Universidade Estadual de Maringá
Centro de Tecnologia
Departamento de Informática
Bacharelado em Ciência da Computação

Trabalho de Conclusão de Curso - TCC 2010

Tcl JIT

Guilherme Henrique Polo Gonçalves
Orientador: Prof. Dr. Anderson Faustino da Silva

Guilherme Henrique Polo Gonçalves

Prof. Dr. Anderson Faustino da Silva

Maringá, 15 de julho de 2010

Tcl JIT

Guilherme Henrique Polo Gonçalves
Orientador: Prof. Dr. Anderson Faustino da Silva

Resumo

Linguagens de programação interpretadas tem trocado performance por maior expressividade, flexibilidade, dinamicidade, entre outros. Reescrita de trechos de código críticos em linguagens compiladas tem sido empregada como meio de reduzir o impacto da máquina virtual no tempo de execução de programas. Diante disso, um compilador JIT para a linguagem de programação Tcl é proposto como forma de diminuir a necessidade de tal reescrita de código. Um modo misto de execução é escolhido, fazendo com que execução de código de máquina, gerado por esse compilador dinâmico, e a interpretação pura se alternem. Procedimentos são definidos como os limites de inicio e término de compilação, e a compilação baseada em regiões é rapidamente mencionada. Simplicidade, manutenibilidade e flexibilidade são características que serão seguidas na construção dessa ferramenta. As etapas de desenvolvimento são, então, brevemente descritas.

Palavras-chave: Tcl, JIT, código de máquina

1 Introdução

Linguagens de programação interpretadas tem sacrificado performance em favor de um alto nível de abstração do funcionamento da máquina envolvida, possibilitando maior expressividade, facilidade de desenvolvimento, portabilidade, flexibilidade, dinamicidade, entre outros. Especificamente a linguagem de programação `Tcl` (*Tool Command Language*), teve como um de seus maiores objetivos a facilidade de incorporação (*embedding*) a programas que desejassem ter uma linguagem de comandos (OUSTERHOUT, 1989). Uma interface simples e extensível para aplicações em linguagem `C` era o fator motivante de seu uso. Programas inteiramente em `Tcl` eram vistos como pequenos scripts, muitos talvez de uma linha no máximo (OUSTERHOUT, 1989). Entretanto, aplicações em `Tcl` com milhares de linhas, como por exemplo exmh (WELCH, 1995), OpenACS (HERNÁNDEZ, 2005) ou mesmo a suíte de testes do GDB (*GNU Debugger*) (GILMORE; SHEBS, 2010), tem surgido e a reescrita de trechos críticos em `C` vem sendo aplicada para reduzir o impacto da máquina virtual no tempo de execução.

Outra forma de melhorar o desempenho de linguagens e, portanto, reduzir a necessidade de reescrita de código em linguagens compiladas, é através da utilização da compilação JIT (*Just-In-Time*) que realiza tradução de código sob demanda. No caso desse trabalho, estamos interessados na tradução de *bytecodes* da máquina virtual `Tcl` para código de máquina durante o tempo de execução. Informações acerca do programa em execução são coletadas conforme necessário para dirigir a compilação dinâmica. Portanto, linguagens tipicamente difíceis de serem analisadas e compiladas estaticamente, devido a uso de, por exemplo, tipagem dinâmica ou escopo dinâmico, ganham a oportunidade de melhoria de desempenho com uso de tal sistema de compilação.

O trabalho presente pretende melhorar o desempenho da linguagem `Tcl`, reduzindo o tempo de decodificação e interpretação de *bytecodes* através da implementação e implantação de um compilador JIT na mesma. Diversos trabalhos, como (DEUTSCH; SCHIFFMAN, 1984) para a linguagem `Smalltalk`, Jalapeño (ALPERN et al., 2000) para `Java`, Psyco (RIGO, 2004) para `Python` ou a implementação do SELF-93 (HÖLZLE, 1994), já demonstraram resultados bastante significativos ao implantar tal método de compilação. Pretende-se ainda manter um nível de manutenibilidade e flexibilidade adequado, permitindo extensões e futuro desenvolvimento sobre esse trabalho inicial.

As seções seguintes dessa proposta estabelecem o que, e como, será feito no decorrer do trabalho.

2 Objetivos

Construir um compilador JIT que gere código de máquina para a arquitetura IA-32 durante a interpretação de programas `Tcl` é o grande objetivo desse trabalho.

Certas construções de linguagem existentes na `Tcl` dificultam o processo de compilação (estática ou dinâmica) e já foram alvo de discussão em trabalhos passados (SAH; BLOW, 1993) (SAH, 1994), levando até mesmo a decisão de se criar uma outra linguagem similar (SAH; BLOW; DENNIS, 1994) eliminando as construções mais problemáticas. Por essa razão, o trabalho discutido aqui pretende focar em apenas um subconjunto significativo da linguagem.

O projeto também pretende explorar como sistemas compactos, simples e com menos recursos se comparados a frameworks como LLVM (*Low Level Virtual Machine*) (LATTNER, 2002), influenciam no desempenho geral. Porém, apesar da simplicidade em vista, espera-se que o trabalho desenvolvido possa servir de base para expansão e criação de novos recursos.

3 Proposta

Para chegar ao código de máquina final, esse projeto pretende em primeiro momento estudar e definir qual subconjunto da linguagem pode se beneficiar mais com tal técnica. O subsistema de Entrada/Saída, por exemplo, dificilmente ganharia em desempenho com compilação dinâmica uma vez que o tempo gasto para transferência e aguardo por dispositivos costuma ser muito maior que o tempo das outras operações envolvidas. Por outro lado, operações aritméticas podem ser bastante favorecidas. O trabalho feito no Psyco, mostra melhoria de 109 vezes no tempo de execução para aritmética de inteiros e 10,9 vezes em aritmética de ponto flutuante (RIGO, 2004).

Em paralelo ao requisito acima, será iniciado o projeto e implementação de uma representação intermediária de baixo nível. Queremos utilizar um conjunto reduzido de instruções, assim como é feito na GNU lightning (FREE SOFTWARE FOUNDATION, 2007), para construir uma árvore de instruções mais próximas da máquina alvo. O uso de árvore aqui é devido a existência de diversos métodos para seleção de instruções que trabalham sobre essa forma de representação (ERTL; CASEY; GREGG, 2006). O sistema de compilação dinâmica precisa ser capaz de determinar quando iniciar e parar a construção de tal árvore. Nesse projeto a intenção é fazer uso dos limites de um procedimento como pontos de inicio e parada. Ainda aqui, é importante que o sistema colete informações, como de tipos

utilizados, necessárias de forma a simplificar o código gerado, para que não repliquemos os resultados do trabalho feito em (VITALE; ABDELRAHMAN, 2004).

Após as etapas acima, chegamos na fase de seleção de instruções seguida de alocação de registradores. Há diversos algoritmos, como *Maximal Munch*, seleção com uso de programação dinâmica ou NOLTIS (KOES; GOLDSTEIN, 2008) para seleção de instruções além de vários outros, por coloração de grafos ou mesmo através de uma varredura linear (POLETTO; SARKAR, 1999), para alocação de registradores. Ainda não está decidido quais dos algoritmos serão aplicados nesse projeto, porém as metas são duas: que a implementação permita utilizar diferentes algoritmos e que o tempo gasto para execução deles seja compatível com um sistema que consome tempo de execução do programa.

Finalmente temos a geração de código. O foco desse trabalho é a arquitetura IA-32, que possui uma ampla quantidade de instruções, extensões e diversas formas de endereçamento. Porém, não pretendemos fazer uso de todos os recursos disponíveis de forma a tornar factível a criação do gerador. Idéias e trechos de trabalhos anteriores, como Harpy (GRABMÜELLER; KLEEBLATT, 2007) ou o *Tiny Code Generator* (incluído nas versões mais recentes do QEMU (BELLARD, 2005)), podem ser reutilizadas aqui. Ainda, de forma semelhante com a etapa anterior, a infraestrutura permitirá a implantação de geração de código para outras arquiteturas.

4 Revisão Bibliográfica

A linguagem `Tcl` já obteve ganhos de performance em diferentes estudos feitos. Um deles, que atualmente faz parte da implementação da linguagem, descrito em (LEWIS, 1996), é a geração e a interpretação de *bytecodes*. Anterior a esse trabalho, foi demonstrado em (SAH, 1994) que o *parsing* do código realizado a todo momento para sua reinterpretação e também a conversão excessiva entre tipos de dados, já que a `Tcl` trata tudo como string, eram os grandes consumidores do tempo de execução. Com esse trabalho feito, a linguagem passou a utilizar representações duplas para os valores presentes na execução do programa. Uma representação é interna, possivelmente mais eficiente para se trabalhar. A outra é a típica representação em string que a linguagem sempre usou. Caso uma delas não esteja disponível, a outra é utilizada para recriar essa representação se necessário.

Um trabalho mais recente, descrito em (VITALE; ABDELRAHMAN, 2004), lida com a eliminação do *overhead* de decodificação dos *bytecodes*, introduzido pelo trabalho descrito anteriormente, fazendo uso de *templates* que contém as instruções em código nativo utilizadas para interpretar cada *bytecode*. Esse código é obtido através 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 trabalho o interpretador foi modificado de forma a sempre executar somente tal código formado por uma concatenação de *templates*, eliminando o *overhead* de decodificação. Demonstrou-se que em certos testes 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. Ele não gera código, mas cópia 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 e assim limita o potencial de melhoria de desempenho.

Outros trabalhos, para diferentes linguagens, se assemelham mais com a proposta aqui discutida. A busca por máquinas virtuais de alta performance tem, atualmente, se dirigido principalmente a linguagem `Java`. É comum a presença de compiladores JIT nessas máquinas virtuais, cada um com diferentes características. A JUDO (CIERNIAK; LUEH; STICHNOTH, 2000), 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. O segundo compilador é utilizado quando informações coletadas indicam que certos métodos são executados muito frequentemente e, portanto, estes podem se beneficiar 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. Esse sistema trabalha com a compilação de métodos por inteiro, assim como o trabalho proposto aqui. Já o trabalho discutido em (SUGANUMA; YASUE; NAKATANI, 2003) avalia a aplicação de compilação dinâmica a regiões de código, evitando a compilação de trechos raramente executados. Além disso, esse sistema utiliza um modo misto de execução, onde interpretação e execução de código nativo se alternam. Nesse ponto, nosso trabalho e aquele em (SUGANUMA; YASUE; NAKATANI, 2003) se assemelham.

5 Metodologia

A metodologia será experimental, onde será implementado uma "ferramenta" e então avaliada para provar a tese. A seguir são resumidas as etapas previstas para a construção desse trabalho.

Levantamento bibliográfico

Selecionar artigos, livros e manuais a serem utilizados ao longo da escrita textual e também na escrita de código. Essa etapa ocorre em paralelo a todas as outras descritas adiante, pois novos documentos serão selecionados conforme o trabalho avança em direção aos objetivos estabelecidos.

Especificação do compilador JIT

Estabelecer um formato para a representação intermediária, avaliar e definir algoritmos para seleção de instruções e alocação de registradores serão realizados aqui. Essa etapa ocorrerá logo no inicio do projeto e também durante o progresso da implementação do compilador de forma a validar as escolhas feitas.

Também será estabelecido como a implementação do compilador deverá acontecer, mantendo o objetivo de permitir que algoritmos alternativos venham a ser utilizados no lugar daqueles aqui selecionados e possibilitando que diferentes arquiteturas explorem os recursos disponíveis de maneira adequada.

Especificação da arquitetura da `Tcl`

Inicialmente será feito uma análise para verificar qual subconjunto da linguagem cabe ao trabalho proposto. A máquina virtual deve então ser ajustada para conseguir determinar quando um procedimento se enquadra no subconjunto definido ou não, decidindo entre inserir o procedimento como candidato a compilação JIT ou não. Outro ajuste está relacionado com a notificação ao compilador JIT sobre comandos que são redefinidos, possivelmente alterando o código anterior associado a ele, e portanto precisam ser recompilados. A máquina virtual da `Tcl` já conta com um contador para cada comando implementado, do usuário ou não, que é incrementado a cada redefinição e, assim, esse compilador fará uso desse recurso.

Essa etapa será feita em diversos momentos, conforme a implementação do compilador progride, validando o que foi já definido.

Implementação do Compilador

Essa etapa refere-se a implementação da representação intermediária de baixo nível, seleção de instruções, alocação de registradores e geração de código de máquina para a arquitetura IA-32. Os algoritmos utilizados para seleção de instruções e alocação de registradores serão aqueles definidos na etapa de especificação do compilador JIT.

Acoplamento do compilador ao ambiente de execução

Definir como detectar e lidar com exceções no código de máquina gerado, possibilitando a retomada da execução por meio de interpretação pura. Essa e outras situações não

estão planejadas para serem implementadas devido ao tempo disponível. Discussões de possíveis soluções serão levantadas no mesmo momento da etapa de avaliação.

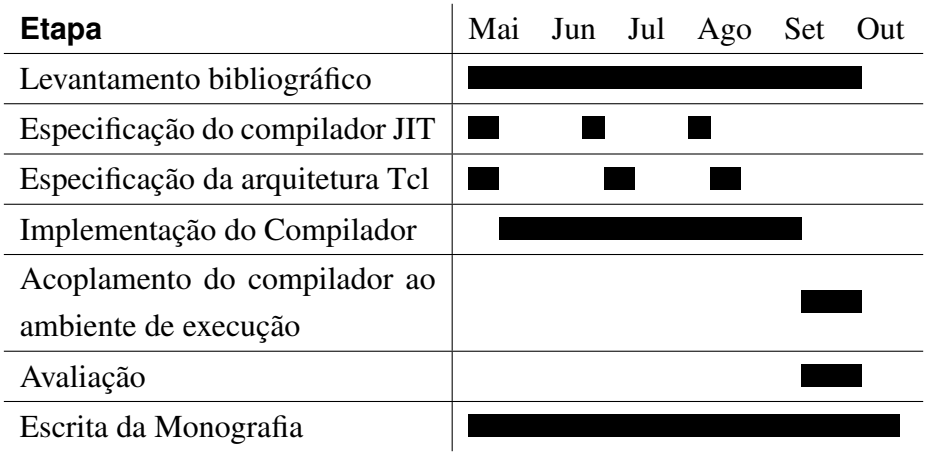
Avaliação

Pretende-se fazer uso da suíte de *benchmarks* tclbench (TCL CORE TEAM, 2008) já que essa tem sido utilizada para verificar e comparar a performance da `Tcl` ao longo de vários anos. Outras formas de avaliação, como tamanho de código gerado e tempo gasto para compilação por *bytecode* também poderão ser aplicadas conforme o tempo permitir. Planeja-se iniciar essa etapa após a implementação, pois há dependência de um sistema JIT em funcionamento para se executar a avaliação.

Elaboração da Monografia

Partes textuais da monografia, que não dependam de resultados obtidos através da realização das etapas de implementação, serão escritas quão antes possível. Experimentos realizados ao longo do desenvolvimento influenciarão diversas seções do trabalho escrito e, portanto, o desenvolvimento dessa parte se dará ao longo de todo o tempo disposto ao trabalho.

6 Cronograma de Execução



7 Referências

ALPERN, B. et al. The jalapeño virtual machine. *IBM Syst. J.*, IBM Corp., Riverton, NJ, USA, v. 39, n. 1, p. 211–238, 2000. ISSN 0018-8670.

BELLARD, F. Qemu, a fast and portable dynamic translator. In: *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA:

USENIX Association, 2005.

CIERNIAK, M.; LUEH, G.-Y.; STICHNOTH, J. M. Practicing judo: Java under dynamic optimizations. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, n. 5, p. 13–26, 2000. ISSN 0362-1340.

DEUTSCH, L. P.; SCHIFFMAN, A. M. Efficient implementation of the Smalltalk-80 system. In: *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*. Salt Lake City, Utah: [s.n.], 1984. p. 297–302.

ERTL, M. A.; CASEY, K.; GREGG, D. Fast and flexible instruction selection with on-demand tree-parsing automata. In: *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2006. p. 52–60. ISBN 1-59593-320-4.

FREE SOFTWARE FOUNDATION. *Using and porting GNU lightning*. 2007. `http://www.gnu.org/software/lightning/manual/lightning.html`. Acessado em 9 de maio 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.

GRABMÜELLER, M.; KLEEBLATT, D. Harpy: run-time code generation in haskell. In: *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. New York, NY, USA: ACM, 2007. ISBN 978-1-59593-674-5.

HERNÁNDEZ, R. Openacs: robust web development framework. In: *12th Annual Tck/Tk Conference*. Portland, Oregon, USA: [s.n.], 2005.

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

KOES, D. R.; GOLDSTEIN, S. C. Near-optimal instruction selection on dags. In: *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2008. p. 45–54. ISBN 978-1-59593-978-4.

LATTNER, C. A. *LLVM: An Infrastructure for Multi-Stage Optimization*. [S.l.], 2002.

LEWIS, B. T. An on-the-fly bytecode compiler for tcl. In: *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996*. Berkeley, CA, USA: USENIX Association, 1996.

OUSTERHOUT, J. K. *Tcl: An Embeddable Command Language*. [S.l.], Nov 1989.

Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/1989/5742.html>.

POLETTO, M.; SARKAR, V. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 21, n. 5, p. 895–913, 1999. ISSN 0164-0925.

RIGO, A. Representation-based just-in-time specialization and the psyco prototype for python. In: *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. New York, NY, USA: ACM, 2004. p. 15–26. ISBN 1-58113-835-0.

SAH, A. *TC: An Efficient Implementation of the Tcl Language*. [S.l.], Apr 1994. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/1994/5189.html>.

SAH, A.; BLOW, J. *A Compiler for the Tcl Language*. 1993.

SAH, A.; BLOW, J.; DENNIS, B. An introduction to the rush language. In: *TCLTK'94: Proceedings of the Tcl/Tk 1994 Workshop*. Berkeley, CA, USA: [s.n.], 1994.

SUGANUMA, T.; YASUE, T.; NAKATANI, T. A region-based compilation technique for a java just-in-time compiler. In: *In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. [S.l.]: ACM Press, 2003. p. 312–323.

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

VITALE, B.; ABDELRAHMAN, T. S. Catenation and specialization for tcl virtual machine performance. In: *In IVME '04 Proceedings*. [S.l.]: ACM Press, 2004. p. 42–50.

WELCH, B. Customization and flexibility in the exmh mail user interface. In: *TCLTK '98: Proceedings of the 3rd Annual USENIX Workshop on Tcl/Tk*. Berkeley, CA, USA: USENIX Association, 1995.