

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Henrique Araújo Lima

**Algoritmos Genéticos aplicados na exploração de
sequências de otimização do compilador**

Uberlândia, Brasil

2017

Resumo

Os compiladores modernos possuem diversos passos de otimizações, mas escolher uma sequência boa não é uma tarefa fácil, pois além da vasta quantidade, a ordem de aplicação de cada passo de otimização também influencia no desempenho final. No entanto, fazer uma busca exaustiva nesse espaço de busca não é uma tarefa viável. Por essa razão há diversas pesquisas que visam reduzir esse espaço de busca sem ter que analisá-lo por completo. Neste trabalho, os Algoritmos Genéticos foram escolhidos para realizar uma busca direcionada para boas soluções e de forma geral obtiveram um melhor desempenho em termos de *speedup* médio se comparado aos níveis padrões de otimização (-Ox) e também em relação à abordagem totalmente aleatória. O *speedup* médio em relação aos níveis padrões de otimizações foi de 1,09 e de 1,20 em relação à abordagem totalmente aleatória.

Palavras-chave: Algoritmos Genéticos, seleção e ordenação dos passos de otimização, exploração do espaço de projeto, compiladores otimizantes.

Lista de ilustrações

Figura 1 – Estrutura de um compilador otimizante.	13
Figura 2 – Exemplo de otimização.	14
Figura 3 – Esquematização do refinamento do espaço de busca para o PSO.	15
Figura 4 – Ambiente de compilação. (MARTINS et al., 2016)	18
Figura 5 – Fluxograma do funcionamento geral de um AG.	21
Figura 6 – Exemplo de População inicial do AG.	22
Figura 7 – Seleção baseada em torneio de 2.	23
Figura 8 – Cruzamento simples de 1 ponto.	23
Figura 9 – Exemplo de mutação.	23
Figura 10 – Reinscrição baseada nos melhores indivíduos.	24
Figura 11 – Ambiente de compilação.	29
Figura 12 – Representação cromossômica de 2 indivíduos da população do AG. . .	30
Figura 13 – Exemplo do operador de crossover. Em a) os indivíduos pais b) os indivíduos filhos resultantes do crossover.	37
Figura 14 – Exemplo de mutação: inserção de um passo de otimização.	38
Figura 15 – Exemplo de mutação de remoção de um passo de otimização. Em a) o indivíduo antes da mutação, já em b) o mesmo indivíduo após a mutação com $p = 1$	38
Figura 16 – Exemplo de mutação: troca entre genes.	39
Figura 17 – Exemplo de mutação: substituição de gene.	39
Figura 18 – Gráfico comparando os <i>speedups</i> médios de cada valor do parâmetro de <i>crossover</i>	43
Figura 19 – Gráfico comparando os <i>speedups</i> médios de cada valor do parâmetro de mutação.	43
Figura 20 – Gráfico comparando os <i>speedups</i> médios de cada valor do parâmetro de tamanho da população.	44
Figura 21 – Gráfico comparando os <i>speedups</i> médios da melhor configuração global com as melhores configurações locais.	44
Figura 22 – Gráfico comparando os <i>speedups</i> médios de cada valor do parâmetro de método de seleção.	46
Figura 23 – Gráfico comparando os <i>speedups</i> médios de cada valor do parâmetro de método de seleção.	48

Lista de tabelas

Tabela 1 – Conjunto de <i>benchmarks</i> da TI.	30
Tabela 2 – Lista de passos de otimização utilizados pelo -O2.	33
Tabela 3 – Lista de passos de otimização utilizados pelo -O3.	34
Tabela 4 – Lista de todos os passos de otimização disponíveis.	35
Tabela 5 – Resultado das formas de inicialização	41
Tabela 6 – Lista de operadores e valores testados.	42
Tabela 7 – Configuração de avaliação da taxa de mutação.	42
Tabela 8 – Tabela consolidando os resultados de cada parâmetro.	45
Tabela 9 – Tabela comparativa entre o melhor de cada parâmetro individual e o melhor geral.	47
Tabela 10 – Tabela comparativa do desempenho do AG com o algoritmo aleatório.	47

Lista de abreviaturas e siglas

Lista de Acrônimos

AG	Algoritmos Genéticos
AST	Asymmetric Synthetic Tree
DSE	Design Space Exploration
IR	Representação Intermediária
JIT	Just in Time
LLVM	Low Level Virtual Machine
ML	Machine Learning
PSO	Problema de Seleção de Otimizações

Sumário

1	INTRODUÇÃO	7
1.1	Justificativa	8
1.2	Objetivos	9
1.3	Organização do texto	10
2	PROBLEMA DA OTIMIZAÇÃO DE CÓDIGOS	11
2.1	Compiladores	11
2.1.1	Etapas da Compilação	11
2.1.2	Otimizações	12
2.2	Problema de Seleção e Ordenação dos Passos de Otimização	14
3	EXPLORAÇÃO DO ESPAÇO DE PROJETO	17
3.1	Algoritmos Genéticos	18
3.1.1	Seleção Natural	19
3.1.2	Funcionamento	20
4	TRABALHOS CORRELATOS	25
5	MODELO PROPOSTO	29
5.1	Ambiente de Desenvolvimento	29
5.2	Representação do Indivíduo	30
5.3	Geração da População Inicial	32
5.3.1	Geração dos indivíduos	32
5.4	Métodos de Seleção	34
5.4.1	Roleta	34
5.5	Cruzamento de indivíduos	36
5.6	Mutação	36
5.6.1	Inclusão de um novo passo de otimização	37
5.6.2	Remoção de um passo de otimização	37
5.6.3	Troca entre genes	38
5.6.4	Substituição de gene	39
5.7	Reinserção e critério de parada	40
6	EXPERIMENTOS	41

6.1	Configuração do AG	41
6.1.1	Análise Comparativa	47
7	CONCLUSÕES E TRABALHOS FUTUROS	49
	REFERÊNCIAS	51

1 Introdução

O desempenho de uma tarefa computacional está diretamente relacionado com a qualidade do código executado (MARTINS et al., 2016). Geralmente, os códigos dos sistemas computacionais são escritos em linguagens de programação de alto nível, como, por exemplo, Java e C. Esses códigos são então traduzidos pelos compiladores em códigos de máquina para a arquitetura alvo da aplicação (PUNTAMBEKAR, 2009). Nesse processo, um dos grandes objetivos do compilador é a otimização de código, ou seja, a geração de um código mais eficiente segundo uma métrica. Dentre os objetivos, o mais largamente investigado é o que visa melhorias no tempo de execução (MARTINS et al., 2016; AGAKOV et al., 2006; COOPER; SCHIELKE; SUBRAMANIAN, 1999). Porém, a otimização pode ser empregada visando diversos objetivos, tais como: redução do código compilado (DEBRAY et al., 2000), melhoria no tempo de execução (COOPER; SCHIELKE; SUBRAMANIAN, 1999) e redução do consumo de energia (DUBACH; JONES; O'BOYLE, 2012). Portanto, os compiladores exercem um papel importante no desempenho do código compilado, bem como na sua qualidade.

A fim de alcançar melhorias no desempenho do código, os compiladores realizam variadas tarefas de otimização (análises e transformações), as quais são denominadas passos de otimização (AHO; SETHI; ULLMAN, 1986). Nos compiladores atuais são utilizados níveis padrões de otimizações (-Ox), os quais aplicam um conjunto, pré-estabelecido, de passos de otimizações em uma ordem fixa, sobre o código fonte (JUNIOR; ANDERSON, 2016). Os conjuntos adotados em cada nível, bem como sua ordem de aplicação são escolhidos pelos desenvolvedores dos compiladores a fim de melhorar o desempenho de um grupo representativo de programas. No entanto, eles não conseguem garantir a otimização de qualquer programa, assim como não garantem que a otimização proposta é a mais eficiente para um código fonte específico (MARTINS et al., 2016). Essa situação é mais crítica para aplicações em sistemas embarcados e de alto desempenho cujas limitações de recursos e a necessidade por códigos eficientes (alto desempenho) impõem requisitos que nem sempre são atingidos pelos níveis de otimização do compilador. Nesses sistemas, é essencial a busca por sequências de otimizações específicas para cada código considerando a arquitetura onde a aplicação será executada, a fim de se alcançar os melhores desempenhos possíveis.

Existe uma vasta quantidade de passos de otimizações, os quais podem variar de acordo com o compilador. A escolha de quais otimizações utilizar e sua ordem de aplicação

pode ter um impacto significativo no desempenho do código alvo (ALMAGOR et al., 2004; HOSTE; EECKHOUT, 2008). Dessa forma, o espaço de soluções (sequências de otimizações possíveis) é muito grande, tornando a busca exaustiva inviável (MARTINS et al., 2016). A escolha manual de uma sequência de otimização boa é árdua mesmo para profissionais experientes (MARTINS et al., 2016). Uma forma de contornar esse problema é utilizar uma técnica computacional para conseguir uma boa solução sem ter que analisar todo o espaço de busca. O uso de heurísticas é indicado, pois elas direcionam para regiões com boas soluções. Neste contexto, o uso de Algoritmos Genéticos (AGs) vêm sendo amplamente utilizado na busca de boas sequências de otimizações (COOPER; SCHIELKE; SUBRAMANIAN, 1999; AGAKOV et al., 2006; MARTINS et al., 2016).

Neste trabalho, foi desenvolvida uma abordagem baseada em AG para a exploração de boas sequências de otimização para códigos-fonte específicos. Inicialmente, será investigada a melhor configuração dos parâmetros do AG para um ambiente de exploração baseado no compilador LLVM e no processador LEON3. Em seguida, será feita uma análise comparativa entre o desempenho obtido pelo modelo desenvolvido e uma abordagem totalmente aleatória, visando determinar a eficiência do uso do Algoritmo Genético.

1.1 Justificativa

Os compiladores modernos geralmente utilizam níveis de otimização, que visam a melhoria do desempenho de um grupo representativo de programas (MARTINS et al., 2016). Entretanto, estes níveis de otimização não garantem a otimização de qualquer programa, nem, tão pouco, que o código gerado seja o mais eficiente. Além disso, sabe-se que, aplicações em sistemas embarcados são ainda mais críticas, pois há limitações de recursos, por exemplo, capacidade de memória e processamento e, neste contexto, é importante que os códigos tenham os melhores desempenhos possíveis.

Como já visto, a busca exaustiva pela melhor sequência é impraticável e a seleção manual, mesmo realizada por um analista experiente, é uma tarefa laboriosa e demorada. Assim, faz-se inerente a utilização de métodos heurísticos que direcionam a otimização para uma boa região do espaço de soluções, como os Algoritmos Genéticos.

Existem vários trabalhos na literatura que empregam AG na exploração de sequências de otimização (COOPER; SCHIELKE; SUBRAMANIAN, 1999) (AGAKOV et al., 2006) (MARTINS et al., 2016). No entanto, cada um utiliza uma configuração diferente e, geralmente, não são apresentadas justificativas para os valores adotados para os parâmetros do AG. Portanto, neste trabalho, avaliamos diferentes configurações do AG em um ambiente de compilação baseado no LLVM e considerando como arquitetura alvo um

processador para sistemas embarcados, LEON3.

1.2 Objetivos

O presente estudo possui como principal objetivo a seleção de boas sequências de otimizações, a fim de melhorar o desempenho dos códigos compilados, em relação aos níveis de otimização do compilador. Para alcançar este objetivo, foi implementada uma abordagem de busca do espaço de soluções (sequência de otimização) baseada em Algoritmos Genéticos.

Além do objetivo geral, também buscamos atender os seguintes objetivos específicos:

- Estudar, instalar e adaptar o ambiente de compilação escolhido às necessidades do nosso modelo de exploração.
- Encontrar sequências de otimização capazes de gerar um código alvo que possua um tempo de execução inferior àquele atingido pelo código original.
- Garantir que as sequências de otimização geradas pelo modelo, resultem em códigos alvo otimizados equivalentes semanticamente aos códigos originais.
- Encontrar sequências de otimizações que resultem em desempenhos ao menos similares aos dos demais modelos avaliados.
- Superar o desempenho dos níveis de otimizações padrões do compilador (-Ox), a partir das sequências selecionadas.
- Propor um modelo em que a execução não comprometa a qualidade do código gerado.
- Desenvolver um modelo que não exija qualquer conceito prévio acerca dos passos de otimizações, bem como a ordem de execução, etapa de aplicação e a relação que esses passos possuem entre si.
- Aplicar diferentes configurações de AG e analisar qual delas possui o melhor desempenho.
- Analisar o impacto da aplicação desta abordagem para um conjunto de códigos (*benchmarks*) bastante utilizado em aplicações de sistemas embarcados.

1.3 Organização do texto

O restante do texto está organizado da seguinte maneira:

- Capítulo 2: possui uma introdução ao problema da otimização de códigos, envolvendo os conceitos de compiladores, otimização e do problema da seleção e ordenação dos passos de otimização, propriamente dito.
- Capítulo 3: possui uma introdução ao conceito de exploração do espaço de projeto (DSE), bem como, a técnica de busca utilizada para o DSE neste trabalho.
- Capítulo 4: contém uma descrição sucinta de alguns trabalhos sobre o tema que dados durante o levantamento bibliográfico.
- Capítulo 5: contém uma descrição do modelo implementado, apresentando a estrutura geral do ambiente de compilação e detalhando as estratégias utilizadas no módulo de exploração baseado em AG.
- Capítulo 6: contém uma descrição dos experimentos realizados para encontrar a melhor configuração do AG e avaliar a eficiência do modelo, bem como uma discussão sobre os resultados obtidos.
- Capítulo 7: possui uma síntese das principais conclusões sobre o trabalho, e sugere possíveis ações que podem ser realizadas para dar continuidade à investigação e melhorar o modelo proposto.

2 Problema da Otimização de Códigos

Este capítulo apresenta uma visão geral acerca dos principais conceitos que permeiam o problema de otimização de códigos: compiladores, otimização e o problema de seleção e ordenação dos passos de otimização.

2.1 Compiladores

Um compilador é um programa de computador que recebe um programa fonte escrito em uma linguagem e gera/traduz outro programa - programa alvo - semanticamente equivalente, geralmente escrito em uma outra linguagem (ex: linguagem de máquina) (ULLMAN; AHO; SETHI, 1995). Neste processo de tradução, o compilador usualmente informa, por *logs*, ao programador se houve algum erro de compilação.

Os compiladores podem ser otimizantes ou não. Um compilador otimizante é aquele que aplica transformações no código fonte para gerar um código mais eficiente segundo algum critério/métrica (ULLMAN; AHO; SETHI, 1995). Neste estudo, estamos trabalhando apenas com os compiladores otimizantes.

2.1.1 Etapas da Compilação

Comumente, os compiladores transformam um código escrito em linguagem de alto nível para um em linguagem de baixo nível. Entretanto também existem compiladores (*source-to-source*) que traduzem um código escrito em uma linguagem de alto nível para outra, também de nível elevado (ULLMAN; AHO; SETHI, 1995). Essa transformação, de um código fonte em um código alvo, é mais conhecida como compilação. Durante o processo de compilação de um código, o compilador realiza uma série de etapas até gerar o código final.

A compilação de um código consiste em quatro etapas principais:

- **Análise Léxica:** divide o código fonte em pedaços menores, chamados de *tokens* e, após ter a lista de *tokens*, verifica se eles fazem parte da linguagem fonte. Caso algum *token* não pertença à linguagem, uma mensagem de erro é emitido ao programador e a compilação é abortada. Caso contrário, a lista de *tokens* é enviada à próxima etapa (AHO; SETHI; ULLMAN, 1986).

- **Análise Sintática:** nesta etapa, o compilador processa os tokens recebidos do analisador léxico e gera uma representação intermediária, denominada árvore sintática assimétrica (AST). Caso haja sequências de tokens não reconhecidas, o compilador acusará um erro sintático (AHO; SETHI; ULLMAN, 1986).
- **Análise Semântica:** esta etapa é responsável por verificar se o código está de acordo com as regras semânticas, ou seja, se o código é coerente. Um exemplo é a verificação dos tipos de expressões e variáveis. Um erro nessa fase é denominado erro semântico (ULLMAN; AHO; SETHI, 1995).
- **Geração de código alvo:** responsável por transformar a representação intermediária em um código de equivalente escrito na linguagem desejada (MARTINS et al., 2016). Normalmente, esta etapa visa gerar o código em linguagens de máquina de acordo com a arquitetura em que se pretende executá-lo.

O compilador divide essas 4 etapas em dois módulos principais: o *front-end* e o *back-end*. O módulo *front-end* é responsável pelas etapas de análise léxica, sintática e semântica e pela geração da representação intermediária (IR). Já o módulo de *back-end* possui algumas etapas que podem variar de acordo com o compilador, no entanto, a única etapa obrigatória é a geração do código alvo. Como pode ser observado, *front-end* está relacionado com a linguagem de programação usada, enquanto o *back-end* é dependente da arquitetura alvo. O compilador pode ainda possuir um módulo intermediário, o *middle-end*, que é responsável por aplicar transformações no código de representação intermediária, para produzir um código mais eficiente (HOLMES, 1995). Diferente das otimizações que podem ser aplicadas no *back-end*, o *middle-end* realiza transformações de alto nível, independente da linguagem de programação usada e da arquitetura alvo. Neste trabalho, utilizaremos um AG para auxiliar o *middle-end* na tarefa de localizar boas sequências de otimização. Na Figura 1 é possível visualizar os módulos de um compilador otimizante, bem como a divisão de tarefas entre cada módulo.

2.1.2 Otimizações

A otimização de um programa é o processo de transformar todo ou parte do seu código, que está em uma representação intermediária (IR), em um código semanticamente equivalente, mas com melhor eficiência. Vale ressaltar que o termo "otimizador" não deve ser compreendido no seu sentido literal, pois dado um código C , ao aplicar as transformações pelo otimizador e gerar um código C' semanticamente equivalente, não se garante que, este



Figura 1 – Estrutura de um compilador otimizador.

código C' seja mais eficiente que o código original C . Além disso, o processo de otimização não garante que o código gerado tenha o melhor desempenho possível (desempenho ótimo), uma vez que, para definir qual sequência de otimização é a ótima, deve-se avaliar todas as sequências possíveis e esta é uma tarefa impraticável para os compiladores modernos. Por isso, as pesquisas são direcionadas a encontrar boas soluções, mas não necessariamente a melhor (MARTINS et al., 2016). O processo de otimização envolve operações de análise e de transformação, denominados passos de otimização. A análise identifica oportunidades de modificações a serem realizadas, enquanto a transformação deve efetuar essas mudanças, sendo assim, um resultado direto da operação de análise (JOHNSON, 2004). O conjunto das várias análises e transformações, aplicadas em um código, é conhecido por otimização.

Uma mesma transformação pode resultar em diferentes efeitos, em duas aplicações distintas. As otimizações de laços (*loops*) costumam ser bastante eficazes e eficientes independentemente da arquitetura alvo (MARTINS et al., 2016). No entanto, não haverá ganho algum ao aplicar essa transformação em um código que não possua laços em sua estrutura. Na Figura 2 é possível ver um exemplo de otimização de código. Na Figura 2 há um fragmento do código original, composto por dois comandos de atribuição que envolvem expressões aritméticas. Nesse caso, seria interessante utilizar as propriedades algébricas de comutatividade e associatividade de adição, permitindo que o fragmento gerado seja reduzido por meio da eliminação de expressões comuns. O resultado dessa otimização é ilustrado na figura 2 b.

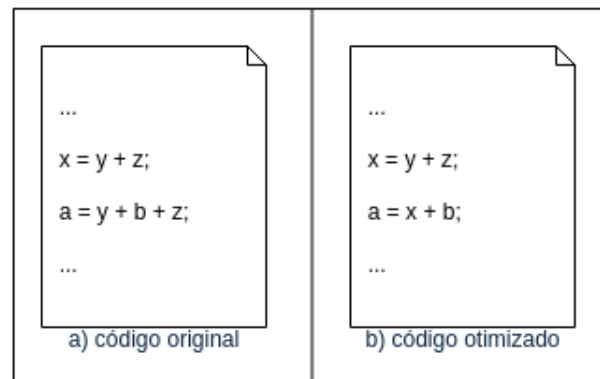


Figura 2 – Exemplo de otimização.

2.2 Problema de Seleção e Ordenação dos Passos de Otimização

Cada otimização consiste em aplicar uma série de transformações que constituem uma sequência de alterações a fim de melhorar a eficiência do código, sem sofrer alteração em sua semântica (KULKARNI; WHALLEY; TYSON, 2007). Muitos desses passos de otimizações utilizam e compartilham recursos, além de precisarem de condições específicas no código para poderem ser aplicadas. Dessa forma, os passos de otimizações interagem uns com os outros por meio da habilitação e desabilitação de oportunidades para outras otimizações serem aplicadas. Essas interações têm sido largamente investigadas (COOPER; SCHIELKE; SUBRAMANIAN, 1999), (AGAKOV et al., 2006), (JUNIOR; ANDERSON, 2016), (MARTINS et al., 2016) em diferentes compiladores, cada qual com seu conjunto particular de passos de otimizações e variadas arquiteturas.

Um requisito básico para encontrar a sequência ideal de otimização está diretamente relacionado com a capacidade do compilador em alterar a ordem de aplicação dos passos de otimização. No entanto, muitos compiladores foram projetados sem essa funcionalidade, por isso, os formatos aceitos como entrada em cada passo de otimização estão associados aos formatos de saída dos passos que os antecedem. Essa dependência entre os passos de otimização implica uma serialização, além de fixar a ordem com a qual eles serão aplicados pelo compilador. Após conscientização sobre os potenciais benefícios da aplicação em ordem diferente, muitos projetos de compiladores adotaram uma nova estratégia de compilação, onde os passos de otimização são aplicados nas representações intermediárias do código. Dessa forma, cada passo pode ser aplicado diversas vezes e em qualquer ordem, resolvendo algumas questões relacionados à ordenação (MARTINS et al., 2016).

Portanto, o problema de seleção e ordenação de otimização consiste em determinar a melhor sequência s de passos de otimizações do compilador, considerando a arquitetura desejada e um código C específico. No entanto, este problema é indecidível pois o tamanho

de s pode ser infinito, e, mesmo se delimitar o tamanho de s , encontrar essa solução ótima deixa de ser indecível, mas continua sendo intratável, pois a quantidade de possíveis soluções cresce exponencial (MARTINS et al., 2016). Dessa forma, ao invés de encontrar um s ótimo, as pesquisas tem buscado encontrar um s' bom, de modo que o desempenho do código compilador supere aquele obtido com os níveis padrões de otimizações dos compiladores (-Ox).

Não há um método que indique qual é a solução ótima s (MARTINS et al., 2016). O espaço de busca é muito grande para fazer uma busca exaustiva a fim de encontrar a melhor sequência. É fundamental adotar algum método iterativo de busca, por exemplo os Algoritmos Genéticos, capaz de explorar parcialmente esse espaço para encontrar boas soluções para o problema. Esse processo é conhecido como exploração do espaço de projeto (DSE).

Na Figura 3 é possível ver a esquematização do Problema de Seleção e Ordenação (PSO), até encontrar uma boa sequência de otimizações s'' que vai melhorar o desempenho do código fonte. Inicialmente vimos que o PSO é indecível, pois há infinitas soluções. Uma alternativa ao PSO é restringir o espaço de busca tornando-o finito. Por exemplo, limitando o tamanho das sequências de otimização. Dessa forma, buscar a melhor solução no espaço se torna decidível, no entanto, é intratável visto que dependendo do tamanho da sequência, o espaço de busca é muito grande, tornando impraticável a busca exaustiva pela solução ótima. Neste caso, são necessárias a exploração desse espaço restrito por meio de uma heurística capaz de direcionar a busca para as regiões potenciais, a fim de encontrar uma boa solução (s'') em um tempo aceitável (DSE). Por fim, a sequência s'' é aplicada ao código fonte, resultando no código alvo, do qual é esperado um desempenho próximo do ótimo.

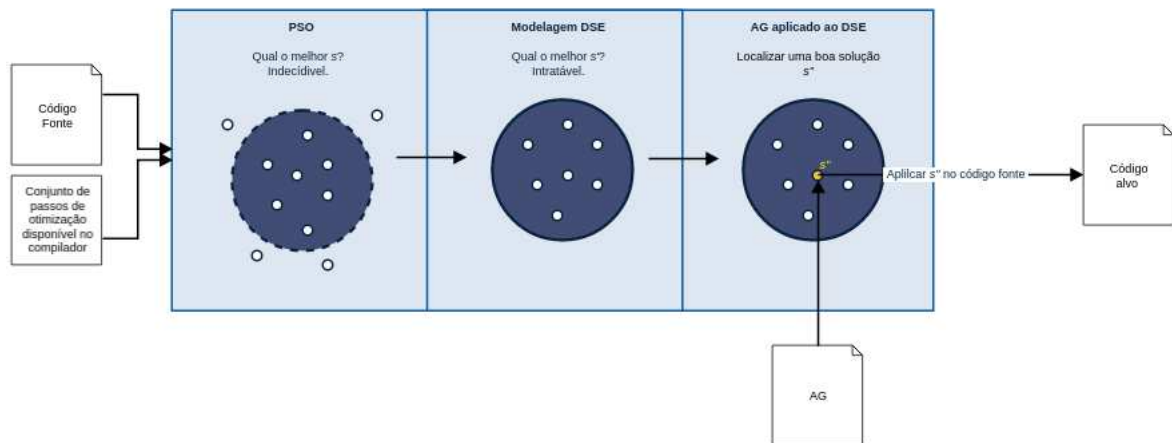


Figura 3 – Esquematização do refinamento do espaço de busca para o PSO.

3 Exploração do Espaço de Projeto

Devido à grande quantidade de passos de otimização implementada nos compiladores modernos, bem como o tamanho e complexidade dos códigos das aplicações embarcadas e os requisitos e restrições impostos ao projeto do sistema, localizar a melhor sequência de otimização do compilador para aplicações e processadores específicos é uma tarefa muito difícil, mas fundamental, em ambientes que necessitam de alto desempenho (sistemas embarcados, por exemplo). A exploração deve ser rápida e eficiente a ponto de avaliar diferentes soluções potenciais (sequências de otimização) e obter uma versão otimizada do programa fonte. Essa tarefa é conhecida como Exploração do Espaço de Projeto. No contexto estudado, o principal objetivo do DSE é encontrar uma sequência de passos de otimização que atenda aos requisitos e restrições do projeto e otimizar o tempo de execução (MARTINS et al., 2016).

O DSE é um processo iterativo que possui duas etapas básicas: a exploração do espaço de forma a cobrir uma área representativa, e a avaliação de possíveis soluções para o problema estudado, uma vez que a busca exaustiva é inviável devido ao tamanho desse espaço. A escolha do método de avaliação faz um ponderamento entre a precisão e o custo do modelo de avaliação. Já a seleção do método de busca avalia a precisão da cobertura do espaço de projeto. Essas decisões possibilitam a automação do processo de exploração (MARTINS et al., 2016).

O processo de exploração do espaço de projeto inicia-se com o modelamento do problema. Nesse processo, o espaço do projeto consiste em todas as possíveis soluções para o problema. Ao definir esse espaço, a exploração, em busca de uma solução ótima, consiste na escolha de uma solução inicial, que é aleatória. Com isso, o próximo passo é avaliar o quão adequada essa solução é. De acordo com essa avaliação, algumas características da solução são alteradas visando seu aprimoramento. A repetição desses passos caracteriza um processo de exploração iterativo do espaço de busca.

Neste trabalho, nós definimos que o tamanho máximo de cada sequência seria de 128 passos de otimização. Dessa forma, a exploração do espaço de busca consiste em encontrar qual a melhor sequência de passos de otimizações cujo tamanho pode variar de 0 (nenhum passo de otimização) até 128. Portanto, para garantir a solução ótima, teríamos que avaliar qual a melhor sequência de tamanho 0, depois de tamanho 1, até a de tamanho 128, e posteriormente, descobrir quais destas obteve o melhor desempenho. Como o compilador possui 140 passos de otimizações distintos, a quantidade total n de

sequências a serem exploradas pode ser definida por:

$$n = \sum_{i=0}^{128} 140^i \quad (3.1)$$

A Figura 4 mostra o processo de retroalimentação e busca pela solução do DSE.

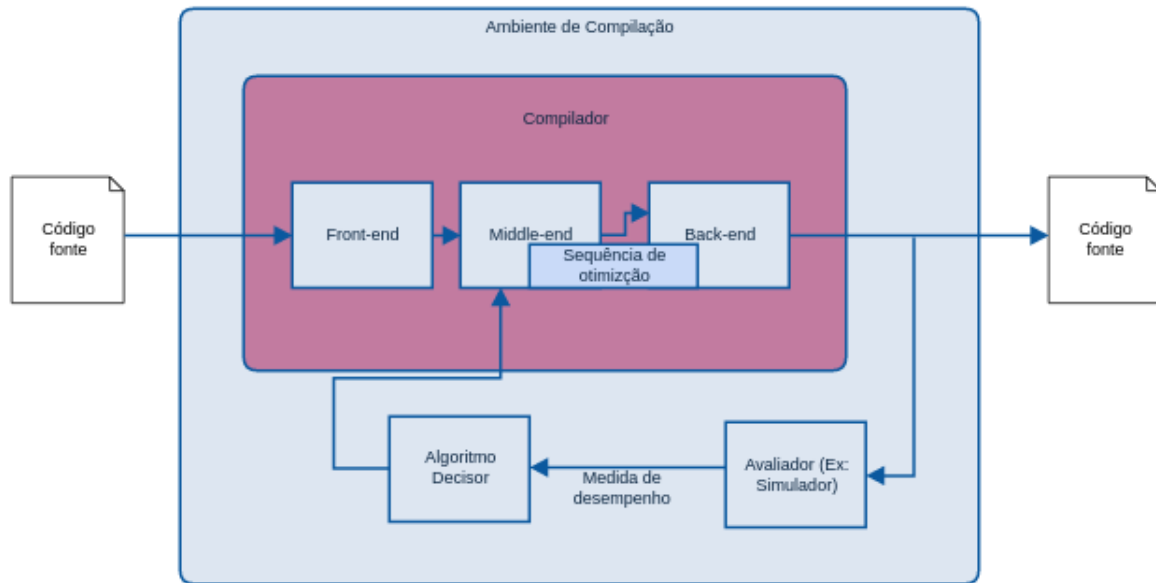


Figura 4 – Ambiente de compilação. (MARTINS et al., 2016)

3.1 Algoritmos Genéticos

Algoritmo Genético (AG) é uma técnica computacional bastante utilizada para problemas de busca e otimização combinatória. Proposto primeiramente por John Holland, este modelo de resolução de problemas teve como inspiração as teorias evolutivas da biologia, com destaque principal à teoria de seleção natural de Charles Darwin. Os Algoritmos Genéticos tem sido largamente investigados em diversas áreas do conhecimento, com os seguintes propósitos: seleção de sequências de otimizações em compiladores (COOPER; SCHIELKE; SUBRAMANIAN, 1999), (MARTINS et al., 2016); composição musical (HORNER; GOLDBERG, 1991); roteamento de pacotes em redes de computadores (ZHENG Ying; BINGXIN; ERDUN, 2001); Extração de regras para *data mining* (FIDELIS; LOPES; FREITAS, 2000); Controle de Sistemas Dinâmicos (VIEIRA, 2009).

Todas as tarefas computacionais de busca e otimizações possuem características comuns, das quais as principais são: espaço de busca e função de avaliação. O espaço de busca é aquele onde se encontram todas as possibilidades de solução para um problema específico, já a função de avaliação consiste em avaliar os integrantes do espaço de busca (GOLBERG, 1989).

Os algoritmos tradicionais de busca e otimizações (minimax, poda alfa-beta, A*, Dijkstra, Kruskal, o algoritmo do vizinho mais próximo, e algoritmo de Prim) iniciam-se com apenas um candidato, o qual é inteiramente manipulado usando heurísticas associadas ao problema. As abordagens evolucionária (Algoritmos Genéticos, Programação Genética, Estratégias Evolutivas e Programação Evolutiva) permitem processar diversas soluções em paralelo, explorando diversas regiões do espaço (GOLBERG, 1989).

A teoria da evolução concilia à genética e as ideias de seleção natural de Darwin, e esta teoria da evolução, inspirou John Holland a desenvolver as bases para os Algoritmos Genéticos. Holland publicou o seu livro *"Adaptation in Natural and Artificial Systems"* (HOLLAND, 1975), sendo um dos principais livros da área, e, a partir desse estudo inicial, diversos outros pesquisadores utilizaram essa técnica para a resolução de grandes problemas de otimizações e aprendizado de máquina (GOLBERG, 1989).

3.1.1 Seleção Natural

O processo de seleção natural, originado das teses evolutivas de Charles Darwin, justifica a capacidade, que certos seres vivos possuem, de sobrevivência em detrimento de outros. Este processo, de seleção natural, pode ser compreendido através da síntese de um exemplo clássico da população de besouros:

- Variedade de características (genes): é a diferença no material genético dos indivíduos de uma determinada população. Neste exemplo, a população é composta por besouros de coloração verde, e por besouros de coloração marrom.
- Reprodução diferenciada: é a capacidade da espécie de se reproduzir. Considerando que, enquanto os besouros verdes são alvos de pássaros, e os marrons não, os besouros verdes se reproduzem menos que os besouros marrons.
- Hereditariedade: é a capacidade dos indivíduos transmitirem parte do seu material genético aos seus descendentes. Os besouros marrons sobreviventes tendem a ter seus descendentes da mesma cor, pois há uma base genética neste traço.
- População homogênea: é quando alguma característica (gene) é predominante na população. O traço mais vantajoso neste exemplo é a coloração marrom, uma vez que

permite, aos besouros dessa cor, terem mais descendentes e, se o processo fosse contínuo, a tendência seria de ter uma população completamente composta por besouros marrons.

3.1.2 Funcionamento

Para solucionar um problema por meio de um algoritmo genético é necessário entender alguns conceitos importantes:

- **Indivíduo:** cada possível solução do problema, seja ela uma solução boa ou ruim, corresponde a um indivíduo da população.
- **População:** conjunto que representa uma amostragem de possíveis soluções do problema. Quanto mais representativa for essa amostragem, melhor será a cobertura do espaço de busca e, conseqüentemente, maiores são as chances do AG convergir para uma solução próxima da ótima.
- **Genes:** correspondem às características de uma solução do problema que, quando juntas, formam o cromossomo do indivíduo. Por exemplo, um indivíduo formado por um vetor binário de tamanho x , onde cada posição representa a ausência ou presença de um passo de otimização, possui x genes.
- **Função de aptidão (*fitness*):** o quão bom aquele indivíduo é.

O funcionamento típico de um AG é apresentado na Figura 5. Inicialmente, uma população de possíveis candidatos à solução do problema é gerada de maneira aleatória. Posteriormente, os indivíduos são submetidos a uma função de avaliação, ranqueando os indivíduos de acordo com suas características (genes). Caso algum dos indivíduos dessa população atenda há algum critério de parada do AG, então a execução é finalizada e este indivíduo é tido como solução. Do contrário, alguns indivíduos, geralmente os melhores, são selecionados para realizar o cruzamento (*crossover*). O cruzamento consiste na recombinação genética aos pares dos indivíduos selecionados, ou seja, dois indivíduos selecionados para serem pais realizam *crossover* e geram dois indivíduos filhos que compartilham características genéticas dos seus pais. Alguns filhos oriundos da etapa anterior podem ser submetidos ao operador de mutação, pela qual, o cromossomo sofre alteração. Os novos indivíduos são então avaliados e os melhores indivíduos entre pais e filhos são mantidos para a próxima geração. Esse processo de seleção até a geração da nova população é executado novamente até que o critério de parada seja atendido. Quando este critério é atendido, o melhor indivíduo da população é tido como a solução do problema.

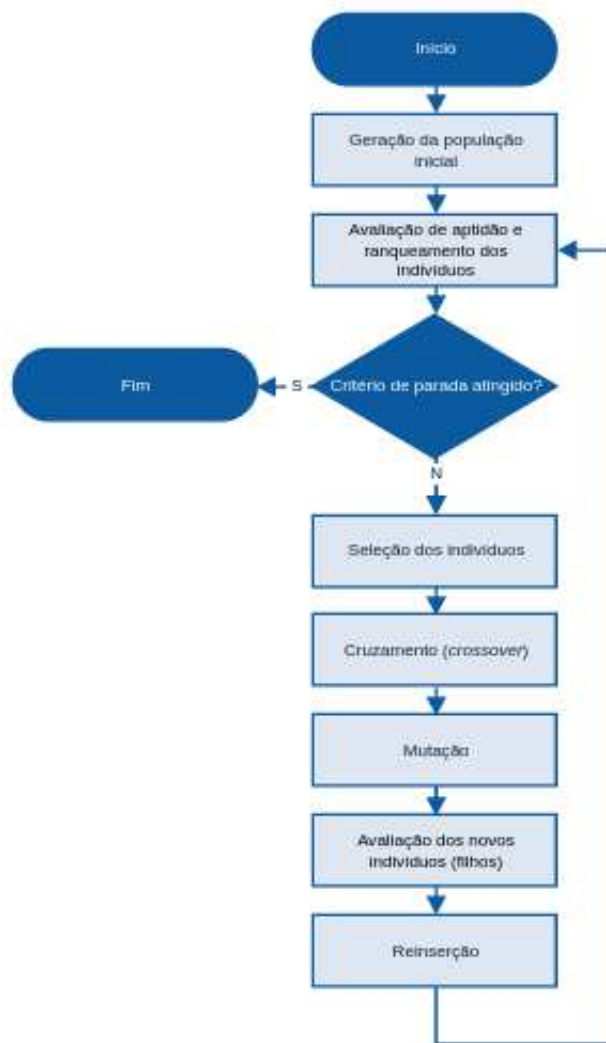


Figura 5 – Fluxograma do funcionamento geral de um AG.

Para exemplificar esse funcionamento, considere um AG para encontrar o valor x que maximize a função $f(x) = x^2$, onde x é um inteiro e $0 \leq x \leq 7$. É intuitivo que $f(x)$ assume o maior valor quando $x = 7$, e por consequência $f(7) = 49$. No entanto, é possível resolver este problema utilizando Algoritmos Genéticos.

Os indivíduos correspondem às possíveis soluções do problema que, neste caso, são todos os possíveis valores que x pode assumir ($x \in \{0, 1, 2, 3, 4, 5, 6, 7\}$). A representação do cromossomo pode ser dada de diversas maneiras. No entanto, uma das representações mais clássicas é um vetor binário. A função de aptidão pode ser a própria função $f(x)$. Considerando que o AG adota uma população de 4 indivíduos, a Figura 6 mostra um exemplo população inicial, na qual os indivíduos foram gerados aleatoriamente ($x = 0, 1, 3$ e 6). Além do valor de cada indivíduo a figura também apresenta sua representação

cromossômica e seu *fitness*.

Valor de x	Cromossomo			Fitness
0	0	0	0	0
1	0	0	1	1
3	0	1	1	9
6	1	1	0	36

Figura 6 – Exemplo de População inicial do AG.

Neste exemplo, utilizaremos duas gerações como critério de parada. Portanto, a próxima etapa é realizar a seleção dos pais e posteriormente fazer o cruzamento e a mutação. Para realizar a seleção dos pais, adotou-se o método de torneio de 2. Sorteia-se arbitrariamente dois indivíduos e então escolhe o que possui o maior *fitness* para ser o primeiro pai. Depois, o procedimento é repetido para escolher o segundo pai. Uma vez que os dois pais foram escolhidos, eles são submetidos ao cruzamento, gerando dois novos filhos. A Figura 7 mostra os indivíduos selecionados em cada torneio. A estrela indica os respectivos vencedores. Um exemplo de um cruzamento de ponto fixo sobre os dois pais selecionados pode ser visto na Figura 8.

O operador de mutação neste problema poderia consistir apenas em inverter um bit de um indivíduo. A Figura 9 ilustra essa mutação sobre um dos novos indivíduos gerados (010 que representa o 2). Nesse exemplo a mutação ocorre no terceiro gene (bit) do cromossomo. Após a mutação o indivíduo passa a representar o número 3.

Por fim, temos 6 indivíduos na população, mas queremos manter a população com o tamanho original (4 indivíduos). Então, os 4 indivíduos mais aptos (maior *fitness*) são selecionados para a nova geração. Este método de reinserção chama-se reinserção ordenada e consiste em manter os melhores indivíduos entre pais e filhos. A Figura 10 mostra que os 4 melhores indivíduos sobreviveram para a segunda geração, enquanto os dois piores foram descartados. Novamente, é feita a verificação do critério de parada e, neste caso, ela

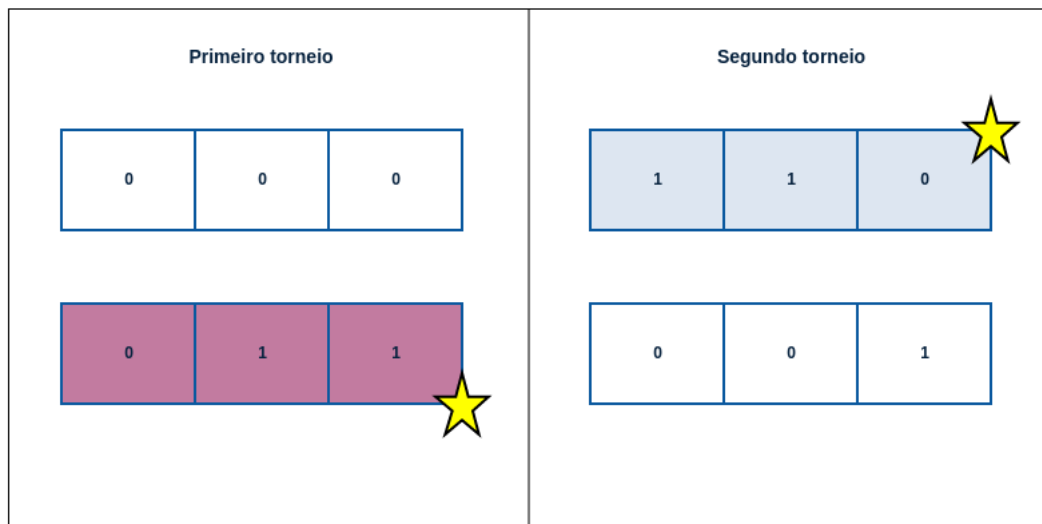


Figura 7 – Seleção baseada em torneio de 2.

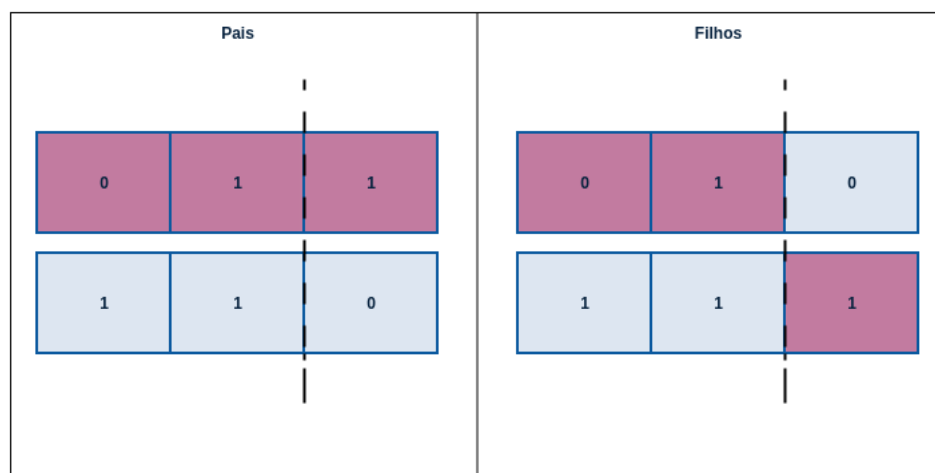


Figura 8 – Cruzamento simples de 1 ponto.

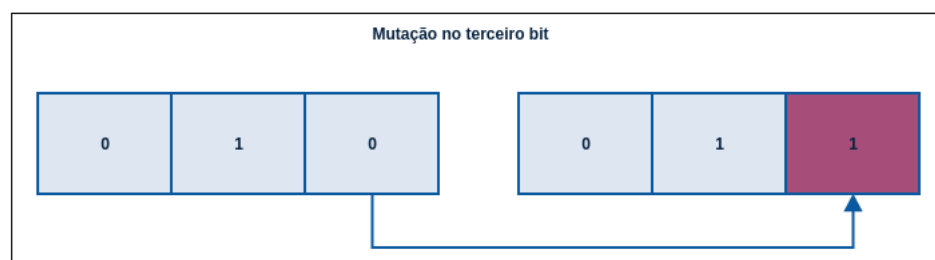


Figura 9 – Exemplo de mutação.

é bem sucedida, pois estamos considerando apenas 2 gerações. Dessa forma, o indivíduo representado por (111) e que está destacado com uma estrela na Figura 10 é a solução encontrada para o problema. Note que neste exemplo coincidiu de encontrar a solução ótima para o problema, mas nem sempre isso ocorre.

Para realizar a seleção vamos utilizar o método torneio de 2. Sorteia-se arbitrariamente dois indivíduos e então escolhe-se o que possui o maior *fitness* deles para ser o primeiro pai, depois repete o procedimento e escolhe o segundo pai. Depois que os dois pais foram escolhidos, eles são submetidos ao cruzamento, gerando dois novos filhos. A Figura 7 mostra os indivíduos coloridos e com uma estrela amarela vencendo o torneio contra os rivais. E posteriormente, o cruzamento dos dois pais selecionados pode ser visualizado na Figura 8

Valor de x	Cromossomo			Fitness
7	1	1	1	49
6	1	1	0	36
3	0	1	1	9
3	0	1	1	9
1	0	0	1	1
0	0	0	0	0

Figura 10 – Reinserção baseada nos melhores indivíduos.

4 Trabalhos Correlatos

As otimizações realizadas pelos compiladores podem visar diversas métricas e objetivos. Na literatura, são encontrados trabalhos que visam a redução do código compilado (DEBRAY et al., 2000), melhoria no tempo de execução (COOPER; SCHIELKE; SUBRAMANIAN, 1999), redução do consumo de energia (DUBACH; JONES; O'BOYLE, 2012), entre outros. Entretanto, a métrica em termos de tempo de execução vem sendo bastante pesquisada.

Embora os níveis padrões de otimizações dos compiladores satisfaçam grande parte das aplicações (KULKARNI; WHALLEY; TYSON, 2007), em sistemas embarcados e sistemas de tempo real é necessário aplicações que apresentem um alto desempenho. Por isso, o problema vem sendo amplamente estudado e os pesquisadores têm proposto diversas abordagens que adotam técnicas e metodologias diferentes para atingirem seus objetivos.

Agakov et al. (2006) trabalha com um conjunto de 14 transformações. Dessas 14 transformações ele extrai quais são relevantes para cada *benchmark* b_i e as armazena em um vetor v_i . Posteriormente, ele utiliza técnicas de aprendizado de máquina (ML - *Machine Learning*) para reduzir o espaço de busca. Essa limitação é alcançada através do algoritmo de vizinhos mais próximo, que irá dizer para cada novo código qual o *benchmark* b_i é mais similar a ele. Feito isso, ele utiliza as transformações contidas no conjunto v_i para guiar suas buscas (AG e aleatório) no DSE. Essa metodologia é capaz de identificar as regiões do espaço de soluções em que a busca deve ser focada por possuir boas soluções.

Cooper, Schielke e Subramanian (1999) consiste na exploração do espaço de busca utilizando Algoritmos Genéticos para encontrar passos de otimizações a fim de reduzir o tamanho do código compilado. Eles comparam o AG com sequências fixas e sequências aleatórias. Baseado nos resultados do AG ele gera uma nova sequência específica para o código. Esse trabalho é parecido com o nosso, no entanto, a métrica da otimização é para redução do tamanho do código compilado diferentemente do nosso que visa atender métricas de tempo de execução.

Purini e Jain (2013) utiliza-se de técnicas de sub-amostragem (*down-sampling*) para reduzir o espaço de busca. Com esta técnica ele seleciona um conjunto pequeno de sequências de otimização (aproximadamente 10) que sejam representativas para todas as classes de programa. Na fase de compilação, todas as sequências são aplicadas no novo programa e a melhor delas é escolhida.

[Junior e Anderson \(2016\)](#) Aplica as técnicas de aprendizado de máquina e posteriormente compilação interativa (AG) para encontrar passos de otimizações a fim de reduzir o tamanho do código compilado. Depois filtra-se as sequências que apresentam desempenho pior que os dos níveis de otimizações genéricos. Um modelo de ML é então criado focado na função mais custosa. Quando um novo código fonte é inserido, suas características são extraídas e com base no modelo gerado pela ML, o preditor escolhe o programa da base que mais se assemelha com o novo programa e então escolhe-se com base em fórmula algumas sequências para compor a população inicial do do AG. Com base nessa sequência gerada, é possível executar o AG. Todos os indivíduos realimentam a base de conhecimento e o melhor ao final da execução é dado como saída.

Esses trabalhos apresentados e outros estudados podem ser agrupados de acordo com suas características:

- **Formulação do problema:**

1. Somente seleção: nessa formulação, as sequências de aplicação dos passos de otimizações é fixa e tem-se apenas a definição de quais passos serão usados na otimização ([ASHOURI et al., 2014](#)).
2. Ordenação: nessa formulação, além de selecionar quais passos irão compor o conjunto das otimizações, a ordem na qual eles serão aplicados também é definida. ([COOPER; SCHIELKE; SUBRAMANIAN, 1999](#)), ([AGAKOV et al., 2006](#)), ([PURINI; JAIN, 2013](#)), ([JUNIOR; ANDERSON, 2016](#)), ([MARTINS et al., 2016](#)).

- **Compilação:**

1. Estática: nesse tipo de compilação, todo o código é compilado para somente depois ser executado ([COOPER; SCHIELKE; SUBRAMANIAN, 1999](#)), ([AGAKOV et al., 2006](#)), ([PURINI; JAIN, 2013](#)), ([ASHOURI et al., 2014](#)), ([JUNIOR; ANDERSON, 2016](#)), ([MARTINS et al., 2016](#)).
2. Dinâmica: na compilação dinâmica ou compilação JIT (*Just in Time*), o código é compilado durante a execução do programa.

- **Granularidade da otimização:**

1. Nível de programa: é a otimização que busca um melhor desempenho do programa como um todo ([COOPER; SCHIELKE; SUBRAMANIAN, 1999](#)), ([ASHOURI et al., 2014](#)), ([JUNIOR; ANDERSON, 2016](#)).

2. Nível de função: visa a otimização de uma função ou fragmento específico do código (AGAKOV et al., 2006), (MARTINS et al., 2016).
3. Classes de programas (níveis de otimizações): nessa otimização, procura-se uma sequência que seja boa para um grupo de programas. Normalmente, visa atender a um conjunto representativo de programas (PURINI; JAIN, 2013).

- **Forma de compilação:**

1. Iterativa: diversas compilações são executadas com várias sequências de otimizações diferentes, e, no final, a sequência com o melhor desempenho é a escolhida. (COOPER; SCHIELKE; SUBRAMANIAN, 1999), (PURINI; JAIN, 2013).
2. Preditivos: utiliza-se de modelos treinados previamente para escolher a sequência que maximize o desempenho, de acordo com características extraídas do código.
3. Abordagem híbrida: utiliza-se os modelos preditivos para reduzir o espaço de busca e, posteriormente, utiliza-se uma compilação iterativa com o e os modelos preditivos para reduzir o espaço de busca espaço reduzido (AGAKOV et al., 2006), (ASHOURI et al., 2014), (JUNIOR; ANDERSON, 2016)(MARTINS et al., 2016).

- **Métodos de busca:**

A principal etapa do DSE refere-se à busca. Existem diversas técnicas que são aplicadas nas buscas. Dentre elas pode-se destacar os Algoritmos Genéticos (COOPER; SCHIELKE; SUBRAMANIAN, 1999), (AGAKOV et al., 2006), (MARTINS et al., 2016) e, a busca aleatória (AGAKOV et al., 2006), (MARTINS et al., 2016), (JUNIOR; ANDERSON, 2016).

Os trabalhos relacionados encontrados na literatura ainda possuem características muito específicas, como, por exemplo, a arquitetura alvo da aplicação, o conjunto e a quantidade de *benchmarks* utilizados, o ambiente de compilação e o número de transformações utilizadas.

Neste estudo, são utilizadas as seguintes abordagens:

- Formulação: ordenação;
- Compilação: estática;
- Granularidade de Otimização: funções;

- Forma de compilação: iterativo;
- Busca: algoritmos genéticos e busca aleatória.

5 Modelo Proposto

Esta seção descreve o modelo desenvolvido durante o estudo. Uma ênfase maior será dada para o módulo de busca (DSE) em Algoritmos Genéticos, mostrando as particularidades de cada etapa: representação do indivíduo, forma de inicialização e geração da população inicial, métodos de seleção, cruzamento, mutações, reinserção dos novos indivíduos na população e critério de parada.

5.1 Ambiente de Desenvolvimento

As estratégias de exploração implementadas neste trabalho foram desenvolvidas em um ambiente de compilação integrado, o qual foi desenvolvido para um ambiente de compilação baseado no LLVM e no processador LEON3. Este trabalho foi implementado com a versão 3.7.1 do LLVM e utilizou 140 passos de otimização, os quais são listados na Tabela 1. A Figura 11 mostra o ambiente de compilação baseado em LARA usado neste trabalho.

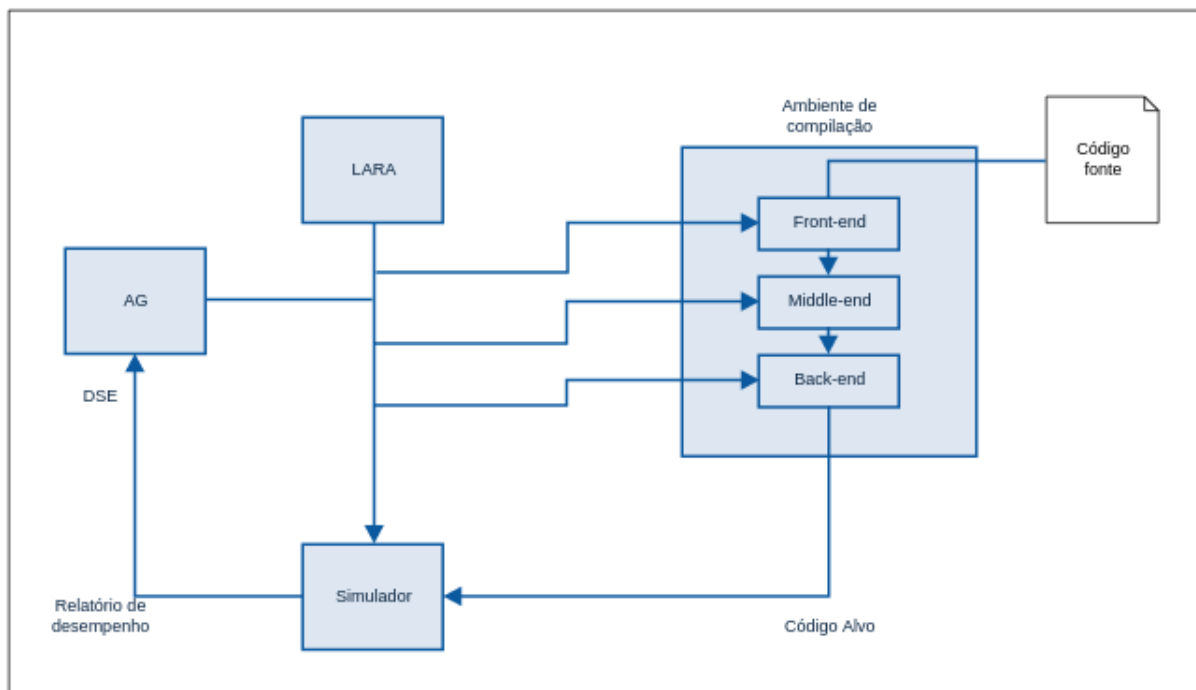


Figura 11 – Ambiente de compilação.

LARA é uma linguagem orientada a aspecto de domínio específico que implementa estratégias de DSE capazes de controlar todas as ferramentas do ambiente de compilação

Tabela 1 – Conjunto de *benchmarks* da TI.

Id	Nome	Id	Nome	Id	Nome	Id	Nome
1	DSP_autocor_c	12	DSP_mat_trans_c	23	IMG_conv_3x3_c	34	IMG_quantize_c
2	DSP_blk_eswap16_c	13	DSP_maxidx_c	24	IMG_corr_gen_c	35	IMG_sad_16x16_c
3	DSP_blk_eswap32_c	14	DSP_maxval_c	25	IMG_dilate_bin_c	36	IMG_sad_8x8_c
4	DSP_blk_eswap64_c	15	DSP_minerror_c	26	IMG_erode_bin_c	37	IMG_sobel_c
5	DSP_blk_move_c	16	DSP_minval_c	27	IMG_fdct_8x8_c	38	IMG_wave_horz_c
6	DSP_dotprod_c	17	DSP_mul32_c	28	IMG_idct_8x8_12q4_c	39	IMG_wave_vert_c
7	DSP_dotp_sqr_c	18	DSP_neg32_c	29	IMG_mad_8x8_c	40	IMG_ycbcr422p_rgb565_c
8	DSP_fir_cplx_c	19	DSP_q15tofl_c	30	IMG_median_3x3_c	41	IMG_yc_demux_be16_c
9	DSP_firlms2_c	20	DSP_vecsumsq_c	31	IMG_perimeter_c	42	IMG_yc_demux_le16_c
10	DSP_ftoq15_c	21	DSP_w_vec_c	32	IMG_pix_expand_c		
11	DSP_mat_mul_c	22	IMG_boundary_c	33	IMG_pix_sat_c		

(compiladores e simuladores) além de ser capaz de avaliar as otimizações. Um aspecto de alto nível é responsável por implementar o algoritmo de DSE de busca. Enquanto outros aspectos são responsáveis por guiar o compilador, bem como por gerar e avaliar os códigos resultantes.

O conjunto de *benchmark* utilizado para realizar a exploração foi retirado da base da TI (Texas Instrument) e constituem de conjuntos de correlação e processamento de imagens. A Tabela 4 mostra todos os *benchmarks* utilizados, bem como seu ID.

5.2 Representação do Indivíduo

O cromossomo do indivíduo corresponde a uma possível solução do problema. Isto significa que um cromossomo representa uma sequência de passos de otimizações.

O tamanho do cromossomo é variável, ou seja, diferentes indivíduos podem ter tamanhos distintos. No entanto, há um parâmetro *maxwidth*, que estabelece qual o tamanho máximo da sequência de otimização. Dessa forma, o tamanho dos indivíduos da população do AG pode variar entre 1 e o tamanho máximo *maxwidth*.



Figura 12 – Representação cromossômica de 2 indivíduos da população do AG.

O tamanho do cromossomo indica a quantidade de passos de otimização que um indivíduo possui. A Figura 12 mostra dois exemplos de indivíduos gerados pelo modelo. O

primeiro indivíduo possui 3 passos de otimização, logo o tamanho do seu cromossomo é 3. O segundo indivíduo possui um cromossomo de tamanho 7 representando uma sequência de 7 passos de otimização.

Uma vez que um indivíduo é criado, deve-se avaliar o quão bom ele é para resolver o problema em questão. Para isto, é utilizada uma função de aptidão, a qual deve ser capaz de quantificar a adequação de uma solução (indivíduo).

A função de aptidão ou simplesmente *fitness* avalia um indivíduo com base em seu cromossomo, sendo que esta avaliação consiste na aplicação da sequência de passos de otimizações no código fonte e na simulação do código otimizado gerado pelo compilador. Portanto, se tivermos dois indivíduos formados pelo mesmo conjunto de passos de otimização, mas ordenados de forma diferente (alteração na ordem de aplicação), o código gerado pelo compilador pode ser diferente, resultando em uma aptidão diferente para seus cromossomos. A eficiência do código é medida pelo número de ciclos para sua execução pelo simulador. Portanto, a otimização do código pode ser vista como um problema de minimização, no qual busca-se sequências de otimização resultem na menor latência possível, sem afetar no resultado (funcionalidade) do código original. Por isso, há a probabilidade de o indivíduo ser considerado inválido, ou seja, inadequado para a otimização de um código específico. Em nosso modelo, consideramos um indivíduo inválido é aquele que:

- Possui o tamanho do seu cromossomo maior que o tamanho máximo permitido para cromossomos, *maxwidth*, que é um parâmetro do AG.
- Se após a aplicação das otimizações, o código compilado não retornar o valor esperado, ou seja, os retornos dos códigos original e alvo serem diferentes, indicando que não são semanticamente equivalentes.
- Quando há problemas na sequência de otimização, ou seja, quando um passo de otimização não pode ser aplicado depois de algum outro passo de otimização específico, ocasionando uma mensagem de erro durante a otimização pelo *middle-end*.
- Quando o código otimizado não consegue ser compilado pelo *back-end* da arquitetura alvo.

Em todos os cenários em que os indivíduos são inválidos, a avaliação deles recebe o maior valor possível $1,7E-20$. A ideia de que os indivíduos inválidos sejam eliminados das próximas gerações durante o processo de reinserção.

Caso o indivíduo seja inválido, o tamanho do cromossomo passa a valer *maxwidth*, ou seja, o tamanho máximo permitido para aquele cromossomo. O tamanho do cromossomo

é usado como critério de desempate durante a classificação/ranqueamento dos melhores indivíduos. Assim, caso dois cromossomos tenham o mesmo *fitness*, aquele que possui a menor sequência (cromossomo com o menor tamanho) é considerado melhor.

5.3 Geração da População Inicial

A população inicial de todo AG é formada por uma quantidade pré-determinada de indivíduos, os quais são geralmente escolhidos de forma aleatória. O tamanho da população inicial é um parâmetro do AG que indica quantos indivíduos irão compor a população durante as gerações. Esse parâmetro afeta o desempenho do AG e uma escolha adequada é importante para sua eficiência. Na fase de experimentos, valores para o tamanho da população foram avaliados a fim de encontrar aquele que resultasse em boas sequências de otimização para o maior número dos códigos (*benchmarks*) testados.

5.3.1 Geração dos indivíduos

A ideia básica de um AG, para formar a população inicial de tamanho n , é escolher arbitrariamente n indivíduos que compõem todo o espaço de exploração. No entanto, para conseguirmos maior diversidade entre os indivíduos da população do AG, criamos outras formas de geração da população inicial, as quais diferenciam entre si por dois pontos principais:

- Grupo de passos de otimizações: refere-se a qual conjunto de passos de otimização que será utilizado para formar os indivíduos da população inicial. Os conjuntos podem ser: todos os passos de otimização disponibilizados pelo LLVM (Tabela 4), os passos de otimização utilizados no nível de otimização -O3 (Tabela 3) ou os passos de otimizações utilizados no nível de otimização -O2 (Tabela 2). Estes diferentes conjuntos foram avaliados durante os experimentos e os resultados indicaram que o uso do conjunto formado pelos passos de otimização do -O3 propiciou uma população inicial com menos ocorrências de indivíduos inválidos.
- Escolha do tamanho dos indivíduos: para definir o tamanho de cada indivíduo i da população inicial foram avaliadas 3 abordagens distintas de distribuição para garantir a presença de indivíduos com diferentes tamanhos na população inicial:
- maneira de escolher o tamanho dos indivíduos: para escolher o tamanho de cada indivíduo i da população inicial pode ser obtido da seguinte maneira:

Tabela 2 – Lista de passos de otimização utilizados pelo -O2.

-adce	-inline	-loops	-tailcallelim
-always-inline	-instcombine	-lower-expect	-targetlibinfo
-argpromotion	-instsimplify	-loweratomic	-no-aa
-codegenprepare	-internalize	-lowerinvoke	-tbaa
-constmerge	-ipconstprop	-lowerswitch	-basicaa
-constprop	-ipsccp	-memcpyopt	-basiccg
-correlated-propagation	-jump-threading	-mergefunc	-functionattrs
-dce	-licm	-mergereturn	-scalarrepl-ssa
-deadargelim	-loop-deletion	-partial-inliner	-domtree
-die	-loop-idiom	-prune-eh	-lazy-value-info
-dse	-loop-instsimplify	-reassociate	-lcssa
-early-cse	-loop-reduce	-scalarrepl	-scalar-evolution
-globaldce	-loop-rotate	-sccp	-memdep
-globalopt	-loop-simplify	-simplify-libcalls	-strip-dead-prototypes
-gvn	-loop-unroll	-simplifycfg	
-indvars	-loop-unswitch	-sink	

- Distribuição aleatória: escolhe-se arbitrariamente um tamanho entre 1 e o tamanho máximo que o cromossomo pode ter para cada indivíduo i da população inicial.
- Distribuição uniforme: distribui a quantidade de indivíduos da população inicial, de forma uniforme, entre diferentes faixas de tamanho do cromossomo. Como trabalhamos com sequências com até 128 passos de otimização (*maxwidth*), adotamos 12 faixas de tamanho: 10, 20, 30, e assim por diante até a faixa tamanho 120.
- Distribuição ponderada: adotou-se as mesmas faixas da distribuição anterior. No entanto, também quanto maior a faixa, maior o seu peso. Isto é, um indivíduo tem 10 vezes mais chance de ter o tamanho 100 do que ter o tamanho 10, da mesma maneira que os indivíduos tem o dobro de chances de ter seu tamanho de 40 ao invés de ter um tamanho de 20. Assim, procuramos privilegiar as sequências de otimização maiores, as quais têm se mostrado mais eficientes na otimização dos códigos.

As abordagens de geração dos indivíduos da população inicial testadas neste trabalho envolvem a combinação dessas duas configurações, sendo que essas combinações foram avaliadas durante os experimentos. A partir dos resultados experimentais, foi definido que a melhor configuração adota o -O3 como o conjunto de otimizações preferido e com o tamanho dos cromossomos sendo definido por meio da distribuição aleatória.

Tabela 3 – Lista de passos de otimização utilizados pelo -O3.

-adce	-elim-avail-extern	-loop-deletion	-sccp
-alignment-from-assumptions	-float2int	-loop-idiom	-scoped-noalias
-argpromotion	-functionattrs	-loop-rotate	-simplifycfg
-assumption-cache-tracker	-globaldce	-loop-simplify	-slp-vectorizer
-barrier	-globalopt	-loop-unroll	-sroa
-basicaa	-gvn	-loop-unswitch	-strip-dead-prototypes
-basiccg	-indvars	-loop-vectorize	-tailcallelim
-bdce	-inline	-loops	-targetlibinfo
-block-freq	-inline-cost	-lower-expect	-tbaa
-branch-prob	-instcombine	-memcpyopt	-tti
-constmerge	-ipsccp	-memdep	-verify
-correlated-propagation	-jump-threading	-mldst-motion	
-deadargelim	-lazy-value-info	-no-aa	
-domtree	-lcssa	-prune-eh	
-dse	-licm	-reassociate	
-early-cse	-loop-accesses	-scalar-evolution	

5.4 Métodos de Seleção

Em algoritmos genéticos, o método de seleção é responsável por escolher quais os indivíduos que irão propagar sua herança genética para os filhos (*crossover*). Este método geralmente prioriza os indivíduos com melhor aptidão em detrimento dos priores. Neste trabalho, foram implementados dois operadores tradicionais: torneio (*tour*) de 3 e roleta. Em um torneio de *tour de 3*, três indivíduos são sorteados aleatoriamente, e então, o indivíduo que possui a melhor aptidão, ou seja, que possui menor *fitness*, entre eles, é selecionado para o cruzamento (*crossover*).

5.4.1 Roleta

Nesse método, os indivíduos recebem "bilhetes" para concorrerem ao sorteio e, quanto maior for a quantidade de bilhetes que o indivíduo possui, mais chances ele tem de ser sorteado para realizar o *crossover*. Os indivíduos recebem a quantidade de bilhetes de acordo com sua aptidão, ou seja, a sequência que resulta na menor latência, recebe uma maior quantidade de bilhetes em relação àquela que gera um código com maior latência. Neste problema, estamos lidando com minimização, portanto o que contém o menor *fitness* é o que recebe mais bilhetes.

A seleção baseada no método da roleta pode ser realizada da seguinte forma:

1. Os indivíduos inválidos recebem 1 bilhete.

Tabela 4 – Lista de todos os passos de otimização disponíveis.

-aa-eval	-safe-stack	-mem2reg	-lcssa
-dse	-barrier	-s.-c.-o.-f.-gep	-objc-arc
-loop-ex.-single	-globalsmodref-aa	-cfl-aa	-str.-d.-proto.
-place-safep.	-loop-unswitch	-instsimplify	-dce
-adce	-sancov	-memcpyopt	-libcall-aa
-early-cse	-basicaa	-simplifycfg	-objc-arc-aa
-loop-idiom	-gvn	-codegenprepare	-strip-d.-declare
-postdomtree	-loop-vectorize	-intervals	-deadargelim
-add-dis.	-scalar-evolution	-memdep	-licm
-elim-avail-ext.	-basiccg	-sink	-objc-arc-apelim
-loop-instsimpl.	-indvars	-consthoist	-strip-nondebug
-prune-eh	-loops	-ipconstprop	-debug-aa
-alig.-f.-ass.	-scalarizer	-mergefunc	-lint
-extract-blocks	-bb-vectorize	-slp-vectorizer	-objc-arc-contrac.
-loop-interchan.	-inline	-constmerge	-structurizecfg
-reassociate	-lower-expect	-ipsccp	-delinearize
-alloca-hoisting	-scalarepl	-mergereturn	-load-combine
-flattencfg	-bdce	-slsr	-objc-arc-expand
-loop-reduce	-inline-cost	-constprop	-tailcallemim
-reg2mem	-loweratomic	-irce	-die
-always-inline	-scalarepl-ssa	-mldst-motion	-loop-accesses
-float2int	-block-freq	-spec.-execution	-pa-eval
-loop-reroll	-instcombine	-correlated-prop.	-targetlibinfo
-regions	-lowerbitsets	-iv-users	-divergence
-argpromotion	-sccp	-mod.-debuginfo	-loop-deletion
-functionattrs	-bounds-checking	-sroa	-part.-inliner
-loop-rotate	-instcount	-cost-model	-tbaa
-rewr.-sta.-for-gc	-lowerinvoke	-jump-threading	-domfrontier
-ass.-cache-track.	-scev-aa	-nary-reass.	-loop-distribute
-globaldce	-branch-prob	-strip	-part.-inl.-libcal.
-loop-simplify	-instnamer	-count-aa	-tti
-rewrite-symbols	-lowerswitch	-lazy-value-info	-domtree
-atomic-expand	-scoped-noalias	-no-aa	-loop-extract
-globalopt	-break-crit-edg.	-str.-dead-d.-info	-pl.-ba.-safe.-im.
-loop-unroll	-instrprof	-da	-verify

2. O indivíduo que possui o pior *fitness*, F_w (maior valor), e não é inválido, recebe 2 bilhetes.
3. Um indivíduo i que possui *fitness* F_i e que não se enquadra nas duas condições anteriores, receberá $F_w - F_i + 2$ bilhetes.
4. Faz-se sorteio de um número entre um e a quantidade total de bilhetes distribuídos.
5. O indivíduo que possuir o bilhete sorteado será selecionado para realizar o crossover.

5.5 Cruzamento de indivíduos

O operador de *crossover* é responsável pela busca global (entre regiões). Ele é aplicado entre dois indivíduos, que são denominados indivíduos pais. Esses pais são escolhidos, entre os indivíduos da população atual, de acordo com sua aptidão (indivíduos com melhores *fitness* tendem a ser escolhidos). O resultado da operação de *crossover* é a geração de dois novos indivíduos, que são denominados indivíduos filhos.

Neste modelo, o *crossover* entre os dois pais, f_1 e f_2 , que resulta em dois filhos, s_1 e s_2 , ocorre da seguinte maneira:

1. Sorteia-se arbitrariamente um ponto, p_1 , do cromossomo de f_1 ;
2. Sorteia-se arbitrariamente um ponto, p_2 , do cromossomo de f_2 ;
3. O filho s_1 é formado por todos os genes de f_1 que estão à esquerda de p_1 , seguidos de todos os genes de f_2 que estão à direita de p_2 .
4. O filho s_2 recebe todos os genes de f_2 que estão à esquerda de p_2 seguido de todos os genes de f_1 que estão à direita de p_1 ;

A Figura 13 exemplifica a operação de crossover. Na Figura 13a são representados os cromossomos dois pais selecionados (f_1 e f_2) e os pontos de corte sorteados aleatórios ($p_1 = 2$ e $p_2 = 1$). Já na Figura 13b é possível ver os filhos resultantes do *crossover*.

5.6 Mutação

O operador de mutação realiza pequenas modificações nos cromossomos de alguns dos filhos gerados pelo cruzamento. A quantidade de filhos afetados pela mutação é definida por um parâmetro do AG conhecido como taxa de mutação. Em nosso modelo, esse

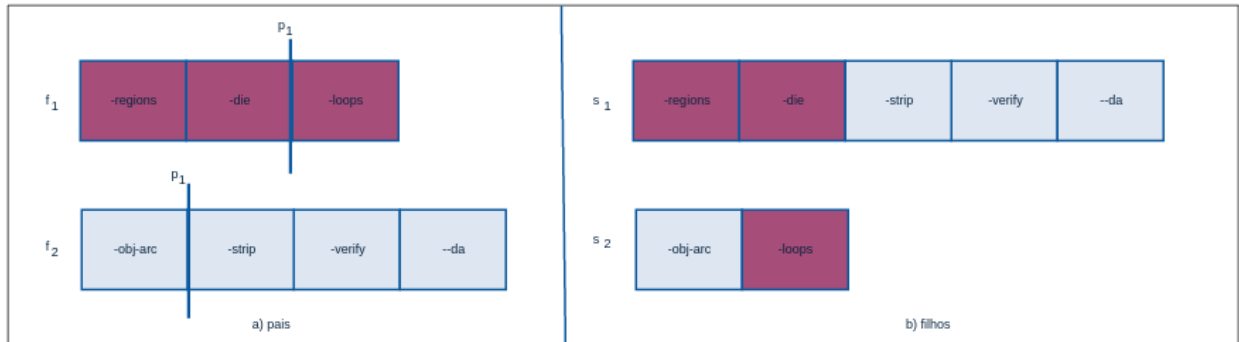


Figura 13 – Exemplo do operador de crossover. Em a) os indivíduos pais b) os indivíduos filhos resultantes do crossover.

operador faz com que o cromossomo do indivíduo possa receber, perder ou trocar genes. O operador de mutação busca garantir diversidade da população com o passar das gerações (GOLBERG, 1989). Quatro formas de mutação foram implementadas no AG, cada uma com 25% de chance de ocorrer. São elas: inclusão de um novo passo de otimização, remoção de um passo de otimização, troca entre genes e substituição de gene.

5.6.1 Inclusão de um novo passo de otimização

Neste operador de mutação, um novo passo de otimização é adicionado em algum ponto do cromossomo do indivíduo. Esse tipo de mutação envolve os seguintes passos:

1. Sorteio de um ponto p arbitrário entre 0 e o tamanho do cromossomo.
2. Seleção aleatória um passo de otimização dentre todos disponíveis no conjunto completo de passos de otimizações do LLVM;
3. Deslocamento dos passos de otimização uma posição à direita a partir de p .
4. Inclusão do novo passo de otimização na posição p .

A Figura 14 ilustra um exemplo desse tipo de mutação. A representação do cromossomo antes de sofrer a mutação é apresentada na Figura 14. A representação do mesmo indivíduo após sofrer a mutação, considerando a posição sorteada ($p = 2$) e otimização selecionado como sendo `-slsr`.

5.6.2 Remoção de um passo de otimização

Neste operador de mutação um passo de otimização é removido em algum ponto do cromossomo do indivíduo. A aplicação desse tipo de mutação envolve os seguintes passos:



Figura 14 – Exemplo de mutação: inserção de um passo de otimização.

1. Sorteio de um ponto p arbitrário entre 0 e o tamanho do cromossomo.
2. Remoção do gene que está na posição p ;
3. Deslocamento de todos os passos de otimização uma unidade para a esquerda a partir dos genes que estão à direita de p .

Um exemplo desse tipo de mutação é mostrado na Figura 15. Na Figura 15a é representado o cromossomo de um indivíduo antes de sofrer a mutação de remoção de um passo de otimização. A Figura 15b mostra a representação do mesmo indivíduo após sofrer a mutação, considerando a posição sorteada ($p = 1$).

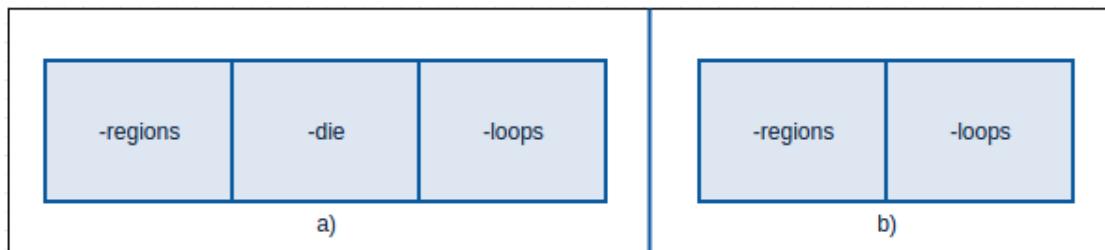


Figura 15 – Exemplo de mutação de remoção de um passo de otimização. Em a) o indivíduo antes da mutação, já em b) o mesmo indivíduo após a mutação com $p = 1$.

5.6.3 Troca entre genes

Neste operador de mutação, dois passos de otimização selecionados arbitrariamente são trocados de lugar no cromossomo do indivíduo, ou seja ocorre uma alteração na ordem de aplicação dos passos selecionados. A aplicação desse tipo de mutação envolve os seguintes passos:

1. Sorteio de um ponto p_1 arbitrário entre 0 e o tamanho do cromossomo.
2. Sorteio de um ponto p_2 arbitrário entre 0 e o tamanho do cromossomo.
3. Troca entre os genes p_1 e p_2 , ou seja, o gene que estava em p_1 vai para p_2 e vice-versa.

A Figura 16 exemplifica esse tipo de mutação. Na Figura 16a temos a representação do cromossomo de um indivíduo antes de sofrer a mutação de troca entre genes. Na Figura 16b é apresentada a representação do cromossomo do mesmo indivíduo após sofrer a mutação, considerando as posições sorteadas, $p_1 = 0$ e $p_2 = 2$.

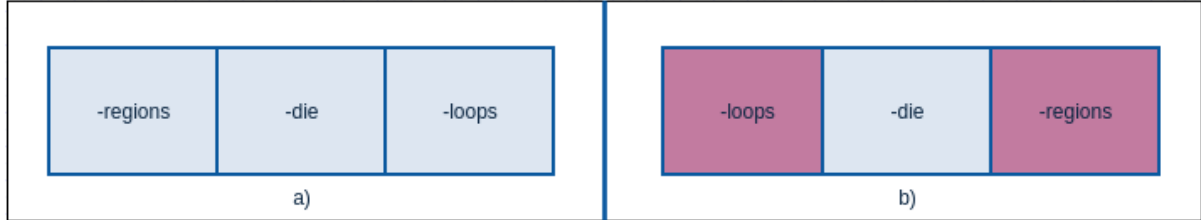


Figura 16 – Exemplo de mutação: troca entre genes.

5.6.4 Substituição de gene

Neste operador de mutação, um gene do cromossomo será alterado por outro que está no conjunto completo, ou seja, que contém todos os passos de otimizações disponíveis no compilador. Esse tipo de mutação envolve os seguintes passos:

1. Sorteio de uma posição p arbitrário entre 0 e o tamanho do cromossomo.
2. Seleção aleatória de um passo de otimização entre aqueles disponíveis no conjunto completo.
3. Troca do conteúdo do gene sorteado (posição p) pelo passo de otimização selecionado no passo anterior.

Um exemplo desse tipo de mutação é apresentado na Figura 17. A Figura 17a mostra a representação do cromossomo de um indivíduo antes de sofrer a mutação de substituição de gene. A Figura 17b mostra a representação do cromossomo do mesmo indivíduo após sofrer a mutação considerando a posição sorteada $p = 1$ e o passo de otimização selecionado como *-domtree*.

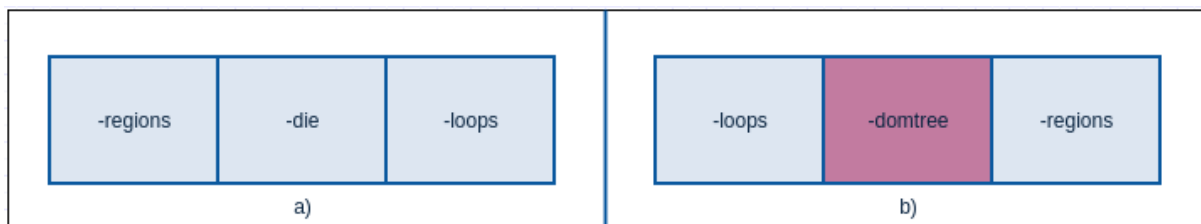


Figura 17 – Exemplo de mutação: substituição de gene.

5.7 Reinscrição e critério de parada

O método de reinscrição é usado para decidir quais indivíduos, entre pais e filhos, que irão sobreviver para a próxima geração do AG. Neste trabalho foi adotada a reinscrição dos melhores indivíduos entre pais e filhos. Por exemplo, considere um AG com população de tamanho 100 e uma taxa de *crossover* de 50%. Ao final de cada geração temos 150 indivíduos dos quais os 100 melhores indivíduos (aqueles que possuem os menores *fitness*), serão escolhidos para compor a população da próxima geração.

O critério de parada é a condição necessária para o AG encerrar sua execução. Essa condição deve ser verificada antes de evoluir uma nova geração de trabalho. O único critério de parada adotado foi o número máximo de gerações.

6 Experimentos

Inicialmente, foram executados alguns experimentos com o objetivo de identificar qual a melhor configuração para os parâmetros do Algoritmo Genético. Uma vez identificados, a eficiência do modelo configurado foi avaliada por meio de uma análise comparativa com uma abordagem baseada em amostragem aleatória, que é bastante utilizada na literatura (MARTINS et al., 2016), (COOPER; SCHIELKE; SUBRAMANIAN, 1999).

6.1 Configuração do AG

Como foi visto no capítulo 5, três formas de inicialização foram avaliados (distribuição aleatória, distribuição uniforme e distribuição ponderada). Além disso, foram testado qual conjunto de passos de otimização seria melhor (-O2, -O3, conjunto completo). Todas as variações entre as formas de inicialização e os possíveis conjuntos de otimização foram testados e a Tabela 5 mostra os resultados obtidos em cada combinação. A partir desses resultados, consideramos que a melhor configuração de inicialização para o nosso modelo é o uso do conjunto de passos do -O3 e a adoção da inicialização aleatória. Essa configuração foi escolhida por apresentar a maior quantidade de indivíduos válidos. Essa escolha foi feita em detrimento da média geométrica de *speedup*, pois a taxa de indivíduos válidos de forma geral foi muito baixa.

Tabela 5 – Resultado das formas de inicialização

Forma de inicialização	Porcentagem de indivíduos válidos (%)	Média Geométrica de speedup sobre -Ox
-O2_aleatório	23,14	0,58
-O3_aleatório	23,60	0,59
-O2_ponderada	11,10	0,73
-O3_ponderada	10,07	0,71
-O2_uniforme	9,33	0,75
-O3_uniforme	9,74	0,71
-Conj_Completo_aleatório	3,74	0,39

O Algoritmo Genético possui diversos parâmetros de configuração que podem interferir no desempenho da busca de acordo com os valores que cada um pode assumir. Assim é fundamental avaliar diversas configurações desses parâmetros a fim de determinar aquele que torna o AG mais eficiente. Neste trabalho, além dos parâmetros taxa de *crossover*, taxa de mutação e tamanho da população, também avaliamos 2 abordagens para a seleção

de indivíduos para cruzamento. A Tabela 6 mostra os valores que foram testados para cada um dos parâmetros.

Tabela 6 – Lista de operadores e valores testados.

Parâmetros	Valor		
Método de Seleção	Roleta	Tour de 3	
Taxa de crossover	90%	80%	70%
Taxa de mutação	30%	20%	10%
Tamanho da população	100	75	50

O objetivo destes experimentos é descobrir qual o melhor valor para cada parâmetro, isto é, qual configuração do parâmetro capaz de alcançar o melhor desempenho na maioria dos *benchmarks* testados.

O ideal seria fazer todas as combinações de parâmetros possíveis para descobrir qual seria a melhor configuração. No entanto, executar estes experimentos consomem um tempo elevado, inviabilizando tal avaliação dentro do prazo disponível para o estudo (TCC). Assim, a alternativa encontrada foi fixar os parâmetros no valor intermediário, e variar apenas um parâmetro por vez. Por exemplo, se o objetivo é determinar a melhor taxa de mutação, são testadas com três configurações nas quais os parâmetros taxa de *crossover* e tamanho da população e o método de seleção assumem um valor fixo único e os 3 valores da taxa de mutação são usados, como pode ser visto na Tabela 7.

Tabela 7 – Configuração de avaliação da taxa de mutação.

Nro. Configuração	Taxa de mutação	Taxa de crossover	Tamanho da população	Seleção
1	10%	80%	75	Tour de 3
2	20%	80%	75	Tour de 3
3	30%	80%	75	Tour de 3

Para cada configuração possível foram realizadas três execuções do AG com o objetivo de encontrar uma sequência de otimização capaz de melhorar o desempenho do código em relação ao -Ox. Realizadas três execuções para cada um dos 42 *benchmarks* que compõem os repositórios de *benchmarks* para processamento de imagens e DSP da *Texas Instruments*. A relação desses 42 códigos é apresentada na Tabela 1. Os resultados dos experimentos são analisados em função do *speedup* médio entre as 3 execuções. O *speedup* médio de cada execução é a razão entre a melhor latência encontrada pelo AG e a latência alcançada pelo melhor nível de otimização (-Ox).

A Figura 18 mostra o gráfico da comparação das diferentes taxa de *crossover*. Nele, é possível notar que a média geométrica dos *speedups* foram muito próximas entre si, no entanto, para a taxa de 90% esse valor foi um pouco inferior em relação às outras. Além

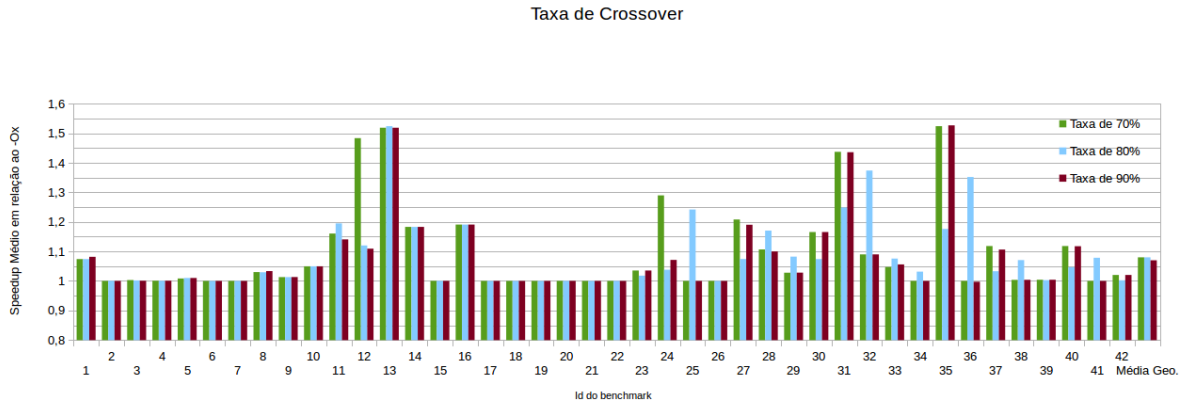


Figura 18 – Gráfico comparando os *speedups* médios de cada valor do parâmetro de *crossover*.

disso, é possível notar que a taxa mais efetiva foi a de 80% que conseguiu ser a melhor em 27 códigos. No entanto, podemos notar que em 16 *benchmarks*, nenhuma das taxas conseguiram ser efetivas, com um *speedup* médio em relação ao -Ox inferior a 1,02.

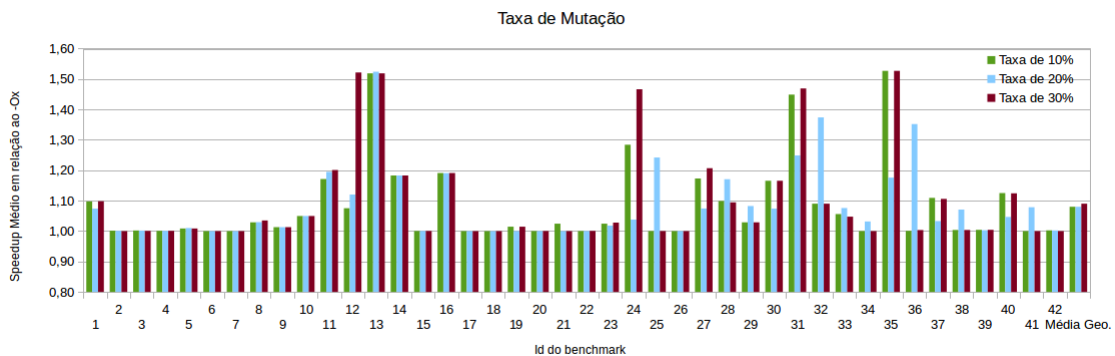


Figura 19 – Gráfico comparando os *speedups* médios de cada valor do parâmetro de *mutação*.

A Figura 19 mostra o gráfico da comparação das diferentes taxa de *mutação*. Nele, é possível notar que a média geométrica dos *speedups* foram muito próximas entre si, no entanto, para a taxa de 30% esse valor foi um pouco superior em relação às outras. Além disso, é possível notar que a taxa mais efetiva foi a de 30% que conseguiu ser a melhor em 27 códigos além do maior *speedup*. No entanto, pode notar que em 14 *benchmarks*, nenhuma das taxas conseguiram ser efetivas, ou seja, nesses *benchmarks* nenhuma taxa conseguiu obter um *speedup* médio de pelo menos 1,02.

A Figura 20 mostra o gráfico da comparação dos diferentes valores testados para o tamanho da população inicial. Nele, é possível notar que a média geométrica dos *speedups*

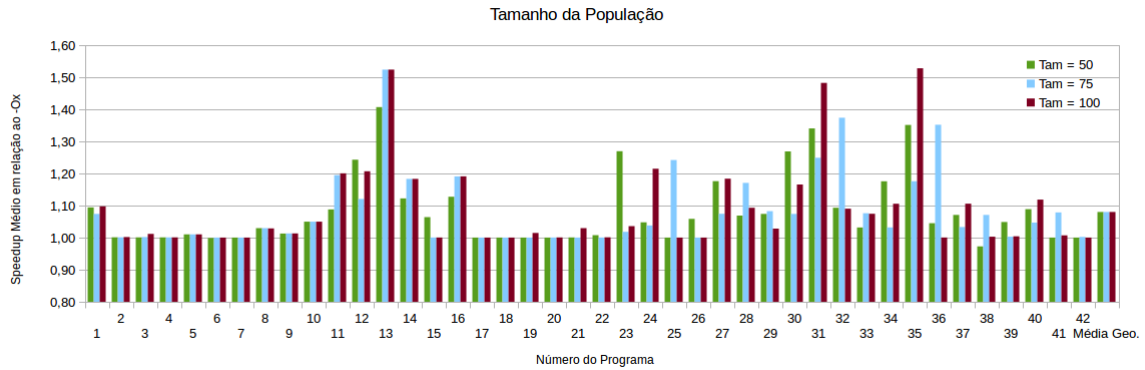


Figura 20 – Gráfico comparando os *speedups* médios de cada valor do parâmetro de tamanho da população.

foi a mesma para os três valores testados (1,08). Além disso, é possível notar que a taxa mais efetiva foi a de 100% que conseguiu ser a melhor em 23 códigos. No entanto, pode notar que em 13 *benchmarks*, nenhuma das taxas conseguiram ser efetivas, ou seja, nesses benchmarks nenhuma taxa conseguiu obter um *speedup* médio de pelo menos 1,02. Para o benchmark de id 38 (IMG_wave_horz_c) o *speedup* médio da população com tamanho 50 foi inferior ao -Ox.

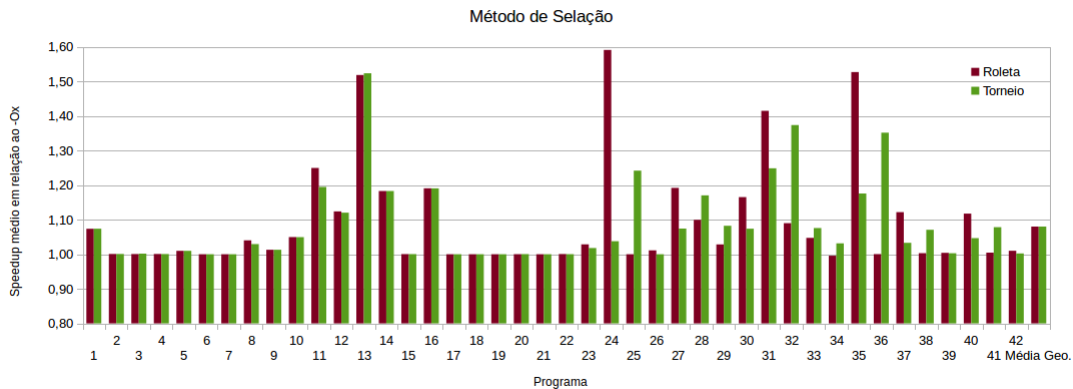


Figura 21 – Gráfico comparando os *speedups* médios da melhor configuração global com as melhores configurações locais.

A Figura 20 mostra o gráfico da comparação para os dois métodos de seleção testados (roleta e torneio). Nele, é possível notar que a média geométrica dos *speedups* foi a mesma para os métodos testados (1,08). Além disso, é possível notar que o método mais efetivo foi a roleta melhor em 23 códigos. No entanto, pode-se notar que em 17 *benchmarks*, nenhum dos métodos conseguiu ser efetivos, ou seja, nesses benchmarks nenhum método conseguiu obter um *speedup* médio de pelo menos 1,02.

Depois que os testes foram executados e os gráficos elaborados, foi necessário criar um critério que auxiliasse na escolha do melhor valor para cada parâmetro avaliado. Para avaliar qual o valor de um parâmetro específico teve o melhor desempenho, fora contabilizado a quantidade de códigos que um atingiu o melhor desempenho (maior *speedup* sobre o -Ox, considerando a sequência de otimização sugerida pelo AG configurado com cada valor do parâmetro em análise). Esse ranqueamento foi definido de acordo com o seguinte algoritmo:

1. Para cada $b_i \in B$, tal que B é o conjunto de todos os *benchmarks*, encontre o s_i que é o maior *speedup* médio, considerando todos os valores testados para um parâmetro específico (Exemplo: taxa de mutação).
2. Se o *speedup* médio do valor v_j para o *benchmark* b_i for igual a s_i , então incremente em um o contador de códigos do valor v_i .
3. Ao final, o valor que possuir o maior contador é elegido como o melhor daquele parâmetro.

Note que para um mesmo *benchmark* e um parâmetro, dois valores diferentes podem resultar em *speedups* iguais (mesma sequência de otimização ou sequências diferentes, mas com mesmo desempenho). Nestes casos, ambos os valores são incrementados. Dessa forma, a somatória de todos os contadores para um mesmo parâmetro é diferente do total de *benchmarks*. A média geométrica dos *speedups* e tempo médio de execução também foram utilizados como métricas para a avaliação dos valores. Portanto, para a escolha da melhor configuração, foram adotados os valores dos parâmetros que supera os demais em pelo menos 2 dessas métricas. A Tabela 8 apresenta as métricas obtidas para cada valor desses parâmetros testados. Nessa tabela, as melhores métricas e o melhor valor de cada parâmetro estão destacados em cinza.

Tabela 8 – Tabela consolidando os resultados de cada parâmetro.

	Parâmetros										
	Mutação			Crossover			Tamanho da população			Método de seleção	
Valor	10%	20%	30%	70%	80%	90%	50	75	100	Roleta	Torneio de 3
Média Geométrica de speedup	1,0766	1,0796	1,0900	1,0845	1,0796	1,0714	1,0769	1,0796	1,0832	1,0840	1,0796
Tempo Médio de exploração (em minutos)	6,1199	7,0723	9,7417	10,9821	7,0723	7,3648	5,5545	7,0723	9,0331	7,6400	7,0723
Quantidade de códigos que obtiveram melhor desempenho	17	18	36	24	27	22	14	15	36	27	21

Como pode ser observado na Tabela 8 os parâmetros coloridos foram os que tiveram o melhor desempenho, se comparado com os demais valores. Portanto, a melhor configuração do AG considerada para o restante dos experimentos é composta por:

- Método de seleção: roleta.
- Taxa de crossover: 80%.
- Taxa de mutação: 30%.
- Tamanho da população: 100.

A Tabela 9 apresenta o desempenho do AG configurado com os "melhores valores" de cada parâmetro (geral), considerando as mesmas métricas usadas na comparação dos valores dos parâmetros. Para facilitar a análise desse desempenho, também foi replicado os desempenhos do AG considerando a melhor configuração individual de cada parâmetro. Como era esperado, a melhor configuração geral conseguiu superar as demais (exceto no tempo médio de exploração). A Figura 22 apresenta o desempenho dessas configurações para cada um dos 42 *benchmarks*. Como pode ser notado, a configuração geral conseguiu o maior *speedup* para a maioria dos *benchmarks*.

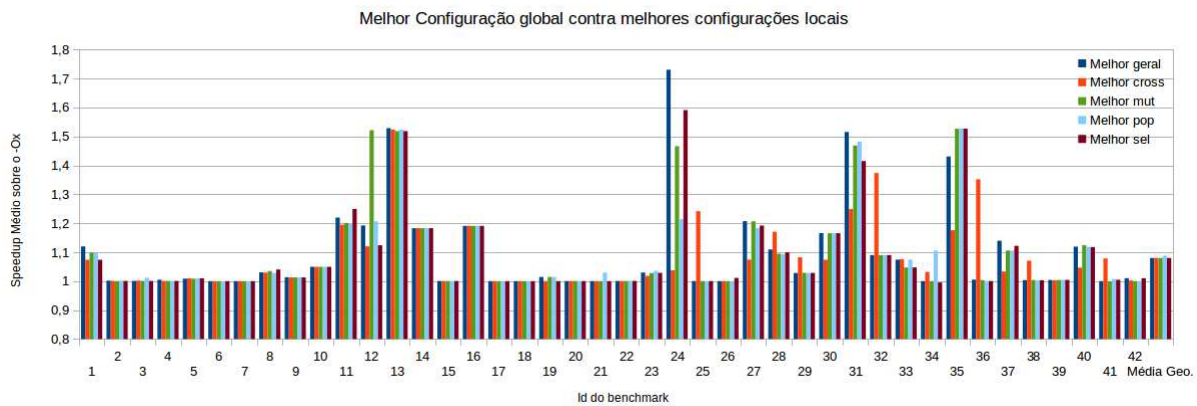


Figura 22 – Gráfico comparando os *speedups* médios de cada valor do parâmetro de método de seleção.

A Figura 22 mostra a comparação entre as melhores configurações individuais (crossover, mutação, método de seleção, tamanho da população) comparado com a melhor configuração global, ou seja, a configuração que reúne os melhores valores de cada método. No gráfico pode-se observar que os *speedups* são muito próximos entre si, variando de 1,08 a 1,09. A configuração que reúne os melhores parâmetros conseguiu ter o melhor desempenho geral, pois teve o melhor desempenho em 19 códigos. No entanto, a diferença em relação às demais configurações não foi muito discrepante.

Tabela 9 – Tabela comparativa entre o melhor de cada parâmetro individual e o melhor geral.

	Melhores Parâmetros				
	Geral	Crossover	Mutação	Tamanho da população	Método de seleção
Média Geométrica de speedup	1,0902	1,0796	1,0900	1,0832	1,0840
Tempo Médio de exploração (em minutos)	11,3462	7,0723	9,7417	9,0331	7,6400
Quantidade de códigos que obtiveram melhor desempenho	19	14	11	17	12

6.1.1 Análise Comparativa

Depois de encontrar a melhor configuração do AG, ele foi colocado em comparação com o algoritmo totalmente aleatório, que gerou 1000 indivíduos para cada execução e escolheu o melhor deles. A Figura 23 mostra a comparação entre o desempenho dos dois algoritmos para cada programa. Já a Tabela 10 mostra a consolidação dos dados. Como era esperado o AG possui um tempo de execução maior, no entanto, o seu desempenho é muito superior que o algoritmo totalmente aleatório. Note que não houve nenhum *benchmark* que o algoritmo aleatório superou o desempenho do algoritmo genético.

Tabela 10 – Tabela comparativa do desempenho do AG com o algoritmo aleatório.

	Melhores Parâmetros	
	AG	Aleatório
Média Geométrica de speedup	1,0902	0,8953
Tempo Médio de exploração (em minutos)	11,3462	3,8774
Quantidade de códigos que obtiveram melhor desempenho	42	0

O gráfico da Figura 23 mostra a comparação de desempenhos entre a melhor configuração encontrada do AG e a busca aleatória. Pode-se observar que para todos os *benchmarks* o desempenho do AG foi superior ao desempenho da busca totalmente aleatória para todos os *benchmarks*. Além disso, pode-se notar que a média geométrica dos *speedups* da busca aleatória foi de 0,9 que é inferior ao -Ox.

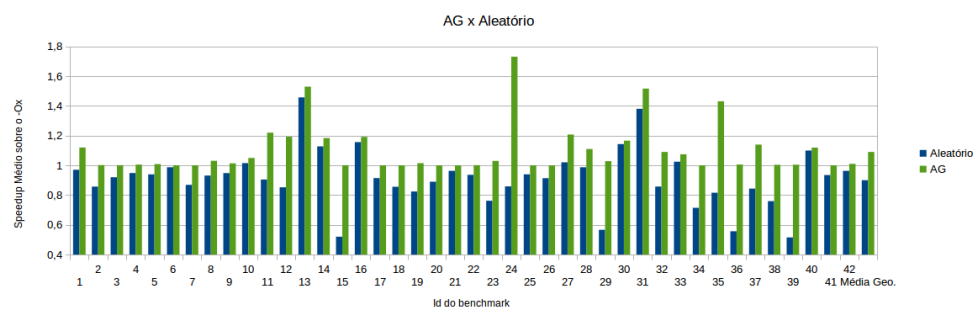


Figura 23 – Gráfico comparando os *speedups* médios de cada valor do parâmetro de método de seleção.

7 Conclusões e trabalhos futuros

Neste trabalho, foi desenvolvido um modelo de exploração iterativa que busca encontrar sequências de otimização específicas para o código compilado e a arquitetura desejada. As sequências indicadas pelo modelo são capazes de gerar um código otimizado mais eficiente, em termos de tempo de execução (latência), que aquele gerado com o uso dos níveis de otimização do compilador.

Como esperado, o nosso modelo de exploração baseado em algoritmo genético obteve um desempenho superior ao dos níveis de otimização do compilador na maioria dos códigos avaliados. Dos 42 *benchmarks* utilizados, o nosso modelo foi superior que o -Ox em 60% dos códigos. Além disso, o *speedup* médio do nosso modelo foi de 1,09 em relação ao -Ox. Se comparar os resultados obtidos com a abordagem baseada em amostragem aleatório, o desempenho do AG é ainda melhor (*speedup* de 1,21 em relação à amostragem aleatória). Para todos os *benchmarks* avaliados, a abordagem baseada em AG foi superior ao modelo aleatório que, por sua vez, obteve um *speedup* médio de apenas 0,9 em relação ao -Ox.

Durante os experimentos foram estudadas diversas alternativas de configuração a fim de encontrar um modelo de AG eficiente para o problema de selecionar boas sequências de otimizações. Além de testar diversos parâmetros do AG, o modelo investigou diferentes estratégias de geração da população inicial. Essa estratégia foi definida para tentar gerar uma quantidade maior de indivíduos válidos na população inicial.

No entanto, algumas adversidades também foram encontradas, como por exemplo, a geração de uma quantidade muito elevada de indivíduos inválidos na população inicial. Outro ponto preocupante é o tempo de exploração gasto pelo AG para encontrar uma solução para um *benchmark* específico. Em média, a melhor configuração do AG gastou 11 minutos para encontrar a solução para cada *benchmark* explorado.

Para tentar contornar essas adversidades, há alguns pontos que poderiam ser investigados em trabalhos futuros. São eles:

- Aplicar o AG com as configurações encontradas em novos benchmarks de teste e comparar o resultado com o -Ox.
- Aplicar técnicas de *data-mining* para identificar correlação entre passos de otimização e conjunto de código.
- Utilizar aprendizado de máquina para agrupar os códigos de acordo com a semelhança

que eles possuem.

- Criar uma base de dados com muitas sequências boas e tentar extrair informações que poderiam ser úteis por exemplo na formação da população inicial.
- Investigar outras abordagens e parâmetros do AG.

Referências

- AGAKOV, F. et al. Using machine learning to focus iterative optimization. In: IEEE COMPUTER SOCIETY. *Proceedings of the International Symposium on Code Generation and Optimization*. [S.l.], 2006. p. 295–305. Citado 6 vezes nas páginas 7, 8, 14, 25, 26 e 27.
- AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers, Principles, Techniques*. [S.l.]: Addison wesley, 1986. Citado 3 vezes nas páginas 7, 11 e 12.
- ALMAGOR, L. et al. Finding effective compilation sequences. *ACM SIGPLAN Notices*, ACM, v. 39, n. 7, p. 231–239, 2004. Citado na página 8.
- ASHOURI, A. H. et al. A bayesian network approach for compiler auto-tuning for embedded processors. In: IEEE. *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. [S.l.], 2014. p. 90–97. Citado 2 vezes nas páginas 26 e 27.
- COOPER, K. D.; SCHIELKE, P. J.; SUBRAMANIAN, D. Optimizing for reduced code space using genetic algorithms. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1999. p. 1–9. Citado 8 vezes nas páginas 7, 8, 14, 18, 25, 26, 27 e 41.
- DEBRAY, S. K. et al. Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems (TOPLAS)*, ACM, v. 22, n. 2, p. 378–415, 2000. Citado 2 vezes nas páginas 7 e 25.
- DUBACH, C.; JONES, T. M.; O’BOYLE, M. F. Exploring and predicting the effects of microarchitectural parameters and compiler optimizations on performance and energy. *ACM Transactions on Embedded Computing Systems (TECS)*, ACM, v. 11, n. 1, p. 24, 2012. Citado 2 vezes nas páginas 7 e 25.
- FIDELIS, M. V.; LOPES, H. S.; FREITAS, A. A. Discovering comprehensible classification rules with a genetic algorithm. In: IEEE. *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*. [S.l.], 2000. v. 1, p. 805–810. Citado na página 18.
- GOLBERG, D. E. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, v. 1989, p. 102, 1989. Citado 2 vezes nas páginas 19 e 37.
- HOLLAND, J. H. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. [S.l.]: U Michigan Press, 1975. Citado na página 19.
- HOLMES, J. *Object-oriented compiler construction*. [S.l.]: Prentice-Hall, Inc., 1995. Citado na página 12.
- HORNER, A.; GOLDBERG, D. E. Genetic algorithms and computer-assisted music composition. 1991. Citado na página 18.

HOSTE, K.; EECKHOUT, L. Cole: compiler optimization level exploration. In: ACM. *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. [S.l.], 2008. p. 165–174. Citado na página 8.

JOHNSON, N. E. *Code size optimization for embedded processors*. [S.l.], 2004. Citado na página 13.

JUNIOR, N. L.; ANDERSON. Uma solução híbrida para o problema de seleção de otimizações. 2016. Citado 4 vezes nas páginas 7, 14, 26 e 27.

KULKARNI, P. A.; WHALLEY, D. B.; TYSON, G. S. Evaluating heuristic optimization phase order search algorithms. In: IEEE. *International Symposium on Code Generation and Optimization (CGO'07)*. [S.l.], 2007. p. 157–169. Citado 2 vezes nas páginas 14 e 25.

MARTINS, L. G. et al. Clustering-based selection for the exploration of compiler optimization sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM, v. 13, n. 1, p. 8, 2016. Citado 12 vezes nas páginas 2, 7, 8, 12, 13, 14, 15, 17, 18, 26, 27 e 41.

PUNTAMBEKAR, A. A. *Principles of compiler design*. [S.l.]: Technical Publications, 2009. Citado na página 7.

PURINI, S.; JAIN, L. Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM, v. 9, n. 4, p. 56, 2013. Citado 3 vezes nas páginas 25, 26 e 27.

ULLMAN, J.; AHO, A.; SETHI, R. *Compiladores—principios técnicas e ferramentas*. LTC, 1995. Citado 2 vezes nas páginas 11 e 12.

VIEIRA, J. P. A. Otimização de controladores utilizando algoritmos genéticos para melhoria da capacidade de sobrevivência a afundamentos de tensão de aerogeradores de indução duplamente excitados. Universidade Federal do Pará, 2009. Citado na página 18.

ZHENGYING, W.; BINGXIN, S.; ERDUN, Z. Bandwidth-delay-constrained least-cost multicast routing based on heuristic genetic algorithm. *Computer communications*, Elsevier, v. 24, n. 7, p. 685–692, 2001. Citado na página 18.