UNIVERSIDADE ESTADUAL DE MARINGÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE INFORMÁTICA

TCL JIT

GUILHERME HENRIQUE POLO GONÇALVES

TG-CC-10

Maringá - Paraná

2010

UNIVERSIDADE ESTADUAL DE MARINGÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE INFORMÁTICA

Tcl JIT

Guilherme Henrique Polo Gonçalves

TG-CC-10

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação, do Centro de Tecnologia, da Universidade Estadual de Maringá.
Orientador: Prof. Dr. Anderson Faustino da Silva

Maringá - Paraná

2010

GUILHERME HENRIQUE POLO GONÇALVES

TCL JIT

Este exemplar corresponde à redação final da monografia aprovada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação da Universidade Estadual de Maringá, pela Banca Examinadora formada pelos seguintes membros:

Prof. Dr. Anderson Faustino da Silva
Departamento de Informática, CTC, DIN

Profa. Dra. Valéria Delisandra Feltrim
Departamento de Informática, CTC, DIN

Prof. Ms. José Roberto Vasconcelos
Departamento de Informática, CTC, DIN

Maringá - Paraná

2010

Resumo

A compilação dinâmica tem sido utilizada em implementações de linguagens de programação interpretadas que buscam maior desempenho. De modo a trazer esta técnica para a linguagem `Tcl`, um compilador JIT não-otimizador para um subconjunto da mesma é proposto. Optou-se pelo modo misto de execução, fazendo com que a execução de código de máquina, gerado por este compilador dinâmico, e a interpretação pura se alternem. A unidade de compilação foi tomada como procedimentos por inteiro, tendo a compilação dos mesmos disparada no momento em que é atingido um limite de execuções interpretadas realizadas com sucesso. Uma representação intermediária construída por quádruplas foi escolhida, sendo estas utilizadas até a fase de geração de código para a arquitetura IA–32. Simplicidade, manutenibilidade e flexibilidade foram as características seguidas na construção desta ferramenta. Para avaliar o desempenho do compilador, foram desenvolvidos 6 *benchmarks* específicos. Em média, os dados coletados indicaram uma melhoria entre 1,25 e 23,55 vezes em relação ao tempo de execução por parte do interpretador. O tempo de compilação ficou abaixo de 80 micro segundos. Além disso, o código gerado reduziu em até 96,9% a quantidade de instruções de máquina executadas em comparação ao interpretador.

Palavras-chave: Tcl, JIT, código de máquina

Abstract

Dynamic compilation has been used in implementations of interpreted programming languages that seek higher performance. In order to bring this technique to the `Tcl` language, a non-optimizing JIT compiler for a subset of it is proposed. Our choice was for the mixed mode of execution, this way the execution may alternate between the machine code generated by this dynamic compiler and the pure interpretation. Procedures as whole were taken as the compilation unit, where the compilation of those are fired upon hitting a limit of successful interpreted executions. Quadruples were chosen to form the intermediate representation and these were used till the code generation phase for the IA–32 architecture. Simplicity, maintainability and flexbility were the characteristics followed in the development of this tool. To measure the performance of the compiler, six specific benchmarks were developed. On average, the collected data indicate an improvement between 1.25 and 23.55 times in relation to the execution time by the interpreter. The compilation time stayed under 80 micro seconds. Also, the generated code reduced up to 96.9% the amount of executed machine instructions in comparison to the interpreter.

Keywords: Tcl, JIT, machine code

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
DAG	Directed Acyclic Graph
EBC	Extended Bytecode
EBNF	Extended Backus-Naur Form
GDB	GNU Debugger
JDK	Java Development Kit
JIT	Just-In-Time
JVM	Java Virtual Machine
LLVM	Low Level Virtual Machine
RISC	Reduced Instruction Set Computer
RVM	Research Virtual Machine
SSA	Static Single Assignment
Tcl	Tool Command Language
TVM	Tcl Virtual Machine
UTF-8	8-bit Unicode Transformation Format

1 Introdução

As implementações de linguagens de programação dividem-se em três classes ao considerar o produto final de seus compiladores e demais ferramentas envolvidas no processo de tradução. A primeira classe envolve linguagens que realizam exclusivamente compilação até chegar-se a código executável, a segunda engloba aquelas que unicamente interpretam o código fonte. A terceira classe, a mais flexível, contem aquelas que operam em um modo misto (ou híbrido). Linguagens que, de alguma forma, realizam tradução em tempo de execução fazem parte deste último modo. A compilação híbrida está diretamente ligada a este trabalho.

Essa divisão em classes está relacionada com o avanço da computação, pois este propiciou a criação de linguagens de mais alto nível que, apesar de terem desempenho inferior à linguagens de baixo nível, são viáveis de se utilizar. Também há uma ligação entre a facilidade e a criação de programas mais complexos, incentivando linguagens que visam portabilidade entre sistemas e rapidez de desenvolvimento sem que sejam necessariamente compiladas (primeira classe). Considerando-se tempo de execução e consumo de memória, é esperado que um programa ganhe nesses quesitos quando implementado numa linguagem da primeira classe. Por essa razão, os primeiros compiladores, devido aos recursos existentes na época, limitavam-se a esta forma e as linguagens ainda não se preocupavam com facilidade de uso. Mas, com o surgimento de programas maiores, viu-se a necessidade de fornecer ao usuário a chance de interagir com o sistema de forma simplificada. No caso do sistema operacional UNIX, isto ocorreu por meio da criação de um *shell* que funciona puramente como um interpretador.

A interpretação por parte das linguagens sacrifica desempenho em favor de um alto nível de abstração, possibilitando maior expressividade, facilidade de desenvolvimento, portabilidade, flexibilidade e dinamicidade, entre outras características.

De maneira semelhante ao *shell* do UNIX, a linguagem de programação `Tcl` (*Tool Command Language*) teve como principal objetivo a facilidade de incorporação (*embed-*

ding) a aplicações que necessitassem de uma linguagem de comandos para interação com usuários (OUSTERHOUT, 1989). Na época não havia um incentivo computacional para utilizar este tipo de linguagem de forma ampla e, assim, o custo despendido pela interpretação não tinha impacto sobre o sistema maior, pois os programas criados com estas linguagens tendiam a ser bastante curtos. O autor da `Tcl` chegou a dizer que "... quase todos "programas" em `Tcl` serão curtos, muitos de apenas uma linha. A maioria dos programas serão escritos, executados uma vez ou talvez poucas vezes e, então, descartados. ..." (OUSTERHOUT, 1989).

Com a evolução da capacidade computacional, as linguagens puramente interpretadas começaram a ser utilizadas em maior escala e o custo envolvido tornou-se mais óbvio. Começaram a surgir aplicativos desenvolvidos com a `Tcl` contendo milhares de linhas, como, por exemplo, exmh (WELCH, 1995), OpenACS (HERNÁNDEZ, 2005) ou mesmo o *benchmark* de testes do GDB (*GNU Debugger*) (GILMORE; SHEBS, 2010). Com isso, problemas relacionados ao desempenho tornaram-se reais. A "solução" inicial encontrada para a `Tcl` foi a reescrita de trechos críticos na linguagem `C` (KERNIGHAN; RITCHIE, 1988), visto que a `Tcl` disponibiliza uma interface para a mesma. Entretanto, com isto perde-se benefícios como gerenciamento de memória automático e maior facilidade de desenvolvimento – características comuns dessas linguagens. Por esta razão, os implementadores têm buscado melhorar a performance de linguagens interpretadas.

Uma maneira de melhorar o desempenho da linguagem, sem retirar suas vantagens, envolve o uso do modo misto de compilação. As linguagens `Java` (GOSLING; JOY; STEELE, 2005) e `Python` (ROSSUM; DRAKE, 2009) por meio do Psyco (RIGO, 2004) são exemplos que fazem uso desta técnica, mas somente a primeira dessas atualmente consegue não depender de reescrita em outras linguagens para alcançar uma boa performance para os mais variados tipos de aplicações.

A definição da terceira classe de implementações de linguagem, envolvendo compilação mista, têm sido vaga. Isso é reflexo do que é possível nesse modo. Poder-se-ia considerar a linguagem `Java` e uma implementação da JVM (*Java Virtual Machine*) (LINDHOLM; YELLIN, 1999) que trabalha com *bytecodes* e também permite compilação JIT (*Just-In-Time*) para código de máquina. Após realizar a tradução de código fonte para *bytecodes*, a máquina virtual pode iniciar sua execução e interpretar essa forma de código. É possível que, durante a execução do programa, de acordo com critérios estabelecidos, os *bytecodes* venham a ser propriamente convertidos em código de máquina e ajustados no ambiente de execução de forma a possibilitar sua execução direta sem uso de um interpretador. Mas

esta situação cobre uma única possibilidade, deixando outras de fora.

De forma geral, JIT refere-se a tradução de código sob demanda. Isso possibilita que o trabalho de Lewis (1996) – responsável por efetivamente trazer o modo misto para a `Tcl` – seja descrito como um sistema JIT para a `Tcl`, pois a conversão de código fonte para *bytecodes* ocorre somente durante a chamada de procedimentos que ainda não tenham sido "*byte*-compilados" ou que tenham sofrido alguma alteração entre chamadas. Ou seja, por assim ocorrer, na `Tcl` a compilação para *bytecodes* é dita ser feita *online* ao invés de *offline* como no caso da JVM mencionada acima. Ainda assim, não há a conversão para código de máquina em momento algum. Neste trabalho, o interesse está especificamente em tratar esta situação.

A compilação dinâmica (ou JIT) pode fazer uso de informações produzidas pelo programa em execução para guiar o processo de compilação. Com isso, linguagens tipicamente difíceis de serem analisadas e compiladas estaticamente, devido a uso de, por exemplo, tipos e/ou escopo dinâmico (HÖLZLE, 1994), ganham a oportunidade de melhoria de desempenho com o uso de tal sistema de compilação. Esta técnica tornou-se mais popular a partir de diversas implementações de JVM, como a CACAO (KRALL; GRAFL, 1997), JUDO (CIERNIAK; LUEH; STICHNOTH, 2000), Jalapeño (Jikes RVM (*Research Virtual Machine*)) (ALPERN et al., 2000) e também a IBM *Development Kit* (SUGANUMA et al., 2004), que reportaram resultados significantes na redução de tempo de execução quando comparado a um interpretador de *bytecodes*. Outras linguagens também têm recebido esforços nessa direção. Um dos primeiros trabalhos foi o de Deutsch e Schiffman (1984) para a `Smalltalk`; a linguagem `Self` contou com a implementação SELF-93, descrita no trabalho de Hölzle (1994) e contribuiu para JVMs criadas mais tarde; para o `Python` há o Psyco (RIGO, 2004) e mais recentemente também a *Unladen Swallow* que faz uso da LLVM (LATTNER; ADVE, 2004) . Todos estes trabalhos têm em comum, além do uso de compilação dinâmica, o uso de compiladores otimizadores.

Um compilador JIT requer que as estruturas internas sejam suficientemente eficientes, caso contrário torna-se inviável o uso do compilador em tempo de execução. As escolhas a cerca de quais representações intermediárias utilizar, como estruturar os dados, quais otimizações aplicar e algoritmos para diversas fases da compilação, devem ser feitas de forma a conseguir balancear baixo tempo de compilação com código gerado de alta qualidade. Ainda há o quesito de consumo de memória principal, que, apesar da crescente capacidade disponível, ainda costuma ser um recurso escasso em dispositivos embarcados. Sabe-se que o interpretador `Tcl` está presente em roteadores da Cisco (CISCO, 2010), pois

vem incluso no Cisco IOS (*Internetwork Operating Systems*) (DOOLEY; BROWN, 2006), mas esse trabalho não tem como foco tal tipo de dispositivo e, portanto, o consumo de memória não será um dos pontos levados em consideração.

Um fator importante durante o desenvolvimento de sistemas deste porte é a sua manutenibilidade. Um nível muito alto de abstração, impediria a utilização de uma aplicação de desempenho crítico. Por outro lado, um nível muito baixo dificultaria a correção/detecção de problemas e melhorias gerais. Uma alternativa para esse problema vem sendo desenvolvida no projeto PyPy (RIGO; PEDRONI, 2006), onde um interpretador para uma linguagem qualquer é escrito em `RPython` (PYPY, 2010) (uma implementação restrita da linguagem `Python`) e o PyPy realiza a tradução do mesmo para a linguagem `C` incluindo (atualmente) juntamente um compilador JIT. Entretanto, um dos objetivos do trabalho discutido aqui é analisar como um sistema de tamanho reduzido compete com sistemas mais robustos. Não se tem a intenção de fornecer um ambiente de alto nível para construção de outras máquinas virtuais com ou sem compiladores JIT para `Tcl`, mas sim uma implementação específica e direta. A escolha da linguagem `C` reflete esse objetivo porque não adiciona novas dependências ao núcleo da linguagem `Tcl` além de possibilitar criação de programas com desempenho aceitável.

Apesar da linguagem `Tcl` ainda não ter recebido um compilador JIT que destina-se a produzir código nativo, o trabalho por Vitale e Abdelrahman (2004) lida com a eliminação do *overhead* de decodificação dos *bytecodes*, introduzido pelo trabalho de Lewis (1996), fazendo uso de *templates* que contém as instruções em código nativo utilizadas para interpretar cada *bytecode*. Esse código é obtido por meio da compilação do próprio interpretador `Tcl` e cada *template* é copiado múltiplas vezes, numa área de memória alocada em tempo de execução, conforme a quantidade de cada *bytecode* gerado. Nesse mesmo trabalho, o interpretador foi modificado de forma a executar somente o código formado pela concatenação de *templates*, eliminando o *overhead* de decodificação. Demonstrou-se que em certos casos o desempenho da linguagem pode melhorar em até 60% com a aplicação dessa técnica. Esse trabalho é provavelmente o mais próximo, quando considerando somente a `Tcl`, do que se pretende produzir aqui. Mas ele não gera código, apenas copia código já gerado por um compilador estático e replica conforme necessário, fazendo os devidos ajustes, em tempo de execução. Por um lado o tempo de "compilação" é bastante baixo, porém, não dá espaço para técnicas de otimização, limitando o potencial de melhoria de desempenho para trabalhos futuros.

O presente trabalho propõe uma implementação inicial de um compilador JIT não-

otimizador para um subconjunto da linguagem `Tcl`, focando-se no trabalho com números. O propósito é obter um tempo de decodificação e interpretação reduzido em relação ao da máquina virtual da mesma. Espera-se que o nível de simplicidade, manutenibilidade e flexibilidade atingidos possam permitir extensões e desenvolvimentos futuros sobre este trabalho inicial. A arquitetura IA-32 (INTEL, 2009c) foi escolhida como alvo, pois está presente em boa parte dos computadores de uso pessoal. Embora a linguagem `Tcl` atualmente execute em outras arquiteturas, como IA-64 (INTEL, 2009c) e ARM (SEAL, 2001), pretendeu-se alcançar um nível de simplicidade que permita a portabilidade do sistema sem tornar a tarefa demasiadamente complexa. Para a representação intermediária, este trabalho fez uso de quádruplas (MUCHNICK, 1997), que são utilizadas para formar blocos básicos e construir grafos de fluxo de controle (CFG – *Control Flow Graph*) . Esta representação é de nível médio e o CFG é diretamente utilizado na geração de código de máquina.

O Capítulo 2 apresenta conceitos da compilação dinâmica e realiza uma revisão bibliográfica descrevendo trabalhos que, de alguma forma, influenciaram o desenvolvimento deste compilador JIT.

O Capítulo 3 inicia com uma descrição da linguagem `Tcl` e, em seguida, descreve a estrutura e o funcionamento do compilador JIT desenvolvido.

No Capítulo 4 são apresentados dados coletados que validam de forma experimental o desempenho alcançado. Os resultados obtidos demonstram até cerca de 29 vezes de redução de tempo de execução em relação ao interpretador da `Tcl`.

E, por fim, o Capítulo 5 apresenta as conclusões e os trabalhos futuros.

2 Compilação Dinâmica e Compiladores JIT

2.1 Compilação Dinâmica

O termo "compilação dinâmica" refere-se a técnica de tradução de código sob demanda. Ela têm sido utilizada para aumentar o desempenho de aplicativos, gerando código em tempo de execução. Aplicações variadas têm feito uso desta técnica (HÖLZLE, 1994), mas pode-se apontar implementações de linguagens de programação como fator motivante para a existência e evolução da mesma. A busca por interpretadores mais rápidos fez surgir a idéia da compilação dinâmica (HÖLZLE, 1994), pois, com ela, é possível reduzir o custo de avaliação de instruções e também elimina-se o *overhead* de decodificação de instruções. A emissão de código nativo não é um requisito dessa compilação, mas é o foco do presente trabalho.

Características como tipos dinâmicos e estruturas de dados dinâmicas incentivam a construção de interpretadores (SAFONOV, 2010) e, apesar de terem o tempo de execução acrescido de 10 vezes (PLEZBERT; CYTRON, 1997) a até 1000 vezes (SAFONOV, 2010), quando comparado a linguagens compiladas, as facilidades fornecidas atraem usuários. Apesar do quesito performance não ser o mais prioritário nessas linguagens, é fácil verificar que as implementações destas linguagens costumam ser feitas em `C` (RIGO; BOLZ, 2007). Logo, aparentemente, a facilidade de desenvolvimento quer estar unida com alto desempenho.

Linguagens interpretadas comumente apresentam-se como problemas a compiladores estáticos, isso é devido a falta de informações no momento da compilação *offline* (HÖLZLE, 1994). Entretanto, um compilador dinâmico tem acesso a todas informações produzidas ao longo da execução. Logo, é possível tratar dos problemas introduzidos por linguagens de mais alto nível e produzir código nativo que não seria possível, ou muito difícil, com compiladores estáticos.

Estes tradutores podem ser construídos das mais variadas formas. Em relação ao modo de operação dos compiladores dinâmicos, o trabalho de Plezbert e Cytron (1997) distingue 3 classes:

JIT (Just in Time) : O ambiente alterna entre a compilação sob demanda para código nativo e a execução do código nativo gerado; somente um das duas situações ocorre num dado momento

Compilação contínua : Realiza compilação em tempo de execução mas não exatamente sob demanda. Esse modo tenta compilar o máximo de código em paralelo a execução em uma tentativa de disponibilizar código nativo quando ocorrer a chamada de um procedimento. Caso ocorra uma chamada para um procedimento que ainda não tenha sido compilado, pode-se utilizar o interpretador

Smart JIT : Permite a execução mista entre máquina virtual e código nativo. Ao invocar um método, não necessariamente ocorre a compilação. Parâmetros precisam ser estabelecidos para determinar as condições que levam a emissão de código

Apesar desta diferenciação existir, diversos trabalhos (SUGANUMA et al., 2001, 2004; CIERNIAK; LUEH; STICHNOTH, 2000) têm utilizado o termo JIT no lugar de *Smart* JIT e o mesmo ocorre com o trabalho corrente.

A consequência de utilizar um dos dois primeiros tipos acima, em comparação ao terceiro, é a possibilidade de *delay* na inicialização do ambiente de execução. Isso ocorre porque logo no inicio a máquina virtual possivelmente invoca muitos métodos. Entretanto, qualquer um dos três impacta no desempenho da aplicação visto que eles consomem tempo de execução. Selecionar parâmetros adequados para compilação dinâmica alivia esse problema, pois pode-se compilar somente procedimentos que são muito executados. Um parâmetro que pode ser utilizado nesta decisão é a quantidade de invocações de um procedimento/método com decaimento ao longo do tempo (HÖLZLE, 1994) ou não. Este número também pode ser combinado com a quantidade de repetições de laços executadas.

Tradicionalmente, compiladores JIT têm trabalhado com compilação de métodos por inteiro. Entretanto, estes compiladores dispõem da flexibilidade de decidir o que constitui sua unidade de compilação. Trabalhos mais recentes descrevem o uso laços (BRUENING; DUESTERWALD, 2000), *traces* (BRUENING; DUESTERWALD, 2000) ou regiões (HANK; HWU, 1995) como alternativas a funções completas para compilação. Estas três outras formas surgiram como propostas de melhoria (HANK; HWU, 1995; BRUENING; DUESTERWALD,

2000) sobre o método tradicional. Elas permitem despender tempo de análise e geração de código somente em partes que realmente são mais frequentemente executadas. A compilação de regiões é uma generalização daquela com uso de *traces*, onde esta última é formada por um único ponto de entrada e múltiplas saídas (BRUENING; DUESTERWALD, 2000) enquanto que a primeira trabalha sobre uma coleção arbitrária de blocos básicos (AHO et al., 2006).

Compiladores dinâmicos podem ser otimizadores ou não. Entretanto, é mais comum encontrar a primeira forma visto que a intenção é gerar código para linguagens com recursos que consomem maior tempo de execução. Também torna-se interessante o uso de técnicas de otimização somente se o custo da compilação for dominado pelo ganho em desempenho. Sistemas mais robustos (HÖLZLE, 1994; CIERNIAK; LUEH; STICHNOTH, 2000; SUGANUMA et al., 2004) que empregam estes compiladores têm feito uso de mais de um compilador. Em um primeiro momento, a compilação ocorre com uso de um tradutor que gera código de forma rápida mas não tão eficiente. Isso reduz a ocorrência de pausas durante a execução. Ao detectar que este código vem sendo frequentemente utilizado, aplica-se um outro compilador que faz uso de otimizações mais dispendiosas. A recompilação dinâmica (HÖLZLE, 1994) também pode ser utilizada com o intuito de remover otimizações e possibilitar a inspeção de código.

2.2 Compiladores JIT

2.2.1 Smalltalk–80

A implementação da `Smalltalk-80` (GOLDBERG; ROBSON, 1983) de Deutsch e Schiffman (1984) faz uso da compilação dinâmica para transformar *v-code* em código nativo (chamado de *n-code*). A motivação para empregar esta técnica teve relação com os recursos da linguagem `Smalltalk`, como alocação dinâmica e procedimentos universalmente polimórficos (SEBESTA, 2009), que são difíceis de serem traduzidos de forma eficiente. Não foi feito uso do modo misto (*Smart JIT*), na ocorrência de uma chamada sempre realiza-se primeiro geração de código nativo, caso ainda não exista, para depois iniciar a execução do método.

Na época em que esta implementação ocorreu, haviam restrições severas de memória. Por este motivo, o trabalho teve a preocupação de verificar se o código nativo gerado seria paginado ou não. Caso isso viesse a ocorrer, o código seria descartado e gerado novamente quando outra invocação ao procedimento ocorresse. No presente trabalho, esse cuidado

não é tomado e não foi levado em consideração.

Durante esta implementação da `Smalltalk`, também houve a preocupação em permitir a utilização de compiladores dinâmicos variados, mas não todos ao mesmo tempo. Deutsch e Schiffman (1984) descrevem duas implementações que foram testadas, a primeira faz uso de otimizações *peephole* (MUCHNICK, 1997) esparsamente e obtém tamanho de código reduzido quando comparado a outra. A segunda implementação é mais agressiva nas otimizações aplicadas, aplicando a expansão *in-line* até para operações aritméticas e relacionais. Nessa configuração mais agressiva foram obtidos os melhores resultados, chegando a um tempo quase 2 vezes menor comparando-se com o interpretador puro. O trabalho aqui proposto alcança reduções maiores, mas apenas para um subconjunto pequeno da linguagem ao passo em que a implementação de Deutsch e Schiffman (1984) trabalha por completo na especificação da linguagem envolvida.

2.2.2 CACAO

A CACAO (KRALL; GRAFL, 1997) é um compilador JIT para a linguagem `Java` com foco no processador ALPHA (DEC, 1996).

Para construir sua representação intermediária, *bytecodes* `Java` são convertidos para uma forma de máquina de registradores. De forma semelhante com o trabalho realizado, buscou-se converter uma representação de máquina de pilha para outra que assemelha-se com a arquitetura de um processador RISC (*Reduced Instruction Set Computer*) (ALETAN, 1992) . Entretanto, o trabalho de Krall e Grafl (1997) tem a preocupação de eliminar as ineficiências da máquina de pilha (*load & store*) quando convertidas para registradores. O trabalho não da detalhes sobre a alocação de registradores, mas menciona que é feito de forma simples e rápida.

A arquitetura deste sistema ainda envolve a definição de um novo *layout* de objetos e métodos em `Java`, tornando o acesso mais rápido e utilizando menos memória.

Com os *benchmarks* utilizados, a CACAO obteve um desempenho de até 85 vezes superior ao comparar-se com o interpretador JDK (*Java Development Kit*). A CACAO também foi testada contra o compilador gcc (FREE SOFTWARE FOUNDATION, 2008) com uso da *flag* `-O3`, verificando-se que o tempo do compilador JIT foi entre 1.01 e 1.66 vezes pior que aquele obtido com o gcc.

2.2.3 Jikes RVM

O projeto Jikes RVM (*Research Virtual Machine*) (ALPERN et al., 2002) (até 2001 conhecida como Jalapeño JVM) teve como objetivos iniciais suportar a arquitetura PowerPC (IBM, 2005) e fornecer uma JVM de alto desempenho. Enquanto conhecida como Jalapeño JVM, sua distribuição era trabalhosa devido as licenças estabelecidas (ALPERN et al., 2005). A renomeação para Jikes RVM deveu-se a correções neste processo, tornando-a em um projeto de código livre. Neste momento também notou-se a necessidade de portar seu código para a arquitetura IA-32 (ALPERN et al., 2005), pelas mesmas razões defendidas neste trabalho.

O trabalho de Alpern et al. (2002), descreve que a Jalapeño/Jikes é escrita com a própria linguagem `Java`. Para que não seja necessário o uso de outra máquina virtual para iniciar sua execução, uma imagem de incialização executável pré-compilada, contendo todos os serviços essenciais para uma JVM, é criada para ser carregada em memória e então permitir sua execução.

Esta máquina virtual não trabalha com interpretação de *bytecodes*, um compilador *baseline* é utilizado para compilar todos os métodos de forma rápida. Métodos executados muito frequentemente são então recompilados com um compilador otimizador. Este compilador otimizador trabalha com 3 representações intermediárias: (1) uma de alto nível independente de arquitetura e que trabalha com transferência de registradores; (2) outra de baixo nível também independente mas com instruções semelhantes com as de uma arquitetura RISC (ALETAN, 1992); (3) e uma a nível da máquina alvo.

O compilador otimizador trabalha com seleção de instruções por meio de padrões em árvores (FRASER; HANSON; PROEBSTING, 1992) e a alocação de registradores é feita através de uma variação do método de varredura linear (POLETTO; SARKAR, 1999). Otimizações de eliminação de subexpressões comuns (MUCHNICK, 1997), eliminação de movimentação redundante, propagação de cópias (MUCHNICK, 1997), eliminação de código morto (ALLEN; KENNEDY, 2001), *in-line* de métodos, entre várias outras são aplicadas por este compilador.

Resultados do trabalho de Alpern et al. (2002) demonstraram que, com a implementação descrita da Jikes RVM para IA-32, a utilização de alocação de registradores e seleção de instruções produzem código até quase 2 vezes mais eficiente quando comparado a compilação sem estas técnicas. O presente trabalho não faz uso destas técnicas, mas espera-se que elas também colaborem com o desempenho quando aplicadas.

2.2.4 JUDO

A JUDO (CIERNIAK; LUEH; STICHNOTH, 2000) é uma outra JVM, ela faz uso de compilação dinâmica com dois tipos de compiladores e coleta informações em tempo de execução.

O primeiro desses compiladores é um mais simples, que gera código rapidamente, destinado a compilação de métodos invocados pela primeira vez. Durante a emissão de código, instruções que coletam dados a cerca da execução são inseridas. O segundo compilador é invocado quando as informações coletadas indicarem que certos métodos são executados muito frequentemente e, portanto, estes podem ser beneficiados com a aplicação de otimizações. Essa recompilação dinâmica é feita com o intuito de balancear o tempo gasto na compilação com o tempo efetivamente gasto na execução do programa.

As otimizações aplicadas incluem: eliminação de subexpressões comuns (MUCHNICK, 1997) durante a construção da representação intermediária; uma fase de verificação de pontos candidatos a *in-line*; propagação de cópias; desdobramento de constantes (MUCHNICK, 1997); eliminação de código morto e eliminação de verificação de subscritos em *array* (BODÍK; GUPTA; SARKAR, 2000). Também são feitas otimizações *offline* para inicialização de classes, *cast* e verificação de subscritos.

Este sistema foi projetado para trabalhar com a compilação de métodos por inteiro, sendo esta a maior semelhança com o trabalho proposto aqui.

De forma geral, a avaliação de performance indicou que o gerador de código ágil (parecido, mas ainda mais avançado por inlcuir algumas otimizações, com o proposto aqui) é o que apresenta pior performance em relação as outras duas formas avaliadas: (1) compilador otimizador sem informações de tempo de execução; (2) e outro que faz uso de recompilação dinâmica.

2.2.5 IBM JDK

O artigo de Suganuma et al. (2004) exibe a evolução de uma JVM desenvolvida na IBM que faz uso de compilação dinâmica com foco na arquitetura IA-32, assim como este trabalho.

A IBM JDK (*Java Development Kit*) utiliza 3 representações intermediárias: (1) EBC (*Extended Bytecode*); (2) quádruplas; (3) DAG (*Directed Acyclic Graph*). A segunda destas equivale à utilizada no trabalho em discussão, sendo baseada em registradores. A

terceira também é baseada em registradores mas faz uso da SSA (*Static Single Assignment*) (CYTRON et al., 1991).

O compilador dinâmico desse sistema utiliza 3 níveis de compilações. A cada nível mais otimizações vão sendo aplicadas, mas somente a de nível 1 (que têm menor custo) é invocada na mesma *thread* de execução da aplicação. Os outros níveis de compilação ocorrem em *threads* separadas e somente são utilizadas se o sistema de compilação detectar métodos que podem ser beneficiados por elas. Para tomar essa decisão, um coletor periodicamente faz amostragens do uso da CPU pelos métodos. O sistema mantém os métodos que mais utilizam a CPU em uma lista ligada, ordenada por maior uso, e os repassa para o compilador adequado em intervalos fixos. Não há necessariamente uma transição do nível de compilação 1 para o 2 e depois para o 3, pode ocorrer de, após passar pelo nível 1, um método ser compilado diretamente no nível 3.

Uma variedade de otimizações são aplicadas em cada nível. Entre elas estão a propagação de cópias e de constantes (CALLAHAN et al., 1986), eliminação de código morto, eliminação de verificação de exceção (KAWAHITO; KOMATSU; NAKATANI, 2000) e otimizações de laço (MUCHNICK, 1997).

Os resultados demonstram que o compilador de nível 1 conseguiu obter o segundo melhor desempenho em alguns casos, apesar de ser o que inclui a menor quantidade de otimizações. Entretanto, a combinação entre os 3 níveis resultou na melhor performance em todos os testes realizados.

2.2.6 Psyco

O Psyco (RIGO, 2004) trabalha com compilação JIT juntamente com especialização sobre a linguagem de programação `Python`. Por especialização entende-se a tradução de um programa qualquer em uma versão mais limitada do mesmo, na expectativa de que a versão especializada seja mais eficiente do que a original. O Psyco trabalha, especificamente, com especialização por avaliação parcial (JONES; GOMARD; SESTOFT, 1993). Houve uma tentativa de aplicar um método semelhante com a `Tcl` mas, diferentemente da linguagem `Python` onde tipos são associados a valores, não conseguiu-se determinar de forma eficiente o uso de tipos ao decorrer da aplicação.

Diversas versões de uma mesma função são possívelmente produzidas. As funções a serem especializadas são selecionadas por meio de um contador com decaímento exponencial, descrito em (HÖLZLE, 1994).

Em um *benchmark* envolvendo aritmética com números inteiros, o Psyco demonstrou melhoria de 109 vezes em relação ao interpretador `Python` padrão. O mesmo teste em `C` executou 281 vezes mais rápido que o interpretador. Ao medir o tempo de aritmética entre números complexos, um melhoria de 3,65 vezes foi reportada. Essa diminuição drástica entre os dois testes está relacionada com o fato do Psyco conseguir representar algumas funções de forma mais eficiente, especialmente aquelas envolvendo o uso de números inteiros.

3 Tcl JIT

3.1 A linguagem Tcl

A linguagem de programação `Tcl` foi criada em 1989 por John K. Ousterhout. Ela é interpretada e atualmente sua máquina virtual trabalha com *bytecodes*. Surgiu como uma linguagem orientada a comandos para servir a programas interativos, fornecendo tanto uma interface procedural quanto uma interface textual. A primeira destas é utilizada por programas que desejam incorporar e estender a `Tcl`, a outra permite que usuários programem diretamente com a linguagem.

Três características básicas diferenciam a `Tcl` de muitas outras linguagens: (1) orientada a comandos; (2) sintaxe curta; (3) tudo é *string* UTF–8 (*8-bit Unicode Transformation Format*). Um programa qualquer nesta linguagem pode ser visto como uma sequência de chamadas de funções, onde cada chamada é descrita pela seguinte EBNF (*Extended Backus-Naur Form*): `comando {argumento}`. Cada `comando` invocado pode: já pertencer a linguagem (*built-in*); ou ser um comando criado por meio de extensão da linguagem; ou ter sido criado com a própria `Tcl`. Os elementos que seguem um comando são passados como argumentos para o mesmo, deixando o tratamento semântico destes argumentos por conta do comando. Todo comando retorna uma *string*, vazia ou não.

Como consequência da primeira característica, *keywords* não existem na `Tcl`. Mesmo instruções como `if` ou `proc` são comandos (esta última é a responsável pela criação de novos comandos) e podem ser redefinidas, renomeadas ou removidas pelo usuário.

Por tratar tudo como *string*, os tipos de dados inicialmente se resumiam a um único tipo. Trabalhos realizados mais tarde (SAH, 1994; LEWIS, 1996) alteraram esta situação, modificando a linguagem de forma que possa haver uma representação interna mais eficiente conforme os valores vão sendo utilizados. Entretanto, representar tudo como *string* ainda é característico da `Tcl` pois essa forma mais eficiente precisa disponibilizar um meio de recuperar a *string* representante desta outra forma. É importante ver que certas opera-

ções podem não fazer sentido entre *strings*, como dividir “abacate” por “42”. Sendo assim, indiferente de haver uma representação interna mais eficiente ou não, os comandos ficam responsáveis por converter os valores, conforme a necessidade, para outros tipos para, então, realizar com sucesso ou não sua respectiva operação.

3.1.1 Sintaxe e semântica da linguagem

A sintaxe da `Tcl` é bastante curta se comparada a outras linguagens de uso geral. Considerando apenas a semântica da linguagem (e desconsiderando a semântica de implementação de cada comando), também tem-se que ela pode ser descrita rapidamente. Para entender como os comandos são avaliados, discute-se como eles são formados e entendidos pelo interpretador.

Os comandos são separados por “;” ou quebra de linha, contanto que estes elementos não estejam entre aspas duplas ou chaves (“{”, “}”). Separados os comandos, ainda é necessário executá-los. Para isso, a linguagem realiza duas etapas para cada comando. Na primeira etapa, a `Tcl` divide o comando em elementos, cuja definição será descrita a seguir, e realiza as substituições necessárias que serão descritas nesta seção. Separados os elementos, a segunda etapa trata de executar o comando. Para isto, o primeiro elemento é utilizado para localizar o procedimento associado ao comando e possibilitar sua execução; os demais elementos são tidos como argumentos para o procedimento.

Para definir o que são estes elementos mencionados acima, algumas notações adicionais (palavra simples/composta) à documentação da `Tcl` (TCL CORE TEAM, 2010) foram estabelecidas aqui. De modo geral, um elemento pode ser formado por uma palavra simples ou por uma palavra composta. As palavras simples são aquelas delimitadas por espaços e não fazem uso de notação adicional. Considerando o seguinte trecho: `puts stderr` π`=3`, que imprime a *string* “π=3” em `stderr`, tem-se três palavras simples. As palavras compostas são assim chamadas pois podem conter várias palavras simples mas, além disso, também contém notações especiais. Há três formas de se construir uma palavra composta. Ao delimitar um conjunto de palavras simples por aspas duplas, como em: `"olá, mundo!"`, forma-se uma palavra composta que representa um único parâmetro para algum comando. Um outro modo envolve trocar as aspas duplas por “{” e “}”. Essa troca apresenta efeitos na substituição, que será tratado adiante. Finalmente, utilizando colchetes também constrõe-se palavras compostas.

As substituições mencionadas anteriormente são agora descritas. Elas são dividi-

das em duas: substituição de comandos e substituição de variáveis. A primeira destas ocorre ao fazer uso da palavra composta que utiliza colchetes. Para entender o que ocorre na substituição de comandos, primeiro considere o seguinte trecho de código: `puts [expr {(1 + sqrt(5))/2}]`, que imprime como resultado uma *string* cujo conteúdo pode ser interpretado como um número que aproxima φ. O trecho anterior contém dois elementos: uma palavra simples e outra composta. A palavra composta foi delimitada por colchetes e, por isso, o interpretador `Tcl` entende que esta palavra inicia com um comando e que os demais elementos são parâmetros para este comando. Sendo assim, a `Tcl` avalia esse comando e substitui a palavra composta pelo resultado obtido. Na palavra composta há ainda outra palavra composta, mas delimitada por chaves. Nesse caso, não ocorre nenhuma substituição e o conteúdo entre chaves é repassado para o respectivo comando. No exemplo anterior, o comando `expr` fica responsável por tratar a *string* "`(1 + sqrt(5))/2`". Caso houvessem outras palavras compostas delimitadas por colchetes, a linguagem `Tcl` define uma ordem natural de avaliação da esquerda para direita em sequência.

A substituição de variáveis ocorre nos casos em que um elemento tem sufixo "$", trocando o elemento pelo valor da variável representada. Há três formatos previstos para esta substituição: `$var`, `$var(chave)` e `${var}`. No primeiro caso, `var` descreve o nome de uma variável escalar. No segundo, `var` é aceito como um *array* e `chave` é uma *string*, que pode ser formada por uma palavra composta, que, ao ser avaliada, dá um nome para um elemento deste *array*. O último caso também se aplica a variáveis escalares, mas aceita caracteres que desconfigurariam a situação de palavra simples do primeiro caso. Por exemplo, pode-se ter o seguinte código: `set {T C L} linguagem`, que, por um motivo qualquer, define uma variável chamada `T C L` com o valor `linguagem`. Para imprimir seu conteúdo é necessário, portanto, utilizar a terceira forma: `puts ${T C L}`.

Uma outra construção sintática existente é a barra invertida. Sua função é possibilitar a inserção dos caracteres que podem ser considerados especiais dependendo do contexto (`"`, `$`, `{`, `[`, `\`) e também de outros não imprimíveis.

Com a distribuição da versão 8.5 da `Tcl`, uma nova regra precisou ser criada e esse parágrafo dedica-se a ela. A razão para isto foi a introdução de um recurso denominado expansão de argumentos. Existe um conflito parcial com essa regra e o que foi apresentado anteriormente, então é necessário considerar algumas exceções que ocorrem por meio da sintaxe para expansão de argumentos. Ao encontrar um elemento com prefixo "`{*}`" com sufixo que não seja caractere em branco, a expansão de argumentos ocorre. O sufixo

é avaliado e substituído seguindo as regras já mencionadas, em seguida outra avaliação ocorre. Como resultado pode-se ter diversos elementos. Para entender esta situação, apresenta-se mais um exemplo:

```
set x {puts abacate}
{*}$x
```

Primeiramente é definido a variável `x` com conteúdo `puts abacate`. Na linha seguinte é feito uma expansão de argumentos, onde primeiramente `$x` é avaliada e substituída pelo conteúdo da variável `x`. Em seguida este mesmo conteúdo é avaliado, causando a execução do comando `puts` com argumento `abacate`.

Finalmente, comentários. A `Tcl` trabalha com comentários que se estendem até o final da linha corrente e utiliza o caractere "#" para essa tarefa. Deve-se tomar cuidado ao criar um comentário, o mesmo só é assim considerado se aparecer no ponto onde o nome de um comando é esperado. Considere o seguinte trecho de código:

```
# Valor aproximado para π
set pi [expr acos(-1)]
```

Nesse caso o comentário está correto. Entretanto, mover o comentário logo em seguida do comando `set`, na mesma linha, causa um erro. A linguagem entenderia que `#`, `Valor`, e os demais elementos devem ser passados para o comando `set` que aceita no máximo três argumentos. Uma outra forma de causar um comportamento (talvez) não esperado é utilizar "#" no lugar do nome de um comando dentro de colchetes: `puts [# oi]`. Apesar do comentário ocorrer num lugar permitido, o colchete que terminaria o comando foi descartado.

3.2 O Ambiente de Execução

A `Tcl` é uma biblioteca em `C` que inclui um *parser* para a linguagem `Tcl`. Para utilizar os recursos ali existentes, uma aplicação primeiramente obtém, por meio de uma chamada, um objeto que representa um interpretador. Este objeto (referência) é a unidade básica de manipulação da `Tcl` (OUSTERHOUT, 1989). A partir dele é possível acessar a máquina virtual da linguagem (denominada de TVM neste trabalho) e proceder para a execução de procedimentos.

A partir da versão 8, a TVM é composta por um compilador que produz *bytecodes* e trabalha com estes utilizando uma pilha. A interpretação desta forma de código ocorre essencialmente através da função `TclExecuteByteCode`, que fica responsável por decodificar todas as instruções e manter um estado consistente do sistema simulado.

Não há preocupação em definir tipos específicos em momento algum. Ou a máquina virtual consegue, por meio de funções da biblioteca, converter um objeto para algum tipo que desejar no momento que estiver executando uma instrução ou um erro é gerado. Isto dificulta a criação de compiladores mais eficientes, tendo sido esta questão discutida nos trabalhos (SAH, 1994) e (LEWIS, 1996). Na implementação atual da linguagem há o conceito de representação dupla para os objetos, sendo uma delas mais eficiente de se trabalhar. Entretanto, ainda não há uma função, ou algo semelhante, que dado um objeto qualquer seja retornado seu tipo. Isso acontece porque dependendo da forma com que se trabalha com um valor, tipos diferentes podem ser assumidos. Pode-se ter, por exemplo, o seguinte código:

```
set x 4
set $x 2
lindex $x 0
```

Isto é, primeiramente a variável `x` tem um valor que se parece com um inteiro mas em seguida este valor é utilizado como um nome para uma variável e, ainda, depois é tratada como uma lista com um elemento: `4`.

O ambiente é dinâmico, comandos podem ser redefinidos/removidos a qualquer momento, código pode precisar ser avaliado em tempo de execução ao fazer uso do comando `eval`, pode-se empregar escopo dinâmico por meio do comando `uplevel` ou também pode-se simular a passagem por nomes ao utilizar o `upvar`. Ou seja, a TVM requer cuidados excessivos para se manter funcionando corretamente. Pode ser necessário recompilar para *bytecodes* um procedimento anteriormente compilado, pois o ambiente pode mudar entre chamadas ou mesmo durante a interpretação de *bytecodes*. De fato, antes de qualquer execução de um procedimento, é feito uma chamada a função interna (não exportada pela biblioteca `Tcl`) `TclProcCompileProc` que determina a necessidade de compilação (primeira chamada) ou recompilação.

Apenas procedimentos criados através do comando `proc` podem vir a ser compilados para *bytecodes*. Toda vez que um comando é criado, a linguagem `Tcl` invoca a função `TclCreateProc` para associar a definição de um procedimento qualquer com uma estrutura

interna `Proc`. Entretanto, somente quando é feito uso do `proc` é que ocorre a associação entre comando e a função interna `TclObjInternProc`. Esta última fica responsável por, indiretamente, chamar a `TclProcCompileProc` e também a `TclExecuteByteCode` para efetivamente interpretar o código do procedimento associado. Em todos os casos, uma tabela *hash* de comandos é atualizada quando comandos são criados e permite que, por meio do nome do comando, a função associada seja acessada pela máquina virtual para proceder à sua execução.

A `Tcl` trabalha num modo misto, com cada comando do usuário sendo interpretado por vez mas também permitindo a execução direta de código `C`. São utilizados 132 *bytecodes* (na versão 8.5.8) para converter código escrito em `Tcl` para a interpretação pela máquina de pilha. Cada comando *built-in* da linguagem fornece uma função para ser chamada durante este processo de compilação, sendo possível aplicar otimizações em casos bastante específicos. O comando `expr`, por exemplo, é capaz de simplificar o código gerado para expressões constantes: `expr {1 + 2 + 3}` será interpretado como `expr {6}`. Para indicar o término de um comando, estas funções utilizadas para esta compilação emitem a instrução `INST_DONE`. Ao encontrar este *bytecode*, a máquina virtual entende que o processo de interpretação daquele comando deve ser encerrado.

Em tempo de execução, a máquina virtual faz acesso a dois objetos de forma mais frequente. Estes objetos são: um *array* de objetos literais e um *array* de variáveis locais (que também inclui os argumentos da função). Estes *arrays* são produzidos durante a geração de *bytecodes* e precisam ser mantidos em um estado consistente ao longo da execução. Cada elemento é formado por uma estrutura `Tcl_Obj`, que é criada para cada objeto no sistema da `Tcl`. Cuidados devem ser tomados para não danificar o conteúdo destes *arrays*, caso contrário uma execução subsequente de um mesmo comando pode produzir resultados incorretos.

Conhecendo o ambiente "hostil" onde o compilador JIT deve ser instalado, as subseções seguintes descrevem como é feito a instalação do JIT neste ambiente e quais condições devem ocorrer para que haja execução de código nativo.

3.2.1 Instalação e execução do compilador JIT

Para possibilitar a execução do compilador dinâmico em momentos específicos durante a execução da máquina virtual alvo, algumas modificações foram realizadas na implementação da `Tcl`. Essas modificações efetivamente "instalam" o compilador JIT no sistema

original.

Como todo comando está associada a uma estrutura interna `Proc`, esta estrutura parece ser o local ideal para conter informações pertinentes a compilação JIT. Isso é devido ao fato de que todos procedimentos criados, que foram escritos em linguagem `Tcl`, conterão uma instância da mesma. Um campo com a estrutura de um `JIT_Proc` foi adicionado a estrutura `Proc`. A Figura 1 descreve o conteúdo desta nova estrutura.

```
struct JIT_Proc {
  int eligible;
  unsigned int callCount;
  unsigned char *ncode;
  int collectingTypes;
  struct JIT_BCType *bytecodeTypes;
};
```

Figura 1: Campos da estrutura JIT_Proc

O campo `collectingTypes`, inicializado em `0`, é utilizado para detectar se já foi alocado memória para o campo `bytecodeTypes` ou não, sendo atualizado para `1` na primeira execução de uma função escrita puramente em `Tcl`. O campo `ncode` é inicializado em `NULL` e atualizado para um endereço quando for gerado código de máquina para o procedimento. O campo `callCount` controla se o procedimento deve ir para compilação dinâmica ou não, sendo decrementado a cada chamada e estabelecendo que o compilador JIT deve ser invocado ao atingir o valor `0`. Atualmente esse campo é inicializado com o valor `1` (definido por `JIT_REQINVOKE`), indicando que o procedimento será interpretado uma vez antes de ser possivelmente compilado e instalado. O primeiro campo, `eligible`, é utilizado para definir se o procedimento é elegível a compilação dinâmica ou não. A elegibilidade é definida da seguinte maneira: se alguma execução do procedimento pelo interpretador retornar um resultado que não seja `TCL_OK`, então o respectivo procedimento é marcado como não elegível. Isso é feito de modo a tentar não gerar código que quase certamente terá que tratar exceções.

Os campos da estrutura `JIT_Proc` requerem manutenção durante a execução da máquina virtual. Porém, o estado dessa máquina pode mudar durante execuções subsequentes de um mesmo procedimento. Se a máquina virtual detectar a necessidade de recompilação, nesse ponto também aproveita-se para reinicializar os campos da estrutura `JIT_Proc` correspondente. A manutenção do campo `bytecodeTypes` é feita por instrumentação da função `TclExecuteByteCode`, trechos de código associados a *bytecodes* relacionados a aritmética e comparação foram incrementados para realizar a coleta de tipos.

3.2.2 Execução do código de máquina

Uma parte da subseção anterior descreveu em que situação o código de máquina para um procedimento específico é gerado e instalado. Agora será apresentado como ocorre a execução do mesmo.

Tomando a estrutura `JIT_Proc` da Figura 1 é possível verificar a existência de informação no campo `ncode`. A indicação de que um procedimento `Tcl` pode ter sua execução realizada por meio de código de máquina direto ao invés de interpretação é a não nulidade de `ncode`. Tendo estabelecido um endereço para esse campo é possível requerer que tal endereço seja entendido como um endereço para uma função. Isto é válido devido ao fato de uma função qualquer ser simplesmente uma sequência de *bytes* que foi gerada por algum compilador para alguma arquitetura. Porém, realizar essa tarefa pode requerer uma certa flexibilidade da linguagem que se deseja utilizar. No caso da linguagem `C`, a escolhida para esse projeto, é permitido realizar um *cast* para resolver esse problema.

Especificamente, o objetivo é tratar o endereço em `ncode` como um ponteiro para uma função que recebe dois parâmetros e tem como retorno um valor inteiro, ou seja: `((int (*)(void *, void *))ncode)(param1, param2)`. A escolha dessa assinatura é devido àquela existente para a função `TclExecuteByteCode`. Essa última função é invocada pela máquina virtual nos casos em que o compilador JIT não gerou código de máquina para o procedimento. Manter a compatibilidade, passando os mesmo parâmetros, proporciona ao código gerado pelo compilador JIT tratar objetos literais, parâmetros do procedimento `Tcl`, variáveis locais e outros detalhes de forma bastante similar ao da implementação da `Tcl`.

3.3 O Sistema de Compilação

Satisfazendo as condições base, descritas na subseção 3.2.1, o compilador dinâmico pode iniciar seu trabalho. O sistema desenvolvido trabalha com compilação de procedimentos por inteiro. A tradução parte dos *bytecodes* existentes, e somente obtém sucesso se todos eles pertencerem ao subconjunto aceito pelo compilador dinâmico atual. Até este momento são aceitos apenas procedimentos folha.

Diversas simplificações foram feitas para tornar factível a criação deste novo compilador. Uma delas envolve trabalhar apenas com instruções que lidam com: manutenção da pilha utilizada pela máquina virtual, desvio condicional e incondicional, relação entre

objetos, aritmética e lógica. Ainda é assumido que os valores poderão ser tratados como inteiros limitados ao tamanho da palavra, esta é a restrição mais severa da implementação atual. Caso um valor realmente não represente um valor inteiro, então o código gerado tem o cuidado de retornar à interpretação. Esse desvio faz com que o procedimento seja interpretado a partir de seu primeiro *bytecode*. A eliminação de *overhead* de interpretação é decorrente das simplificações aplicadas e a remoção do custo de decodificação de instruções é naturalmente obtido pela utilização de código nativo.

Antes de gerar o código nativo, uma outra forma de representação intermediária, além da existente em *bytecodes*, foi criada.

3.3.1 Representação Intermediária

Inicialmente o objetivo era utilizar uma forma de representação intermediária em árvore. E, em uma etapa mais adiante, fazer uso de um algoritmo, entre os diversos existentes (KOES; GOLDSTEIN, 2008; ERTL; CASEY; GREGG, 2006; AHO; GANAPATHI; TJIANG, 1989), para seleção de instruções baseada em árvores. Porém, a partir da leitura de textos diversos(AHO et al., 2006; MUCHNICK, 1997) essa decisão inicial foi repensada de forma a simplificar o projeto. Com isso, ficou definido o uso de uma representação intermediária linear de nível médio na forma de quádruplas. Ainda estava previsto a transformação para a forma SSA (CYTRON et al., 1991) que tem sido considerada por diversos compiladores, como LLVM (LATTNER; ADVE, 2004), GCC (FREE SOFTWARE FOUNDATION, 2010) e Jalapeño (ou Jikes RVM) (ALPERN et al., 2000). Tal representação foi deixada para um projeto futuro, mas vale ressaltar que um dos principais motivos para uso da SSA, de acordo com Cytron et al. (1991), é a possibilidade de executar otimizações clássicas de maneira eficiente. A construção do grafo de fluxo de controle (CFG – *Control Flow Graph*) (MUCHNICK, 1997) de um trecho de código específico faz parte dos requisitos para a transformação em SSA, porém ele também é atualmente utilizado diretamente na geração de código pelo compilador desse trabalho. Com isso foi decidido utilizar quádruplas para compor blocos básicos que representam os nós de um CFG.

A representação em quádruplas é relativamente simples de se construir, além de apresentar uma característica relevante para a otimização que realiza movimentação de código: mover uma quádrupla tende a requerer menor esforço do que rearranjar uma representação em forma de árvore. Entretanto, conforme discutido em Muchnick (1997), também é possível que a representação em árvore apresente vantagens em relação à quádruplas, sendo a eliminação de registradores virtuais temporários e armazenamentos nos mesmos

uma delas.

Na versão atual do compilador a estrutura apresentada na Figura 2 representa uma quádrupla.

```
struct Quadruple {
  Value *dest;
  unsigned char instruction;
  Value *src_a, *src_b;
  struct Quadruple *next;
};
```

Figura 2: Estrutura para uma quádrupla

O campo `instruction` tem tipo `unsigned char` pois atualmente representa um dos *bytecodes* da `Tcl` ou uma das instruções `JIT_INST_SAVE`, `JIT_INST_MOVE`, `JIT_INST_GOTO`, `JIT_INST_JTRUE`, `JIT_INST_JFALSE`, `JIT_INST_INCR` e `JIT_INST_NOP`, que podem ser representadas com um *byte* (todas elas tem um número associado no intervalo [0, 255]). Essas novas instruções foram criadas para eliminar detalhes da máquina de pilha ou para deixar mais claro seu significado. No caso da `JIT_INST_SAVE`, ela apenas substitui o uso do nome da instrução, da `Tcl`, `INST_DONE` mas faz a mesma tarefa: salva um objeto `Tcl_Obj` para representar o resultado da avaliação do procedimento e também se prepara para retornar um código (sucesso ou falha). A `JIT_INST_MOVE` é utilizada para agrupar os *bytecodes* `INST_PUSH1`, `INST_PUSH4`, `INST_LOAD_SCALAR1`, `INST_LOAD_SCALAR4`, `INST_STORE_SCALAR1` e `INST_STORE_SCALAR4` que possuem pouca diferença, na implementação atual, para a geração de código. As novas instruções de desvio eliminam a necessidade de distinção entre o uso de 1 ou 4 *bytes* para indicar o destino do desvio. O comando `incr` da `Tcl` aceita um valor inteiro qualquer como incremento, mas, para poder fazer uso de instruções de máquina mais eficientes, foi criada a instrução `JIT_INST_INCR` que opera somente com os valores 1 e -1. A intrução `JIT_INST_NOP` foi adicionada porque não há nada equivalente nos *bytecodes* da `Tcl`, e é utilizada em casos específicos na construção da representação intermediária descritas mais adiante.

O campo `next`, da Figura 2, existe para lidar de forma eficiente com casos de otimização que requerem a movimentação de código, apesar de otimizações ainda não terem sido implementadas. Os campos de tipo `Value`, inicialmente, permitiam assumir valores inteiros, ponteiros para `Tcl_Obj` ou uma forma de registrador. Na versão atual essa estrutura contém os campos `flags` e `offset`. O primeiro é utilizado para indicar onde o valor deverá ser encontrado em tempo de execução. O segundo indica um deslocamento para o valor caso o mesmo se encontre em um *array*.

A Figura 3 dscreve a estrutura utilizada para agrupar as quádruplas e formas or blocos básicos.

```
struct BasicBlock {
  int id;
  int exitcount;
  struct Quadruple *quads, *lastquad;
  int *exitblocks;
};
```

Figura 3: Estrutura para um bloco básico

Como não há um campo "`next`" nessa abstração, `exitcount` guarda o número de arcos que saem de um bloco de modo a permitir o controle de sua travessia por meio do campo `exitblocks`. O ponteiro `lastquad` é utilizado na construção do CFG, eliminando a necessidade de travessia das quádruplas presentes em um bloco para identificar se a última instrução é de desvio ou não. O campo `id` serve, até o momento, somente para *debugging*. A identificação dos blocos básicos e construção do CFG não difere daquela apresentada em Muchnick (1997), porém não é feito, até o momento, o uso de blocos básicos estendidos.

Para exemplificar a construção de um CFG é considerado o procedimento descrito na Figura 4.

```
proc palindromo {n} {
  set num $n
  set inv 0
  while {$num} {
    set digito [expr {$num % 10}]
    set inv [expr {$inv * 10 + $digito}]
    set num [expr {$num / 10}]
  }
  expr {$n == $inv}
}
```

Figura 4: Procedimento que determina se um número n é considerado palíndromo

Após a execução do procedimento `JIT_Compile` tem-se como resultado o grafo de fluxo de controle. A construção do mesmo ocorre em duas etapas, o resultado da primeira é apresentado na Figura 5.

Para se chegar a esta representação foi feita uma conversão de máquina virtual baseada em pilha para algo semelhante a uma máquina de infinitos registradores. Uma pilha temporária é utilizada para simplificar essa transição, sendo acessada e atualizada durante a construção de diversas instruções. Também são pré-alocados registradores virtuais para as variáveis locais. A implementação da `Tcl` disponibiliza uma lista dinâmica com todas

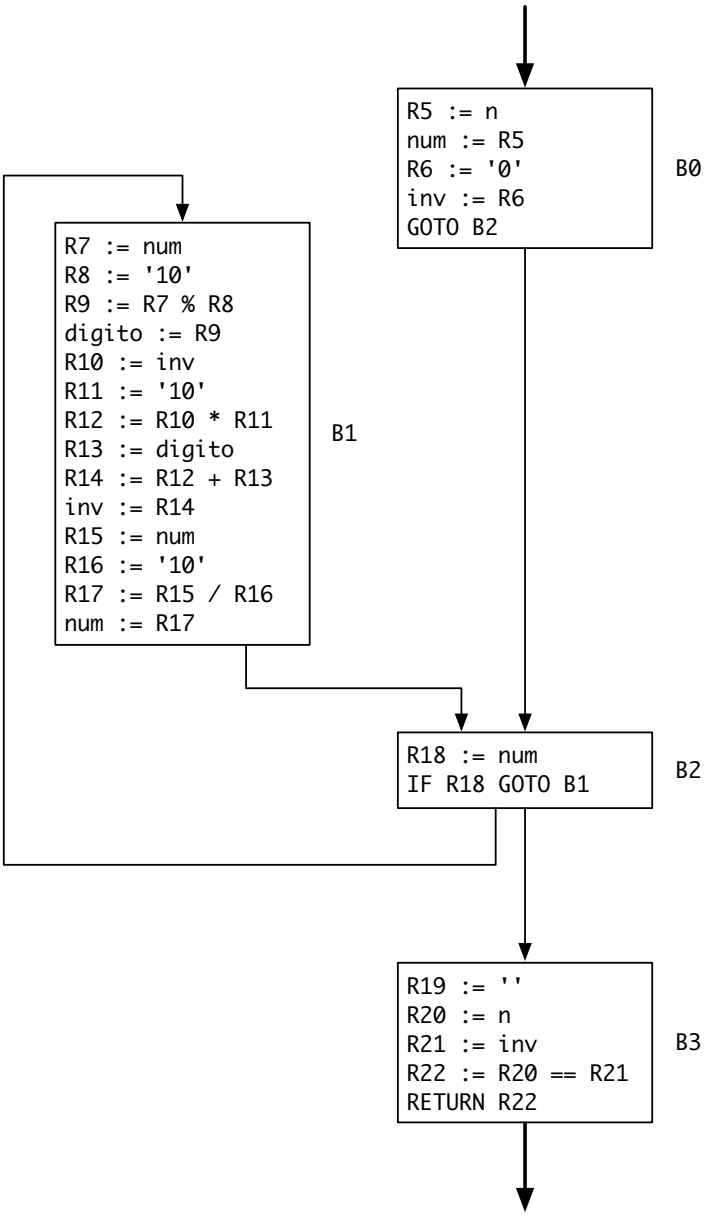

Figura 5: Blocos básicos e grafo de fluxo de controle construídos, etapa 1

as variáveis locais, incluindo também os parâmetros formais da função, sendo necessário apenas atribuir índices a elas. Os registradores que se enquadram nessa lista tiveram seus nomes alterados para simplificar a visualização. O registrador `R1` foi renomeado para `n` na Figura 5, `R2` para `num`, `R3` para `inv` e `R4` para `digito`.

Diversas instruções de atribuição recebem um valor que se parece com número, como `'0'` ou `'10'`, mas que estão sendo representados como *strings*. Na realidade esses valores são ponteiros para uma estrutura `Tcl_Obj`, podendo assumir qualquer tipo interno presente na linguagem. No exemplo anterior estes dados realmente serão convertidos para inteiros. Todos esses ponteiros estarão armazenados em um *array* quando a função

for executada, sendo necessário definir os campos `flags` e `offset` da estrutura `Value` correspondente.

Após construir os blocos básicos iniciais (Figura 5), uma segunda etapa de construção é realizada. Nesta fase, as quádruplas que tiverem operandos `Tcl_Obj` serão movidas para um bloco básico "especial" e, ainda, aquelas que operarem sobre um parâmetro serão também copiadas para esse novo bloco e a quádrupla original será ajustada. Essa etapa foi desenvolvida baseando-se na observação de como os *bytecodes* são interpretados. Na Figura 5, todos os `Tcl_Obj` presentes são oriundos do *array* de objetos literais (Seção 3.2). Carregar estes elementos têm um custo tanto em quantidade de *bytes* produzidos quanto no desempenho do código nativo. No bloco básico 1, pode-se ver que há 3 movimentações de objetos literais (representado por '10'). Nos blocos 0 e 3 há dois acessos envolvendo o parâmetro formal. É mais simples realizar movimentação entre registradores do que acessar o '10' por meio de um *array* que precisará ser percorrido a cada iteração do laço. De modo semelhante, é mais eficiente primeiramente carregar o valor recebido em um parâmetro para um registrador e operar sobre ele do que buscar seu valor atual no *array* de variáveis locais. Com essas transformações, a representação atualizada é exibida na Figura 6.

Na Figura 6, o bloco básico chamado de "especial" tornou-se o bloco inicial. A primeira quádrupla ali contida indica que o parâmetro na posição 0 do *array* de variáveis locais deve ser carregada para um registrador onde n vive. As instruções `NOP` ficam a cargo do gerador de código decidir entre emitir código para cada uma ou não. Essa segunda etapa traz ainda outro benefício. Se um objeto `Tcl_Obj` não puder ser transformado para um número inteiro, então logo no primeiro bloco essa situação é detectada. Isso permite que a volta ao interpretador ocorra de forma mais rápida.

O código da Figura 4 utiliza 107 *bytecodes*, aqui (Figura 6) representado com 32 quádruplas. Apesar das quádruplas estarem em número inferior, o consumo de memória é bastante superior (mas temporário), pois após a geração de código as quádruplas não precisam mais estar em memória. Além disto, algumas são passíveis a eliminação, ficando a cargo desta eliminação etapas de otimização.

Os *bytecodes* de desvio emitidos pela `Tcl` utilizam operandos que descrevem posições relativas (negativas ou positivas) à posição atual, porém, na representação utilizada há somente desvios absolutos para blocos básicos. Para essa tarefa cada *bytecode* é mapeado para cada bloco básico antes da construção do conteúdo dos blocos, assumindo que não existirão mais quádruplas do que *bytecodes*.

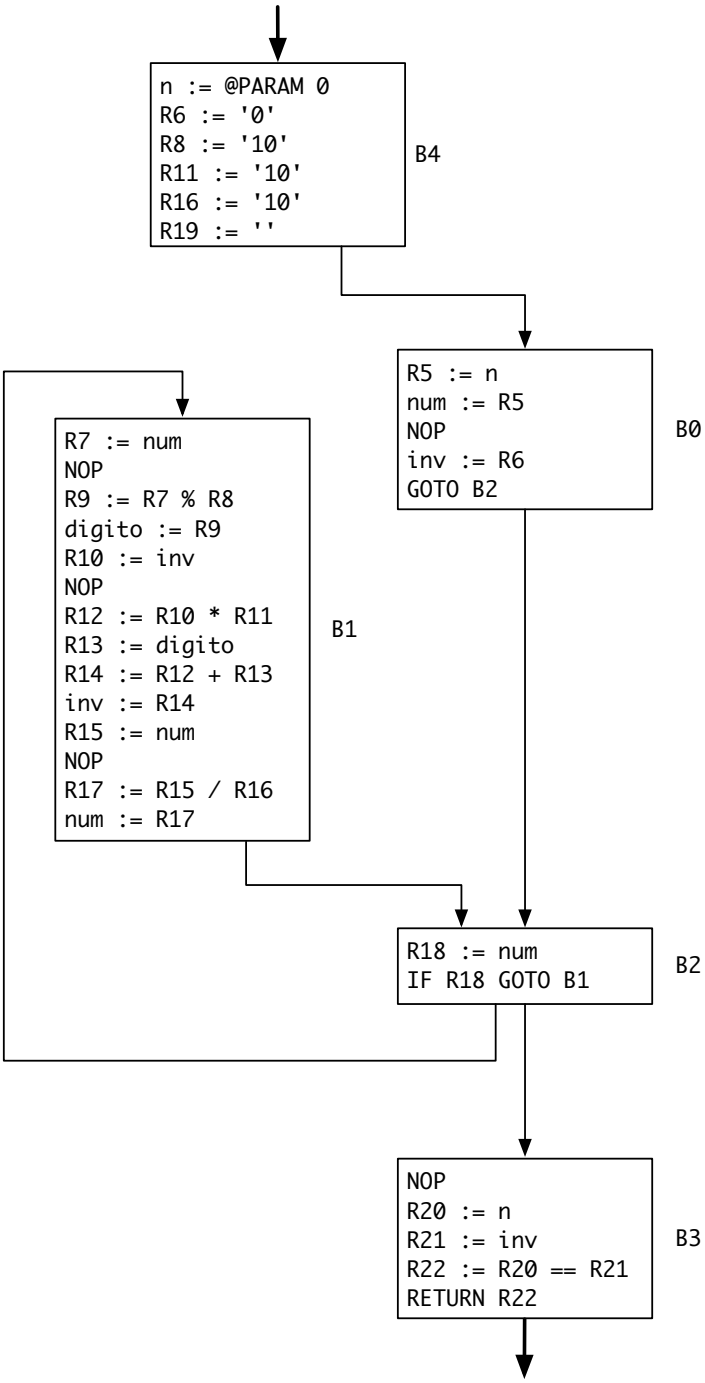

Figura 6: Blocos básicos e grafo de fluxo de controle final

3.3.1.1 Mapeamento de alguns *bytecodes* para quádruplas

A seguir é realizada uma breve análise de como alguns *bytecodes* são convertidos para quádruplas. O exemplo da Figura 7 é tomado como base para a discussão.

A Tabela 1 apresenta, lado a lado, os *bytecodes* produzidos pela `Tcl` e o código intermediário gerado pelo compilador JIT para o procedimento da Figura 7. Para simplificar a

```
proc limiar-bipolar {x} {
  if {$x >= 0} {
    return 1
  } else {
    return -1
  }
}
```

Figura 7: Procedimento exemplo para análise *bytecodes* → quádruplas

visualização dos *bytecodes*, fez-se uso de *labels* que não existem no código produzido pela `Tcl`.

Tabela 1: Saída produzida para procedimento da Figura 7

Bytecode	**Registradores**
`LOAD_SCALAR x`	`R2 := x`
`PUSH '0'`	`R3 := '0'`
`GE`	`R4 := R2 >= R3`
`JUMP_FALSE label1`	`IF NOT R4 GOTO B2`
`START_CMD`	
`PUSH '1'`	`R5 := '1'`
`DONE`	`RETURN R5`
`JUMP label2`	`GOTO B3`
`label1: START_CMD`	`B2:`
`PUSH '-1'`	`R6 := '-1'`
`DONE`	`RETURN R6`
`label2: DONE`	`B3:`

O lado direito da Tabela 1 representa o resultado da etapa 1 da construção de CFG, o resultado final da etapa 2 não auxiliaria no propósito desaa discussão.

A instrução `START_CMD` não possui uma quádrupla equivalente. Isso é resultado da atual simplificação feita. Diferentemente da máquina virtual, o código nativo não verifica se o procedimento sendo executado foi alterado e simplesmente assume que isso não ocorrerá.

A instrução `PUSH` é mapeada utilizando uma pilha temporária e um *array* de objetos literais. Um operando que segue a instrução `PUSH` é tomado como o índice do *array*. O registrador criado neste ponto, `R3` no exemplo, é então empilhado. A instrução `LOAD_SCALAR` é mapeada de forma similar, mas ela se baseia em obter o endereço da variável escalar de

uma lista de variáveis locais criada pelo compilador da `Tcl` para o procedimento corrente. Neste ponto os registradores R2 e R3 estão na pilha e a instrução `GE` deve ser convertida. Típico de uma máquina de pilha, esta instrução não requer que ela seja seguido de operandos pois os mesmos equivalem aos dois últimos empilhados. Da mesma forma, a pilha artificial criada é utilizada para simular esse comportamento. O resultado da comparação é empilhado, mas a representação utilizada pelo compilador dinâmico também requer que tal resultado seja armazenado em um novo registrador.

Para converter a instrução de desvio `JUMP_FALSE`, a pilha utilizada é consultada para descobrir qual registrador contém um valor que determina a realização do desvio ou não. O destino é calculado utilizando o mapeamento mencionado no final da Seção 3.3.1.

A instrução `DONE` é utilizada para indicar um ponto de retorno, sua conversão para o modelo de registradores consiste simplesmente em retornar o último objeto empilhado. Nota-se que a última instrução não teve código produzido, isso é devido a uma heurística aplicada de que uma sequencia de instruções `DONE` equivale a uma única.

3.3.2 Geração de Código de Máquina

Após descrever como a representação intermediária é construída e o funcionamento do sistema para executar o código gerado, o próximo passo é descrever como tal código de máquina é gerado considerando o uso de uma arquitetura CISC (*Complex Instruction Set Computer*) e, em específico, a IA-32 (INTEL, 2009c).

3.3.2.1 Reservando e controlando espaço para os bytes

Um ponto importante é decidir onde armazenar os bytes que serão gerados. De acordo com a Seção 3.2.2, uma função é apenas uma sequência de bytes que pode ser executada. De forma direta é possível pensar em armazenar os bytes em um região de memória alocada por uma função como `malloc`. Porém, simplesmente fazer isso não garante que o sistema operacional permitirá a execução desta sequência de bytes.

Com a introdução do bit NX (*No-eXecute*) pela AMD e depois pela Intel (que renomeou para *eXecute-Disable*), o hardware passou a prevenir a execução de código em páginas destinadas a dados (INTEL, 2009b) e, dessa forma, eliminar parcialmente ataques relacionados a *buffer overflow*[1]. Por essa razão uma região de memória alocada por meio

[1]O artigo "Code Injection Attacks on Harvard-Architecture Devices" de Aurélien Francillon e Claude Castelluccia publicado na ACM CCS 2008 menciona trabalhos que contornam a proteção por bit NX

do `malloc` não poderá ter seu conteúdo executado em sistemas que fazem uso desse recurso. Uma forma de contornar esse problema é utilizar a função `mprotect`, definindo a região com proteção de leitura e execução após escrever os dados desejados. Entretanto, essa função requer que o tamanho da região alocada seja um múltiplo do tamanho de página – denominado de *page-aligned*. Para contornar este outro problema é possível utilizar a função `memalign` ou `valloc`, de acordo com a disponibilidade. Porém, o mais adequado é definir o tamanho da região como um múltiplo do tamanho de página, eliminando a necessidade de alinhamento e reduzindo questões de portabilidade. A estrutura exibida na Figura 8 juntamente com as funçõesexibidas no Apêndice 5.1 tratam dos problemas mencionados e também de questões de portabilidade entre os sistemais operacionais Windows e UNIX.

```
struct MCode {
  unsigned char *mcode, *codeEnd;
  int limit, used;
};
```

Figura 8: Estrutura para controlar uso de memória do código gerado

Os campos `limit` e `used` determinam o tamanho total alocado e o espaço usado até o momento, respectivamente. Atingindo o valor limite, uma nova página precisa ser alocada e o campo `limit` tem seu valor dobrado. Os campos `mcode` e `codeEnd` apontam para o início do código de máquina e o ponto atual na região de memória, respectivamente. O endereço contido em `mcode` é o que será instalado no campo `ncode` da estrutura `JIT_Proc` de um procedimento específico caso a geração de código tenha sucesso.

3.3.2.2 Início e término da geração

Antes de gerar código específico para um CFG, é gerado um prólogo genérico que pode ser visto na Figura 9. Após a geração de código para o CFG é gerado um epílogo que também é apresentado na Figura 9. O código gerado pode lidar com registros de ativação de tamanho variado, por isso é definido um ponto base para acessar parâmetros ou variáveis locais de forma que seja respeitada as condições da ABI (SANTA CRUZ OPERATION, 1997). Essa base, ou ponto de referência, fica definida como o endereço contido no registrador `ebp` após a execução do prólogo. É possível encontrar definições de prólogo que incluam código para salvar os registradores `ebx`, `edi`, `esi` e `esp` uma vez que, de acordo com a ABI seguida, esses registradores (além do `ebp` que já é salvo) precisam ser preservados entre chamadas. Na implementação atual, código para salvar qualquer um desses registradores

além de outras proteções que tem sido desenvolvidas.

precisa ser gerado conforme necessário. O epílogo possui código antônimo daquele contido no prólogo, efetivamente desfazendo o registro de ativação construído e retornando para o endereço contido na topo da pilha corrente.

```
#define PROLOGUE(code, stsize) {  \
  PUSH_REG(code, EBP); \
  MOV_REG_REG(code, ESP, EBP) \
  if (stsize > 0) { \
    if (4 * stsize > 127) { SUB_IMM32_REG(code, 4 * stsize, ESP); } \
    else { SUB_IMM8_REG(code, 4 * stsize, ESP); } \
  } \
}

#define EPILOGUE(code, stsize) { \
  if (stsize > 0) { \
    if (4 * stsize > 127) { ADD_IMM32_REG(code, 4 * stsize, ESP); } \
    else { ADD_IMM8_REG(code, 4 * stsize, ESP); } \
  } \
  LEAVE(code); RETN(code) \
}

#define PUSH_REG(code, reg) *code++ = 0x50 + reg
#define MOV_REG_REG(code, src, dest) \
    *code++ = 0x89;                  \
    *code++ = MODRM(0x3, src, dest)
#define LEAVE(code) *code++ = 0xC9
#define RETN(code) *code++ = 0xC3

#define ADD_IMM8_REG(code, imm, reg) \
    *code++ = 0x83; \
    *code++ = MODRM(0x3, 0, reg); \
    *code++ = imm
#define ADD_IMM32_REG(code, imm, reg) \
    *code++ = 0x81; \
    *code++ = MODRM(0x3, 0, reg); \
    IMM32(code, imm)
#define SUB_IMM8_REG(code, imm, reg) \
    *code++ = 0x83; \
    *code++ = MODRM(0x3, 0x5, reg); \
    *code++ = imm
#define SUB_IMM32_REG(code, imm, reg) \
    *code++ = 0x81; \
    *code++ = MODRM(0x3, 0x5, reg); \
    IMM32(code, imm)

#define MODRM(mod, reg, rm) (mod << 6) + (reg << 3) + rm

#define IMM32(code, v) \
    *code++ = v; *code++ = v >> 8; \
    *code++ = v >> 16; *code++ = v >> 24
```

Figura 9: Código completo para epílogo e prólogo em x86

Para os códigos exibidos adiante assume-se que `code` seja um ponteiro da forma do `codeEnd` da estrutura `MCode` (Figura 8) e disponha de uma região de memória suficiente

para a execução correta dessas macros.

Na Figura 9, o código entre as linhas 1 e 16, é semelhante com código em `Assembly` (BLUM, 2005) de sintaxe AT&T (FREE SOFTWARE FOUNDATION, 2009). A partir da linha 18 é semelhante, porém simplificado, a um montador para x86. Para entender o significado das linhas 18, 20, 21 e 42, a Tabela 2 apresenta uma descrição sucinta das instruções `MOV` e `PUSH`.

Tabela 2: Uma variação das instruções MOV e PUSH

Opcode	Instrução	Operando 1	Operando 2	Modo 64-Bits
0x89 /r	`MOV r32, r/m32`	ModRM:reg (r)	ModRM:r/m (w)	Válido
0x50+rd	`PUSH r32`	reg (r)		N.C.

A coluna "*Opcode*" apresenta os códigos hexadecimais das variações de `MOV` e `PUSH` utilizadas e também as informações "/r" e "+rd". A primeira indica que a instrução faz uso de um byte chamado de ModR/M, que é dividido em três partes: mod (2 bits), reg/opcode (3 bits) e r/m (3 bits), da direita para esquerda seguindo a ordem *little-endian*. O formato final do byte é formado conforme a linha 13 da Figura 9. A parte "reg/opcode" determina ou um número de registrador ou mais três bits de informação de acordo com a especificação do *opcode* (que não é utilizado aqui). O campo "r/m" ou específica um registrador como um operando ou pode ser combinado com o "mod" para codificar um modo de endereçamento. Para mais detalhes sobre o byte ModR/M recomenda-se a consulta ao manual da Intel (2009a, capítulo 2). Além disto, a informação "+rd" indica que um código entre 0 e 7, simbolizando um registrador de 32 bits, deve ser somado ao valor em hexadecimal. A segunda coluna indica que os operandos são registradores de 32 bits (`r32`) ou que pode-se buscar algum conteúdo no endereço de memória calculado (`r/m32`). No caso dessa variação do `MOV` o objetivo é mover dados de um registrador para outro. A última coluna da tabela indica se o *opcode* apresentado pode ser utilizado no modo 64-bits ou não; no caso do `PUSH` está indicado que a sintaxe da instrução não é codificável no modo 64-bits. Ou seja, essa última coluna indica quais das instruções atualmente codificadas pelo compilador teriam que ser no mínimo ajustadas para gerar código para uma arquitetura x64.

A informação contida entre parênteses "r" ou "w", indica que o conteúdo do operando será lido (***r**ead*) ou atualizado (***w**ritten*) pelo processador. Para a instrução `MOV` é possível utilizar 32 combinações de endereçamento, devido ao uso do byte ModR/M. Entretanto, o interesse é apenas na especificação de registradores. De acordo com a Tabela 2.2 do

manual em Intel (2009a, seção 2.1.5) é necessário estabelecer o campo "mod" do byte ModR/M em 3 para essa situação e, assim, juntamente com os números dos registradores fonte e destino é possível completar esse byte adicional. Dessa forma, a linha 9 do código da Figura 9 está resolvida.

3.3.2.3 Percorrendo Blocos e Gerando Código

O gerador de código percorre os blocos básicos do CFG por meio de uma busca em profundidade e emite código para as instruções contidas em cada bloco. Durante o percurso do CFG, cada quádrupla do bloco básico corrente é acessada e um *switch* de instruções define qual cláusula corresponde a instrução contida na quádrupla.

Devido a granularidade alta, imposta pela representação intermediária atual, mesmo instruções simples requerem uma quantidade relativamente alta de *bytes* para codificá-las. Para realizar uma discussão detalhada, faz-se uso de uma única instrução de multiplicação.

O exemplo da Figura 10 gera 2 blocos básicos com um total de 5 quádruplas, o processo de tradução de *bytecodes* é exibido na Tabela 3.

```
proc twice {x} {
  expr {$x * $x}
}
```

Figura 10: Código exemplo para análise da geração de código

Tabela 3: Processo de conversão do código da Figura 10

Bytecode	**CFG, etapa 1**	**CFG, etapa 2**	**Instruções**

`LOAD_SCALAR x`
`LOAD_SCALAR x`
`MULT`
`DONE`
	`BB0:`
	`  R2 := x`
	`  R3 := x`
	`  R4 := R2 * R3`
	`  RETURN R4`
		`BB1:`
		`  R1 := @PARAM 0`
		`BB0:`
		`  R2 := R1`
		`  R3 := R1`
		`  R4 := R2 * R3`
		`  RETURN R4`
			`LOAD_SCALAR src, dest`
			`JIT_INST_MOVE src, dest`
			`JIT_INST_MOVE src, dest`
			`INST_MULT s1, s2, dest`
			`JIT_INST_SAVE src`

A última coluna da Tabela 3 identifica as instruções representantes de cada quádrupla. Os nomes das instruções que não começam com `JIT_` foram reaproveitados da `Tcl`. Outra informação contida diz respeito a quantidade de *bytes* a serem reservados na pilha, definindo o parâmetro `stsize` para o código do prólogo e epílogo da Figura 9. Por tratar-se de um compilador não otimizador, também não é aplicado uma fase de alocação

de registradores. Por isso, cada registrador virtual é mapeado para um endereço na pilha e seus endereços são derivados a partir do seu número exibido nas figuras anteriores.

Ao encontrar uma instrução `LOAD_SCALAR`, o gerador de código sabe que precisará armazenar uma variável local em uma posição da pilha. A Seção 3.2.2 descreveu a assinatura da função criada pelo compilador JIT, sem detalhar os tipos dos argumentos. Para manter a compatibilidade com as estruturas internas do ambiente de execução, o primeiro parâmetro é uma estrutura interna `Interp` que será utilizada para acessar a variável local. A partir desta informação, os *bytes* gerados que imediatamente seguem o prológo gerado são referentes a movimentação do primeiro parâmetro para um registrador. O código na Figura 11 realiza essa primeira função de acordo com a ABI escolhida.

```
#define COPY_PARAM_REG(code, paramn, reg) \
  MOV_DISP8DREG_REG(code, 8 + 4 * paramn, EBP, reg)

#define MOV_DISP8DREG_REG(code, disp, src, dest) \
  *code++ = 0x8B;                                \
  *code++ = MODRM(0x1, dest, src);               \
  *code++ = disp
```

Figura 11: Cópia de parâmetro para registrador seguindo ABI escolhida

Assumindo um uso da forma: `COPY_PARAM_REG(code, 0, EAX)`, será possível encontrar em tempo de execução o endereço de uma estrutura `Interp` no registrador `eax`. Vale ressaltar o uso de outra variação da instrução `MOV` aqui (em complemento àquela da Tabela 2), que é traduzida para `Assembly` como: `movl disp(%src), %dest`. Nesse ponto é possível atravessar a estrutura `Interp` até alcançar o *array* que contém todas as variáveis locais, utilizando trecho de código apresentado na Figura 12.

```
long int offset;
offset = offsetof(Interp, varFramePtr);
MOV_DISP8DREG_REG(code, offset, EAX, EAX);
offset = offsetof(CallFrame, compiledLocals);
MOV_DISP8DREG_REG(code, offset, EAX, EAX);
```

Figura 12: Percorrendo Interp até variáveis locais

A estrutura `Interp` possui o campo `varFramePtr` que aponta para um `CallFrame`. A estrutura `CallFrame`, por sua vez, possui um `array` de tipo `Var`, chamado de `compiledLocals`, que armazena as variáveis locais. Portanto, a Figura 12 gera código equivalente a: `interp->varFramePtr->compiledLocals`, assumindo que `interp` aponte para o parâmetro de tipo `Interp`. Após isso, é utilizado o campo `offset` da estrutura `Value` de `src_a`, deslocando em `compiledLocals` até o ponto desejado e tornando possível acessar o `Tcl_Obj` que representa a variável local procurada (código apresentado na Figura 13,

que é a continuação do código da Figura 12). A variável é um parâmetro para a função escrita em `Tcl`. Além disso, é o primeiro argumento e, por essa razão, o deslocamento será 0 no `array compiledLocals`.

```
#define ADD_IMM8_REG(code, imm, reg) \
  *code++ = 0x83;                    \
  *code++ = MODRM(0x3, 0, reg);      \
  *code++ = imm


if (src_a->offset) {
  ADD_IMM8_REG(code, sizeof(Var) * src_a->offset, EAX);
}
offset = offsetof(Var, value.objPtr);
MOV_DISP8DREG_REG(code, offset, EAX, EAX);
```

Figura 13: Acessando variável local

Após acessar o objeto desejado, é possível obter o valor inteiro contido no mesmo (se houver). Objetos `Tcl_Obj` contém uma estrutura responsável por armazenar o valor inteiro (assim como os demais tipos existentes), mas não há garantias de que o mesmo contenha um valor válido mesmo se o objeto "parecer" ser um número. Por essa razão, é necessário fazer uso da função `Tcl_GetLongFromObj` que verifica se a representação em *bytes* do objeto pode ser convertida para um inteiro e, então, atualiza a sua representação interna mais eficiente com o valor obtido. Sendo assim, e seguindo o modelo simplificado mencionado no ínicio da Seção 3.3, a conclusão da codificação da instrução `LOAD_SCALAR` requer uma chamada a `Tcl_GetLongFromObj`, verificação de erro e armazenamento do valor obtido. A Figura 14 apresenta o código utilizado neste momento.

Na Figura 14, o variável `staddr` refere-se a um endereço na pilha calculado por meio campo `offset` do registrador virtual. O alinhamento da pilha à 16 *bytes* é feito por causa da ABI do sistema operacional Mac OS X (APPLE, 2009), que exige este alinhamento para chamadas de funções mesmo na arquitetura IA-32. Um outro ponto a considerar neste código, é a verificação de código de erro. Esse rótulo `LINTERP` (última linha da Figura 14) aparece logo abaixo do epílogo criado inicialmente (Subseção 3.3.2.2), agindo como um outro epílogo mas que define o conteúdo do registrador `eax` como sendo `JIT_RESULT_INTERPRET`. Quando essa posição do código é atingida, significa que a suposição de que os valores seriam inteiros falhou e o código deve ser interpretado pela TVM.

As duas instruções `JIT_INST_MOVE` que seguem são mais simples de serem codificadas. Cada uma realiza apenas a movimentação do conteúdo de um endereço de pilha para outro endereço de pilha, fazendo uso de duas instruções `MOV`.

```
AND_IMM8_REG(code, -16, ESP); /* Alinhamento */
PUSH_REG(code, EBX);          /* calle-save  */

/* End. mem. para armazenar valor inteiro. */
LEA_DISP8DREG_REG(code, staddr, EBP, EDX);

PUSH_REG(code, EDX);          /* End. armaz. valor */
PUSH_REG(code, EAX);          /* (Tcl_Obj *) */
PUSH_DISP8DREG(code, 8, EBP); /* (Interp *) */

MOV_IMM32_REG(code, (ptrdiff_t)Tcl_GetLongFromObj, EBX);
CALL_ABSOLUTE_REG(code, EBX);

ADD_IMM8_REG(code, 12, ESP);
POP_REG(code, EBX);

/* Verificar por cod. retorno. */
CMP_IMM8_REG(code, 0, EAX);
JUMP_NOTEQ(code, ''LINTERP'');
```

Figura 14: Conclusão da instrução LOAD_SCALAR

A instrução `INST_MULT` carrega o conteúdo de dois endereços de pilha para registradores, aplica a instrução `IMUL` e armazena o resultado na posição de pilha descrita pelo registrador virtual de destino.

Nesse momento o resultado de x^2 foi calculado, mas existe a necessidade de retornar esse valor num formato aceito pelo restante do ambiente. Essa é a tarefa da instrução `JIT_INST_SAVE`. De acordo com o funcionamento da `Tcl`, é necessário definir o objeto resultado da função `twice` para o interpretador em que a função executa. Efetivamente não é feito uso de interpretador aqui, entretanto é necessário armazenar um objeto `Tcl_Obj` em `interp->objResultPtr` pois possívelmente será acessado por outros comandos `Tcl` por meio da função `Tcl_GetObjResult`. Para preencher esse requerimento é feito uso da função `Tcl_NewLongObj` e `Tcl_SetObjResult`. A primeira é utilizada para criar um `Tcl_Obj` com o valor resultante de x^2, a segunda é responsável por armazenar o objeto criado pela primeira no lugar adequado. Somente depois dessas tarefas é possível retornar o código `TCL_OK` (`0`) para indicar sucesso. O código contido nas Figuras 15 e 16 realiza essas tarefas finais.

Após concluir a geração de código, a função `JIT_CodeGen` está pronta para encerrar; ajustando a permissão das páginas utilizadas e retornando o endereço inicial para a sequência de bytes gerada. Para o código da Figura 10 foram gerados 120 *bytes*. O código de máquina para esse trecho é exibido na Tabela 4 na forma hexadecimal, `Assembly` e instruções de nível médio, respectivamente.

```
#define PUSH_DISP8REG(code, disp, reg) \
  *code++ = 0xFF;                      \
  *code++ = MODRM(0x1, 6, reg);        \
  *code++ = disp
#define MOV_IMM32_REG(code, imm32, reg) \
  *code++ = 0xB8 + dest;                \
  IMM32(code, imm32)
#define CALL_ABSOLUTE_REG(code, reg) \
  *code++ = 0xFF;                    \
  *code++ = MODRM(0x3, 2, reg)
#define POP_REG(code, reg) *code++ = 0x58 + reg
#define XOR_REG_REG(code, src, dest) \
  *code++ = 0x33;                    \
  *code++ = MODRM(0x3, src, dest)

#define IMM32(code, v) \
  *code++ = v; *code++ = v >> 8; \
  *code++ = v >> 16; *code++ = v >> 24
```

Figura 15: Macros adicionais requeridos pela Figura 16

```
MOV_DISP8DREG_REG(code,
    INDEX_TO_STACKADDR(VREG_OFFSET(src)), EBP, EDX);

/* Chamar Tcl_SetObjResult. */
PUSH_REG(code, EDX); /* Resultado de x * x. */
MOV_IMM32_REG(code, (ptrdiff_t)Tcl_NewLongObj, EAX);
CALL_ABSOLUTE_REG(code, EAX);
/* EAX recebeu o Tcl_Obj* resultante. */

PUSH_REG(code, EAX);
PUSH_DISP8DREG(code, 8, EXP); /* (Interp *) */
MOV_IMM32_REG(code, (ptrdiff_T)Tcl_SetObjResult, ECX);
CALL_ABSOLUTE_REG(code, ECX);

/* ``Remove'' 8(%ebp) e EAX da pilha. */
ADD_IMM8_REG(code, 4, ESP);

/* Retornar TCL_OK (ver ABI). */
XOR_REG_REG(code, EAX, EAX);
GOTO(code, ``LEAVE'');
```

Figura 16: Tarefas finais para o código da Figura 10

Apesar do procedimento codificado ser bastante simples, é possível reutilizar código para outras instruções. A parte do código que acessa variáveis locais é reaproveitada pelo código que acessa objetos literais, com pequenas modificações. A movimentação entre registradores é utilizada na grande maioria das instruções aceitas, e com pequenas modificações no código da instrução de multiplicação consegue-se realizar outras operações aritméticas.

Tabela 4: Visualização do código gerado para Figura 10

0x55 0x89 0xE5 0x83 0xEC 0x10	0x00002600 <twice+0>: push %ebp 0x00002601 <twice+1>: mov %esp,%ebp 0x00002603 <twice+3>: sub $0x10,%esp	**PRÓLOGO**
0x8B 0x45 0x8 0x8B 0x40 0x64 0x8B 0x40 0x28 0x8B 0x40 0x4 0x83 0xE4 0xF0 0x53 0x8D 0x55 0xFC 0x52 0x50 0xFF 0x75 0x8 0xBB 0xBE 0x15 0x8 0xA 0xFF 0xD3 0x83 0xC4 0xC 0x5B 0x83 0xF8 0x0 0xF 0x85 0x3C 0x0 0x0 0x0	0x00002606 <twice+6>: mov 0x8(%ebp),%eax 0x00002609 <twice+9>: mov 0x64(%eax),%eax 0x0000260c <twice+12>: mov 0x28(%eax),%eax 0x0000260f <twice+15>: mov 0x4(%eax),%eax 0x00002612 <twice+18>: and $0xfffffff0,%esp 0x00002615 <twice+21>: push %ebx 0x00002616 <twice+22>: lea -0x4(%ebp),%edx 0x00002619 <twice+25>: push %edx 0x0000261a <twice+26>: push %eax 0x0000261b <twice+27>: pushl 0x8(%ebp) 0x0000261e <twice+30>: mov $0xa0815ca,%ebx 0x00002623 <twice+35>: call *%ebx 0x00002625 <twice+37>: add $0xc,%esp 0x00002628 <twice+40>: pop %ebx 0x00002629 <twice+41>: cmp $0x0,%eax 0x0000262c <twice+44>: jne 0x266e <twice+110>	**LOAD_SCALAR**
0x8B 0x55 0xFC 0x89 0x55 0xF8	0x00002632 <twice+50>: mov -0x4(%ebp),%edx 0x00002635 <twice+53>: mov %edx,-0x8(%ebp)	**JIT_INST_MOVE**
0x8B 0x55 0xFC 0x89 0x55 0xF4	0x00002638 <twice+56>: mov -0x4(%ebp),%edx 0x0000263b <twice+59>: mov %edx,-0xc(%ebp)	**JIT_INST_MOVE**
0x8B 0x45 0xF8 0x8B 0x55 0xF4 0xF7 0xEA 0x89 0x45 0xF0	0x0000263e <twice+62>: mov -0x8(%ebp),%eax 0x00002641 <twice+65>: mov -0xc(%ebp),%edx 0x00002644 <twice+68>: imul %edx 0x00002646 <twice+70>: mov %eax,-0x10(%ebp)	**INST_MULT**
0x8B 0x55 0xF0 0x52 0xB8 0x13 0x19 0x8 0xA 0xFF 0xD0 0x50 0xFF 0x75 0x8 0xB9 0xE4 0x1 0x9 0xA 0xFF 0xD1 0x83 0xC4 0x8 0x31 0xC0 0xE9 0x0 0x0 0x0 0x0	0x00002649 <twice+73>: mov -0x10(%ebp),%edx 0x0000264c <twice+76>: push %edx 0x0000264d <twice+77>: mov $0xa08191f,%eax 0x00002652 <twice+82>: call *%eax 0x00002654 <twice+84>: push %eax 0x00002655 <twice+85>: pushl 0x8(%ebp) 0x00002658 <twice+88>: mov $0xa0901f0,%ecx 0x0000265d <twice+93>: call *%ecx 0x0000265f <twice+95>: add $0x8,%esp 0x00002662 <twice+98>: xor %eax,%eax 0x00002664 <twice+100>:jmp 0x2669 <twice+105>	**JIT_INST_SAVE**
0x83 0xC4 0x10 0xC9 0xC3	0x00002669 <twice+105>:add $0x10,%esp 0x0000266c <twice+108>:leave 0x0000266d <twice+109>:ret	**EPÍLOGO “LEAVE”**
0xB8 0x0 0x4 0x0 0x0 0x83 0xC4 0x10 0xC9 C3	0x0000266e <twice+110>:mov $0x400,%eax 0x00002673 <twice+115>:add $0x10,%esp 0x00002676 <twice+118>:leave 0x00002677 <twice+119>:ret	**EPÍLOGO “LINTERP”**

4 Avaliação Experimental

4.1 Metodologia

Para avaliar o desempenho do compilador JIT desenvolvido, foi feito uma análise detalhada com seis *benchmarks* simples. Todos os testes foram realizados em um computador com processador Intel Core 2 Duo, modelo E4700; memória principal de 2 GiB de 666 MHz, DDR2 DIMM; *kernel* Linux 2.6.32-25-generic. Foram coletados dados macro acerca de: tempo total de execução (*wallclock*), tempo da compilação JIT, tempo exclusivo de execução de procedimentos com e sem o JIT desenvolvido. Também coletou-se dados detalhados envolvendo a arquitetura do processador utilizado. Em todos os casos fez-se uso da ferramenta PAPI (PAPI, 2010) 4.1.1 e, com exceção do *wallclock*, a implementação da `Tcl` foi instrumentada para coletar os dados específicos. Foram utilizadas duas versões da `Tcl` 8.5.8, a padrão e outra modificada que inclui o compilador JIT, ambas compiladas com gcc `-O2`.

4.2 Benchmarks

A implementação atual do compilador cobre apenas um subconjunto da linguagem `Tcl`, portanto alguns *benchmarks* específicos e rudimentares foram criados para possibilitar uma avaliação inicial. Há uma suíte de testes para a `Tcl`, a Tclbench (TCL CORE TEAM, 2008), porém somente uma quantidade bastante pequena dos testes lá existentes podem ser executados com sucesso neste sistema e, portanto, seu uso não foi considerado.

Em todos os *benchmarks* fez-se uso de um parâmetro n com significado específico para cada teste. Os *benchmarks* utilizados foram:

fact Fatorial iterativo. Calcula n vezes o fatorial dos números de 1 a 12

gcd Máximo divisor comum. Este teste é realizado para todos os elementos do produto cartesiano $I \times J = \{(i, j) \mid i \in \mathbb{N} \wedge j \in \mathbb{N}, i \leq n \wedge j \leq n\}$

gray Calcula o código gray (GRAY, 1953) de um número; quase um *microbenchmark*. O parâmetro n indica o intervalo $[0, n)$ para ter seus respectivos códigos gray gerados.

prime Verifica se um número é primo ou não. Parâmetro n define o intervalo ($[0, n]$) de números a serem verificados.

sum$_1$ Somatório de números naturais no intervalo $[1, i]$, onde i representa o número da iteração atual no intervalo $[1, n]$

sum$_2$ Somatório definido por:

$$\sum_{i=0}^{k} a_i, \text{ onde } a_i = \begin{cases} i & \text{se } i \bmod 4 = 0 \\ -i & \text{se } i \bmod 3 = 0 \\ 0 & \end{cases}$$

com k equivalente a iteração atual no intervalo $[1, n]$

4.3 Desempenho

Para verificar o desempenho a nível macro do compilador JIT, realizou-se primeiramente a coleta do tempo gasto em execuções completas. As Figuras 17, 18, 19, 20, 21 e 22 apresentam os dados obtidos para os seis *benchmarks* distintos. Em todos os casos foram realizadas 100 execuções com tamanho n uniformemente espaçado entre os valores mínimo e máximo utilizados.

Em grande parte das execuções, a linha que indica o tempo com uso do compilador JIT permaneceu abaixo da linha do interpretador padrão da `Tcl`. Isto era esperado em vista das restrições do sistema implementado.

Dentre os testes, **sum$_1$** (Figura 21) apresenta a maior redução (mais de 20 vezes em média). Isto é reflexo do código ali contido, sendo constituído principalmente de deslocamentos entre endereços da pilha e soma de inteiros. O gerador de código atual consegue eliminar boa parte de toda a movimentação de objetos de tipo `Tcl_Obj`, realizando todas as conversões de `Tcl_Obj` para inteiro (se possível) num bloco básico especial dedicado para esta tarefa (Seção 3.3.1). Com isso, é possível trabalhar com instruções de máquina que operam diretamente sobre esses valores inteiros e elimina-se muito do *overhead* existente na máquina virtual `Tcl`. O teste **sum$_2$** (Figura 22) também apresentou um ganho significativo, apesar de fazer uso de divisão por constante com uso da `IDIV`. Esta é a forma mais simples e geral de realizar a divisão, mas os compiladores otimizadores a

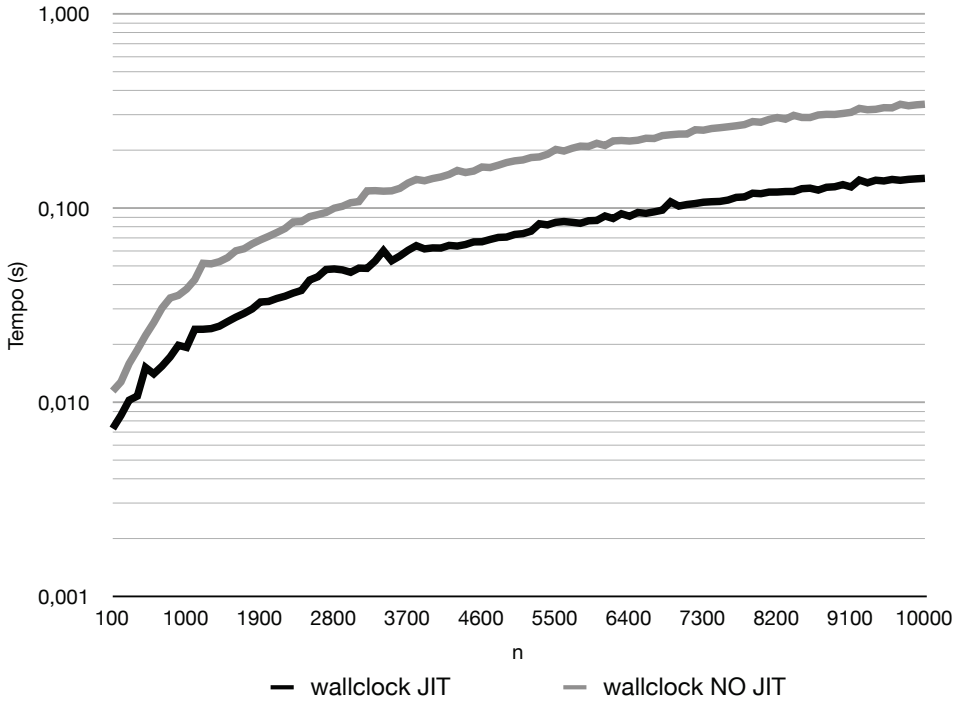

Figura 17: Tempo total de execução para fact

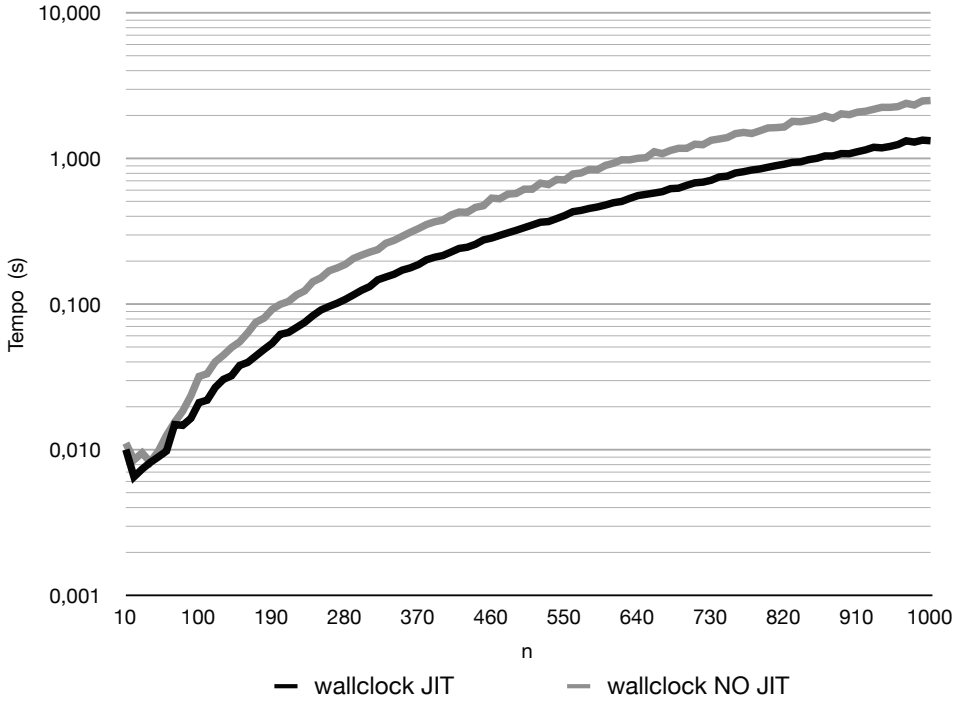

Figura 18: Tempo total de execução para gcd

evitam, quando possível, (GRANLUND; MONTGOMERY, 1994) por consumir muitos ciclos.

Os *benchmarks* **fact** (Figura 17), **gcd** (Figura 18) e **gray** (Figura 19) não apresentam resultados tão expressivos. No caso do **gray**, tem-se que ele é o teste que requer a menor quantidade de *bytes* para sua codificação e talvez procedimentos muito pequenos não sejam

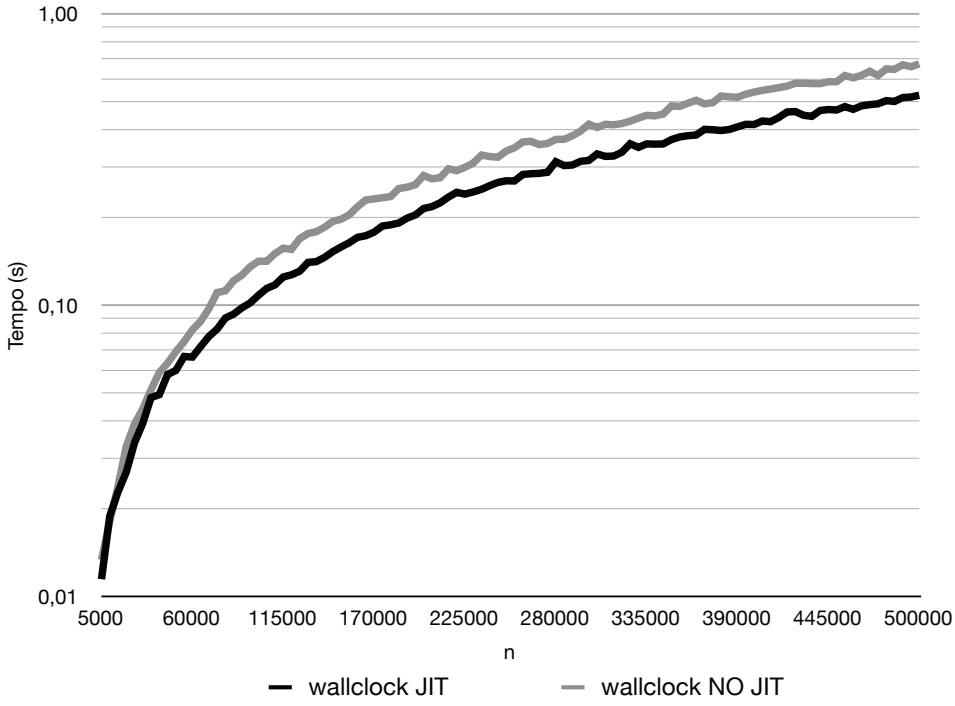

Figura 19: Tempo total de execução para gray

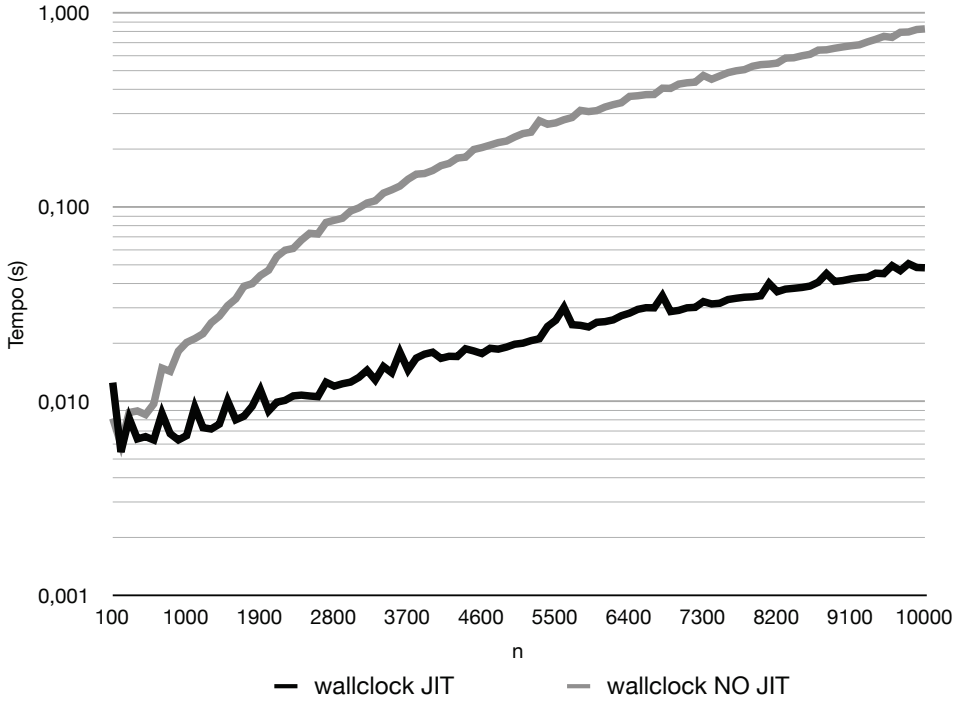

Figura 20: Tempo total de execução para prime

altamente beneficiados pelo compilador atual. Os outros dois têm tamanho próximo do $\mathbf{sum_1}$, mas fazem uso de instruções de multiplicação ou divisão.

Nota-se que **prime** (Figura 20), e $\mathbf{sum_2}$ apresentaram um comportamento diferenciado nas primeiras execuções. Apesar de n aumentar, houve um decréscimo de tempo entre

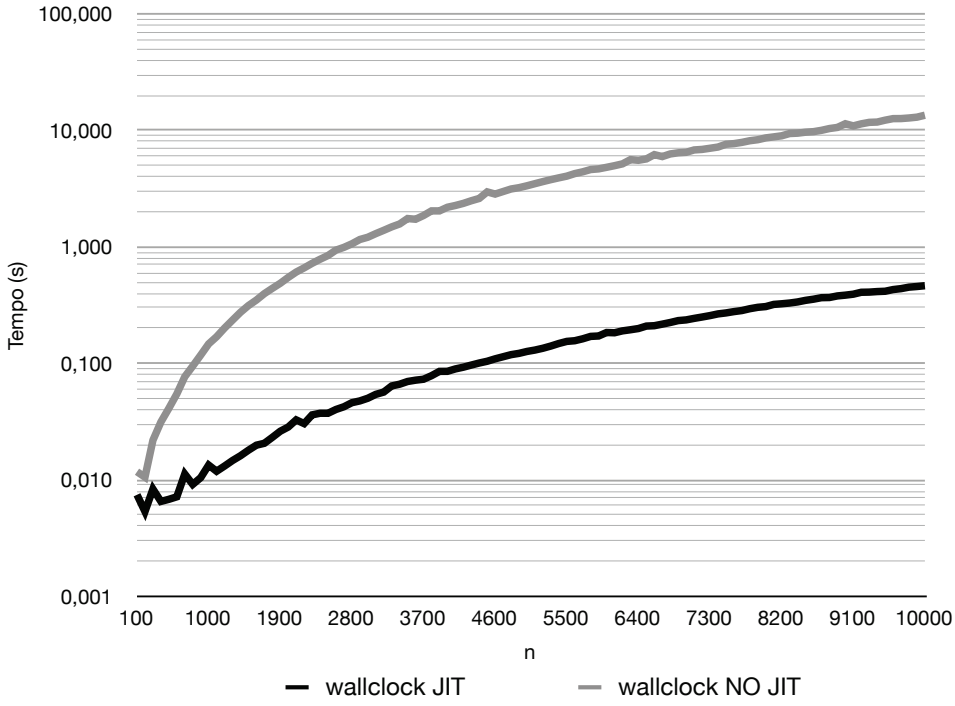

Figura 21: Tempo total de execução para sum_1

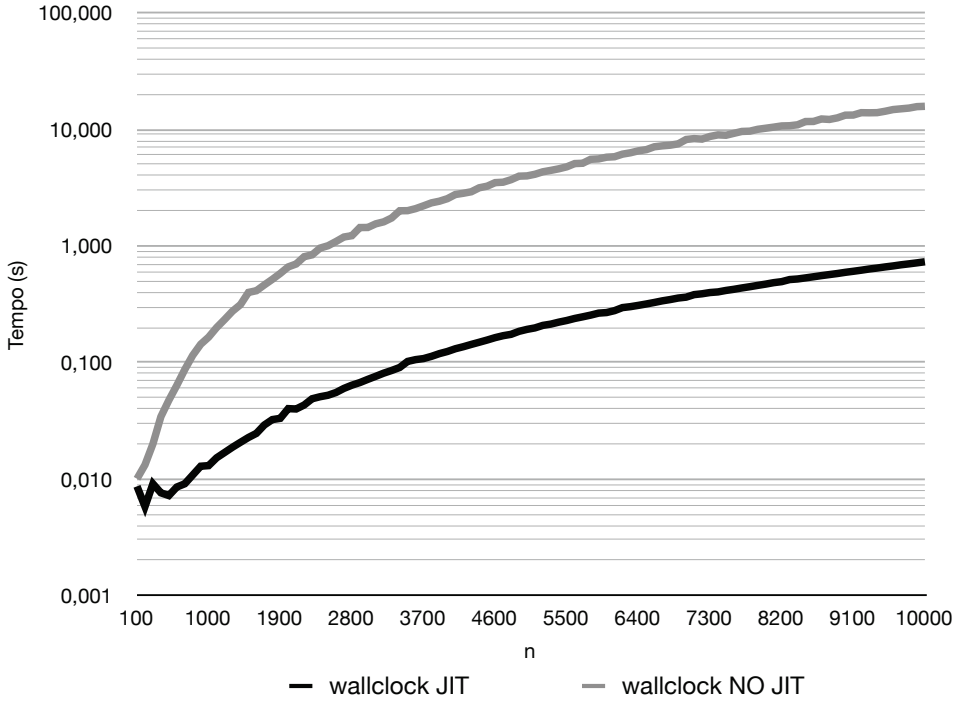

Figura 22: Tempo total de execução para sum_2

a primeira e a segunda execução nestes casos ao passo em que isto não ocorre quando o código é interpretado. Coincidentemente **prime** e $\mathbf{sum_2}$ são os que mais requerem do compilador dinâmico, sendo necessário emitir, na etapa final, 725 e 609 *bytes*, respectivamente, e, portanto, requerem um tempo maior na compilação antes da primeira execução nativa. Entretanto, deve-se notar que os tempos de execução em questão são baixos e

que pequenas modificações no ambiente do sistema operacional podem causar pequenas flutuações e anomalias nos resultados. Com estes dados, ainda não é possível concluir que, de fato, o tempo de compilação influenciou nesse fenômeno observado.

Para facilitar a visualização da diferença de desempenho, destaca-se a Figura 23. O *microbenchmark* apresenta o menor ganho (cerca de 25%) mas pode-se verificar que o tempo gasto pelo laço que executa o teste domina o tempo total. Somente para executar `for {set i 0} {$i < $n} {incr i} { .. }`, no maior teste, gasta-se cerca de 0,45 segundos, deixando menos de 0,07 segundos para efetivamente executar o `gray`.

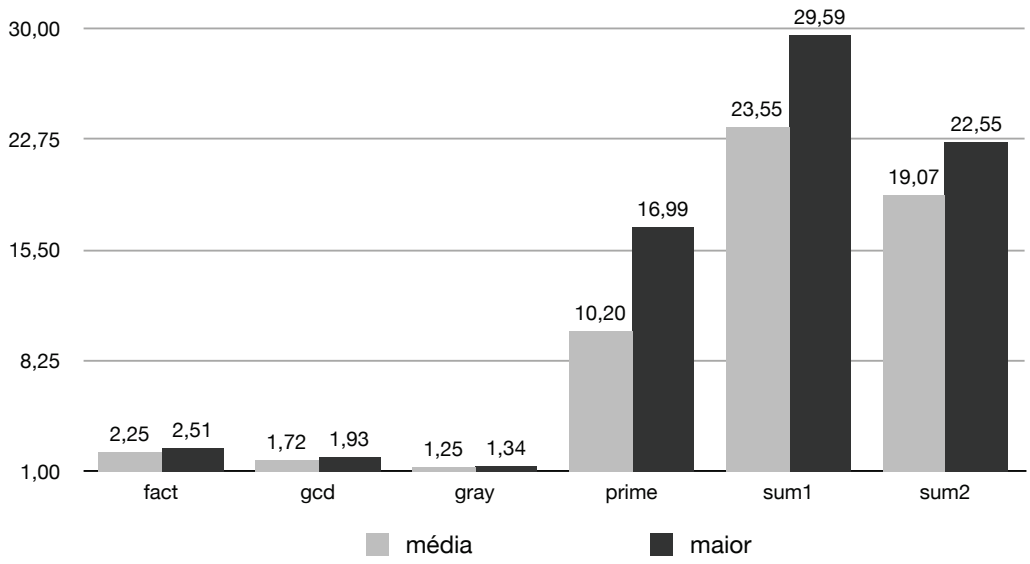

Figura 23: Melhoria em relação ao tempo do interpretador padrão (vezes)

Apesar de apresentar resultados positivos, os dados exibidos até agora estão "contaminados" com *overhead* herdado da máquina virtual `Tcl`. Para determinar quanto tempo exatamente é consumido pelo compilador dinâmico, mais dois testes foram realizados.

Com a Figura 24 verifica-se que o tempo de compilação tem influência quase nula no tempo total, custando entre 49×10^{-6} e 79×10^{-6} segundos. Também nota-se que o crescimento do tempo de compilação não acompanha o crescimento dos *bytes* na mesma proporção. A linha de tendência 1, para os *bytes*, é descrita por $y = 10,143x^2 + 31,771x + 168.8$ e apresenta um índice de correlação com os dados de 0,9379. Enquanto isso, a linha de tendência 2 é dada por $y = 1,3571x^2 - 3,9x + 52,4$ com índice de correlação de 0,9876, onde $1 \leq x \leq 6$. A tendência é que o tempo de compilação continue baixo.

A Tabela 5 apresenta os resultados do último teste desta seção. Para chegar-se a estes resultados, utilizou-se dados dos resultados anteriores para n no maior caso e depois n foi fixado de acordo com o valor máximo utilizado nos testes específicos de *wallclock* e,

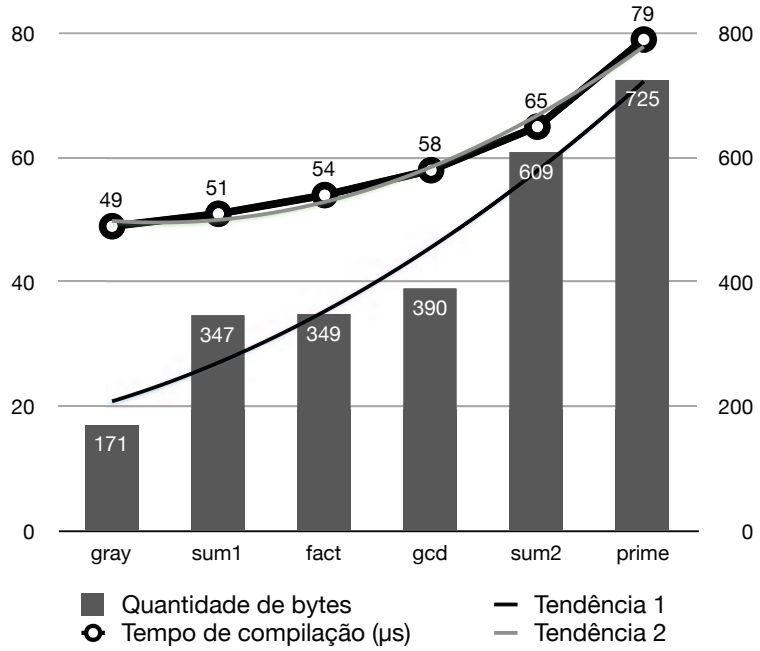

Figura 24: Quantidade de bytes x86 e tempo de compilação

então, foram feitas 10 execuções em cada situação nova e tomada a média do tempo.

Tabela 5: Tempo de execução dos *benchmarks* no maior caso

	JIT (s)		NO JIT (s)		
Benchmark	Total	Exclusivo	Total	Exclusivo	Melhoria (vezes)
fact	0,14	0,04	0,34	0,23	5,75
gcd	1,32	0,37	2,49	1,55	4,18
gray	0,52	0,13	0,67	0,27	2,07
prime	0,05	0,04	0,82	0,80	20,00
sum_1	0,46	0,44	13,48	13,10	29,72
sum_2	0,73	0,71	15,81	15,05	21,19

A última coluna da Tabela 5 refere-se a melhoria na nova situação: tempo exclusivo com JIT e sem JIT. Em relação aos tempos observados, nota-se que os *benchmarks* que fazem uso de dois níveis de laços (**gcd** e **fact**), que executam sem serem compilados, e também o **gray** que teve n definido em 500.000 para o maior caso, são os que apresentam as maiores reduções quando considerado seus respectivos tempos exclusivos. Isso ocorre independente de utilizar o JIT ou não, pois em ambas situações esse custo é removido. Entretanto, ao realizar $\sum_{i=1}^{6} \left(\frac{exclusivo}{total}\frac{1}{6}\right)$ com os dados da Tabela 5 para o caso com JIT e o caso sem JIT, verifica-se que os valores obtidos são, respectivamente, $0,59$ e $0,77$. Isso indica que, além do tempo exclusivo com JIT ser relativamente mais rápido que sem JIT, ao mesmo tempo que o restante do sistema de compilação dinâmica impacta no tempo

total também consegue-se recuperar este tempo por meio da execução não interpretada.

4.4 Análise Detalhada

De forma a verificar a qualidade do código gerado e também o motivo do mesmo ter maior desempenho que aquele gerado por um compilador otimizador para o interpretador, fez-se uso dos recursos disponíveis para monitoramento de desempenho em *hardware*. A ferramenta PAPI possibilitou o acesso a estas informações de mais baixo nível. Dentre os eventos monitorados pelo *hardware* utilizado, aqueles descritos na Tabela 6 foram analisados.

Tabela 6: Eventos coletados com uso da PAPI

Evento	Significado
L1_DCM	Falhas para cache de dados L1
L1_ICM	Falhas para cache de instruções L1
L2_DCM	Falhas para cache de dados L2
L2_ICM	Falhas para cache de instruções L2
HW_INT	Interrupções de *hardware*
BR_MSP	Desvios condicionais erroneamente previstos
BR_PRC	Desvios condicionais corretamente previstos
TOT_INS	Total de instruções completadas
TOT_CYC	Total de ciclos utilizados
RES_STL	Ciclos suspensos (*stalled*) por qualquer recurso
L1_DCA	Acessos ao cache de dados L1
L1_ICA	Acessos ao cache de instruções L1
L2_DCA	Acessos ao cache de dados L2
L2_ICA	Acessos ao cache de instruções L2

Os resultados dividem-se entre as Tabelas 7 e 8. Somente os *benchmarks* **fact**, **gray** e $\mathbf{sum_1}$ participaram nesta coleta. Tanto o **gcd** quanto o **fact** obtiveram um aumento de cerca de duas vezes quando considerado seu tempo exclusivo (Tabela 5) e, portanto, escolheu-se aleatoriamente o **fact** para representar essa classe de resultados. Os outros dois, **gray** e $\mathbf{sum_1}$, apresentaram-se como o pior e melhor em aumento de desempenho e, por isso, foram escolhidos. Os dados apresentados resultam da média de quatro execuções considerando exclusivamente a execução do procedimento que implementa o *benchmark*. Além disso, n foi fixado em 12 para **fact**, 1000 para **gray** e 10000 para $\mathbf{sum_1}$ e foi feito uso de uma única chamada ao procedimento do *benchmark* em cada execução.

Na Tabela 7, a coluna HW_INT demonstra que *benchmarks* que necessitam de menos instruções (coluna TOT_INS da Tabela 8) não registram interrupções de *hardware*. Mais

Tabela 7: Resultados dos eventos monitorados com PAPI (parte 1)

		L1_DCM	L1_ICM	L2_DCM	L2_ICM	HW_INT	BR_MSP	BR_PRC
JIT	fact	25,50	25,25	53,25	16,00	0,00	22,00	35,50
	gray	24,00	25,00	62,00	18,50	0,00	13,00	23,00
	sum_1	27,00	24,00	43,25	14,00	0,25	15,75	10.028,25
NO JIT	fact	99,75	137,25	405,25	133,00	0,00	171,75	1.707,50
	gray	61,50	79,75	233,75	74,50	0,00	34,50	89,50
	sum_1	125,75	189,00	489,50	156,25	1,50	46.801,75	1.498.345,50

Tabela 8: Resultados dos eventos monitorados com PAPI (parte 2)

		TOT_INS	TOT_CYC	RES_STL	L1_DCA	L1_ICA	L2_DCA	L2_ICA
JIT	fact	834,00	10.666,75	3.256,75	773,00	1.579,00	86,50	51,00
	gray	379,00	7.763,25	1.816,50	307,00	925,50	86,75	43,00
	sum_1	310.492,25	235.629,00	154.379,50	331.987,50	230.957,00	99,00	58,25
NO JIT	fact	12.553,00	48.928,00	8.507,50	8.522,00	13.073,50	534,75	295,00
	gray	975,00	26.577,00	4.743,25	721,25	2.312,25	344,75	192,25
	sum_1	10.059.777,00	6.134.075,75	479.867,75	6.229.973,25	5.507.760,25	953,00	608,75

execuções destes *benchmarks* demonstraram resultados consistentes com aqueles da coluna HW_INT, com o $\mathbf{sum_1}$ requerendo cerca de 1 interrupção a cada 4 execuções enquanto na configuração JIT e 1,5 interrupção a cada execução quando interpretado.

Para avaliar o caso de acesso ao cache L1, os eventos L1_DCM, L1_ICM, L1_DCA, L1_ICA foram agrupados $((L1\_DCA + L1\_ICA) - (L1\_DCM + L1\_ICM))$ de forma a produzir a Figura 25. O mesmo foi feito com os respectivos eventos para o cache L2 e o resultado é apresentado na Figura 26.

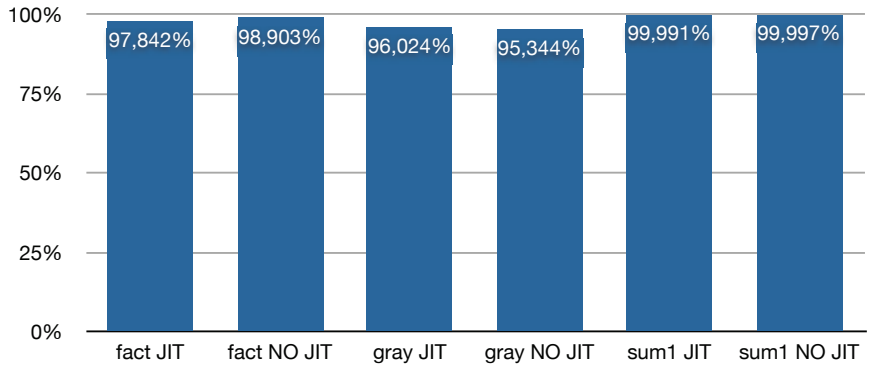

Figura 25: Taxa de acerto ao cache L1

Somente o *benchmark* **gray** demonstra uma pequena vantagem na taxa de acerto ao cache L1, sendo que no geral essa taxa foi penalizada com o uso do compilador dinâmico. Para o cache L2 a situação é inversa, apenas o **gray** teve esta taxa de acerto reduzida enquanto o **fact** e $\mathbf{sum_1}$ aumentaram a mesma em cerca de 14% e 5%, respectivamente.

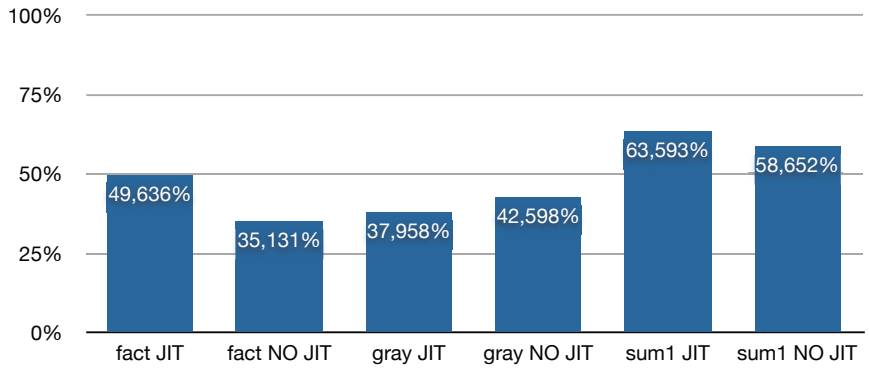

Figura 26: Taxa de acerto ao cache L2

Entretanto, por meio das duas últimas colunas da Tabela 8, verifica-se que o acesso ao cache L2 foi baixo no geral e, assim, essa taxa de acerto maior pouco influência no custo de execução.

Em relação a desvios incondicionais (Figura 27), apenas o código gerado para o *benchmark* $\mathbf{sum_1}$ apresentou melhorias em comparação ao interpretador. Nos demais casos, houve uma taxa maior de predições errôneas ao utilizar o código emitido pelo compilador dinâmico. Este resultado negativo pode estar ligado ao fato dos outros testes terem um número de $BR\_PRC + BR\_MSP$ baixo quando comparado ao do $\mathbf{sum_1}$, talvez afetando preditores de desvio existentes no processador utilizado. De qualquer modo, as execuções sem uso do JIT podem estar indicando um possível ponto de melhoria no interpretador da linguagem `Tcl`: forma de *dispatch* de instruções da máquina virtual. Na implementação atual da `Tcl`, é feito uso do método do "`switch` gigante" (ERTL; GREGG, 2001) que envolve desvios e custa cerca de três vezes mais que a técnica *direct threading* (ERTL; GREGG, 2001).

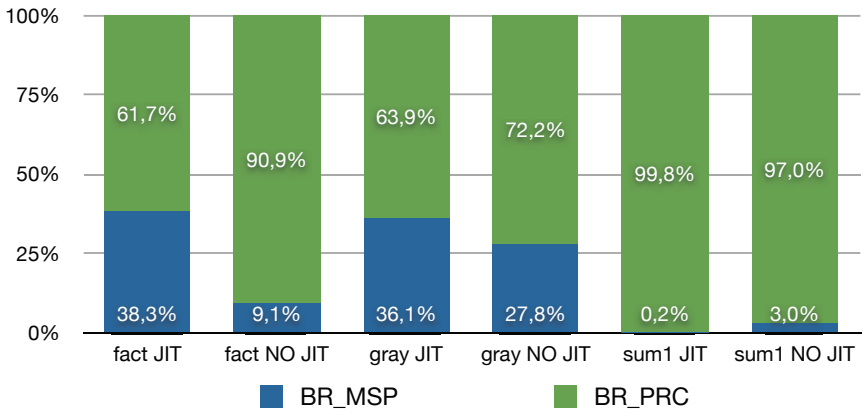

Figura 27: Predição em desvios condicionais

Os dados obtidos para o evento RES_STL (Figura 28) continuam demonstrando que

o código emitido pelo compilador dinâmico é, em geral, pior que aquele emitido pelo gcc. Proporcionalmente, perde-se mais ciclos por conflito de recursos com o código do compilador JIT. No pior caso, 66% dos ciclos do **sum$_1$** foram "desperdiçados" com JIT ao passo em que registrou-se apenas 8% para o mesmo evento na situação sem o JIT.

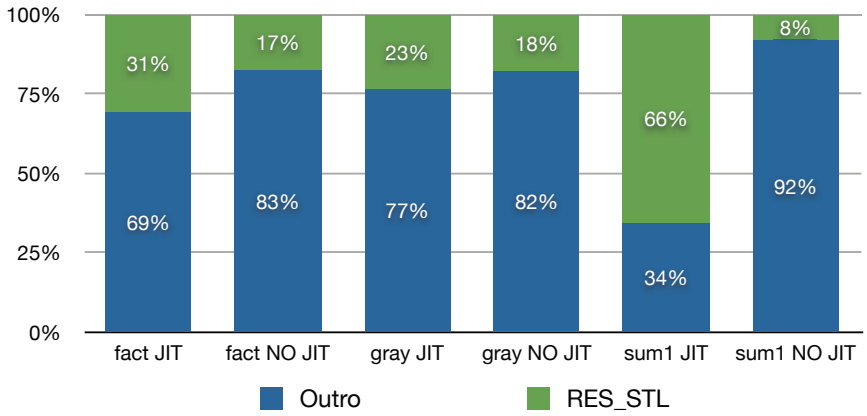

Figura 28: Taxa de ciclos suspensos

No geral, verificou-se que o código do compilador dinâmico foi inferior. Isso era esperado por não se tratar de um compilador otimizador. Assim, a vantagem do compilador JIT são as simplificações realizadas durante a emissão de código. A Figura 29 apresenta a quantidade de instruções executadas para o código do compilador JIT em relação àquela do interpretador.

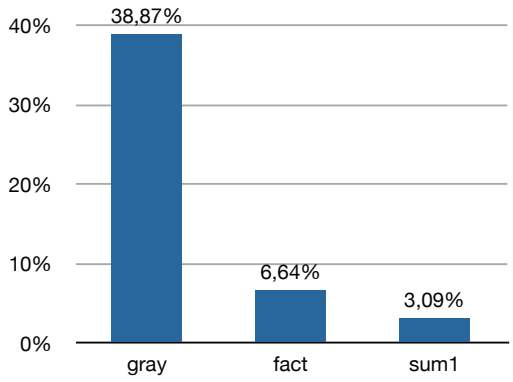

Figura 29: Quantidade de instruções executadas em relação ao interpretador

Com a Figura 29 fica confirmado que, considerando as análises anteriores, a redução do tempo de execução, no caso deste compilador JIT não-otimizador, está diretamente ligada a diminuição de instruções necessárias para executar um mesmo código.

5 Conclusões

Um compilador não-otimizador JIT de modo misto para um subconjunto da linguagem `Tcl` foi desenvolvido e validou-se a melhoria de desempenho por meio da compilação dinâmica. Foi demonstrado um aumento de desempenho de mais de 29 vezes no melhor caso. Também verificou-se que o tempo de compilação foi baixo, levando cerca de 79×10^{-6} segundos para emitir 725 *bytes*. Identificou-se que a redução na quantidade de instruções de máquina executadas, em comparação ao interpretador, foi a causa do ganho de tempo.

Entretanto, mesmo para um pequeno conjunto de instruções suportadas, a construção do compilador JIT não foi uma tarefa simples. A geração de código de máquina, em especial para uma arquitetura CISC, de forma manual é dispendiosa, propensa a erros e de difícil depuração. Devido a alta granularidade imposta pela representação intermediária atual, mostrou-se necessidade de uma quantidade de 120 *bytes* para a codificação simplificada de um pequeno procedimento em `Tcl` que requer apenas 5 quádruplas. Logo, procedimentos maiores requerem uma quantidade muito maior de *bytes* e a verificação da corretude do código gerado é trabalhosa.

Ainda resta descobrir uma forma de coletar tipos de forma eficiente quando considerada a linguagem `Tcl`. Mesmo com informações existentes em tempo de execução, a linguagem não tem um modelo bem definido de identificação de tipos. Isso é resultado do conceito fundamental que cerca a linguagem: tudo é *string*. Sem realizar esta tarefa, a tarefa de geração de código de forma eficiente parece não ser possível. O trabalho presente simplifica esta situação, tenta-se trabalhar com números inteiros mas, nos casos em que não for possível, também utiliza-se o interpretador.

A vantagem de desempenho ao empregar um compilador JIT é clara. Por esta razão, a técnica tem feito parte das implementações de máquinas virtuais que demandam alta performance. Entre as linguagens de programação, `Java` destaca-se ao ter recebido suficiente atenção ao ponto de empresas e pesquisadores desenvolverem diversas JVMs com compiladores dinâmicos que fazem uso de uma variedade de técnicas.

5.1 Trabalhos futuros

O trabalho desenvolvido pode ser considerado como um compilador *baseline* para um conjunto da `Tcl`. Desse modo, ainda falta explorar a utilização de um compilador otimizador. A conversão de *bytecodes* da máquina de pilha para a representação na forma de máquina de registradores trouxe as ineficiências em relação a carregamento e armazenamento de dados. Logo, este é um primeiro ponto a ser tratado. Além disso, a alocação de registradores é inexistente e simplesmente atribui-se posições na pilha para cada registrador virtual. O compilador otimizador poderia trabalhar sobre estes problemas.

Em relação a representação intermediária, ainda é necessário pelo menos outra de mais baixo nível. A representação atual está muito longe do código final, dificultando o processo de alocação de registradores e seleção de instruções de forma eficiente. Em um primeiro momento pretendeu-se já fazer uma representação de baixo nível, mas a conversão de *bytecodes* `Tcl` para esta de nível médio demonstrou-se simples e conseguiu-se eliminar detalhes da máquina de pilha. Logo, essa nova representação seria mais bem-vinda se acoplada a atual.

Aceitando apenas procedimentos folha, um recurso fundamental da linguagem `Tcl` fica excluído. Por ser uma linguagem de comandos, não suportar a realização de chamadas é uma grande restrição. O trabalho a ser desenvolvido nesta direção precisa se preocupar com os detalhes de interação entre: (1) máquina virtual e código nativo; (2) código nativo e código nativo.

Referências

AHO, A. V.; GANAPATHI, M.; TJIANG, S. W. K. Code generation using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, ACM, New York, NY, USA, v. 11, p. 491–516, 1989.

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison Wesley, 2006.

ALETAN, S. O. An overview of RISC architecture. In: *Proceedings of the Symposium on Applied computing: Technological Challenges of the 1990's*. New York, NY, USA: ACM, 1992. p. 11–20.

ALLEN, R.; KENNEDY, K. *Optimizing Compilers for Modern Architectures*. San Francisco, CA, USA: Morgan Kaufmann, 2001.

ALPERN, B.; ATTANASIO, C. R.; BARTON, J. J.; BURKE, M. G.; CHENG, P.; CHOI, J.-D.; COCCHI, A.; FINK, S. J.; GROVE, D.; HIND, M.; HUMMEL, S. F.; LIEBER, D.; LITVINOV, V.; MERGEN, M. F.; NGO, T.; RUSSELL, J. R.; SARKAR, V.; SERRANO, M. J.; SHEPHERD, J. C.; SMITH, S. E.; SREEDHAR, V. C.; SRINIVASAN, H.; WHALEY, J. The Jalapeño Virtual Machine. *IBM Systems Journal*, IBM Corp., Riverton, NJ, USA, v. 39, n. 1, p. 211–238, 2000.

ALPERN, B.; AUGART, S.; BLACKBURN, S. M.; BUTRICO, M.; COCCHI, A.; CHENG, P.; DOLBY, J.; FINK, S.; GROVE, D.; HIND, M.; MCKINLEY, K. S.; MERGEN, M.; MOSS, J. E. B.; NGO, T.; SARKAR, V. The Jikes Research Virtual Machine project: building an open-source research community. *IBM Systems Journal*, IBM Corporation, Riverton, NJ, USA, v. 44, p. 399–417, January 2005.

ALPERN, B.; BUTRICO, M.; COCCHI, A.; DOLBY, J.; FINK, S.; GROVE, D.; NGO, T. Experiences Porting the Jikes RVM to Linux/IA32. In: *Proceedings of the Virtual Machine Research and Technology Symposium*. Berkeley, CA, USA: USENIX Association, 2002. p. 51–64.

APPLE. *Mac OS X ABI Function Call Guide*. 2009. `http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/LowLevelABI/Mac_OS_X_ABI_Function_Calls.pdf`. Acessado em 24 de setembro de 2010.

BLUM, R. *Professional Assembly Language*. Birmingham, UK: Wrox, 2005.

BODÍK, R.; GUPTA, R.; SARKAR, V. ABCD: Eliminating Array Bounds Checks on Demand. In: *Proceedings of the Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2000. p. 321–333.

BRUENING, D.; DUESTERWALD, E. Exploring Optimal Compilation Unit Shapes for and Embedded Just-In-Time Compiler. In: *Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization*. New York, NY, USA: ACM, 2000.

CALLAHAN, D.; COOPER, K. D.; KENNEDY, K.; TORCZON, L. Interprocedural Constant Propagation. In: *Proceedings of the Symposium on Compiler Construction*. New York, NY, USA: ACM, 1986. p. 152–161.

CIERNIAK, M.; LUEH, G.-Y.; STICHNOTH, J. M. Practicing JUDO: Java under Dynamic Optimizations. *SIGPLAN Notices*, ACM, New York, NY, USA, v. 35, n. 5, p. 13–26, 2000.

CISCO. *Cisco Systems, Inc.* 2010. `http://www.cisco.com/`. Acessado em 5 de novembro de 2010.

CYTRON, R.; FERRANTE, J.; ROSEN, B. K.; WEGMAN, M. N. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, v. 13, n. 4, p. 451 – 490, 1991.

DEC. *Alpha Architecture Handbook – Version 3*. Maynard, MA, USA, 1996.

DEUTSCH, L. P.; SCHIFFMAN, A. M. Efficient Implementation of the Smalltalk-80 System. In: *Proceedings of the Symposium on Principles of Programming Languages*. Salt Lake City, Utah: ACM, 1984. p. 297–302.

DOOLEY, K.; BROWN, I. *Cisco IOS Cookbook*. Sebastopol, CA, USA: O'Reilly Media, 2006.

ERTL, M. A.; CASEY, K.; GREGG, D. Fast and flexible instruction selection with on-demand tree-parsing automata. In: *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2006. p. 52–60.

ERTL, M. A.; GREGG, D. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. In: *Proceedings of the Conference on Parallel Processing*. London, UK: Springer-Verlag, 2001. (Euro-Par '01), p. 403–412.

FRASER, C. W.; HANSON, D. R.; PROEBSTING, T. A. Engineering a Simple, Efficient Code-generator Generator. *ACM Letters on Programming Languages and Systems*, ACM, New York, NY, USA, v. 1, p. 213–226, September 1992.

FREE SOFTWARE FOUNDATION. *GNU Compiler Collection*. 2008. `http://gcc.gnu.org/onlinedocs/gcc-4.5.1/gcc/`. Acessado em 5 de novembro de 2010.

FREE SOFTWARE FOUNDATION. *Gas manual – Using as*. 2009. `http://sourceware.org/binutils/docs/as/index.html`. Acessado em 5 de novembro de 2010.

FREE SOFTWARE FOUNDATION. *GNU Compiler Collection Internals – SSA*. 2010. `http://gcc.gnu.org/onlinedocs/gccint/SSA.html`. Acessado em 23 de setembro de 2010.

GILMORE, J.; SHEBS, S. *Testsuite – GDB Internals*. 2010. `http://sourceware.org/gdb/current/onlinedocs/gdbint/Testsuite.html`. Acessado em 8 de maio de 2010.

GOLDBERG, A.; ROBSON, D. *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN 0-201-11371-6.

GOSLING, J.; JOY, B.; STEELE, G. *Java Language Specification*. Boston, MA, USA: Addison Wesley, 2005.

GRANLUND, T.; MONTGOMERY, P. L. Division by Invariant Integers using Multiplication. In: *Proceedings of the Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 1994. p. 61–72.

GRAY, F. *Pulse Code Communication*. March 1953. U.S. Patent 2,632,058.

HANK, R. E.; HWU, W. W. Region-Based Compilation: An Introduction and Motivation. In: *Proceedings of the annual International Symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995.

HERNÁNDEZ, R. OpenACS: Robust Web Development Framework. In: *Annual Tck/Tk Conference*. Portland, Oregon, USA: USENIX, 2005.

HÖLZLE, U. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. Thesis (PhD), Stanford, CA, USA, 1994.

IBM. *PowerPC Architecture Book, Version 2.02*. (Online), 2005. Acessado em 5 de novembro de 2010. Disponível em: <http://www.ibm.com/developerworks/systems/library/es-archguide-v2.html>.

INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 2A: Instruction Set Reference, A-M*. (Online), 2009. Acessado em 23 de setembro de 2010. Disponível em: <http://www.intel.com/products/processor/manuals/>.

INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 3A: System Programming Guide, Part 1*. (Online), 2009. Acessado em 23 de setembro de 2010. Disponível em: <http://www.intel.com/products/processor/manuals/>.

INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture*. (Online), 2009. Acessado em 24 de setembro de 2010. Disponível em: <http://www.intel.com/products/processor/manuals/>.

JONES, N. D.; GOMARD, C. K.; SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Upper Saddle River, NJ, USA: Prentice Hall, 1993.

KAWAHITO, M.; KOMATSU, H.; NAKATANI, T. Effective NULL Pointer Check Elimination Utilizing Hardware Trap. *SIGPLAN Notices*, ACM, New York, NY, USA, v. 35, p. 139–149, November 2000.

KERNIGHAN, B. W.; RITCHIE, D. M. *C Programming Language*. Upper Saddle River, NJ, USA: Prentice Hall, 1988.

KOES, D. R.; GOLDSTEIN, S. C. Near-optimal Instruction Selection on Dags. In: *Proceedings of the International Symposium on Code generation and optimization*. New York, NY, USA: ACM, 2008. p. 45–54.

KRALL, A.; GRAFL, R. CACAO – A 64 bit JavaVM Just-in-Time Compiler. *Concurrency - Practice and Experience*, v. 9, n. 11, p. 1017–1030, 1997.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2004. p. 75.

LEWIS, B. T. An On-The-Fly Bytecode Compiler for Tcl. In: *Proceedings of the USENIX Tcl/Tk Workshop*. Berkeley, CA, USA: USENIX Association, 1996.

LINDHOLM, T.; YELLIN, F. *Java Virtual Machine Specification*. Upper Saddle River, NJ, USA: Prentice Hall, 1999.

MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

OUSTERHOUT, J. K. *Tcl: An Embeddable Command Language*. Berkeley, CA, USA, Nov 1989. Disponível em: <`http://www.eecs.berkeley.edu/Pubs/TechRpts/1989/5742.html`>.

PAPI. *Performance Application Programming Interface*. 2010. `http://icl.cs.utk.edu/papi/`. Acessado em 5 de novembro de 2010.

PLEZBERT, M. P.; CYTRON, R. K. Does "Just in Time" = "Better Late than Never". In: *Proceedings of the Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1997. p. 120–131.

POLETTO, M.; SARKAR, V. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, ACM, New York, NY, USA, v. 21, n. 5, p. 895–913, 1999.

PYPY. *PyPy Coding Guide – Restricted Python*. 2010. `http://codespeak.net/pypy/dist/pypy/doc/coding-guide.html`. Acessado em 5 de novembro de 2010.

RIGO, A. Representation-based Just-In-Time Specialization and the Psyco Prototype for Python. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation*. New York, NY, USA: ACM, 2004. p. 15–26.

RIGO, A.; BOLZ, C. F. How to not write Virtual Machines for Dynamic Languages. In: *Proceeding of the Workshop on Dynamic Languages and Applications*. Berlin, Germany: [s.n.], 2007.

RIGO, A.; PEDRONI, S. PyPy's Approach to Virtual Machine Construction. In: *Symposium on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 2006. p. 944–953.

ROSSUM, G. van; DRAKE, F. L. *Python 2.6 Reference Manual*. Seattle, WA, USA: CreateSpace, 2009.

SAFONOV, V. O. *Trustworthy Compilers*. Hoboken, NJ, USA: Wiley, 2010.

SAH, A. *TC: An Efficient Implementation of the Tcl Language*. Berkeley, CA, USA, Apr 1994. Disponível em: <`http://www.eecs.berkeley.edu/Pubs/TechRpts/1994/5189.html`>.

SANTA CRUZ OPERATION. *System V Application Binary Interface*. 4. ed. 1997. Acessado em 24 de setembro de 2010. Disponível em: <http://www.sco.com/developers/devspecs/abi386-4.pdf>.

SEAL, D. *ARM Architecture Reference Manual*. Boston, MA, USA: Addison Wesley, 2001.

SEBESTA, R. W. *Concepts of Programming Languages*. 9th. ed. USA: Addison-Wesley Publishing Company, 2009.

SUGANUMA, T.; OGASAWARA, T.; KAWACHIYA, K.; TAKEUCHI, M.; ISHIZAKI, K.; KOSEKI, A.; INAGAKI, T.; YASUE, T.; KAWAHITO, M.; ONODERA, T.; KOMATSU, H.; NAKATANI, T. Evolution of a Java Just-In-Time Compiler for IA-32 platforms. *IBM Journal of Research and Development*, IBM Corp., Riverton, NJ, USA, v. 48, n. 5/6, p. 767–795, 2004.

SUGANUMA, T.; YASUE, T.; KAWAHITO, M.; KOMATSU, H.; NAKATANI, T. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2001. p. 180–195.

TCL CORE TEAM. *Tcllib*. 2008. `http://tcllib.sourceforge.net`. Acessado em 9 de maio de 2010.

TCL CORE TEAM. *Tcl Library – man pages*. 2010. `http://www.tcl.tk/man/tcl8.5/TclLib/contents.htm`. Acessado em 5 de novembro de 2010.

VITALE, B.; ABDELRAHMAN, T. S. Catenation and Specialization for Tcl Virtual Machine Performance. In: *Proceedings of the workshop on Interpreters, Virtual Machines and Emulators*. New York, NY, USA: ACM Press, 2004. p. 42–50.

WELCH, B. Customization and flexibility in the exmh mail user interface. In: *Proceedings of the USENIX Workshop on Tcl/Tk*. Berkeley, CA, USA: USENIX Association, 1995.

Apêndice A. Código para alocação e ajuste de páginas

Código 5.1: Alocação de página(s) para o compilador JIT

```
void *
newpage(void *size)
{
    void *page;
#ifdef _WIN32
    page = VirtualAlloc(NULL, *((DWORD *)size),
                        MEM_COMMIT | MEM_RESERVE,
                        PAGE_EXECUTE_READWRITE);
    if (!page) {
        perror("VirtualAlloc");
        exit(1);
    }
#else
    page = mmap(NULL, *((size_t *)size),
                PROT_READ | PROT_WRITE | PROT_EXEC,
                MAP_ANON | MAP_PRIVATE, 0, 0);
    if (page == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }
#endif
    return page;
}
```

Código 5.2: Tamanho, em bytes, de uma página

```
#ifdef _WIN32
DWORD
pagesize(void)
{
    DWORD pagesize;
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    return si.dwPageSize;
}
#else
int pagesize(void)
{
    return getpagesize();
}
#endif
```

Código 5.3: Remoção da permissão de escrita de uma ou mais páginas

```
void
pagenowrite(void *page, size_t len)
{
#ifdef _WIN32
    DWORD oldProtect;
    if (VirtualProtect(page, len, PAGE_EXECUTE_READ,
                       &oldProtect) == 0) {
        perror("VirtualProtect");
        exit(1);
    }
#else
    if (mprotect(page, len, PROT_READ | PROT_EXEC) < 0) {
        perror("mprotect");
        exit(1);
    }
#endif
}
```

Universidade Estadual de Maringá

Departamento de Informática

Av. Colombo 5790, Maringá - PR, CEP 87020-900

Tel: (44) 3261-4324 Fax: (44) 3263-5874

www.din.uem.br