Capítulo 28

Uma Solução para o Problema de Alocação de Registradores baseada em Meta-heurísticas

Carla Négri Lintzmayer, Mauro Henrique Mulati e Anderson Faustino da Silva*

Resumo: Um alocador de registradores tem por objetivo alocar um número ilimitado de valores do programa para um número finito de registradores da máquina. Neste contexto, algoritmos baseados em coloração de grafo tem sido tradicionalmente utilizados. Contudo, a alocação gerada pode ocasionar sérios problemas, como constante acesso á hierarquia de memória, devido ao uso de heurísticas simples. Por outro lado, algoritmos baseados meta-heurísticas tendem a fornecer melhores soluções do que aqueles algoritmos que não são baseados em meta-heurísticas. Assim, este capítulo apresenta a aplicação de meta-heurísticas na resolução do problema de alocação de registradores. Os experimentos realizados demonstram que um alocador com uma fase meta-heurística é capaz de melhorar a qualidade do código gerado.

Palavras-chave: Alocação de Registradores, Coloração de Grafo, Meta-heurísticas.

Abstract: *A register allocator aims to allocate an unlimited number of program values for a finite number of machine registers. In this context, algorithms based on graph coloring has been traditionally used. However, the allocation may result serious problems such as constant access to memory hierarchy due to the use of simple heuristics. Moreover, algorithms based on metaheuristics tend to provide better solutions than those algorithms which are not based on metaheuristics. This chapter presents the application of metaheuristics in register allocation problem. The experiments show that an allocator with a metaheuristic phase is able to improve the quality of generated code.*

Keywords: *Register allocation, Graph coloring, Metaheuristics.*

1. Introdução

Alocação de registradores, uma das mais importantes otimizações aplicadas por um compilador, determina quais valores do programa (variáveis e temporários) devem ser mantidos em registradores (ou memória) durante sua execução (Muchnick, 1997).

Em uma máquina real, registradores são geralmente poucos porém rápidos para acessar (Patterson & Hennessy, 2008; Stallings, 2010), então o problema abordado aqui é como minimizar o tráfego entre registradores e a hierarquia de memória. Portanto, o desafio é representar a menor quantidade possível de valores do programa em memória.

A alocação de registradores pode ser mapeada como um problema de coloração gráfico (PCG) (Chaitin et al., 1981; Glover & Laguna, 1997). O PCG tradicional consiste em encontrar o mínimo valor de k para que um gráfico seja k-colorível. Já o k-PCG consiste em tentar colorir um grafo com k cores fixas, minimizando a quantidade de conflitos (vértices adjacentes atribuído a mesma cor). Na alocação de registradores o k-PCG tem uma ligeira variação: ele é forçado a eliminar os conflitos existentes além de colorir o gráfico com apenas k cores, devido ao fato de existirem somente k registradores.

Coloração de grafos (Muchnick, 1997) é uma abordagem altamente eficaz para alocação de registradores intraprocedural, e pode ser brevemente descrita como segue. Durante a geração de código, o compilador (Fischer et al., 2010) utiliza infinitos registradores simbólicos para manter os valores do programa e determina os valores que são bons candidatos para alocação em registradores. Inicialmente, o alocador de registradores gera um grafo de interferência, cujos vértices representam os valores do programa e os registradores reais da máquina e cujas arestas representam as interferências. Neste grafo, uma aresta (interferência) é adicionada, se dois valores estão simultaneamente vivos ou um valor não pode ser atribuído a um determinado registrador. Após, o alocador irá colorir os vértices do grafo de interferência com k cores, de modo que quaisquer dois

*Autor para contato: anderson@din.uem.br

vértices adjacentes tenham cores diferentes. E, finalmente, o alocador irá alocar cada valor ao registrador que tem a mesma cor a ele atribuído.

Em aplicações onde o tempo de compilação é uma preocupação, como sistemas de compilação dinâmica (Arnold et al., 2011; Ishizaki et al., 2003; Suganuma et al., 2004), pesquisadores tentam equilibrar o tempo necessário para a alocação de registradores e a qualidade do código resultante. Para alcançar este equilíbrio, geralmente não escolhem um algoritmo de alocação de registradores baseado em coloração de grafo, pelo fato deste ser um algoritmo complexo e possuir um alto tempo de execução. No entanto, alocadores (Poletto & Sarkar, 1999; Johansson & Sagonas, 2002; Mössenböck & Pfeiffer, 2002; Wimmer & Mössenböck, 2005) que são considerados mais rápidos do que aqueles baseados em coloração de grafo resultam em um código de qualidade inferior (Muchnick, 1997; Smith et al., 2004; Cooper & Dasgupta, 2006; Appel, 1998).

Um problema com algumas abordagens de alocação de registradores baseadas em coloração de grafo é o fato de aplicarem métodos heurísticos simples, resultando em uma má alocação de registradores. Neste caso, haverá um constante tráfego de dados entre o processador e a hierarquia de memória ocasionando uma perda de desempenho. Contudo, encontrar uma solução para o k-PCG é um problema $\mathcal{NP}$-completo (Karp, 1972). Nenhum algoritmo exato em tempo polinomial é conhecido, estimulando o uso de meta-heurísticas para encontrar boas soluções.

Este capítulo apresenta duas meta-heurísticas aplicadas ao problema de alocação de registradores intraprocedural. Desta forma, este capítulo descreve uma extensão do alocador de registradores proposto por George e Appel (George & Appel, 1996; Appel, 1998) para utilizar um algoritmo heurístico baseado em colônia de formigas ou um algoritmo heurístico híbrido evolucionário. Portanto, foram projetados dois novos alocadores de registradores interprecedurais baseados em coloração de grafo. A avaliação destes novos alocadores irá demonstrar que um alocador que utilize uma heurística mais agressiva gera um código de melhor qualidade.

O restante deste capítulo está organizado da seguinte forma. A Seção 2 descreve o alocador proposto por George e Appel. A Seção 3 apresenta os detalhes das meta-heurísticas aplicadas ao problema de alocação de registradores. A Seção 4 apresenta os resultados alcançados com as meta-heurísticas. E, finalmente, a Seção 5 apresenta as conclusões.

2. O Alocador de Registradores Proposto por George e Appel

Com base na observação de que um bom alocador de registradores baseado em coloração de grafo deve não apenas atribuir cores diferentes aos valores do programa, mas também tentar atribuir a mesma cor para temporários relacionados por cópias[1], George e Appel desenvolveram um algoritmo iterativo de alocação de registradores, daqui para frente chamado de George-Appel (George & Appel, 1996; Appel, 1998).

George-Appel itera até que não existam mais *derramamentos*[2]. Os resultados demonstraram como intercalar as reduções na coloração com heurísticas de *fundir*[3], levando a um algoritmo que é seguro e agressivo.

O pressuposto desta abordagem é que o compilador é livre para gerar novos temporários e cópias, porque quase todas as cópias serão *fundidas*.

As fases de George-Appel são como segue:

Construir: Nesta fase, o gráfico de interferência é construído por meio de análise de fluxo de dados (Muchnick, 1997) e seus vértices são categorizados como relacionados ou não a instruções de movimentação (cópia), significando que o vértice é a origem ou destino de uma movimentação.

Simplificar: George-Appel utiliza uma heurística simples para simplificar o grafo. Se o grafo G contém um vértice n com menos de k vizinhos, então G' é construído fazendo $G' = G - \{n\}$. Em seguida, se G' pode ser colorido com k cores, então G também pode. Esta fase repetidamente remove do grafo os vértices não relacionados a movimentação se eles possuirem grau baixo ($< k$) armazenando-os em uma pilha.

Fundir :Esta fase tenta encontrar movimentações no grafo reduzido obtido na fase *Simplificar* para aglutinar. Se dois temporários $T1$ e $T2$ não interferem é desejável que sejam alocados em um mesmo registrador. Esta fase elimina todas possíveis instruções de movimentação, unindo a origem e destino em um novo vértice. Se possível, esta fase também remove as instruções redundantes do programa. As fases *Simplificar* e *Fundir* são repetidas enquanto o grafo conter vértices não relacionados a movimentação ou vértices de grau baixo.

Congelar: Às vezes, nem *Simplificar* nem *Fundir* podem ser aplicadas. Neste caso, George-Appel *congela* um vértice de grau baixo relacionado a movimentação considerando-o um vértice não relacionado a

[1] Instruções de movimentação cuja origem e o destino são temporários, ou seja, uma instrução do tipo: $T1 \leftarrow T4$.
[2] Do original em inglês *spill*: valor do programa que será efetivamente representado em memória.
[3] Do original em inglês *coalescing*: ação de fundir dois vértices do grafo.

movimentação, o que potencialmente permitirá uma maior simplificação. Após isso, *Simplificar* e *Fundir* são retomadas.

Derramamento Potencial :Se o grafo, em algum momento, tiver apenas vértices de grau $\geq k$, esses são marcados para *derramamento*, pois provavelmente serão representados em memória. Mas, neste ponto, eles são apenas removidos do grafo e armazenados na pilha.

Selecionar: Esta fase remove os vértices da pilha e tenta colorí-los a medida que reconstrói o grafo original. Este processo não garante que o grafo será k-colorível. Se os vértices adjacentes já estiverem coloridos com k cores, o vértice atual, não poderá ser colorido e será um *derramamento real*. Este processo continuará até que não hajam mais vértices na pilha.

Derramamento Real: Em caso da fase *Selecionar* identificar um *derramamento real*, o programa é reescrito para buscar o vértice *derramado* na memória antes de cada utilização e armazená-lo após cada definição. Agora, o algoritmo executará uma nova iteração. Portanto, a execução do algoritmo somente termina quanto em uma iteração não existirem derramamentos.

Desta forma as fases de George-Appel estão organizadas como apresentadas na Figura 1.

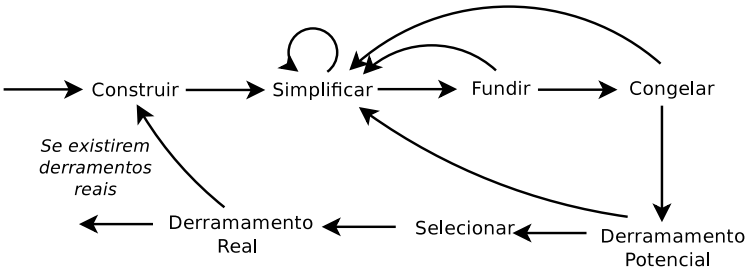

Figura 1. Fases de George-Appel (George & Appel, 1996; Appel, 1998).

3. Solução do Problema Utilizando Meta-heurísticas

Para solucionar o problema da alocação de registradores foram utilizadas duas meta-heurísticas: uma baseada em colônia de formigas e outra baseada em algoritmo heurístico híbrido evolucionário.

Cada meta-heurística deu origem a um algoritmo heurístico, o qual foi utilizado para substituir a fase *Selecionar* de George-Appel. Portanto, são propostas duas soluções para o problema da alocação de registradores baseada em coloração de grafo. Ambas soluções realizam duas modificações em George-Appel:

1. A fase *Selecionar* foi substituída pelo algoritmo *ColorAnt*$_3$*-RT* (Lintzmayer et al., 2011d) ou pelo algoritmo *HCA* (Galinier & Hao, 1999), desta forma as duas novas versões de George-Appel possuem uma fase de coloração mais agressiva do que aquela implementada pelo alocador original; e
2. A estratégia utilizada para selecionar valores para representar em memória não é baseada no grau do vértice, mas na quantidade de conflitos.

Inicialmente, as fases clássicas de George-Appel constroem o grafo de interferência e o reduz. Após, uma das meta-heurísticas reconstrói o grafo colorindo-o. E, finalmente, a nova fase *Derramamento* seleciona um vértice apropriado para ser representado em memória. Portanto, a arquitetura da nova versão do alocador é como apresentada na Figura 2.

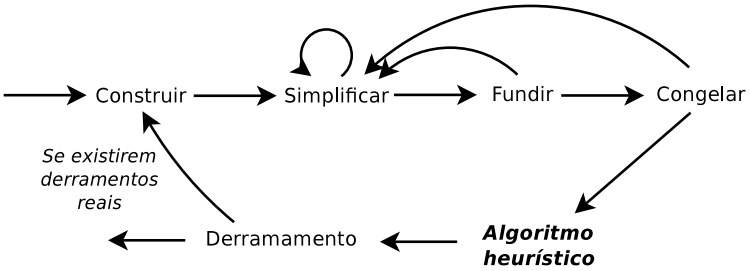

Figura 2. Fases do novo alocador de registradores.

As próximas seções descrevem os algoritmos implementados, os quais deram origem aos alocadores: CARTRA (Lintzmayer et al., 2011b), que utiliza o algoritmo *ColorAnt*$_3$*-RT*; e HCRA, que utiliza o *HCA*.

3.1 O algoritmo *colorAnt$_3$-RT*

A primeira abordagem utilizada para implementar a fase *Selecionar* usa um algoritmo heurístico baseado em colônia de formigas artificiais (Dorigo & Stützle, 2004) com busca local.

Para alcançar tal objetivo foram desenvolvidas três versões de *ColorAnt-RT*. A primeira versão se mostrou capaz de obter soluções satisfatórias, isto relativo à redução da quantidade de conflitos (Lintzmayer et al., 2011c). Porém, novas investigações demonstraram que mudanças na maneira de depositar feromônio ocasionam uma menor quantidade de conflitos, impulsionando o desenvolvimento de duas novas versões de *ColorAnt-RT* (Lintzmayer et al., 2011a,d). CARTRA utiliza a melhor versão de *ColorAnt-RT*: *ColorAnt$_3$-RT* (Lintzmayer et al., 2011d).

ColorAnt$_3$-RT utiliza como método construtivo para cada formiga, o procedimento chamado de *Ant_Fixed_K*, que é apresentado no Algoritmo 1 e foi sugerido como um método construtivo para uma versão do *ANTCOL* para a k-PCG (Costa & Hertz, 1997).

Algoritmo 1 *Ant_Fixed_k*.

```
ANT_FIXED_K(G = (V, E), k)    // V: vértices; E: arestas
1  NC = V;                    // conjunto de vértices não coloridos
2  s(i) = 0  ∀i ∈ V;          // s mapeia um vértice a uma cor
3  while NC ≠ {} do
4      escolhe um vértice v com o maior grau de
       saturação em NC;
5      escolhe uma cor c ∈ 1..k com probabilidade p de acordo
       com a Equação 1;
6      s(v) = c;
7      NC = NC \{v};
8  return s;    // retorna a solução construída
```

Em cada etapa da construção da solução, *Ant_Fixed_K* escolhe um vértice v não colorido que tenha o maior grau de saturação [4] e uma cor c para atribuir a v. A cor c é escolhida com probabilidade p, apresentada na Equação 1, que é calculada com base na trilha de feromônio τ, apresentada na Equação 2, e com base na informação heurística η, apresentada na Equação 3.

$$p(s,v,c) = \frac{\tau(s,v,c)^\alpha \cdot \eta(s,v,c)^\beta}{\sum_{i \in \{1,\ldots,k\}} \tau(s,v,i)^\alpha \cdot \eta(s,v,i)^\beta} \tag{1}$$

onde α e β são parâmetros do algoritmo e controlam a influência dos valores a eles associados na equação, e

$$\tau(s,v,c) = \begin{cases} 1 & \text{se } C_c(s) = \{\} \\ \dfrac{\sum_{u \in C_c(s)} P_{uv}}{|C_c(s)|} & \text{caso contrário} \end{cases} \tag{2}$$

$$\eta(s,v,c) = \frac{1}{|N_{C_c(s)}(v)|} \tag{3}$$

onde P_{uv} é a trilha de feromônio entre os vértices u e v. $C_c(s)$ é a classe de cor c da solução s, isto é, o conjunto de vértices já coloridos, com c nesta solução e $N_{C_c(s)}(v)$ são os vértices $x \in C_c(s)$ adjacentes a v na solução s.

A trilha de feromônio, armazenada na matriz $P_{|V|\times|V|}$, é inicializada com "1" para cada aresta entre vértices não adjacentes e com "0" para cada aresta entre vértices adjacentes. A sua atualização implica na persistência da trilha atual por um fator ρ, o que significa que $1-\rho$ é a taxa de evaporação utilizando a experiência obtida pelas formigas. Arestas entre pares de vértices não adjacentes são reforçados quando recebem a mesma cor. A evaporação é apresentada na Equação 4 e a forma geral de depósito de feromônio é apresentada na Equação 5.

$$P_{uv} = \rho P_{uv} \quad \forall u, v \in V \tag{4}$$

[4] Grau de saturação é o número de cores diferentes que já foram atribuídas aos vértices adjacentes de um vértice.

$$P_{uv} = P_{uv} + \frac{1}{f(s)} \quad \forall u, v \in C_c(s) \mid (u, v) \notin E, c = 1..k \tag{5}$$

onde $C_c(s)$ é o conjunto de vértices coloridos com c na solução s e f é a função objetivo, que retorna o número de arestas conflitantes da solução.

O reforço da trilha de feromônio é como segue. A melhor formiga do ciclo (s') e a melhor formiga da solução (s^*) intercambiavelmente reforçam a trilha de feromônio. Inicialmente, s' reforça mais frequentemente que s^*. A troca gradual nesta frequência é feita com base no número máximo de ciclos do algoritmo: a cada intervalo de um número fixo de ciclos, a quantidade de ciclos em que s^* reforça a trilha (em vez de s') é aumentado em uma unidade. Nas versões anteriores esse processo possui diferenças: em *ColorAnt*$_1$*-RT*, além das soluções de todas as formiga do ciclo serem utilizadas para atualizar a trilha de feromônio, s' e s^* também depositam feromônio; e em *ColorAnt*$_2$*-RT*, apenas s' e s^* são utilizadas para atualizar a trilha de feromônio, porém não intercambiavelmente. *ColorAnt*$_3$*-RT* é apresentado no Algoritmo 2.

Algoritmo 2 *ColorAnt*$_3$*-RT*.

ColorAnt$_3$-RT($G = (V, E)$, k) // V: vértices; E: arestas

```
1  P_uv = 1  ∀(u,v) ∉ E;
2  P_uv = 0  ∀(u,v) ∈ E;
3  f* = ∞;        // melhor valor da função objetivo até o momento
4  while cycle < max_cycles and time < max_time
   and f* ≠ 0 do
5      f' = ∞;        // melhor valor da função em um ciclo
6      for a = 1 to nants do
7          s = Ant_Fixed_k(G, k);
8          s = React_Tabucol(G, k, s);
9          if f(s) == 0 or f(s) < f' then
10             {s' = s; f' = f(s');}
11     if f' < f* then
12         {s* = s'; f* = f(s*);}
13     P_uv = ρP_uv   ∀u,v ∈ V; (Equação 4)
14     if cycle mod √max_cycles == 0 then
15         phero_counter = cycle ÷ √max_cycles;
16     if phero_counter > 0 then
17         P_uv = P_uv + 1/f(s*)
               ∀u,v ∈ C_c(s*) | (u,v) ∉ E, c = 1..k; (Equação 5)
18     else
19         P_uv = P_uv + 1/f(s')
               ∀u,v ∈ C_c(s') | (u,v) ∉ E, c = 1..k; (Equação 5)
20     phero_counter = phero_counter−1;
21     cycle = cycle+1;
```

ColorAnt$_3$*-RT* utiliza um método de busca local para melhorar a qualidade dos resultados de sua solução: a busca tabu reativa *React-Tabucol* (RT) (Blöchliger & Zufferey, 2008) que é aplicada a todas as formigas em cada ciclo. Em *ColorAnt*$_1$*-RT* e *ColorAnt*$_2$*-RT* a busca local é aplicada apenas na melhor formiga do ciclo. A busca tabu reativa será detalhada em uma seção posterior

3.2 O algoritmo *HCA*

A segunda abordagem utilizada para implementar a fase *Selecionar* usa um algoritmo heurístico híbrido evolucionário (*Hybrid Coloring Algorithm* – HCA) baseado em busca local e um operador de *crossover* altamente especializado (Galinier & Hao, 1999).

Normalmente, um algoritmo híbrido evolucionário começa com uma *população* e repete um processo iterativo por um certo número de *gerações*: em duas configurações dessa população (*pais*) o operador de *crossover* é aplicado para gerar uma nova configuração (*filho*), na qual a busca local será aplicada de forma a melhorá-la antes que ela seja devolvida à população (ao invés de haver uma mutação).

O *HCA* é como apresentado no Algoritmo 3. Este trabalha com uma população fixa de tamanho p e funciona conforme descrito a seguir. Uma população inicial é construída de acordo com o Algoritmo 4. A cada geração, duas configurações pais são escolhidas aleatoriamente pelo método *Escolher_Pais* e a elas são

aplicado o operador de *crossover*. A configuração filha gerada é melhorada pela aplicação de uma busca local, a $Tabucol_{HCA}$, por um número fixo de L iterações antes que ela seja incluída novamente na população, substituindo o pior dos pais que a geraram.

Algoritmo 3 *HCA*, adaptado de Galinier & Hao (1999).

```
HCA(G = (V, E), k, p)
 1  P = POPULACAO_INICIAL(G, p);
 2  while não atingiu critérios de parada do
 3      (s1,s2) = ESCOLHER_PAIS(P);
 4      s = CROSSOVER(s1, s2);
 5      s = TABUCOL_HCA(s, L);
 6      if f(s1) > f(s2) then    // o pior dos pais é substituído
 7          P = (P\{s1}) ∪ s;
 8      else
 9          P = (P\{s2}) ∪ s;
10  encontrar melhor solução s* da população final P;
11  return s*;
```

Galinier & Hao (1999) desenvolveram uma nova classe de operadores de *crossover*, e utilizaram o *Greedy Partition Crossover* (GPX) no HCA. Este operador funciona construindo sucessivamente as k classes de cores do filho. A cada cor c, um dos pais é escolhido (alternadamente) e neste é escolhida a classe de maior número de vértices para se tornar a classe C_c do filho. Os vértices dessa classe são removidos de ambos os pais e, ao final, os vértices ainda não coloridos recebem cores aleatoriamente. O operador de *crossover* GPX é apresentado no Algoritmo 5, no qual C_i^A representa a classe de cores C_i do pai A.

A diversidade da população é calculada como sendo a média das distâncias entre todos os indivíduos. A distância entre duas soluções é o número de modificações que devem ser feitas em uma para que ela fique igual à outra. Por exemplo, dado um grafo com 3 vértices, seja uma solução $s_1 = \{(0,1),(1,2),(2,1)\}$ (ou seja, o vértice 0 está colorido com a cor 1, o vértice 1 está colorido com a cor 2 e o 2 está colorido com a cor 1) e outra solução $s_2 = \{(0,1),(1,2),(2,2)\}$, a distância entre s_1 e s_2 é 1, pois basta trocar a cor do vértice 2.

3.3 A busca local

A busca local utilizada por *ColorAnt*$_3$*-RT* e *HCA* é um algoritmo do tipo busca *tabu* (*Tabucol*) que foi criado inicialmente por Hertz & Werra (1987) e vem sendo utilizado em vários trabalhos por ser o algoritmo mais simples, rápido e eficiente entre os melhores procedimentos de busca local (Blöchliger & Zufferey, 2008). Galinier & Hao (1999) apresentaram uma versão melhorada deste algoritmo, e é esta versão, $Tabucol_{HCA}$, que é utilizada por *HCA*.

A busca local *Tabucol* é como descrita a seguir. Dados a função objetivo f que retorna o número de arestas conflitantes, um espaço de soluções S onde cada solução é formada por k classes de cores e todos os vértices estão coloridos (provavelmente com arestas conflitantes) e uma solução inicial $s_0 \in S$, f deve ser minimizada sobre S. A cada iteração, a melhor solução vizinha é escolhida para substituir a solução atual. Uma solução vizinha é obtida movendo um vértice v de sua classe de cor atual ($C_{s(v)}$) para uma classe nova C_c, esse *movimento* é representado por (v,c). O vértice v deve ser conflitante com pelo menos um vértice que pertença à sua classe. Quando o movimento (v,c) ocorre, o par $(v,s(v))$ é classificado como *tabu* pelas próximas tl iterações, garantindo que v não volte a pertencer à classe de cores $C_{s(v)}$ neste período.

A busca gera então uma sequência $s_1, s_2, \ldots$ de soluções em S, na qual s_{i+1} é vizinha de s_i e deve ser gerada por um movimento não-*tabu* e possuir o menor número de conflitos entre as possíveis soluções vizinhas de s_i, a não ser que ela leve a um valor de função objetivo melhor do que o melhor encontrado até o momento durante a busca (critério de aspiração).

O parâmetro tl é chamado de *tabu tenure*, ou, algumas vezes, de *comprimento da lista tabu*. Normalmente é utilizado um esquema *dinâmico* de *tabu tenure*, cujo valor depende da solução atual e do movimento que foi executado. O *tabu tenure dinâmico* é dado por $tl = \alpha f(s) + \text{RANDOM}(A)$, onde A e α são parâmetros passados ao algoritmo. O ajuste do *tabu tenure* depende de como a função objetivo evolui durante a busca. Três parâmetros auxiliam nessa atualização: φ (frequência), η (incremento) e δ (limiar). A cada φ iterações

Algoritmo 4 População Inicial do *HCA*.

```
POPULACAO_INICIAL(G = (V, E), p)
 1  P = {};
 2  for i = 1 to p do
 3      ncoloridos = 0;
 4      while ncoloridos < |V| do
 5          escolher vértice v não colorido de grau máximo de
 6          saturação; se possível, c recebe a cor mínima viável;
            se não, c é escolhida aleatoriamente entre 1 e k;
 7          C_c = C_c ∪ {v};
 8          ncoloridos++;
 9      s = {C_1, ..., C_k};
10      s = TABUCOL_HCA(s, L);
11      P = P ∪ s;
12  return P;
```

Algoritmo 5 *Crossover* GPX do HCA.

```
CROSSOVER(G = (V, E), s_1 = {C_1^1, ..., C_k^1}, s_2 = {C_1^2, ..., C_k^2})
1  for c = 1 to k do
2      if c mod 2 == 1 then A = 1;
3      else A = 2;
4      escolher i cujo |C_i^A| é máximo;
5      C_c = C_i^A;
6      remover vértices em C_i^A de s_1 e s_2;
7  atribuir cores aleatoriamente aos vértices de V\(C_1 ∪ ... ∪ C_k);
8  s = {C_1, ..., C_k};
9  return s;
```

determina-se Δ, que é a diferença entre os valores máximo e mínimo que a função objetivo atingiu nas últimas φ iterações. O novo valor de tl será:

$$tl = \begin{cases} tl + \eta & \text{se } \Delta \leq \delta \\ tl - 1 & \text{caso contrário} \end{cases}$$

$Tabucol_{HCA}$ agora nomeia o algoritmo com uso do *tabu tenure dinâmico*, enquanto *React-Tabucol* nomeia o algoritmo com uso do *tabu tenure reativo*, cujo valor é determinado com base no histórico de busca. Portanto, a diferença entre os dois está no mecanismo de atualização de tl. Enquanto em *React-Tabucol* tl é atualizado de acordo com o esquema tabu reativo, em $Tabucol_{HCA}$ tl é atualizado de acordo com um esquema dinâmico. *Tabucol* é apresentado no Algoritmo 6.

É importante notar que $Tabucol_{HCA}$ e *React-Tabucol* trabalham em um espaço de soluções que contém k-colorações (próprias ou impróprias). Assim, qualquer que seja a solução inicial s_0 fornecida ao algoritmo, ela deve obedecer a esta "restrição".

3.4 A fase derramamento

George e Appel demonstraram que o critério conservativo proposto por Briggs et al. (1994) poderia ser flexibilizado para permitir uma fase de fusão mais agressiva sem a introdução de representações em memória adicionais. Além disso, eles descrevem um algoritmo que preserva os vértices fundidos encontrados antes de *derramamentos* potenciais serem descobertos. CARTRA e HCRA utilizam a mesma estratégia para fundir vértices, mas utilizam uma abordagem diferente para escolher os vértices do grafo que serão representados em memória.

Em George-Appel, se não há oportunidade para *Simplificar* ou *Congelar*, o vértice será derramado. Neste caso, a fase *Derramamento Potencial* irá calcular as prioridades de derramamento para cada vértice utilizando a Equação 6.

Algoritmo 6 *Tabucol*, adaptado de Hertz & Zufferey (2006).

TABUCOL($G = (V, E)$, k, $s_0 = \{C_1, ..., C_k\}$)

```
1  s = s_0;
2  s* = s;
3  lista_tabu = {};
4  inicializar tl de acordo com o esquema escolhido;
5  while  não atingiu critérios de parada  do
6      escolher um movimento (v, c) ∉ lista_tabu
7      com o menor valor de δ(v, c);
                    // onde δ(v, c) = f(s ∪ (v, c)) − f(s)
8      s = (s ∪ (v, c))\(v, s(v));
9      determinar tl de acordo com o esquema escolhido;
10     lista_tabu = lista_tabu ∪ {(v, s(v))};   // por tl iterações
11     if  f(s) < f(s*)  then
12         s* = s;
13 return  s*;
```

$$P_n = \frac{(usos_f + defs_f) + 10 \times (usos_d + defs_d)}{grau} \tag{6}$$

onde $usos_f$ é o conjunto de temporários que o vértice utiliza fora de um laço; $defs_f$ é o conjunto de temporários que ele define fora de um laço; $usos_d$ é o conjunto de temporários que ele usa dentro de um laço; $defs_d$ é o conjunto de temporários que ele define dentro de um laço, e $grau$ é o número de arestas incidentes no vértice.

O vértice que tem a prioridade mais baixa será selecionado para ser derramado em primeiro lugar. A abordagem de George e Appel é uma aproximação otimista: o vértice removido do grafo não interfere com qualquer um dos outros vértices do grafo.

CARTRA e HCRA utilizam uma abordagem diferente para selecionar um vértice para derramamento. Uma vez que o grafo resultante dado pelo algoritmo heurístico pode ter arestas conflitantes, a fase *Derramamento* seleciona o vértice com mais frequência no conjunto de vértices conflitantes, em outras palavras, o vértice de cor c que tem o maior número de arestas incidentes conflitantes é removido do grafo e considerado como um *derramamento real*. Se houver *derramamento real*, o programa será reescrito como em George-Appel, e então uma nova iteração será realizada.

4. Resultados Experimentais

Para avaliar a qualidade dos resultados obtidos pelas meta-heurísticas utilizadas foram implementados e comparados os alocadores de registradores: George-Appel, CARTRA e HCRA. Tal implementação foi realizada em um compilador que gera código para a arquitetura Intel IA32. Além disto, os compiladores implementados foram executados em um computador Intel Xeon E5620 de 2,40 GHz, 8 GB de memória RAM executando o sistema operacional Rocks Cluster Linux.

O conjunto de programas utilizado na avaliação é composto de onze programas de SNU-RT [5] e do programa *Queens*. Para cada programa, cada alocador foi executado 10 vezes para medir o desempenho, sendo que os dados apresentados são as médias entre as 10 execuções. Os parâmetros de CARTRA foram escolhidas de uma forma relativamente arbitrária, são eles: $nants = 80$, $\alpha = 3$, $\beta = 16$, $\rho = 0.7$ e $max\_cycles = 625$. Em geral, para instâncias pequenas os melhores resultados são obtida para α menor que β. A busca tabu foi limitada por um máximo de 300 ciclos. CARTRA pára se não há melhoria na redução do número de arestas conflitantes para mais de $max\_cycles/4$. HCRA utiliza os parâmetros p fixado em 10 e L fixado em 2000 ciclos.

4.1 Derramamentos e buscas

O objetivo principal de cada alocador de registradores é minimizar a quantidade de dados representados em memória (*derramamentos*) e consequentemente a quantidade de dados que precisam ser buscados da memória (*buscas*). Como pode ser observado pelos resultados na Tabela 1, CARTRA e HCRA superam George-Appel.

[5] http://www.cprover.org/goto-cc/examples/snu.html

Tabela 1. Desempenho de CARTRA, HCRA e George-Appel. Para cada alocador duas colunas são apresentadas: a quantidade de dados representados em memória (Derr.) e a quantidade de dados que o programa necessita buscar da memória (Buscas).

Programa	CARTRA		HCRA		George-Appel	
	Derr.	**Buscas**	**Derr.**	**Buscas**	**Derr.**	**Buscas**
Binary Search	**18,5**	**19,0**	**18,3**	**19,5**	126	142
FFT	**55,1**	**91,5**	**56,1**	**84,6**	68	103
Fibonacci	**4,5**	**3,6**	5,5	**4,3**	5	5
FIR	**52,9**	**139,4**	**54,3**	**144,4**	68	128
Insert Sort	**13,2**	**32,9**	**15,6**	**35,0**	21	39
Jfdctint	94,4	186,6	98,5	193,1	**87**	**165**
LMS	**86,7**	**137,2**	**108,2**	**156,4**	136	186
Quick sort	**41,5**	**103,2**	**43,7**	**105,4**	171	277
Queens	**17,5**	**44,0**	18,0	**24,0**	18	44
Qurt	**29,4**	**40,7**	**31,3**	**44,0**	95	126
Select	**45,4**	**88,1**	**48,1**	**90,6**	191	265
Sqrt	**9,1**	**12,6**	**9,7**	**14,0**	12	19

CARTRA e HCRA tendem a representar em memória menos temporários, pelo fato de encontrarem uma melhor coloração para o grafo de interferência, de maneira que o número de arestas com conflito seja mínimo. Neste caso, CARTRA e HCRA são capazes de utilizar uma quantidade menor de registradores por função, o que minimiza o seu custo pelo fato de reduzir a quantidade de instruções de acesso a memória, instruções que tipicamente têm um maior custo quando comparadas com outras classes de instruções. Além disso, como CARTRA e HCRA tendem a derramar uma quantidade menor de temporários e utilizar uma quantidade menor de registradores, eles são capazes de encontrar mais oportunidades para fundir vértices do grafo, e assim representar dois temporários em um único registrador.

CARTRA consegue reduções de 2,78% a 85,32% na quantidade de derramamentos. Apenas para um único programa George-Appel obteve melhores resultados, a saber: `Jfdctint`. Além disso, CARTRA atinge reduções de 11,17% a 86,62% na quantidade de buscas. No entanto, para buscas, o algoritmo de George-Appel, obteve melhores resultados para `FIR` e `Jfdctint`. Em resumo, apenas para um programa CARTRA não conseguiu um desempenho superior a George-Appel.

HCRA consegue reduções de 17,50% a 85,48% na quantidade de derramamentos, sendo que para `Queens` a quantidade ficou estável e George-Appel obteve melhores resultados para `Fibonacci` e `Jfdctint`. Para dois programas HCRA não conseguiu desempenho superior a George-Appel.

Na média, a quantidade de derramamentos é 39,02, 43,11 e 83,17 para CARTRA, HCRA e George-Appel, respectivamente. Para buscas esses números são 74,90, 76,28 e 124,92. Isto demonstra que a estratégia utilizada tanto por CARTRA quanto por HCRA é a melhor abordagem para minimizar o número de *derramamentos* e de *buscas*.

Estes resultados demonstram que embora todos os programas derramem alguns temporários, o número de instruções de armazenamento é similar a quantidade de buscas, o que sugere que os vértices que foram derramados possuem poucas definições e usos. Além disto, os resultados também demonstram que utilizar um algoritmo heurístico é uma boa opção para minimizar a quantidade de dados representados em memória.

4.2 Convergência

É importante observar que os três alocadores utilizam um algoritmo iterativo, isto é, o alocador de registradores somente termina quando não existem derramamentos (ver Figura 1 – *Se existirem derramamentos reais*). Portanto, a finalização do algoritmo somente ocorre se existir a possibilidade de todos os valores representados no grafo de interferência serem armazenados em registradores, em outras palavras, se o grafo de interferência for k-colorível.

A Tabela 2 apresenta o padrão médio de convergência de cada alocador de registradores. Para cada grafo de interferência é apresentada uma lista contendo a quantidade de *derramamentos* em cada iteração do algoritmo e o tamanho da lista (indicando a quantidade de iterações necessárias para que o grafo de interferência seja k-colorível).

Os resultados demonstram que CARTRA e HCRA encontram uma coloração que elimina a quantidade de *derramamentos* em menos iterações (reconstruções) do que George-Appel. Em geral, o número de iterações necessárias por George-Apple é de até 5 vezes a quantidade necessária por CARTRA ou HCRA.

Tabela 2. Convergência.

Programa		Alocador		
Nome	**Função**	**CARTRA**	**HCRA**	**George-Appel**
Binary Search	bs	**[9,0](2)**	**[9,0](2)**	[9,3,1,1,1,2,1,1,1,0](10)
	main	**[3,0](2)**	**[3,0](2)**	[18,16,16,16,16,0](6)
FFT	sin	[5,1,0](3)	[5,2,0](3)	**[5,0](2)**
	init_w	**[5,0](2)**	**[6,1,3](3)**	[6,3,3,1,1,1,1,0](8)
	fft	**[22,0](2)**	[21,2,0](3)	[18,3,0](3)
	main	**[4,2,0](3)**	**[4,2,0](3)**	[4,2,2,2,0](5)
Fibonacci	fib	**[2,1,0](3)**	**[2,1,0](3)**	[2,1,1,0](4)
	main	**[0](1)**	**[0](1)**	**[0](1)**
FIR	sin	[6,0](2)	**[5,0](2)**	[6,0](2)
	sqrt	**[8,1,0](3)**	**[7,0](2)**	[8,1,1,0](4)
	fir_filter	**[8,1,0](3)**	**[8,2,0](3)**	[9,1,0](3)
	gaussian	**[5,0](2)**	**[5,0](2)**	[5,1,1,2,1,1,0](7)
	main	[8,0](2)	**[7,0](2)**	**[7,0](2)**
Insert Sort	main	**[8,0](2)**	**[9,0](2)**	[7,7,8,1,0](5)
Jfdctint	fdct	[33,0](2)	[39,0](2)	**[24,0](2)**
	main	[5,1,1,0](4)	[5,2,1,1,1,1,0](7)	**[6,0](2)**
LMS	sqrt	**[8,0](2)**	**[7,0](2)**	[8,1,1,0](4)
	sin	**[6,0](2)**	**[5,0](2)**	**[6,0](2)**
	gaussian	**[5,0](2)**	**[5,0](2)**	[5,1,1,2,1,1,0](7)
	lms	**[22,3,1,0](4)**	**[23,4,1,0](4)**	[18,10,9,10,5,3,1,0](8)
	main	**[17,0](2)**	**[16,0](2)**	[15,2,1,1,0](5)
Quick Sort	sort	**[15,0](2)**	**[16,2,1,0](4)**	[16,2,2,2,2,2,0](7)
	main	**[2,0](2)**	**[2,0](2)**	[3,0](2)
Queens	print	**[6,0](2)**	[6,1,0](3)	[8,0](2)
	tree	[7,0](2)	7,0](2)	**[6,0](2)**
	main	**[1,0](2)**	**[1,0](2)**	**[1,0](2)**
Qurt	sqrt	**[7,0](2)**	**[7,0](2)**	**[7,0](2)**
	qurt	**[11,2,0](3)**	**[11,4,1,0](4)**	[11,2,3,2,3,1,1,1,0](9)
	main	**[2,0](2)**	**[2,0](2)**	[10,9,9,9,9,0](5)
Select	select	**[16,0](2)**	**[16,1,1,0](4)**	[19,1,6,4,5,4,3,2,0](9)
	main	**[2,0](2)**	**[2,0](2)**	[21,20,20,20,20,0](6)
Sqrt	sqrt	**[7,0](2)**	**[7,0](2)**	**[7,0](2)**
	main	**[0](1)**	**[0](1)**	**[0](1)**

A abordagem de George-Appel para selecionar valores para serem representados em memória ocasiona uma diminuição gradual na quantidade de *derramamentos* até que esta alcance o valor zero. Por outro lado, qualquer versão que utilize uma fase meta-heurística conduz a uma convergência mais rápida.

Em geral CARTRA e HCRA não necessitam de mais de quatro iterações, enquanto George-Appel necessita em muitos casos, de pelo menos cinco iterações. Além disso, em George-Appel alguns ciclos não reduzem a quantidade de *derramamentos* resultando em mais iterações.

4.3 Tamanho do código

A Tabela 3 apresenta a quantidade média de instruções `Assembly` e o tamanho médio do código em *bytes* para cada programa.

A redução da quantidade de instruções Assembly varia entre 1,53% e 35,35% para CARTRA e 2,79% e 35,35% para HCRA. Isto ocasiona uma redução no tamanho do código entre 1,38% e 20,56% para CARTRA e HCRA. Comparando CARTRA com George-Appel é possível perceber que CARTRA não ultrapassa o desempenho de George-Appel apenas para o programa `Jfdctint`. Porém, HCRA não ultrapassa George-Appel para três programas, a saber: `Fir`, `Jfdctint` e `Queens`.

A redução do tamanho do código é importante para sistemas que utilizam microprocessadores embarcados, devido ao fato de seus componentes consistirem geralmente de recursos limitados, seja em poder computacional

Tabela 3. Tamanho do Código (IA: Instruções Assembly, TC: Tamanho de código em bytes).

Programa	CARTRA		HCRA		George-Appel	
	IA	TC	IA	TC	IA	TC
Binary Search	**523**	**4884**	**523**	**4884**	809	6148
FFT	**797**	**6276**	809	6320	840	6412
Fibonacci	**43**	**860**	**43**	**860**	47	872
FIR	**1732**	**15104**	1801	15208	1759	15192
Insert Sort	**337**	**3496**	348	3508	358	3564
Jfdctint	1525	10276	1516	10704	**1501**	**10204**
LMS	**867**	6352	898	**6336**	1000	6996
Quick sort	**1314**	**12132**	1355	12212	1676	13840
Queens	**398**	**3740**	401	3748	**398**	**3740**
Qurt	**802**	**7496**	836	7536	981	8180
Select	**1216**	**11392**	1254	11468	1618	13324
Sqrt	**108**	**1436**	113	1452	119	1472

ou em memória. CARTRA e HCRA foram capazes de alcançar este objetivo em um proporção maior que George-Appel, para a maioria dos casos aqui apresentados.

Vale ressaltar que o objetivo tradicional de um compilador é tanto gerar um código que melhore o desempenho do processador, como minimizar o tempo de compilação para um nível aceitável de desempenho do processador. Redes de Sensores Sem Fio (Akyildiz & Vuran, 2010; Ilyas & Mahgoub, 2004; Hać, 2003), por outro lado, muitas vezes exigem cuidadosa atenção para o armazenamento do programa, restrições de memória e consumo de energia. Tanto CARTRA, quanto HCRA são boas opções para alcançar estes objetivos.

4.4 Tempo de compilação

A Tabela 4 apresenta o tempo médio de compilação dos alocadores. Estes resultados demonstram que George-Appel é mais rápido do que CARTRA entre 5,43 a 606,52 vezes, mas apenas entre 1,08 e 3,91 vezes do que HCRA. Além disto, como pode ser observado neste resultados HCRA é mais rápido que CARTRA entre 39,66 e 372,36 vezes.

Tabela 4. Tempo de Compilação (segundos).

Programa	CARTRA		HCRA		George-Appel	
	Média	Desvio	Média	Desvio	Média	Desvio
Binary Search	26,059	0,322	0,657	0,037	**0,168**	**0,001**
FFT	43,351	4,461	0,657	0,037	**0,213**	**0,008**
Fibonacci	2,431	0,483	**0,087**	**0,020**	0,448	0,004
FIR	235,817	18,697	1,421	0,034	**1,317**	**0,053**
Insert Sort	22,495	7,287	0,285	0,106	**0,110**	**0,000**
Jfdctint	634,870	386,634	1,705	0,360	**1,402**	**0,045**
LMS	84,079	11,072	0,810	0,089	**0,358**	**0,002**
Quick sort	327,153	242,891	1,326	0,247	**1,180**	**0,000**
Queens	21,322	3,621	0,339	0,206	**0,112**	**0,002**
Qurt	144,351	45,332	0,674	0,059	**0,238**	**0,006**
Select	363,318	180,627	**1,327**	**0,079**	1,510	0,065
Sqrt	3,413	0,019	0,079	0,000	**0,047**	**0,008**

Tais resultados demonstram uma restrição a utilização de CARTRA. Este não é desejável para ser utilizado em um contexto onde o tempo de compilação deve ser reduzido ao máximo, por exemplo, em sistemas dinâmicos. Até mesmo HCRA não é desejável neste contexto. Por outro lado, em um sistema de compilação estática, o tempo consumido durante o processo de compilação não é necessariamente um problema. Desta forma, em compilação estática CARTRA e HCRA são ótimos candidados para gerar código de boa qualidade.

Estes resultados também demonstram a instabilidade do alocador de registradores baseado em colônia de formigas. Este, CARTRA, obteve o desvio padrão muito elevado entre os alocadores. Embora, o tempo de execução do compilador tenha uma variação para HCRA, esta variação é bem menor que CARTRA, mas ainda é superior aquela obtida por George-Appel. Isto ocorre pela natureza de CARTRA e HCRA em utilizarem um algoritmo heurístico, e não um algoritmo exato como George-Appel.

Um tempo de execução relativamente alto é geralmente um problema em algoritmos baseados em colônia de formigas. Embora estes algoritmos sejam capazes de encontrar soluções satisfatórias para muitos problema, o tempo de execução é um custo que deve ser pago. Assim, muitos pesquisadores utilizam abordagens alternativas evitando algoritmos baseados em colônia de formigas. Algoritmos evolucionários são uma boa alternativa quando o tempo de compilação é um objetivo a ser alcançado, juntamente com um código de boa qualidade.

Por fim, é importante ressaltar a capacidade de CARTRA em reduzir a quantidade de *derramamentos*, o que elimina a quantidade de ciclos de *clock* e o tamanho do código, questões que são extremamente importantes em sistemas embarcados. Embora CARTRA tenha um elevado tempo de execução, ele é capaz de alcançar vários objetivos, tais como: reduzir o tamanho do código, reduzir a quantidade de acessos à memória, e consequentemente, reduzir a quantidade de energia consumida. Contudo, quando além destes objetivos deseja-se um tempo de compilação reduzido a melhor alternativa é utilizar HCRA.

5. Conclusões

O problema de alocação de registradores visto como um problema de coloração de grafos foi inicialmente proposto por Chaitin et al. (1981), o qual foi utilizado em um compilador experimental para a máquina IBM 370. Atualmente, versões deste alocador ou alguma derivada deste são utilizadas em compiladores comerciais. Diversos trabalhos melhoraram o trabalho de Chaitin *et al.* (Bergner et al., 1997; Bernstein et al., 1989). O projeto mais bem sucedido foi desenvolvido por Briggs et al. (1994), que reprojetou o alocador de Chaitin *et al.* para adiar ao máximo as decisões de derramento.

George e Appel (George & Appel, 1996; Appel, 1998) projetaram um alocador de registradores que utiliza os passos de simplificação do projeto de Chaitin *et al.* com a estratégia de fundir vértices do grafo empregada por Briggs *et al.*. George e Appel asseguram que esta abordagem elimina mais instruções de movimentação do que a abordagem de Briggs *et al.*, além de garantir não introduzir mais derramamentos.

Um ponto negativo destes alocadores de registradores é o fato destes aplicarem um método heurístico simples para colorir o grafo, o que não garante que será encontrada uma boa coloração. Alocadores que utilizam métodos mais agressivos envolvendo probabilidade são capazes de encontrar uma coloração melhor, consequentemente gerando um código de melhor qualidade.

CARTRA e HCRA são dois alocadores de registradores que modificam o projeto de George e Appel com o objetivo de utilizar um método heurístico mais agressivo. No primeiro a coloração do grafo é baseada na meta-heurística colônia de formigas, enquanto no segundo em uma meta-heurística híbrida evolucionária.

Tanto CARTRA quanto HCRA fornecem soluções significativamente melhores que George-Appel, o qual possui um ótimo tempo de execução. Embora CARTRA possua um elevado tempo de execução, ele é capaz de encontrar soluções ligeiramente melhores que HCRA. Este último, por sua vez tem tempo de execução pouco pior embora seja competitivo com George-Appel. Por conseguinte, a utilização de CARTRA exige uma troca: qualidade da solução *versus* tempo de compilação. Enquanto que a utilização de HCRA é bastante adequada: boa qualidade de solução com um bom tempo de execução.

Referências

Akyildiz, I.F. & Vuran, M.C., *Wireless Sensor Networks*. New York, USA: J. Wiley & Sons, 2010.

Appel, A.W., *Modern Compiler Implementation in C*. Cambridge, UK: Cambridge University Press, 1998.

Arnold, M.; Fink, S.; Grove, D.; Hind, M. & Sweeney, P.F., Adaptive optimization in the Jalapeño JVM. *SIGPLAN Notices*, 46(4):65–83, 2011.

Bergner, P.; Dahl, P.; Engebretsen, D. & O'Keefe, M., Spill code minimization via interference region spilling. *SIGPLAN Notices*, 32(5):287–295, 1997.

Bernstein, D.; Golumbic, M.; Mansour, Y.; Pinter, R.; Goldin, D.; Krawczyk, H. & Nahshon, I., Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, 1989.

Blöchliger, I. & Zufferey, N., A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & Operations Research*, 35(3):960–975, 2008.

Briggs, P.; Cooper, K.D. & Torczon, L., Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, 1994.

Chaitin, G.J.; Auslander, M.A.; Chandra, A.K.; Cocke, J.; Hopkins, M.E. & Markstein, P.W., Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

Cooper, K.D. & Dasgupta, A., Tailoring graph-coloring register allocation for runtime compilation. In: *Proceedings of the International Symposium on Code Generation and Optimizatio*n. Washington, USA: IEEE Computer Society, p. 39–49, 2006.

Costa, D. & Hertz, A., Ants Can Colour Graphs. *The Journal of the Operational Research Society*, 48(3):295–305, 1997.

Dorigo, M. & Stützle, T., *Ant Colony Optimization*. Cambridge, USA: MIT Press, 2004.

Fischer, C.N.; Cytron, R.K. & LeBlanc, R.J., *Crafting a Compiler*. Reading, USA: Addison-Wesley, 2010.

Galinier, P. & Hao, J.K., Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.

George, L. & Appel, A.W., Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, 1996.

Glover, F. & Laguna, M., *Tabu Search*. Norwell, USA: Kluwer Academic Publishers, 1997.

Hać, A., *Wireless Sensor Network Designs*. New York, USA: J. Wiley & Sons, 2003.

Hertz, A. & Werra, D., Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.

Hertz, A. & Zufferey, N., A new ant algorithm for graph coloring. In: Pelta, D.A. & Krasnogor, N. (Eds.), *Proceedings of the Workshop on Nature Inspired Cooperative Strategies for Optimization*. Granada, Espanha, p. 51–60, 2006.

Ilyas, M. & Mahgoub, I. (Eds.), *Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems*. Boca Raton, USA: CRC Press, 2004.

Ishizaki, K.; Takeuchi, M.; Kawachiya, K.; Suganuma, T.; Gohda, O.; Inagaki, T.; Koseki, A.; Ogata, K.; Kawahito, M.; Yasue, T.; Ogasawara, T.; Onodera, T.; Komatsu, H. & Nakatani, T., Effectiveness of cross-platform optimizations for a Java just-in-time compiler. *SIGPLAN Notices*, 38(11):187–204, 2003.

Johansson, E. & Sagonas, K.F., Linear scan register allocation in a high-performance erlang compiler. In: *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*. London, UK: Springer-Verlag, p. 101–119, 2002.

Karp, R.M., Reducibility among combinatorial problems. In: Miller, R.E. & Thatcher, J.M. (Eds.), *Complexity of Computer Computations*. New York, USA: Plenum Press, The IBM Research Symposia Series, p. 85–103, 1972.

Lintzmayer, C.N.; Mulati, M.H. & da Silva, A.F., Algoritmo heurístico baseado em colônia de formigas artificiais colorant2 com busca local aplicado ao problema de coloração de grafo. In: *Anais do X Congresso Brasileiro de Inteligência Computacional*. Fortaleza, SP, 2011a.

Lintzmayer, C.N.; Mulati, M.H. & da Silva, A.F., Register Allocation with Graph Coloring by Ant Colony Optimization. In: *Proceedings of the XXX International Conference of the Chilean Computer Science Society*. Curicó, Chile, 2011b.

Lintzmayer, C.N.; Mulati, M.H. & da Silva, A.F., RT-ColorAnt: um algoritmo heurístico baseado em colônia de formigas artificiais com busca local para colorir grafos. In: *Anais do XLIII Simpósio Brasileiro de Pesquisa Operacional*. Ubatuba, SP, 2011c.

Lintzmayer, C.N.; Mulati, M.H. & da Silva, A.F., Toward better performance of ColorAnt ACO algorithm. In: *Proceedings of the XXX International Conference of the Chilean Computer Science Society*. Curicó, Chile, 2011d.

Mössenböck, H. & Pfeiffer, M., Linear scan register allocation in the context of SSA form and register constraints. In: *Proceedings of the International Conference on Compiler Construction*. London, UK: Springer, p. 229–246, 2002.

Muchnick, S.S., *Advanced Compiler Design and Implementation*. San Francisco, USA: Morgan Kaufmann, 1997.

Patterson, D.A. & Hennessy, J.L., *Computer Organization And Design: The Hardware/software Interface*. San Francisco, USA: Morgan Kaufmann, 2008.

Poletto, M. & Sarkar, V., Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

Smith, M.D.; Ramsey, N. & Holloway, G., A generalized algorithm for graph-coloring register allocation. *SIGPLAN Notices*, 39(6):277–288, 2004.

Stallings, W., *Computer Organization and Architecture*. Prentice Hall, 2010.

Suganuma, T.; Ogasawara, T.; Kawachiya, K.; Takeuchi, M.; Ishizaki, K.; Koseki, A.; Inagaki, T.; Yasue, T.; Kawahito, M.; Onodera, T.; Komatsu, H. & Nakatani, T., Evolution of a Java just-in-time compiler for IA-32 platforms. *IBM Journal of Research and Development*, 48(5/6):767–795, 2004.

Wimmer, C. & Mössenböck, H., Optimized iinterval splitting in a linear scan register allocator. In: *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*. New York, USA: ACM, p. 132–141, 2005.

Notas Biográficas

Carla Négri Lintzmayer é graduada em Ciência da Computação (UEM, 2011) e atualmente é discente de mestrado no programa de pós-gradução em Ciência da Computação na Universidade Estadual de Campinas.

Mauro Henrique Mulati é graduado em Informática (Universidade Estadual de Maringá – UEM, 2005) e mestre em Ciência da Computação (UEM, 2009). Atualmente é Professor Assistente do Departamento de Ciência da Computação da Universidade Estadual do Centro-Oeste – UNICENTRO.

Anderson Faustino da Silva é graduado em Ciência da Computação (Universidade Estadual do Oeste do Paraná – UNIOESTE, 1999), mestre e doutor em Engenharia de Sistemas e Computação (COPPE/UFRJ, 2003 e 2006, respectivamente). Atualmente é Professor Adjunto do Departamento de Informática da Universidade Estadual de Maringá.