Compilação Just-In-Time: Histórico, Arquitetura, Princípios e Sistemas

George Souza Oliveira [1]
Anderson Faustino da Silva [1]

Resumo: Diversas implementações de linguagens de alto nível focam no desenvolvimento de sistemas baseados em mecanismos de compilação *just-in-time*. Esse mecanismo possui o atrativo de melhorar o desempenho de tais linguagens, mantendo a portabilidade. Contudo, ao preço da inclusão do tempo de compilação ao tempo total de execução. Diante disso, as pesquisas na área têm por objetivo balancear o custo de compilação com eficiência de execução. Os primeiros sistemas de compilação *just-in-time* empregavam estratégias estáticas para selecionar e otimizar as regiões de código propícias para gerar bom desempenho. Sistemas mais sofisticados aprimoraram tais estratégias com o objetivo de aplicar otimizações de forma mais criteriosa. Nesse sentido, este tutorial apresenta os princípios que fundamentam a compilação *just-in-time* e sua evolução ao longo dos anos, bem como a abordagem utilizada por diversos sistemas para garantir o balanceamento entre custo e eficiência. Embora seja difícil definir a melhor abordagem, trabalhos recentes mostram que o uso de uma boa estratégia para detecção e otimização de código, juntamente com recursos de paralelismo oferecidos pelas arquiteturas *multi-core* formarão a base dos futuros sistemas de compilação *just-in-time*.

Abstract: Several implementations of high-level languages focus on the development of just-in-time compilation systems. This strategy has the atractive of achieving performance without losing portability. However, in these systems the compile time is now included in overall runtime. Thus, researches have balanced the cost of compilation and execution efficiency. The first just-in-time compilation systems used static strategies to select and optimize some parts of the program to obtain performance. More sophisticated systems improves theses strategies in order to apply optimizations more carefully. In this context, this tutorial introduces the principles underlying the just-in-time compilation and its evolution over the years, as well as the approach used by several systems to ensure the balancing of cost and efficiency. Although it is difficult to determine the best approach, recent researches have shown that good techniques for code optimization with features offered by the multi-core architectures are the basis of future just-in-time compilation systems.

[1]Universidade Estadual de Maringá
Departamento de Informática
Programa de Pós-graduação em Ciênca da Computação
Avenida Colombo, 5790 - Bloco C56 - Maringá - PR - CEP 87020-900
{geo.soliveira@gmail.com,anderson@din.uem.br}

1 Introdução

As implementações originais de linguagens de programação interativas e de alto-nível [1, 2] focavam no desenvolvimento de máquinas virtuais eficientes que provessem portabilidade aos programas desenvolvidos nestas linguagens. Mais recentemente diversas pesquisas têm focado não apenas no desenvolvimento de máquinas virtuais que proveêm portabilidade, mas também em máquinas virtuais cuja execução seja eficiente, em termos de uso de recursos e tempo de execução. Para alcançar este objetivo, as máquinas virtuais atuais incluem em sua arquitetura um sistema de compilação *just-in-time* (JIT).

Em contraste com compiladores tradicionais, em uma máquina virtual com compilação JIT o tempo de compilação está incluído no tempo total de execução, pois a compilação ocorre durante a execução do programa. Portanto, é uma questão crítica decidir quando, o que e como compilar os programas. Mais especificamente, o sistema de compilação deve apenas compilar códigos se o tempo gasto na compilação for amortizado pelo desempenho ganho pelo código compilado. O custo de compilação e o desempenho obtido é uma questão crucial em ambientes com compilação JIT. Porém, tais ambientes possuem um grande benefício: eles podem explorar informações obtidas em tempo de execução para decidir quais otimizações aplicar, além de quais porções de código compilar.

As primeiras máquinas virtuais com compilação JIT utilizavam estratégias estáticas simples para escolher as porções de código que seriam compiladas. Tipicamente, tais máquinas compilavam cada porção de código com um conjunto fixo de otimizações durante a primeira invocação destas. Estas máquinas incluem os trabalhos com `Smalltalk-80` [3], `SELF-91` [4, 5, 6], e KAFFE [7].

Máquinas virtuais mais sofisticadas, ao invés de utilizarem estratégias estáticas simples, selecionam dinamicamente subconjuntos de códigos para otimizar. Esses subconjuntos são as regiões críticas do programa, isto é, porções de código executas com alta frequência. Exemplos de máquinas virtuais desta categoria incluem `SELF-93` [8, 9] e a primeira versão da Máquina Virtual `Java` (JVM - do inglês *Java Virtual Machine*) da IBM [10]. Embora, estas máquinas virtuais também incluam formas limitadas de otimizações baseadas em informações obtidas em tempo de execução, estes trabalhos não desenvolveram mecanismos gerais para otimizações baseadas em tais informações.

Recentemente diversos trabalhos têm explorado formas mais agressivas de compilação JIT [11, 12, 13], utilizando informações estruturais e comportamentais, obtidas em tempo de compilação e/ou execução, para adaptar o ambiente às características do programa. Contudo, alguns destes trabalhos não são totalmente automáticos, desta forma, não aparecendo em máquinas virtuais populares. Porém, estes têm demonstrado que otimizações baseadas em tais informações melhoram substancialmente o desempenho da máquina virtual.

Isto demonstra que o componente principal desta nova geração de máquinas virtuais é um mecanismo sofisticado de otimização baseado em informações coletadas dinamicamente. Contudo, tais técnicas não são aplicáveis em todos os programas. De fato, em alguns casos como, por exemplo, programas com curto tempo de execução, o alto custo de tais estratégias geralmente ocasionam uma perda de desempenho [14].

Técnicas de compilação JIT existem desde 1960 com LISP [15] e propagaram por diversas linguagens de programação ao longo dos anos. Contudo, somente com o advento da JVM [16] é que o termo JIT ficou realmente conhecido pela comunidade científica [17].

Sendo uma abordagem promissora, este tutorial tem por objetivo apresentar os fundamentos da compilação JIT, de modo a oferecer as seguintes contribuições:

- Apresentar um histórico da compilação JIT;
- Descrever a arquitetura padrão de um sistema de compilação JIT;
- Detalhar os princípios de compilação JIT; e
- Apresentar as principais características de diversos sistemas de compilação JIT.

O restante deste tutorial está organizado da seguinte forma. A Seção 2 apresenta um histórico da compilação JIT, destacando desde os primeiros trabalhos até os trabalhos mais recentes. A Seção 3 apresenta a arquitetura padrão utilizada em sistemas de compilação JIT, descrevendo cada possível componente da arquitetura. A Seção 4 aborda os princípios que norteiam a compilação JIT. A Seção 5 apresenta as principais características de sistemas atuais de compilação JIT. E por fim, a Seção 6 apresenta as considerações finais.

2 Histórico

O primeiro trabalho relacionado a compilação JIT data de 1960, o qual trata sobre a linguagem LISP [15]. Neste trabalho, McCarthy menciona a compilação de funções para uma linguagem de máquina rápida o suficiente para que a saída do compilador não necessitasse ser salva. Em 1968, os desenvolvedores da linguagem LC^2 (ou LCC) observaram que o código compilado pode ser gerado em tempo de execução a partir de um interpretador, simplesmente anotando as ações realizadas [18].

Dois anos depois, APL [19] incorporou duas técnicas que se relacionam com compilação JIT. *Drag-along* consistia em adiar a avaliação de expressões até que informações de contexto fossem reunidas. Por outro lado, *beating* transformava código para reduzir a quantidade de manipulação de dados envolvidos durante a avaliação das expressões. Em APL, os

tipos dos dados não são conhecidos até o tempo de execução, o que forçava o ambiente adiar a aplicação dessas técnicas.

Em 1971, Knuth [20], por dados de estudos empíricos, observou que a maior quantidade de tempo de execução gasto em um programa ocorre em uma minoria de código. Esta base serviu para que um trabalho posterior [21] investisse em técnicas que permitessem um balanceamento entre tempo de execução e espaço, os quais fundamentavam o argumento para compilação JIT na época. Dentre as técnicas desenvolvidas, uma tinha como objetivo executar os programas em duas versões de código: a nativa, para as regiões críticas, e a interpretada, para as demais. Essa técnica foi denominada código misto e ainda é aplicável em alguns sistemas JIT atuais. Por volta do ano de 1976, Brown desenvolveu a técnica *throw-away* [22], que buscava otimizar o espaço compilando partes do programa sob-demanda, em vez de aplicar compilação estática. Assim, esgotando a memória, todo o código já compilado (ou partes dele) seria descartado e recompilado posteriormente, caso necessário.

No trabalho de Hansen com `Adaptive Fortran`, de 1974, foi apresentada uma visão fundamental para sistemas JIT em geral [23]. Esta visão agregava três princípios básicos sobre a forma com que o código deveria ser tratado pelos sistemas JIT. Hansen descreveu a importância de qual código compilar, além de quando e como compilá-los. Ao mesmo tempo, Hansen procurou satisfazer tais fundamentos, utilizando e mantendo contadores vinculados a cada região de código para identificar aquelas que são críticas, embora aplicasse otimizações de código (o como) de forma muito conservativa sobre tais regiões.

Por outro lado, no desenvolvimento de `Smalltalk-80` em 1984 [3], Deutsch e Schiffman optaram por gerar código nativo em tempo de execução na medida em os métodos eram invocados, em vez de compilar somente os críticos, como fez Hansen. O código nativo era então armazenado em uma *cache*, para serem recuperados posteriormente. Mais tarde, `Smalltalk-80` serviu de influência para uma implementação de `SELF-91` [4, 5, 6].

`SELF-91` possui características bem peculiares, quando comparada com trabalhos anteriores. Nesta linguagem toda ação é dinâmica e os tipos não são conhecidos até o momento de execução. Essas características levaram ao desenvolvimento de técnicas de compilação e otimização JIT mais agressivas até aquele momento. Uma delas foi a técnica de customização de métodos [4] desenvolvida por Chambers e Ungar em 1989, cuja funcionalidade era especializar código para um tipo de dado específico coletado no tempo de execução. O sistema desenvolvido ainda introduziu o conceito de compilação adiantada para casos incomuns: eventos como *overflow* aritmético não teriam, a princípio, código compilado. Em vez disso, uma informação seria vinculada para que código fosse gerado somente se tais eventos ocorressem.

Em 1995 o trabalho de Hölzle, com a terceira geração de `SELF` (`SELF-93`) [8, 9], inovou com a implementação da técnica de otimização adaptativa: primeiramente, todo método

invocado era compilado por um compilador não otimizador rápido e somente as regiões críticas, detectados por meio de contadores, eram recompiladas por um compilador otimizador. Quatro anos antes, a linguagem `Java` surgiu para ser puramente interpretada. As JVMs iniciais, entretanto, eram interpretadores puros e resultavam em execuções muito lentas [24, 25], o que obrigou os seus desenvolvedores a buscarem nos conceitos de compilação JIT a solução para resolver tal problema.

A visão inicial de compilação JIT para `Java` foi dada por Cramer *et al* [24]. Neste trabalho, os autores observaram que somente a tradução de código Java para código nativo não era suficiente, além disso a otimização de código era também necessária. Seguindo essa visão, em 1997, Plezbert e Cytron apresentaram diferentes modelos teóricos de sistemas JIT, dentre eles, um que compila e executa código de forma concorrente. Dois destes modelos interpretavam as regiões não críticas e compilavam aquelas que são críticas após um certo tempo [26].

Porém, contrário às versões apresentadas por Plezbert e Cytron, Burke *et al* apresentaram em 1999 a máquina virtual Jalapeño [11], que somente compila código, embora ainda incorporasse um compilador JIT. A mesma idéia foi seguida um ano depois por Kazi *et al*, no desenvolvimento de JUDO [27].

No mesmo ano do surgimento de JUDO, o trabalho teórico de Reinholtz relatou a possibilidade de `Java` se tornar mais rápida que `C++` [28]. O autor se baseou nos compiladores JIT para `Java` na época que eram auxiliados por informações coletadas dinamicamente, além do surgimento de sistemas embarcados que, segundo o autor, forçariam a necessidade para o desenvolvimento de técnicas mais agressivas e eficientes de otimização de código, visto que tais sistemas possuem recursos muito limitados de hardware. Contudo, em 2010 Foleiss e Silva [29] demonstraram que `Java` não é mais rápida que `C++`, embora atualmente mantenham desempenho compatível.

Na verdade, as JVMs atuais obtém melhor desempenho de execução dos programas em comparação às JVMs primitivas,e uma parcela significativa desse ganho foi obtido pela seleção correta de regiões críticas dos programas, ao contrário do que previa Reinholtz. Embora otimizações de código sejam importantes, como descrito por Cramer *et al* [24], a sua aplicação não é suficiente. Tal afirmação foi justificada anos mais tarde, quando a maioria dos trabalhos focaram na precisão da obtenção de regiões críticas, obtendo resultados satisfatórios. O trabalho de Krintz, de 2003, empregou *offline* e *online profiling* para detectar tais regiões e alcançou melhorias de desempenho de 9%, em média [30]. No mesmo ano, Suganuma *et al* [31], adotaram o modelo de compilação baseada em região e obtiveram código 5% melhor, em termos de execução, além de reduzir o *overhead* de compilação entre 20 a 30% frente aos compiladores baseados em sub-rotinas (que eram os modelos mais comuns da época). Um ano depois, Kumar empregou no seu trabalho uma técnica denominada estimação relativa que, em tempo de execução determinava a criticidade dos métodos. Tal abordagem

garantiu melhorias de 4% nos programas avaliados [32].

Em 2006, Agosta *et al* [33] propuseram uma heurística estática para identificar blocos básicos e estimar o número de laços do programa. Algumas métricas foram elaboradas para fornecer estimativas da complexidade de um método. Essas estimativas acompanhavam o uso de contadores de frequência para detectar regiões críticas de forma mais precisa. Mais tarde, Lee *et al* propuseram um trabalho similar, com a característica adicional de prever a contagem de frequência dos métodos menos comuns, tomando por base a frequência dos métodos já compilados [34]. Esta abordagem reduziu o do custo de manutenção dos contadores dos métodos e alcançou bom desempenho.

Outra abordagem totalmente diferente foi utilizada por Gal *et al* na implementação de HOTPATHVM [35]. Os autores utilizaram o recurso de compilação baseada em caminhos de execução, anteriormente apresentado em DYNAMO [36]. O desempenho obtido não foi comparável àquele obtido por Lee *et al*, mas mostrou ser uma técnica vantajosa na aplicação de diversas otimizações de código [37]. Cinco anos depois, Hayashizaki *et al* propuseram uma técnica para detecção e remoção de falso laço dentro de um caminho de execução e conseguiram uma melhoria de desempenho de 37% na execução dos programas avaliados [38].

Por outro lado, no que tange as otimizações de código, há ainda poucas contribuições. Na verdade o estado-da-arte de compilação JIT consiste da otimização adaptativa, já presente em JIKES RVM (antes chamada JALAPEÑO) desde 1999 [11]. Este sistema implementa diferentes níveis de otimização, que consistem de um conjunto restrito de otimizações atribuídas manualmente. Recentemente, um trabalho atribuído a Hoste *et al* tem procurado utilizar busca evolucionária para atribuir otimizações para cada nível de forma automática, evitando assim o esforço manual [39].

Outro sistema que também implementa otimização adaptativa é a MÁQUINA VIRTUAL DA IBM [13], porém, os conjuntos de otimizações atribuídas são baseados em resultados de estudos empíricos realizados por Ishizaki *et al* [40]. Embora a contribuição de Hoste *et al* tenha sido focada em JIKES RVM, sua contribuição pode ser empregada em qualquer sistema que implementa otimizações em níveis.

As pesquisas voltadas a JIT, no entanto, não se limitam somente na precisão na detecção de regiões críticas ou nas otimizações de código. Trabalhos mais recentes têm focado na descoberta de boas políticas de compilação JIT em máquinas *multi-core* [41]. Em 2007, Kulkarni *et al* abordou questões sobre o escalonamento da *thread* de compilação e descobriu que o desempenho de execução dos sistemas JIT é proporcional ao aumento de sua prioridade [42]. Quatro anos depois, Kulkarni procurou alcançar, de forma experimental, a melhor política de compilação para máquinas *multi-core*, e descobriu que o aumento do número de *threads* de compilação, juntamente com a aceleração na detecção de regiões críti-

cas é uma boa estratégia para obter bons resultados [43]. Sistemas JIT, como JIKES RVM [44, 11], HOTSPOT SERVER [45], ILDJIT [46, 47] e a MÁQUINA VIRTUAL DA IBM [13] não seguem a melhor política descoberta nesses trabalhos, mas usufruem do real paralelismo oferecido pelas arquiteturas atuais.

De fato, a pressão sobre uma JVM rápida impulsionou diversas pesquisas voltadas à compilação JIT para a linguagem `Java` [17], que resultou em diversas técnicas e princípios que foram gradualmente inseridos e aperfeiçados para a construção dos diversos sistemas que surgiram. Atualmente é conhecido diversas JVMs, tais como: SUN HOTSPOT [48, 45], JUDO [27] e JIKES RVM [44, 11] que empregam um conjunto de otimizações sobre os programas. Não obstante, os mesmos princípios foram também empregados na construção de diversos outros sistemas para outras linguagens, como `Erlang` [49], `Lua` [50], `Prolog` [12], `JavaScript` [51] e `PHP` [52].

3 Arquitetura de Sistemas com Compilação JIT

No decorrer da evolução de sistemas JIT, diversas técnicas e princípios foram elaborados e incorporados na medida em que vinham sendo criados. Atualmente é conhecido sistemas que apostam na detecção de regiões críticas viáveis para otimização/compilação [50, 53, 49] e outros que sistematizam o emprego de otimizações de código [44, 13]. Contudo, mesmo que o objetivo de tais sistemas seja compilar e executar código nativo sem gerar degradação no desempenho, os meios empregados para alcançar esse objetivo podem ser bastante diversificados.

Um sistema JIT é projetado para funcionar em uma das seguintes formas: interpretação mais compilação ou compilação mais recompilação. Sistemas baseados na primeira forma interpretam todas as unidades de código do programa (função, método ou cláusula, dependendo das características da linguagem de programação para a qual o sistema foi desenvolvido) e geram código nativo para as regiões críticas com um compilador otimizador. Em contrapartida, sistemas que compilam e recompilam sempre executam código nativo. Nestes, toda unidade de código invocada é compilada por um compilador base rápido e as regiões críticas são:

- Recompiladas apenas uma vez por um compilador otimizador; ou
- Recompiladas várias vezes, caso o sistema verifique que as unidades já compiladas se tornaram inválidas (como no caso de um ou mais de seus dados terem o conteúdo modificado para outro tipo) ou se o gerente de compilação julgar que uma unidade já otimizada necessita ser reotimizada com técnicas mais agressivas.

A arquitetura padrão de um sistema JIT é como apresentado na Figura 1.

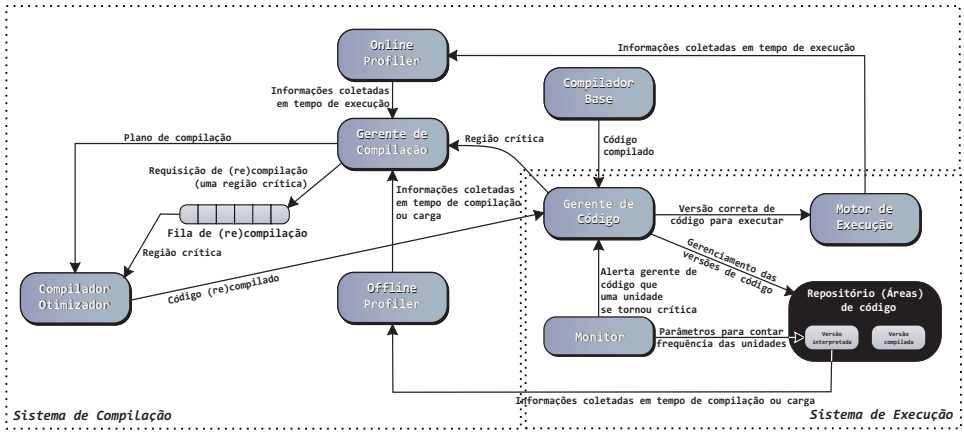

Figura 1. Arquitetura genéria de um sistema JIT.

Dentre os componentes do sistema de compilação o principal é o compilador otimizador. As otimizações aplicadas por ele e o modo como elas são aplicadas satisfazem a decisão de projeto de cada desenvolvedor. Tais otimizações são adicionalmente guiadas por informações de perfil (como tipos de dados), coletadas em tempo de compilação (pelo *offline profiler*) ou execução (pelo *online profiler*), para especializar o código e consequentemente torná-lo mais eficiente. Adicionalmente, tais informações podem ser enviadas ao gerente de compilação, caso existir, para que este crie um plano de compilação que será enviado ao compilador otimizador para auxiliá-lo na otimização e geração de código.

Independente da forma de funcionamento, o sistema implementa um componente, chamado monitor, cuja finalidade é instrumentar cada unidade de código e medir a frequência de execução destas. Este monitoramento pode ser feito por meio de contadores ou tempo. Contadores são incrementados a cada invocação/término de execução da unidade de compilação [26]. E frações de tempo são verificadas durante a execução. Em outras palavras, a função do monitor é detectar regiões críticas, isto é, regiões de código cujo contador (ou tempo) excedeu um limite e emitir um alerta ao gerente de código para que o mesmo envie a região crítica recém detectada para (re)compilação. Contudo, existem outros sistemas que geram código com um compilador otimizador no momento em que as unidades são invocadas, nesse caso, o monitor não é utilizado.

Na medida em que as regiões críticas são detectadas, estas são encaminhados do gerente de código para o gerente de compilação e inseridas na fila de compilação. Então, o compilador otimizador compila a região crítica do início da fila e encaminha o código gerado para o gerente de código.

O gerente de código gerencia as versões de código para cada unidade de código existente no ambiente. Embora, o gerente de código possa não estar enquadrado como um componente particular de alguns sistema JIT, suas funcionalidades subsistem, sendo elas:

- Enviar a versão correta de código para ser executada pelo motor de execução. Em sistemas que interpretam e compilam, se uma unidade está disponível nas duas versões, a prioridade de execução é da versão compilada, pois esta é mais rápida. Por outro lado, em sistemas que compilam e recompilam, a prioridade é da versão mais recente. O último caso prevalece em sistemas que recompilam várias vezes, pois nestes, o código tende a ser mais eficiente a cada recompilação.

- Enviar código para o gerente de compilação, que por sua vez, solicita (re)compilação.

- Receber código provido pelo compilador base ou otimizador após uma unidade (ou região crítica) ser compilada e armazená-la no repositório de código.

O repositório de código consiste de estruturas que armazenam as versões do código: versão interpretada e versão nativa. A versão interpretada é padrão dos sistemas de modo misto e pode ser gerada estaticamente, como os *bytecodes* das JVMs, ou no tempo de carga do sistema, como o código YAAM [54] do YAP com compilação JIT [12]. A versão nativa, por sua vez, consiste de código de máquina gerado pelo compilador base ou ainda das regiões críticas que foram geradas em tempo de execução. Embora o repositório de código tenha tanto código intepretado quanto nativo, apenas os sistemas de modo misto executam ambas as versões.

O motor de execução é responsável por executar o código recebido do gerente de código, independente de sua versão. Este componente pode variar entre vários modos de execução para partes específicas do programa. Sistemas que interpretam unidades não frequentes e compilam as demais funcionam em um modo misto de execução. Analogamente, sistemas que (re)compilam código funcionam apenas no modo de execução de código nativo.

As próximas subseções fornecem mais detalhes sobre os possíveis componentes da arquitetura de um sistema JIT.

3.1 Compilador Base e Compilador Otimizador

O compilador base gera código nativo em tempo de execução para todas as unidades de código invocadas pelo sistema, imediatamente antes da primeira invocação. O objetivo deste compilador é ser um gerador rápido de código nativo. Desta forma, o compilador base não aplica às unidades compiladas um conjunto de otimizações.

Com uma abordagem diferente do compilador base, o compilador otimizador gera código nativo em tempo de execução para as regiões críticas. Por padrão, todo código compilado é otimizado com um conjunto de otimizações pré-definido, independente do programa em execução. Contudo, o sistema pode ajustar automaticamente o compilador para que este altere o nível de otimização que será aplicado em uma determinada região crítica. Isto possibilita que o sistema recompile unidades de código buscando melhorar o desempenho, como também proporciona que o sistema se auto ajuste as características do programa em execução.

3.2 Offline Profiler

O *offline profiler* é utilizado para derivar informações em tempo de compilação ou carregamento do programa, que possam guiar a geração de código especializado. Desta forma, o objetivo do uso de tais informações é melhorar o desempenho do código compilado, aumentando a velocidade de execução e/ou reduzindo o tamanho do código. A idéia geral é executar o programa sobre um domínio conservativo, a fim de satisfazer um conjunto de restrições. No término da execução, os resultados encontrados fornecem uma aproximação correta das informações sobre o programa analisado. Tais informações incluem tipos de dados (no caso de sistemas para linguagens com tipagem dinâmica), ou mesmo regiões críticas de código.

3.3 Online Profiler

O *online profiler* permite que o sistema JIT se adapte às características do programa em execução. Para alcançar tal objetivo, o *online profiler* monitora a execução dos programas e coleta informações que possibilitam a geração de código especializado. Isto é independente da forma que o sistema foi projetado, ou seja, é independente do uso de um modo interpretação mais compilação ou compilação mais recompilação. Nesse sentido, o *online profiler* pode coletar as seguintes informações:

Tipos de variáveis Sistemas JIT empregados em linguagens com tipos dinâmicos podem se beneficiar da coleta do tipo das variáveis. Neste caso específico, o sistema é tradicionalmente implementado de forma genérica para todos tipos de dados possíveis, o que reflete em instruções de checagem de tipos durante a execução. Portanto, o uso de tais informações permite que tais checagens sejam removidas.

Informações sobre as unidades de código Informações tais como: tamanho da unidade, existência de chamadas aninhadas, quantidade de parâmetros, tipos dos parâmetros e recursividade auxiliam na especialização do código que será compilado, o que tenderá a aumentar o desempenho do sistema.

Tempo de interpretação Essa informação é útil dependendo da métrica utilizada para determinar se uma unidade é crítica ou não e consequentemente acionar o sistema de

compilação. As métricas utilizadas pelos sistemas JIT serão descritas na Seção 4.2.

Tempo de compilação Assim como o tempo de interpretação, o tempo de compilação é outra informação necessária ao sistema, dependendo da métrica utilizada, porque permite definir um tempo ideal de interpretação para que as unidades de código se tornem regiões críticas. O uso de tal informação também será descrito na Seção 4.2.

É importante observar que as informações coletadas podem ser facilmente modificadas durante a execução, dependendo das características da linguagem de programação para a qual o sistema foi desenvolvido.

3.4 Monitor

O sistema JIT instrumenta as unidades de código do programa com parâmetros de frequência a fim de medir a frequência de execução de tais unidades. Esta instrumentação pode ser feita com o uso de contadores ou fração de tempo. Desta forma, é papel do monitor atualizar o parâmetro de frequência e a medida que tal parâmetro atinja um limite pré-definido este deve alertar ao gerente de código que tal unidade se tornou frequente.

3.5 Gerente de Código

O gerente de código gerencia as áreas de código do sistema. Suas tarefas consistem basicamente em:

1. Enviar a versão correta da unidade acionada para execução, priorizando o envio do código nativo mais atual (ou seja, mais especializado);
2. Enviar uma unidade para compilação.

A interação do gerente de código com os demais componentes, bem como o seu comportamento com uma unidade de código é definido pelo estado atual da unidade em questão. Em modo de execução interpretação mais compilação, cada unidade do programa pode assumir um dos seguintes estados: interpretado, frequente ou nativo. No modo compilação mais recompilação, os estados de uma unidade são: nativo, frequente ou recompilado.

Durante a execução do programa, o gerente de código interaje com o monitor para identificar quais unidades são frequentes. Desta forma, assim que uma determinada unidade é detectada, o gerente de código a envia para o gerente de compilação para que este providencia uma nova versão da unidade. Após o processo de especialização (seja ele compilar ou recompilar, dependendo da forma de implementação do sitema JIT) de uma unidade é papel do gerente fornecer ao motor de execução a nova versão da unidade.

Normalmente, o gerente de código verifica a existência de código especializado e o envia para execução, permanecendo nesse processo até não encontrar mais código especializado para executar ou até finalizar o programa. Adicionalmente, a execução de código especializado pode ser interrompida no caso de ocorrer alguma exceção. Nesse caso, o gerente de código precisa restaurar o estado do sistema até um ponto anterior ao qual ocorreu a exceção e enviar ao motor de execução uma versão não especializada da unidade.

3.6 Gerente de Compilação

O gerente de compilação é responsável por criar um plano de (re)compilação para o (re)compilador otimizador utilizar durante a geração de código. Basicamente, as informações coletadas pelo *offline* e/ou *online profilers* são inferidas para uma região crítica específica e enviadas ao gerente de compilação para que este crie o plano de compilação adequado. Em geral, este plano define um conjunto de otimizações pré-determinado pelo desenvolvedor do sistema, que é o mais adequado àquela região crítica [40]. Além disto, tal 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 [55, 39, 56].

3.7 Repositório de Código

O repositório de código consiste de estruturas que armazenam versões de código interpretado e/ou compilado. Em sistemas que interpretam e compilam, o código interpretado é o padrão na execução. Este pode ser criado a partir de uma representação de mais alto nível e então simplesmente carregado na inicialização do sistema ou criado durante o tempo de carregamento do programa. Por outro lado, sistemas que compilam e recompilam mantém somente a área de código nativo.

A área de código nativo se expande na medida em que regiões críticas são compiladas. Nesse sentido, sistemas de compilação JIT podem implementar políticas para garantir que tal região não cresça mais que um limite determinado, descartando código pouco invocado e recompilando-o quando necessário, como acontece em Smalltalk [3].

Além disso, sistemas que empregam *online* e/ou *offline profiler* podem construir versões de código especializadas para cada tipo de informação coletada. Portanto, nestes sistemas a prioridade de execução é sempre do código especializado para o comportamento ativo. Porém, é sempre importante manter uma versão genérica para garantir estabilidade caso o comportamento de uma região de código altere, por exemplo, a mudança de tipo de uma determinada variável. Outra consideração importante é a construção de um coletor de lixo [57] associado a sistemas desse tipo. Visto que várias versões de código nativo induzem a uma expansão contínua da área de código nativo, é importante conter essa expansão. Geral-

mente, isto consiste em eliminar código inutilizável (não executado por um longo período de tempo) ou inválido (especializado para uma informação que não é equivalente ao comportamento atual do programa).

3.8 Motor de Execução

O motor de execução é o componente que executa as unidades de código independente do seu estado. Em sistemas que empregam interpretação mais compilação, o motor de execução é capaz de interpretar código como também invocar código nativo. Isto indica que para esta forma de implementação do sistema, o motor de execução é um interpretador que possui um mecanismo de invocação de código nativo. Para sisstemas que empregam compilação mais recompilação, o motor de execução é um módulo composto apenas de um mecanimso de invocação de código nativo.

4 Princípios de Compilação JIT

Visto que em sistemas JIT os compiladores compilam e otimizam código em tempo de execução é necessário que estes sejam rápidos, efetivos, leves e capazes de gerar código nativo de alta qualidade. Entre essas necessidades, entretanto, existe um *tradeoff* entre o tempo de compilação e eficiência do código gerado, que forma a base desses sistemas. Em outras palavras, um compilador JIT deve ser sensível à eficiência de tempo e espaço dos seus algoritmos de otimização, pois uma compilação lenta pode desacelerar o tempo de resposta do programa. Além disso, é preciso saber o que compilar, pois devido ao fato da compilação JIT ocorrer em tempo de execução, nem todo trecho de código deve ser compilado. Na verdade, gerar código nativo para uma unidade específica, cuja execução não compense o *overhead* de compilação geralmente não trará resultados satisfatórios.

Além de ser importante conhecer quando compilar, o que compilar e como compilar, é necessário considerar de forma precisa o mecanismo utilizado na manutenção dos sistemas de compilação e execução, como também as questões referentes aos atrasos que podem ocorrer, pois se considerados de forma ingênua, podem ocasionar uma degradação no desempenho. Por essa razão, as seguintes questões devem ser analisadas:

1. O atraso decorrente da espera da unidade de código para que se torne uma região crítica; e
2. O atraso decorrente da espera da região crítica na fila de atendimento de compilação.

Formas de tratar estes atrasos são difíceis de projetar ou implementar, devido ao fato de uma configuração negligente nas formas de detecção de regiões críticas poder ocasionar

um alto *overhead* ao sistema como um todo. Pesquisadores têm tentado contornar o atraso decorrente da unidade à espera para se tornar região crítica, como é o caso dos trabalhos de Krintz [30], Namjoshi e Kulkarni [58] e Campanoni *et al* [59], porém com o custo de requerer execução prévia do sistema ou um processador adicional. Com respeito ao atraso decorrente da espera na fila de compilação, diversos trabalhos procuram utilizar prioridades nas *threads* [60] do compilador [44, 43, 42, 13].

4.1 Manutenção dos Sistemas de Compilação e Execução

Plezbert e Cytron [26] classificaram as possíveis abordagens para o gerenciamento entre a compilação e a execução em um sistema JIT da seguinte maneira:

JIT Neste tipo de sistema, a unidade de código é compilada imediatamente antes de sua execução e uma única *thread* é utilizada para alternar entre compilação e execução. Isso significa que o motor de execução é suspenso durante a atividade do compilador. Como essa abordagem traduz código sob-demanda, ela evita que o sistema compile inutilmente alguns trechos de código, devido ao fato de grandes programas possuirem porções de código que as vezes não são executados. Um sistema que segue essa abordagem é a JVM KAFFE [7].

Smart JIT Apesar da vantagem inerente aos sistemas que utilizam a abordagem anterior, esta pode acarretar perda de desempenho para uma situação na qual o sistema compile todas as unidades de código invocadas. Como a maior parte do tempo de execução da maioria dos programas está relacionado a execução de uma pequena faixa de código [61, 62], pode não ser viável compilar faixas se elas não forem executadas com frequência. Em outras palavras, uma compilação eficiente deve saber escolher o que compilar. *Smart* JIT emprega técnicas que visam detectar unidades de código frequentes e então, compilar somente estas. Em tais sistema também é utilizado somente uma *threads*, o que torna obrigatório suspender para compilação a execução de código. Sistemas *Smart* JIT são: SELF-93 [8, 9], JUDO [27], JIKES RVM [44], YAPC [12], LUAJIT [50] e HIPE [49].

Compilação Contínua A técnica de compilação contínua foi inicialmente apresentada como um modelo teórico e tem sido empregada em alguns sistemas atuais [44, 45, 13], principalmente após o surgimento (e expansão) de arquiteturas *multi-core*. A explicação para a implantação de compilação contínua após o surgimento dessas arquiteturas é que tal abordagem utiliza, mais de uma *thread*: uma para execução e as demais para compilação. Embora, ainda detecte regiões críticas, assim como *Smart JIT*, a vantagem intrínseca é que o ambiente não precisa suspender a execução para gerar código nativo. Entretanto, o custo de implementação é maior: as *threads* precisam ser gerenciadas, bem como o acesso aos dados compartilhados, como o gerente de código e a fila de

regiões críticas escalonadas para compilação (no caso de haver mais de uma *thread* de compilação).

4.2 Acionamento do Sistema de Compilação

Basicamente, um sistema de compilação JIT funciona da seguinte forma: em primeiro lugar, ele instrumenta cada unidade de código com informações que medem a frequência de sua execução. Em seguida, o programa é interpretado [51, 35, 48, 45, 13] ou compilado com um compilador rápido [44, 27, 9] e então executado. Durante a execução do programa, o sistema monitora as unidades de código com o objetivo de detectar quais unidades são frequentes, ou seja, quais regiões são críticas. Este monitoramento é realizado verificando a informação que mede a frequência de execução de cada unidade. Desta forma, uma unidade de código é então dita crítica se sua informação de frequência atinge o limite pré-definido pelo sistema.

Para detecção das regiões críticas, sistemas JIT atuais empregam contadores [51, 23, 9, 48, 50, 49, 12], amostras de tempo [44] ou uma combinação entre contadores e amostras de tempo [27]. O uso de contadores requer que o sistema conte o número de invocações de cada unidade, enquanto a amostragem de tempo precisa interromper a execução periodicamente para atualizar o tempo da unidade. De qualquer forma, independente da abordagem utilizada, a unidade é enviada para compilação quando a respectiva métrica alcançar um limite pré-definido.

A escolha de um limite correto é crucial para alcançar um bom desempenho de execução. Um limite muito baixo pode compilar código em demasia, o que aumenta o *overhead* de compilação e gera muito código nativo, que provavelmente não proporcionará ganhos de desempenho em execuções posteriores (se ocorrer). Por outro lado, um limite muito alto pode tornar o sistema muito conservativo que impede que regiões críticas ideais sejam compiladas.

Uma base teórica para ajustar a métrica utilizada pelo compilador foi dada por Karp [63]. Tal base afirma que o ideal é a unidade ser interpretada um número de vezes suficiente para compensar o *overhead* de compilação. O trabalho de Plezbert e Cytron [26] explora outras métricas, a saber: *crossover* e *balance*. *Crossover* utiliza o tempo de compilação da unidade, juntamente com um parâmetro chamado ponto de *crossover* para encontrar um limite ideal para a amostra de tempo. Enquanto isso, *balance* envia para compilação as unidades cujo tempo de interpretação ultrapassa seu tempo de compilação. Tais abordagens, entretanto, são difíceis de implementar porque na prática, encontrar o tempo de compilação das unidades requer uma estimativa (que não é precisa) ou uma passagem adicional pelo programa (que pode não ser desejável em alguns casos).

No entanto, alguns trabalhos recentes têm buscado formas de enviar as unidades de código o mais rápido possível para compilação. O trabalho de Krintz [30] explora um mecan-

ismo em tempo de compilação para detectar previamente as regiões críticas, enquanto que Namjoshi e Kulkarni [58] propuseram uma técnica de predição utilizando um processador adicional. Outra técnica de predição, implementada no trabalho de Campanoni *et al* [59], procura dar ênfase às unidades mais próximas da unidade em execução atual, julgando-as possuírem maior probabilidade de serem invocados futuramente. Além disso, as arquiteturas *multi-core* têm sido exploradas ultimamente para manter *threads* específicas somente para compilar ou executar código. Sistemas que utilizam esta abordagem são: SUN HOTSPOT [48, 45], JIKES RVM [44] e ILDJIT [46, 47].

4.3 Seleção de Unidades de Compilação

Um fundamento importante relacionado às questões abordadas por Hansen [23], coube a Hölzle no seu trabalho com SELF-93 [8, 9]. Hölze observou que o mecanismo de seleção das unidades que serão efetivamente compiladas (o que compilar) é mais importante que o mecanismo para disparo de compilação (quando compilar). O mecanismo de seleção de unidades é uma forma de "previsão" de frequência de execução das unidades de código. Pois, como não é possível identificar o número de vezes que uma unidade será executada no futuro, é considerado que sua frequência de execução a partir de um determinado ponto seja proporcional a sua frequência de execuções antes deste ponto. Desde os primeiros trabalhos relacionados a compilação JIT até os mais recentes, três estratégias de seleção de unidades viáveis para compilação foram propostas, são eles: (1) seleção baseada em sub-rotinas; (2) seleção baseada em regiões; e (3) seleção baseada em caminhos de execução.

4.3.1 Seleção Baseada em Sub-rotinas Compiladores JIT possuem, em muitos casos, uma estrutura similar à dos compiladores estáticos e como os tais, herdam os mesmos fundamentos. Basicamente, um compilador estático constrói, para cada sub-rotina, um grafo de fluxo de controle e aplica uma série de otimizações sobre ele. A sub-rotina completa, bem como todas as demais alcançadas a partir desta são compiladas e otimizadas sem qualquer preocupação quanto ao tempo, pois neste caso, esse é um parâmetro irrelevante. Como mencionado, compiladores JIT funcionam de forma similar, mas com a única diferença de que compilam as sub-rotinas sob demanda, ou na primeira invocação ou apenas aquelas que são consideradas importantes para o desempenho, sejam elas funções [50, 49], métodos [27, 3, 7] ou cláusulas [12].

No entanto, regiões críticas possuem trechos de código que são pouco ou nunca executados [62, 64], o que pode causar efeitos adversos e reduzir a efetividade das otimizações [65]. Um exemplo típico é a aplicação de *inline* que é muitas vezes restringinda a sub-rotinas cujo tamanho do código ultrapasse um limite pré-estabelecido, para evitar possíveis explosões de código [66]. Consequentemente, uma sub-rotina já integrada, porém com muito código infrequente, pode impedir que sub-rotinas frequentes sejam também integradas à sub-rotina

principal na medida que *inline* se expande.

A escolha de sub-rotinas como a unidade básica de compilação em sistema JIT é, de fato, uma escolha conveniente (fácil de manter), mas se for assumido o fato de que a maioria das sub-rotinas são compostas por porções de código não frequentes, o ideal seria detectar e eliminar estas porções, a fim de reduzir o esforço das otimizações e gerar um código mais eficiente.

4.3.2 Seleção Baseada em Regiões No contexto de compilação JIT, esta técnica procura por porções de código pouco frequentes dentro de uma unidade e as eliminam do mecanismo de seleção. Portanto, as regiões críticas são os trechos de código frequentes dentro das unidades e não necessariamente uma sub-rotina.

Essa estratégia foi inicialmente baseada em informações coletadas estaticamente e explorada por um *framework* de compilação para um ambiente estático [67]. Basicamente, estatégias para formação de regiões são dependentes da classe do sistema que a implementa. Enquanto sistemas estáticos utilizam informações coletadas estaticamente para guiar a formação de regiões e preferem escolher porções de código (geralmente blocos básicos) de maior frequência para compor tais regiões, sistemas de otimização binária, como DYNAMORIO [68] e MOJO [69], se baseiam em informações coletadas dinamicamente, tomando o fluxo de execução corrente para reotimizá-lo em tempo de execução.

Por outro lado, compiladores JIT não são beneficiados no todo por informações coletadas estaticamente. Além disso, utilizar apenas contadores ou amostras de tempo pode ocasionar a exclusão de blocos críticos não dominantes na formação de uma região. Um exemplo é um bloco `if-then-else`, no qual, tanto `then` quanto `else` são executados frequentemente. Neste caso, informações coletadas dinamicamente auxiliariam na escolha de apenas um destes blocos (aquele com uma frequência levemente superior) para a formação da região de compilação. Se isso não ocorrer, há grandes chances de que o bloco ignorado leve a degradação o desempenho do sistema [65].

O trabalho de Suganuma *et al* [31] foi o primeiro a abordar a técnica para detecção de regiões para compiladores JIT, em sistemas Java. A idéia básica é inicialmente assumir que cada método seja representado como um grafo de fluxo de controle com apenas uma entrada e uma saída. Em seguida, os blocos básicos (que consistem nos nós do grafo) são marcados como frequentes ou não frequentes com base em algumas heurísticas. Normalmente, blocos que contém classes não referenciadas, que terminam com uma instrução de exceção ou que manipulam exceções são considerados não frequentes e blocos que finalizam com instruções de retorno normais são marcados como frequentes. Adicionalmente, informações coletadas em tempo de execução podem ajudar a detectar blocos nunca executados, marcando-os também como não frequentes. Finalmente, uma passagem pelo grafo de fluxo analisa cada bloco básico, marcando-o como não frequente na sua saída somente se a entrada de todos os seus

sucessores definirem este como frequente. Desta forma, é definido como não frequente todo bloco básico definido como não frequente na sua entrada e sua saída. Tais blocos, enfim, podem ser desconsiderados dentro do método e não terão código final gerado.

4.3.3 Seleção Baseada em Caminhos de Execução Sistemas baseados em seleção por caminhos de execução [51, 53, 35] procuram definir como regiões críticas os caminhos de execução frequentes que podem ultrapassar os limites de uma sub-rotina. O primeiro trabalho a explorar os conceitos desta técnica foi um sistema para tradução binária, chamado DYNAMO [36]. Este sistema utiliza informações coletadas em tempo de execução para detectar e otimizar caminhos críticos de execução. Contudo, foi somente com o HOTPATHVM [35] que essa técnica foi utilizada em compilação JIT.

Basicamente, a detecção de caminhos críticos de execução funciona da seguinte maneira: um contador é mantido no início de cada *backward branch* (desvios cujo endereço alvo é menor que o endereço da instrução de desvio), que é incrementado toda vez que o desvio é executado. Quando o valor de um contador de destino excede um limite, o caminho de execução é gravado, compilado, armazenado em uma *cache* específica e invocado pelo interpretador ou pelo ponto de saída de outro caminho de execução compilado.

Dessa forma, as partes constituintes de um caminho de execução é o início dele (a cabeça), que é uma instrução ou unidade executada frequentemente e um corpo que é estendido ao longo do código, a partir da cabeça até um ponto onde uma das seguintes condições ocorrem:

- Um ciclo de instruções é detectado;
- O caminho gravado excede um tamanho pré-definido;
- Uma instrução incomum, como um manipulador de exceções, é encontrada; ou
- Uma instrução que forma a cabeça de um caminho de execução é encontrada.

Desde HOTPATHVM, muitos compiladores JIT baseados em seleção de caminhos foram implementados, dentre eles, TRACEMONKEY [53], LUAJIT [50], TAMARIN-TRACE [51] e HAPPYJIT [52].

Provavelmente a próxima tendência na construção de compiladores JIT utilizará seleção por caminhos de execução para detectar regiões críticas mais eficientes. Hayashizaki *et al* [38] demonstraram que a remoção de falsos laços dos caminhos detectados é uma boa estratégia para melhorar o desempenho, embora outro trabalho atribuído a Inoue *et al* [70] tenha demonstrado que detectar grandes caminhos de execução e escaloná-los para compilação é também uma boa estratégia. Adicionalmente, a vantagem em manter grandes caminhos de

execução é que, quanto maior for, maior as oportunidades de otimização [37], que convergem em código mais eficiente.

4.4 Geração de Versões Especializadas

Com a ocorrência de novas pesquisas, novas técnicas que auxiliam compilação JIT foram criadas e aperfeiçoadas, como *drag-along* [19], comumente conhecida como avaliação tardia e código misto [21] e ainda empregada em sistemas que interpretam e compilam. No contexto de linguagens orientadas a objetos, SELF [4, 5, 6, 8, 9] inovou no desenvolvimento de otimizações mais agressivas, como a customização de código que é utilizada para criar métodos especializados por tipo de dado e *type feedback*, que extrai informações de tipos de execuções prévias para auxiliar no processo de compilação.

O processo de otimização visa transformar unidades de código para torná-los mais eficientes em termos de tempo, espaço e/ou consumo de energia. Diversas técnicas que integram esse processo são especializadas para um conjunto de tarefas, como simplificar expressões constantes [57], eliminar expressões comuns [71] ou simplificar laços dos programas [72]. A saída da unidade otimizada deve ser mantida e a única diferença notada pelo usuário final é o benefício gerado por tais técnicas, como a execução rápida de código e reduções do espaço ocupado e energia consumida. No que se refere à compilação JIT, é esperado que o processo de otimização consuma pouco tempo de compilação.

A aplicação de técnicas de otimização [73], em compiladores JIT em particular, gera um *tradeoff* entre o tempo de compilação e a eficiência do código final. Em compiladores estáticos, o *overhead* inerente pode ser ignorado, portanto tais técnicas são aplicadas em tempo de compilação. Como resultado, otimizações mais agressivas podem ser aplicadas arbitrariamente. Por outro lado, ambientes de compilação JIT precisam ser criteriosos, uma vez que o tempo de compilação (e otimização) integra o tempo total de execução. Por esta razão, uma decisão negligente pode resultar em uma degradação no desempenho do sistema como um todo. Nesse sentido diversos sistemas têm adotado diferentes abordagens.

`Adaptive Fortran` [23] escolhe e aplica técnicas diferentes a cada invocação de uma unidade código. Por outro lado, os compiladores de SELF-93 [8, 9], JUDO [27], SUN HOTSPOT [48, 45] apostam em um conjunto de otimizações pré-definidas e pré-ordenadas aplicadas uma única vez por unidade de código. Embora, apenas SELF e SUN HOTSPOT CLIENT evitem otimizações mais agressivas.

Por outro lado, JIKES RVM [44, 11] implementa um sistema de otimização adaptativa. Geralmente, o sistema utiliza um cálculo de custo-benefício para julgar o nível de otimização que uma unidade prestes a ser compilada receberá. Outro sistema que implementa otimizações em nível é a MÁQUINA VIRTUAL DA IBM [13]. Contudo, as otimizações aplicadas em cada nível são técnicas atribuídas manualmente pelos desenvolvedores e consomem um

tempo dispendioso para avaliação. Nesse sentido, um trabalho recente [39] tem buscado aplicar busca evolucionária para encontrar planos de otimização que regulam o *tradeoff* existente entre o tempo de compilação e a eficiência do código.

Em geral, as otimizações aplicadas pelo sistema de compilação JIT utilizam análises de fluxo de dados que precisam encontrar os usos de cada definição de variável ou a definição de cada uso em cada expressão. O canal *def-uso* é uma estrutura de dados que torna este acesso mais eficiente: para cada declaração no grafo de fluxo, o compilador mantém uma lista de ponteiros para todos os usos da variável definida nesta declaração, e para cada instrução uma lista de ponteiros para todas as definições das variáveis usadas nela. Desta maneira, o compilador pode rapidamente saltar da definição para o uso ou vice-versa.

Uma melhoria nesta idéia é o uso da representação *Static Single-Assignment* (SSA) [74, 75], uma representação intermediária na qual cada variável possui apenas uma definição no programa. A única (estática) definição pode estar em um laço que é executado diversas (dinâmicas) vezes, logo o nome representação *static single-assignment* ao invés de representação *single assignment* (onde variáveis nunca são redefinidas). A representação SSA é popular em sistemas JIT por diversas razões:

1. Análises de fluxo de dados e algoritmos de otimização podem ser implementados de maneira simples quando cada variável possui apenas uma definição.

2. Se a variável possui `N` usos e `M` definições, o espaço necessário para representar as estruturas *def-uso* é proporcional a $N \times M$. Para a maioria dos programas reais, o tamanho da representação SSA é linear no tamanho do programa original.

3. Usos e definições das variáveis na representação SSA simplificam algoritmos como construção de um grafo de interferência, que é utilizado por alocadores de registradores baseados em coloração de grafo, como o empregado pelo sistema SUN HOTSPOT SERVER.

4. Usos não relacionados da mesma variável em um programa tornam-se diferentes variáveis na representação SSA, eliminando relacionamentos desnecessários.

A escolha por esta representação está baseada nas razões descritas acima. Contudo, entender o comportamento do programa é crucial para obter um bom desempenho. Embora o uso de uma representação SSA torne os algoritmos de otimizações mais leves e simples, ela possui o problema de aumentar o tamanho do código gerado e a pressão por registradores. O tamanho do código aumenta devido ao uso de uma única definição para cada variável, acarretando o aumento dos acessos à memória. Consequentemente, o aumento da quantidade de variáveis aumenta a pressão por registradores. Em uma arquitetura com poucos registradores

isto pode acarretar uma quantidade expressiva de *spills*, isto é, uma expressiva representação de variáveis na memória, ocorrida pela falta de registradores.

Além do uso de uma representação SSA, sistemais modernos utilizam diferentes *profilers* para também auxiliar o processo de geração de versões especializadas. *Profilers* [30, 58] são projetados para coletar informações estruturais e comportamentais do programa, como tipos de dados, modos de execução e *backward branches*, estaticamente e/ou dinamicamente. *Type feedback profiler* foi implementado pela primeira vez no compilador SELF-93 [8, 9] de modo a extrair informações de tipo de execuções prévias e retorná-las ao compilador. Este retorno poderia acontecer dinamicamente ou estaticamente. Para tal, o sistema instrumentava o código para gravar os tipos coletados e propagá-los ao sistema de compilação na medida em que a execução prosseguia. Dessa forma, baseado nas informações coletadas, o compilador poderia prever prováveis classes receptoras (no contexto de `SELF`), ou tipos de dados (em um contexto geral) e gerar código mais eficiente. Em uma situação onde *type feedback* indicar que uma determinada variável sempre assumirá um valor inteiro, o compilador poderá gerar código especializado para esta situação.

Além das questões já abordadas, um fato que deve ser observado no desenvolvimento de sistemas de compilação JIT é o fato de algumas otimizações altamente efetivas serem complicadas por causa da flexibilidade de determinadas linguagens de programação. No contexto de `Java`, por exemplo, métodos virtuais limitam a aplicação de *inline*. Aplicar esta técnica é difícil para métodos virtuais porque não se sabe estaticamente que método será invocado. Desta forma, *inlining* de métodos virtuais pode acarretar, em alguns casos, a necessidade do método compilado ser invalidado quando uma nova classe for carregada. Nestes casos, o método é compilado novamente sem esta otimização.

Remover o código compilado resulta em um retorno ao interpretador. Uma solução para este caso é substituir o código compilado pelo interpretado. Esta situação mostra uma vantagem de ter simultaneamente código compilado e interpretado. A transição entre código compilado e interpretado é chamada de desotimização. Para tal, o compilador deve gerar uma estrutura de dados que permita a reconstrução do estado do interpretador até o ponto de chamada do código compilado.

Desotimização é fundamental para que o compilador possa realizar otimizações agressivas que aceleram a execução normal e seja capaz de gerenciar situações onde uma determinada otimização deva ser desfeita. Existem casos críticos onde o método compilado é desotimizado, por exemplo, quando ocorrem exceções. Com desotimização, o código compilado não precisa gerenciar tais situações, que em geral são casos incomuns. Sistemas que utilizam este recurso são SELF [4, 5, 6, 8, 9], SUN HOTSPOT CLIENT [48] e SUN HOTSPOT SERVER [45].

4.5 Implementação e Desempenho

Uma questão importante no tocante a implementação de um sistema JIT é a escolha adequada dos príncipios que nortearão tal implementação. As escolhas de projeto devem balancear o *tradeoff* existente entre tempo de compilação e a eficiência do código gerado. De fato, implementação e desempenho são duas questões que estão relacionadas. Portanto, o projeto de um sistema deste porte deve considerar as seguintes questões:

Modo de implementação A escolha entre um sistema misto ou um com compilação e recompilação possui vantagens e desvantagens. Sistemas que utilizam um interpretador proveêm um ambiente portável, seguro e de fácil implementação. Contudo, tais sistemas incorrem no *overhead* de interpretação. Por outro lado, embora tal *overhead* não ocorra em sistemas que utilizam apenas compilação, tais sistemas são mais complexos para manter. Além disto, a tendência é que sistemas que utilizam apenas compilação tenham um desempenho superior aqueles que ainda interpretam código (isto em relação a sistemas que utilizem as mesmas abordagens para implementar/solucionar as demais questões). Pelo fato, do tempo gasto em interpretação ser geralmente uma ordem de grandeza maior que a execução de código compilado.

Sistemas de Compilação e Execução A abordagem ideal utilizada no mecanismo de compilação e execução deve aproveitar ao máximo a capacidade das arquiteturas de hardware atuais. Com a limitação no aumento gradual do *clock* do processador [76], as arquiteturas modernas tem investido em máquinas *multi-core*. Tais máquinas proveêm diversos núcleos de processamento, que são utilizados para executar diversas tarefas simultaneamente. Desta forma, o ideal é explorar tais arquiteturas utilizando um sistema de compilação contínua. De fato, a abordagem JIT incorre em pelo menos dois problemas. Primeiro, tal abordagem não utiliza a detecção de regiões críticas e consequentemente pode incorrer em *overhead* de compilação de unidades que não amortizaram o tempo gasto pelo compilador. Segundo, não utiliza informações coletadas em tempo de execução. Como cada unidade é compilada imediatamente antes de sua invocação, não existe no sistema informações estruturais e comportamentais da unidade já que esta não foi ainda executada. Porém, é possível que tal abordagem seja beneficiada pelo uso de um *offline profiler*. A abordagem *smart* JIT possui o atrativo de minimizar o *overhead* de compilação, gerando código especializado apenas para regiões críticas. Contudo, incorre no problema de parar o motor de execução para que o sistema de compilação seja acionado. A compilação contínua além de possuir o atrativo de minimizar os problemas inerentes as outras duas abordagens, possui o pontencial de explorar as arquiteturas modernas. Isto decorrente do uso de pelo menos duas *threads*, uma para o motor de execução e outra para o sistema de compilação.

Acionamento do Sistema de Compilação O uso de contadores para acionar o sistema de

compilação possui o atrativo de ser uma abordagem simples de implementar, em comparação com a abordagem de utilizar amostragem de tempo. *Balance* necessita da implementação de um mecanismo de monitoramento de tempo de execução e *crossover* necessita que o programa seja compilado previamente. Abordagens que utilizam amostragem de tempo não são adequadas para sistemas interativos. Mas possuem potencial para sistemas servidores, nos quais não existe a interação direta com o usuário e geralmente as aplicações possuem um longo tempo de execução. É importante ressaltar, que embora com o uso de *crossover* exista a necessidade de compilar previamente o programa, a estimativa correta do limite máximo do contador requer informações impíricas, situação que também requer execuções prévias da aplicação. De fato, uma boa estimativa irá requerer diversos experimentos, por outro lado *crossover* requer apenas uma única execução. Nesta questão, o trabalho de Suganuma *et al* [77] demonstrou que o uso de contadores pode ser mais eficiente do que o uso de amonstras de tempo. Portanto, embora a estimativa do limite do contador seja uma tarefa dispendiosa esta pode proporcionar um bom desempenho ao sistema.

Seleção de Unidades de Compilação A escolha de sub-rotinas como unidades de compilação possui o atrativo de ser uma abordagem natural. Contudo, o fluxo de execução de um determinado programa pode se concentrar apenas em uma porção de código contida em uma sub-rotina, com ainda ultrapassa tal fronteira. O uso de regiões possui o atrativo de eliminar porções de código que não são executados frequentemente. E o uso de caminhos de execução além de possuir este atrativo, possui o potencial de integrar em uma única porção compilada diversas unidades de código. Esta integração possui a tendência de minimizar o custo do chaveamento entre diversas versões de código. Como também, de aumentar o escopo de aplicação de otimizações de código. Consequentemente, aumentando o desempenho do sistema. Por outro lado, caminhos de execução requer uma esforço maior de implementação do que as outras duas abordagens. Isto pode ser evidenciado pelo fato de tal estratégia requerer algoritmos mais elaborados para sua implementação, como também para que os caminhos detectados não degradem o desempenho do sistema. Tal degradação pode ocorrer pelo fato do caminho de execução ser tomado a partir de uma unidade não frequente, o que potencialmente não amortizará o *overhead* de compilação, como também o chaveamento entre versões diferentes de unidades.

Geração de Versões Especializadas Esta tarefa é uma necessidade em sistemas JIT, pelo fato de tal sistemas terem por objetivo aumentar o desempenho de ambientes projetados para executar programas. Independente do modo de implementação do sistema, o projetista deve prover um mecanismo que seja capaz de especializar o código que está sendo gerado durante a execução do programa, em outras palavras o projetista deve implementar um mecanismo que aplique otimizações ao código compilado. Uma abordagem conservativa embora seja simples do ponto de vista da implementação, possui a

tendência de não prover um código com uma alta eficiência, embora que gerar código de alta qualidade seja um problema complexo de ser resolvido. O ideal é que o sistema se adapte as características do programa em execução e desta forma escolha dinamicamente quais otimizações aplicar. Ainda nesta questão, é papel do desenvolvedor do sistema escolher quais algoritmos utilizar para implementar cada otimização. Isto pelo fato de existirem algoritmos que são mais eficientes em determinadas situações, pois a eficiência de um algoritmo de otimização está relacionada ao contexto de sua aplicação como também a representação interna utilizada para o código do programa.

5 Sistemas

Mesmo que os primeiros sistemas de compilação JIT tenham surgido a partir de 1960, com `LISP`, somente uma década após, com o trabalho de Hansen, é que os desafios a serem enfrentados por tais sistemas foram formalizados [23]. Desde então, sistemas de compilação JIT tem incorporado diversos princípios que diversificaram decisões de projeto, arquitetura e técnicas. Tais decisões são destacadas na apresentação dos sistemas a seguir.

Adaptive Fortran foi projetado para suportar execução de código em modo misto [23]. Todos as unidades de código do programa são instrumentadas com contadores, que são atualizados a cada chamada da unidade. Dessa forma, uma unidade se torna um candidato para a "próxima otimização" caso seu contador ultrapasse o limite de frequência pré-estabelecido. Durante esse passo, um código supervisor é invocado entre as unidades de código, que acessa os contadores e aplica otimizações (se necessário). Por fim, o sistema transfere o controle de execução para a próxima unidade, dando preferência na execução da versão nativa dessa unidade, se existir. As unidades de código escalonadas para compilação são otimizados por uma técnica de otimização por vez e recompilados com uma técnica de otimização diferente caso se tornem regiões críticas novamente. Esta estratégia mostrou ser uma esquema de (re)compilação primitivo, que foi mais tarde aprimorado por outros sistemas [44, 27, 9, 13].

Sistema de Smalltalk-80 traduz código da MÁQUINA VIRTUAL SMALLTALK [78] para código nativo no momento em que um método é invocado [3]. Após esse processo, o código gerado é armazenado em uma *cache* para execuções posteriores. O sistema projetado é vinculado a um gerenciamento de memória, que impede a paginação de código nativo caso a *cache* ultrapasse um limite de armazenamento. Se por algum momento esse limite for alcançado, o código nativo é simplesmente descartado e novamente gerado, caso necessário.

SELF possui um ambiente de execução que implementa o mecanismo de compilação mais recompilação [8, 9]. Basicamente, um compilador rápido não otimizador compila as

unidades de código que estão prestes a serem executadas e, utilizando contadores, o sistema identifica os métodos críticos e compila-os com um compilador otimizador. Este último emprega compilação baseada em região, porém diferente daquela apresentado por Suganuma et al. [65]. Para tal, um sistema de predição de tipos é utilizado para detectar e remover código das mensagens enviadas às classes pouco comuns, o que significa que apenas código de mensagens enviadas às classes frequentes são compilados.

Kaffe pode executar código tanto em forma interpretada quanto compilada [7]. No modo de compilação, KAFFE compila todos métodos no momento de sua invocação e não executa código misto. Isso significa que o sistema se enquadra no modelo JIT, segundo a visão de Plezbert e Cytron [26]. O compilador JIT incorporado converte os *bytecodes* de entrada em uma representação intermediária chamada *KaffeIR*, que é mapeada para código nativo do hardware utilizado, por meio de um conjunto de macros pré-definidas.

Jikes RVM emprega a estratégia de compilação mais recompilação [44, 11]. Ele compila todos os métodos na medida em que são invocados por um compilador base não otimizador e recompila as regiões críticas com um compilador otimizador [11], que emprega várias otimizações separadas em tês níveis. O primeiro consiste de um conjunto de otimizações aplicadas durante a tradução dos *bytecodes* para uma representação intermediária utilizada pelo compilador. O segundo nível aplica todas as otimizações do primeiro nível, em adição com *inline* e outras otimizações locais. Por fim, o nível mais agressivo emprega todas as otimizações do nível anterior, em adição com otimizações baseadas na representação SSA para variáveis escalares [79]. Basicamente, JIKES RVM detecta regiões críticas empregando amostras de tempo. As regiões críticas são, portanto, encaminhadas para uma fila e julgadas propícias para (re)compilação (em níveis maiores de otimização) ou não, baseado em uma análise de custo-benefício realizada pelo sistema. Dessa forma, se parecer rentável, a região crítica é (re)otimizada de acordo com o nível de otimização avaliado por uma *thread* específica, de modo a evitar pausas na execução e se beneficiar de paralelismo.

JUDO (JAVA UNDER DYNAMIC OPTIMIZATION) compila todo o código por um gerador rápido de código e instrumenta-o de modo que o sistema possa coletar informações em tempo de execução [27]. Por fim, as regiões críticas detectadas são escalonadas para compilação por um compilador otimizador, que aplica otimizações agressivas. Tais regiões são detectadas de duas formas possíveis: (1) utilizando contadores, ou (2) utilizando tempo. Na primeira abordagem, cada método é instrumentado com um contador fixado com um valor limite. No momento em que o método é chamado, seu contador é decrementado e, chegando a zero, o método é imediamente enviado para compilação. Já, na segunda abordagem uma *thread* separada é invocada em determinados intervalos de tempo para verificar o valor-limiar (medido em segundos) de

todos os métodos (seu tempo de execução) e enviar para compilação aqueles que se tornaram regiões críticas. Com o uso de contadores, o método é enviado para compilação imediatamente após o contador vinculado atingir o valor zero, contadores que são decrementados antes da execução do método. Por outro lado, com o uso de tempo, se o sistema identificar que um método atingiu um limiar de execução, este só será enviado para compilação na sua próxima chamada.

HiPE é um compilador JIT para a linguagem `Erlang` baseado em compilação de sub-rotinas [49]. Este compilador ainda inclui extensões para que o sistema suporte execução em modo misto de código. Todas as sub-rotinas são interpretadas pela máquina virtual no início da execução e somente aquelas frequentemente executadas são compiladas. Medidas utilizadas para verificar a frequência de chamada de sub-rotinas incluem contadores e tempo.

Máquina Virtual da IBM gera código nativo somente para os métodos críticos e interpreta os demais [13]. A contagem de frequência se baseia no uso de contadores, que são incrementados a cada chamada ao método. Unidades de código que representam laços frequentes também são considerados para compilação. Tal como em JIKES RVM, o sistema implementa compilação em multi-nível, cuja configuração foi baseada em medidas empíricas de custo *versus* benefício obtidas do trabalho de Ishizaki *et al* [40]. Três níveis de otimização são aplicados: o primeiro nível aplica um conjunto muito reduzido de otimizações, próprio para minimizar o *overhead* de compilação, enquanto os demais aplicam todas as otimizações disponíveis no compilador repetidas vezes, porém em quantidade limitada no nível intermediário. Na medida em que um método é compilado, este é monitorado periodicamente pelo sistema. Dependendo do valor do contador encontrado (a frequência), o método é promovido com otimizações de nível 2 ou 3. No nível 1, a compilação é regida pela própria *thread* de aplicação (configuração *Smart JIT*), porém os níveis 2 e 3, empregam *threads* separadas para compilação em segundo plano (compilação contínua).

HotPathVM é um compilador JIT baseado em caminhos de execução projetado para a máquina virtual JamVM [35, 80], própria para dispositivos embarcados. O sistema primeiramente interpreta os *bytecodes* Java construindo os caminhos para os trechos de código executados em maior frequência e, em seguida, gera código nativo para estes. Infelizmente, informações relacionadas a forma de detecção de regiões críticas não são fornecidas. Segundo os autores, HOTPATHVM utiliza apenas 150 KB de memória, incluindo código e dados, e consegue gerar código compatível àqueles gerados por outros compiladores JIT de grande porte.

YAP com compilação JIT foi projetado para melhorar o desempenho de linguagens lógicas [12]. Inicialmente todas as cláusulas são instrumentadas com contadores e interpretadas. Cada invocação, contudo, incremeta o contador em uma unidade. Dessa

forma, uma cláusula se torna uma região crítica caso seu contador atinja um limite pré-estabelecido, embora somente as cláusulas com um tamanho mínimo sejam escalonadas para compilação. Esta última restrição é para evitar que o compilador gaste tempo compilando uma unidade que possivelmente não trará benefício na execução. O sistema em si aplica poucas otimizações de código se comparado a outros sistemas JIT existentes, mas consegue acelerar de forma significativa a execução de Prolog frente a sua contraparte puramente interpretada.

Sun HotSpot inicialmente interpreta todos os métodos e vincula a estes contadores que medem a frequência de execução. Portanto, cada método é escalonado para compilação quando o seu contador atinge um limite. Adicionalmente, métodos com alto tempo de execução também podem ser compilados, mesmo se invocados poucas vezes. O ambiente de execução compartilha dois compiladores: o compilador cliente [48] e o compilador servidor [45], que podem ser escolhidos manualmente a cada execução. O compilador SUN HOTSPOT CLIENT [48] é indicado para programas interativos, cujo tempo de resposta é mais importante que o desempenho, sendo então projetado para alcançar um *tradeoff* entre a velocidade de execução e o desempenho do código final gerado. A compilação é realizada em diversas etapas, que transformam o código gradualmente até gerar o código final da plataforma alvo. O compilador SUN HOTSPOT SERVER [45] é indicado para programas de longa duração, cujo tempo de execução de código nativo compense o tempo de compilação gasto. Este compilador, se comparado ao cliente, é mais agressivo nas otimizações e algumas delas são guiadas por análise global. Além disso, o compilador servidor é capaz de detectar métodos virtuais pouco comuns e referências para classes não inicializadas para evitar gerar código para essas partes. Na visão de Plezbert e Cytron [26], SUN HOTSPOT SERVER é classificado como um sistema de compilação contínua, pois pode se beneficiar de compilação em segundo plano, caso o hardware possua múltiplos processadores.

Tamarin-Trace é um compilador para `JavaScript` baseado em compilação por seleção de caminhos de execução [51]. Cada *backward branch* do programa possui, vinculado ao seu destino, um contador que mede a frequência de execução. A partir disso, considerando que a maior parte do destino do desvio é um *loop header*, o interpretador não somente interpreta o código, como também constrói o caminho de execução a partir dele. Quando a gravação do caminho alcança novamente o *loop header*, o processo é finalizado e o caminho é transformado para uma representação intermediária de baixo nível que, por sua vez, é traduzido para código de máquina. Como no momento da gravação do caminho, os tipos das variáveis são conhecidos, o compilador pode realizar especialização de código para cada tipo detectado. Adicionalmente, decisões em tempo de execução, como desvios condicionais, são marcados para que a execução retorne ao modo interpretado caso um fluxo de controle ainda não compilado seja encontrado. Se isso ocorrer, o novo caminho é gravado e acoplado ao caminho

anterior, formando assim, uma árvore de caminhos de execução, que é então tratada com qualquer caminho gravado.

TraceMonkey é um compilador JIT para `JavaScript` baseada em compilação por caminhos de execução, que segue o modelo de interpretação e compilação [53]. Os pontos críticos do programa são detectados a partir de contadores e consistem de arestas de retorno dos laços. Quando um destes pontos é detectado, o interpretador entra em um modo especial para gravar o caminho de execução a partir dele, enquanto ainda interpreta código. A gravação do caminho é finalizado quando o ponto inicial da gravação é encontrado. Após a geração do código nativo, o sistema mantém a execução do código gerado até que a avaliação de tipos retorne um tipo diferente, ou até que um caminho não-compilado seja tomado. Em qualquer destes casos, o sistema retorna ao modo interpretado e reinicia a construção do caminho de execução.

LuaJIT implementa compilação baseada em caminhos de execução [50]. Primeiramente, o sistema interpreta código e mantém estatísticas de execução dos laços e chamadas de funções, que por sua vez, é medida com contadores. Quando um limite é alcançado, a unidade de código (função ou laço) é considerada crítica e, a partir dela, o caminho de execução é construído. A interpretação permanece até que o caminho seja definitivamente construído. As condições para o término de construção do caminho é um pouco diferente do apresentado no decorrer do texto. Basicamente, LuaJIT finaliza um caminho de execução após: (1) uma instrução não-tratável, como uma exceção, for encontrada ou quando (2) o início do caminho em construção for encontrado. No primeiro caso, o caminho é descartado e no segundo, é enviado para compilação. Após a compilação, o sistema prioriza a execução da versão nativa, como em todo o sistema baseado em código misto. Durante este passo, decisões tomadas em tempo de execução, como desvios condicionais e checagem de tipos, podem causar a saída do caminho de execução. Isso pode ocorrer caso um caminho diferente do fluxo de controle (ainda não compilado) for tomado ou quando uma checagem de tipos falha. Em qualquer destes casos, o sistema retorna a execução ao modo interpretado e reinicia o processo de contrução do caminho.

ILDJIT é um sistema de compilação JIT que utiliza a política de interpretação mais compilação [46, 47]. A idéia desse sistema é se beneficiar do paralelismo oferecido pelas máquinas *multi-core* atuais. A região crítica básica de ILDJIT é flexível, embora um método seja a unidade de compilação mínima. A detecção de tais regiões é garantida pela política *ahead-of-time*, utilizando a técnica *look ahead compilation* [59]. Com essa técnica, o sistema verifica a constituição do programa a partir do método atualmente em execução e seleciona os métodos mais próximos a este, enviando-os para compilação mais cedo. Isso é baseado em uma heurística, que espera que tais métodos possuam maiores chances de serem invocados mais recentemente que outros. O sistema ainda

explora automaticamente as configurações da máquina adjacente e dispara *threads* de compilação conforme a máquina oferece. O paralelismo entre execução e compilação é garantido pelo próprio sistema.

HappyJIT é um compilador JIT [52] para `PHP` desenvolvido a partir de PyPy [81], um *framework* para construção de máquinas virtuais que é composto por dois componentes: (1) um interpretador `Python` [82] escrito em um subconjunto restrito da linguagem, chamada `RPython` [83] e (2) um tradutor, que gera código nativo para o primeiro componente. O objetivo dos autores com essa estratégia é facilitar a implementação de linguagens dinâmicas, de modo que o projetista se preocupe apenas em escrever o interpretador para sua linguagem em `RPython`. Os detalhes de baixo nível e a geração de código fica a cargo do próprio *framework*. A fim de facilitar o esforço de implementação, HappyJIT reutiliza o *parser* do motor Zend [84], que converte código `PHP` internamente para um *bytecode* linear. Esse *bytecode* é recuperado e traduzido para a representação suportada pelo interpretador construído. Dessa forma, somente as estruturas de dados necessárias para representar os tipos de dados de `PHP` em memória e este interpretador foram implementados. Por fim, o interpretador construído é passado como entrada para PyPy. Isso significa que, durante a execução há dois intepretadores envolvidos: o projetado para interpretar o programa `PHP` do usuário e o utilizado para interpretar este interpretador. Essa forma de execução só não possui um alto *overhead* por causa do compilador JIT acoplado ao segundo interpretador [85]. Este compilador gera código para caminhos de execução frequentes e consegue gerar bons resultados, segundo os próprios autores.

Os princípios de compilação JIT descritos na Seção 4 podem ser utilizados para classificar os sistemas que empregam este tipo de compilação. Desta forma, os sistemas descritos no presente capítulo são classificados da seguinte forma:

- **Tipo do sistema** que pode ser: interpretação mais compilação, compilação mais recompilação ou ainda uma variação deste último, na qual cada unidade de código é compilada imediatamente antes de sua primeira invocação, sem ser recompilada.
- **Manutenção do sistema**, podendo ser: JIT, *smart* JIT, ou compilação contínua.
- **Forma de detecção das regiões críticas** que pode ser por meio de contadores ou fração de tempo. É importante ressaltar que em implementações cuja manutenção do sistema utiliza a abordagem JIT não existe um mecanismo de detecção de regiões críticas, pelo fato de cada unidade de código invocada ser compilada antes de sua invocação. Portanto, em tais sistemas não existe uma forma de detecção de regiões críticas.
- **Estrutura das regiões críticas** podendo ser: sub-rotina, região ou caminho de execução.

- **Política de aplicação de otimizações de código** que indica se o processo de compilação emprega ou não otimizações com o objetivo de gerar um código de qualidade. As possíveis políticas são: ingênua (poucas otimizações são aplicadas), agressiva (diversas otimizações são aplicadas), ou níveis (o sistema alterna durante a execução o nível de otimização que será aplicado).

A Tabela 1 apresenta uma comparação entre os sistemas apresentados nesta seção.

Sistema	**Tipo de Sistema**	**Formas de Manutenção**	**Forma de Detecção**	**Estrutura das Regiões Críticas**	**Política de Otimizações**
Adaptive Fortran	IC	*Smart* JIT	Contadores	Sub-rotina	Ingênua
Sistema de Smalltalk-80	CR	JIT	NE	Sub-rotina	INF
SELF	CR	*Smart* JIT	Contadores	Região	Ingênua
Kaffe	C	JIT	NE	Sub-rotina	INF
Jikes RVM	CR	CC	Tempo	Sub-rotina	Níveis
JUDO	CR	*Smart* JIT	Contadores Tempo	Sub-rotina	Agressiva
HiPE	IC	*Smart* JIT	Contadores Tempo	Sub-rotina	INF
Máquina Virtual IBM	IC	*Smart* JIT CC	Contadores	Sub-rotina	Níveis
HotPathVM	IC	*Smart* JIT	INF	Caminho	INF
YAP com JIT	IC	*Smart* JIT	Contadores	Sub-rotina	Ingênua
Sun Hotspot Client	IC	*Smart* JIT	Contadores Tempo	Sub-rotina	Ingênua
Sun Hotspot Server	IC	CC	Contadores Tempo	Sub-rotina	Agressiva
Tamarin-Trace	IC	*Smart* JIT	Contadores	Caminho	Agressiva
TraceMonkey	IC	*Smart* JIT	Contadores	Caminho	INF
LuaJIT	IC	*Smart* JIT	Contadores	Caminho	Ingênua
ILDJIT	IC	*Smart* JIT	INF	Sub-rotina	INF
HappyJIT	IC	*Smart* JIT	Contadores	Caminho	INF

Tabela 1. Comparação entre os sistemas JIT.

Na Tabela 1 a primeira coluna apresenta o nome do sistema, a segunda o seu tipo, a terceira a forma utilizada na manutenção, a quarta a forma de detecção de regiões críticas, a quinta a estrutura das regiões críticas e por fim, a última coluna apresenta a política utilizada na aplicação de otimizações de código. As siglas nesta tabela significam: (IC) interpretação mais compilação; (CR) compilação mais recompilação; (C) compilação - para aqueles sistemas que compilam na primeira invocação sem recompilar; (CC) compilação contínua; (NE) não existente; (INF) informação não fornecida.

Em síntese, a maioria dos sistemas apostam na estratégia de interpretar e compilar código. Provavelmente, a pouca quantidade de sistemas que compilam e recompilam se deve pelo trabalho de implementar um compilador-base que, apesar de simples, é ainda mais complexo que um interpretador. Outro ponto importante é referente às estratégias de otimização de código, que mostram ser pouco investigadas no campo de compilação JIT. Por outro lado, mecanismos para detectar regiões críticas tem se mostrado mais importantes. De fato, diversos sistema têm procurado alcançar desempenho por meio da detecção e geração de código para porções de código que não se limitam ao escopo de uma sub-rotina.

6 Considerações Finais

Uma característica diferencial de compiladores JIT frente ao compiladores tradicionais é que a geração de código nativo ocorre em tempo de execução. Mesmo que isso seja preocupante no quesito de desempenho, sistemas que empregam compiladores JIT são atualmente, os principais componentes integrados que incrementam desempenho às máquinas virtuais. Isso se deve principalmente à criação e aprimoramento de técnicas que se baseiam em hipóteses relacionadas à organização e comportamento dos programas que são, infelizmente, difíceis (senão impossíveis) de se prever, mesmo durante o tempo de execução. Dessa forma, as pesquisas não possuem outra alternativa senão (1) unir estratégias para obtenção de outra [30], (2) estender as existentes [35, 34, 31] ou (3) buscar melhorias nas arquiteturas de hardware [59, 43, 42]. Portanto, é de se esperar que sistemas JIT futuros mantenham o uso das mesmas estratégias, porém aprimoradas, empregadas nos sitemas atuais.

As estratégias de interpretação mais compilação e compilação mais recompilação ainda permanecerão por muito tempo. Para sistemas de interpretação mais compilação existe o *overhead* da interpretação. Por outro lado, em sistemas que utilizam compilação mais recompilação existe o *overhead* de compilar o código de todo o programa, embora sem ou com poucas otimizações. Por isso é difícil avaliar qual abordagem se sobressairá sobre outra. Contudo, Suganuma *et al* [77] conseguiram importantes informações de desempenho relativas às abordagens clássicas de compilação JIT. Com um *framework* específico, os autores observaram que a estratégia de interpretação mais compilação surpreedentemente executou programas de forma mais rápida que a estratégia de compilação mais recompilação. No

mesmo trabalho, o uso de contadores na detecção de regiões críticas provou ser mais preciso que o uso de tempo. Tais resultados, entretanto, não foram necessariamente adotados na construção de todos os sistemas JIT que vieram depois, contudo, podem ser considerados (juntamente com o resultado de outros trabalhos) no projeto de futuras máquinas virtuais que incorporam compilação JIT.

Outro questão que permancerá é o uso de um sistema de otimização adaptativa. O fato de a tendência dos sistemas JIT convergir ao uso de níveis de otimização se deve ao fato de que tais sistemas se adaptam melhor às características dos programas. Programas rápidos quando compilados com o uso de otimizações mais leves ocasionam pouco *overhead* de compilação. E programas com longo tempo de execução quando compilados com o uso de otimizações mais agressivas ocasionam um código mais eficiente [44].

Diversas pesquisas têm apresentado tendências para a próxima geração de sistemas com compilação dinâmica. Pesquisas na área de otimização [55, 39, 56] têm demonstrado que os sistemas poderão se beneficiar de estudos sobre os campos de ação das otimizações, projetadas automaticamente por algoritmos heurísticos. Adicionalmente, os sistemas poderão se basear em aprendizagem de máquina, para que baseado nas otimizações e características de programas executados anteriormente, o sistema se adapte com o tempo [86, 87].

Bruening e Duesterwald [62] demonstraram que a escolha de métodos (e similares) como unidades de compilação é uma escolha muito ruim. Adicionalmente, na visão de sistemas JIT, a compilação baseada em regiões é muito conservativa. Portanto, baseado nas observações de Inoue *et al* [70], sistemas JIT baseados em grandes caminhos de execução podem ser a melhor escolha de implementação.

Kulkarni [43] destaca que o sistema JIT pode ajustar a quantidade de *threads* de compilação e o limite (contador ou tempo) para tornar uma região crítica. É necessário salientar que, quanto mais *threads* de compilação ativas, melhor seria um valor baixo de limite, segundo as observações do próprio autor.

Provavelmente uma diferença substancial virá do uso de ferramentas como GNU LIGHTNING [88] e LLVM [89] na produção de novos compiladores, devido as facilidades oferecidas por elas. Relacionado a isso, é de esperar que em não muito tempo novos compiladores estarão sendo desenvolvidos com base nestas ferramentas. GNU LIGHTNING contém uma série de desvantagens: possui uma quantidade limitada de registradores, não implementa um otimizador *peephole*, não implementa um escalonador de instruções, e não possui uma ferramenta para análise de código, porém é uma ferramenta rápida na construção dos compiladores, ocupa pouco espaço em memória e garante portabilidade. LLVM é uma ferramenta mais robusta e oferece recursos de modo que o projetista se preocupe somente em implementar um *parser* que transforme a linguagem fonte na representação intermediária suportada. Além disto, LLVM provê diversas otimizações em tempo de compilação, ligação e

execução. Outra vantagem desta ferramenta é a possibilidade de ordenar as otimizações da melhor forma possível, o que abre oportunidades para a incorporação de planos de otimização específicos.

Referências

[1] M. L. Scott. *Programming Languages Pragmatics*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2009.

[2] R. W. Sebesta. *Concepts of Programming Languages*. Addison Wesley, San Francisco, CA, USA, 2009.

[3] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, United States, 1984. ACM.

[4] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the ACM Conference on Programming language Design and Implementation*, pages 146–160, Portland, Oregon, United States, 1989. ACM.

[5] C. Chambers and D. Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 150–164, 1990.

[6] C. D. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, Stanford, CA, USA, 1992.

[7] The Kaffe Team. The Kaffe Virtual Machine. Disponível em: https://github.com/kaffe/kaffe. Acesso em: 09 dez. 2011.

[8] U. Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford, CA, USA, 1995.

[9] U. Hölzle and D. Ungar. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the ninth annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 229–243, Portland, Oregon, United States, 1994. ACM.

[10] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-time Compiler. *IBM Systems Journal*, 39:175–193, January 2000.

[11] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings of the ACM Conference on Java Grande*, pages 129–141, San Francisco, California, United States, 1999. ACM.

[12] A. F. da Silva and V. S. Costa. Design, Implementation, and Evaluation of a Dynamic Compilation Framework for the YAP System. In *Proceedings of the International Conference on Logic Programming*, pages 410–424, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] T. Suganuma, T. Ogasawara, K. Kawachiya, M. Takeuchi, K. Ishizaki, A. Koseki, T. Inagaki, T. Yasue, M. Kawahito, T. Onodera, H. Komatsu, and T. Nakatani. Evolution of a Java Just-In-Time Compiler for IA-32 Platforms. *IBM Journal of Research and Development*, 48:767–795, 2004.

[14] A. F. da Silva and V. S. Costa. Our Experiences with Optimizations in Sun's Java Just-In-Time Compilers. *Journal of Universal Computer Science*, 12(7):788–810, jul 2006.

[15] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[16] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[17] J. Aycock. A Brief History of Just-In-Time. *ACM Computing Surveys*, 35:97–113, 2003.

[18] J. G. Mitchell, A. J. Perlis, and H. R. Van Zoeren. LC^2: A Language for Conversational Computing. In *Proceedings of 1967 ACM Symposium*, New York, NY, USA, 1968. Academic Press.

[19] P. S. Abrams. *An APL Machine*. PhD thesis, Stanford, CA, USA, 1970.

[20] D. E Knuth. An Empirical Study of FORTRAN Programs. *Software Practice and Experience*, 1(2):105–133, 1971.

[21] R. J. Dakin and P. C. Poole. A Mixed Code Approach. *The Computer Journal*, 16(3):219–222, 1973.

[22] P. J. Brown. Throw-Away Compiling. *Software: Practice and Experience*, 6(3):423–434, 1976.

[23] G. J. Hansen. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs.* PhD thesis, Pittsburgh, PA, USA, 1974.

[24] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java Just in Time. *IEEE Micro*, 17:36–43, 1997.

[25] P. Tyma. Why Are We Using Java Again? *Communications of the ACM*, 41:38–42, 1998.

[26] M. P. Plezbert and R. K. Cytron. Does "Just In Time" = "Better Late than Never"? In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 120–131, Paris, France, 1997. ACM.

[27] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM Conference on Programming language Design and Implementation*, pages 13–26, Vancouver, British Columbia, Canada, 2000. ACM.

[28] K. Reinholtz. Java will be Faster than C++. *ACM SIGPLAN Notices*, 35:25–28, 2000.

[29] J. H. Foleisss and A. F. da Silva. Reavaliando a Lacuna do Desempenho entre as Linguagens Java, C e C++. In *Anais da Conferência Latino-Americana de Informática*, pages 1–15, Assunção, Paraguai, Novembro 2010.

[30] C. Krintz. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 69–78, San Francisco, California, 2003. IEEE Computer Society.

[31] T. Suganuma, T. Yasue, and T. Nakatani. A Region-Based Compilation Technique for a Java Just-In-Time Compiler. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 312–323, San Diego, California, USA, 2003. ACM.

[32] K. V. S. Kumar. When and What to Compile/Optimize in a Virtual Machine? *ACM SIGPLAN Notices*, 39:38–45, 2004.

[33] G. Agosta, S. C. Reghizzi, P. Palumbo, and M. Sykora. Selective Compilation Via Fast Code Analysis and Bytecode Tracing. In *Proceedings of the ACM Symposium on Applied Computing*, pages 906–911, Dijon, France, 2006. ACM.

[34] S.-W. Lee, S.-M. Moon, and S.-M. Kim. Enhanced Hot Spot Detection Heuristics for Embedded Java Just-In-Time Compilers. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 13–22, Tucson, AZ, USA, 2008. ACM.

[35] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an Effective JIT Compiler for Resource-Constrained Devices. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 144–153, Ottawa, Ontario, Canada, 2006. ACM.

[36] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a Transparent Dynamic Optimization System. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000. ACM.

[37] C. Zhao, Y. Wu, J. G., and C. Amza. Lengthening Traces to Improve Opportunities for Dynamic Optimization. In *Proceedings of the Workshop on Interaction Between Compilers and Computer Architectures*, pages 1–10, Salt Lake City, UT, 2008.

[38] H. Hayashizaki, P. Wu, H. Inoue, M. J. Serrano, and T. Nakatani. Improving the Performance of Trace-Based Systems by False Loop Filtering. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 405–418, Newport Beach, California, USA, 2011. ACM.

[39] 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*, pages 62–72, Toronto, Ontario, Canada, 2010. ACM.

[40] 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. In *Proceedings of the ACM Conference on Object-oriented Programing, Systems, Languages, and Applications*, pages 187–204, Anaheim, California, USA, 2003. ACM.

[41] Willian Stallings. *Arquitetura e Organização de Computadores*. Prentice Hall, Porto Alegre, 8 edition, 2005.

[42] P. A. Kulkarni, M. Arnold, and M. Hind. Dynamic Compilation: the Benefits of Early Investing. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 94–104, San Diego, California, USA, 2007. ACM.

[43] P. A. Kulkarni. JIT Compilation Policy for Modern Machines. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 773–788, Portland, Oregon, USA, 2011. ACM.

[44] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, Minnesota, United States, 2000. ACM.

[45] M. Paleczny, C. Vick, and C. Click. The Java Hotspot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 1–12, Monterey, CA, USA, 2001.

[46] S. Campanoni, G. Agosta, and S. C. Reghizzi. A Parallel Dynamic Compiler for CIL Bytecode. *SIGPLAN Notices*, 43:11–20, 2008.

[47] S. Campanoni, G. Agosta, S. C. Reghizzi, and A. Di Biagio. A Highly Flexible, Parallel Virtual Machine: Design and Experience of ILDJIT. *Software–Practice & Experience*, 40:177–207, 2010.

[48] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5:1–32, 2008.

[49] M. Pettersson, K. F. Sagonas, and E. Johansson. The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation. In *Proceedings of the International Symposium on Functional and Logic Programming*, pages 228–244, London, UK, 2002. Springer-Verlag.

[50] Mike Pall. The LuaJIT project. Disponível em: http://luajit.org. Acesso em: 02 dez. 2011.

[51] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for Web 3.0: Trace Compilation for the Next Generation Web Applications. In *Proceedings of the ACM International Conference on Virtual Execution Environments*, pages 71–80, Washington, DC, USA, 2009. ACM.

[52] A. Homescu and A. Şuhan. HappyJIT: a Tracing JIT Compiler for PHP. In *Proceedings of the 7th Symposium on Dynamic Languages*, pages 25–36, Portland, Oregon, USA, 2011. ACM.

[53] Andreas Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-Based Just-In-Time Type Specialization for Dynamic Languages. In *Proceedings of the ACM Conference on Programming language Design and Implementation*, pages 465–478, Dublin, Ireland, 2009. ACM.

[54] R. N. Lopes. Execução de Prolog com Alto Desempenho. Master's thesis, Universidade do Porto, Porto, Portugal, Master thesis, Universidade do Porto, Portugal. 1996.

[55] J. Cavazos and M. F. P. O'Boyle. Automatic Tuning of Inlining Heuristics. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 14–24, Washington, DC, USA, 2005. IEEE Computer Society.

[56] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-Space Exploration. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 204–215, San Francisco, California, 2003. IEEE Computer Society.

[57] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2007.

[58] M A. Namjoshi and P. A. Kulkarni. Novel Online Profiling for Virtual Machines. In *Proceedings of the ACM International Conference on Virtual Execution Environments*, pages 133–144, Pittsburgh, Pennsylvania, USA, 2010. ACM.

[59] S. Campanoni, M. Sykora, G. Agosta, and S. C. Reghizzi. Dynamic Look Ahead Compilation: A Technique to Hide JIT Compilation Latencies in Multicore Environment. In *Proceedings of the International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software*, pages 220–235, Berlin, Heidelberg, 2009. Springer-Verlag.

[60] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3 edition, 2007.

[61] M. Arnold, Fink S. J., Grove D., Hind M., and Sweeney P. F. A Survey of Adaptive Optimization in Virtual Machines. In *Proceedings of the IEEE. Special Issue on Program Generation, Optimization, and Adaptation*, 2004.

[62] D. Bruening and E. Duesterwald. Exploring Optimal Compilation Unit Shapes for an Embedded Just-In-Time Compiler. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 13–20, Monterey, California, 2000.

[63] R. M. Karp. On-line Algorithms Versus Off-line Algorithms: How Much is it Worth to Know the Future? In *Proceedings of the IFIP World Computer Congress on Algorithms, Software, Architecture*, volume 1, pages 416–429, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[64] J. Whaley. Partial Method Compilation Using Dynamic Profile Information. In *Proceedings of the ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 166–179, Tampa Bay, FL, USA, 2001.

[65] T. Suganuma, T. Yasue, and T. Nakatani. A Region-Based Compilation Technique for Dynamic Compilers. *ACM Transactions on Programming Languages and Systems*, 28:134–174, 2006.

[66] M. Serrano. Inline Expansion: When and How? In *Proceedings of the International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 143–157, London, UK, 1997. Springer-Verlag.

[67] R. E. Hank, W.-M. W. Hwu, and B. R. Rau. Region-Based Compilation: an Introduction and Motivation. In *Proceedings of the International Symposium on Microarchitecture*, pages 158–168, Ann Arbor, Michigan, United States, 1995. IEEE Computer Society Press.

[68] D. Bruening and S. Amarasinghe. Maintaining Consistency and Bounding Capacity of Software Code Caches. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 74–85, Washington, DC, USA, 2005. IEEE Computer Society.

[69] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A Dynamic Optimization System. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization*, New York, 2000. ACM Press.

[70] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A Trace-Based Java JIT Compiler Retrofitted from a Method-Based Compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 246–256, Chamonix, France, 2011. IEEE.

[71] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code Generation for Expressions with Common Subexpressions. *Journal of the ACM*, 24:146–160, 1977.

[72] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26:345–420, 1994.

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

[74] Marc M. Brandis and Hanspeter Mössenböck. Single-pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Transactions on Programming Languages and Systems*, 16:1684–1698, November 1994.

[75] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating Out of Static Single Assignment Form. In *Proceedings of the International Symposium on Static Analysis*, pages 194–210, London, UK, 1999. Springer-Verlag.

[76] David A. Patterson and John L. Hennessy. *Organização e Projeto de Computadores: Interface Hardware e Software*. Campus, Rio de Janeiro, 3 edition, 2005.

[77] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and Evaluation of Dynamic Optimizations for a Java Just-In-Time Compiler. *ACM Transactions on Programming Languages and Systems*, 27:732–785, 2005.

[78] A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[79] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[80] Robert Lougher. Jam Virtual Machine. Disponível em: http://jamvm.sourceforge.net/. Acesso em: 29 jan. 2012.

[81] A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Proceedings of the ACM Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 944–953, Portland, Oregon, USA, 2006. ACM.

[82] M. Lutz and D. Ascher. *Aprendendo Python*. Bookman, Porto Alegre, PR, Brasil, 2007.

[83] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the Symposium on Dynamic Languages*, pages 53–64, Montreal, Quebec, Canada, 2007. ACM.

[84] The PHP Company. PHP and Zend Engine. Disponível em: http://www.zend.com/en/community/php/. Acesso em: 29 jan. 2012.

[85] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, Genova, Italy, 2009. ACM.

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

[87] J. Cavazos and M. F. P. O'Boyle. Method-Specific Dynamic Compilation Using Logistic Regression. In *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 229–240, Portland, Oregon, USA, 2006. ACM.

[88] GNU Operating System. GNU Lightning. Disponível em: http://www.gnu.org/software/lightning/. Acesso em: 09 dez. 2011.

[89] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, Palo Alto, California, 2004.