Proposta de um Mecanismo Adaptativo para Seleção de Planos de Compilação em Compiladores JIT

A. S. Mignon; R. L. A. Rocha

***Resumo*—Linguagens de programação que têm como uma de suas principais características a portabilidade de seus programas têm utilizado sistemas baseados em compilação *Just-in-Time* (JIT) para melhorar o desempenho de seus programas. Esses sistemas utilizam políticas de compilação para identificar *se* e *quando* diferentes regiões do programa devem ser compiladas e também *como*, através de um *plano de compilação*, compilar essas regiões. Em geral, os planos de compilação são definidos manualmente e um único plano de compilação é aplicado a todas as regiões do programa. Técnicas para a geração automática de planos de compilação também têm sido utilizadas. Entretanto, elas apenas selecionam, dentro de um conjunto fixo de planos de compilação, o melhor plano para uma determinada unidade de código. Este trabalho apresenta uma proposta de um mecanismo adaptativo para a seleção de planos de compilação em compiladores JIT. Esse mecanismo tem por objetivo permitir que o processo de seleção de planos de compilação adapte-se ao histórico de utilização de uma unidade de código e também a mudanças na plataforma de execução do programa.**

Keywords*—*otimização em compiladores, compiladores JIT, técnicas adaptativas

I. Introdução

Linguagens de programação como Java e C# têm como uma de suas principais características a portabilidade de seus programas, isto é, eles podem ser executados em qualquer dispositivo equipado com a máquina virtual correspondente. Entretanto, a portabilidade limita o formato de distribuição do programa para uma forma que é independente de qualquer processador ou sistema operacional. Para lidar com arquivos nesse formato, máquinas virtuais são usadas para a interpretação do programa ou compilação dinâmica antes da execução do programa. Como a execução interpretada de um programa é inerentemente lenta, é essencial que se utilize um modelo de compilação dinâmica ou compilação *Just-in-Time* (JIT) para atingir um desempenho em época de execução eficiente para tais programas [1].

A compilação JIT opera em época de execução o que contribui para o tempo total de execução do programa e, se realizada imprudentemente, pode piorar ainda mais a execução ou tempo de resposta do programa. Portanto, políticas de compilação JIT precisam ser cuidadosamente ajustadas para identificar *se* e *quando* diferentes regiões do programa são compiladas para atingir o melhor desempenho. Além disso, *como* compilar regiões do programa também é um importante componente de qualquer política de compilação.

Um dos componentes de uma política de compilação é o *plano de compilação*. Em um compilador otimizador, o plano de compilação é responsável por guiar as transformações aplicadas sobre um programa (ou parte dele) como o objetivo de melhorar determinada característica como, por exemplo, tempo de execução ou consumo de energia. As primeiras máquinas virtuais com compilação JIT utilizavam políticas estáticas de compilação. As máquinas virtuais modernas utilizam diversas técnicas para selecionar dinamicamente um subconjunto de código para compilação e também para selecionar o melhor plano de compilação. Tais técnicas são conhecidas na literatura como técnicas de otimização adaptativa [2]. Entretanto, apesar destas técnicas utilizarem o termo adaptativo, elas não modificam a estrutura ou o conjunto de regras a serem aplicadas pelo compilador JIT. Elas somente selecionam um determinado plano de compilação dentro de um conjunto de planos a partir das características da unidade de código que será compilada.

Este trabalho apresenta uma proposta de um mecanismo adaptativo, baseado em aprendizado por reforço, para a seleção de planos de compilação em compiladores JIT. O objetivo principal é que o processo de aprendizado seja contínuo e a escolha de um plano de compilação para uma unidade de código possa variar de acordo com seu histórico de utilização e arquitetura na qual ela está executando.

Este trabalho está dividido da seguinte forma: na seção II, baseada em [3], apresenta-se os principais conceitos de compiladores JIT e sua arquitetura genérica. Na seção III, baseada em [2], apresenta-se algumas técnicas disponíveis na literatura denominadas de otimização adaptativa. Na seção IV apresenta-se brevemente os conceitos da tecnologia adaptativa utilizados neste trabalho. Na seção V apresenta-se a proposta do mecanismo adaptativo. Por fim, na seção VI apresenta-se as conclusões e trabalhos futuros.

II. Compiladores JIT

Linguagens de programação que têm como um de seus objetivos permitir a portabilidade entre diferentes plataformas utilizam o modelo de máquinas virtuais para a execução de seus programas. Em geral, utilizam um modelo de interpretação de código para permitir a independência do *hardware* e do sistema operacional em que executam. Entretanto, código interpretados são mais lentos que códigos compilados. Um meio para se evitar a sobrecarga de interpretação é permitir ao interpretador gerar código de máquina para um segmento de código interpretado, antes que esse segmento de código seja necessário. Isto é chamado de compilação *Just-in-Time* (JIT) [4]. Uma vantagem dessa técnica é que o processo de compilação é ocultado do usuário, e que o código gerado pode ser adaptado para um processador particular. Ela possibilita ainda que o código compilado passe por um processo de otimização.

Esta técnica é adequada somente se o processo de compilação for rápido o suficiente para ser aceitável, já que o tempo de compilação está incluído no tempo total de execução do programa, pois a compilação ocorre durante sua execução.

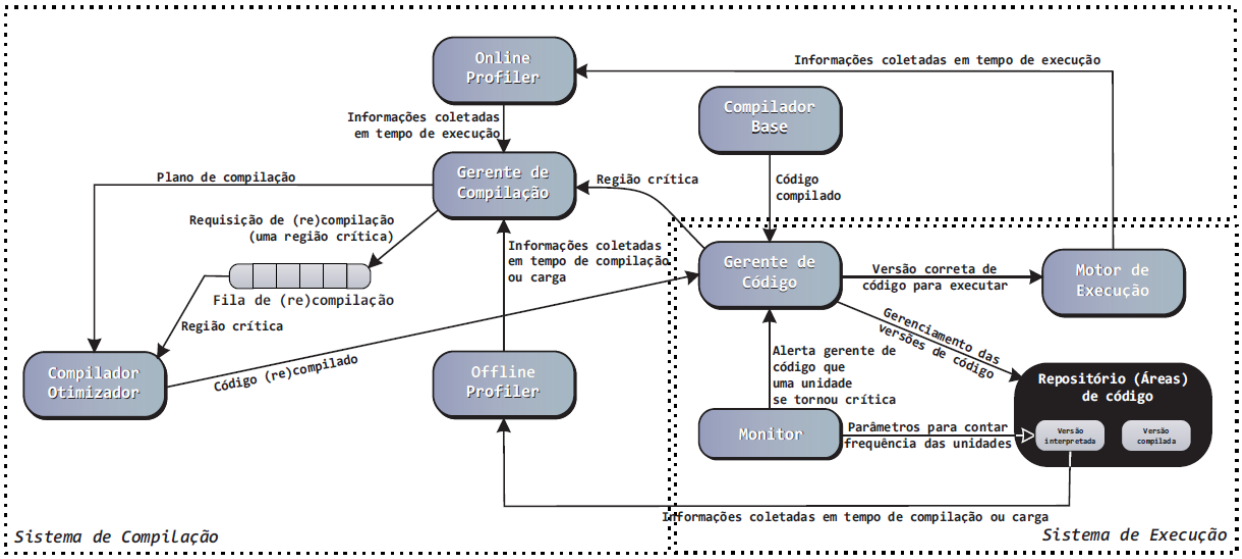

Figura 1. Arquitetura genérica de uma máquina virtual com compilação JIT. Extraída de [3].

Uma questão crítica é decidir *o que* e *quando* um segmento de código deve ser compilado e também *como* esse segmento de código deve ser compilado.

As primeiras máquinas virtuais com compilação JIT utilizavam políticas estáticas simples para decidir quais porções de código deviam ser compiladas e compilavam cada segmento de código com um conjunto fixo de otimizações [3]. Máquinas virtuais modernas utilizam diversas técnicas para selecionar dinamicamente um subconjunto de código para compilação e otimização. Elas somente compilam partes do código que contribuem significantemente para o tempo de execução do programa; esses códigos são chamados de regiões críticas (*hot spots*) do programa. Em alguns casos utilizam múltiplos níveis de qualidade na geração do código: uma compilação rápida produz código de qualidade moderada para a maior parte do código e, um código mais sofisticado, porém de compilação mais lenta, para regiões críticas mais importantes [4].

Os compiladores JIT podem ser classificados de duas formas em relação ao seu funcionamento: interpretação mais compilação ou compilação mais recompilação. Compiladores que utilizam a primeira forma interpretam todas as unidades de código do programa (funções ou métodos) e geram código nativo para regiões críticas com um compilador otimizador. Compiladores que utilizam a segunda forma compilam todas as unidades de código para código nativo utilizando um compilador base rápido e as regiões críticas são [3]: recompiladas apenas uma vez utilizando um compilador otimizador ou; recompiladas várias vezes de acordo com as políticas estabelecidas por um gerente de compilação.

A Figura 1, extraída de [3], apresenta os módulos de uma arquitetura genérica de uma máquina virtual com compilação JIT. A seguir apresenta-se uma breve descrição de cada um dos módulos.

O **compilador base** é responsável por gerar código nativo em época de execução para todas as unidades de código invocadas pelo sistema, antes da primeira invocação. Ele tem por objetivo gerar o código nativo de forma rápida, sendo assim, não aplica otimizações às unidades compiladas.

O **compilador otimizador** é um dos principais componentes da arquitetura apresentada. Ele é responsável por gerar código nativo em época de execução para as regiões críticas do programa. Em geral, todo código compilado é otimizado utilizando-se um conjunto de otimizações pré-definido. Esse conjunto de otimizações é descrito em um *plano de compilação*. As otimizações contidas no plano de compilação e o modo como elas são aplicadas satisfazem a decisão de projeto de cada desenvolvedor. Contudo, pode-se utilizar técnicas para ajustar automaticamente as configurações do compilador, alterando o conjunto de otimizações e também a ordem em que elas são aplicadas em uma determinada região crítica. Algumas destas técnicas são apresentadas na seção III.

As técnicas para ajuste automático das configurações do compilador utilizam informações de perfil das unidades de código, coletadas em época de compilação pelo *offline profiler* ou em época de execução pelo *online profiler*, para especializar o código com o objetivo de torná-lo mais eficiente.

O **offline profiler** tem por objetivo coletar informações em época de compilação ou de inicialização do programa. Essas informações são usadas pelo compilador otimizador para guiar o processo de geração de código.

O **online profiler** tem por objetivo coletar informações em época de execução do programa. Ele monitora a execução das unidades de código e coleta informações que possibilitam a geração de código mais especializado para essas unidades. Dentre as informações coletadas têm-se: tipos de variáveis, tamanho da unidade de código, quantidade de parâmetros e seus tipos, existência de chamadas aninhadas, existência de recursividade, tempo de interpretação, tempo de compilação e tempo de execução.

O **monitor** tem por objetivo atualizar os parâmetros de frequência que são usados para medir a frequência de execução das unidades de código do programa. Esses parâmetros são instrumentalizados nas unidades de código e podem ser de dois

tipos: *contadores* ou *fração de tempo*. Quando o parâmetro atinge um limite pré-definido, o monitor alerta ao gerente de código que a unidade de código se tornou frequente.

O **gerente de código** tem por objetivo gerenciar as áreas de código do programa. Ele executa basicamente duas tarefas: 1) envio da versão correta da unidade de código acionada para execução, priorizando o envio do código nativo mais atual; 2) envio de uma unidade de código para (re)compilação. O gerente de código interage com o monitor para identificar quais unidades de código são frequentes. Assim que elas são detectadas o gerente de código as envia para o gerente de compilação para que seja providenciada uma nova versão desta unidade.

O **gerente de compilação** tem por objetivo criar um *plano de compilação* para o compilador otimizador utilizar durante a geração de código. O gerente de compilação pode utilizar as informações obtidas pelo *offline* e/ou *online profilers* para a criação de um plano de compilação adequado para uma determinada unidade de código. Em geral, um plano de compilação define um conjunto de otimizações pré-determinado que é o mais adequado para uma determinada região de código. Além disto, esse plano pode definir um conjunto de otimizações mediante a análise do próprio programa em execução. Portanto, neste último caso as otimizações serão habilitadas durante a execução do programa, bem como a ordem em que elas serão aplicadas.

O **repositório de código** tem por objetivo armazenar versões de código interpretado e/ou compilado. Em sistemas do tipo interpreta e compila, o código interpretado é o padrão de execução. Nos sistemas que compilam e recompilam é somente mantida a área de código nativo. A medida que as regiões críticas são (re)compiladas a área de código nativo se expande. Em geral, a prioridade de execução é sempre do código especializado para o comportamento ativo.

O **motor de execução** tem por objetivo executar as unidades de código. Em sistemas do tipo interpretação mais compilação, ele é responsável por interpretar o código e também invocar o código nativo. Nos sistemas do tipo compilação mais recompilação ele é responsável apenas pela invocação do de código nativo.

III. Otimização Adaptativa em Compiladores JIT

O advento das arquiteturas de máquinas virtuais e o processo de compilação dinâmica introduziu novos desafios para se atingir um alto desempenho de execução dos programas. Para lidar com esses desafios, diversas pesquisas têm sido realizadas em tecnologia de otimização adaptativa [2]. Este tipo de tecnologia tem como objetivo melhorar o desempenho de execução dos programas através do monitoramento do seu comportamento e utilizando as informações coletadas em época de execução para direcionar decisões de otimização.

O termo *adaptativo* é utilizado nesta seção conforme definição dos autores dos trabalhos citados. Esse termo não está de acordo com a definição utilizada na Tecnologia Adaptativa. Na seção V apresenta-se um termo mais adequado, de acordo com [5], para este tipo de técnica de otimização adaptativa.

Em [2] os autores dividem as técnicas para otimização adaptativa em quatro categorias: 1) *otimização seletiva* são técnicas utilizadas para determinar quando e em quais partes do programas devem ser aplicadas otimizações em tempo de execução; 2) *profiling techniques for feedback-directed optimization (FDO)* são técnicas para coletar informações do perfil de execução dos programas com alta granularidade; 3) *feedback-directed code generation* são técnicas usadas utilizando as informações de perfil do programa para melhorar a qualidade do código gerado pelo compilador otimizador; 4) *outras técnicas FDO* que usam informações de perfil do programa para melhorar o seu desempenho.

O processo de *Feedback Directed Optimization* coleta as informações sobre o tempo de execução utilizando técnicas de perfil (*profile*) de execução do programa. A seguir, são citadas algumas destas técnicas:

- *Contadores de Desempenho*: Contadores são inseridos no código para detectar os caminhos utilizados com mais frequência. Esses contadores são incrementados no momento em que determinada parte do programa é executada.
- *Amostragem*: O sistema coleta um subconjunto representativo de uma classe de eventos, permitindo um limitado gasto de tempo para se obter informações do perfil do programa.
- *Monitoramento de Serviços*: Monitora os serviços requisitados pelo programa durante sua execução. São exemplos de serviços: a entrada e saída e o gerenciamento de memória.
- *Instrumentação*: Adição de código para coleta de informações.

Um sistema que utiliza otimização seletiva é composto basicamente de três componentes [2]: 1) um mecanismo de perfil para identificar regiões de código para otimização futura; 2) um componente de decisão para selecionar quais otimizações aplicar a cada região de código; 3) um compilador otimizador dinâmico para realizar as otimizações selecionadas. Em um sistema com otimização seletiva é possível aplicar para cada unidade de código um conjunto diferente de otimizações, dependendo das informações de perfil coletadas.

O componente de decisão em um sistema de otimização seletiva utiliza heurísticas para guiar o processo de transformações do código para um determinado objetivo. Em geral, essas heurísticas são definidas manualmente pelos desenvolvedores do compilador. Este processo é caro, consome muito tempo e é propenso a erros, e pode levar a um desempenho sub-ótimo [6]. Para lidar com tais questões, têm-se utilizado aprendizado de máquina em compiladores com o objetivo de automatizar o processo de geração de heurísticas [7], [8].

Um dos primeiros trabalhos utilizando aprendizado de máquina em compiladores foi para lidar com transformações *looping unrolling* [9]. Este trabalho utilizou o compilador GNU FORTRAN alterando-se a heurística de ativação de *loop unrolling* para utilizar um modelo aprendido através de exemplo. O algoritmo de aprendizado utilizado era baseado

em árvores de decisão, o que permitia que o modelo gerado pudesse ser interpretado por um especialista.

Cavazos e O'boyle [7] aplicaram a técnica de *logistic regression* para habilitar ou desabilitar transformações em planos de compilação na Jikes RMV para a compilação específica de método. Estudos apresentados mostraram que as transformações poderiam obter melhores resultados se fossem selecionados planos de compilação específicos para cada método do programa. A Jikes RVM foi modificada para compilar métodos individuais variando as transformações aleatoriamente, uma medição de tempo foi adicionada a cada método e os planos de compilação com melhor tempo de execução foram selecionados e usados para o treinamento da *logistic regression*. A versão utilizando aprendizado de máquina superou a configuração padrão presente na Jikes RVM.

Hoste, Georges e Eeckhout [6] apresentam uma proposta de sintonia automática de planos de compilação em um compilador JIT baseada em uma busca evolucionária com múltiplos objetivos. A ideia é aplicar uma sintonia fina no compilador para cenários específicos: uma dada plataforma de *hardware*, um conjunto de aplicações, ou um conjunto de entradas para aplicações de interesse. A sintonia inicia com uma exploração dos planos de compilação para descobrir quais deles são ótimo de Pareto (*Pareto-optimal*), e então um subconjunto deles é atribuído ao compilador JIT.

Sanchez et al. [8] aplicaram a técnica de *Support Vector Machine* (SVM) para habilitar ou desabilitar transformações em planos de compilação em uma máquina virtual de grande porte, a Testarossa da IBM, para a compilação específica de método. O processo de modificação dos planos de compilação partia do plano original da Testarossa realizando pequenas alterações ao longo do tempo em busca de melhorias locais. O desempenho de vazão não sofreu alterações, mas foi possível melhorar o desempenho de inicialização.

Kulkarni e Cavazos [10] apresentaram uma técnica que automaticamente seleciona a melhor ordem prevista de transformações para métodos diferentes de um programa. A técnica utiliza uma rede neural artificial para predizer a ordem de transformações que é mais benéfica para um determinado método. As redes neurais artificiais são automaticamente inferidas usando *NeuroEvolution for Augmenting Topologies* (NEAT). A técnica foi implementada na máquina virtual Jikes RVM e apresentou melhorias de desempenho em um conjunto de *benchmarks* se comparado com a aplicação de transformações em uma ordem fixa.

Leather, Bonilla e O'boyle [11] apresentaram um mecanismo para automaticamente buscar características de um programa que tem um maior impacto na qualidade de heurísticas de aprendizado de máquina. O espaço de características é descrito por uma gramática e é então percorrido com programação genética. Esse mecanismo foi aplicado para a transformação *loop unrolling* no compilador GCC.

IV. Tecnologia Adaptativa

Adaptatividade é a capacidade que um sistema tem de modificar seu próprio comportamento, em resposta a algum estímulo de entrada ou ao seu histórico de operações, sem a interferência de qualquer agente externo [12]. A tecnologia adaptativa trata de técnicas e dispositivos que tem característica adaptativa. Dispositivos adaptativos são descrições abstratas de problemas que tem comportamento dinâmico. Essas descrições são associadas a dispositivos não adaptativos subjacentes que representam problemas com comportamento estático.

Um dispositivo adaptativo é constituído de: 1) uma componente não-adaptativa, na forma de um dispositivo subjacente guiado por regras, o qual serve como base para a construção da versão adaptativa; 2) um mecanismo adaptativo acoplado ao dispositivo subjacente, a qual é capaz de alterar o conjunto de regras que define o comportamento desse dispositivo subjacente [13].

O mecanismo adaptativo é composto por um conjunto de funções especiais, denominadas *funções adaptativas*, que são acionadas no momento da aplicação de alguma regra definida pelo dispositivo adaptativo. As funções adaptativas é que possibilitam uma forma de alterar o conjunto de regras do dispositivo adaptativo.

As regras adaptativas são formuladas associando-se a uma regra não-adaptativa do dispositivo subjacente duas funções adaptativas. Uma função adaptativa a ser executada antes e a outra depois da mudança de configuração do dispositivo. Caso não se queira executar nenhuma ação antes e/ou depois da mudança de configuração, associa-se a regra não-adaptativa uma função adaptativa nula.

Durante a operação do dispositivo adaptativo, uma vez escolhida uma regra adaptativa a ser aplicada, executa-se inicialmente a função adaptativa anterior, aplica-se a regra a ela associada, e por fim, executa-se a função adaptativa posterior.

Na próxima seção, apresenta-se um mecanismo com adaptatividade básica para a seleção de planos de compilação em compiladores JIT.

V. O Mecanismo Adaptativo

Em geral, os planos de compilação utilizados pelo compilador otimizador em compiladores JIT são definidos de forma estática e aplicados em todas as unidades de código que são compiladas. Com a utilização de técnicas de otimização adaptativa, apresentadas na seção III, os planos de compilação são selecionados dinamicamente a partir das características da unidade de código, através de informações de seu perfil de utilização. Entretanto, essas técnicas ditas adaptativas não alteram o comportamento do dispositivo, isto é, do compilador JIT. Elas apenas selecionam, dentro de um conjunto fixo de planos de compilação, o melhor plano para uma determinada unidade de código. Portanto, de acordo com a terminologia apresentada em [5], essas técnicas ditas adaptativas seriam melhor definidas como técnicas de um *processo configurável*.

Mesmo as técnicas que utilizam aprendizado de máquina podem ser classificadas como configuráveis. Em geral, os trabalhos relacionados ao aprendizado de máquina em compiladores utilizam técnicas de aprendizado supervisionado [14]. Com o uso destas técnicas, o aprendizado é feito através de exemplos fornecidos por algum supervisor externo em uma fase de treinamento. No caso do uso de aprendizado supervisionado em compiladores, o agente aprendiz é treinado para selecionar as melhores otimizações em relação à plataforma de execução

e ao conjunto de programas usados para o treinamento. Caso se queira utilizar uma nova plataforma de execução ou o perfil de utilização do programa se altere, o processo de treinamento deve ser refeito. Portanto, após o processo de treinamento inicial, é sempre selecionado, para uma determinada unidade de código, o mesmo plano de compilação.

A proposta deste trabalho é de um mecanismo adaptativo que permite o aprendizado contínuo na seleção de planos de compilação, isto é, permite que seja selecionado, para uma determinada unidade de código, diferentes planos de compilação ao longo do tempo, dependendo não somente das características da unidade de código a ser compilada como também do seu histórico de execução e dos planos de compilação usados anteriormente.

A realização deste mecanismo adaptativo é feita utilizando um modelo baseado em aprendizado por reforço [15] [16]. Aprendizado por reforço é uma forma de aprendizado de máquina em que um agente aprende através de sua interação com um ambiente. Ele permite que o aprendizado seja contínuo, através de um processo de tentativa e erro. O agente não é ensinado por meio de exemplos fornecidos por um supervisor externo. O uso de aprendizado por reforço em compiladores pode permitir que o processo de aprendizado seja contínuo e adaptável ao perfil de execução do programa ou também a mudanças de ambiente como, por exemplo, novas plataformas de *hardware* ou sistema operacional.

A Figura 2 apresenta os componentes básicos do dispositivo adaptativo. A camada subjacente é constituída pelo compilador JIT e os módulos apresentados na seção II. A camada adaptativa é constituída por um agente aprendiz, baseado em aprendizado por reforço, responsável selecionar o melhor plano de compilação para uma determinada unidade de código. Ao selecionar um plano de compilação, o mecanismo adaptativo pode alterar o plano de compilação previamente especificado no compilador JIT, alterando assim a regra de sua aplicação.

Um componente de *Controle* é adicionado ao *Compilador JIT*, na camada subjacente, com o objetivo de criar um mecanismo de interação entre o compilador JIT e o mecanismo adaptativo. Isso permite reduzir a quantidade de alterações necessárias no compilador JIT para o acoplamento do mecanismo adaptativo.

Os estímulos de entrada do dispositivo subjacente são as informações coletadas em época de compilação, pelo módulo *offline profiler*, e também pelas informações que são coletadas em época de execução, pelo módulo *online profiler*. No momento em que o compilador JIT identifica uma unidade de código que deve ser (re)compilada, ele aciona o componente *Controle*, que então passa as informações recebidas para o mecanismo adaptativo. Neste momento, o acionamento do mecanismo adaptativo é equivalente à chamada de uma função adaptativa anterior.

O mecanismo adaptativo então cria um determinado plano de compilação a partir das informações recebidas e de sua configuração atual e envia esse plano para o *Controle* que é responsável por alterar a regra de aplicação do plano de compilação no compilador JIT. O compilador JIT então compila a unidade de código de acordo com o plano de compilação recebido. Após a unidade de código ser executada, são obtidas informações de perfil dessa execução, que são

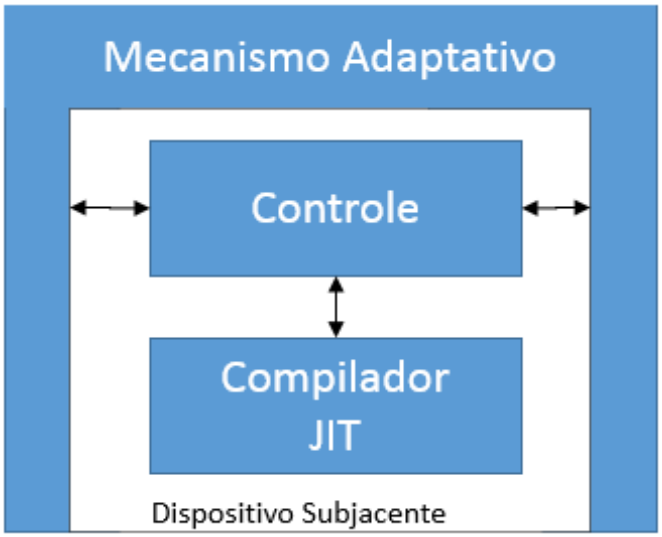

Figura 2. Componentes básicos do mecanismo adaptativo.

enviadas para o *Controle* e deste para o mecanismo adaptativo. Essas informações são utilizadas para atualizar as regras de inferência do modelo baseado em aprendizado por reforço e é equivalente à chamada de uma função adaptativa posterior.

VI. Conclusão

Este trabalho apresentou uma proposta de um mecanismo adaptativo, baseado em aprendizado por reforço, para a seleção de planos de compilação em compiladores JIT. O objetivo deste mecanismo é realizar um processo de aprendizado contínuo na seleção do melhor plano de compilação para uma determinada unidade de código permitindo que o processo de seleção se adapte ao histórico de utilização da unidade de código ou a mudanças na plataforma de execução do programa.

Como trabalhos futuros pretende-se implementar o mecanismo adaptativo utilizando a máquina virtual Jikes RVM [17] e realizar experimentos para a avaliação do modelo de aprendizado proposto com o objetivo de verificar se com o uso do mecanismo adaptativo houve alguma redução do tempo de execução das unidades de código e/ou do tempo de execução global do programa.

Referências

[1] P. A. Kulkarni, "Jit compilation policy for modern machines," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 773–788, 2011.

[2] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A survey of adaptive optimization in virtual machines," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 449–466, 2005.

[3] G. S. Oliveira and A. F. da Silva, "Compilação just-in-time: Histórico, arquitetura, princípios e sistemas," *Revista de Informática Teórica e Aplicada*, vol. 20, no. 2, pp. 174–213, 2013.

[4] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen, *Modern Compiler Design*, 2nd ed. New York, NY, USA: Springer Science & Business Media, 2012.

[5] J. J. NETO, "Um glossário sobre adaptatividade," in *3º Workshop de tecnologia Adaptativa–WTA, São Paulo*, 2009.

[6] K. Hoste, A. Georges, and L. Eeckhout, "Automated just-in-time compiler tuning," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 62–72.

[7] J. Cavazos and M. F. O'boyle, "Method-specific dynamic compilation using logistic regression," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 229–240, 2006.

[8] R. N. Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley, "Using machines to learn method-specific compilation strategies," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 257–266.

[9] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. London, UK: Springer-Verlag, 2002, pp. 41–50.

[10] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 147–162, 2012.

[11] H. Leather, E. Bonilla, and M. O'boyle, "Automatic feature generation for machine learning–based optimising compilation," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 1, pp. 14:1–14:32, Feb. 2014.

[12] J. J. Neto, "A small survey of the evolution of adaptivity and adaptive technology," *Revista IEEE América Latina*, vol. 5, no. 7, pp. 496–505, 2007, (in Portuguese).

[13] J. J. NETO, "Adaptatividade: Generalização conceitual," in *3º Workshop de tecnologia Adaptativa–WTA, São Paulo*, 2009.

[14] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, 1997.

[15] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.

[16] A. S. Mignon and R. L. A. Rocha, "epsilon-greedy adaptativo," in *Memórias do VIII Workshop de Tecnologia Adaptativa – WTA 2014*, 2014, pp. 57–62.

[17] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive optimization in the jalapeño jvm," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 47–65, Oct. 2000.