Finding Good Compilation Plans: A Strategy to Enhance an Adaptive Optimization System

A. Ferrari, C. Segawa, and A. da Silva

***Abstract*—An important component of virtual machines is the adaptive optimization system, which decides what methods to optimize and what compiler optimization set to enable. In this context, this paper presents the development of an auto-tuning adaptive optimization system (a strategy to find good compiler optimizations), in order to obtain the best possible performance; that is, the reduction of runtime. Such system was implemented over the Jikes Research Virtual Machine, and the results indicate that the proposal is capable of achieving 11% better average performance, at a cost of less than 10% of the total runtime.**

***Index Terms*—Adaptive optimization system, Just-in-time compiler, Virtual machine, Performance.**

I. Introdução

COMPILADORES são programas que transformam código escrito em uma linguagem, para código em outra linguagem [1]. Tal processo é dividido em diversas fases, sendo uma das mais importantes a fase de otimização; devido ao fato de sua capacidade de melhorar a qualidade do código final.

Um aspecto importante de um compilador é sua habilidade de prover diversas otimizações (também conhecidas como transformações) [2]. Contudo, nem todas otimizações ocasionam um ganho de desempenho a um determinado programa. Neste contexto, surge o problema da seleção de otimizações o qual consiste em encontrar efetivas otimizações para um determinado programa.

Diversas pesquisas tem investigado diferentes estratégias para encontrar o melhor conjunto de otimizações para um determinado programa [3][4][5]. Tais pesquisas empregam tradicionalmente estratégias de compilação iterativa, que são capazes de gerar e avaliar diferentes conjunto de otimizações. Ao final, tais estratégias retornam o código final com a melhor qualidade. Vale ressaltar que neste contexto qualidade pode ter diferentes aspectos, tais como: redução do tempo de execução, redução do tamanho de código, redução do consumo de energia, ou outros.

Máquinas Virtuais Java (`JVM` – *Java Vitual Machine*) utilizam um método misto para execução, no qual métodos do programa são inicialmente interpretados, ou compilados por um compilador não-otimizado, e caso um método seja executado com frequência (método quente), o mesmo provavelmente será compilado, ou recompilado, por um compilador otimizador *Just-In-Time* (`JIT`) para melhorar o desempenho.

O Sistema de Otimização Adaptativo (`SOA`) de um compilador `JIT` é responsável por decidir quais métodos serão recompilados, em qual nível de otimização e qual conjunto de otimizações será utilizado pelo nível especificado. Deste modo, o desafio é encontrar a melhor configuração para o `SOA`, pois uma configuração *ótima* é altamente dependente do programa (código fonte), dos planos de otimização (nível de otimização e conjunto de otimizações, deste nível, que serão utilizados pelo compilador para compilar/recompilar um método) e da plataforma de *hardware*.

Neste contexto, os trabalhos [6] e [7] descrevem como planos de otimização podem ser determinados manualmente. Por outro lado, os trabalhos [8] [9] [10] aplicam diferentes planos de otimização a diferentes métodos. O trabalho de Zhao [10] demonstrou que algoritmos inteligentes para automatizar o processo de configuração do `SOA`, na busca por planos de otimização, tendem a obter bons resultados.

A proposta deste artigo é desenvolver um `SOA` seja capaz de trocar dinamicamente o plano de otimização durante toda a execução da aplicação, o que não é realizado na proposta de Zhao. Em outras palavras, a proposta é desenvolver um `SOA`, para uma `JVM` que utilizada diferentes níveis de otimização, que encontre bons conjuntos de otimizações e não se limite a um determinado nível de otimização.

O sistema proposto por Zhao aguarda a primeira recompilação do método, feita pelo `SOA`, para iniciar a busca por uma melhor configuração. Além disto, tal sistema utiliza o mesmo nível de otimização durante todas as recompilações de um método, ajustando apenas as otimizações que compõem tal nível de otimização. Portanto, embora o trabalho de Zhao tenha sido desenvolvido em uma JVM que utiliza diversos níveis de otimização, nem todos os níveis são utilizados.

No sistema proposto neste artigo, a busca por bons planos de otimização inicia a partir da primeira invocação do método; além disto, o sistema pode alterar o nível de otimização em qualquer iteração além de ajustar a configuração do nível de compilação (otimizações por ele utilizadas).

Os experimentos utilizando os *benchmarks* `DaCapo` [11] mostram que o sistema proposto é capaz de melhorar, na média, em 11% o desempenho, quando comparado com o sistema original. Tais resultados reforçam a necessidade de se configurar o compilador `JIT` às características de cada programa.

As contribuições deste artigo são como segue.

1) Um `SOA` capaz de se adaptar as características do programa em execução, visando reduzir o tempo de

A. F. Cardoso é analista de software na Mendes Cardoso Arquitetura, Maringá/Paraná. email: adriano_cardoso83@hotmail.com.

C. H. S. Tonetti é aluno de pós-graduação no Programa de Pós-graduação em Ciência da Computação da Universidade Federal de Minas Gerais, Minas Gerais/Brasil. email: caio.tonetti@gmail.com.

A. F. da Silva é professor associado da Universidade Estadual de Maringá, Paraná/Brasil. email: anderson@din.uem.br.

execução de tal programa.
2) Uma estratégia de troca de planos de otimização, que não se limita a determinados momento mas pode potencialmente ocorrer durante toda a execução do programa.
3) Uma estratégia que melhora a qualidade do código gerado por um compilador JIT.
4) Um SOA capaz de melhorar o desempenho médio de 11%, com baixo custo de execução.

O restante deste artigo está organizado como descrito a seguir. A Seção II apresenta trabalhos relacionados. A Seção III fornece os detalhes da proposta apresentada neste artigo. A Seção IV contém os resultados obtidos. Por fim, a Seção V apresenta as conclusões.

II. Trabalhos Relacionados

Os trabalhos [6] e [7] descrevem como níveis de otimização podem ser determinados manualmente. Tais trabalhos diferem do trabalho aqui proposto, pelo fato do proposto automatizar o processo de descoberta de níveis de otimização, como também o processo de descoberta de conjuntos de otimizações.

Silva e Outros [12] exploram os parâmetros da JVM com o objetivo de encontrar uma boa configuração para a execução de programas paralelos. Tal trabalho demonstrou que é possível melhorar o desempenho de programas paralelo Java, a medida que a máquina virtual seja parametrizada corretamente. Assim como nos dois trabalhos citados acima, Silva e Outros não automizatiza o sistema mas avalia várias parâmetros da JVM manualmente. Portanto, o trabalho de Silva e Outros se difere do aqui apresentado pelo fato deste automatizar a JVM.

Cavazos e O'boyle utilizam uma abordagem diferente, aplicando diferentes planos de otimização para cada método compilado pela JVM [8]. As otimizações nesses planos são determinadas utilizando uma função de regressão logística que prevê quais otimizações serão mais úteis para determinado método de acordo com as características do *bytecode*.

Jantz e Kulkarni [9] também abordaram o problema de seleção de otimizações, no contexto da JVM. Os autores desenvolveram uma estratégia que analisa os métodos a serem compilados e baseado em tal análise remove do conjunto de otimizações padrão as otimizações que possivelmente terão um impacto negativo no desempenho.

Os trabalhos [8] e [9] se assemelham ao aqui proposto por ambos automatizarem o processo de seleção de otimizações.

III. Estratégias para um Sistema de Otimizaçao Adaptativo

Um bom ambiente de compilação é aquele capaz de se adaptar as características do programa a ser compilado. Isto indica, que tal ambiente é capaz de escolher bons planos de otimização, que são aqueles que efetivamente melhoram o desempenho do programa em questão.

Este trabalho tem por objetivo melhorar o desempenho de um SOA, proporcionando que diferentes planos de otimização sejam aplicados ao programa em execução. De fato, este trabalho adiciona duas contribuições ao trabalho de Zhao [10], são elas:

1) O sistema proposto adianta o processo de busca por novos planos de otimização. Enquanto no trabalho original, a busca por um novo plano de compilação inicia no momento de recompilação de um método, aqui é proposto um sistema no qual a busca inicia no momento de invocação de um método.
2) O sistema proposto é capaz de alterar o nível de otimização de um determinado médodo. No trabalho original, não existe a troca de um determinado nível de otimização, apenas o ajuste das otimizações que compôem tal nível. Isto indica, que um determinado método nunca é promovido a outro nível de otimização durante a execução do programa. O sistema proposto além de alterar o nível de otimização, ajusta as otimizações que o compôem. Processo este que pode ocorrer em diversos momentos da execução.

Assim como o trabalho de Zhao, o trabalho aqui apresentado é desenvolvido na Jikes RVM [13] [14]. Tal JVM fornece um sistema aberto e flexível para o desenvolvimento e experimentação de estratégias de projeto para máquinas virtuais.

É importante perceber que o objetivo do trabalho apresentado a seguir não é melhorar o desempenho do SOA da máquina virtual Jikes RVM, mas sim demonstrar estratégias que possam melhorar o desempenho de um SOA que utiliza diferentes níveis de otimização. Portanto, é importante notar que Jikes RVM foi escolhido por ser um sistema aberto, flexível, e utilizar diferentes níveis de otimização.

O SOA proposto é baseado em um algoritmo heurístico, o qual tem por objetivo escolher planos de otimização. De fato, foram criadas duas versões para o SOA, são elas:

1) uma utilizando um algoritmo genético (AG) convencional; e
2) uma utilizando o SPEA2 (*Strength Pareto Evolutionary Algorithm 2*) [15].

Ambas versões utilizam *feedback online* (sem a necessidade de execução prévia) para coletar informações que serão utilizadas para realizar otimizações adaptativas [6].

Cada versão inicia com uma população aleatória, na qual cada indivíduo possui seu valor de *fitness*. Este é diretamente relacionado com o desempenho que um método obteve ao usar um determinado plano de otimização, e é calculado por meio de duas funções objetivo:

1) taxa de compilação (número de *bytecodes* compilados por milissegundo); e
2) número de amostras (quantidade de vezes que aparecem chamadas de um método na pilha de execução) em um intervalo de tempo.

Um ponto importante a ser destacado é o fato dos algoritmos heurísticos serem sem supervisão, pois é desconhecido qual plano de otimização é melhor ou pior antes do início da busca em tempo de execução.

O AG convencional proposto neste trabalho é similar ao utilizado por Zhao. Contudo, enquanto Zhao utilizou como segunda abordagem o algoritmo Random-Mutation Hill Climbing [10], este trabalho utiliza o SPEA2.

A. Os Algoritmos Genéticos

AG convencional A população inicial do AG é um vetor que contém um conjunto de 30 DNAs criados aleatoriamente, onde cada DNA é composto por objetos *OptOptions* (otimizações disponibilizadas na máquina virtual Jikes RVM). Tal população é criada aplicando mutação na configuração inicial do compilador JIT, cuja taxa é de 4% indicando que somente uma das otimizações de cada DNA será modificado. Novos indivíduos são gerados aplicando um operador *crossover*, em um DNA escolhido aleatoriamente e no DNA que foi utilizado na última recompilação do método. Após a criação de dois novos indivíduos, cada um possui uma chance de 10% de sofrer mutação. Por fim, os dois indivíduos criados substituem seus pais na população corrente, e um deles é escolhido, aleatoriamente, para recompilar o método (este é denominado de plano de otimização corrente).

SPEA2 É uma variação do AG convencional, que utiliza uma população intermediária para armazenar os indivíduos dominantes e realizar as operações de *crossover* e mutação. O cálculo do *fitness* também é diferente, sendo necessário calcular o valor de *raw fitness* $R(j) = \sum S(i)$ e a densidade $D(i) = \frac{1}{d+2}$. $S(i)$ é o número de membros j da população que são dominados ou iguais a i em relação aos valores das funções objetivo e a densidade d é a distância euclidiana entre as funções objetivo de i com o indivíduo mais próximo. O *fitness* final de um indivíduo é $f(i) = R(i) + D(i)$ [15].

A escolha por utilizar um AG se deve ao fato de tal algoritmo alcançar bons resultados, quando aplicado à diversos contextos, como por exemplo: descoberta de conjuntos de otimizações [4], sistema de inspeção industrial [16], planejamento de sequencia de montagem [17], programação inteira [18], entre outros.

No contexto de descoberta de conjuntos de otimizações, o uso de um AG tem se monstrado ser uma boa estratégia [4] [5].

B. Estrutura Baseada em Árvore

A estrutura baseada em árvore é a infraestrutura básica do SOA, a qual indica qual plano de otimização deve ser utilizado para recompilar um determinado método. Cada nó da árvore contém um conjunto de DNAs (população). Além disto, cada nó contém um *apontador* que indica qual DNA da população será o plano de otimização corrente.

Em nossa proposta, o SOA utiliza uma estratégia de recompilação multi-nível. Nessa estratégia, procura-se encontrar um nível de otimização maior para um método que precisa ser recompilado (ou seja, promover um método a um nível de otimização mais alto). Como existem três níveis de otimização na Jikes RVM (Opt0, Opt1 e Opt2) existem três árvores no SOA, uma correspondente para cada nível.

Na implementação de Zhao, quando a máquina virtual evita mudar o nível de otimização de determinado método, começa o processo de busca por um bom plano de otimização no nível de otimização em que o método foi recompilado pela última vez, percorrendo a mesma árvore de DNA até o final da execução deste método. No trabalho aqui proposto, a busca é iniciada a partir da compilação do método pelo compilador base (primeira invocação) e, independente se a máquina virtual decide ou não mudar o nível de otimização durante as recompilações, o uso da estrutura baseada em árvore ocorre dinamicamente. Desta forma, caso a máquina virtual decida alterar o nível de otimização para um nível maior, a busca é transferida para a nova árvore.

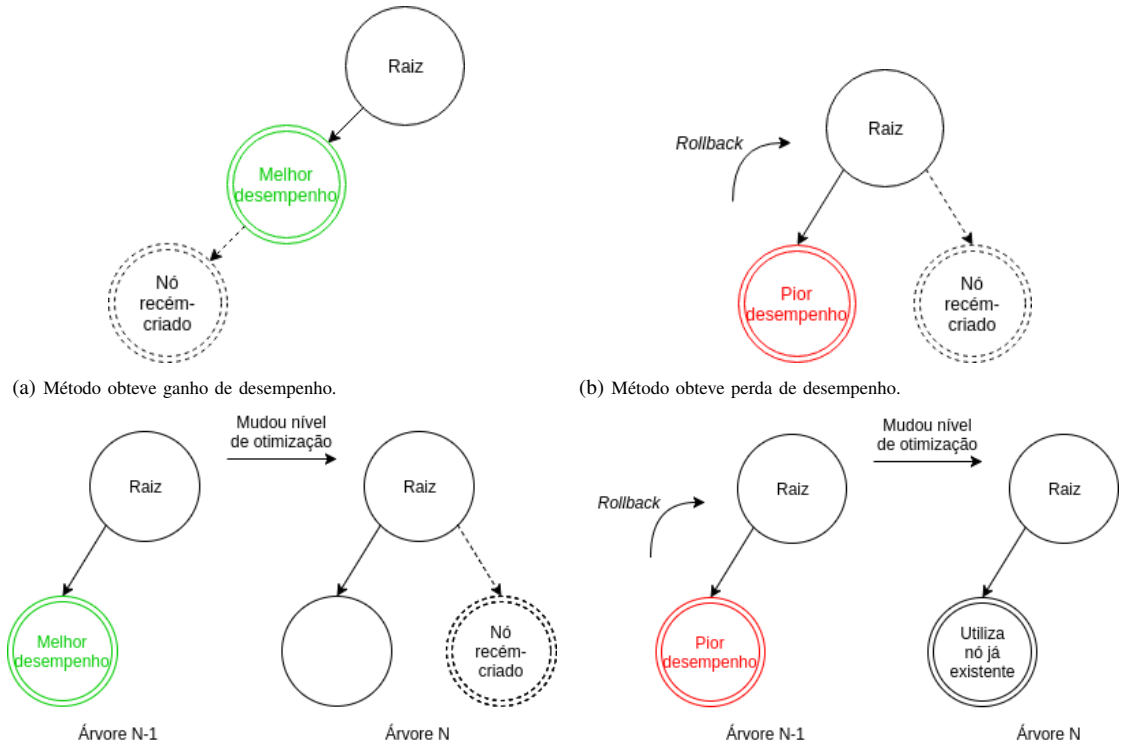
Quando um método é relacionado para recompilação pela primeira vez pelo SOA, ele "aponta" para a raiz da árvore e envia o plano de otimização corrente desse nó para o compilador JIT realizar a recompilação propriamente dita. O plano de otimização corrente da raiz encapsula a configuração básica do compilador JIT, ou seja, todos os métodos que possuem referência para a raiz são recompilados pela configuração original da Jikes RVM. Após a recompilação do método com o plano de otimização corrente da raiz, o SOA armazena o desempenho do método executado, para calcular o *fitness* e realizar futuras comparações com novos nós que podem ser criados na árvore. A Figura 1 apresenta um exemplo do gerenciamento da árvore.

Caso o SOA decida recompilar novamente o método, ele deverá apontar para um novo nó e analisar o desempenho obtido com a última recompilação. Neste caso, quatro situações devem ser analisadas, são elas:

1) Se a última recompilação obteve ganho de desempenho, o método passa a apontar para o primeiro filho de seu nó atual ou, caso não haja filhos o SOA cria um nó com os componentes do nó atual, realizando *crossover* e mutação. Neste caso, o nó criado será referenciado como filho do nó atual como pode ser visto na Figura 1a;
2) Se a última recompilação obteve perda de desempenho é realizado *rollback*, como pode ser visto na Figura 1b;
3) Se a última recompilação obteve ganho de desempenho e foi determinado pela máquina virtual a mudança no nível de otimização, o SOA utiliza os componentes do nó atual para criar um novo nó, o qual será inserido na árvore referente ao novo nível como um filho da raiz, como mostra a Figura 1c;
4) Se a última recompilação obteve perda de desempenho e foi determinado que o método mude o nível de otimização, é realizado o *rollback*, como pode ser visto na Figura 1d.

Sempre que um novo nó é criado, os dois novos DNAs gerados são inseridos na população substituindo quem os gerou. Porém, caso a mudança não seja vantajosa para o desempenho do método, é interessante voltar a um estado anterior da população.

Para alcançar tal objetivo, o SOA proposto cria *snapshots*. Neste caso, a população representada em um nó é uma cópia original da população do nó pai, fornecendo um ponto seguro para realizar *rollback*, já que todas as modificações feitas na população de um nó não afetam a população de seu pai.



(a) Método obteve ganho de desempenho.

(b) Método obteve perda de desempenho.

(c) Ganho de desempenho com mudança no nível de otimização.

(d) Perda de desempenho com mudança no nível de otimização.

Fig. 1. Funcionamento da estrutura em árvore.

C. Estratégia de Otimização Por-Método

Os métodos considerados para recompilação devem percorrer os nós das árvores de maneira independente, pois um plano de otimização pode resultar em diferentes desempenhos para métodos distintos [8]. Devido a este fato, o SOA proposto utiliza uma estratégia por-método, o que indica que cada método percorre de maneira independente a árvore para encontrar seu plano de otimização.

A busca por um plano de otimização finaliza quando um dos dois casos ocorre:

1) Quando um método deixa de ser considerado "quente". Isso acontece quando o método não atinge um valor mínimo de amostras coletadas, sendo esse valor dinâmico, iniciando em 1% e pode decair até 0,25% [19], ou seja, se um método possui mais de 1% do número de amostras coletadas em um intervalo de tempo ele é considerado um método quente e, se a máquina virtual não estiver encontrando muitos métodos quentes, o valor cai gradativamente até 0,25%;
2) Quando um método já foi recompilado ao menos uma vez no nível máximo de otimização.

D. Medição de Desempenho

A Jikes RVM realiza uma contagem do número de amostras de um método quando ocorre troca de *threads*. Em geral, é utilizado o valor de amostras coletadas durante 20 operações de troca de *threads* (valor padrão da máquina virtual Jikes RVM), sendo definido como um intervalo de tempo. Esse mecanismos foi preservado na proposta aqui apresentada.

Para cada método recompilado, é registrado o desempenho no intervalo de tempo atual e no intervalo de tempo cujo método foi recompilado pela última vez, assim os valores podem ser usados para comparação no futuro.

No SOA original da Jikes RVM, um evento de método quente é gerado para cada método que foi capturado no processo de amostragem dentro de um intervalo de tempo. Para registrar o desempenho no intervalo de tempo atual foram utilizados dois atributos da classe *HotMethodEvent*:

1) o número de amostras capturadas até a criação do evento de método quente; e
2) a taxa de compilação.

Para o desempenho no intervalo de tempo que o método foi recompilado pela última vez foi criado uma estrutura *hash map*, que armazena o número de amostras e a taxa de compilação sempre que um evento de método quente é criado, sendo a chave desse valor o identificador do método.

Os valores salvos na *hash map* e no atual evento são usados pelo SOA, aqui proposto, para calcular o *fitness* de um DNA, da seguinte forma:

- Quanto menor o número de amostras coletadas, menor foi o processamento gasto com o método, consequentemente melhor desempenho, e maior será o *fitness*;
- Quanto menor a taxa de compilação, melhor é a qualidade do código recompilado e maior será o *fitness*.
- Se o SOA está utilizando o SPEA2, é necessário verificar a dominância de todos os DNAs de uma população para calcular o valor de *strength* e *raw fitness*.

IV. Resultados e Discussão

Esta seção apresenta os resultados alcançados pelo SOA proposto. Inicialmente é apresentado a configuração do sistema, para após apresentar os resultados obtidos.

A. Configuração do Sistema

Hardware Os experimentos foram realizados em uma máquina com arquitetura Intel, mais especificamente com um processador Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz com um Ubuntu 16.04.02 LTS. O sistema possui memória de 16 GB.

Jikes RVM A configuração de *build* da Jikes RVM foi a *FastAdaptiveGenImmix*, também conhecida como *production build* em plataformas linux.

Níveis de otimização Nos experimentos são utilizados três níveis de otimização da Jikes RVM: Opt0, Opt1 e Opt2. Os dois primeiros são como definidos pela máquina virtual. Contudo, para o nível Opt2 foi adicionado as otimizações: SSA_EXPRESSION_FOLDING, SSA_REDUNDANT_BRANCH_ELIMINATION, e SSA_LOAD_ELIMINATION; as quais são desabilitadas por padrão na máquina virtual.

Nível máximo de otimização Opt2

Tamanho da população no algoritmo genético 30

Aplicações O desempenho do sistema foi avaliado utilizando seis aplicações dos *benchmarks* DaCapo [11], são elas: Avrora, Sunflow, Lusearch, H2, Pmd, e Xalan. Embora o benchmark DaCapo seja composto por 14 aplicações, apenas estas seis funcionam corretamente na máquina virtual Jikes RVM. É importante relembrar que a máquina virtual Jikes RVM é uma máquina virtual de pesquisa, que não fornece suporte a todas funcionalidades da máquina virtual Java, consequentemente nem todos *benchmarks* Java podem ser executados.

Iteração *Online* Os experimentos utilizam iteração *online*, indicando que um programa teste será executado várias vezes sem que a máquina virtual reinicie ao final de cada execução. Desta forma é possível realizar a busca por planos de otimização da primeira a última iteração, e assim coletar o desempenho de cada plano. Ao utilizar esta estratégia, a árvore de DNA pode ser construída consistentemente. Caso a máquina virtual fosse reiniciada entre cada iteração, para cada execução as árvores seriam reconstruídas do início; portanto, com iteração *online* é possível que as três árvores *cresçam* em uma maior proporção, melhorando a busca por planos de otimização.

Número de iterações online 30

B. Experimentos e Discussão

O SOA proposto neste artigo foi avaliado em 3 configurações distintas, a saber:

1) o SOA proposto com AG (AG-SOA);
2) o SOA proposto com SPEA2 (SPEA2-SOA); e
3) o SOA proposto sem uso de *rollback* (AG-SOA sem Rollback).

O desempenho obtido por tais configurações foi comparado com o desempenho obtido pelo sistema padrão (SOA-Original) e com a versão de Zhao (SOA-Zhao).

Todos os experimentos foram realizados com a estratégia de otimização por-método, cujos resultados são apresentados ns Figuras 2, 3, 4, 5, 6 e 7.

Os resultados obtidos para Avrora (Figura 2) mostram uma melhora média, no desempenho, de aproximadamente 4% com picos de até 10% no decorrer das iterações utilizando o AG-SOA em relação ao SOA-Original. Contudo, a partir da oitava iteração o desempenho do AG-SOA se mantém constante até o final. Para o SPEA2-SOA, o desempenho foi pior que o AG-SOA, mas apresenta uma melhora média de 2% em relação ao SOA-Original. Com relação ao SOA-Zhao é observável que no geral o seu desempenho foi inferior ao SOA proposto neste artigo, exceto para a iteração 14 na qual o SOA-Zhao obteve um desempenho 2.5% superior ao obtido por AG-SOA.

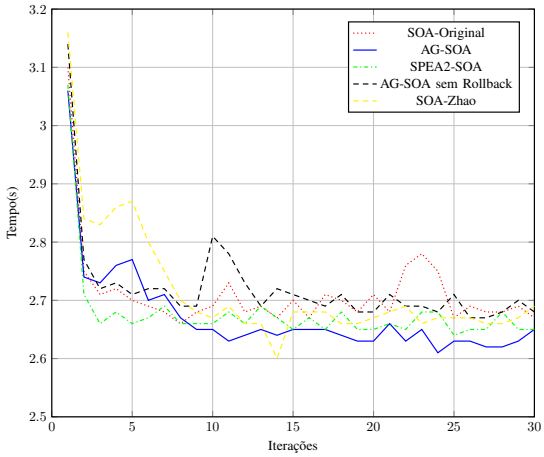


Fig. 2. Avrora.

O *benchmark* Sunflow (Figura 3) com o AG-SOA apresenta melhoras em relação ao SOA-Original a partir da sétima iteração, obtendo tempo de execução menor ou igual e com um pico de até 12% melhor (iteração 20). Com o SPEA2-SOA, o desempenho foi pior que o AG-SOA e semelhante ao SOA-Original. O desempenho de AG-Zhao segue o mesmo padrão de AG-SOA, contudo com uma ligeira perda de desempenho.

O *benchmark* Lusearch (Figura 4) com AG-SOA apresenta desempenho semelhante ao SOA-Original até a 20ª iteração. A partir desta iteração ocorre uma redução do tempo de execução, com melhora de 11% na iteração 25. O SPEA2-SOA se mostra mais lento que o AG-SOA e SOA-Original durante praticamente todas as iterações do experimento. O desempenho obtido por SOA-Zhao está compreendido entre o desempenho obtido por AG-SOA e SPEA2-SOA. Como pode ser observado, SOA-Zhao obteve no geral um desempenho inferior a SOA-Original, exceto para a iteração 13 para a qual SOA-Zhao foi superior a todas outras versões.

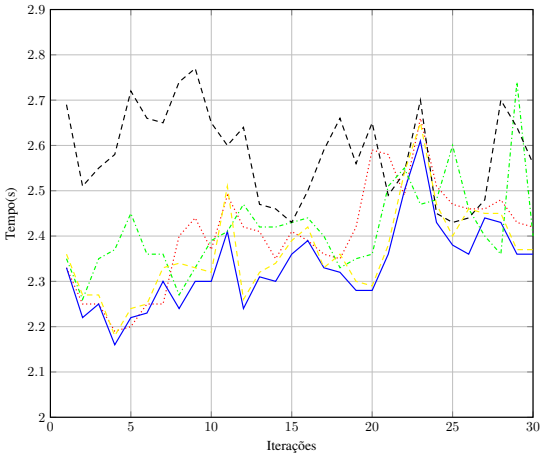


Fig. 3. Sunflow.

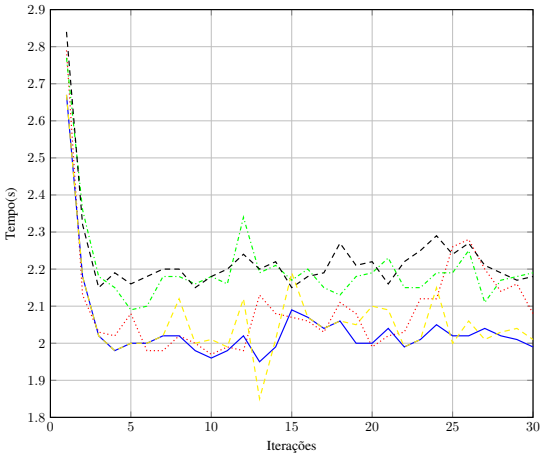


Fig. 4. Lusearch.

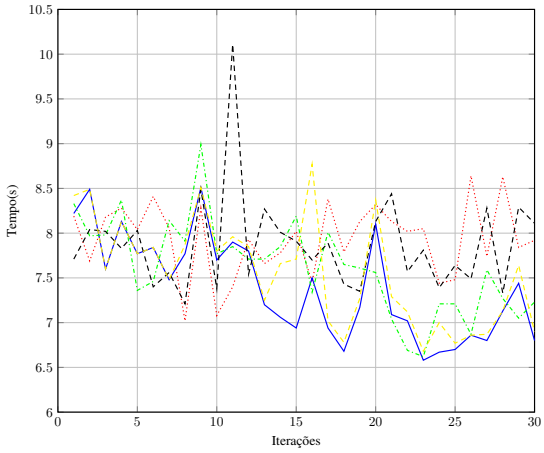


Fig. 5. H2.

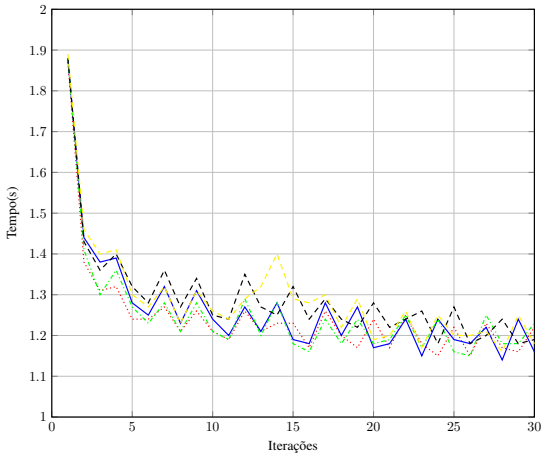


Fig. 6. Pmd.

As iterações do *benchmark* H2 (Figura 5) com `AG-SOA` apresentam melhoras a partir da 12ª iteração, com picos de até 25% e uma melhora geral de aproximadamente 15%. O `SPEA2-SOA` também obtém resultados melhores que o `SOA-Original` a partir da 12ª iteração, porém com desempenho 5% pior que o `AG-SOA`. Até a 10ª iteração, `SOA-Zhao` segue o mesmo padrão de `AG-SOA`, após ocorre uma queda de desempenho.

O *benchmark* PMD (Figura 6) é um pouco mais lento com o `AG-SOA` até a iteração 12 e, a partir daí, obtém em média o mesmo tempo de execução que o `SOA-Original`, não sendo perceptível uma melhora no desempenho. O `SPEA2-SOA` se mantém semelhante ao `SOA-Original`. Executando PMD, `SOA-Zhao` obteve o pior desempenho entre as iterações 13 e 16, após o desempenho segue o padrão das outras versões.

O último experimento foi realizado com o XALAN (Figura 7) e, semelhante a PMD, o `AG-SOA` apresenta desempenho semelhante ao sistema adaptativo original, porém um pouco diferente que o *benchmark* anterior devido aos seguintes pontos:

- entre a 12ª e 16ª iteração é possível perceber uma melhora de até 10%;
- uma piora na iteração 21 de 8%;
- e uma melhora média de 11,5% nas últimas cinco iterações.

Para o XALAN, o `SPEA2-SOA` é pior que o `AG-SOA`, como também o `SOA-Original`. `AG-SOA` chega a obter um desempenho 20% superior ao obtido por `SOA-Zhao`.

Os experimentos indicam que `SOA` proposto sem *rollback* não apenas deixa de obter melhor desempenho que o `AG-SOA`, como também se mostra mais lento que o sistema adaptativo padrão da `Jikes RVM`, com picos de perda de desempenho de até 25% (como percebido no *benchmark* XALAN) e média de perda de 5%. A única exceção é o *benchmark* PMD, cujo desempenho é ligeiramente pior do que `SOA-Original` e o `AG-SOA`.

O `SPEA2-SOA` mostra-se pior ou igual ao `AG-SOA` em todos os experimentos e pior que o `SOA-Original` em alguns experimentos. Isto é devido a complexidade das operações realizadas pelo `SPEA2`, como verificar a dominância de todos os membros da população para calcular o *fitness*.

Como existem fatores aleatórios em um algoritmo genético, a medição de desempenho sem *rollback* foi utilizada para averiguar se os resultados obtidos são representativos. Caso o `AG-SOA` sem *rollback* obtivesse resultados semelhantes à versão com *rollback*, poderia indicar que as operações

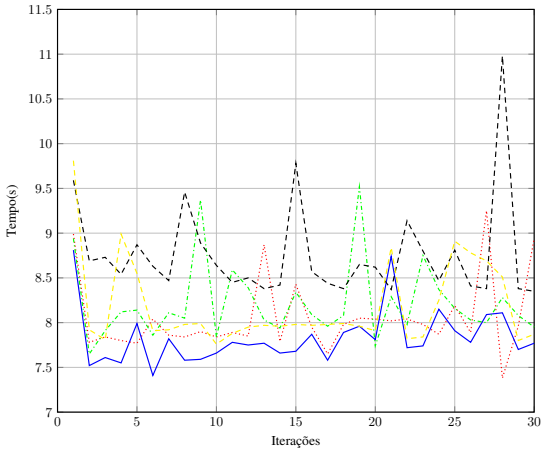


Fig. 7. Xalan.

realizadas durante o processo de busca não influenciavam tanto no desempenho, sendo o fator de maior peso a própria configuração padrão da Jikes RVM.

A implementação apresentada neste trabalho melhorou o desempenho de quase todos os *benchmarks* analisados utilizando o AG−SOA. Além disto, tal implementação conseguiu convergir mais rápido na maioria dos experimentos, sendo necessário cerca de 8 iterações apenas para o SOA proposto apresente um melhor desempenho do que o SOA original.

O fato do SOA proposto considerar uma troca dinâmica entre as diferentes árvores de otimização possibilita que os métodos troquem diversas vezes o plano de otimização, até que o término da busca por um bom plano seja alcançado. Tal estratégia possibilita uma rápida convergência para melhores resultados.

É importante perceber que as operações realizadas pelo SOA, aqui proposto, possuem um impacto no tempo total de execução da aplicação. Por isto é importante medir tal impacto.

A Figura 8 apresenta a proporção do tempo consumido em cada *benchmark*, pelo AG−SOA e pelo SPEA2−SOA.

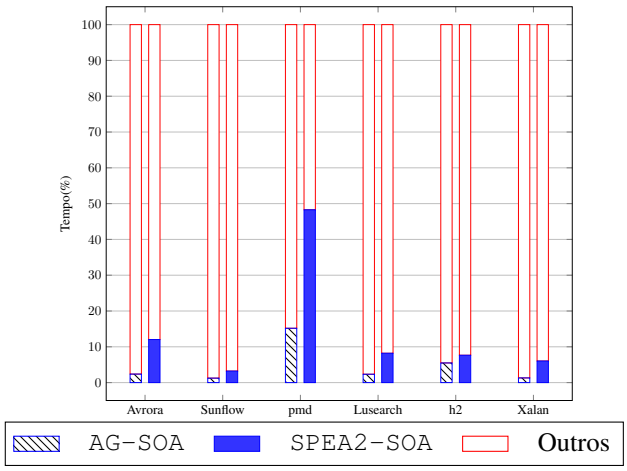


Fig. 8. Tempo gasto pelo novo SOA.

O custo do AG−SOA é menor de 6% do tempo total de execução, para a maioria dos *benchmarks*. PMD, para o qual o SOA proposto não apresentou melhorias, foi o único *benchmark* cujo custo do AG−SOA ultrapassa 6%. De fato, para este *benchmark* o custo do SOA proposto é de 15% do tempo total de execução. O SPEA2−SOA possui um custo maior que o AG−SOA em todos os *benchmarks*, chegando este custo a ser 3 vezes maior.

Os resultados dos experimentos indicam que o novo mecanismo de busca do sistema adaptativo funciona corretamente e é capaz de selecionar bons planos de otimização para diversas aplicações.

A implementação com AG, organizado em uma estrutura de árvore, converge a um ponto ótimo do sistema (máquina virtual e aplicação), o que resulta em um melhor desempenho em relação à versão original do SOA. A implementação com SPEA2 não atinge os mesmos resultados, o que demonstra não ser eficiente. Portanto, os resultados indicam duas coisas. Primeiro, nem sempre utilizar um algoritmo complexo irá geral bons resultados. Segundo, alterar um sistema de compilação adaptativo, embora aumente o o custo do sistema geral, tem o potencial de melhorar o desempenho final.

V. Conclusões

O conjunto de otimizações, aplicado pelo compilador durante o processo de geração de código final, irá impactar o desempenho do código final. Devido às otimizações alterarem a estrutura do código original; embora sem alterar a sua semântica. De fato, é esperado que o código final tenha uma melhor qualidade, quando comparado com o código original. Contudo, tal premissa nem sempre é verdadeira; pois uma determinada otimização pode melhorar a qualidade de um determinado programa mas não necessariamente de outro. Isto indica, que o problema de seleção de otimizações é um problema dependente de programa. Portanto, é importante desenvolver estratégias que considerem individualmente cada programa e assim adapte o sistema ao programa em questão.

Nos últimos anos diversos trabalhos apresentaram diferentes estratégias para mitigar o problema de seleção de otimizações. Embora a maioria dos trabalhos apresentem estratégias para compiladores *standalone*, os trabalhos que apresentam estratégias para compiladores JIT demonstram que é possível adaptar tais compiladores as características do programa em execução e desta forma reduzir o tempo de execução de tal programa.

No contexto de compiladores JIT em JVMs, este artigo apresenta um SOA capaz de melhorar o desempenho dos programas a medida que adapta o sistema as características específicas do programa em questão. Tal SOA utiliza um algoritmo genético convencional, permitindo que a JVM realize otimizações adaptativas de forma inteligente.

Os resultados alcançados demonstram que o sistema proposto chega a obter um desempenho médio 11% superior à versão original, com picos de até 25%. Tais resultados indicam que as escolhas feitas durante a implementação de um sistema de otimização adaptativa automático refletem diretamente no desempenho.

Outro ponto importante a ser destacado é o fato da construção de um sistema adaptativo considerar o consumo dos recursos computacionais. Como pode ser visto nos resultados obtidos, o tempo gasto para executar as operações do SOA proposto foram mínimas quando comparadas ao tempo total de execução do programa. Contudo, isto ao utilizar um algoritmo simples. Pois utilizar um algoritmo mais complexo, como o SPEA2, tende a aumentar o custo do sistema; além de não garantir a obtenção de bons resultados.

A próxima área a ser explorada é modificar o cálculo de *fitness* para não depender somente das amostragens realizadas. Siraichi e Outros [20] utilizaram um método para representar funções quentes inspirado no DNA biológico, codificando as instruções do compilador em uma *string*. O objetivo é adaptar tal abordagem para ao SOA, realizando uma comparação entre as otimizações dos indivíduos da população de um nó recentemente inserido na árvore. Neste caso, o grau de similaridade de dois indivíduos originados da recombinação com outros da população será levado em consideração para calcular o *fitness* de cada um. Desta forma, mesmo que algum membro da população não seja selecionado, é possível obter um valor da função *fitness* aproximado, o qual será uma previsão de seu possível desempenho.

REFERÊNCIAS

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

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

[3] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan, "Extendable pattern-oriented optimization directives," *ACM Transaction on Architecture and Code Optimization*, vol. 9, no. 3, pp. 14:1–14:37, Oct. 2012.

[4] M. Tartara and S. C. Reghizzi, "Continuous learning of compiler heuristics," *ACM Transactions on Architecture Code Optimization*, vol. 9, no. 4, pp. 46:1–46:25, Jan. 2013.

[5] 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.

[6] M. Arnold, M. Hind, and B. G. Ryder, "Online feedback-directed optimization of java," in *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, vol. 37, no. 11. Rutgers University, Piscataway, NJ, 08854: ACM, 2002, pp. 111–129.

[7] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani, "Effectiveness of cross-platform optimizations for a java just-in-time compiler," *SIGPLAN Notices*, vol. 38, no. 11, pp. 187–204, Oct. 2003.

[8] J. Cavazos and M. F. O'boyle, "Method-specific dynamic compilation using logistic regression," in *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, vol. 41, no. 10. ACM, 2006, pp. 229–240.

[9] M. R. Jantz and P. A. Kulkarni, "Performance potential of optimization phase selection during dynamic jit compilation," *SIGPLAN Notices*, vol. 48, no. 7, pp. 131–142, Mar. 2013.

[10] J. Zhao, "Jikes rvm adaptive optimization system with intelligent algorithms," Ph.D. dissertation, PhD thesis, School of Computer Science, The University of Manchester, 2004.

[11] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, and H. Lee, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, vol. 41, no. 10. New York, NY, USA: ACM, Oct. 2006, pp. 169–190.

[12] L. G. Silva, C. A. P. S. Martins, and L. F. W. Goes, "Jvm configuration parameters space exploration for performance evaluation of parallel applications," *IEEE Latin America Transactions*, vol. 13, no. 8, pp. 2776–2784, Aug 2015.

[13] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, Jan. 2000.

[14] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive optimization in the jalapeno jvm," *SIGPLAN Notices*, vol. 46, no. 4, pp. 65–83, May 2011.

[15] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," in *Proceedings of the Evolutionary Methods for Design, Optimization and Control With Applications to Industrial Problems*, Zurich, Switzerland, 2001.

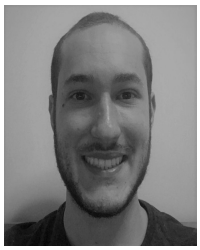
[16] F. G. Bulnes, R. Usamentiaga, D. F. Garcia, and J. Molleda, "A parallel genetic algorithm for optimizing an industrial inspection system," *IEEE Latin America Transactions*, vol. 11, no. 6, pp. 1338–1343, Dec 2013.

[17] G. Pedraza, M. Diaz, and H. Lombera, "An approach for assembly sequence planning by genetic algorithms," *IEEE Latin America Transactions*, vol. 14, no. 5, pp. 2066–2071, May 2016.

[18] P. H. da Silva Palhares and L. da Cunha Brito, "Constrained mixed integer programming solver based on the compact genetic algorithm," *IEEE Latin America Transactions*, vol. 16, no. 5, pp. 1493–1498, May 2018.

[19] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive optimization in the jalapeno jvm," in *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, vol. 35, no. 10. New York, NY, USA: ACM, 2000, pp. 47–65.

[20] M. Y. Siraichi, C. Tonetti, and A. F. da Silva, "A design space exploration of compiler optimizations guided by hot functions," in *Proceedings of the International Conference of the Chilean Computer Science Society*. Washington, DC, USA: IEEE, 2016, pp. 1–12.



Adriano Ferrari Cardoso é bacharel em Ciência da Computação pela Universidade Estadual de Maringá, Paraná/Brasil. Seus interesses de pesquisa envolvem compiladores e arquitetura e organização de Compiladores.

Caio Henrique Segawa Tonetti é bacharel em Ciência da Computação pela Universidade Estadual de Maringá, Paraná/Brasil. Atualmente é aluno de pós-graduação no Programa de Pós-graduação em Ciência da Computação da Universidade Federal de Minas Gerais, Minas Gerais/Brasil. Seus interesses de pesquisa envolvem compiladores, arquitetura e organização de Compiladores, IOT e data science.

Anderson Faustino da Silva é doutor em Engenharia de Sistemas e Computação pela COPPE/Universidade Federal do Rio de Janeio, Rio de Janeiro/Brasil. Atualmente é professor associado da Universidade Estadual de Maringá, Paraná/Brasil. Seus interesses de pesquisa envolvem compiladores, arquitetura e organização de Compiladores e programação paralela.