Uma Estratégia Baseada na Metaheurística VNS para Encontrar Efetivas Sequências de Otimizações

João Fabrício Filho
Universidade Tecnológica Federal do Paraná Câmpus Campo Mourão
Via Rosalina Maria dos Santos, 1233 - Campo Mourão, Paraná
joaof@utfpr.edu.br

Ewerton Daniel de Lima
Departamento de Informática – Universidade Estadual de Maringá
Avenida Colombo, 5790 – Bloco C56 – 87.020-900 – Maringá – PR – Brazil
ewertondanieldelima@gmail.com

Anderson Faustino da Silva
Departamento de Informática – Universidade Estadual de Maringá
Avenida Colombo, 5790 – Bloco C56 – 87.020-900 – Maringá – PR – Brazil
anderson@din.uem.br

RESUMO

O problema de seleção de otimizações visa encontrar as melhores otimizações a serem aplicadas em um determinado programa. A busca exaustiva é uma abordagem impraticável para solucionar tal problema, devido às inúmeras possibilidades alcançáveis. Este artigo tem por objetivo mitigar o problema de seleção de otimizações utilizando a metaheurística *Variable Neighborhood Search*. Os resultados alcançados pelo algoritmo heurístico proposto, para o benchmark SPEC CPU2006, indicam que o algoritmo tem potencial para encontrar boas soluções, além de possuir um desempenho superior ao algoritmo proposto por Pan e Eigenmann - *Combined Elimination* - em diversos casos.

PALAVRAS CHAVE. Variable Neighborhood Search, Otimização de Compilador, Combined Elimination, Metaheurísticas.

ABSTRACT

The Optimization Selection Problem aims to find the best optimizations to use in a specific source code. The exhaustive search is an impracticable approach to solve this problem because are length of the search space. This article aims to mitigate the Optimization Selection Problem with an application of the *Variable Neighborhood Search* metaheuristic. The achieved results by the proposed algorithm, for the SPEC CPU2006 benchmark, indicate that it has potential to find good solutions and outperforms the algorithm proposed by Pan e Eigenmann, *Combined Elimination*.

KEYWORDS. Variable Neighborhood Search, Compiler Optimization, Combined Elimination, Metaheuristics.

1. Introdução

Compiladores que aplicam otimizações, denominados compiladores otimizadores, fornecem geralmente dezenas de otimizações. Dentre elas é possível escolher quais serão aplicadas, pois nem toda otimização modificará o código de maneira benéfica, podendo em alguns casos até ocasionar perda de desempenho.

Nesse contexto, o Problema da Seleção de Otimizações (PSO) é: *escolher as melhores otimizações e seus respectivos parâmetros para um código X, tal que X seja ótimo para um dado objetivo*.

Tal problema pode ser tratado de três formas distintas, a saber: busca por um conjunto, na qual somente se insere ou remove otimizações; busca por uma sequência sem repetição, na qual é considerada a ordem de aplicação de cada otimização; ou busca por uma sequência com repetição, na qual são consideradas tanto a ordem de aplicação quanto a repetição de otimizações.

Na tentativa de procurar soluções viáveis para o PSO, várias abordagens foram propostas na literatura. Dentre essas abordagens estão: busca exaustiva (Foleiss et al., 2011), técnica estatística (Haneda et al., 2005), eliminação iterativa (Pan e Eigenmann, 2006), algoritmos genéticos (Leather et al., 2009), aprendizagem de máquina (Cavazos et al., 2007; Park et al., 2011) e também abordagens que combinam duas ou mais dessas técnicas (Purini e Jain, 2013).

Utilizando uma abordagem diferente, o presente trabalho propõe o uso da clássica metaheurística *Variable Neighborhood Search* (`VNS`), para mitigar o PSO considerando a busca por uma sequência com repetição.

A estratégia implementada mostrou ser eficiente para encontrar boas sequências para os *benchmarks* do SPEC CPU2006 (Henning, 2006). Além disso, a estratégia proposta se mostrou mais atrativa do que a implementada por Pan e Eigenmann, *Combined Elimination* (`CE`) (Pan e Eigenmann, 2006).

O texto segue com a seguinte organização: a Seção 2 apresenta trabalhos da literatura para mitigação do PSO; a Seção 3 descreve a estratégia proposta para mitigar o PSO; a Seção 4 descreve o ambiente experimental; a Seção 5 apresenta os resultados experimentais; e por fim, a Seção 6 apresenta as conclusões e os trabalhos futuros.

2. Trabalhos Relacionados

O trabalho de Foleiss et al (Foleiss et al., 2011) utilizou busca exaustiva em algumas classes específicas de otimização, e não para cada otimização isoladamente. Essa restrição permitiu que fosse viável a busca exaustiva no problema apresentado. Diferentemente, o trabalho proposto no presente artigo avalia cada otimização individualmente. Além disso, enquanto o trabalho de Foleiss et al tem por objetivo tamanho de código, o presente trabalho tem como objetivo tempo de execução.

Técnicas estatísticas foram utilizadas no trabalho de Haneda et al (Haneda et al., 2005), no qual o teste de hipótese de Mann-Whitney serviu para verificar se determinada otimização afeta ou não o código gerado, permitindo assim decidir se é viável selecioná-la ou não. O trabalho de Haneda et al se difere do proposto neste artigo, pelo fato de este não utilizar técnicas estatísticas e sim um algoritmo heurístico de busca.

O trabalho de Pan e Eigenmann (Pan e Eigenmann, 2006) utilizou eliminação iterativa para selecionar bons conjuntos de otimizações. O objetivo foi investigar quais otimizações prejudicavam a qualidade para determinado código e então retirá-las do conjunto padrão. Enquanto o trabalho de Pan e Eigenmann apenas remove as otimizações prejudiciais, o presente trabalho propõe o uso de inserção e troca de otimizações, além da remoção. Dessa forma, o presente trabalho tenta explorar um espaço de busca maior do que aquele explorado pelo algoritmo de Pan e Eigenmann.

Abordagens genéticas foram utilizadas tanto para busca de bons conjuntos de otimização em termos de *speedup* e tamanho de código, quanto para construções de heurísticas a serem utilizadas em otimizações específicas como, por exemplo, para decidir o melhor nível de aplicação da otimização *loop unroll* (Leather et al., 2009). O algoritmo proposto, que é baseado na VNS, possui similaridade com tais trabalhos pelo fato de ambos utilizarem uma abordagem heurística.

Alguns trabalhos como os de Cavazos et al (Cavazos et al., 2007) e Park et al (Park et al., 2011) utilizam uma abordagem aleatória para construir um conjunto de treino ao qual serão aplicados algoritmos de aprendizagem de máquina capazes de prever bons conjuntos de otimizações dadas as características de um programa. O algoritmo baseado na VNS, proposto no presente artigo, pertence a outra classe de soluções para o PSO - compilação iterativa - na qual não existe uma fase de treinamento.

O trabalho de Lau et al (Lau et al., 2006) combinou aleatoriedade e técnicas estatísticas para determinar quais otimizações são boas pra uma determinada região de código no contexto das máquinas virtuais (Smith e Nair, 2005) e compilação *just-in-time*. O algoritmo heurístico proposto no presente trabalho aplica otimizações a todo o código, e não à regiões específicas e, portanto, todas as otimizações aplicadas são consideradas para o todo do programa.

Uma abordagem mais recente para o PSO (Purini e Jain, 2013) combinou pelo menos quatro técnicas diferentes para seleção de otimizações, a saber: busca aleatória, algoritmos genéticos, aprendizagem de máquina e eliminação iterativa. A diferença principal entre esta abordagem e à proposta neste artigo, esta no fato desta última não combinar diferentes técnicas.

3. Mitigação do PSO Utilizando a Metaheurística `VNS`

Variable Neighborhood Search (`VNS`) é uma metaheurística proposta em 1995 por Mladenović (Mladenović, 1995) e amplamente utilizada para resolução de problemas combinatórios e de otimização global. Basicamente, essa metaheurística efetua mudanças sistemáticas na vizinhança de uma solução com uma busca local para encontrar novas soluções para o problema. Um algoritmo que utiliza a metaheurística `VNS` segue o seguinte comportamento, em ordem:

Inicialização Definir uma solução inicial x, uma condição de parada e um conjunto de operadores que serão utilizados na busca. Operadores são métodos que modificam uma solução para buscar soluções vizinhas a ela, seguindo critérios pré-determinados. A solução inicial é o ponto de partida para a busca de novas soluções.

Repetição Repetir os seguintes passos, até que acabem os operadores definidos na inicialização:

Perturbação gerar uma solução x' vizinha de x de forma aleatória.
Busca local aplicar uma busca local com o operador atual, utilizando x' como solução inicial e encontrar um ótimo local x''.
Trocar ou não Se o ótimo local x'' for melhor que a solução x então fazer $x = x''$ e prosseguir na iteração com o mesmo operador. senão, selecione o próximo operador.

Alguns problemas combinatórios conhecidos já tiveram a aplicação da metaheurística VNS, como: problema do escalonador de processos flexível (Zhang, 2012); problema de fluxo de montagem com máquinas paralelas (Javadian et al., 2009); e o problema do caixeiro viajante (Lei-fu e Wei, 2010).

As características do PSO permitem que ele também seja abordado com a metaheurística VNS. Efetuando uma busca por sequências de otimizações, considerando-se tanto repetições quanto ordem, o uso da metaheurística VNS sugere ser uma boa abordagem para o PSO. Portanto, essa é a proposta do presente trabalho e é descrita nas subseções seguintes.

3.1. Operadores de Vizinhança

Os operadores de vizinhança norteiam a busca local. Para o PSO, é interessante explorar não só quais otimizações estarão no conjunto mas também em que ordem elas estarão – formando assim uma sequência. Além disso, é possível permitir a repetição de aplicação de otimizações. Para atingir esses objetivos, foram definidos os seguintes operadores de vizinhança:

Troca seleciona duas otimizações aleatoriamente e troca uma pela outra;
Remoção escolhe uma otimização de forma arbitrária e a remove; e
Inserção escolhe aleatoriamente uma otimização de uma tabela de otimizações O e a adiciona ao conjunto, em uma posição aleatória.

O operador de troca explora a ordem de aplicação das otimizações, enquanto os operadores de inserção e remoção exploram quais otimizações estarão na solução. Nenhuma restrição é feita na inserção quanto à qual otimização está sendo adicionada. Assim, é possível que uma mesma otimização seja adicionada mais de uma vez.

3.2. As Rotinas de Perturbação e Busca Local

A rotina de perturbação tem por objetivo explorar uma diferente região de soluções, ampliando desta forma as chances de encontrar um ótimo global.

Perturbar uma solução significa escolher um operador de forma aleatória e aplicá-lo a solução que deve ser perturbada, fornecendo desta forma uma nova solução.

A busca local tem por objetivo procurar a melhor solução em uma certa vizinhança. Isso pode ser feito explorando toda a vizinhança da solução. Entendem-se por toda vizinhança todas as soluções que podem ser geradas a partir de uma determinada solução com um determinado operador. No entanto, explorar todas essas possibilidades possui um alto custo pois envolve, em geral, a execução do programa a cada avaliação[1]. Portanto, uma

[1] Avaliar uma solução significa compilar o programa com a sequência encontrada, executá-lo e coletar seu tempo de execução.

estratégia menos dispendiosa é explorar um número limitado de soluções em cada vizinhança. Na solução proposta isso é realizado com base em uma taxa de exploração, a qual é relativa ao tamanho da melhor solução encontrada até o momento.

Seguindo tal estratégia, foram desenvolvidas três buscas locais com os operadores troca, remoção e inserção.

A busca local de troca efetua uma troca aleatória entre duas otimizações e avalia a nova solução. Se a troca for benéfica, então essa solução é utilizada na próxima iteração. A busca finaliza após Tx_{troca}[2] trocas aleatórias. Esse operador tem por objetivo melhorar a solução explorando a ordem de aplicação das otimizações.

A busca local de remoção retira uma otimização de S, gerando S'. O desempenho de S' é então avaliado e, se S' tem desempenho superior a S, uma próxima remoção será realizada em S'. Basicamente, a busca local de remoção remove otimizações enquanto houver melhoria em relação à solução anterior. Tendo o máximo de remoções possíveis definido por Tx_{remove}. Isso explora a eliminação de otimizações que afetam negativamente o desempenho.

A busca local de inserção adiciona Tx_{insere} novas otimizações à solução inicial. Tanto a posição de inserção da otimização quanto a própria otimização a ser inserida são aleatórias. Isso permite a inserção de otimizações em diferentes posições, além da repetição de aplicação de uma mesma otimização.

O uso de diferentes buscas locais amplia o espaço de busca, fornecendo novas possibilidades para encontrar boas soluções para o PSO. Além disso, a utilização de taxas para cada operador possibilita encontrar soluções em um espaço de busca mais amplo, pois, quando as tentativas não encontram uma solução melhor, as taxas alteram o comportamento do algoritmo com o objetivo de ampliar o espaço de busca e assim aumentar a convergência do algoritmo para um ótimo global.

3.3. Algoritmo `VNS` para o PSO

Tendo definidos a rotina de perturbação, os operadores de vizinhança e as buscas locais correspondentes, é possível construir um algoritmo com a metaheurística `VNS`, definindo o critério de parada e a solução inicial.

O algoritmo proposto finaliza a busca por novas soluções quando um dos seguintes critérios ocorre:

1. **Tempo de execução**: este critério limita o tempo de execução da metaheurística, determinando o tempo máximo no qual o sistema deve fornecer uma resposta. Este critério indica o tempo total de execução do sistema. Portanto, ele inclui: (1) tempo de perturbar uma solução, (2) tempo de buscar novas soluções, (3) tempo de compilar o *benchmark* com uma determinada solução e (3) tempo de executar o programa para validar a solução encontrada.
2. **Iterações**: sendo um algoritmo iterativo é importante definir a quantidade máxima de iterações.
3. **Avaliações**: dada a natureza do PSO, este critério tem por objetivo definir a quantidade máxima de compilações que um *benchmark* sofrerá.
4. **Desempenho**: este critério define um limiar máximo ao desempenho esperado.

[2] Tx_{troca} é a taxa de trocas, indicando a quantidade máxima de trocas que serão avaliadas.

Esses critérios de parada possuem o atrativo de flexibilizar o ajuste da metaheurística de acordo com as características do *benchmark* em questão.

Uma questão importante é a escolha da sequência inicial. Algoritmos heurísticos podem utilizar uma abordagem construtiva, melhorativa ou ambas.

Devido a natureza da VNS, o algoritmo proposto pode ser visto como um algoritmo melhorativo. Como o objetivo final é melhorar o desempenho do melhor conjunto padrão do compilador em questão, o ideal é que a solução de partida seja uma solução conhecida previamente. Desta forma, a solução de partida é a melhor sequência padrão do compilador em questão.

O Algoritmo 1 apresenta a metaheurística VNS proposta para o PSO.

Algorithm 1: VNS para o PSO

Input: Benchmark (B), Sequência inicial (S)
$lista\_taxas = [Tx\_troca, Tx\_remove, Tx\_insere]$
$k = 0$
$melhor\_t = tempo\_de\_execucao\_da\_sequencia\_inicial()$
$melhor\_seq = seq$
$count = 0$
$convergiu = false$
while *condição de parada não for satisfeita* **do**
  while $k < |lista\_operadores|$ **do**
    $count = count + 1$
    $S, T = perturbacao(S, B)$
    /* k = 0 troca; k = 1 remove; k = 2 insere */
    $S, T = busca\_local_k(S, T, lista\_taxas[k], B)$
    if $T < melhor\_t$ **then**
      $melhor\_t = t$
      $melhor\_seq = seq$
      $k = 0$
      $convergiu = true$
      Atribuir valores-padrão às taxas
    else
      $S = melhor\_seq$
      $k = k + 1$
  if $((count \,\%\, (|lista\_operadores| \times 2) = 0) \;\&\; NOT(convergiu))$ **then**
    Dobrar todos os valores de taxas
    $count = 0$
  if $convergiu$ **then**
    $convergiu = false$
return $melhor\_seq$

4. Ambiente Experimental

Hardware A arquitetura utilizada foi um computador com processador Core I7-2600 executando em uma frequência de 3.4GHz com cache L1 de instruções e dados de 32KB,

cache L2 de 256KB, cache L3 de 8MB e memória RAM de 4GB. O sistema operacional executando na máquina foi o Ubuntu 13 x86_64, sob a versão de *kernel* 3.11.0-15-generic.

Benchmark Para avaliar a solução proposta foram utilizados os *benchmarks* do SPEC CPU2006 (Henning, 2006) implementados em C ou C++, para a entrada `train`. Isso devido a entrada `ref` possuir um alto tempo de execução, o que elevaria o tempo dos experimentos. Para evitar flutuações no tempo de execução, cada *benchmark* foi executado diversas vezes, até que a variância da amostragem de execução fosse menor do que 1.

O Framework de Compilação O processo de compilação de cada *benchmark* representa o ciclo básico de realização de todos os experimentos. Esse processo foi realizado com o auxílio da infraestrutura de compilação LLVM versão 3.4[3] (Lattner e Adve, 2004), a qual é formada por uma coleção de ferramentas.

O arquivo-fonte escrito em C ou C++ é submetido à ferramenta *clang*, a qual é configurada para gerar código na representação intermediária da infraestrutura sem a utilização de nenhuma otimização. Após, a ferramenta *opt* transforma o código aplicando uma sequência de otimizações específica. Em seguida, a ferramenta *llc* gera a partir da versão otimizada, o código assembly para a arquitetura alvo. Por fim, as ferramentas *as* e *ld* são utilizadas para gerar o executável.

Baseline Para definir o nível de otimização que seria utilizado como *baseline* um conjunto de experimentos foi realizado com os *benchmarks* do SPEC CPU2006. Nesses experimentos o nível de otimização -O2 obteve no geral o melhor desempenho. Dessa forma, a sequência -O2 foi escolhida como *baseline*. Essa sequência é também a sequência de partida para a metaheurística `VNS`, e as otimizações presentes nela formam o conjunto no qual novas otimizações serão buscadas.

Comparação Uma questão importante é avaliar o algoritmo proposto com algum algoritmo apresentado na literatura. O algoritmo `CE`, desenvolvido por Pan e Eigenmann (Pan e Eigenmann, 2006), foi o escolhido para comparação. Isso devido a dois fatores: (1) por `CE` pertencer à mesma classe do algoritmo proposto – compilação iterativa; (2) pela literatura apresentar bons resultados para `CE`.

Parâmetros da Metaheurística Os parâmetros do algoritmo proposto são apresentados na Tabela 1.

Parâmetro	Valor	Parâmetro	Valor
Conjunto inicial	[-O2]	Taxa de perturbação	10%
Tempo de execução	`CE` e 4h	Taxa de troca	40%
Iterações	25	Taxa de remoção	20%
Avaliações	5000	Taxa de inserção	10%
Desempenho	50%		

Tabela 1. Parâmetros utilizados para a metaheurística `VNS`.

[3]http://www.llvm.org

Cenários A análise experimental compreende dois cenários diferentes:

1. O algoritmo proposto foi parametrizado com o tempo de execução do algoritmo `CE`. O primeiro cenário tem por objetivo identificar o desempenho do algoritmo proposto mediante o mesmo tempo de execução do algoritmo `CE`.
2. O algoritmo proposto foi parametrizado com o tempo de execução de 4 horas. O segundo cenário tem por objetivo padronizar a execução do algoritmo independente do *benchmark* em questão.

O tempo de execução do algoritmo `CE` é apresentado na Tabela 2.

Benchmark	Tempo de Execução	Benchmark	Tempo de Execução
400.perlbench	4:14:25.771	401.bzip2	8:03:37.629
403.gcc	2:54:07.430	429.mcf	3:18:52.776
433.milc	0:42:21.966	444.namd	2:10:15.865
445.gobmk	21:51:54.263	447.dealII	5:15:19.524
450.soplex	1:45:17.497	453.povray	2:53:19.272
456.hmmer	8:33:15.959	458.sjeng	18:37:17.695
462.libquantum	0:19:29.499	464.h264ref	8:59:54.149
470.lbm	3:29:05.226	471.omnetpp	14:05:05.937
473.astar	16:29:09.700	482.sphinx3	1:57:18.804
483.xalancbmk	28:19:00.061	-	-

Tabela 2. Tempos de execução de `CE` para cada *benchmark* do SPEC CPU2006 (hh:mm:ss).

A Metaheurística VNS O algoritmo proposto, por ser baseado em uma metaheurística, pode apresentar diferentes respostas em execuções distintas. Tradicionalmente, um algoritmo heurístico é executado N vezes e o resultado final computado a partir dessas execuções. A abordagem utilizada para avaliar o algoritmo proposto foi executá-lo 3 vezes e para cada *benchmark* e apresentar o resultado de cada execução.

5. Resultados Experimentais

A avaliação experimental é norteada por duas métricas: (1) melhoria e (2) avaliações. A primeira indica o percentual de melhoria alcançado por um determinado algoritmo, em relação ao *baseline*. A segunda indica a quantidade de compilações necessárias para o *benchmark* em questão.

5.1. Melhoria

A Figura 1 apresenta a melhoria alcançada em cada algoritmo. Para cada *benchmark* são apresentados 7 barras na seguinte ordem da esquerda para a direita: `CE`, `VNS(1)`, `VNS(2)`, `VNS(3)`, `VNS(4H.1)`,`VNS(4H.2)` e `VNS(4H.3)`. `VNS(`x`)` é a configuração do primeiro cenário, e `VNS(4H.`x`)` a do segundo cenário.

Cenário 1 Os resultados indicam que a estratégia utilizada para mitigar o PSO é melhor do que a estratégia utilizada por `CE`. Enquanto `CE` tem como premissa remover as otimizações prejudiciais, o algoritmo proposto parte da premissa que um ótimo global será encontrado a medida que o conjunto de otimizações possa ser expandido ou reduzido. Outro ponto importante, é o fato de `CE` mitigar apenas soluções pré-estabelecidas sem modificar sua ordem, enquanto o algoritmo proposto por meio do operador de trocas, tentar mitigar em conjunto o problema da ordenação de otimizações.

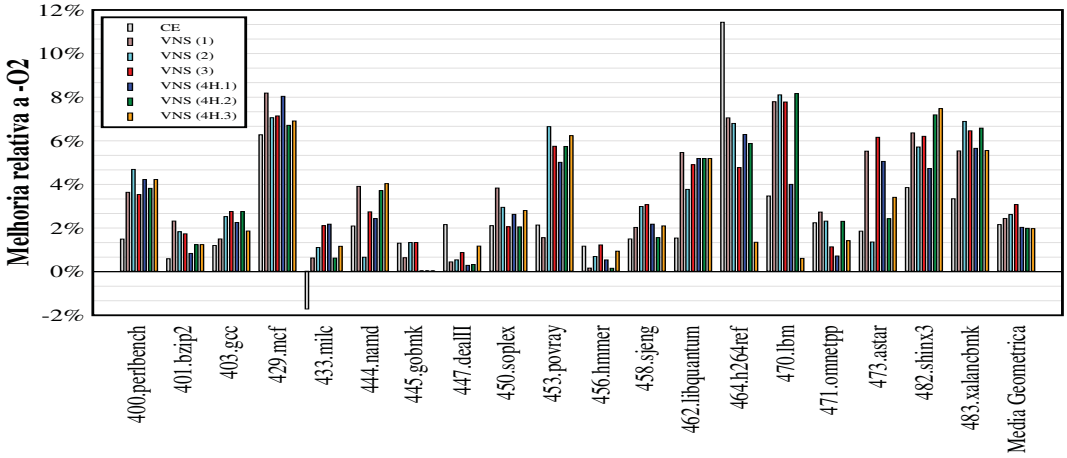

Figura 1. Melhoria alcançada por cada algoritmo, em relação ao *baseline*.

Apenas para dois *benchmarks* (447 e 464) a metaheurística não superou `CE`. Para os outros *benchmarks* pelo menos uma execução superou `CE`, sendo que para 10 *benchmarks* todas execuções superaram `CE`.

O *speedup* médio[4] de `CE` é de $1,023$, enquanto o do algoritmo proposto é de $1,036$. Sendo de $1,036$, $1,035$ e $1,038$, para `VNS(1)`, `VNS(2)`, `VNS(3)`, respectivamente.

Uma análise da melhoria do algoritmo proposto (valor médio entre as três execuções) indica que este é capaz de alcançar um desempenho superior a `CE` que varia entre $15,71\%$ (429) e 250,38% (433), desconsiderando os *benchmarks* para os quais houve perda de desempenho (que variou entre $-8,89\%$ e $-261,85\%$, para os *benchmarks* 444, 445, 447, 456, 464 e 471).

Cenário 2 Esta configuração pode beneficiar alguns *benchmarks*, enquanto reduz as chances da metaheurística encontrar um ótimo global para outros. Para o *benchmark* 400, o tempo de execução do algoritmo proposto foi praticamente o mesmo do utilizado no cenário 1. Por outro lado, para os *benchmarks* 401, 445, 447, 456, 458, 464, 471, 473 e 483 houve uma redução no tempo de execução, enquanto para o restante houve aumento.

Esse fator possui prós e contras. Tradicionalmente, quanto maior o tempo de execução de um algoritmo heurístico maior é a probabilidade de ser encontrado um ótimo global e não um ótimo local. Por outro lado, um tempo de execução elevado não garante necessariamente que um ótimo global será encontrado, pela natureza do algoritmo. Por isso é necessário ter vários critérios de parada, além de um auto-ajuste nas características do algoritmo. No algoritmo proposto, exitem 4 critérios de parada, além do auto-ajuste das taxas dos operadores.

Em uma análise geral, neste segundo cenário, o algoritmo proposto não superou `CE` para 445 e 456, além dos dois *benchmarks* do cenário 1. Contudo, nesta nova configuração o algoritmo proposto superou `CE` para 12 *benchmarks*, em todas as execuções.

O *speedup* médio obtido pelo algoritmo proposto é de $1,033$. Sendo de $1,033$, $1,035$ e $1,030$, para `VNS(4H.1)`, `VNS(4H.2)`, `VNS(4H.3)`, respectivamente.

[4]Os valores médio se referem a média geométrica.

Uma análise da melhoria do algoritmo proposto (valor médio entre as três execuções, como no cenário anterior) indica que o cenário 2 tem desempenho similar ao cenário 1 no tocante a quantidade de *benchmarks* para os quais o algoritmo proposto obteve uma melhor solução do que CE. Quanto ao ganho, este variou entre $12,80\%$ e $246,81\%$. Enquanto a perda variou entre $-28,07\%$ e $-3499,73\%$, para os *benchmarks* 445, 447, 456, 464, 470 e 471.

Observando o desempenho do algoritmo proposto é possível perceber que a redução (ou aumento) do tempo de execução impactou o percentual de melhoria. Para todos os *benchmarks* que tiveram seu tempo de execução reduzido houve uma queda de desempenho comparado tanto com a execução do cenário 1, consequentemente uma perda do ganho sobre CE. Embora, neste último caso, a melhor opção ainda seja utilizar o algoritmo proposto.

Para 5 *benchmarks* (403, 444, 453, 462 e 482) o aumento do tempo de execução resultou em um ganho de desempenho. É importante perceber que o aumento do tempo de execução não é proporcional ao ganho obtido. Para 444 o tempo de execução dobrou e o ganho em relação a CE, que para um tempo de execução maior foi de $1,923\%$, passou para $3,319\%$. Para 462 o tempo de execução aumentou em um fator de 12 vezes, contudo o ganho passou de $4,649\%$ para $5,191\%$. Além disso, para 429, 450 e 470 houve uma perda de desempenho para um tempo de execução maior.

Em síntese os resultados indicam que a melhor abordagem é aumentar o tempo de execução do algoritmo. Além disso, a análise detalhada dos dados indica que a estratégia de auto-ajuste das taxas dos operadores é uma boa estratégia para garantir que o algoritmo continue convergindo por um longo período. Para todos os *benchmarks*, o algoritmo terminou por atingir o tempo de execução máximo. Além disso, para todos foram encontradas soluções parciais (as quais não são apresentadas graficamente pela restrição de espaço).

5.2. Avaliações

A Figura 2 apresenta a quantidade de avaliações de cada algoritmo. Como a Figura 1, para cada *benchmark* são apresentados 7 barras na seguinte ordem da esquerda para a direita: CE, VNS(1), VNS(2), VNS(3), VNS(4H.1),VNS(4H.2) e VNS(4H.3). VNS(x) é a configuração do primeiro cenário, e VNS(4H.x) a do segundo cenário.

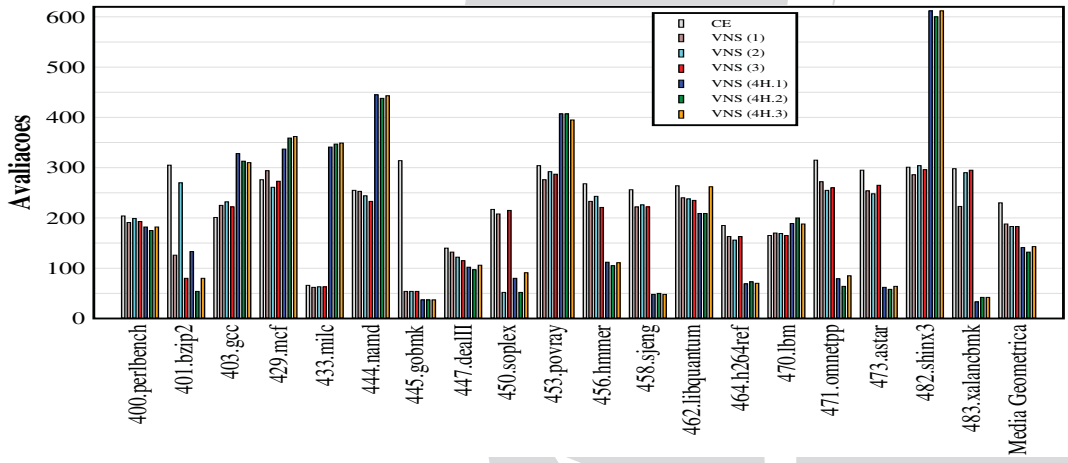

Figura 2. Avaliações necessárias por cada algoritmo.

Como esperado a quantidade de avaliações necessárias por `VNS(x)` é similar a quantidade necessária por `CE`. Isso por ambos os algoritmos terem o mesmo tempo de execução.

Para alguns *benchmarks* existe uma variação nessa quantidade, que pode ser explicada pela natureza do conjunto gerado pelo algoritmo proposto. Em geral, o `CE` avalia otimização por otimização para determinar se ela é nociva ao desempenho do *benchmark* em questão. Como o algoritmo proposto, além de remover, investe em trocas e inserções, converge mais rápido para uma melhor solução. Consequentemente, reduz o tempo de execução do *benchmark* proporcionando que novas avaliações sejam realizadas.

Um ponto importante a ser ressaltado é o fato de ser possível obter um ganho de desempenho sobre um nível de otimização do compilador em questão, com uma quantidade menor de avaliações do que a necessária por `CE`.

Isso é um ponto importante, pois assim como `CE` o algoritmo proposto é um algoritmo de compilação iterativa. Desta forma, quanto menor a quantidade de avaliações necessárias menor será o tempo de resposta do sistema.

6. Conclusões e Trabalhos Futuros

Dada a necessidade de produzir códigos com mais qualidade, os projetistas de compiladores implementam dezenas de otimizações. No entanto, a complexidade da relação entre as diferentes otimizações torna complexo determinar qual conjunto irá garantir um bom desempenho. Dessa forma, é necessário avaliar diversas possibilidades. Contudo, a exploração de todas as possibilidades é inviável devido ao tamanho do espaço de busca.

A abordagem proposta neste artigo para mitigar o problema em questão permitiu a exploração de um amplo espaço de busca, apresentando bons resultados. A utilização de taxas de exploração proporcionou a maximização do espaço de busca, permitindo que em uma possível estagnação do algoritmo a busca se concentrasse em outro ponto do espaço.

A avaliação experimental indica as contribuições deste trabalho. Os resultados evidenciaram que os operadores de busca local definidos para a metaheurística utilizada são adequados ao PSO. Para todos os experimentos realizados não houve sequer um *benchmark*, para o qual o algoritmo proposto não conseguisse alguma melhoria. O algoritmo proposto mostrou que é possível obter resultados melhores do que aqueles obtidos por `CE`, mesmo com uma quantidade menor de avaliações.

Um trabalho futuro será implementar novas estratégias baseadas em outras metaheurísticas. Um trabalho mais ambicioso é avaliar um código gerado com um conjunto de otimizações e estimar o tempo de execução, sem a necessidade de uma execução real. Dessa forma, será possível reduzir o tempo de resposta do sistema, mesmo mediante centenas de *avaliações*.

Referências

Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M. F. P., e Temam, O. (2007). Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA. IEEE Computer Society.

Foleiss, J. H., da Silva, A. F., e Ruiz, L. B. (2011). The Effect of Combining Compiler Optimizations on Code Size. In *Proceedings of the International Conference of the Chilean Computer Science Society*, pages 1–8, Curicó, Chile. Sociedad Chilena de Ciencias de la Computación.

Haneda, M., Knijnenburg, P. M. W., e Wijshoff, H. A. G. (2005). Automatic Selection of Compiler Options Using Non-parametric Inferential Statistics. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA. IEEE Computer Society.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17.

Javadian, N., Mozdgir, A., Kouhi, E., Qajar, D., e Shiraqai, M. (2009). Solving Assembly Flowshop Scheduling Problem with Parallel Machines Using Variable Neighborhood Search. In *Computers Industrial Engineering, 2009. CIE 2009. International Conference on*, pages 102–107.

Lattner, C. e Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, Palo Alto, California.

Lau, J., Arnold, M., Hind, M., e Calder, B. (2006). Online Performance Auditing: Using Hot Optimizations Without Getting Burned. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 239–251, New York, NY, USA. ACM.

Leather, H., Bonilla, E., e O'Boyle, M. (2009). Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 81–91, Washington, DC, USA. IEEE Computer Society.

Lei-fu, G. e Wei, D. (2010). A Parallel Variable Neighborhood Search for the Traveling Salesman Problem. In *Advanced Management Science (ICAMS), 2010 IEEE International Conference on*, volume 3, pages 150–152.

Mladenović, N. (1995). A Variable Neighborhood Algorithm – A New Metaheuristics for Combinatorial Optimization. In *Abstracts of Papers Presented at Optimization Days. Montreal*.

Pan, Z. e Eigenmann, R. (2006). Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA. IEEE Computer Society.

Park, E., Kulkarni, S., e Cavazos, J. (2011). An Evaluation of Different Modeling Techniques for Iterative Compilation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 65–74, New York, NY, USA. ACM.

Purini, S. e Jain, L. (2013). Finding Good Optimization Sequences Covering Program Space. *ACM Transactions on Architecture and Code Optimization*, 9(4):56:1–56:23.

Smith, J. e Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Zhang, G. (2012). Hybrid Variable Neighborhood Search for Multi Objective Flexible Job Shop Scheduling Problem. In *Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on*, pages 725–729.