Redução do *Spill Code* no Algoritmo *Linear Scan*

Guilherme Akira Demenech Mori[1], **Wesley Attrot**[1]

[1]Departamento de Computação – Universidade Estadual de Londrina (UEL)
Caixa Postal 10.011 – CEP 86057-970 – Londrina – PR – Brasil

akira.demenech@uel.br, wesley@uel.br

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.*

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.*

1. Introdução

O princípio da localidade temporal diz que, em geral, dados recentemente usados tentem a ser novamente usados em breve [15].

Os programas não utilizam com igual frequência toda a memória, acessando parte relativamente pequena de seus dados a cada instante. Por causa dessa heterogeneidade de usos, pode-se projetar os computadores com tipos diferentes de memória, uma parte lenta e barata em grande quantidade e uma fração muito menor, mas muito mais rápida e cara. Em função dessa organização hierárquica representada na Figura 1, os programas podem utilizar a pouca memória mais próxima do processador para os dados mais usados num certo momento e deixar no armazenamento maior o que será usado mais tarde ou com menos frequência [15].

Sendo a memória de mais rápido acesso e mais cara (Figura 1), os registradores são recursos escassos nos computadores e, dada a grande quantidade de variáveis necessárias pela maior parte dos programas, não há registradores suficientes para dedicar a todas elas [3].

Como a utilização de registradores para armazenar os valores das variáveis permite alto desempenho, a alocação e a atribuição de registradores são processos muito importantes para a qualidade do código gerado. Ambos realizados pelo alocador, esses processos ocorrem no *back end* do compilador, durante a geração de código da máquina

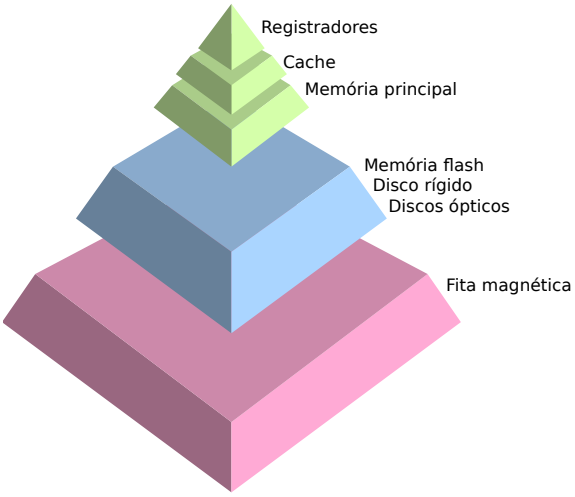

Figura 1. Hierarquia de memória: mais cara, em menor quantidade e mais próxima do processador quanto mais alta na pirâmide.

Fonte: autoria própria.

alvo, como apresenta a Figura 2. Na alocação de registradores são escolhidas quais variáveis utilizarão registradores a cada momento. Na atribuição são escolhidos os registradores que serão utilizados por cada variável. A atribuição pode ser resolvida em tempo polinomial, mas a alocação ótima, porém, é NP-completa[1], sofrendo ainda com as restrições específicas de *hardware* ou sistema operacional [3, 11].

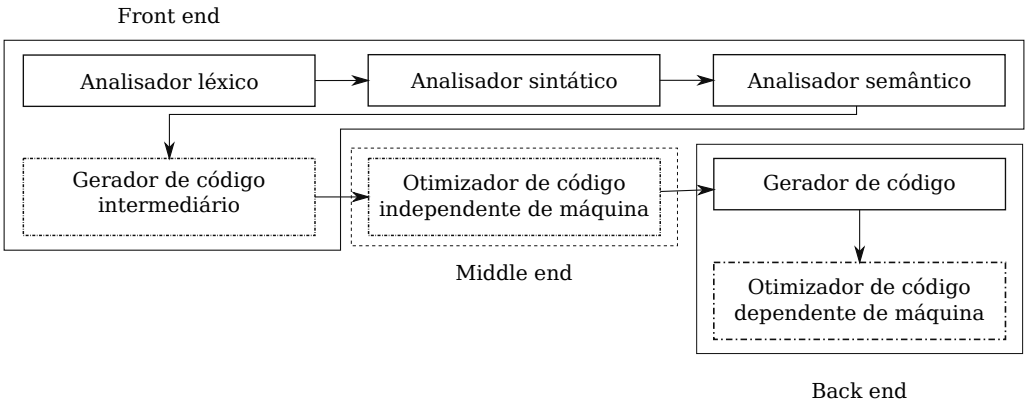

Figura 2. Fases da compilação: as fases pontilhadas podem não estar presentes em todos os compiladores.

Fonte: autoria própria.

Considerando-se quando os valores das variáveis são úteis e quando não serão mais utilizados (ou seja, em quais instruções estão vivas ou não), poucos registradores podem atender diversas variáveis. 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*[2] [15].

[1]Para os problemas não-determinísticos de tempo polinomial completo (NP-completos) são conhecidas somente soluções com pelo menos tempo exponencial [4].

[2]Usualmente traduzido como derramamento.

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.

Geralmente são alocados e atribuídos registradores 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 [16]. Essas informações permitem conhecer os *live ranges*[3] 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 compilação. Quando baixo tempo de compilação é necessário, técnicas mais rápidas também são necessárias, como na compilação *just-in-time* (JIT)[4] [12].

A compilação JIT pode se privilegiar de informações do código de alto nível e de detalhes conhecidos 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 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, como fixar os registradores disponíveis somente para as variáveis mais usadas, ou complexas demais, 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*. Esse algoritmo não demanda muito processamento e pode rapidamente gerar código satisfatório para acelerar a execução do programa.

A técnica de alocação *linear scan* [16] assume uma ordenação das instruções e simplifica *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,

[3] Tempos de vida.

[4] 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 [5].

ou este novo ou algum dos demais deverá sofrer *spill*. Mantendo a compilação rápida, essa alocação busca também desempenho aceitável para o código gerado. Embora não faça as escolhas mais otimizadas para geração de código de *spill*, a *linear scan* reduz significativamente o tempo de compilação se comparada à coloração de grafos [16].

A política de *spill* apresentada pelo algoritmo *linear scan* tradicional [16], 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` [7, 12]. Essa abordagem para geração de *spill code* muito simples pode ser aprimorada.

Com base no que foi apresentado, o objetivo deste trabalho é melhorar a geração de *spill code* na alocação de registradores *linear scan* sem causar perda de desempenho do alocador. A abordagem de *spill everywhere* será comparada experimentalmente com técnicas de *interference region spilling* [6] e *live range splitting* [10].

A seguir, a Seção 2 apresenta a revisão da literatura e trabalhos relacionados. A Seção 3 enumera os objetivos do trabalho. A Seção 4 explica o processo de execução e na Seção 5 estão previstos os prazos para ele. As contribuições esperadas deste trabalho estão expostas na Seção 6.

2. Fundamentação Teórico-Metodológica e Estado da Arte

A importância e complexidade da alocação de registradores faz com que costume ser implementada como uma etapa separada dentre as várias tarefas do gerador de código [12]. Geralmente ela é abordada como um problema de coloração de grafos ou de empacotamento e demandam muitas heurísticas para obter soluções satisfatórias no tempo disponível [14]. Os alocadores por coloração de grafos são muitos, com muitas melhorias nas heurísticas e políticas de *spill* desde a primeira proposta de Chaitin [9, 12].

O método de coloração de grafos consiste em construir o grafo de interferência com base nos *live ranges* das variáveis, simplificá-lo com diversas estratégias 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. Com *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].

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, como *linear scan*, quando a qualidade do código gerado puder ser reduzida em prol de limitações computacionais mais rígidas [16].

2.1. *Linear scan*

O algoritmo *linear scan* escaneia os *live intervals* e aloca os registradores em tempo linear. Alocar rapidamente registradores e gerar código de bom desempenho é seu objetivo.

Inicialmente, utilizando uma ordenação sequencial das instruções, percorre o código intermediário uma vez e identifica o *live interval* de cada variável. Ordena, então, os intervalos por início crescente. O *linear scan* irá passar pelos intervalos nessa ordem, considerando os intervalos ativos a cada ponto. Na Figura 3 estão ilustrados intervalos de variáveis.

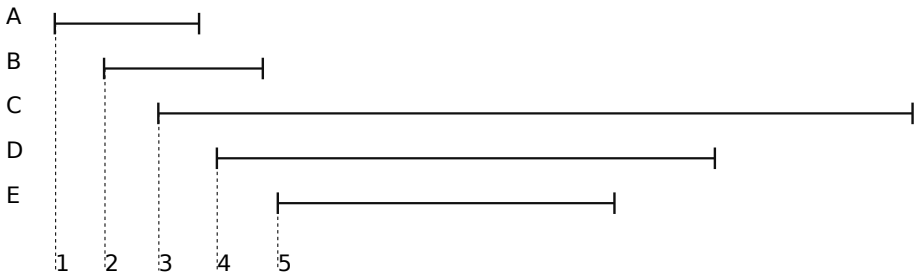

Figura 3. *Live intervals* das variáveis A a E com os pontos de início enumerados em ordem crescente.

Fonte: adaptado de Poletto e Sarkar [16].

Na lista de intervalos ativos 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 de início 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 [16] escolhe o intervalo que termina por último. Essa heurística muito simples permite uma rápida decisão de *spill* e evita que mais decisões sejam necessárias, minimizando a quantidade de intervalos movidos para memória.

No exemplo da Figura 3, 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, $\langle A \rangle$;
2. B é adicionado à lista, que resulta em $\langle A, B \rangle$;
3. Uma decisão de *spill* é necessária. C termina por último, então sofre *spill* e não entra na lista, que se mantém como $\langle A, B \rangle$;
4. A é removido da lista e D é adicionado, $\langle B, D \rangle$;
5. B é removido da lista e E é adicionado, $\langle E, D \rangle$.

Para facilitar a remoção dos *live intervals* expirados, a lista de intervalos ativos pode ser mantida em ordem crescente de ponto de término. Essa ordenação da lista também permite que a decisão de *spill* seja realizada com bem pouco custo [16]: somente uma comparação, entre o último intervalo da lista e o novo intervalo.

Variáveis movidas para a memória sofrem *spill everywhere* [16]. Elas residem na memória principal durante todo o *live interval*, devendo ser carregadas para todos os usos e armazenadas novamente a cada redefinição.

2.2. 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. Muitas pesquisas foram desenvolvidas a respeito dessa escolha e de como alterar o código do programa. Muitas melhorias à política de *spill everywhere* do alocador de Chaitin [9] foram propostas, como as diferentes heurísticas e a limpeza (abordagem *spill almost everywhere*) de Bernstein et al. [7] e a rematerialização de valores cuja recomputação é mais barata que a transferências em memória [8]. Foram selecionadas para este trabalho duas técnicas para reduzir *spilling*.

As duas políticas que serão descritas a seguir, *interference region spilling* e *live range splitting*, evitam *spill everywhere* restringindo a maneira de gerar *spill code* quando ele é necessário. Essas técnicas não dependem do método de coloração de grafos, podendo ser aplicadas em outros tipos de alocadores como, neste trabalho, no *linear scan*. São relacionadas entre si pela ideia de gerar código de *spill* mais preciso, somente onde é necessário. Também não são antagônicas, pois ambas adotam a estratégia *spill everywhere* quando não conseguem se provar benéficas. Portanto podem ser utilizadas conjuntamente, escolhendo a menos custosa das duas, ou mesmo a própria *spill everywhere*.

2.2.1. *Interference region spilling*

A técnica de *interference region spilling* [6] consiste em observar, entre dois *live ranges* que se interferem, em qual parte do programa estão ativos simultaneamente. 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 4 mostra a diferença entre *spill everywhere* (no centro) e *interference region spill*.

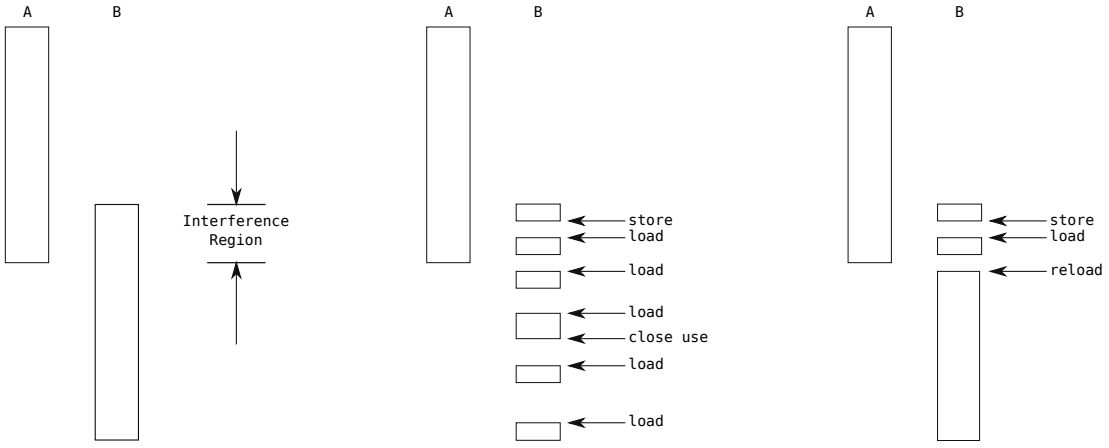

Figura 4. À esquerda, os *live ranges* das variáveis A e B, identificada a região de interferência. Ao centro, B sofre *spill everywhere*. À direita, B sofre *spill* somente na região de interferência.

Fonte: Bergner, Dahl, Engebretsen e O'Keefe [6].

Essa técnica insere menos `load` e `store` que *spill everywhere* por restringir o *spill* à região de interferência, porém, como é necessário recarregar a variável se depois ela for utilizada, é possível que sejam inseridas instruções `load` que não seriam inseridas pela *spill everywhere*. Para evitar que esses carregamentos adicionais sejam piores que o *spill* completo, basta que nessas situações seja utilizada a abordagem de *spill everywhere* em seu lugar.

2.2.2. *Live range splitting*

A técnica de *live range splitting* [10] 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. Para isso, é preciso que os pontos de *split* remova interferências ou as divida 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*.

No fluxograma à esquerda, a Figura 5 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 lado direito da Figura 5). 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*.

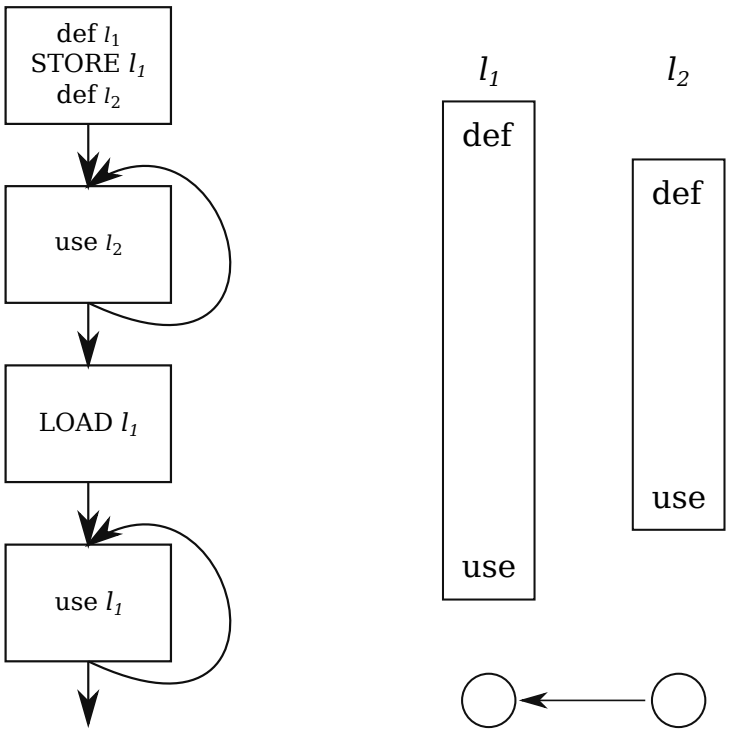

Figura 5. *Splitting* de l_1 ao redor de l_2 e seus respectivos *live ranges*.
Fonte: adaptado de Cooper e Simpson [10].

3. Objetivos

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* e *live range splitting*. 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.

4. Procedimentos metodológicos

Para o desenvolvimento do trabalho, serão implementados o alocador *linear scan* tradicional e, nele, as técnicas de *interference region spilling* e *live range splitting* com o *framework* LLVM [1, 13].

O LLVM fornece ferramentas de código aberto para criação de compiladores e otimizadores. Sua plataforma *open source* cria ambiente propício para que diversos subprojetos de análise, otimização e geração de código para as mais diversas linguagens de programação [1].

A geração de código será implementada para a arquitetura x86-64 e a comparação de desempenho utilizará testes de *benchmark* SPEC CPU®2017 [2]. O desempenho do alocador aprimorado será comparado com o alocador tradicional implementado e com os módulos de alocação de registradores e geração de *spill code* existentes no LLVM. Além dos tempos de compilação, serão comparadas as quantidades de instruções `load` e `store` nos códigos gerados e também o desempenho de execução deles.

5. Cronograma

As atividades previstas para o trabalho são:

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

A Tabela 1 apresenta a distribuição dessas atividades pelo tempo de execução do trabalho.

Tabela 1. Cronograma de Execução

Atividade	**Mês**							
	01	02	03	04	05	06	07	08
Levantamento bibliográfico	X	X	X					
Implementação do alocador			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

6. Resultados esperados

É esperado que ambas as implementações *linear scan* (tradicional e aprimorada) tenham alocação significativamente mais rápida que a coloração de grafos. Embora a geração de código não vá se demonstrar superior que a coloração de grafos, há expectativa de que o desempenho seja comparável.

Espera-se, também, que seja pequena a redução de velocidade da alocação com as técnicas de *interference region spilling* e *live range splitting* e que essa seja justificada pelo melhor desempenho do código gerado. Essa melhoria está associada à expectativa de redução de geração de *spill code* e do seu custo na execução.

7. Espaço para assinaturas

Londrina, 13 de setembro de 2023.

Guilherme Akira Demenech Mori

Wesley Attrot

Referências

[1] The LLVM compiler infrastructure project. `https://llvm.org/`. Acessado em: 08 set. 2023.

[2] SPEC CPU®2017. `https://www.spec.org/cpu2017/`. Acessado em: 28 ago. 2023.

[3] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman, editors. *Compilers: principles, techniques, & tools*. Pearson/Addison Wesley, Boston, 2nd edition, 2007. OCLC: ocm70775643.

[4] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman, editors. *Compiladores: princípios, técnicas e ferramentas*. LTC, Rio de Janeiro, 1995.

[5] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[6] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. *ACM SIGPLAN Notices*, 32(5):287–295, 1997.

[7] David Bernstein, M Golumbic, Y Mansour, R Pinter, D Goldin, Hugo Krawczyk, and Itai Nahshon. Spill code minimization techniques for optimizing compliers. *ACM SIGPLAN Notices*, 24(7):258–263, 1989.

[8] Preston Briggs, Keith D Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.

[9] Gregory J Chaitin. Register allocation & spilling via graph coloring. *ACM Sigplan Notices*, 17(6):98–101, 1982.

[10] Keith D Cooper and L Taylor Simpson. Live range splitting in a graph coloring register allocator. In *International Conference on Compiler Construction*, pages 174–187. Springer, 1998.

[11] Keith D Cooper and Linda Torczon. *Construindo Compiladores*. Elsevier, Rio de Janeiro, 2nd edition, 2014.

[12] Keith D Cooper and Linda Torczon. *Engeneering a Compiler*. Elsevier, Morgan Kaufmann, Cambridge, MA, 3nd edition, 2023.

[13] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[14] Steven S Muchnick. Advanced compiler design and implementation, 1997.

[15] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Elsevier, Morgan Kaufmann, Amsterdam Heidelberg, 3rd edition, 2005.

[16] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.