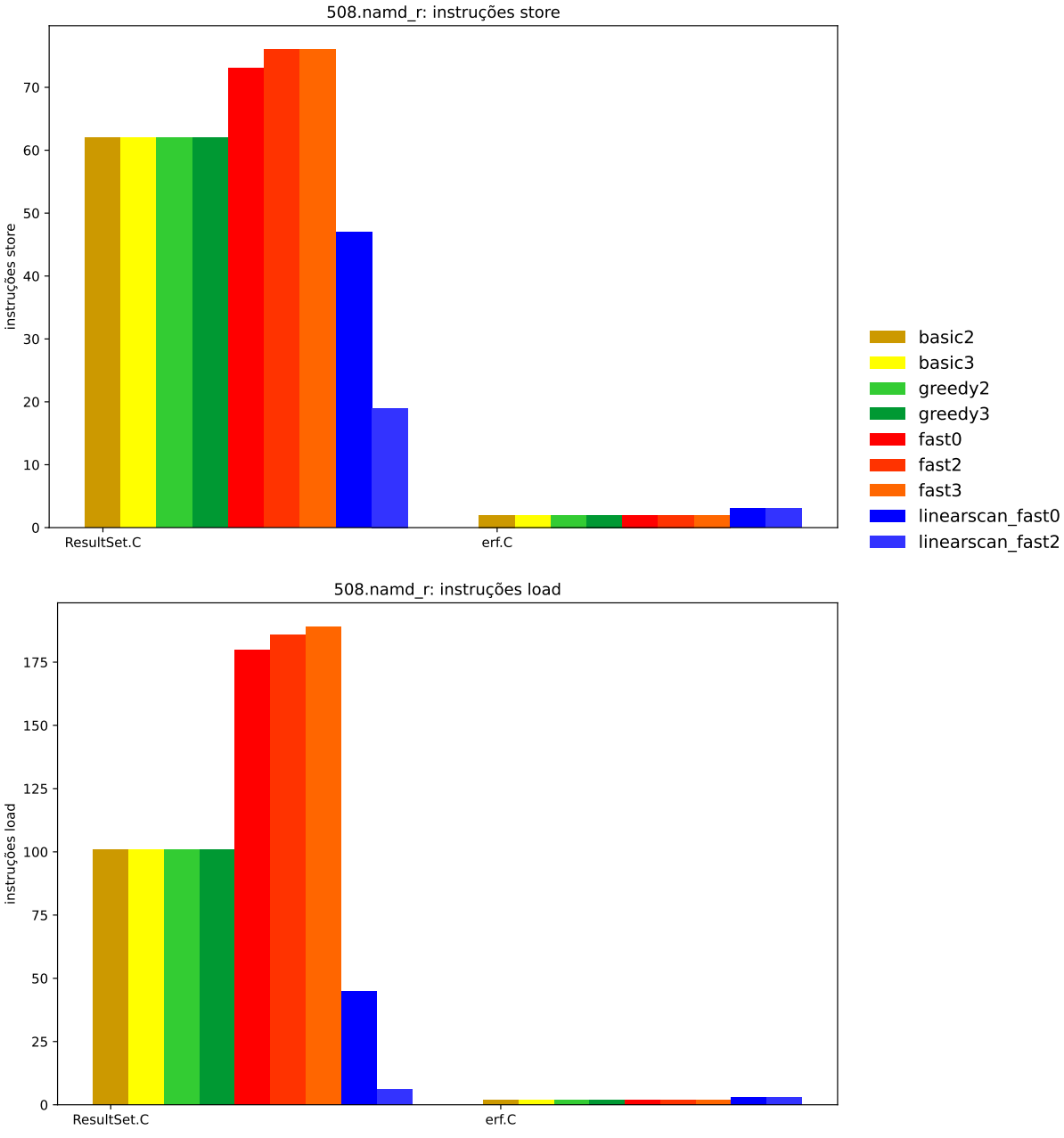


Figura 25 – *Benchmark* `508.namd_r`

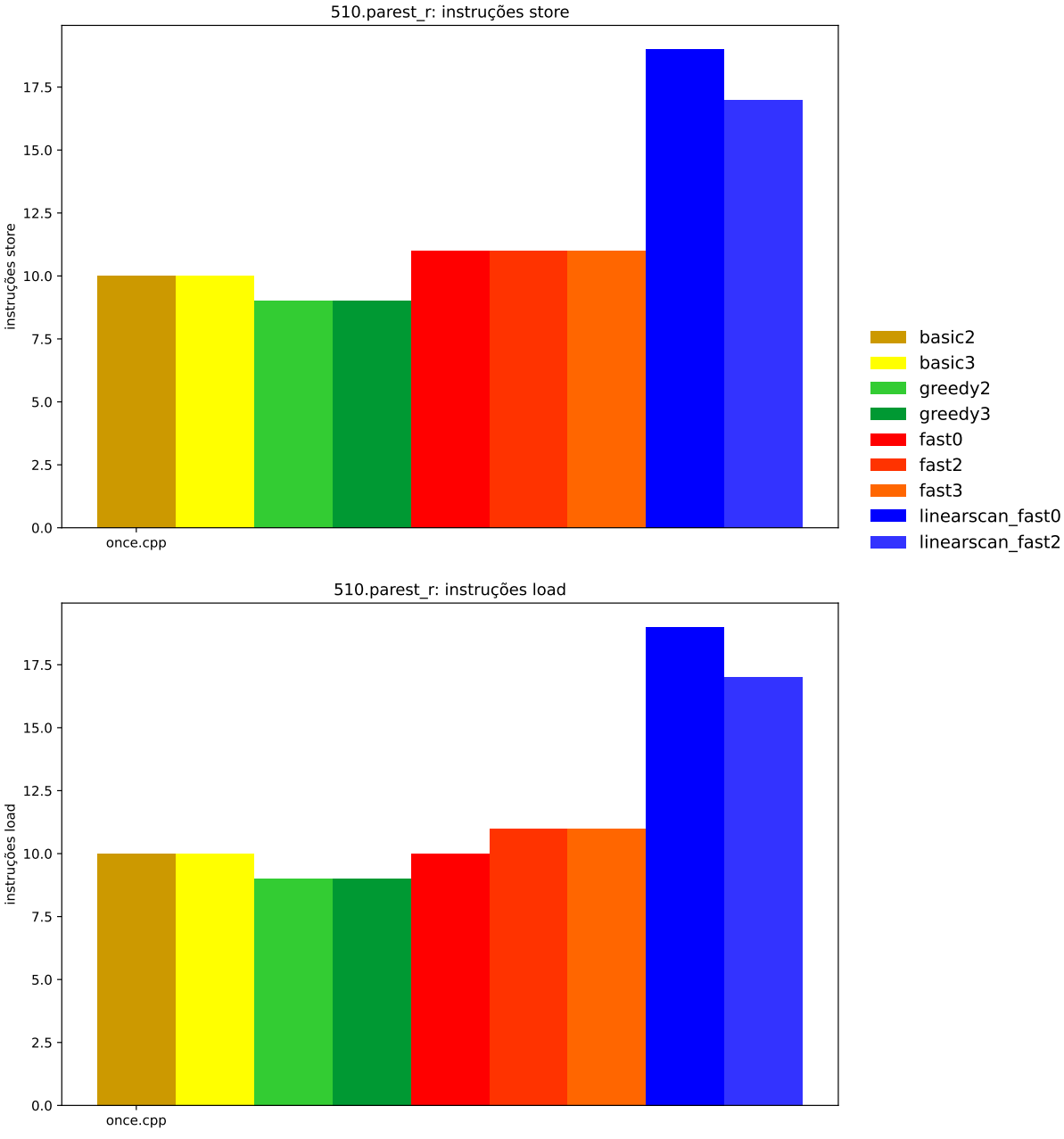


Figura 26 – *Benchmark* `510.parest_r`

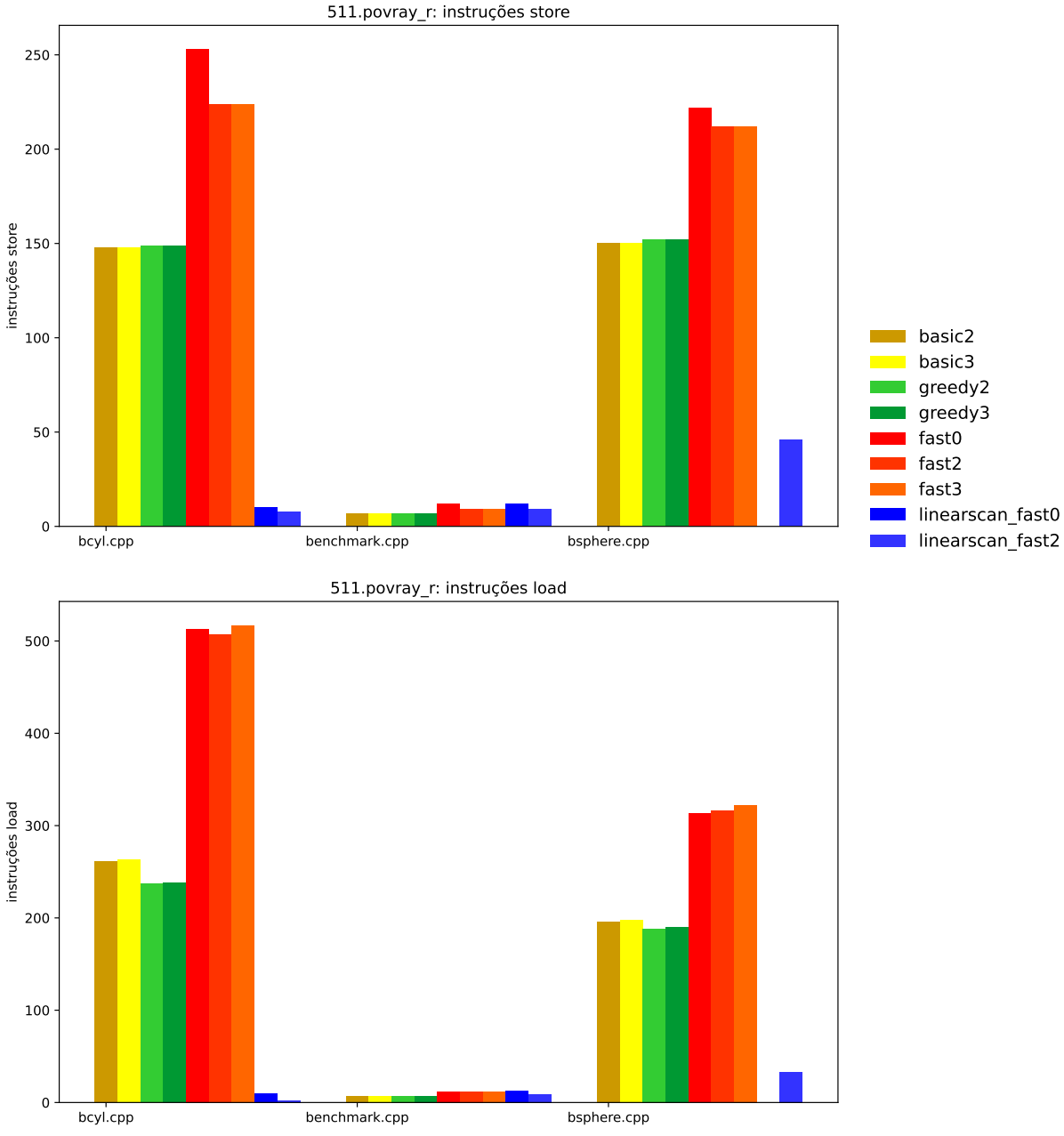


Figura 27 – *Benchmark* `511.povray_r`

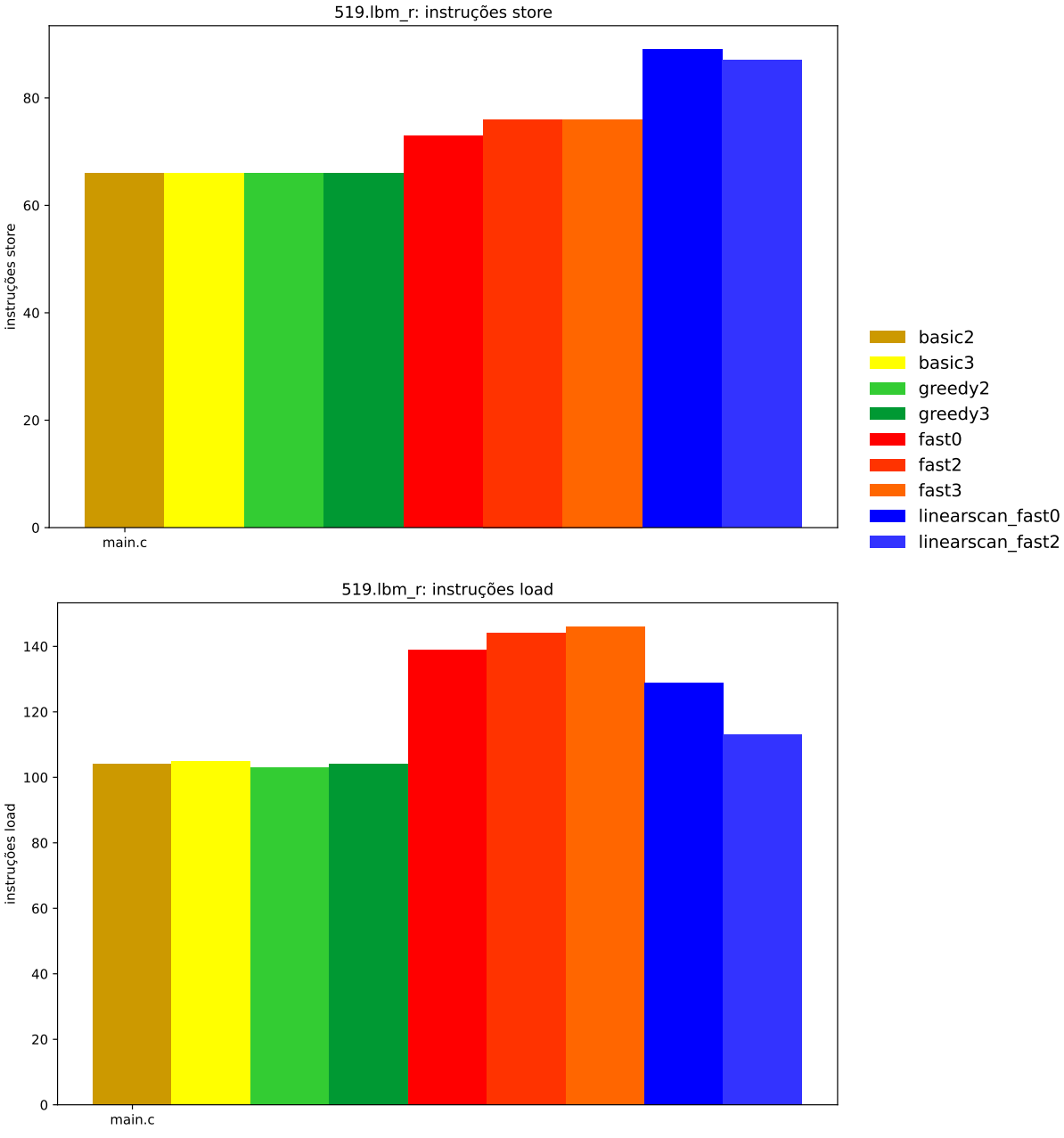


Figura 28 – *Benchmark* `519.lbm_r`

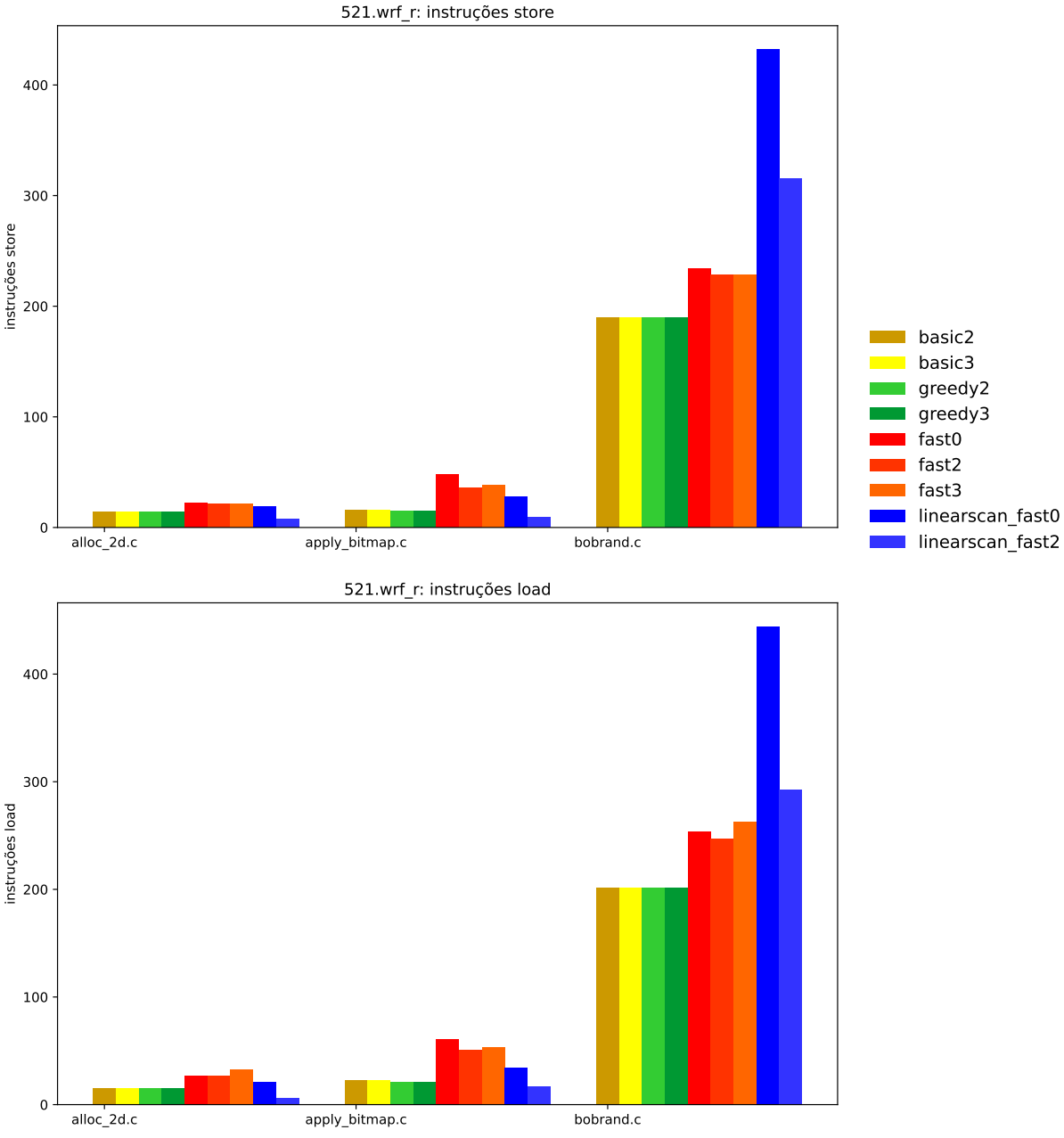


Figura 29 – *Benchmark* `521.wrf_r`

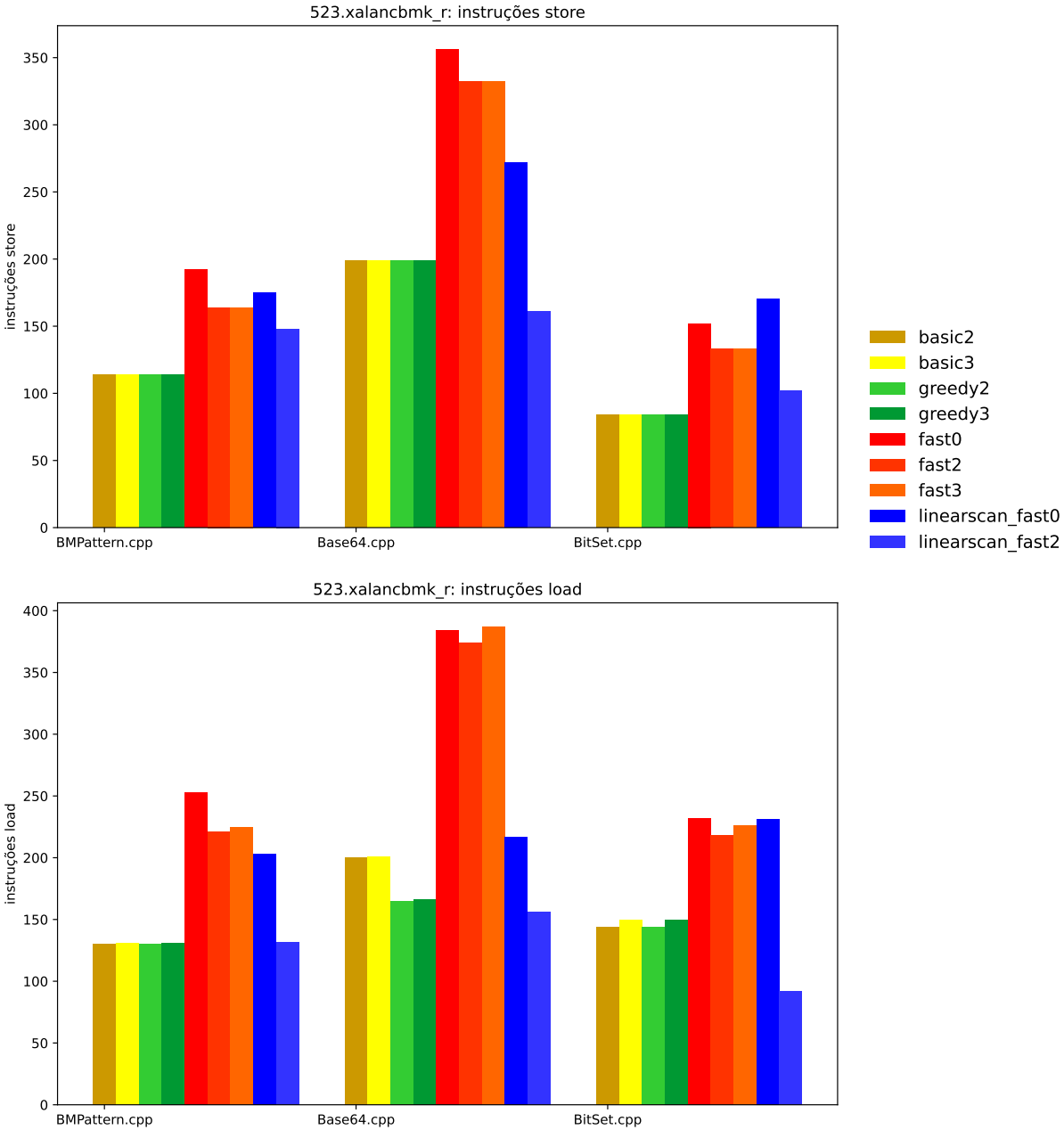


Figura 30 – *Benchmark* `523.xalancbmk_r`

implementadas). A coluna **ew** indica a quantidade de *spill* e *reload* gerada pela política de *spill everywhere* e **LRs** a quantidade gerada por *live range splitting*. Foi omitida a coluna de *interference region spilling* porque o algoritmo, ao escolher a menor geração de código de *spill* dentre as políticas, privilegiou sempre *live range splitting*.

Tabela 3 – *Benchmark* `502.gcc_r` `argv.c`

| **Alocadores** | | **Store** | **Load** | **Spill** | **ew** | **LRs** | **Reload** | **ew** | **LRs** |
|---|---|---|---|---|---|---|---|---|---|
| | basic2 | 80 | 120 | 54 | | | 69 | | |
| | basic3 | 80 | 123 | 54 | | | 69 | | |
| | greedy2 | 77 | 111 | 51 | | | 60 | | |
| | greedy3 | 77 | 115 | 51 | | | 61 | | |
| | fast0 | 207 | 267 | 184 | | | 215 | | |
| | fast2 | 131 | 205 | 78 | | | 158 | | |
| | fast3 | 131 | 213 | 84 | | | 162 | | |
| linearscan | fast0 | 139 | 172 | 116 | 39 | | 120 | 43 | |
| linearscan | fast2 | 24 | 8 | 23 | | | 7 | | |
| aprimorado | fast0 | 118 | 150 | 95 | | 2 | 98 | | 5 |

Tabela 4 – *Benchmark* `502.gcc_r` `graphds.c`

| **Alocadores** | | **Store** | **Load** | **Spill** | **ew** | **LRs** | **Reload** | **ew** | **LRs** |
|---|---|---|---|---|---|---|---|---|---|
| | basic2 | 192 | 290 | 113 | | | 132 | | |
| | basic3 | 192 | 298 | 111 | | | 134 | | |
| | greedy2 | 191 | 282 | 113 | | | 124 | | |
| | greedy3 | 191 | 290 | 113 | | | 126 | | |
| | fast0 | 382 | 577 | 304 | | | 411 | | |
| | fast2 | 318 | 534 | 192 | | | 377 | | |
| | fast3 | 318 | 543 | 187 | | | 385 | | |
| linearscan | fast0 | 331 | 412 | 253 | 32 | | 246 | 25 | |
| linearscan | fast2 | 152 | 118 | 126 | | | 77 | | |
| aprimorado | fast0 | 312 | 404 | 234 | | 3 | 238 | | 7 |

É visível o impacto do nível de otimização aplicado no alocador *fast* e no *linear scan* implementado. Para os *benchmarks* em que foi necessário *spill*, o algoritmo *linear scan* utilizando a MIR emitida em -O0 gerou código em quantidade comparável ao alocador *fast* nos níveis -O2 e -O3.

Para os arquivos `argv.c` (Tabela 3) e `graphds.c` (Tabela 4) do *benchmark* `502.gcc_r`, a aplicação de *live range splitting* no lugar de *spill everywhere* reduziu tanto as instruções `store` quanto `load` (em nível de otimização -O0, pois em nível -O2 não foi necessário *spilling*). Para o arquivo `task_for_point.c` do *benchmark* `521.wrf_r` (Tabela 5), porém, a aplicação não foi benéfica, adicionando 3 instruções `load` se comparada ao *spill everywhere*.

Tabela 5 – *Benchmark* `521.wrf_r` `task_for_point.c`

| **Alocadores** | | **Store** | **Load** | **Spill** | **ew** | **LRs** | **Reload** | **ew** | **LRs** |
|---|---|---|---|---|---|---|---|---|---|
| | basic2 | 31 | 86 | 23 | | | 58 | | |
| | basic3 | 31 | 88 | 23 | | | 60 | | |
| | greedy2 | 35 | 56 | 26 | | | 28 | | |
| | greedy3 | 35 | 56 | 28 | | | 28 | | |
| | fast0 | 94 | 161 | 87 | | | 133 | | |
| | fast2 | 67 | 141 | 48 | | | 113 | | |
| | fast3 | 67 | 145 | 47 | | | 117 | | |
| linearscan | fast0 | 40 | 61 | 33 | 6 | | 33 | 6 | |
| linearscan | fast2 | 13 | 20 | 12 | | | 5 | | |
| aprimorado | fast0 | 40 | 64 | 33 | | 3 | 36 | | 6 |

# 5 CONCLUSÃO

Com esta pesquisa foi possível observar a aplicabilidade de *interference region spilling* e *live range splitting* no alocador *linear scan*. Os testes da implementação, contudo, não apresentaram dados suficientes para avaliação consistente.

É necessário investigar mais aprofundadamente a configuração do *front end*, do *middle end* e da geração de código anterior à alocação de registradores para que os *benchmarks* utilizados gerem *live ranges* maiores (com menos instruções de acesso à memória inseridas implicitamente) e possam exigir mais *spilling* e, assim, oportunizar maior produção de dados para comparação. Deve-se aprofundar a investigação do *renumbering* e de outras transformações aplicadas aos *live ranges* no Clang e no llc para buscar oportunidades e necessidades de geração de *spill* e obter melhores dados comparativos dos diferentes alocadores. Pode ser viável (e talvez necessária) a inclusão de *benchmarks* de outras fontes, como dos *test suites* do LLVM e do GCC, para agregar à coleta de dados.

A predominância do *live range splitting* dentre as técnicas convida mais estudo. É possível que a simplificação de *live range* em *live interval*, com a ordenação das instruções por busca em profundidade, exija muito *spill* e *reload* de fora da região de interferência e, assim, faça *interference region spilling* se diferenciar pouco de *spill everywhere*. É possível também que o *renumbering* realizado pelo LLVM tenha reduzido o tamanho e a quantidade de interferências dos *live intervals* de forma a evitar casos de contenção mútua e, assim, oferecendo vantagens para *live range splitting*.

A aplicação de passagens de limpeza posterior ao *spill everywhere* e *interference region spill* pode trazer benefícios, já que da forma em que são inseridos pode haver definições seguidas de usos com somente o *spill* e o *reload* entre elas. Aplicar rematerialização como substituição de *spilling* em casos de constantes e expressões é outra melhoria que pode ser adicionada para reduzir os acessos à memória.

Seria valorosa a implementação no LLVM, aproveitando as ferramentas de análise e a variedade de arquiteturas alvo disponíveis, do *linear scan* convencional e das políticas de geração de código de *spill* abordadas neste trabalho. Com isso seria possível gerar *assembly* para x86 e comparar, também, a velocidade de execução do código produzido.

# REFERÊNCIAS


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

[2] CHAITIN, G. J. Register allocation & spilling via graph coloring. *ACM Sigplan Notices*, ACM, New York, v. 17, n. 6, p. 98–101, 1982. Disponível em: <https://dl.acm.org/doi/10.1145/872726.806984>. Acesso em: 24 abr. 2024.

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

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

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

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

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

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

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

[10] 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.

[11] POLETTO, M.; SARKAR, V. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, New York, v. 21, n. 5, p. 895–913, 1999. Disponível em: <https://dl.acm.org/doi/abs/10.1145/330249.330250>. Acesso em: 24 abr. 2024.

[12] AYCOCK, J. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, ACM, New York, v. 35, n. 2, p. 97–113, 2003. Disponível em: <https://dl.acm.org/doi/10.1145/857076.857077>. Acesso em: 24 abr. 2024.

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

[14] BERNSTEIN, D. et al. Spill code minimization techniques for optimizing compilers. *ACM SIGPLAN Notices*, ACM, New York, v. 24, n. 7, p. 258–263, 1989. Disponível em: <https://dl.acm.org/doi/10.1145/73141.74841>. Acesso em: 24 abr. 2024.

[15] 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. Disponível em: <https://dl.acm.org/doi/10.1145/258915.258941>. Acesso em: 24 abr. 2024.

[16] 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. Disponível em: <https://link.springer.com/chapter/10.1007/BFb0026430>. Acesso em: 24 abr. 2024.

[17] CHOW, F.; HENNESSY, J. Register allocation by priority-based coloring. In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*. New York: ACM, 1984. (SIGPLAN '84), p. 222–232. ISBN 0897911393. Disponível em: <https://dl.acm.org/doi/10.1145/502949.502896>. Acesso em: 24 abr. 2024.

[18] CHOW, F. C.; HENNESSY, J. L. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, ACM, New York, v. 12, n. 4, p. 501–536, oct 1990. ISSN 0164-0925. Disponível em: <https://dl.acm.org/doi/10.1145/88616.88621>. Acesso em: 24 abr. 2024.

[19] BRIGGS, P. *Register Allocation via Graph Coloring*. Tese (Doutorado) — Rice University, Houston, Texas, 1992. Disponível em: <https://www.cs.utexas.edu/users/mckinley/380C/lecs/briggs-thesis-1992.pdf>. Acesso em: 24 abr. 2024.

[20] 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. Disponível em: <https://dl.acm.org/doi/abs/10.1145/177492.177575>. Acesso em: 24 abr. 2024.

[21] CALLAHAN, D.; KOBLENZ, B. Register allocation via hierarchical graph coloring. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. New York: ACM, 1991. (PLDI '91), p. 192–203. ISBN 0897914287. Disponível em: <https://dl.acm.org/doi/10.1145/113446.113462>. Acesso em: 24 abr. 2024.

[22] CHAITIN, G. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, ACM, New York, v. 39, n. 4, p. 66–74, apr 2004. ISSN 0362-1340. Disponível em: <https://dl.acm.org/doi/abs/10.1145/989393.989403>.

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

[24] 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, v. 32, n. 5, p. 109–121, 1997. Disponível em: <https://dl.acm.org/doi/abs/10.1145/258916.258926>. Acesso em: 24 abr. 2024.

[25] PALANCIUC, V.; BADEA, D. A spill code minimization technique – application in the metrowerks starcore c compiler. *International Journal of Parallel Programming*, v. 32, p. 475–499. Disponível em: <https://link.springer.com/article/10.1023/B:IJPP.0000042083.16504.5e>. Acesso em: 24 abr. 2024.

[26] 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*. 1992. p. 311–321. Disponível em: <https://dl.acm.org/doi/abs/10.1145/143103.143143>. Acesso em: 24 abr. 2024.

[27] Clang: a C language family frontend for LLVM. Disponível em: <https://clang.llvm.org/>. Acesso em: 24 abr. 2024.

[28] The LLVM Compiler Infrastructure Project. Disponível em: <https://llvm.org/>. Acesso em: 24 abr. 2024.

[29] opt - LLVM optimizer. Disponível em: <https://llvm.org/docs/CommandGuide/opt.html>. Acesso em: 24 abr. 2024.

[30] llc - LLVM static compiler. Disponível em: <https://llvm.org/docs/CommandGuide/llc.html>. Acesso em: 24 abr. 2024.

[31] Machine IR (MIR) Format Reference Manual. Disponível em: <https://llvm.org/docs/MIRLangRef.html>. Acesso em: 24 abr. 2024.

[32] Using the New Pass Manager. Disponível em: <https://llvm.org/docs/NewPassManager.html>. Acesso em: 24 abr. 2024.

[33] Writing an LLVM Pass. Disponível em: <https://llvm.org/docs/WritingAnLLVMNewPMPass.html>. Acesso em: 24 abr. 2024.

[34] Writing an LLVM Pass (legacy PM version). Disponível em: <https://llvm.org/docs/WritingAnLLVMPass.html>. Acesso em: 24 abr. 2024.

[35] LLVM::RegAllocBase Class Reference. Disponível em: <https://llvm.org/doxygen/classllvm_1_1RegAllocBase.html>. Acesso em: 24 abr. 2024.

[36] SPEC CPU® 2017. Disponível em: <https://www.spec.org/cpu2017/>. Acesso em: 24 abr. 2024.