

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA**

Felipe dos Santos Silveira

ALOCAÇÃO DE REGISTRADORES UTILIZANDO INTELIGÊNCIA ARTIFICIAL

Florianópolis

2013

Felipe dos Santos Silveira

ALOCAÇÃO DE REGISTRADORES UTILIZANDO INTELIGÊNCIA ARTIFICIAL

Trabalho de Conclusão de Curso submetido ao Curso de Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Ricardo Azambuja Silveira

Florianópolis

2013

Catálogo na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:

<http://www.bu.ufsc.br/design/Catalogacao.html>

Felipe dos Santos Silveira

ALOCAÇÃO DE REGISTRADORES UTILIZANDO INTELIGÊNCIA ARTIFICIAL

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de “Bacharel em Ciências da Computação”, e aprovado em sua forma final pelo Curso de Ciências da Computação.

Florianópolis, 27 de maio 2013.

Prof. Dr. Vitório Bruno Mazzola
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Ricardo Azambuja Silveira
Orientador

Profa. Dra. Jerusa Marchi

Prof. Dr. Olinto José Varela Furtado

Program the brain, not the heartbeat.

Computer God, Black Sabbath

RESUMO

Alocação de registradores é o processo que define quais variáveis e valores intermediários do código fonte ficam em registradores e quais ficam em memória. O alocação de registradores é uma fase importante da geração de código pois ela pode criar acessos à memória em partes do código que são muito executadas, deteriorando o desempenho.

O problema da alocação é comumente resolvido construindo um grafo de interferência, que modela quais variáveis não podem residir no mesmo registrador, e achando uma *k-coloração* para ele, onde *k* é o número de registradores disponíveis na arquitetura alvo da compilação.

Neste trabalho foi implementado um alocador de registradores utilizando o algoritmo de otimização por colônia de abelhas para a coloração do grafo de interferência. Os testes foram feitos em três casos, com baixa, média e alta demanda por registradores.

O algoritmo proposto mostrou-se viável após implementado no LLVM, tendo resultados satisfatórios em todos os casos de teste.

Palavras-chave: Alocação de Registradores, Compiladores, Geração de Código, Inteligência Artificial.

ABSTRACT

Register allocation is the process that define which variables and temporary values of the source code will be allocated to registers and which ones will be allocated on memory. Register allocation is important phase on the code generation process because it can generate memory access on very busy areas of the code, thus decreasing performance.

The register allocation problem is usually solved creating a interference graph, that represents which variables cannot reside on the same register, and finding a *k-coloring* for it, where *k* is the number of available registers on the target machine architecture.

In this work a register allocator was implemented using the Bee Colony Optimization algorithm to color the interference graph. The tests were wade base on 3 cases, with low, medium and high register demand.

The proposed algorithm proved viable when implemented on LLVM, obtaining satisfiable results on all test cases.

Keywords: Register Allocation, Compilers, Code Generation, Artificial Intelligence

LISTA DE FIGURAS

Figura 1	<i>Live ranges</i> de um trecho de código (TORCZON; COOPER, 2011)	25
Figura 2	Grafo de interferência do trecho de código da figura 1	27
Figura 3	Coloração a partir de uma sequência	34
Figura 4	Geração de uma sequência aleatória de nodos	35
Figura 5	Aplicação da Estratégia I	36
Figura 6	Aplicação da Estratégia II	37
Figura 7	Aplicação da Estratégia III	37

LISTA DE TABELAS

Tabela 1	Resultados para a soma de parâmetros	46
Tabela 2	Tempos de compilação (em segundos) para a soma de parâmetros com o alocador implementado	47
Tabela 3	Resultados para o cálculo de determinante	47
Tabela 4	Tempos de compilação (em segundos) para o cálculo de determinante com o alocador implementado	47
Tabela 5	Números médio de <i>spills</i> para o cálculo de determinante com o alocador implementado	48
Tabela 6	Resultados para a multiplicação de matrizes	48
Tabela 7	Tempos de compilação para a multiplicação de matrizes com o alocador implementado	49
Tabela 8	Números médios de <i>spills</i> para a multiplicação de matrizes com o alocador implementado	49

SUMÁRIO

1 INTRODUÇÃO	19
1.1 MOTIVAÇÃO	19
1.2 OBJETIVOS	20
1.2.1 Objetivo Geral	20
1.2.2 Objetivos Específicos	20
2 COMPILADORES E ALOCAÇÃO DE REGISTRADORES	21
2.1 VISÃO GERAL DE UM COMPILADOR	21
2.2 ALOCAÇÃO DE REGISTRADORES	23
2.2.1 Alocação Local	24
2.2.2 Alocação Global	25
3 TRABALHOS CORRELATOS	31
3.1 ALGORITMOS EVOLUCIONÁRIOS	31
3.2 ESCOLHA DE IMPLEMENTAÇÃO	31
4 ALOCAÇÃO DE REGISTRADORES COM ALGORITMOS EVOLUCIONÁRIOS	33
4.1 OTIMIZAÇÃO POR COLÔNIA DE ABELHAS	33
4.2 COLORAÇÃO DE GRAFOS COM OTIMIZAÇÃO POR COLÔNIA DE ABELHAS	34
4.2.1 Obtenção de Sequências Vizinhas	35
5 FERRAMENTAS	39
5.1 LLVM	39
6 IMPLEMENTAÇÃO	41
6.1 IMPLEMENTAÇÃO NO LLVM	41
6.1.1 Spill de Variáveis	42
6.1.2 Representação de Convenções da Máquina Alvo	43
6.1.3 Sequências Vizinhas	44
7 RESULTADOS	45
7.1 ALGORITMOS DE ALOCAÇÃO DE REGISTRADORES DO LLVM	45
7.1.1 Basic	45
7.1.2 Greedy	45
7.2 CASOS DE TESTE	46
7.2.1 Parâmetros de comparação	46
7.2.2 Função Simples	46
7.2.3 Função Aritmética de Média Complexidade	47

7.2.4 Função com Alta Demanda de Registradores	48
7.3 REPRODUÇÃO DOS DADOS.....	49
8 CONCLUSÕES	51
8.1 TRABALHOS FUTUROS.....	51
ANEXO A – Código do caso de teste de função simples	55
ANEXO B – Código do caso de teste de cálculo de determinante	59
ANEXO C – Código do caso de teste de multiplicações de matrizes	63
Referências Bibliográficas	65

1 INTRODUÇÃO

Arquiteturas modernas se baseiam na hierarquia de memória para armazenar os dados de seus programas. No topo desta hierarquia, ficam os registradores, que são pequenas memórias e têm o menor tempo de acesso (operações que operam somente sobre registradores não duram mais do que um ciclo de relógio) (PATTERSON; HENNESSY, 2011). Porém, os registradores apresentam custo muito alto, tornando o seu uso em altas quantidades impraticável. Como exemplo de máquinas reais, pode-se citar a muito popular arquitetura x86, que apresenta apenas 8 registradores de propósito geral.

Alocação de registradores é umas das etapas finais da compilação, ela decide quais variáveis devem ser armazenadas em registradores e quais devem ficar em memória. Acessos à memória principal podem ser até 150 vezes mais lento que um acesso à registrador, por isso é desejável que o código gerado pelo alocador tenha o mínimo possível de acessos à memória.

Compiladores modernos modelam a etapa de alocação de registradores utilizando um grafo, e um dos passos intermediários para resolver o problema é achar uma coloração para o grafo com no máximo k cores, onde k é o número de registradores de propósito geral na arquitetura da máquina alvo (TORCZON; COOPER, 2011).

1.1 MOTIVAÇÃO

A alocação de registradores é uma etapa que está presente em todos os compiladores e impacta fortemente a performance do código gerado. Além disso, ela trabalha com problemas difíceis aplicados com requisitos do mundo real.

A abordagem mais utilizada para implementação de alocação de registradores é redução da alocação à uma coloração de grafos, porém, este é um problema NP-difícil. Por isso, a coloração é feita por meio de heurísticas, entretanto, elas nem sempre produzem os melhores resultados. É neste ponto que a inteligência artificial se encaixa. Ela é muito utilizada para obter soluções boas para problemas difíceis, como o caso da coloração de grafos.

Como já foi mostrado em outros trabalhos (LINTZMAYER; MULATI; SILVA, 2011), é possível ter uma solução aceitável para a alocação de registradores utilizando inteligência artificial. Este trabalho pretende estudar os efeitos causados pela substituição do algoritmo de alocação de um compilador já estabelecido e bastante utilizado por um algoritmo utilizando inteligência artificial.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Proposta e implementação de um algoritmo de alocação de registradores que utilize uma técnica de inteligência artificial em um compilador que seja de código aberto e amplamente utilizado.

1.2.2 Objetivos Específicos

- Estudo sobre como compiladores implementam Alocação de Registradores.
- Proposta e implementação de um algoritmo de alocação utilizando inteligência artificial em um compilador real.
- Comparação entre a implementação original e a proposta.

2 COMPILADORES E ALOCAÇÃO DE REGISTRADORES

2.1 VISÃO GERAL DE UM COMPILADOR

Como definido por (AHO et al., 2006), um compilador é um programa que é capaz de ler um programa em uma linguagem e traduzí-lo para um programa equivalente em outra linguagem, reportando possíveis erros no programa origem.

Para corretamente traduzir um programa de uma linguagem para outra, o compilador precisa conhecer perfeitamente a linguagem origem. É necessário saber todos os símbolos permitidos na linguagem origem, saber como esses símbolos podem ser agrupados para formar um programa e finalmente entender o que o programa origem quer fazer para criar um programa equivalente na linguagem destino.

A estrutura de um compilador reflete essas necessidades (TORCZON; COOPER, 2011), sendo claramente dividida em duas partes: o *frontend*, que lida com a linguagem de origem e consiste basicamente da parte de análise do programa de entrada e o *backend*, que lida com a geração do programa na linguagem destino.

O *frontend* pode ser dividido em quatro partes (TORCZON; COOPER, 2011):

- **Análise léxica:** Agrupa os símbolos do programa de entrada em pares $\langle token, valor \rangle$. Onde um token é o elemento presente na gramática da linguagem (por exemplo, *numero*) e o seu valor é o que estava no programa de entrada, por exemplo *42*.
- **Análise sintática:** Verifica se o programa entrada é válido de acordo a especificação de sua gramática. Além disso, também é responsável por gerar mensagens de erro e, se desejável, inserir tokens extras no programa de entrada para tentar corrigir os erros encontrados e seguir com a análise para reportar mais erros.
- **Análise semântica:** É responsável por garantir que o programa de entrada, que já foi verificado pelo analisador léxico e sintático, faça sentido. Ou seja, o analisador semântico verifica se as chamadas de funções são feitas com parâmetros certos, que atribuições sejam feitas entre tipos compatíveis, e que as regras de escopo e visibilidade sejam respeitadas.
- **Geração de código intermediário:** Para ajudar a traduzir um programa, o compilador constrói uma ou mais representações do código de entrada, como por exemplo uma árvore de sintaxe. Após a análise semântica, é gerado um código próximo de linguagem de máquina, porém independente de alguma arquitetura específica. Esse código é mani-

pulado por várias fases futuras da compilação, como otimização, escolha de instruções e alocação de registradores. O código intermediário de baixo nível é altamente interessante para o reuso de código das fases da compilação pois desassocia o *frontend* do *backend*, possibilitando que programas em várias linguagens sejam traduzidas para a mesma representação intermediária que utilizará apenas um *backend* para gerar o código de máquina final. De forma análoga, Um mesmo programa pode ser traduzido para várias arquiteturas apenas utilizando-se de diferentes *backends*.

Após a geração do código intermediário, há uma fase opcional chamada de **Otimização**. A otimização trabalha sobre o código intermediário para encontrar um código equivalente ao original, porém melhor. A qualificação de “melhor” geralmente significa mais rápido, porém pode significar *menor* ou *menor gasto de energia*, dependendo do foco do usuário. Outra informação relevante é que após a otimização, o código resultado não é o *ótimo*, como o nome da fase sugere, pois muitos problemas tratados por otimizações são computacionalmente difíceis e múltiplas otimizações influenciam umas às outras (TORCZON; COOPER, 2011).

Há dois conjuntos de otimizações (MUCHNICK, 1997) que podem ser aplicadas sobre o código intermediário, o primeiro conjunto é de *otimizações independentes de arquitetura*, elas são focadas em análises que são verdade para todas as arquiteturas. Esse conjunto inclui substituição de constantes, remoção de invariantes de laço. O outro conjunto é o de *otimizações dependentes de arquitetura*, essas otimizações levam em conta características da máquina, como modos de endereçamento e sequências de instruções.

Ambos os conjuntos de otimizações podem ser aplicados sobre a mesma representação ou também podem ser aplicados sobre duas representações diferentes: as otimizações independentes de arquitetura podem ser aplicadas sobre uma representação intermediária de mais alto nível, e as otimizações dependentes de arquitetura são feitas sobre uma representação de mais baixo nível que expõe características da arquitetura alvo.

A parte final do processo de compilação é feita pelo *backend*. As tarefas do backend são (TORCZON; COOPER, 2011):

- **Seleção de instruções:** Mapeia as instruções utilizadas no código intermediário para instruções presentes na arquitetura alvo. Algumas instruções no código intermediário podem não estar presentes no conjunto de instruções da máquina alvo e portanto são mapeadas para um conjunto de instruções equivalentes.
- **Escalonamento de código:** Tem a responsabilidade de reordenar as instruções para tirar proveito de características de tempo de execução de cada instrução. Por exemplo, é possível carregar o valor de um endereço de memória para um registrador enquanto se faz operações aritméticas não dependentes deste valor. O escalonamento de código pode ser

visto como uma otimização, pois produz um código equivalente ao original e caso não seja feito não há nenhum outro efeito negativo além de maior tempo de execução (processadores modernos detectam acessos a dados que não estão válidos e inserem *stalls*, ou seja, operações que não alteram nenhum dado e só servem para atrasar a execução, no pipeline para garantir a execução correta (PATTERSON; HENNESSY, 2011)).

- **Alocação de registradores:** A alocação de registradores consiste em decidir quais variáveis deverão residir em memória e quais deverão residir em registradores presentes no processador. Dependendo de como foi montada a representação intermediária, ela pode ser vista como um passo necessário ou uma otimização (TORCZON; COOPER, 2011). Se a representação intermediária só trabalha com valores em memória, o alocador tem um papel de otimizador, pois o código original já era funcional e as mudanças apenas melhorarão a performance. Caso contrário, o código intermediário usou quantos registradores foram necessários, assim o alocador precisa decidir quais variáveis ficarão em memória e quais ficarão em registradores físicos e gerar código para refletir essas decisões.

A alocação de registradores apresenta grande complexidade computacional quando aplicada em compiladores reais (TORCZON; COOPER, 2011). A próxima sessão será dedicada a explicar o problema da alocação de registradores e como ele é tratado por compiladores atuais.

2.2 ALOCAÇÃO DE REGISTRADORES

Registradores são as unidades de armazenamento mais rápidas encontradas na hierarquia de memória. E em algumas máquinas são os únicos lugares que a maior parte das instruções podem acessar (PATTERSON; HENNESSY, 2011). Nos últimos anos, a performance interna dos processadores aumentou significativamente, porém a latência de acesso à memória não acompanhou esse desenvolvimento, deixando ainda mais crítico o acesso à ela e aumentando a importância de compiladores gerarem um código com uso eficiente de registradores (KENNEDY; ALLEN, 2002).

A parte do compilador responsável por gerar o código com um bom uso de registradores é o *Alocador de Registradores*. O alocador tem como entrada uma das partes finais da compilação, o código já foi analisado léxica, sintática e semanticamente, otimizado, e já foi feita a substituição de instruções da representação intermediária por instruções da máquina alvo. Sua saída é um código equivalente que usa apenas registradores presentes na máquina alvo, assim tendo que adicionar instruções para salvar e carregar dados que não podem ser mantidos em registradores.

Como já exposto, a escolha da representação intermediária tem alto impacto no trabalho

do alocador. Quando se trabalha com uma representação registrador-registrador, muito conhecimento sobre o código fica explícito nas instruções, porém, quando se usa um modelo memória-memória, há o trabalho extra de verificar se um determinado valor pode ser copiado para um registrador antes de fazer a alocação. Por isso, é mais comum representações intermediárias trabalharem com o modelo registrador-registrador (TORCZON; COOPER, 2011).

Um alocador de registradores resolve dois problemas distintos em sequência:

- **Alocação:** Garante que o resultado final respeite as restrições da máquina alvo, gerando código de *spill* para guardar e carregar variáveis que não podem residir em registradores.
- **Atribuição:** Atribui um registrador da máquina alvo para cada registrador virtual no código gerado pela alocação.

2.2.1 Alocação Local

Alocação de registradores é um processo computacionalmente difícil, podendo ser resolvida em tempo polinomial apenas no caso mais simples e para apenas um bloco básico, tornando-se NP-difícil quando adicionamos mais blocos (KENNEDY; ALLEN, 2002), ou qualquer outra característica presentes em máquinas reais, como por exemplo a arquitetura x86, que permite um registrador de 32 bits armazenar um dado de 32 bits, 2 de 16 ou ainda 4 dados de 8 bits. A atribuição é relativamente mais simples, pois já foi garantido pelo alocador que o número de registradores usados não é maior que a quantidade disponível, e geralmente pode ser resolvido em tempo polinomial.

Os algoritmos de alocação de registradores possuem duas abordagens básicas como descritas em (TORCZON; COOPER, 2011)

- **Alocadores Top-Down:** Tentam manter os valores mais usados em registradores, para isso usam informações de alto nível para decidir quais variáveis deverão ser armazenadas em memória.
- **Alocadores Bottom-Up:** Usa apenas o código intermediário para tomar suas decisões, somente tendo informações sobre as definições e usos de cada valor.

Como definido em (PATTERSON; HENNESSY, 2011), um bloco básico é uma sequência de instruções sem instruções de desvio, exceto possivelmente no fim, e que não definem *labels*, exceto no início. Dado à essa característica sequencial de um bloco básico, algoritmos de alocação simples geram bons resultados, e alguns autores consideram os resultados como ótimos (TORCZON; COOPER, 2011).

1	loadI	...	⇒	rarp		
2	loadAI	rarp, @a	⇒	ra	1	rarp [1,11]
3	loadI	2	⇒	r2	2	ra [2,7]
4	loadAI	rarp, @b	⇒	rb	3	ra [7,8]
5	loadAI	rarp, @c	⇒	rc	4	ra [8,9]
6	loadAI	rarp, @d	⇒	rd	5	ra [9,10]
7	mult	ra, r2	⇒	ra	6	ra [10,11]
8	mult	ra, rb	⇒	ra	7	r2 [3,7]
9	mult	ra, rc	⇒	ra	8	rb [4,8]
10	mult	ra, rd	⇒	ra	9	rc [5,9]
11	storeAI	ra	⇒	rarp, @a	10	rd [6,10]

Figura 1 – *Live ranges* de um trecho de código (TORCZON; COOPER, 2011)

Utilizando a nomenclatura de algoritmos de otimização, os algoritmos que trabalham apenas sobre um bloco básico são chamados algoritmos *locais*, e os que trabalham sobre o conjunto de blocos básicos que formam o corpo de um procedimento (ou função) são chamados de algoritmos *globais* (TORCZON; COOPER, 2011). Para tirar proveito de casos entre blocos básicos e ter uma melhor visão de variáveis dentro de laços (que não são cobertos por apenas um bloco básico) é necessário um algoritmo de *alocação de registradores global*.

2.2.2 Alocação Global

Os algoritmos de alocação global refinam um pouco mais o conceito do que é alocado para registradores. Ao invés de variáveis, eles trabalham com o conceito de *live ranges*, que é o intervalo entre a definição de um valor até o último uso desta definição. Um *live range* não se restringe à somente variáveis, todos os valores usados no código são parte de algum *live range*, como por exemplo, resultados intermediários de operações aritméticas, ou ainda resultados de cálculos de endereço. Com essa definição, é possível que uma variável possa estar em um registrador diferente em cada um dos *live ranges* que a compõe.

A figura ?? apresenta um trecho de código de um bloco básico e identifica todos os *live ranges* encontrados. Na tabela apresentada, à cada *live range* é atribuído um número, é identificado qual é o registrador que ele está associado e suas linhas de definição e último uso.

A aplicação do conceito de *live ranges* é trivial quando só se trabalha com um bloco básico. Porém, ela não é diretamente aplicável quando mudamos o escopo do alocador para global, pois os algoritmos de alocação local não levam em consideração o que acontece depois do fim do bloco básico. Por exemplo, uma variável x pode estar em um registrador r_1 em

um bloco b_1 e em um registrador r_2 em um bloco b_2 , logo, em um bloco b_3 que é sucessor tanto de b_1 como de b_2 , não se tem certeza sobre qual registrador contém a variável x . Para contornar essa dificuldade do algoritmo é preciso utilizar alguns conceitos usados em técnicas de otimização global. Para o procedimento que está sendo analisado, é necessário saber o seu *Grafo de Controle de Fluxo* e para cada bloco básico, o seu conjunto *Live Out*.

O *Grafo de Controle de Fluxo* é um grafo direcionado $G(v,a)$ onde v é um vértice que representa um bloco básico contido no procedimento e existe uma aresta v,v' se e somente se existe uma instrução de desvio em v que tem como alvo algum label dentro de v' (TORCZON; COOPER, 2011). Ou seja, existe uma aresta de v para v' se e somente se v' pode ser executado depois de v . O *Conjunto Live Out* é o conjunto de variáveis que está viva, ou seja que não foi redefinida dentro do bloco, ao fim do bloco básico (KENNEDY; ALLEN, 2002). O conjunto *Live Out* pode ser encontrado segundo a seguinte equação:

$$LiveOut(n) = \bigcup_{b \in \text{sucessores}(n)} VPE(b) \cup (LiveOut(b) \cap \overline{Definicoes(b)})$$

Onde o conjunto *VPE* é definido pelas variáveis previamente expostas ao bloco, ou seja, as variáveis que podem ser usadas pois já foram definidas em algum bloco básico anterior. O conjunto *Definicoes* contém todas as variáveis que foram definidas dentro do bloco b , e o seu complemento contém todas as variáveis do procedimento que não foram definidas dentro do bloco.

Utilizando o conjunto *Live Out* de cada bloco básico, pode-se, ao final do bloco, gerar código para salvar em memória as variáveis que pertencem ao seu conjunto *Live Out* ao início de cada bloco, gerar código para carregar as variáveis que serão utilizadas. Essa modificação produz um código correto para todos os casos ao custo de muitos acessos à memória serem inseridos, sendo que alguns deles poderiam não ser necessários, ou ainda, podem deteriorar muito o desempenho pois são executados muitas vezes, por exemplo se forem inseridos dentro de laços.

Para evitar a adição de acessos à memória que não são necessários, o compilador precisa calcular os *live ranges globais*. Para isso, primeiro é feito a transformação do código para a forma *Static Single-Assignment (SSA)*, onde um uso de uma variável só pode ser alcançado por apenas uma definição dela (esta condição ocorre quando blocos predecessores redefinem a variável, como é comum em cláusulas *if-then-else*), e cada definição introduz um novo nome, assim eliminando múltiplas definições de uma mesma variável. No início de um bloco básico b que contém mais que um predecessor, se for identificado que para todos os blocos predecessores $pd_{1..n}$, se todo caminho até pd_i contém uma definição de uma variável v que algum dos outros predecessores não tem, é inserido uma *função- ϕ* , que agrupa todas as definições anteriores em

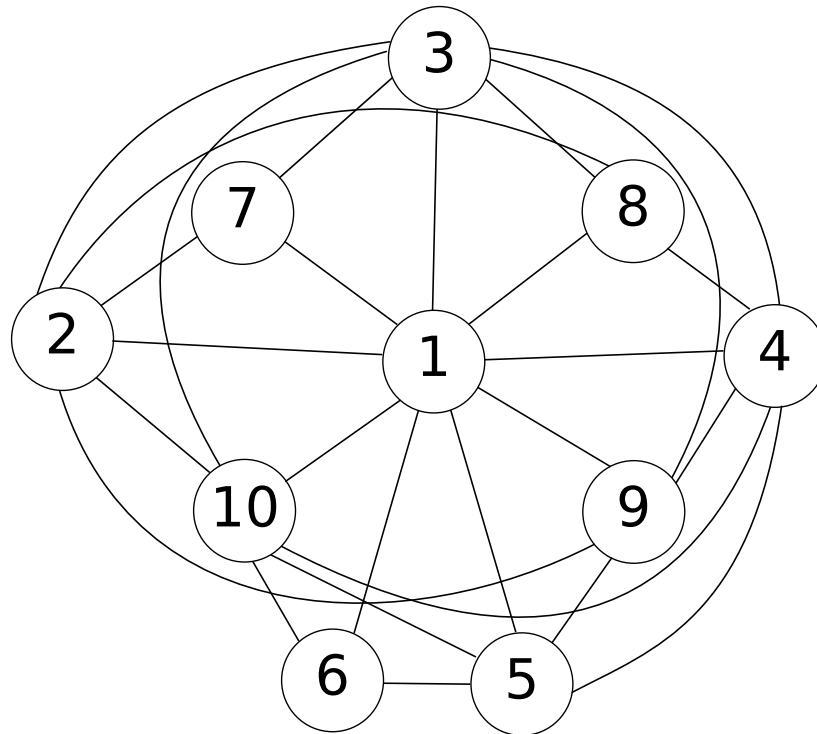


Figura 2 – Grafo de interferência do trecho de código da figura 1

uma nova definição para manter a forma *Static Single-Assignment* (KENNEDY; ALLEN, 2002).

A estratégia mais adotada por compiladores atuais para a alocação global de registradores é reduzir o problema da alocação de registradores ao problema de coloração de grafos. Para atingir esse objetivo, utiliza-se o código na forma *SSA* substituindo todas as definições de uma variável por somente um *live range*. À partir dessa mudança, constrói-se um *Grafo de interferência* $G(v,a)$ onde v é um *live range* no código e existe uma aresta não direcionada $a = (v_1, v_2)$ se e somente se v_1 interfere com v_2 , ou seja, se no momento da definição de v_1 ainda existe um uso para v_2 sem que este seja redefinido, ou seja, se v_2 ainda está *vivo*.

A figura 2 mostra o *grafo de interferência* construído para o trecho de código apresentado na figura 1. Nesse grafo, os nós estão nomeados de acordo com a sua numeração na tabela da figura 1. Exemplificando a construção do grafo, o nó **1** interfere com *todos* os outros nós, pois o seu *live range* cobre todas as linhas do código. Já o nó **4** não interfere com o nó **7**, pois a sua definição é após o último uso do *live range* **7**.

Com o *Grafo de Interferência* construído, o processo de alocação passa a ser encontrar uma *k-coloração* para ele, onde k é o número de registradores presentes na máquina alvo. Obviamente, em vários casos não é possível achar uma k -coloração para o grafo de interferência e o alocador precisa decidir inserir código de *spill* para alguma variável. Assim como na alocação local, existem duas abordagens para os alocadores globais (TORCZON; COOPER, 2011).

Os alocadores *Top-Down* utilizam informações de baixo nível para atribuir cores aos *live*

ranges e informações de alto nível para definir a ordem de coloração dos *live ranges*. A ordem de coloração é definida por um custo associado ao *spill* de cada *live range*. Algumas heurísticas ajudam a definir o custo, por exemplo, se um *live range* é composto somente por um *load* e um *store* no mesmo endereço, significa que o uso do dado foi removido por um passo de otimização ou o registrador contém o resultado de uma chamada de função, logo pode-se priorizar o *spill* deste *live range*, atribuindo um custo de *spill* negativo, pois este *spill* pode evitar que seja gerado código de cópia de valores, diminuindo o tempo de execução. Outra heurística é utilizada quando nenhum *live range* é definido entre o início e fim de um *live range*. Neste caso, é melhor não gerar código de *spill*, atribuindo ao *live range* um custo infinito, pois a demanda por registradores não cresce dentro do *live range*, logo o código gerado só deterioraria a performance.

Por fim, o compilador, ao fazer a análise do código pode adicionar, nos blocos básicos informações sobre o seu esperado número de execuções, por exemplo, pode-se assumir que um bloco dentro de um laço execute 10 vezes, e um laço aninhado execute 100 vezes. Com as informações sobre custos de *spill*, o alocador *Top-Down* prioriza a alocação dos *live ranges* de maior custo para registradores, e quando não é possível, gera código de *spill* para os *live ranges* de menor custo.

Alocadores *Bottom-Up* trabalham somente com o conhecimento sobre a estrutura do grafo de interferência. Os algoritmos criam uma ordem de prioridade para os nodos do grafo. Primeiro são retirados todos os nodos (e suas arestas) que estão *irrestritos*, ou seja, têm grau menor que k , isso pode transformar nodos previamente restritos em irrestritos. Quando só existir nodos restritos no grafo, é necessário escolher algum nodo para remover, essa decisão é feita por um critério externo e pode usar as definições de custos de *spill* para os alocadores *Top-Down*.

Ambas as técnicas apresentam uma questão que precisa ser tratada. Assume-se que a máquina alvo possui k registradores disponíveis, porém, não se leva em conta os registradores necessários para executar o código de *spill*. Assim, o desenvolvedor do compilador tem que escolher entre as duas possíveis alternativas:

- Sempre reservar a quantidade necessária de registradores para executar o código de *spill* e em alguns casos ter que gerar código de *spill* mesmo que a quantidade de registradores necessária não seja maior que o número de registradores reservados.
- Nunca reservar registradores para *spill* e alguns casos gastar mais poder computacional para voltar atrás na alocação até ter a quantidade suficiente de registradores livres para poder gerar o código de *spill*.

Em um compilador real, ainda é necessário tratar as idiosincrasias de certas máquinas, como, por exemplo, instruções que esperam argumentos em certos registradores ou tipos que

ocupam mais (ou menos) de um registrador. Os tipos complexos podem ser incorporados no grafo de interferência, adicionando arestas extras para representar a necessidade de mais registradores. Já o problema da convenção de chamada pode ser resolvido na geração de código. O compilador pode gerar pequenos *live ranges* com cópias para satisfazer essas restrições.

3 TRABALHOS CORRELATOS

Os trabalhos correlatos encontrados podem ser divididos em duas categorias: os que trabalham apenas com a coloração do grafo de interferência e os que modelam o problema da alocação como um todo, levando em consideração que o grafo de interferência muda quando algum registrador virtual é colocado em memória.

Apesar da segunda classe de trabalhos ser um pouco mais complexa, o problema geral, que é a alocação de registradores, ainda está presente em ambas as classes e é resolvido com a mesma técnica: algoritmos evolucionários.

3.1 ALGORITMOS EVOLUCIONÁRIOS

Algoritmos evolucionários são uma classe de algoritmos que utilizam ideias de evolução simulada para adaptar soluções iniciais. Nos algoritmos evolucionários, seguindo a ideia de evolução, algumas soluções iniciais são definidas e são constantemente modificadas para melhorar a sua qualidade (evolução natural). As soluções com pouca qualidade são descartadas (JONES, 2008). Os algoritmos evolutivos são comumente usados para resolver problemas difíceis como otimização numérica.

Nos trabalhos estudados, foram utilizadas uma série de abordagens diferentes, como Evolução Diferencial (FISTER; BREST, 2011), Otimização por Enxame de Partículas (REBOLLO-RUIZ; GRANA, 2011), Otimização por Colônia de Formigas (LINTZMAYER; MULATI; SILVA, 2011) e Otimização por Enxame de Abelhas (DORRIGIV; MARKIB, 2012).

3.2 ESCOLHA DE IMPLEMENTAÇÃO

O algoritmo de Evolução Diferencial proposto por (FISTER; BREST, 2011) gerou resultados satisfatórios, porém ele foi construído para a 3-coloração de um grafo, ou seja, a coloração de um grafo utilizando apenas 3 cores. Como na alocação de registradores o número de registradores disponíveis é geralmente muito maior que 3, o algoritmo necessitaria de muitas alterações e elas poderiam acarretar em perdas das características originais do algoritmo, essa opção foi descartada.

A proposta de algoritmo de otimização por Colônia de Formigas de (LINTZMAYER; MULATI; SILVA, 2011) também apresentou resultados interessantes. Esse algoritmo modela toda a alocação de registradores, e a sua inviabilidade está na necessidade de fazer a simplificação do grafo de alocação múltiplas vezes, necessitando muitas mudanças na estrutura da

ferramenta escolhida.

As duas propostas restantes são bastante semelhantes e tratam somente do problema da coloração de grafos. O algoritmo de (REBOLLO-RUIZ; GRANA, 2011) usa a ideia de partículas afetadas por campos gravitacionais, onde cada partícula é um nodo do grafo e os campos gravitacionais são os registradores físicos. Já a proposta de (DORRIGIV; MARKIB, 2012) trabalha com um algoritmo de Otimização por Colônia de Abelhas, onde cada fonte de comida é uma sequência de nodos para colorir.

Entre as duas propostas, foi escolhida a de (DORRIGIV; MARKIB, 2012) pois é mais direto adaptar o algoritmo para fazer *spill*.

4 ALOCAÇÃO DE REGISTRADORES COM ALGORITMOS EVOLUCIONÁRIOS

O algoritmo proposto para a alocação de registradores segue as linhas do algoritmo de Otimização por Colônia de Abelhas de (DORRIGIV; MARKIB, 2012). Porém, antes de entrar nas especificidades e adaptações feitas será feita uma descrição do algoritmo original.

4.1 OTIMIZAÇÃO POR COLÔNIA DE ABELHAS

No algoritmo original de otimização por colônia de abelhas, existe três tipos de abelhas: trabalhadoras, observadoras e batedoras. Uma abelha esperando para escolher uma fonte de comida é chamada de observadora. Uma abelha em um fonte de comida que já foi visitada é chamada de trabalhadora e uma abelha fazendo uma busca aleatória por fontes de comida é chamada de batedora. Para cada fonte de comida ativa, existe somente uma abelha trabalhadora nela e essa abelha se torna temporariamente batedora quando sua fonte se esgota.

O algoritmo pode ser sumarizado em poucos passos:

- Inicia parâmetros

- **Enquanto parâmetros não são atingidos**
 - Cada abelha trabalhadora trabalha em uma fonte de comida e mede sua quantidade de néctar
 - Abelhas observadores procuram fontes de comida próximas aonde as abelhas trabalhadores estão
 - Se uma fonte de comida é esgotada, uma abelha batedora é criada para achar uma nova fonte de comida

Cada iteração do algoritmo consiste em três passos: enviar abelhas trabalhadoras para fontes de comidas já conhecidas e medir as suas quantidades de nectar; as abelhas observadoras vão para uma fonte de comida e tentam achar alguma fonte próxima que seja melhor; e finalmente criar uma abelha batedora para procurar novas fontes de comida aleatoriamente.

O algoritmo proposto por (DORRIGIV; MARKIB, 2012) implementa todas as partes do algoritmo geral, porém são necessárias algumas definições para a implementação.

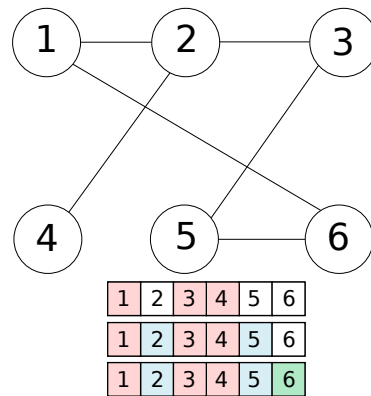


Figura 3 – Coloração a partir de uma sequência

4.2 COLORAÇÃO DE GRAFOS COM OTIMIZAÇÃO POR COLÔNIA DE ABELHAS

O problema de coloração de grafos é um problema combinatorial, e os algoritmos de otimização de IA são comumente utilizados para problemas de otimização contínua. Assim, é necessário criar uma representação do problema para ser utilizada pelo algoritmo e obter uma coloração válida como resultado.

A solução adotada por (DORRIGIV; MARKIB, 2012) foi representar a solução do problema como uma sequência de nodos onde a ordem dos nodos representa a ordem que os nodos devem ser coloridos em uma determinada sequência fixa de cores.

Para cada cor

Para cada nodo não colorido na sequência

Se não há nodo adjacente colorido com a mesma cor
`colore(nodo, cor)`

A figura 3 ilustra um exemplo de execução do algoritmo de coloração do grafo seguindo uma sequência de nodos. No exemplo, a primeira cor é escolhida e o nodo 1 é colorido. O nodo 2 não pode ser colorido com a mesma cor pois ele é adjacente ao nodo 1. Em seguida, o nodo 3 é colorido com a cor inicial, pois não é adjacente a 1. O nodo 4 também é colorido, pois não é adjacente a 1 nem 2. Os nodos 5 e 6 não são coloridos pois estão ligados a 3 e 1 respectivamente. Como a sequência foi totalmente percorrida e ainda existem nodos não coloridos, uma nova cor é escolhida. O primeiro nodo não colorido na sequência é 2, que pode ser colorido. O nodo 5 também é colorido, pois não é adjacente a 2. E finalmente, o nodo 6 não pode ser colorido, pois é adjacente ao nodo 5, por isso ele recebe uma nova cor. Ao final do algoritmo, todo o grafo foi colorido e foi necessário um número total de 3 cores.

Como demonstrado no artigo, a abordagem de colorir o grafo à partir de uma sequência de nodos consegue representar todas as colorações possíveis, por isso também é possível identificar

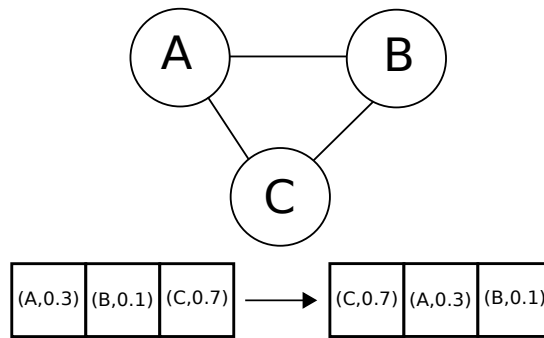


Figura 4 – Geração de uma sequência aleatória de nodos

uma coloração com o número cromático, desde que se teste todas as sequências.

A função de avaliação da qualidade de uma sequência é dada pelo número de cores necessárias para colorir o grafo naquela sequência de nodos. Assim, só falta definir como é feita a busca no espaço de todas as sequências.

A busca por outras sequências é feita em um espaço de n -dimensional de números reais, onde n é o número de nodos no grafo. Cada nodo é associado à uma dimensão, formando um par (g, v) onde g é um nodo do grafo e v é o valor de sua dimensão.

Dada uma sequência de pares (g, v) , a ordem de nodos para coloração é dada pela ordem dos nodos depois que os pares forem ordenados em relação aos valores de v .

A figura 4 demonstra como uma sequência aleatória pode ser gerada para um grafo. A cada nodo é associado um valor, que é o segundo valor do par ordenado, esse valor é então usado como critério para ordenação da sequência.

Com isso, o mapeamento de otimização contínua para um problema combinatorial está completa, restando somente a estratégia de achar sequências próximas à uma determinada sequência. O trabalho de (DORRIGIV; MARKIB, 2012) propôs três estratégias diferentes, aumentando em complexidade e na diferença entre sequências geradas.

4.2.1 Obtenção de Sequências Vizinhas

Estratégia I: A primeira estratégia proposta é a mais simples sob um ponto de vista computacional e que produz a menor diferença entre sequências. Para uma sequência atual (SA) ela pode ser descrita com os seguintes passos:

- Obtenha uma nova sequência aleatória NS .
- Escolha uma dimensão i arbitrária.
- Atualize a dimensão i na sequência a atual seguindo a fórmula:

$$SA_i = SA_i + |SA_i - NS_i| * Aleatorio[-1, 1]$$

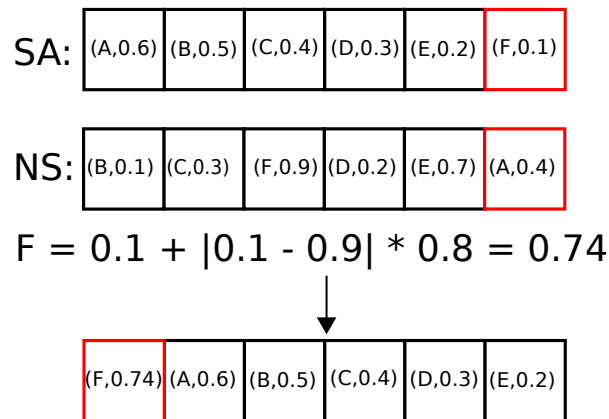


Figura 5 – Aplicação da Estratégia I

- Reordene a sequência com base nos valores.

A figura 5 demonstra este algoritmo sendo aplicado. Nela são representadas as duas sequências *NS* e *SA*. Primeiro, a sexta dimensão é escolhida e seu valor em *SA* é atualizado conforme a fórmula. Por último, a sequência é reordenada seguindo o critério dos valores de cada nodo. Essa abordagem modifica somente um nodo, porém depois que a sequência é reordenada, alguns nodos são relocados devido à mudança dos valores, como na figura 5.

Estratégia II: A segunda estratégia move um conjunto de nodos ao invés de somente um. Essa estratégia tem como objetivo diversificar mais o espaço amostral. Seu algoritmo para uma sequencial atual *SA* pode ser descrito da seguinte forma:

- Obtenha uma nova sequência aleatória *NS*.
- Escolha um nodo aleatório *N*, cujo valor é *VN*, em *SA*
- Determine o valor *NVN* de *N* em *NS*
- Atualize o valor de *NA* em *SA* para:

$$VN = VN + |VN - NVN| * Aleatorio[-1, 1]$$
- Todos os nodos entre as localizações de *N* em *SA* e *NS* têm seus valores atualizados para um número aleatório

A figura 6 mostra a execução da **Estratégia II** em uma sequência. Primeiro é selecionado o nodo *E* em *SA* e é marcado a sua posição em *NS*. Depois, o nodo *E* tem a seu valor atualizado conforme à fórmula. Após, todos os nodos entre a posição de *E* em *SA* e a posição de *E* em *NS* têm seus valores atualizados para valores aleatórios. Por fim, a sequência é ordenada segundo os valores.

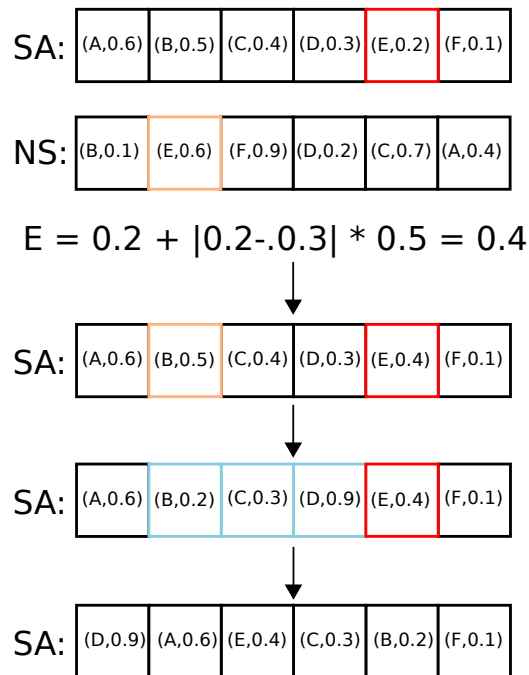


Figura 6 – Aplicação da Estratégia II

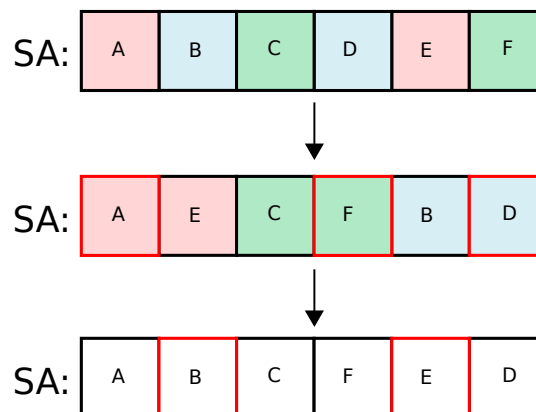


Figura 7 – Aplicação da Estratégia III

Observando o resultado obtido na figura 6 é possível notar a grande diferença entre a sequência original e a final.

Estratégia III: A terceira, e última, estratégia proposta trabalha com a ideia de classificar os nodos de acordo com a cor que lhe foi atribuída. Um nodo de cada classe é mantido em posição, os outros são permutados entre classes. O seu algoritmo pode ser resumido em:

- Reordene a sequência de modo que todos os nodos com a mesma cor estejam em ordem
- Escolha um nodo arbitrário em cada classe de cores
- Troque de posição os nodos não escolhidos com nodos pertencentes à outras classes

A figura 7 ilustra os passos da **Estratégia III**. Primeiro, diferentemente das estratégias propostas, a entrada é uma sequência de nodos já coloridos. Após isso, os nodos são agrupados

de acordo com suas classes e um nodo de cada classe é selecionado para não ser movido. Por fim, os nodos que não foram selecionados são trocados de posição com nodos de outras classes.

Esta abordagem, assim como a **Estratégia II** modifica bastante a sequência original, por isso, ela encontra sequências mais “distantes” da sequência original.

5 FERRAMENTAS

Neste trabalho foi estabelecido que a implementação seria feita em algum compilador que seja livre, de código e que seja amplamente utilizado por aplicações. A motivação para este requisito é observar como é a implementação de algo teórico em uma aplicação do “mundo real”, pois muitos artigos mostram resultados aplicados à sistemas próprios.

Atualmente, existem apenas dois compiladores que satisfazem estes requisitos, eles são o GCC e o LLVM. O GCC é o compilador criado para o projeto GNU (GCC, 2013), porém atualmente ele é o compilador mais utilizado em sistemas GNU/Linux. O LLVM é um compilador criado pela universidade de Illinois que tinha como propósito original ser uma infraestrutura para pesquisas, porém em 2005 ele começou a ser usado em todas as ferramentas de desenvolvimento da Apple.

O GCC hoje possui *frontends* para diversas linguagens, como C, C++, Objective C, Java, etc. Além disso existem *backends* escritos para as mais diversas arquiteturas, como ARM, MIPS, x86 e amd64. Seu código é escrito na maior parte na linguagem C e após investigação não foram encontradas boas fontes de documentação para as decisões de *design*. Baseado principalmente na qualidade da documentação do GCC, foi adotado o LLVM como plataforma de implementação.

5.1 LLVM

O LLVM, é um conjunto de ferramentas para construção de compiladores e ferramentas que trabalham com código, como máquinas virtuais e ambientes controlados de execução. Ele, assim como outros compiladores, é dividido em duas partes o *frontend* e o *backend*. Atualmente existem *frontends* para diversas linguagens, desde as mais tradicionais como C, C++ e Objective C, e até linguagens dinâmicas como Python e Ruby. E *backends* para várias arquiteturas, como x86, amd64, ARM, MIPS e também a linguagem C (LLVM, 2013c).

Devido à arquitetura modular do *LLVM* todas as suas fases são separadas em programas diferentes. Por exemplo, o *frontend* para a linguagem C é o *clang*. A saída do *frontend* é uma representação intermediária do LLVM chamada de *bitcode*. O *bitcode* gerado pelo *frontend* é então opcionalmente passado para o módulo otimizador *opt*, que então finalmente é passado para o *backend llc*.

6 IMPLEMENTAÇÃO

6.1 IMPLEMENTAÇÃO NO LLVM

Na arquitetura do LLVM a alocação de registradores é um passo obrigatório para a geração de código. Ela é feita pelo *backend*, porém, em uma parte inicial, onde só são executados passos independentes de arquitetura alvo. Apesar de os dados necessários para a alocação de registradores estarem intimamente ligados à arquitetura alvo, as dependências, como número de registradores e convenções, podem ser abstraídas em interfaces comuns à todas as arquiteturas, facilitando a escrita de novos alocadores (LLVM, 2013b).

Para a implementação de um alocador de registradores, a classe *RegAllocBase* fornece algumas funções auxiliares, e também fornece um controlador de alocação. A interface de um alocador é muito simples, pois a maior parte do trabalho fica sob responsabilidade do controlador, bastando implementar o método *selectOrSplit*, que decide se um *live range* deve ser alocado ou se é necessário guardá-lo em memória. Essa função é chamada somente uma vez por *live range*.

Devido à característica iterativa do algoritmo proposto, foi necessário abandonar o controlador para poder ter maior controle sobre o processo de alocação.

O alocador de registradores tem como principais parâmetros um objeto *MachineFunction* que representa características da função que está sendo trabalhada (quais registradores virtuais são parâmetros ou retornos, etc) e também um objeto *LiveRegMatrix* que representa o grafo de interferência.

O algoritmo implementado é descrito em linhas gerais da seguinte forma:

```
repita para sempre:
```

```
  gera sequências aleatórias iniciais
```

```
  iterações = 0
```

```
  melhor_candido_spill = <nulo>
```

```
  para i = 1 até número máximo de iterações:
```

```
    para cada sequência:
```

```
      se sequência é uma alocação completa:
```

```
        retorne sequência
```

```
      senão:
```

```

candidato_spill = obtenha o live range que há interferência
                  que tem o menor custo de spill
se custo(candidato_spill) > custo(melhor_candidato_spill):
    melhor_candidato_spill = candidato_spill
atualize número de nodos alocados

```

para cada sequência:

```

se número de nodos alocados < 20% do número total de nodos:
    sequência = nova sequência aleatória
senão:
    sequência = sequencia vizinha

```

```

iterações += 1

```

gera código de spill para melhor_candidato_spill

Agora que o algoritmo e suas fundamentações foram descritos, as próximas seções tratam dos aspectos técnicos da implementação no LLVM.

6.1.1 Spill de Variáveis

A maior adaptação feita no algoritmo proposto por (DORRIGIV; MARKIB, 2012) foi devido ao fato do algoritmo proposto tratar somente de buscar a coloração do grafo e não ter ação definida para o caso de não ser possível achar uma alocação com um número máximo de cores, como acontece quando não há registradores o suficiente para uma parte do código.

Para adicionar este novo requisito, foi adicionado um número máximo de iterações para o ciclo principal do algoritmo, assim, se o número máximo de iterações for alcançado sem uma alocação ser encontrada assume-se que não é possível colorir o grafo com o número de registradores da máquina atual e é necessário fazer *spill* de um registrador.

Portanto, é preciso pensar em uma estratégia para escolher qual *live range* deve-se colocar em memória. Como o objetivo do alocador é diminuir o máximo possível o impacto dos *spills*, pode-se tentar fazer *spill* do *live range* com o menor peso possível, lembrando que o peso de um *live range* é um valor estimado da quantidade de usos, assim ele aumenta caso o *live range* esteja dentro de loops. Essa estratégia não é adequada pois ela pode gerar *spills* para *live ranges* que não interferem com muitos outros *live ranges*.

Uma outra análise simplista leva à ideia de fazer *spill* de algum *live range* que tenha mais interferências do que registradores disponíveis na arquitetura alvo, assim o grau dos nodos

adjacentes será diminuído e o grafo terá mais chances de ser colorido.

Assim, foi implementado uma combinação dos pontos fortes das duas estratégias: a cada tentativa falha de alocação é analisado quais são os *live ranges* que interferem com o *live range* que não foi possível atribuir a um registrador físico, dentre esses *live ranges* (incluindo o que não foi possível atribuir a um registrador) é selecionado o que tem o menor custo de *spill*.

É mantido um histórico de qual foi o *live range* que esteve envolvido em conflitos e que tem o menor custo de *spill* e quando o número máximo de iterações é alcançado esse *live range* é transferido para a memória e o código de salvamento e carregamento dele é gerado.

Com essa estratégia tenta-se simultaneamente diminuir o grau dos nodos do grafo e também evitar deteriorização de desempenho por muitos acessos à memória devido *spills* de valores inadequados.

6.1.2 Representação de Convenções da Máquina Alvo

Como já comentado previamente, o alocador de registradores precisa levar em conta algumas características específicas da arquitetura para qual o código será gerado. Essas características podem variar entre parâmetros serem passados em registradores designados ou valores que ocupam dois registradores (como o caso de variáveis do tipo *double* e *long long* na arquitetura *x86*).

Dentro da arquitetura do LLVM esse problema é resolvido pela classe *LiveRegMatrix*. Esta classe é um mapeamento direto para o grafo de interferência descrito na literatura, ou seja, ela fornece as funções para atribuir e desatribuir registradores virtuais à registradores físicos. A própria função de atribuição de registradores físicos tem a responsabilidade de alocar os devidos registradores físicos baseados no tipo de um registrador virtual.

Pensando em termos do grafo de interferência, uma outra operação necessária é a checagem de interferências entre nodos para uma determinada cor. Essa operação é feita pelo método *checkInterference* da classe *liveRegMatrix*, esse método recebe como parâmetros um *live range* e um registrador físico e retorna o tipo de interferência que existiria caso aquele *live range* fosse atribuído àquele registrador físico. Por isso, há quatro tipos diferentes de interferência:

- **IK_Free:** É retornado quando não há nenhum tipo de interferência.
- **IK_VirtReg:** É retornado quando há somente interferências entre *live ranges*. Ou seja, ela pode ser resolvida fazendo *spill* de algum *live range* envolvido.
- **IK_RegMask:** É retornado quando há interferência causada por código *assembly inline* inserido pelo usuário.

- **IK_RegUnit:** É retornado quando não é possível alocar o *live range* para o registrador físico devido às restrições da máquina (por exemplo, tentar alocar um parâmetro num registrador temporário, ou ainda se todos os registradores necessários para alocar a variável não estão disponíveis).

Para facilitar o trabalho do alocador de registradores, cada *backend* fornece uma implementação da classe *Order*. Esta classe tem a função de definir a ordem de teste dos registradores físicos para a alocação. Essa ordem é definida para o tipo do registrador virtual (se é um número inteiro, ponto flutuante ou endereço base de um vetor) e para a função dele na função atual (parâmetro, variável comum, ou variável de retorno).

6.1.3 Sequências Vizinhas

Para a busca de sequências vizinhas foi utilizada a **Estratégia I** descrita anteriormente. Esta estratégia tem a deficiência de não gerar sequências muito diferentes da original. Para remediar este problema, é usado um número grande de sequências no algoritmo. Também, associado ao número de sequências utilizadas foi adicionado uma nova forma de diversificação da sequência.

Esse método de diversificação consiste de anotar quantos *live ranges* foram atribuídos a registradores físicos à cada tentativa de alocação não sucedida. Assim, todas as sequências que não conseguirem alocar no mínimo uma fração dos *live ranges*, que foi estabelecido em 20%, são substituídas por novas sequências aleatórias.

Assim, tenta-se buscar um equilíbrio entre a complexidade de gerar sequências próximas e a abrangência do espaço busca.

7 RESULTADOS

Para o teste da implementação foram criados alguns casos de teste para reproduzir alguns cenários comuns em compilações. Os testes foram criados para contemplar casos em que o algoritmo é extensamente usado, quando há alta necessidade por registradores, e também casos simples onde o algoritmo não é utilizado em todo o seu potencial, o que é muito comum.

Dentre as várias possíveis linguagens para implementação dos testes, eles foram escritos na linguagem C. O motivo desta decisão foi o maior controle sobre o código gerado, pois em linguagens de alto nível é maior o nível das abstrações, que acabam gerando mais código, e a representação do código em C é muito semelhante ao *assembly* gerado.

7.1 ALGORITMOS DE ALOCAÇÃO DE REGISTRADORES DO LLVM

O LLVM já oferece uma infraestrutura para testes de alocadores de registradores, por isso, já existem alguns algoritmos diferentes para a alocação (LLVM, 2013a). Os algoritmos são descritos à seguir:

7.1.1 Basic

Este algoritmo visita os *live ranges* em ordem decrescente de custo de *spill*. Sempre que não é possível alocar um registrador físico para um *live range*, é feito o *spill* e os novos *live ranges* criados pelo código de *spill* são inseridos na lista de nodos à visitar com um peso infinito. Este algoritmo é bem simples e tem como objetivo ser usado como fator de comparação entre outros alocadores.

7.1.2 Greedy

Este algoritmo tenta circundar o fato de que pequenos *live ranges* geralmente apresentam alto peso, pois são mais usados, e primeiro aloca registradores físicos aos *live ranges* maiores. Assim, ao invés dos *live ranges* grandes competirem por espaço entre os pequenos, os *live ranges* pequenos tentam se encaixar entre os grandes. Quando não há mais registradores físicos disponíveis para algum *live range*, é verificado se existe algum *live range* já alocado que apresenta um custo de *spill* menor. Se existir, ele é desatribuído e o *live range* antigo é alocado. Quando não há mais nada a se fazer, o algoritmo tenta quebrar *live ranges* em pedaços menores, para tentar encaixar as partes em algum registrador.

Algoritmo	Tempo de compilação (s)	Número de spills
Basic	0.0001	0
Greedy	0.0003	0

Tabela 1 – Resultados para a soma de parâmetros

7.2 CASOS DE TESTE

Os testes foram realizados para cada um dos algoritmos mencionados e para o algoritmo implementado. No algoritmo implementado, o número de sequências utilizado e o número de máximo de iterações antes de realizar um *spill* foram parametrizados e os resultados foram obtidos variando os parâmetros.

7.2.1 Parâmetros de comparação

Como parâmetros para comparação entre os alocadores existentes e o implementado foram utilizados dois parâmetros: números de *spill* gerados e tempo de compilação. O número de *spills* gerados é interessante pois permite observar o aumento e diminuição do uso da memória em cada algoritmo. Já o tempo de compilação é importante para verificar a viabilidade do algoritmo implementado, pois devido à característica evolucionária do algoritmo implementado, ele aumenta proporcionalmente ao número de iterações e ao tamanho da população. O tempo de execução do código gerado não foi levado em consideração, pois devido à fatores externos de difícil controle, como a memória *cache* do processador, os resultados mostraram-se muito parecidos.

7.2.2 Função Simples

Esse caso representa uma função simples, onde é baixa demanda por registradores e foi implementado como uma função que soma dois números passados como parâmetro. Em programas reais, esse caso pode ser encontrado em funções aritméticas simples e também em métodos *set* e *get*.

A motivação para este caso de teste é observar a diferença do tempo de compilação entre os algoritmos originais e o proposto.

Como explicitado na tabela 1, ambos os alocadores pré existentes tiveram resultados parecidos, com diferença de apenas 12 milissegundos no tempo de compilação e ambos não geraram *spill*.

Os resultados para o alocador implementado, na tabela 2, também foram parecidos com

Sequências \ Iterações	1	2	3	4	5	10
1	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
2	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
5	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
10	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
20	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001

Tabela 2 – Tempos de compilação (em segundos) para a soma de parâmetros com o alocador implementado

Algoritmo	Tempo de compilação (s)	Número de spills
Basic	0.0007	4
Greedy	0.0040	5

Tabela 3 – Resultados para o cálculo de determinante

os dos alocadores do LLVM. Em todos os casos o tempo de compilação foi de 6 milissegundos ou menos. Este bom resultado em todas as configurações possíveis é devido à simplicidade do problema, pois poucas (ou nenhuma) sequências possíveis gerarão interferências e o algoritmo retornará sempre nas sequências iniciais. Como nenhuma combinação de limite de iterações e número de sequências gerou algum *spill*, sua tabela foi omitida.

7.2.3 Função Aritmética de Média Complexidade

Esse caso representa uma função onde há muitos valores e a dependência entre eles é grande. Ele foi implementado como o cálculo de o determinante de uma matriz 3x3 sem utilizar loops. Todos os elementos da matriz são carregados para variáveis e o resultado é obtido à partir destes valores. Apesar de imprático, um código com um número parecido de dependências pode ser criado à partir de fórmulas matemáticas complexas.

Nesse caso, é notável a diferença no tempo de compilação entre o algoritmo *Basic* e o

Sequências \ Iterações	1	2	3	4	5	10
1	0.0017	0.0027	0.0039	0.0050	0.0065	0.0146
2	0.0027	0.0051	0.0080	0.0113	0.0150	0.0393
5	0.0069	0.0161	0.0283	0.0409	0.0591	0.1763
10	0.0165	0.0448	0.0843	0.1326	0.2037	0.6218
20	0.0490	0.1471	0.2866	0.4860	0.7299	2.3700

Tabela 4 – Tempos de compilação (em segundos) para o cálculo de determinante com o alocador implementado

Sequências \ Iterações	1	2	3	4	5	10
1	3.8300	3.6400	3.6100	3.4200	3.4100	3.1500
2	3.8600	3.6400	3.5900	3.4700	3.4000	3.1800
5	3.8700	3.7000	3.6100	3.3800	3.3800	3.1100
10	3.8200	3.6200	3.4900	3.3900	3.4800	3.1000
20	3.9100	3.6700	3.5200	3.4700	3.4400	3.1300

Tabela 5 – Números médio de *spills* para o cálculo de determinante com o alocador implementado

Algoritmo	Tempo de compilação (s)	Número de <i>spills</i>
Basic	0.0068	33
Greedy	0.1775	35

Tabela 6 – Resultados para a multiplicação de matrizes

Greedy, o segundo levando mais de cinco vezes o tempo do primeiro. Neste caso específico, o algoritmo *Basic* também gerou menos *spills*.

Como esperado, o algoritmo implementado apresentou maior tempo de compilação quando o número de iterações e sequências é pequeno, como é possível ver na tabela 4. Porém, quando o número de iterações cresce, o tempo de compilação do algoritmo cresce muito, chegando a levar 2,3 segundos quando se executa o algoritmo com um máximo de 20 iterações e utiliza-se 10 sequências.

Em questão de número de *spills*, conforme a tabela 5 o algoritmo implementado se comportou melhor que os prévios, inclusive para valores pequenos de número de iterações e sequências, que não deixam o algoritmo proibitivo no quesito tempo.

7.2.4 Função com Alta Demanda de Registradores

Esse caso extremo representa uma função onde há uma demanda enorme por registradores. Ele foi implementado como a multiplicação de duas matrizes 4x4, porém sem usar loops. Assim, todos os 18 valores são mantidos em variáveis, e como na multiplicação de matrizes os valores são reusados muitas vezes, essa demanda se mantém até o final da função. Esse caso dificilmente acontecerá em um programa escrito por humanos, porém, pode acontecer como resultado de alguma ferramenta de geração de código, ou ainda como resultado de otimizações agressivas, como *loop unrolling* (PATTERSON; HENNESSY, 2011).

Nesse caso, fica clara a diferença entre o algoritmo *Basic* e os outros. Como sua linha de raciocínio é sem *backtracking* e simples, seu tempo de compilação é muito menor que o algoritmo *Greedy* e o implementado, como visto nas tabelas 6 e 7.

Sequências \ Iterações	1	2	3	4	5	10
1	0.0247	0.0561	0.0971	0.1501	0.2113	0.6780
2	0.0562	0.1489	0.2834	0.4508	0.6604	2.3586
5	0.2066	0.6577	1.3687	2.3720	3.5260	13.6234
10	0.6685	2.3107	5.1520	8.7815	13.6845	53.0702
20	2.3048	8.8563	19.4664	33.2967	52.5189	206.5860

Tabela 7 – Tempos de compilação para a multiplicação de matrizes com o alocador implementado

Sequências \ Iterações	1	2	3	4	5	10
1	34.5000	34.2000	34.2000	34.0000	34.0000	34.0000
2	34.2000	34.1000	34.0000	34.0000	34.0000	34.0000
5	34.2000	34.0000	34.1000	34.0000	34.0000	34.0000
10	34.2000	34.2000	34.0000	34.0000	34.0000	34.0000
20	34.0000	34.2000	34.0000	34.0000	34.0000	34.0000

Tabela 8 – Números médios de *spills* para a multiplicação de matrizes com o alocador implementado

No algoritmo implementado, o tempo de compilação foi muito maior que os outros. Pequenos incrementos nos parâmetros fizeram o tempo de compilação aumentar muito, e com os maiores valores de parâmetros tentados, obteve-se um tempo de compilação superior a 3 minutos.

Em relação ao número de *spills*, tanto o algoritmo implementado quanto o algoritmo *Greedy* apresentaram resultados inferiores ao algoritmo *Basic*, porém o algoritmo implementado foi o que mais se aproximou dos resultados do algoritmo *Basic*.

7.3 REPRODUÇÃO DOS DADOS

Os dados utilizados para comparação dos algoritmos, assim como outros, são fornecidos pela própria estrutura do LLVM. Eles foram obtidos com o comando adicionando as opções *stats* e *time-passes* ao comando de compilação. Os resultados são a média dos valores para 100 execuções, e foram compilados para a arquitetura *x86*.

Também é necessário notar que a versão do LLVM utilizada para implementação, mesmo com os maiores níveis de otimização, não força o uso de variáveis em registradores. Assim, todas as variáveis ficam na pilha, e são sempre acessadas de lá.

Para gerar código que sempre acessa os registradores é necessário executar o otimizador (módulo *opt*) no *bitcode* gerado. Assim, a compilação é feita em 3 passos: no primeiro o código

fonte é transformada em *bitcode* LLVM através do *clang*, no segundo o *bitcode* é otimizado com o *opt* e finalmente no terceiro é gerado o código executável com o *llc*.

8 CONCLUSÕES

Este trabalho teve como objetivo a proposta, implementação e comparação de um alocador de registradores utilizando inteligência artificial com um algoritmo já existente.

Após a implementação foi possível verificar a viabilidade do alocador proposto em um compilador grande e complexo, como o LLVM. Com um tempo de compilação não muito maior que os algoritmos já existentes no compilador, para parâmetros pequenos, foi possível obter resultados próximos e em alguns casos, até melhores aos resultados dos algoritmos presentes atualmente no LLVM.

8.1 TRABALHOS FUTUROS

Possíveis trabalhos futuros podem ser feitos tanto na melhoria do algoritmo em si quanto na sua implementação.

Na parte de implementação, pode-se fazer um *profiling* e identificar quais partes do algoritmo precisam de mais atenção em relação ao desempenho. Também pode-se fazer uma implementação mais intrusiva, alterando algumas estruturas de dados do LLVM.

No algoritmo proposto também foram identificados algumas possíveis fontes de melhoria que não foram implementadas, como, por exemplo, adicionar a possibilidade de cada sequência ser evoluída independentemente das outras, ou seja, os *spills* serem inseridos de acordo com a sequência, e não global à todas. Outro trabalho possível é adicionar a ideia de dividir *live ranges* utilizada pelo algoritmo *Greedy*.

Também é possível melhorar os resultados do algoritmo implementando uma forma híbrida de geração de sequências. Pode-se utilizar as saídas dos algoritmos presentes no LLVM como sequências iniciais, assim, algumas das sequências iniciais já apresentam um resultado aproveitável e podem ser melhoradas ainda mais.

Ainda no algoritmo proposto, uma possível otimização seria a adaptação dos parâmetros de acordo com a execução do algoritmo, assim, em casos como o teste da multiplicação de matrizes, gastaria-se menos tempo na parte inicial, onde não há uma solução possível, e gastaria-se mais tempo (por executar mais iterações e usar mais sequências) conforme os *spills* ocorrem.

ANEXO A – Código do caso de teste de função simples


```
int sum(int a, int b) {  
    return a + b;  
}
```


ANEXO B – Código do caso de teste de cálculo de determinante


```
int determinant(int** matrix) {  
  
    int a = matrix[0][0];  
    int b = matrix[0][1];  
    int c = matrix[0][2];  
    int d = matrix[1][0];  
    int e = matrix[1][1];  
    int f = matrix[1][2];  
    int g = matrix[2][0];  
    int h = matrix[2][1];  
    int i = matrix[2][2];  
  
    return a*e*i - i*d*b + b*f*g - c*e*g + c*d*h - f*h*a;  
  
}
```


ANEXO C – Código do caso de teste de multiplicações de matrizes


```
void determinant(int** matrixa, int** matrixb, int** result) {  
  
    int a00 = matrixa[0][0];  
    int a01 = matrixa[0][1];  
    int a02 = matrixa[0][2];  
    int a03 = matrixa[0][3];  
    // ...  
  
    int b00 = matrixb[0][0];  
    int b01 = matrixb[0][1];  
    int b02 = matrixb[0][2];  
    int b03 = matrixb[0][3];  
    // ...  
    int sum;  
  
    sum = 0;  
    sum += a00 * b00;  
    sum += a01 * b10;  
    sum += a02 * b20;  
    sum += a03 * b30;  
    result[0][0] = sum;  
  
    // ...  
  
    sum = 0;  
    sum += a30 * b03;  
    sum += a31 * b13;  
    sum += a32 * b23;  
    sum += a33 * b33;  
    result[3][3] = sum;  
  
}
```


REFERÊNCIAS BIBLIOGRÁFICAS

- AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.
- DORRIGIV, M.; MARKIB, H. Algorithms for the graph coloring problem based on swarm intelligence. In: *Artificial Intelligence and Signal Processing (AISP), 2012 16th CSI International Symposium on*. [S.l.: s.n.], 2012. p. 473–478.
- FISTER, I.; BREST, J. Using differential evolution for the graph coloring. In: *Differential Evolution (SDE), 2011 IEEE Symposium on*. [S.l.: s.n.], 2011. p. 1–7.
- GCC. *GCC, the GNU Compiler Collection*. 2013. <<http://gcc.gnu.org>>. Acessado em 05/18/2013.
- JONES, T. *Artificial Intelligence: A Systems Approach*. 1st. ed. USA: Jones and Bartlett Publishers, Inc., 2008. ISBN 0763773379, 9780763773373.
- KENNEDY, K.; ALLEN, J. R. *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN 1-55860-286-0.
- LINTZMAYER, C.; MULATI, M.; SILVA, A. da. Register allocation with graph coloring by ant colony optimization. In: *Computer Science Society (SCCC), 2011 30th International Conference of the Chilean*. [S.l.: s.n.], 2011. p. 247–255. ISSN 1522-4902.
- LLVM. *Greedy Register Allocation in LLVM 3.0*. 2013. <<http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html>>. Acessado em 05/18/2013.
- LLVM. *LLVM Documentation*. 2013. <<http://llvm.org/docs/>>. Acessado em 05/18/2013.
- LLVM. *The LLVM Compiler Infrastructure*. 2013. <<http://llvm.org/>>. Acessado em 05/18/2013.
- MUCHNICK, S. S. *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1-55860-320-4.
- PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface*. 4th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 0123747503, 9780123747501.
- REBOLLO-RUIZ, I.; GRANA, M. Further results of gravitational swarm intelligence for graph coloring. In: *Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on*. [S.l.: s.n.], 2011. p. 183–188.
- TORCZON, L.; COOPER, K. *Engineering A Compiler*. 2nd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012088478X.