The Use of Different Strategies of Search Space Reduction in Mitigation of Optimization Selection Problem

Nilton Luiz Queiroz Junior
Departamento de Informática
Universidade Estadual de Maringa
niltonlqjr@gmail.com

Anderson Faustino da Silva
Departamento de Informática
Universidade Estadual de Maringa
andersonfaustino@gmail.com

Abstract—**Compiler optimizations are transformations that are applied on the source code to improve its performance. Many times its a complex task choose which optimizations set must be used, so, usually are chosen optimization levels given by the compiler. However, this optimization levels not always are good enough to all programs. Thus, is needed to search for sets to improve specific programs. Currently the Best10 algorithm one of the best algorithms to mitigate the optimizations selection problem. This algorithm require one reduced search space to infer which are the optimization sets that must be applied during the compilation of programs. This work present the impact of the use of different search space creation strategies used by the Best10 algorithm. The results shows that sophisticated strategies do not always provide the best results.**

Keywords: Optimization Selection Problem, Best10 Algorithm, Performance.

I. Introdução

Compiladores são programas que transformam o código fonte de uma linguagem de programação para outra, geralmente transformando de uma linguagem de alto nível para uma linguagem a nível de máquina. Durante este processo podem ser aplicadas transformações (também conhecidas como otimizações) para melhorar o desempenho do código final [1] [2] [3].

Existem diversas otimizações que podem ser aplicadas [4]. Geralmente elas são independentes da arquitetura, assim, podem ser aplicadas diretamente no código intermediário. A aplicação destas otimizações nem sempre melhora o desempenho do programa, podendo algumas vezes inclusive piorá-lo.

Assim, o usuário do compilador deve usar seu conhecimento prévio para escolher as otimizações que serão aplicadas, porém isso é um grande desafio [5]. Quando o usuário não é muito experiente, ele pode usar algum dos conjuntos que normalmente os compiladores oferecem, tais como os níveis -O1, -O2, -O3 (conjuntos de otimizações), que são disponibilizados por compiladores como: GCC [6], LLVM [7], open64 [8], ekopath [9], entre outros.

O mesmo conjunto de otimizações não gera o melhor código final para todos programas. Assim, algumas vezes para obter melhorias em alguns programas, devemos construir conjuntos específicos para cada programa, ou então tentar encontrar conjuntos que sejam bons para cobrir programas parecidos. Com isso surge o Problema de Seleção de Otimizações (PSO). Esse problema se trata de encontrar o melhor conjunto de otimizações para um programa, com o objetivo de melhorar o seu desempenho.

Para mitigar o PSO existem diversas maneiras propostas na literatura [10] [11] [12] [13] [14] [15] [16] [17] [18] [19]. Destas, as mais utilizadas são aquelas que fazem uso de aprendizagem de máquina, pelo fato destas reduzirem os pontos do espaço de busca que serão avaliados.

Dentre estas propostas, está a técnica que faz uso de um algoritmo de extração, apresentado por Purini e Jain [16], onde são extraídos, a partir de um espaço de busca reduzido, os 10 melhores conjuntos. Esta abordagem é nomeada, pelos autores, de Best10.

Atualmente, Best10 apresenta uma das melhores estratégias para mitigação do PSO. Tal estratégia, inicialmente, cria um espaço de busca reduzido e em seguida extrai deste espaço os 10 melhores conjuntos. Os quais serão aplicados durante a compilação do programa teste. Portanto, o objetivo desta estratégia é encontrar os melhores conjuntos que cubram diferentes classes de programas, de forma a alcançarem um desempenho superior ao maior nível de otimização do compilador em questão.

Desta forma, Best10 necessita de alguma estratégia eficiente para a criação do espaço de busca. Purini e Jain propõem uma elegante estratégia para a criação deste espaço. Contudo, tais autores não discutem a qualidade dos resultados obtidos por Best10, mediante o uso de diferentes espaços de busca.

Este trabalho tem como objetivo cobrir esta lacuna e desta forma descrever processos de desenvolvimento de bons espaços de busca, que possam ser utilizados por diferentes estratégias de mitigação do PSO. Neste trabalho nós avaliamos distintamente as estratégias propostas por Purini e Jain, a saber: algoritmo genético com seleção por classificação, algoritmo genético com seleção por torneio e algoritmo aleatório. Além destas estratégias, propomos o uso de *Simulated Annealing* para a criação de um espaço de busca reduzido.

Surpreendentemente, os resultados experimentais indicaram que estratégias *mais elaboradas* nem sempre fornecem os melhores resultados. De fato, os melhores resultados, obtidos

978-1-4673-9143-6/15/$31.00 ©2015 IEEE

por Best10, foram alcançados utilizando um espaço de busca reduzido por meio de uma estratégia aleatória.

O artigo está organizado da seguinte maneira: a Seção II apresenta o referencial teórico; a Seção III aborda as estratégia de redução do espaço de busca utilizado pelo algoritmo Best10; a Seção IV explica como foram criados os espaços de busca; a Seção V caracteriza os espaços reduzidos criados; a Seção VI aborda sobre os conjuntos extraídos dos espaços de busca, pelo algoritmo Best10; a Seção VII apresenta os resultados experimentais; e por fim, a Seção VIII apresenta as conclusões e trabalhos futuros.

II. Referencial Teórico

A. *O Problema da Seleção de Otimização*

Compiladores otimizantes [4] proveem diversos conjuntos de otimização (níveis), os quais são determinados durante o seu desenvolvimento e/ou parametrização. Porém, cada programa requer um conjunto específico para obter um bom desempenho. Portanto, determinar o conjunto correto, aquele que deve ser utilizado no processo de compilação, é uma importante questão que precisa ser analisada.

O PSO, cujo objetivo é encontrar o melhor conjunto que alcança o objetivo esperado, em geral reduzir o tempo de execução, pode ser definido como segue. Seja P um código fonte, G o objetivo desejado, tenta-se encontrar o conjunto $O = < t_1, t_2, ..., t_n >$, onde $t_1, t_2, ..., t_n \in T$ e $n \in \mathbb{N}$, tal que O alcance um bom desempenho. Em outras palavras, que O que maximize o objetivo desejado e tenha um desempenho superior aquele objetivo pelo mais alto nível de compilação, do compilador em questão.

Este enunciado demonstra que o PSO é um problema exploratório, que pode ser resolvido utilizando um algoritmo de busca. Porém, explorar todo o espaço de busca é algo impraticável. Se a estratégia de busca considerar 20 otimizações, a quantidade de possíveis conjuntos será 2^{20}. Se também for considerado a ordem das otimizações, esta quantidade cresce para 20^{20}. Isto indica que a melhor estratégia é explorar uma pequena fração do espaço de busca.

B. *Algoritmo Genético*

Algoritmos genéticos são métodos baseados na evolução biológica. Eles procuram encontrar boas soluções fazendo busca em um espaço. A busca começa com uma população inicial composta por indivíduos. Durante sua execução novas gerações são criadas por meio dos processos de *crossover* e mutação sobre a população atual [20].

O algoritmo faz uso de uma função de *fitness*. Esta função avalia os indivíduos da geração atual. Com base nos resultados desta avaliação, alguns indivíduos são escolhidos para compor a nova geração. Esta função varia de acordo com o problema, porém sempre tem como entrada um indivíduo e irá retornar um valor real. Este valor será maior para os melhores indivíduos [21].

A representação dos indivíduos é feita por meio de uma *string* sobre um alfabeto finito. Cada elemento da *string* é chamado de gene. A reprodução desses indivíduos ocorre por meio da operação de *crossover*. Esta operação escolhe um ponto, chamado de ponto de *crossover*, e cria dois novos indivíduos à partir de dois outros já existentes na população atual. Um dos indivíduos criado é a primeira parte do primeiro indivíduo concatenado a segunda parte do segundo, e o outro é a segunda parte do primeiro indivíduo concatenado a primeira parte do segundo [21].

Cada indivíduo da nova geração pode ser alterado pelo operador de mutação, que normalmente, altera apenas um gene do indivíduo.

Para a seleção da nova geração são adotados alguns métodos:

- Seleção por Classificação: Os indivíduos são ordenados pela função *fitness*. A probabilidade um indivíduo ser escolhido é proporcional a sua classificação [22];
- Seleção por Torneio: Dois indivíduos são escolhidos da população atual. Então com uma probabilidade pré definida p, o indivíduo com maior *fitness* é escolhido, e com uma probabilidade $(1 - p)$ o outro indivíduo é o escolhido [22].

C. *Simulated Annealing*

Este Algoritmo é inspirado no processo de *Annealing* da metalúrgica. Neste processo, o material tem sua temperatura elevada, e então esta é abaixada gradualmente até alcançar um estado de baixa energia [21].

O *Simulated Annealing* é um algoritmo que faz busca de estados ótimos. Esta busca faz uso de um valor de temperatura, o qual é um parâmetro que inicialmente tem um valor alto, e é ajustado durante o processo de busca [23].

A ideia do algoritmo é escolher um estado vizinho ao estado atual, e adotá-lo nos casos em que ele é uma solução melhor que a atual. O estado também pode ser adotado com uma probabilidade que é calculada dependendo da temperatura na iteração atual [21].

O algoritmo tem uma função de variação de energia ΔE, a qual é calculada à partir do estado atual e do vizinho escolhido. Esta função tem como objetivo decidir se o novo estado será ou não aceito.

A temperatura, ao início do algoritmo, deve ser alta o suficiente para permitir que todas as transações (de um estado à outro) sejam aceitas, mesmo que a probabilidade seja baixa [24]. Assim o pior estado pode ser aceito no começo do algoritmo. Esta é uma estratégia para escapar de mínimos, ou máximos, locais.

Para reduzir a temperatura, usada para o calculo de aceitação do estado vizinho, é usado um método, que é normalmente uma equação matemática. O método mais comum dentre os usados é o método geométrico, o qual multiplica a temperatura atual por uma constante α. Esta constante normalmente esta no intervalo $0.8 < \alpha < 0.99$ [25] [24].

O algoritmo também conta com a função de perturbação, que é a função responsável por criar o novo vizinho a partir do estado atual. Essa função apenas provoca algumas alterações, geralmente aleatórias, no estado atual.

D. Trabalhos Relacionados

A literatura apresenta diversos trabalhos cujo objetivo é mitigar o PSO. Alguns destes serão apresentados nesta seção.

Lima *et al.* [5] propôs o primeiro sistema de Raciocínio Baseado em Casos para encontrar bons conjuntos de otimizações para um programa. Esta abordagem usa três diferentes estratégias para a seleção dos conjuntos, e duas estratégias para a medida de similaridade entre os programas. Além de desempenho, o trabalho de Lima *et al.* buscava também por otimizações que reduziam o consumo de energia, fazendo uma ponderação entre desempenho e consumo de energia. Este trabalho conseguiu constatar que é possível reduzir o consumo de energia sem sacrificar o desempenho.

Cavazos [13] fez uso de técnicas de regressão logística e introduziu o uso de *performance counters* (PC) para predizer bons conjuntos. Foram criados 500 conjuntos de otimizações para diversos programas de treino, e então um modelo de predição foi criado a partir de um processo *leave-one-out cross-validation*. Para caracterizar os programas foi feita a extração de 60 PCs, esses normalizados pelo valor do total de instruções de cada programa, e então colocados em vetores. Os resultados indicaram que na média, o modelo proposto obteve *speedups* de 1.17. Esta abordagem foi comparada a abordagem *combined elimination* que obteve *speedups* de 1.09, e a uma abordagem que gerava conjuntos aleatórios, a qual conseguiu alcançar o mesmo desempenho do modelo proposto, porém precisou compilar mais vezes o programa.

Long e O'Boyle [26] utilizaram uma técnica de aprendizado baseado em instâncias para procurar por melhores conjuntos de otimizações. Foi feito o uso de características estáticas do código fonte para caracterizar os programas. Para a construção dos exemplos anteriores os programas foram otimizados e compilados usando uma estratégia de busca simples. Os experimentos empregaram uma estratégia de *cross-validation* para a avaliação. Os resultados alcançados apontam um *speedup* médio de 1.15 sobre os programas não otimizados.

Park [27] introduz um modelo de caracterização de programas fazendo uso do grafo de fluxo de controle. Para isso, foram gerados 600 conjuntos aleatórios, e então construído um modelo de predição por meio de um processo *leave-one-out cross-validation* com a SVM. O modelo baseado em grafo fez uso do *kernel* de caminhos mais curtos para a comparação entre os grafos. Este modelo foi comparado também à caracterização de programas por meio de *performance counters*, reações e características de código fonte. O modelo baseado em grafo obteve os melhores resultados. Com este modelo foi alcançado um ganho médio de 88% do ganho máximo nas 600 sequências geradas.

Zhou e Lin [28] fizeram uso do algoritmo genético NSGA-II para investigar compilações multiobjetivo, comparando-o com conjuntos gerados aleatoriamente. Os resultados indicam que o algoritmo NSGA-II tem um desempenho superior a conjuntos gerados aleatoriamente. Por sua vez, os conjuntos aleatórios tiveram melhor desempenho que os níveis padrões de otimização disponíveis no GCC.

III. O Algoritmo Best-10

O algoritmo Best-10, proposto por Purini e Jain [16], é um dos melhores algoritmos para mitigação do PSO.

Este faz uso de um espaço de busca reduzido, construído em um processo anterior, e de um conjunto de programas teste. Esses programas serão compilados e executados com os conjuntos pertencentes ao espaço de busca. Desta forma, o ganho, ou perda, de desempenho é calculado com base em algum nível de otimização padrão do compilador utilizado.

Após a execução e o cálculo de como o desempenho foi afetado, é criada uma matriz de sumário dos desempenhos, onde as linhas correspondem aos programas e as colunas correspondem aos conjuntos de otimizações.

Assim, é então extraído um conjunto dentre os presentes no espaço de busca reduzido. Para a extração deste conjunto são utilizados os seguinte critérios:

- O conjunto que aumenta o desempenho do maior número de programas comparado ao nível de otimização base (tradicionalmente o maior nível de otimização do compilador) é o escolhido;
- Em caso de empate, é escolhido o conjunto que tem a maior média geral de ganho de desempenho.

Após a seleção deste conjunto, o mesmo é armazenado em uma lista e sua coluna é retirada da tabela. Junto à coluna do conjunto, são também retiradas as linhas dos programas que obtiveram ganho de desempenho com o conjunto retirado.

Este processo é repetido até que não existam mais linhas na matriz, ou então, até que um numero pré determinado de conjuntos sejam escolhidos.

A estratégia de selecionar os melhores 10 conjuntos (Best-10) é uma estratégia interessante, pois ela pode prover conjuntos os quais se adaptariam a diversas classes de programas. Isso seria semelhante a propor 10 níveis padrões de otimização dados pelos compiladores, porém não tão genéricos. Assim, bons conjuntos poderão ser encontrados e usados como padrões para otimização de código, utilizando o Best-10 com um conjunto de programas representativos para a construção de espaços de busca reduzidos.

Purini e Jain [16] demonstraram que o algoritmo Best-10 é capaz de obter bons resultados. Embora, não apresentem uma avaliação do uso de distintos espaços de busca. Este trabalho preenche esta lacuna avaliando mais detalhadamente o algoritmo Best-10. Além disto, o presente trabalho apresenta uma diferente estratégia para a criação do espaço de busca, *Simulated Annealing*.

O pseudo código de Best-10 é apresentado no Algoritmo 1.

IV. A Criação dos Espaços de Busca

A criação dos espaços de busca reduzidos é um fator que influenciará a qualidade dos resultados obtidos. Isso ocorre pois os conjuntos de otimizações vem destes espaços, e os mesmos não são modificados.

Algoritmo 1: Best-10 [16]

```
Entrada: Lista com os conjuntos do espaço de busca
         reduzido, lista dos programas Microkernel,
         Matriz de sumario das performances
Saída: Os conjuntos selecionados presentes na lista
       Best10
conjuntos ← Lista com os conjuntos do espaço de
busca reduzido
programas ← lista dos programas Microkernel
PM ← Matriz de sumario das performances
para cada conjunto em conjuntos faça
    para cada programa em programas faça
        O3imp ← total de instruções da execução de
        programa otimizado com -O3 da LLVM
        se PM[programa, conjunto] < O3imp então
            Dict[conjunto].append([programa,
            melhoria_sobre_O2])
Best10 ← {}
para i = 1 até 10 faça
    /*A funcao getmaximum retorna a sequência que
    cobriu mais programas. Em caso de empate a com
    melhor media na melhoria e escolhida*/
    S ← getmaximum(Dict)
    Best10.append(S)
    /esse para exclui todos programas que são cobertos
    por S*/
    para cada conjunto em Dict.chaves() faça
        para cada programa em Dict[S] faça
            se programa ∈ Dict[seq] então
                Dict[seq].delete(programa)
```

Para a criação dos espaços de busca reduzidos utilizamos as seguintes estratégias:

- Algoritmo genético com seleção por classificação;
- Algoritmo genético seleção por torneio;
- Aleatório;
- *Simulated Annealing*; e
- A união de todos os espaços;

Algoritmos genéticos são utilizados para mitigar o PSO, não só para tempo de execução mas também para tamanho de código, por meio de compilação iterativa [29] [19] [28]. Como são usados para a compilação iterativa, onde são gerados conjuntos por meio da exploração do espaço, esses algoritmos tornam-se uma estratégia interessante para a redução do espaço de busca [30].

O algoritmo *Simulated Annealing*, é um algoritmo que busca por estados, permitindo movimentos que piorem o desempenho atual para alcançar bons resultados no futuro [25]. Porém, suponha um conjunto C, um pouco pior que o melhor conjunto encontrado, M, para o programa P. Onde P é um programa que está sendo usado de exemplo, e suponha também um programa Q desconhecido, porém semelhante a P. O resultado de C pode ser um resultado com bons ganhos para Q e P, porém o resultado de M pode ser bom somente para P, assim utilizar a estratégia de escolher movimentos piores e armazená-los pode fornecer bons resultados.

A estratégia aleatória é uma das mais utilizadas dentro da literatura para gerar conjuntos de exemplos. O uso dessa estratégia pode ser visto principalmente nas abordagens que utilizam aprendizagem de máquina [5] [13] [27].

Purini e Jain [16] utilizaram a estratégia aleatória e algoritmos genéticos em seu trabalho. Além disto, eles utilizaram a redução de conjuntos, eliminando as otimizações que não alteravam o desempenho. Esses conjuntos gerados a partir da redução de conjunto também são melhorados por sugestões disponíveis no manual das otimizações da LLVM [31].

Neste trabalho, além das estratégias utilizadas por Purini e Jain, utilizaremos também outra estratégia para a redução do espaço de busca, e a simplificação dos conjuntos não será executada, pois a ideia é avaliar os diferentes modos de reduzir os espaço de busca.

A. *Os Conjuntos de Otimizações*

Um conjunto de otimizações é um grupo de otimizações, que desconsidera a ordem em que elas são aplicadas. Desconsiderar a ordem é muitas vezes válido, pois a maior parte dos compiladores não permite que a ordem das otimizações seja alterada [13]. Porém o problema de ordenar o conjunto e decidir quantas vezes cada otimização será aplicada é também alvo de pesquisas. Contudo, estas duas últimas questões não serão consideradas neste trabalho.

As otimizações utilizadas por formar os conjuntos são as otimizações do nível -O3 da LLVM, que, além de usado como *baseline*, é também o nível que contém o maior número de otimizações. Essas otimizações são exibidas na Tabela I.

Tabela I. OTIMIZAÇÕES USADAS PELO NÍVEL -O3 DA LLVM

Otimizações		
-inline	-prune-eh	-scalar-evolution
-argpromotion	-inline-cost	-indvars
-gvn	-functionattrs	-loop-idiom
-slp-vectorizer	-sroa	-loop-deletion
-globaldce	-domtree	-loop-unroll
-constmerge	-early-cse	-memdep
-targetlibinfo	-lazy-value-info	-memcpyopt
-no-aa	-jump-threading	-sccp
-tbaa	-loop-unswitch	-dse
-basicaa	-tailcallelim	-adce
-notti	-reassociate	-barrier
-globalopt	-loops	-branch-prob
-ipsccp	-loop-simplify	-block-freq
-deadargelim	-lcssa	-loop-vectorize
-instcombine	-loop-rotate	-strip-dead-prototypes
-simplifycfg	-licm	-verify
-basiccg	-correlated-propagation	

É mais justo utilizar somente otimizações do conjunto com o qual os resultados serão comparados, este é então outro motivo para restringir as otimizações somente às usadas pelo nível -O3 da LLVM.

Todos conjuntos foram gerados por meio do uso dos algoritmos citados na seção IV, onde não é considerada a ordem, ou seja, a otimização pode aparecer em qualquer posição.

B. Os Programas Treino

Todos os conjuntos de otimização foram criados utilizando 61 aplicações *microkernel* retiradas da LLVM *test-suite*, as quais são apresentadas na Tabela II.

Os programas escolhidos são programas simples e curtos de apenas um arquivo. Assim, a geração dos espaços reduzidos pôde ser realizada em um tempo reduzido. Além disso, a diversidade dos programas é grande. Este conjunto de programas inclui programas de ordenação, programas com operações sobre ponto flutuante, programas recursivos (incluindo recursão em cauda), transformadas, como a transformada rápida de Fourier, operações sobre strings, entre outros.

Tabela II. PROGRAMAS MICROKERNEL PARA A REDUÇÃO DOS ESPAÇOS

Programas Microkernel		
ackermann	hash	perlin
ary3	heapsort	perm
bubblesort	himenobmtxpa	pi
chomp	huffbench	puzzle
dry	intmm	puzzle-stanford
dt	lists	queens
fannkuch	lowercase	queens-mcgill
fbench	lpbench	quicksort
ffbench	mandel-2	random
fib2	mandel	realmm
fldry	matrix	recursive
flops-1	methcall	reedsolomon
flops-2	misr	richards_benchmark
flops-3	n-body	salsa20
flops-4	nestedloop	sieve
flops-5	nsieve-bits	spectral-norm
flops-6	objinst	strcat
flops-7	oourafft	towers
flops-8	oscar	treesort
flops	partialsums	whetstone
fp-convert		

C. Estratégias para a Criação dos Espaços de Busca

Utilizamos 5 estratégias para criar diferentes espaços de busca reduzido. Os conjuntos criados obedecem os mesmos critérios, buscando sempre conjuntos com 40 otimizações, dentro das apresentadas na Tabela I.

1) Algoritmo Genético com Seleção por Classificação: Os parâmetros usados, para a criação do espaço de busca, pelo algoritmo genético com seleção por classificação foram:

- População composta por 60 indivíduos;
- Taxa de *crossover* 0,9;
- Taxa de mutação 0,02;
- Função *fitness* retorna a média do total de instruções de hardware de 5 execuções, excluindo a melhor e a pior;
- Foi empregado elitismo; e
- Foram utilizados dois critérios de parada: quando o desvio padrão da variância das *fitness* alcançar um valor menor que 0,01; ou o algoritmo não converge em 3 iterações consecutivas.

2) Algoritmo Genético com Seleção por Torneio: Os parâmetros que usamos para a criação do espaço reduzido pelo algoritmo genético com seleção por torneio foram:

- População composta por 60 indivíduos;
- Taxa de *crossover* 0,9;
- Taxa de mutação 0,02;
- Função *fitness* retorna a média do total de instruções de hardware de 5 execuções, excluindo a melhor e a pior;
- Foi empregado elitismo; e
- Foram utilizados dois critérios de parada: quando o desvio padrão da variância das *fitness* alcançar um valor menor que 0,01; ou então o algoritmo não converge durante 3 iterações consecutivas.

3) Aleatório: Para a criação dos conjuntos aleatórios usamos apenas os seguintes parâmetros:

- Seleção de otimizações aleatoriamente; e
- Criação de um total de 500 conjuntos.

4) Simulated Annealing: Usamos o algoritmo *Simulated Annealing* com a seguinte parametrização:

- Temperatura inicial é dada pela metade de instruções de hardware numa média de 5 execuções excluindo a melhor e a pior;
- Método de ajustamento de temperatura Geométrico com constante $\alpha = 0,95$;
- A função de perturbação altera somente uma otimização do conjunto de uma posição aleatória;
- A probabilidade de aceitação das soluções piores que a atual é descrita pela equação 1; e
- O critério de parada é dado pela execução de 500 iterações do algoritmo.

$$P(S_c) = e^{-\frac{(\Delta(S_c)-\Delta(S_v))}{T}} \quad (1)$$

5) Todos: A criação do espaço que contém todos foi feita por meio da concatenação das demais.

V. OS ESPAÇOS DE BUSCA CRIADOS

Como foram gerados utilizando diferentes estratégias, os espaços de busca reduzidos apresentam diferentes características, as quais serão analisadas a seguir.

Para a criação de cada espaço de busca, os algoritmos foram executados apenas uma vez. Como são algoritmos que não geram sempre as mesmas saídas, executá-los mais de uma vez faria com que gerássemos mais de uma base por estratégia.

Dentro do escopo desta análise, serão considerados bons casos aqueles que atingiram um desempenho superior ao nível de otimização O3 da LLVM.

Ao analisarmos o máximo desempenho das estratégias para a redução do espaço os programas *nested-loops* e *puzzle-standford* serão desconsiderados pois as melhorias foram muito acima do comum, para todas as estratégias usadas na redução dos espaços de busca.

A representação gráfica dos espaços é exibida em forma de gráficos de violino pois neles pode-se ter uma noção da distribuição da amostra dos dados por meio do formato do violino. Nos gráficos de violino apresentados nesta seção, o ponto central no interior de cada violino representa a mediana da amostra de melhorias coletadas.

A. *Algoritmo Genético com Seleção por Classificação*

Pode-se observar pela Figura 1 que nem todos programas conseguiram encontrar conjuntos com desempenho superior à O3.

Nota-se também que para a maior parte dos programas as medianas estão muito próximas do desempenho que é alcançado pelo nível O3. A porcentagem de bons casos encontrados foi de 39,42% , ou seja, 39,42% são conjuntos com melhor desempenho que o nível de otimização O3.

A maior melhoria alcançada para o algoritmo genético com seleção por classificação foi de 51,87% para o programa *quicksort*.

Foram achados conjuntos com desempenho melhor que o do nível O3 para 57 dos 61 programas teste.

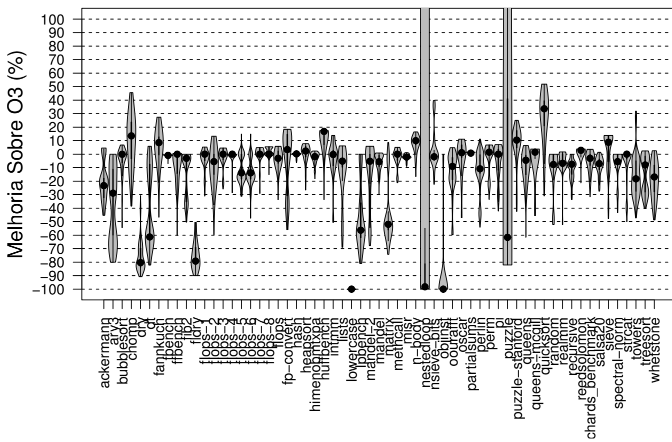

Figura 1. Espaço de busca reduzido criado pelo algoritmo genético com seleção por classificação

B. *Algoritmo Genético com Seleção por Torneio*

Uma das principais características do espaço de busca gerado pelo algoritmo genético com seleção por torneio é a quantidade de bons casos alcançados. O total de conjuntos encontrados que superaram o nível de otimização O3 foi de 56,77%.

Pode-se observar pela Figura 2 que as medianas estão quase todas próximas do nível de otimização O3, e algumas delas inclusive se encontram muito próximas do melhor conjunto encontrado para o programa. Isto significa que metade dos conjuntos tem desempenho próximo ao melhor desempenho encontrado, isso para mais da metade dos casos.

A maior melhoria alcançada por este algoritmo foi também para o programa *quicksort* e, também de 51,87%. Foram achados conjuntos com desempenho melhor que o do nível O3 para 56 dos 61 programas.

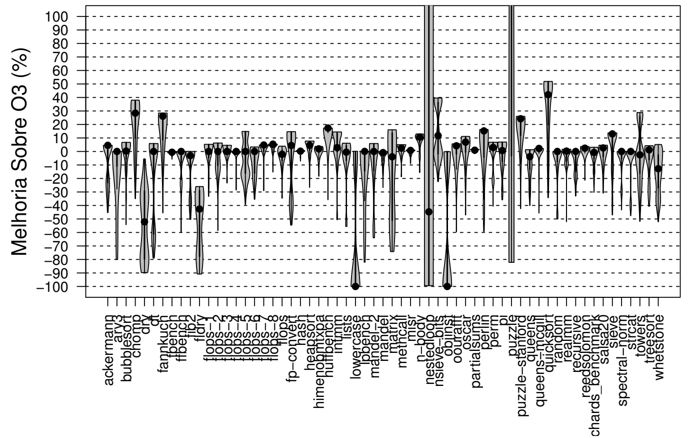

Figura 2. Espaço de busca reduzido criado pelo algoritmo genético com seleção por torneio

C. *Algoritmo Aleatório*

O algoritmo aleatório teve um desempenho inferior ao dos algoritmos genéticos. Ainda assim, conseguiu encontrar bons conjuntos e para alguns programas encontrou conjuntos melhores que os algoritmos genéticos, porém não em mesma quantidade.

Surpreendentemente, a quantidade de bons casos é maior que a do *Simulated Annealing*.

Pode-se observar que o espaço gerado pela estratégia aleatória não obteve tantas medianas altas como os espaços gerados pelos algoritmos genéticos. Porém o melhor caso foi de 58,33%, sendo o melhor dentre os encontrados (descartando os programas *nestedloops* e *puzzle-standford* que estão com melhorias fora do normal). Foram achados conjuntos com desempenho melhor que o do nível O3 para 55 dos 61 programas.

O espaço gerado pelo algoritmo aleatório é representado pela Figura 3.

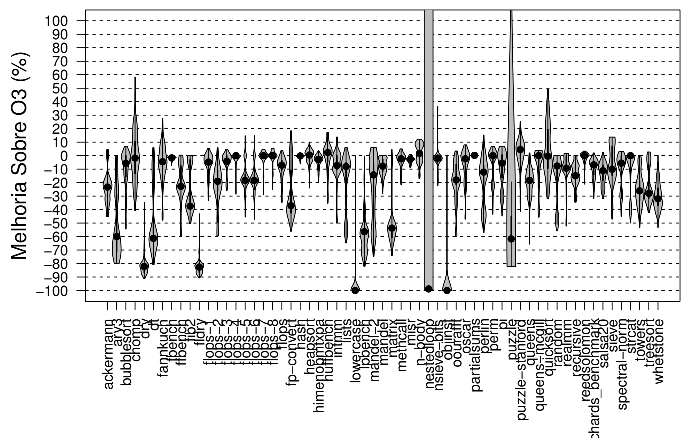

Figura 3. Espaço de busca reduzido criado pelo algoritmo aleatório

D. *Simulated Annealing*

Apesar de ser uma meta-heurística muito utilizada e geralmente bem sucedida, o número de bons casos encontrados foi pequeno. Porém, levando em consideração a quantidade de programas para os quais foram encontrados conjuntos que trazem um ganho de desempenho maior que 20%, esta foi a melhor abordagem. Isso abre margens para a investigação de seu uso na compilação iterativa. Este espaço é representado pela Figura 4

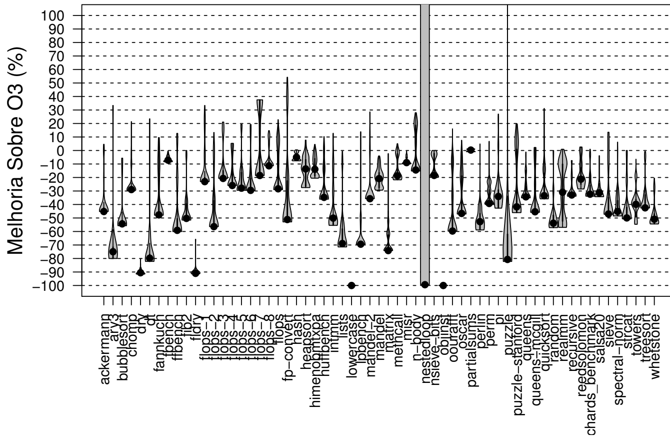

Figura 4. Espaço de busca reduzido criado pelo *Simulated Annealing*

Uma característica que essa abordagem apresenta diferente das demais é que a maior parte dos conjuntos encontrados estão concentrados nas áreas de degradação. Porém ainda assim, a busca conseguiu encontrar conjuntos com um bom desempenho.

Observa-se que as medianas estão quase todas próximas a área de pior desempenho. Porém, para o programa *fp-convert* foi encontrado um conjunto com ganho de 54,22%. Além disso, o total de programas os quais foram encontrados conjuntos que obtiveram melhorias acima do nível O3 foi o menor dentre todos os outros encontrados, totalizando 47 dos 61.

Isso pode ter ocorrido pois a função de perturbação não utiliza uma abordagem muito agressiva. Diferente do algoritmo genético que altera uma grande parte da solução e do algoritmo aleatório que "refaz"a solução, o *Simulated Annealing* altera somente uma otimização por vez.

Isso faz com que seja feita uma busca entre os vizinhos, explorando melhor as propriedades daquela região. Porém, para uma quantidade maior de programas não foram encontrados bons casos, pois foi uma exploração que não buscou em todas partes do espaço, devido a condição de parada permitir poucas iterações e a função de perturbação não mudar muito a solução.

E. *A União de Todos os Espaços*

O espaço composto pela união todos não conseguiu alcançar desempenho acima do nível O3 para todos os programas. Porém a base mostra-se a mais balanceada. A Figura 5 ilustra esse espaço.

Como já esperado, o melhor conjunto achado prove um ganho de 58,33% sobre o nível de otimização O3, que é o

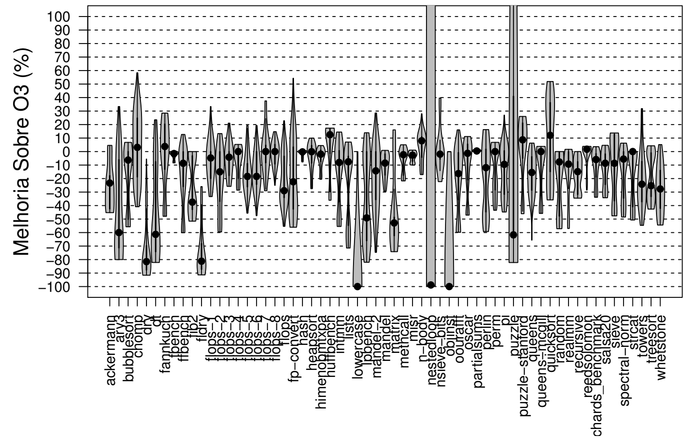

Figura 5. Espaço de busca reduzido criado pela união dos demais espaços

conjunto encontrado pela estratégia aleatória. Sendo o total de programas, para os quais o desempenho está acima do nível de otimização O3, de 57 dos 61, ou seja, o algoritmo genético com seleção por classificação já foi o suficiente para cobrir todos os programas que os demais métodos cobriram.

F. *Uma Visão Geral dos Espaços*

Pelos gráficos de violino das Figuras 1, 2, 3, 4 e 5 pode-se observar que os espaços criados pelos algoritmos genéticos alcançaram o melhor resultado (considerando os programas treino), dentre todos os métodos propostos. O algoritmo genético com seleção por torneio foi o que mais achou conjuntos acima do nível O3. Enquanto, o algoritmo genético com seleção por classificação conseguiu alcançar a maior cobertura, cobrindo todos os programas que os demais algoritmos cobriram.

Também é possível observar que, os algoritmos aleatório e *Simulated Annealing* sacrificam um pouco de cobertura para aumentar o valor das melhorias. O *Simulated Annealing* conseguiu a maior quantidade de melhorias acima de 20% e o algoritmo aleatório conseguiu a maior melhoria, sendo de 58,33%.

VI. Os Conjuntos Selecionados por Best10

Como mencionado anteriormente, o algoritmo Best10 extrai do espaço de busca reduzido, os 10 melhores conjuntos que cobrem a maioria dos programas. Esta seção tem por objetivo analisar tais conjuntos.

A. *Algoritmo Genético com Seleção por Classificação*

Os 10 conjuntos extraídos do espaço gerado pelo algoritmo genético com seleção por classificação apresentam mais de 15% de suas otimizações sendo otimizações para laços. A otimização eliminação de código morto aparece com frequência nestes conjuntos, tendo a otimização de eliminação de código morto agressiva (*adce*) presente em metade desses conjuntos.

Outra otimização muito frequente nos conjuntos, aparece em 9 dos 10, é a otimização *inline*, que faz com que o código do programa aumente, porém reduz o número de saltos, pois substitui a chamada da função. Outra otimização que provoca

Tabela III. CONJUNTOS EXTRAÍDOS PELO BEST10

Conjuntos extraídos do espaço criado pelo algoritmo genético com seleção por classificação
-jump-threading, -lcssa, -loop-unswitch, -lazy-value-info, -no-aa, -sccp, -loop-rotate, -basiccg, -tailcallelim, -dse, -sroa, -loop-idiom, -constmerge, -basicaa, -strip-dead-prototypes, -block-freq, -loop-simplify, -loop-vectorize, -tbaa, -inline, -globalopt, -memcpyopt, -loop-simplify, -tailcallelim, -verify, -ipsccp, -instcombine, -targetlibinfo, -adce, -indvars, -memdep, -prune-eh, -loops, -loop-unroll, -licm, -reassociate, -gvn, -scalar-evolution, -jump-threading,-domtree
-globaldce, -no-aa, -loop-rotate, -memdep, -gvn, -scalar-evolution, -loop-unroll, -memcpyopt, -prune-eh, -dse, -prune-eh, -loop-simplify,-memdep, -loop-idiom, -simplifycfg, -deadargelim, -memcpyopt, -targetlibinfo,-sccp, -loop-rotate, -prune-eh, -loop-deletion, -adce, -deadargelim, -loop-deletion, -loop-deletion, -loop-vectorize, -constmerge, -sroa, -inline, -verify, -block-freq, -slp-vectorizer, -basicaa, -lazy-value-info, -indvars,-block-freq, -functionattrs, -licm,-gvn
-loop-unroll, -strip-dead-prototypes, -lcssa, -licm, -loop-deletion, -memdep, -gvn, -ipsccp, -indvars, -notti, -scalar-evolution, -early-cse, -inline, -tailcallelim, -loops, -reassociate, -verify, -loop-rotate, -correlated-propagation,-domtree, -constmerge, -slp-vectorizer, -dse, -instcombine, -indvars, -basicaa, -basicaa, -tbaa, -domtree, -sccp, -deadargelim, -barrier, -lazy-value-info, -jump-threading, -loop-vectorize, -loop-unswitch, -simplifycfg, -loop-unroll,-instcombine, -loop-deletion
-globalopt, -tbaa, -gvn, -basiccg, -globaldce, -block-freq, -inline, -correlated-propagation, -domtree, -dse, -globalopt, -ipsccp, -sroa, -no-aa, -branch-prob, -jump-threading, -early-cse, -inline-cost, -sroa, -gvn, -no-aa, -instcombine, -basicaa, -jump-threading, -early-cse, -loop-unroll,-tailcallelim, -loop-unroll, -notti, -loop-unroll, -sccp, -deadargelim, -basiccg, -prune-eh, -loops, -inline, -licm, -memdep, -simplifycfg,-loop-rotate
-block-freq, -tbaa, -early-cse, -loops, -sroa, -targetlibinfo, -gvn, -barrier, -slp-vectorizer, -dse, -tailcallelim, -functionattrs, -lazy-value-info, -correlated-propagation, -loop-rotate, -sccp, -constmerge, -loop-deletion, -loop-vectorize, -branch-prob, -loop-vectorize, -memdep, -jump-threading, -globalopt, -loop-simplify, -indvars, -verify, -prune-eh, -slp-vectorizer, -argpromotion, -early-cse, -loop-unroll, -domtree, -licm, -tbaa, -lcssa,-dse, -notti, -instcombine, -basiccg
-functionattrs, -inline, -basicaa, -licm, -loop-unroll, -correlated-propagation, -memdep, -jump-threading, -sroa, -loop-unswitch, -loop-idiom, -memcpyopt, -instcombine, -scalar-evolution, -domtree, -dse, -lcssa, -loop-rotate, -constmerge, -adce, -verify, -block-freq, -licm, -tailcallelim, -loop-deletion, -deadargelim, -lazy-value-info, -inline-cost, -tbaa, -basiccg, -branch-prob, -notti, -strip-dead-prototypes, -simplifycfg, -reassociate, -prune-eh, -globaldce, -slp-vectorizer, -barrier, -loops
-basiccg, -loop-unroll, -simplifycfg, -dse, -tailcallelim, -notti, -notti, -sroa, -correlated-propagation, -inline, -loop-rotate, -lazy-value-info, -early-cse, -instcombine, -domtree, -simplifycfg, -tbaa, -loop-unswitch, -verify, -block-freq, -loop-deletion, -correlated-propagation, -memcpyopt, -scalar-evolution, -loop-unroll, -tailcallelim, -ipsccp, -loops, -memcpyopt, -functionattrs, -tbaa, -lcssa, -slp-vectorizer, -reassociate, -inline-cost, -argpromotion, -jump-threading, -loop-unroll, -gvn, -no-aa
-inline, -globalopt, -memcpyopt, -branch-prob, -strip-dead-prototypes, -sroa, -gvn, -notti, -no-aa, -instcombine, -sccp, -dse, -correlated-propagation, -inline, -adce, -dse, -inline-cost, -basicaa, -barrier, -loop-rotate, -argpromotion, -lcssa, -basiccg, -loop-unswitch, -memcpyopt, -ipsccp, -branch-prob, -lazy-value-info, -licm, -globaldce, -functionattrs, -constmerge, -indvars, -deadargelim, -simplifycfg, -basicaa, -loop-vectorize, -early-cse, -domtree, -reassociate
-inline, -globalopt, -deadargelim, -licm, -lazy-value-info, -gvn, -adce, -barrier, -basicaa, -loop-rotate, -loops, -branch-prob, -slp-vectorizer, -verify, -sroa, -sccp, -memdep, -early-cse, -loop-vectorize, -strip-dead-prototypes, -loop-rotate, -licm, -correlated-propagation, -tbaa, -inline, -ipsccp, -dse, -targetlibinfo, -loop-unswitch, -constmerge, -dse, -sroa, -loop-idiom, -scalar-evolution, -slp-vectorizer, -early-cse, -loop-unroll, -memcpyopt, -instcombine, -barrier
-loops, -domtree, -indvars, -gvn, -basiccg, -tbaa, -sroa, -no-aa, -lcssa, -sroa, -correlated-propagation, -lcssa, -targetlibinfo, -jump-threading, -inline-cost, -strip-dead-prototypes, -notti, -instcombine, -licm, -loop-vectorize, -loops, -tailcallelim, -inline-cost, -functionattrs, -loop-rotate, -lazy-value-info, -loop-unroll, -loop-unswitch, -targetlibinfo, -branch-prob, -memdep, -block-freq, -inline-cost, -lazy-value-info, -inline, -reassociate, -slp-vectorizer, -deadargelim, -loop-deletion, -argpromotion
Conjuntos extraídos do espaço gerado pela união de todos espaços
-inline, -gvn, -loop-rotate, -sroa, -adce, -loop-vectorize, -correlated-propagation, -memdep, -loop-idiom, -basicaa, -notti, -inline-cost, -deadargelim, -memcpyopt, -loop-unswitch, -reassociate, -lazy-value-info, -scalar-evolution, -slp-vectorizer, -constmerge, -loop-unroll, -simplifycfg, -barrier, -indvars, -loop-simplify, -loop-deletion, -globalopt, -block-freq, -argpromotion, -jump-threading, -tbaa, -tailcallelim, -domtree, -dse, -inline, -no-aa, -loop-vectorize, -licm, -ipsccp, -lcssa
-loop-simplify, -jump-threading, -jump-threading, -inline, -memdep, -no-aa, -reassociate, -loop-unswitch, -memcpyopt, -sroa, -gvn, -sroa, -barrier, -deadargelim, -constmerge, -adce, -memcpyopt, -adce, -licm, -loop-rotate, -indvars, -lcssa, -branch-prob, -loop-vectorize, -instcombine, -loops, -slp-vectorizer, -loop-unroll, -block-freq, -loop-unroll, -verify, -simplifycfg, -notti, -loop-unroll, -verify, -prune-eh, -functionattrs, -loop-deletion, -notti, -ipsccp
-globalopt, -tbaa, -gvn, -basiccg, -globaldce, -block-freq, -inline, -correlated-propagation, -domtree, -dse, -globalopt, -ipsccp, -sroa, -no-aa, -branch-prob, -jump-threading, -early-cse, -inline-cost, -sroa, -gvn, -no-aa, -instcombine, -basicaa, -jump-threading, -early-cse, -loop-unroll, -tailcallelim, -loop-unroll, -notti, -loop-unroll, -sccp, -deadargelim, -basiccg, -prune-eh, -loops, -inline, -licm, -memdep, -simplifycfg, -loop-rotate
-scalar-evolution, -verify, -adce, -loop-unroll, -loop-unswitch, -simplifycfg, -simplifycfg, -loop-unswitch, -basicaa, -lazy-value-info, -block-freq, -ipsccp, -strip-dead-prototypes, -tbaa, -reassociate, -notti, -indvars, -notti, -globalopt, -loop-rotate, -adce, -loop-rotate, -loop-deletion, -loop-unswitch, -loop-deletion, -constmerge, -deadargelim, -slp-vectorizer, -sroa, -memcpyopt, -domtree, -barrier, -tailcallelim, -memdep, -early-cse, -loop-vectorize, -loop-idiom, -instcombine, -loop-deletion, -dse
-targetlibinfo, -loop-simplify, -sroa, -strip-dead-prototypes, -memdep, -lcssa, -adce, -licm, -dse, -memcpyopt, -sccp, -scalar-evolution, -lazy-value-info, -loop-vectorize, -ipsccp, -basicaa, -loop-idiom, -globalopt, -notti, -block-freq, -jump-threading, -loop-unroll, -basiccg, -loop-deletion, -tailcallelim, -verify, -inline, -loops, -loop-unswitch, -no-aa, -loop-rotate, -argpromotion, -domtree, -barrier, -tbaa, -inline-cost, -indvars, -reassociate, -branch-prob, -constmerge
-instcombine, -argpromotion, -domtree, -functionattrs, -globalopt, -dse, -correlated-propagation, -sroa, -lcssa, -loop-unswitch, -simplifycfg, -deadargelim, -loop-rotate, -scalar-evolution, -strip-dead-prototypes, -inline, -sccp, -loop-unroll, -instcombine, -no-aa, -licm, -loops, -barrier, -ipsccp, -tailcallelim, -prune-eh, -loop-vectorize, -indvars, -loop-deletion, -memdep, -notti, -constmerge, -loop-idiom, -basicaa, -adce, -jump-threading, -memcpyopt, -reassociate, -tbaa, -indvars
-verify, -globalopt, -branch-prob, -inline, -reassociate, -gvn, -instcombine, -gvn, -basiccg, -loop-unroll, -ipsccp, -no-aa, -loop-deletion, -correlated-propagation, -prune-eh, -prune-eh, -loop-rotate, -deadargelim, -licm, -simplifycfg, -basicaa, -loop-unswitch, -domtree, -slp-vectorizer, -instcombine, -basicaa, -early-cse, -globaldce, -tailcallelim, -memcpyopt, -simplifycfg, -jump-threading, -sccp, -loops, -constmerge, -memdep, -loop-vectorize, -lazy-value-info, -indvars, -loop-vectorize
-basiccg, -loop-unroll, -simplifycfg, -dse, -tailcallelim, -notti, -notti, -sroa, -correlated-propagation, -inline, -loop-rotate, -lazy-value-info, -early-cse, -instcombine, -domtree, -simplifycfg, -tbaa, -loop-unswitch, -verify, -block-freq, -loop-deletion, -correlated-propagation, -memcpyopt, -scalar-evolution, -loop-unroll, -tailcallelim, -ipsccp, -loops, -memcpyopt, -functionattrs, -tbaa, -lcssa, -slp-vectorizer, -reassociate, -inline-cost, -argpromotion, -jump-threading, -loop-unroll, -gvn, -no-aa
-loops, -domtree, -indvars, -gvn, -basiccg, -tbaa, -sroa, -no-aa, -lcssa, -sroa, -correlated-propagation, -lcssa, -targetlibinfo, -jump-threading, -inline-cost, -strip-dead-prototypes, -notti, -instcombine, -licm, -loop-vectorize, -loops, -tailcallelim, -inline-cost, -functionattrs, -loop-rotate, -lazy-value-info, -loop-unroll, -loop-unswitch, -targetlibinfo, -branch-prob, -memdep, -block-freq, -inline-cost, -lazy-value-info, -inline, -reassociate, -slp-vectorizer, -deadargelim, -loop-deletion, -argpromotion
-targetlibinfo, -loop-simplify, -sroa, -strip-dead-prototypes, -memdep, -lcssa, -adce, -licm, -dse, -memcpyopt, -sccp, -scalar-evolution, -lazy-value-info, -loop-vectorize, -ipsccp, -basicaa, -loop-idiom, -globalopt, -notti, -block-freq, -jump-threading, -loop-unroll, -basiccg, -loop-deletion, -tailcallelim, -verify, -inline, -loops, -loop-unswitch, -no-aa, -loop-rotate, -argpromotion, -domtree, -barrier, -tbaa, -inline-cost, -indvars, -reassociate, -branch-prob, -constmerge

diversas modificações no código é a de simplificação no grafo de fluxo de controle (*simplifycfg*) que merge blocos básicos e elimina código morto [31]. Esta otimização aparece em 6 dos 10 conjuntos.

A Tabela III apresenta os conjuntos extraídos por Best10. Por motivo de restrição de espaço não são apresentados os conjuntos extraídos de todos os espaços criados.

B. Algoritmo Genético com Seleção por Torneio

Os 10 conjuntos extraídos do espaço criado pelo algoritmo genético com seleção por torneio tem menores quantidade de otimizações aplicadas em laços, do que os extraídos do espaço gerado pelo algoritmo genético com seletor por classificação. Porém, todos estes conjuntos apresentam a otimização *inline*.

Dos 10 conjuntos, 7 apresentam a otimização *adce*. Outro detalhe importante é que a otimização *simplifycfg* aparece em mais conjuntos (8 dos 10). Isso mostra que neste caso, Best10 selecionou alguns conjuntos mais agressivos, do que aqueles selecionados do espaço criado pelo algoritmo genético com seleção por classificação.

C. Algoritmo Aleatório

Todos conjuntos do espaço gerado aleatoriamente também possuem mais de 15% de suas otimizações sendo otimizações aplicadas a laços.

Dos 10 conjuntos selecionados por Best10, 9 possuem a otimização *inline*, e apenas 6 possuem a otimização *simplifycfg*. Assim como a otimização *inline*, a otimização *adce* também está em 9 conjuntos.

D. Simulated Annealing

Este espaço é onde o Best10 selecionou os conjuntos com maior número de otimizações dedicadas a laços, em alguns casos alcançando um quarto das otimizações do conjunto.

As otimizações *inline*, *simplifycfg* e *adce* aparecem em 9, 8 e 8 conjuntos respectivamente. Isso mostra que os conjuntos selecionados pelo Best10 também foram bem agressivos.

E. Todos

Diferente do esperado, apenas 8 dos 10 conjuntos do espaço formado pela união de todos os outros estão também entre os selecionados pelo Best10 em outro espaço.

Dos conjuntos selecionados, 2 não estão entre os 10 melhores de nenhuma das bases, 2 pertencem aos 10 melhores do *Simulated Annealing*, 3 pertencem ao espaço do algoritmo genético com seleção por classificação e 3 pertencem a base do algoritmo genético com seleção por torneio. Nenhum dos conjuntos deste espaço está entre os escolhidos pelo Best10, quando utilizado o espaço criado pela estratégia aleatória.

Ainda assim, a quantidade de conjuntos com a otimização *inline* é considerável, pois são 9 dos 10 conjuntos que a apresentam. A otimização *simplifycfg* está presente em 7 conjuntos, e a *adce* está em 6 conjuntos, como pode ser visto na Tabela III.

VII. Resultados Experimentais

A. Ambiente Experimental

O ambiente para a execução dos experimentos foi o seguinte:

- **Hardware**: A máquina usada era composta por um processador Intel Core I7-3779, 8 MB de memória cache, e 8 GB memória de RAM.
- **Sistema Operacional**: O sistema operacional usado foi a distribuição Ubuntu 14.04 com a versão do *kernel* 3.13.0-37-generic.
- **Sistema de Compilação**: Foi utilizada a LLVM 3.5 [31] como o sistema de compilação.
- ***Baseline***: Como *baseline* foi usado o nível de otimização -O3 da LLVM. O *baseline* indica o desempenho ao qual se deseja superar, com a mitigação do PSO. Este é o valor base utilizado para calcular o ganho com os novos conjuntos de otimizações.
- **Validação**: Todos os resultados apresentados são a média aritmética de 5 execuções, excluindo a pior e a melhor. A execução foi feita de maneira sequencial sem interferência externa.
- **Programas Teste**: Para avaliar o desempenho de Best10, mediante diferentes espaços de busca foram utilizados 3 diferentes *benchmarks*. Dois deles com aplicações de menor porte, cBench e PolyBench, e um com aplicações mais complexas, SPEC CPU2006.
- **Métricas**: A avaliação utiliza três métricas, a saber: (1) MG representando a média geral; (2) MGE a média das melhorias dos programas cobertos; (3) e NPM o número de programas que obtiveram desempenho. Além disto, a análise apenas considera desempenho os ganhos superiores ou igual a 1%. Valores entre 1% e 0% são considerados como desempenho similar ao *baseline*, enquanto valores menores que 0% são considerados como perda de desempenho.
- **Desempenho**: As melhorias no desempenho são calculadas de acordo com a equação 2:

$$Melhoria = (\frac{InstrO3}{InstrConj} - 1) * 100 \qquad (2)$$

B. O Desempenho utilizando Diferentes Espaços

Os resultados obtidos são apresentados nas Figuras 6, 7, 8, 9 e 10. Com base nos dados obtidos podemos chegar as seguintes conclusões para cada um dos espaços de busca reduzidos:

1) Algoritmo Genético com Seleção por Classificação: O espaço criado por este algoritmo genético se mostrou eficiente tanto em melhoria quanto em cobertura. A cobertura do espaço não foi tão ampla quanto a do espaço aleatório, porém, ainda assim foi possível cobrir 49 dos 80 programas. Além disso, o espaço apresenta uma melhoria de 9,63% para os programas que conseguiu cobrir. E a média de melhoria foi 4,86%.

2) Algoritmo Genético com Seleção por Torneio: O espaço criado pelo algoritmo genético com seleção por torneio teve uma cobertura baixa, cobrindo apenas 46 dos 80 programas. Porém, o desempenho para os programas cobertos foi significativo, alcançando uma melhoria de 9,98%. A média geral de melhoria chegou a 5,09%. Porém, não ultrapassou a abordagem aleatória nem em melhoria nem em cobertura.

3) Espaço Aleatório: Apesar de não esperado, o espaço gerado aleatoriamente mostrou-se o mais eficiente, conseguindo inclusive o melhor desempenho geral entre os 3 *benchmarks*. A cobertura, foi a maior de todas as estratégias, conseguindo alcançar desempenho para 52 dos 80 programas. A melhoria alcançada para apenas os programas cobertos foi de 9,24%. A melhoria média geral foi de 5,48%. Assim, surpreendentemente, esse foi o melhor método utilizado para a criação dos espaços reduzidos.

4) Simulated Annealing: O espaço gerado pela estratégia *Simulated Annealing* não se mostrou eficiente no quesito de cobertura, cobrindo apenas 45 dos 80 programas. Este espaço teve um desempenho ruim, em especial para o *benchmark* PolyBench, quando comparado aos outros espaços. Se levado em consideração somente os programas cobertos, a média da melhoria de desempenho foi de 9,88%, para todos *benchmarks*. Esse espaço também teve o pior desempenho dentre o ganho de

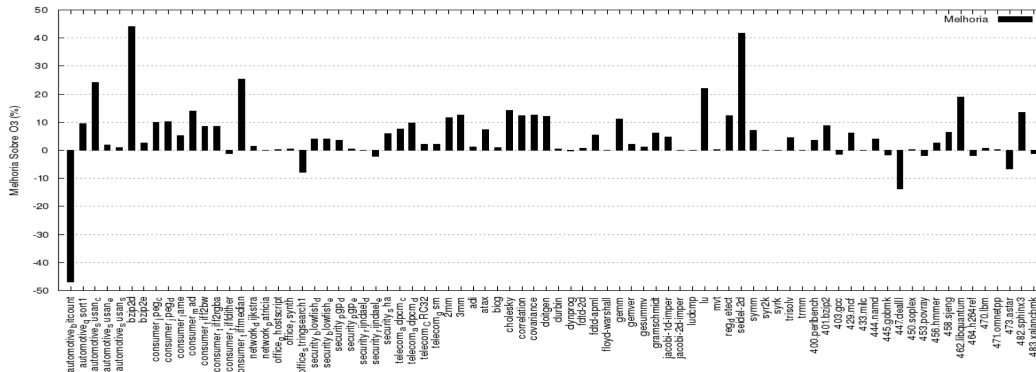

Figura 6. Resultados obtidos para o espaço gerado pelo Algoritmo Genético com classificação

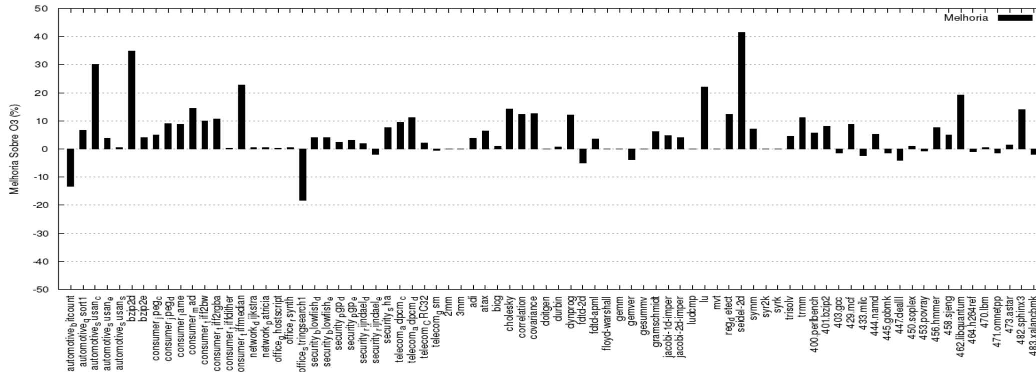

Figura 7. Resultados obtidos para o espaço gerado pelo Algoritmo Genético com torneio

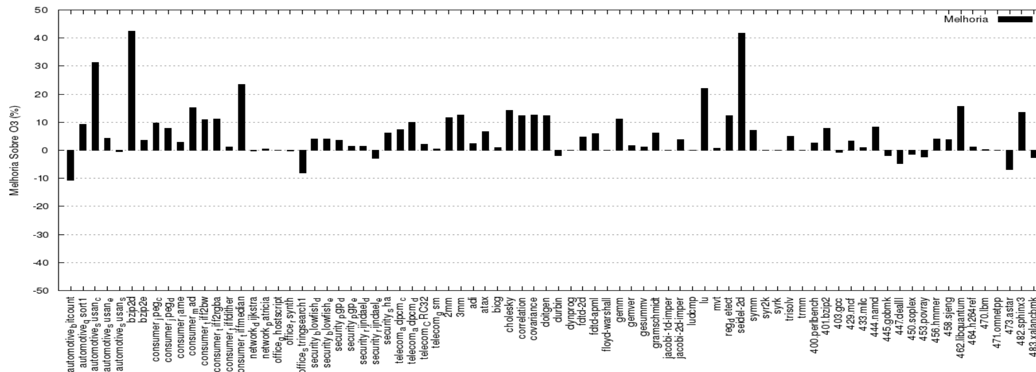

Figura 8. Resultados obtidos para o espaço gerado aleatoriamente

desempenho geral, alcançando a média de 3,84% de melhoria por programa.

5) União dos Espaços: Pode-se observar que o espaço composto pela união de todos os espaços não obteve o melhor desempenho. Isso vai contra a intuição, afinal o espaço composto por todos conjuntos tem também o melhor conjunto. Porém, deve ser observado que os conjuntos selecionados por Best10 levam em consideração somente os programas treino (*microkernel*). Assim, o melhor conjunto para os programas treino, nem sempre será o melhor para os demais programas. Mas ainda assim, existe a possibilidade de que seja um bom conjunto para outros programas. O total de programas cobertos pelo neste caso foi o menor dentre todos os métodos avaliados, sendo um total de 43 de 80. Isso ocorre pois várias melhorias estão entre 0% e 1%. A média de melhoria para os 80 programas é de 4,92%, e a melhoria para os programas cobertos é de 10.29%, que é a melhor dentre todas as avaliadas.

C. Comparação Entre as Estratégias

A Tabela IV apresenta um sumário dos resultados.

Os resultados apresentados na Tabela IV, comprovam o que foi observado anteriormente. O espaço criado pela abor-

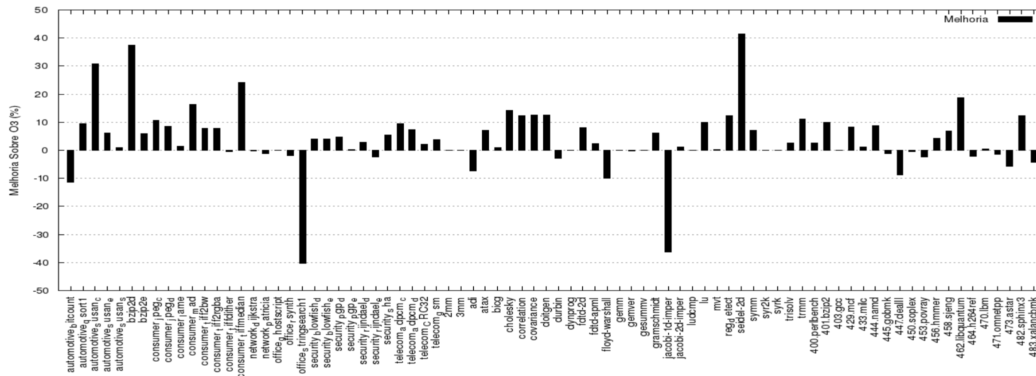

Figura 9. Resultados obtidos para o espaço gerado pelo Simulated Annealing

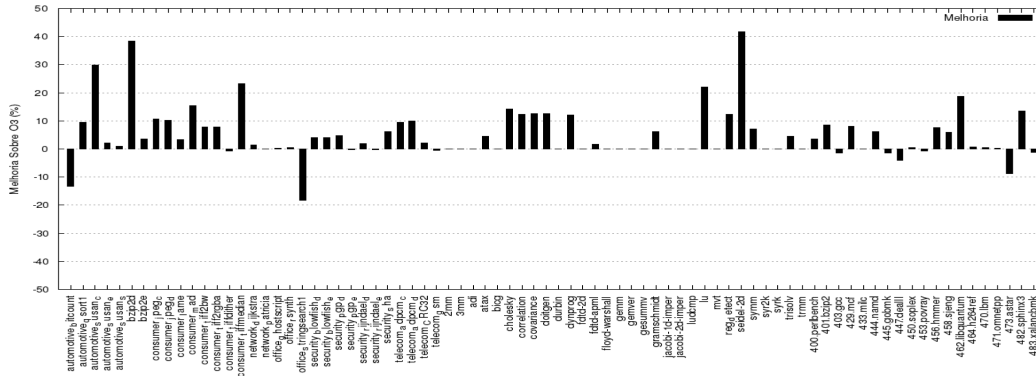

Figura 10. Resultados obtidos para a união de todos os espaços

Tabela IV. RESUMO DE DESEMPENHO PARA OS ESPAÇOS REDUZIDOS

	Classificação	Torneio	Aleatório	Simulated	Todos
cBbench					
MG	4,7805	5,618	6,1663	4,9778	5,6182
MGE	9,3452	9,8142	9,668	10,0318	9,4089
NPM	22	21	22	21	22
PolyBench					
MG	6,8215	5,7099	6,8994	3,5222	5,4554
MGE	10,6484	11,1365	10,3507	10,7365	12,5805
NPM	19	16	20	15	13
SPEC CPU2006					
MG	1,9330	3,2839	2,1226	2,4897	2,9433
MGE	8,0297	8,3455	6,1130	8,1039	8,9964
NPM	8	9	10	9	8

dagem aleatória obteve os melhores resultados. O uso deste espaço, por Best10, obteve a maior cobertura e para todos os *benchmarks*, e melhor desempenho geral para os *benchmarks* cBench e PolyBench.

O melhor desempenho geral para o *benchmark* SPEC CPU2006 foi alcançado pelo algoritmo de torneio, o que mostra que encontrar a maior parte de bons conjuntos pode trazer melhores resultados, quando os programas treino são simples que os programas teste. Estes resultados indicam que é crítico obter desempenho para aplicações complexas.

VIII. CONCLUSÕES E TRABALHOS FUTUROS

Nesse trabalho investigamos a influência de diferentes espaços de busca reduzidos, utilizados pelo algoritmo Best10 para mitigar o PSO. Este trabalho avaliou distintamente as estratégias propostas por Purini e Jain, além de uma nova estratégia.

Os experimentos realizados indicam que nem sempre, estratégias sofisticadas irão gerar um desempenho superior aquele obtido pelo maior nível de otimização do compilador em questão.

Pode-se observar também que a estratégia mais simples de geração do espaço de busca foi a que trouxe o melhor desempenho. De fato isto não era o esperado. Contudo, isto indica que estratégias simples não devem ser subestimadas, e aplicá-las pode ser vantajoso.

Podemos ainda observar que tanto a cobertura quanto a melhoria para os programas testes se tornam menores quando os programas são mais complexos. Isso pode indicar que o espaço de busca reduzido talvez não seja representativo o suficiente para cobrir os diferentes tipos de programas. Desta forma, é necessário a utilização de aplicações complexas na criação do espaço de busca reduzido.

Os ganhos obtidos nos experimentos foram consideráveis, atingindo melhorias na média de 5,48%. Além disto, estes foram superiores ao obtido pelo nível de otimização O3 para 65% dos programas.

Fica evidenciado que embora Best10 seja um dos melhores algoritmos para mitigação do PSO, o seu desempenho está relacionado diretamente a qualidade do espaço de busca.

Para trabalhos futuros serão criados espaços de busca reduzidos utilizando não somente programas *microkernel*, mas também aplicações complexas. Isso irá aumentar o custo da criação de tais espaços, mas provavelmente trará bons resultados para as aplicações mais complexas, as quais tiveram menores desempenho e cobertura. Além disto, pretendemos desenvolver uma nova versão de Best10 que considere a aplicação de diferentes conjuntos de otimizações, para diferentes partes do mesmo programa teste.

REFERÊNCIAS

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and tools*. Prentice Hall, 2006.

[2] Y. N. Srikant and P. Shankar, *The Compiler Design Handbook: Optimizations and Machine Code Generation*, 2nd ed. Boca Raton, FL, USA: CRC Press, Inc., 2007.

[3] K. Cooper and L. Torczon, *Engineering a Compiler*, 2nd ed. USA: Morgan Kaufmann, 2011.

[4] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[5] E. D. Lima, T. C. de Souza Xavier, A. F. da Silva, and L. B. Ruiz, "Compiling for Performance and Power Efficiency," in *In Proceedings of the International Workshop onPower and Timing Modeling, Optimization and Simulation*, Sept 2013, pp. 142–149.

[6] R. M. Stallman and DeveloperCommunity, "GCC, the GNU Compiler Collection," https://gcc.gnu.org/. Data de acesso: Janeiro, 20 - 2015.

[7] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *In Proceedings of the International Symposium on Code Generation and Optimization*, Palo Alto, California, Mar 2004.

[8] Open64 Developer Community, "x86 Open64 Compiler Suite," http://developer.amd.com/tools-and-sdks/cpu-development/x86-open64-compiler-suite/. Data de acesso: Maio, 12 - 2015.

[9] PathScale Inc, "EKOpath," http://www.pathscale.com/ekopath.html. Data de acesso: Maio, 12 - 2015.

[10] E. Park, S. Kulkarni, and J. Cavazos, "An Evaluation of Different Modeling Techniques for Iterative Compilation," in *In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY, USA: ACM, 2011, pp. 65–74.

[11] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle, "Iterative Compilation," in *In Proceedings of the Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation*. London, UK, UK: Springer-Verlag, 2002, pp. 171–187.

[12] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, and H. A. G. Wijshoff, "Iterative Compilation in Program Optimization," in *In Proceedings in Compiler for Parallel Computers*, 2000, pp. 35–44.

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

[14] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, "Exhaustive Optimization Phase Order Space Exploration," in *In Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 306–318.

[15] S. Kulkarni and J. Cavazos, "Mitigating the Compiler Optimization Phase-ordering Problem Using Machine Learning," in *In Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. New York, NY, USA: ACM, 2012, pp. 147–162.

[16] S. Purini and L. Jain, "Finding Good Optimization Sequences Covering Program Space," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 1–23, Jan. 2013.

[17] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using Machine Learning to Focus Iterative Optimization," in *In Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 295–305.

[18] J. Thomson, M. O'Boyle, G. Fursin, and B. Franke, "Reducing Training Time in a One-shot Machine Learning-Based Compiler," in *In Proceedings of the International Conference on Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 399–407.

[19] M. R. Jantz and P. A. Kulkarni, "Performance Potential of Optimization Phase Selection During Dynamic JIT Compilation," in *In Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2013, pp. 131–142.

[20] A. H. Mahdi, M. A. Kalil, and A. Mitschele-Thiel, "Dynamic Packet Length Control for Cognitive Radio Networks," in *In Proceedings of the Vehicular Technology Conference*, Sept 2013, pp. 1–5.

[21] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[22] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[23] T. Dean, J. Allen, and Y. Aloimonos, *Artificial Intelligence: Theory and Practice*. Addison-Wesley Publishing Company, 1995.

[24] J. Hromkovi, *Algorithmics for Hard Problems (Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics)*. Springer, 2001.

[25] M. M. Keikha, "Improved Simulated Annealing Using Momentum Terms," *Second International Conference on Intelligent Systems, Modelling and Simulation*, pp. 44–48, Jan. 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5730318

[26] S. Long and M. O'Boyle, "Adaptive java optimisation using instance-based learning," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ser. ICS '04. New York, NY, USA: ACM, 2004, pp. 237–246. [Online]. Available: http://doi.acm.org/10.1145/1006209.1006243

[27] E. Park, J. Cavazos, and M. A. Alvarez, "Using Graph-based Program Characterization for Predictive Modeling," in *In Proceedings of the Tenth International Symposium on Code Generation and Optimization*. New York, NY, USA: ACM, 2012, pp. 196–206.

[28] Y.-Q. Zhou and N.-W. Lin, "A Study on Optimizing Execution Time and Code Size in Iterative Compilation," *Third International Conference on Innovations in Bio-Inspired Computing and Applications*, pp. 104–109, Sep. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6337646

[29] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for Reduced Code Space Using Genetic Algorithms," in *In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*. New York, NY, USA: ACM, 1999, pp. 1–9.

[30] M. Chakraborty and U. Chakraborty, "An analysis of linear ranking and binary tournament selection in genetic algorithms," in *Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on*, vol. 1, Sep 1997, pp. 407–411 vol.1.

[31] LLVM Team, "The LLVM Compiler Infrastructure," 2015, http://llvm.org. Access: Janeiro, 20 - 2015.