Evandro Augusto Marucci
Luigi Jacometti de Oliveira

Investigando a Otimização Automática de Códigos através do Paradyn

São José do Rio Preto – SP

Novembro / 2006

Evandro Augusto Marucci
Luigi Jacometti de Oliveira

Investigando a Otimização Automática de Códigos através do Paradyn

Monografia apresentada ao Departamento de Ciências da Computação e Estatística da UNESP/IBILCE como parte dos requisitos necessários para a aprovação na disciplina Projeto Final

Orientador:
Prof. Dr. Aleardo Manacero Jr.

Departamento de Ciências da Computação e Estatística
Instituto de Biociências, Letras e Ciências Exatas
Universidade Estadual Paulista

São José do Rio Preto – SP

Novembro / 2006

Monfografia de projeto final de graduação sob o título *"Investigando a otimização automática de código através do Paradyn"*, apresentada por Evandro Augusto Marucci e Luigi Jacometti de Oliveira e aprovada em 30 de novembro de 2006, em São José do Rio Preto, São Paulo, pela banca examinadora constituída pelos professores:

Prof. Dr. Aleardo Manacero Jr.
Orientador

Profª. Drª. Renata Spolon Lobato

Prof. Dr. Sílvio Alexandre Araújo

Evandro:

Aos meus pais, ao meu irmão e à minha namorada.

Luigi:

Dedico esse trabalho aos meus pais Ubiratan e Damaris, aos meus irmãos Lucas e Leonardo e à minha namorada Mariana.

Agradecimentos

Nossos sinceros agradecimentos:

– À Deus, por todas as bençãos oferecidas e por mais uma conquista, sempre nos iluminando, nos dando sabedoria e nos conduzindo nos momentos mais difíceis.

– Aos nossos familiares, que sempre nos apoiaram em cada etapa de nossas vidas; em especial aos nossos pais, por nos terem proporcionado condições para estarmos aqui hoje e pelo amor incondicional sempre manifestado.

– Às nossas namoradas, Camila (Evandro) e Mariana (Luigi), pelo companheirismo, carinho, incentivo e paciência, compartilhando todos os momentos alegres e difíceis no decorrer de grande parte do curso.

– Ao professor, amigo e orientador Aleardo Manacero Junior, que de forma fraterna, auxiliou-nos para a elaboração do presente trabalho. Seu companheirismo, incentivo e suas lições de caráter, muitas vezes implicitamente manifestadas, nos ajudaram a prosseguir nesta área e em vários aspectos de nossa vida profissional.

– Aos nossos amigos que, durante todo o curso, compartilharam momentos indispensáveis à nossa vida acadêmica. Aos amigos André, Caio, Eli, Geraldo, Jorge e Willian Lima, em especial, pela sólida amizade construída durante os 4 anos. Os momentos divertidos vividos e a ajuda de vocês nos momentos em que mais precisamos nos deram forças para chegarmos até aqui.

– À FAPESP - Fundação de Amparo à Pesquisa do Estado de São Paulo - pelo apoio financeiro para a execução deste projeto.

"The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet."

Michael Anthony Jackson

Resumo

O desenvolvimento dos compiladores teve um importante papel quanto ao aumento de desempenho dos programas, obtido a partir de técnicas cada vez mais aprimoradas de otimização de código. As otimizações empregadas pelos compiladores, no entanto, estão limitadas a heurísticas estáticas e não têm informações sobre como o programa é executado. Uma análise do programa, através de sua execução, é necessária para coletar algumas informações relevantes, possibilitando uma otimização mais eficiente.

Este projeto se encaixa nesse contexto, procurando identificar os pontos da aplicação de otimização pós-compilação. Isso é feito através da instrumentação de laços em funções previamente identificadas como gargalo pelo Paradyn. O Paradyn é uma ferramenta utilizada na análise de desempenho de programas paralelos e distribuídos e realiza instrumentação dinâmica da aplicação. Assim, torna-se possível concentrar os esforços da otimização em regiões comprovadamente responsáveis pela degradação de desempenho da aplicação.

Uma vez que o Paradyn trabalha apenas com o código binário da aplicação, investigou-se as técnicas possíveis de serem aplicadas em um código de baixo nível. Além disso, com essas técnicas, explorou-se possibilidades de otimização não abordadas por compiladores. Entre as técnicas de otimização que satisfazem essas particularidades, foram implementadas: alinhamento de funções, propagação de constantes e remoção de instruções `LOAD/RESTORE` redundantes. Observa-se que a aplicação do alinhamento de funções habilita as duas outras técnicas.

Os resultados obtidos pelo módulo otimizador, que implementa as três técnicas assinaladas acima, mostram um ganho de desempenho da aplicação, especialmente quando um pequeno laço gargalo, alvo da otimização, itera muitas vezes e faz chamadas a funções pequenas, que serão alinhadas dentro do corpo do laço.

Abstract

The development of compilers had played an important role in the increasing of programs performance which can be achieved through even more improved optimization code techniques. However, the optimizations employed by compilers are limited to static heuristics and they don't have knowledge about how the program is executed. A program analysis, trough its execution, is needed to collect some important information to enable a more efficient optimization.

This project fits in this context, trying to identify points for optimization after compilation time. This is done trough the loop instrumentation over previously identified bottleneck functions by Paradyn tool. The Paradyn is a tool used for performance analysis of parallel and distributed programs and performs dynamic instrumentation of the application. Therefore, it makes possible to concentrate the optimization efforts in areas that really degenerates the application performance.

Since Paradyn works only with application's binary code, the possible techniques that can be applied in low level code were investigated. Moreover, some possibilities that are not attempted by compilers were explored with these techniques. Among the optimization techniques which attend these details, were implemented: function inlining, constant propagation, redundant `LOAD/RESTORE` elimination. It must be observed that the application of function inlining allows the other two techniques.

The results achieved by the optimizer module, which implements the three techniques described above, shown an application performance improvement, especially when a short bottleneck loop, as the target of optimization, iterates many times and make calls to short functions, which will be inlined within the loop's body.

Sumário

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

As tecnologias para a produção de compiladores desenvolveram-se amplamente nos últimos anos. Este desenvolvimento teve um papel fundamental quanto ao aumento de desempenho dos programas, obtido a partir de técnicas cada vez mais aprimoradas de otimização de código. Muitas delas são incorporadas aos compiladores, produzindo um código comparável ou até melhor do que um escrito em código de máquina manualmente. Otimizações clássicas, como remoção de código morto, propagação de constantes e eliminação de sub-expressões comuns, diminuem o tempo de execução eliminando computação redundante [1]. Outras otimizações, como remoção de código invariante de laço e eliminação das variáveis de indução de laço diminuem o tempo de execução movendo instruções de regiões frequentemente executadas para outras menos freqüentes [1]. Estas otimizações, no entanto, estão limitadas a heurísticas estáticas e não têm informações sobre como o programa é executado.

Uma análise do programa, através de uma execução inicial, é suficiente para coletar algumas informações relevantes, possibilitando uma otimização mais eficiente. O projeto desenvolvido faz essa análise em uma etapa inicial, durante a fase de instrumentação e medição de desempenho dos laços em funções previamente identificadas como gargalos pelo Paradyn [2]. Isto identifica regiões mais frequentemente executadas no programa, permitindo algumas otimizações como alinhamento de funções e *loop unrolling*. Estas técnicas, em geral, melhoram o tempo de execução dos programas, pois minimizam o *overhead* gerado com as chamadas de função e o número de instruções de salto, além de permitir a expansão de outras otimizações. No entanto, sua aplicação tem como custo o aumento significativo no tamanho do executável. Por esse motivo, a busca por trechos gargalos no programa deve ser muito bem efetivada, garantindo um ganho de desempenho

considerável apesar do aumento no tamanho do programa.

1.1 Objetivos

Nesse documento apresenta-se uma continuação ao trabalho recentemente realizado sobre a instrumentação de laços de repetição com o Paradyn. O que se propôs foi uma investigação preliminar sobre a viabilidade da aplicação de técnicas de otimização de laços utilizadas em compiladores como possível suporte para a otimização automática de código a partir da detecção de gargalos de execução.

Após essa fase inicial, objetiva-se avançar nessa análise através da implementação de módulos que permitam a alteração dinâmica do código binário, seu teste por meio de nova instrumentação e sugestão ao usuário, caso a alteração seja eficiente, da aplicação da mesma sobre o código fonte.

A versão do Paradyn utilizada nesse projeto baseia-se na versão 4.0.1 da ferramenta estendida em um trabalho anterior [3], o qual basicamente oferece uma expansão das capacidades de análise e decisão oferecidas pelo Paradyn [2], permitindo o refinamento do grão mínimo de análise para o nível de blocos estruturais internos às funções (como laços de repetição). Essa extensão do Paradyn teve como alvo a arquitetura Sparc. No entanto, utilizou-se neste projeto um *cluster* de processadores x86 para implementação e execução de testes, visto que as estações Sun utilizadas no projeto anterior já se tornavam obsoletas.

1.2 Organização da Monografia

No capítulo dois é descrita a fundamentação teórica necessária para a realização do projeto, centrando-se no estudo do Paradyn e das técnicas de otimização possíveis de serem aplicadas sobre o código analisado.

As etapas de desenvolvimento do projeto, incluindo sua especificação e implementação são abordadas no capítulo três. No capítulo quatro são apresentados os testes e verificação da funcionalidade do sistema. Finalmente, no capítulo cinco apresentam-se as conclusões e os possíveis trabalhos futuros na linha desse projeto.

2 Fundamentação teórica do projeto

No decorrer do projeto estudou-se a versão estendida do Paradyn, procurando compreender seus princípios de funcionamento e a interação entre a ferramenta e o módulo de instrumentação de laços desenvolvido no projeto anterior. Uma breve descrição do Paradyn, bem como das técnicas de instrumentação de laços são apresentadas na seção 2.1.

A outra etapa do projeto, que envolveu o desenvolvimento do módulo otimizador, exigiu um estudo sobre as principais técnicas de otimização de código, possíveis de serem aplicadas em um código executável. Entre essas técnicas, revelaram-se mais importantes: alinhamento de funções ou *function inlining* (descrita em 2.2.1), otimização *peephole* (descrita em 2.2.2), propagação interprocedural de constantes (descrita em 2.2.3), remoção de `LOAD/RESTORE` redundantes (descrita em 2.2.4) e desdobramento de laços ou *loop unrolling* (descrita em 2.2.5).

2.1 O Paradyn

O Paradyn é uma ferramenta utilizada na análise de desempenho de programas paralelos e distribuídos. Ele realiza a instrumentação dinâmica da aplicação, buscando extrair traços de eventos sem que o usuário precise alterar o código fonte do programa ou necessite de um compilador especial [2, 3]. A instrumentação empregada pelo Paradyn, sendo dinâmica, possibilita a análise de desempenho durante a execução da aplicação, em tempo real. Essa técnica permite ainda adicionar a instrumentação no programa em execução apenas

enquanto ela for necessária, removendo-a quando não for mais preciso realizar medições. Há também a possibilidade de análise dos arquivos gerados após a execução do programa (*post-morten*), embora isso não seja o objetivo da ferramenta.

A instrumentação é feita por meio da criação de trampolins, em que chamadas de função são substituídas por desvios incondicionais para medições do Paradyn [2]. Como essas substituições ocorrem sem a alteração do restante do código, torna-se possível manter o programa em execução mesmo durante a operacionalização dessas alterações, viabilizando assim sua instrumentação dinâmica. A figura 2.1 ilustra o desvio da função original da aplicação para uma região de código (compartilhado entre o Paradyn e a aplicação) contendo a função original mais as instrumentações adequadas.

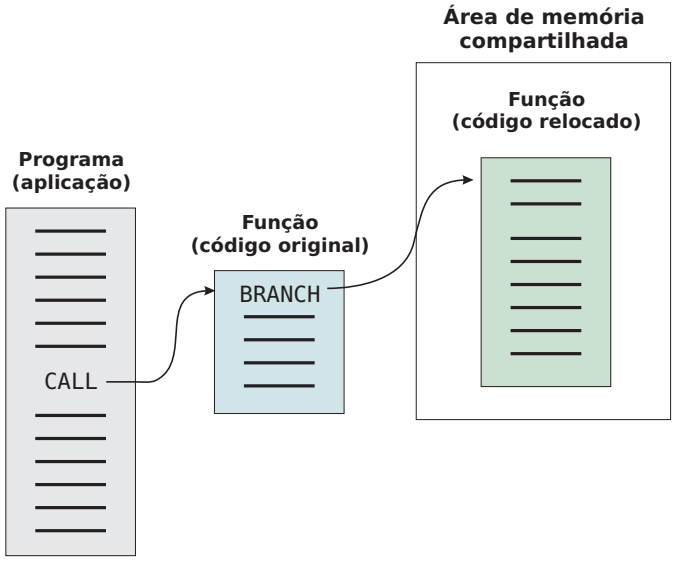


Figura 2.1: Representação da relocação de função realizada pelo Paradyn

O Paradyn, na versão 4.0.1, é capaz de identificar gargalos de execução em programas paralelos com granulação média (funções). A versão do Paradyn utilizada nesse projeto é resultante de um projeto anterior [3], que usa como base a versão 4.0.1 da ferramenta. No projeto anterior foram desenvolvidos módulos que permitem o refinamento do grão de análise da ferramenta, de forma a identificar quais laços de repetição dentro da função apontada como gargalo são os reais responsáveis pela degradação de desempenho. A versão atual do Paradyn (5.0), desenvolvida pelo grupo de pesquisa da Universidade de Wisconsin, também determina os pontos de degradação no desempenho das aplicações em nível de

laços de repetição. A abordagem por eles empregada, no entanto, difere da utilizada nesse projeto.

O grupo de Wisconsin realiza a instrumentação dos laços de todas as funções, independente de essas serem ou não gargalos, como mostrado na figura 2.2. Já a abordagem adotada no projeto anterior e seguida por esse projeto consiste em identificar inicialmente uma função como gargalo para, posteriormente, efetuar as medições nos laços inserindo as instrumentações adequadas. Essa abordagem é ilustrada na figura 2.3.

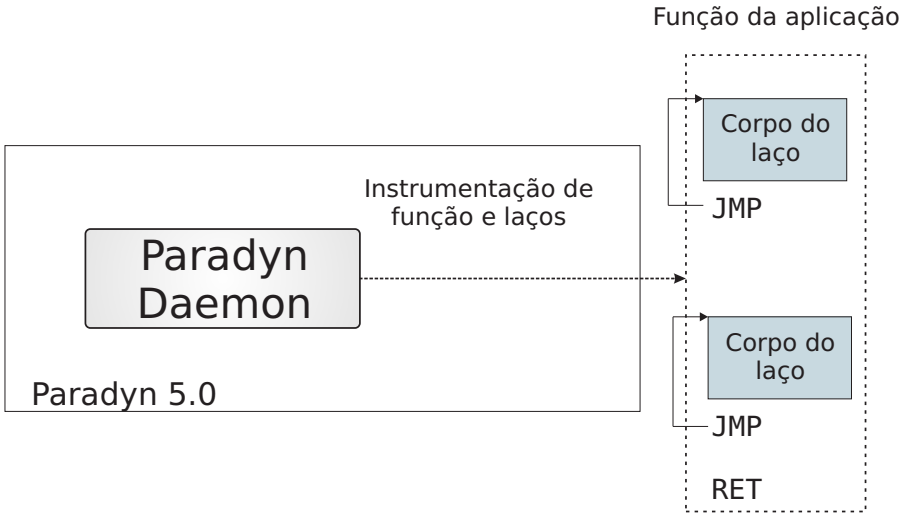


Figura 2.2: Ilustração da abordagem de instrumentação de laços original do Paradyn

2.1.1 Estrutura do Paradyn

A ferramenta divide-se em um *front-end* e um *back-end*. O *front-end* é um sistema *multi-threaded* composto por vários módulos, os quais são apresentados a seguir:

- *User Interface* (UI): é responsável por receber os comandos do usuário e gerenciar as janelas de exibição;
- *Data Manager* (DM): é responsável por tratar as requisições dos outros *threads* por dados coletados nas medições e pelo recebimento dos dados sobre o desempenho da aplicação, coletados pelo *daemon* do Paradyn (*back-end*). Toda comunicação entre o *front-end* e o *back-end* é realizada por meio do DM;

- *Performance Consultant* (PC): é responsável por automatizar a busca por gargalos de desempenho na aplicação. Ele requisita dados obtidos pelas medições dos *daemons* e determina se o bloco analisado é ou não gargalo;

- *Visi Manager*: é responsável pelo gerenciamento dos processos de visualização e pela comunicação entre os processos de visualização e o DM.

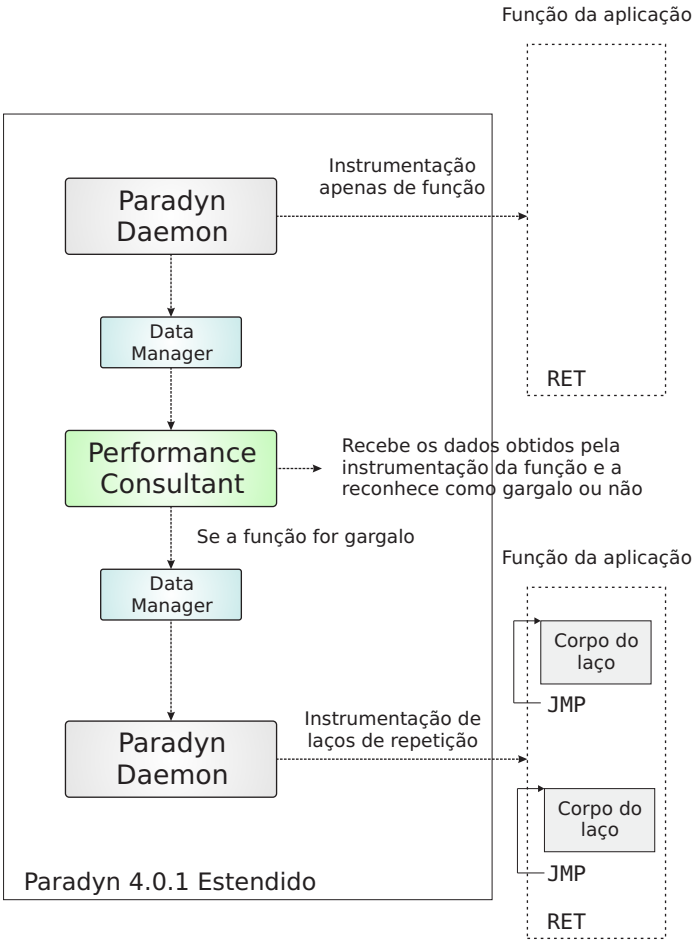


Figura 2.3: Ilustração da abordagem de instrumentação de laços utilizada no projeto

A figura 2.4 mostra de forma simplificada a arquitetura do Paradyn.

O *back-end* é dado pelo *Paradyn Daemon* (paradynd). Quando um programa paralelo é executado, é criado um *daemon* para cada nó. Ele desempenha as seguintes funções [4]:

- Iniciar e controlar a execução da aplicação;

- Ler a tabela de símbolos da aplicação;
- Ler a imagem binária da aplicação para determinar os pontos de instrumentação;
- Avaliar métricas, gerar código e inserir a instrumentação na aplicação
- Periodicamente obter amostras de dados a respeito do desempenho da aplicação e enviar os valores obtidos para o *Paradyn front-end*.

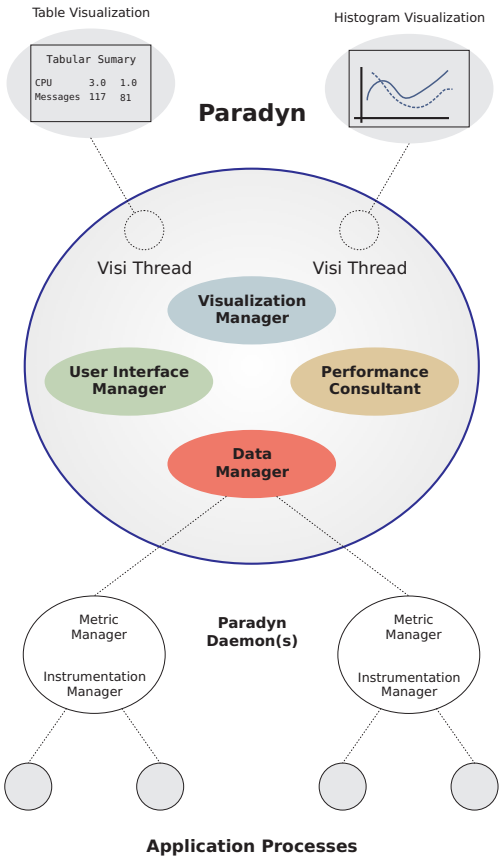


Figura 2.4: Arquitetura do Paradyn[2]

O *Paradyn Daemon* contém toda as partes dependentes de plataforma da ferramenta, uma vez que é o módulo que trabalha diretamente com o código executável da aplicação. A instrumentação realizada pelos *daemons* é feita através da biblioteca de instrumentação dinâmica *DyninstAPI*, por meio da qual o Paradyn adiciona pequenas porções de instrumentação no código executável da aplicação para procurar possíveis pontos de degradação

de desempenho (como gargalos de comunicação ou entrada/saída). A cada ponto de degradação encontrado, a ferramenta adiciona uma quantidade maior de instrumentação para pesquisar o possível problema.

2.2 Técnicas de otimização

A otimização, no contexto computacional, é o processo de modificar um sistema de *hardware* ou *software* com o objetivo de maximizar a sua eficiência. A otimização aplicada em código (fonte ou executável) é frequentemente realizada pelo compilador. Os compiladores, no entanto, aplicam técnicas de otimização baseados em heurísticas estáticas, uma vez que não possuem informações sobre a execução do programa. O uso de uma ferramenta de análise de desempenho, como o Paradyn, permite coletar tais informações, possibilitando focalizar o esforço de otimização nas regiões do programa identificadas como gargalos.

Nessa seção serão abordadas algumas técnicas de otimização possíveis de serem aplicadas em um código executável, e que após um estudo inicial, mostraram-se mais adequadas em termos de ganho de desempenho proporcionado e viabilidade de implementação.

2.2.1 Alinhamento de funções

O alinhamento de funções (*function inlining*) é uma otimização na qual uma instrução `CALL` é substituída por uma instância da função chamada. O objetivo é eliminar o *overhead* associado a uma chamada `CALL` e a instrução de retorno de função, permitindo ainda novas otimizações, como propagação de constantes. No entanto, como esta técnica cria uma cópia do procedimento, no lugar da sua chamada, o tamanho do código pode crescer consideravelmente, provocando o efeito chamado *code bloat* [5], em o que tempo de compilação e o espaço de memória ocupado tornam-se intoleráveis.

A coleta de dados pelo Paradyn durante a execução da aplicação permite determinar quais pontos são bons candidatos para esse tipo de otimização. Esta análise é de grande importância, pois o aumento do código pode ter um efeito devastador no comportamento do programa, principalmente em relação ao desempenho da *cache*. A diferença entre a velocidade de acesso a memória RAM e a CPU é tão grande, que as perdas de *cache*

podem compensar qualquer ganho esperado com a otimização.

Há vários trabalhos relacionados com o alinhamento de funções. O trabalho desenvolvido por Zhao e Amaral [5] estabelece a seguinte orientação para alinhamento de funções:

1. A instrução de chamada de função deve ser executada frequentemente (por exemplo, uma chamada de função dentro de um laço de repetição com muitas iterações).
2. A função chamadora e a função chamada não devem ser muito grandes, pois quanto maior for o tamanho do código resultante, maior será a possibilidade de os dados não conseguirem ser todos armazenados no conjunto de registradores, sendo necessária a inserção de instruções de `LOAD/RESTORE` no código.

Um outro trabalho, proposto por McFarling [6], apresenta um esquema para analisar o comportamento da memória *cache* para um determinado programa. O método utiliza informações coletadas, como o tamanho da *cache* e o prejuízo com as perdas da *cache*, para a escolha da otimização. O método avalia a relação entre os benefícios de remover instruções de chamada de função e o aumento da taxa de perdas da *cache*, introduzido pelo aumento de código.

Já o trabalho proposto por Ball [7], considera que chamadas de funções com argumentos constantes são melhores candidatas ao alinhamento, pois neste caso pode ser feita a remoção de código morto e algumas operações matemáticas em tempo de compilação.

Davidson e Holler [8, 9], por sua vez, especificaram uma implementação de um *function inliner* em linguagem C. Nela, as funções mais profundamente aninhadas são as primeiras a serem escolhidas. O programa termina quando não há mais registradores disponíveis para alocar as variáveis definidas pelo programador, podendo piorar a eficiência do programa. Os autores verificaram que os principais benefícios do alinhamento de funções vêm da remoção de:

- Instruções `CALL` e `RET`;
- Movimentação dos parâmetros;
- Ajustes da pilha;

- Registradores `SAVE` e `RESTORE` da função chamada

Outros trabalhos importantes são os de Steenkiste [10], Hwu e Chang [11, 12]. Steenkiste implementou um programa que faz alinhamento de funções em procedimentos menores que um tamanho limite. Conforme esse limite é aumentado, o tamanho do código cresce exponencialmente e o número de instruções executadas descresce logaritimicamente. Combinando esses dois efeitos, o ganho de desempenho atingido foi de 4%.

Hwu e Chang implementaram um programa que faz alinhamento de funções em ordem descrescente do número de vezes que as funções do programa foram executadas, até que o tamanho do código atinja um limite estabelecido. Essa técnica resultou em uma redução de 59% de funções chamadas para um aumento de código de 17%.

Com a aplicação dessas técnicas, muitas outras otimizações são também habilitadas. Dentre elas inclui-se a alocação de registradores, agendamento de instruções, propagação de constantes e eliminação de sub-expressões. Richarson e Ganapathi mostram os efeitos da expansão através do alinhamento de funções e a otimização entre procedimentos [1].

2.2.2 *Peephole*

Peephole é uma técnica de otimização realizada em nível de código intermediário, código *assembly* e código de máquina. Ela substitui uma sequência de instruções consecutivas por outra semanticamente equivalente, porém mais eficiente [13]. Em geral, ela é aplicada em código de máquina, sendo o último passo da compilação, e procura explorar as particularidades da arquitetura alvo.

Grande parte dos trabalhos sobre *peephole* utiliza algoritmos de casamento de padrões (*pattern matching*), que transformam um conjunto de instruções em uma *string* e então comparam a *string* obtida com a expressão regular que representa o padrão a ser otimizado. Quando o padrão é encontrado, ele é substituído por outra *string* mais eficiente. Um trabalho que utiliza essa técnica é o de Spinellis [14], que realiza a otimização de instruções com desvio condicional dentro de laços.

Um outro trabalho na área, desenvolvido por Bansal e Aiken [15], propõe uma nova abordagem sobre otimizadores *peephole*. Segundo eles, os otimizadores existentes usam

regras de casamento de padrão previamente escritos por algum especialista na área, o que limita uma exploração mais sistemática de todas as oportunidades de otimização. Com base nisso, eles desenvolveram um otimizador capaz de aprender novas regras de otimização. Para isso, para cada padrão, o otimizador testa várias substituições possíveis, analisando com qual delas obtem-se um melhor desempenho.

A substituição de determinadas seqüências por outras equivalentes, porém, mais eficientes, é feita através de um conjunto de técnicas de otimização. Neste projeto aplicamos apenas duas delas e após o alinhamento de funções. Essas otimizações são: propagação de constantes e remoção de `LOAD/RESTORE` redundantes.

2.2.3 Propagação de constantes

As várias possibilidades de aplicação de propagação de constantes entre procedimentos são talvez o motivo principal para fazer otimizações em tempo de execução. Ferramentas como ALTO [16] e a sua correspondente para a arquitetura x86/IA-32 PLTO [17] mostraram que a propagação de constantes entre procedimentos atinge um ótimo ganho de desempenho em relação às demais otimizações. O projeto ALTO, que consiste em um otimizador atuando em tempo de ligação dos códigos objetos da aplicação, voltado para a arquitetura Compaq/ALPHA, mostrou que a propagação interprocedural de constantes fez os programas utilizados como *benchmarks* executarem, em média, 10% mais rapidamente.

No projeto PLTO, a propagação de constantes apresenta os seguintes objetivos:

- Diminuir o número de instruções `LOAD/RESTORE`, ao propagar argumentos constantes de uma função chamada para dentro de seu corpo. Essa diminuição torna-se possível devido à substituição de instruções que referenciam os argumentos localizados na pilha - que é uma estrutura armazenada em memória - por outras equivalentes que usam os valores constantes.

- Eliminar saltos condicionais, quando a instrução de salto referencia um dos valores constantes propagados.

Outro trabalho, desenvolvido por Laurenz [18], define um algoritmo que efetua a propaga-

ção de constantes em um simples procedimento. Ele identifica os valores que são constantes em todas as possíveis execuções, para então propagá-los pelo código. Sua técnica é avaliar, em tempo de compilação, todas as expressões cujos operandos são todos constantes, propagando posteriormente pelo código os resultados possíveis de serem calculados. Esse procedimento melhora consideravelmente o desempenho quando as expressões avaliadas em tempo de execução encontram-se em blocos de repetição.

Triantafyllis [19] propôs um *framework* de compilação com o objetivo de superar as limitações de compiladores tradicionais que realizam otimizações baseadas em procedimentos. Seu trabalho enfatiza a análise e otimização interprocedural e utiliza uma técnica chamada *Procedure Boundary Elimination*. Ela basicamente unifica os procedimentos existentes no código por meio da união dos grafos de fluxo de execução individuais, aumentando o escopo da otimização sem aumentar o tamanho do código, como no alinhamento de funções. Com a unificação de procedimentos, torna-se possível propagar os valores constantes usados como argumentos nas chamadas de função. Esta técnica permitiu alcançar ganhos de desempenho semelhantes à aplicação do alinhamento de funções e gerar códigos menores, porém com um tempo de compilação consideravelmente maior.

2.2.4 Remoção de `LOAD/RESTORE` redundantes

Uma das otimizações mais importantes realizadas com o uso de *peephole* é a remoção de instruções de `LOAD/RESTORE` redundantes. Nesse caso, procura-se encontrar múltiplas instruções `LOAD` de uma mesma localização e substituí-las por instruções `MOV` entre registradores. A idéia é encontrar duas instruções, `I` e `J`, que carregam em diferentes registradores a partir de uma mesma posição de memória. Para isso é preciso provar que a posição de memória não foi alterada entre a execução de `I` e `J`, e que o registrador carregado por `I` ainda contém o valor a ser passado para `J`. O ato de substituir instruções de acesso à memória por instruções de troca entre registradores é semelhante ao ato de propagar constantes no lugar de registradores ou regiões de memória durante a propagação de constantes.

2.2.5 Desdobramento de laços

Uma outra técnica de otimização bastante eficiente para aplicação em laços gargalos é o desdobramento de laços (*loop unrolling*). O desdobramento de laços é uma técnica que busca diminuir o número de iterações de um laço estendendo o código contido em seu corpo. Isso provoca a diminuição do *overhead* de controle do laço e permite escalonar instruções que originalmente pertenciam a iterações distintas. A restrição dessa técnica está no conhecimento do número de iterações do laço em tempo de compilação, bem como na disponibilidade de registradores para armazenar os resultados intermediários da iteração [20].

Como exemplo, considere o fragmento de código da figura 2.5. Nele são considerados dados do tipo *double* e latências arbitrárias para as operações, resultando em algumas paradas no *pipeline* representadas pelos *stalls*. Após a aplicação do desdobramento de laços, as paradas ocasionadas pelo código original foram eliminadas e as instruções de controle foram divididas por 3 (fator de desdobramento) [20].

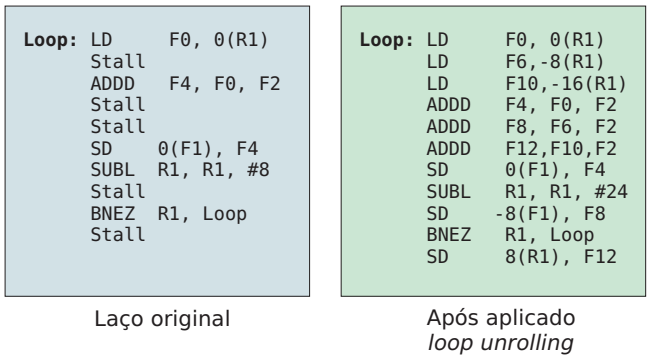


Figura 2.5: Exemplo da aplicação do desdobramento de laços

Nos últimos anos, várias técnicas de desdobramento de laços foram propostas [21–23]. Muitos estudos estão restritos em efetuar o desdobramento de laços contendo apenas um bloco básico e cujos contadores de iteração são facilmente determináveis em tempo de compilação. Estudos mais recentes mostram que esta abordagem perde muitas oportunidades para a criação de laços mais eficientes e apresentam resultados de análise do desdobramento de laços não apenas em tempo de compilação, mas também em tempo de execução [24].

No entanto, algumas perguntas como quando e onde fazer o desdobramento de laços

ainda permanecem. Dependendo da forma de aplicação, o desdobramento de laços pode ter um impacto contrário no desempenho, caso ocorra sobrecarregamento na *cache* de instruções. Esta degradação no desempenho depende do tamanho, organização e políticas de substituição da *cache*. Se todas as instruções no laço se ajustarem na *cache* de instruções, então nenhuma falha por falta de capacidade ocorrerá durante a execução deste laço. Ao contrário, se todas as instruções não se ajustarem, ou seja, o conjunto de trabalho for maior do que a *cache*, uma falha por falta de capacidade ocorrerá[24]. Para ter certeza que isso não ocorra, é necessário determinar o tamanho ótimo do código do laço desdobrado em termos das instruções de linguagem de máquina.

3 Detalhamento e desenvolvimento do projeto

O Paradyn, na versão utilizada como base para esse projeto, realiza a instrumentação de laços de repetição, permitindo um refinamento do tamanho de grão de medida (bloco mínimo analisado), que antes era o de funções [3]. Não se realiza, entretanto, qualquer tentativa de melhorar a eficiência dos laços identificados como gargalo. Isso implica na necessidade de o usuário identificar nos laços gargalos os fatores responsáveis pela degradação de desempenho e então alterar o código fonte do programa.

O projeto atual expande as capacidades do Paradyn, investigando as técnicas que podem ser aplicadas para otimizar os laços reconhecidos como gargalo e automatizando o processo de otimização.

Neste capítulo apresenta-se uma descrição do porte da instrumentação de laços da arquitetura Sparc para a arquitetura x86, bem como o detalhamento dos principais módulos de otimização desenvolvidos no projeto. Na seção 3.1 apresentam-se as etapas do porte de instrumentação. Na seção 3.2 descreve-se o módulo de otimização, detalhando quais técnicas foram implementadas e como elas atuam no Paradyn.

3.1 Porte da instrumentação de laços para o processador x86

Após o estudo inicial do Paradyn e da instrumentação de laços implementada para a arquitetura Sparc, implementou-se a mesma técnica de instrumentação tendo como alvo a arquitetura x86. Inicialmente será descrita a técnica de instrumentação de laços. Em

seguida, serão detalhadas as etapas do porte, bem como as implicações da mudança de arquitetura na decodificação de instruções, correção de deslocamento de instruções de salto e na alteração das instruções de salto. O processo de porte da instrumentação realizado pode ser dividido nas seguintes etapas:

1. Identificação das modificações no código original do Paradyn e dos trechos dependentes de arquitetura.
2. Decodificação e manipulação de instruções x86 através de métodos existentes no Paradyn.
3. Identificação dos pontos de instrumentação.
4. Correção do deslocamento de instruções de salto e verificação da necessidade de alteração do tamanho das instruções de salto.
5. Instrumentação de laços.
6. Medição dos laços instrumentados.

3.1.1 Instrumentação de laços de repetição

A etapa inicial da instrumentação consiste na identificação dos laços, que são caracterizados pela existência de uma instrução de salto para algum endereço anterior ao da própria instrução. Ao identificar um laço, no entanto, três casos podem ser observados, como mostrado na figura 3.1. No laço simples, o corpo do laço é delimitado pelo endereço da instrução de salto e o endereço de destino do salto. Os laços entrelaçados ocorrem quando um laço não está totalmente contido dentro de outro, mas há uma intersecção entre eles. Se um laço está contido dentro de outro, então ele é chamado de laço aninhado.

As informações sobre os laços identificados na função gargalo são armazenadas em uma estrutura de árvore binária, em razão da organização hierárquica dos laços dentro do programa. Para cada laço são determinados os seus pontos de instrumentação, que podem ser:

- Extremidades do laços (entrada e saída de laço);
- Possíveis entradas e saídas adicionais;
- Chamadas de funções

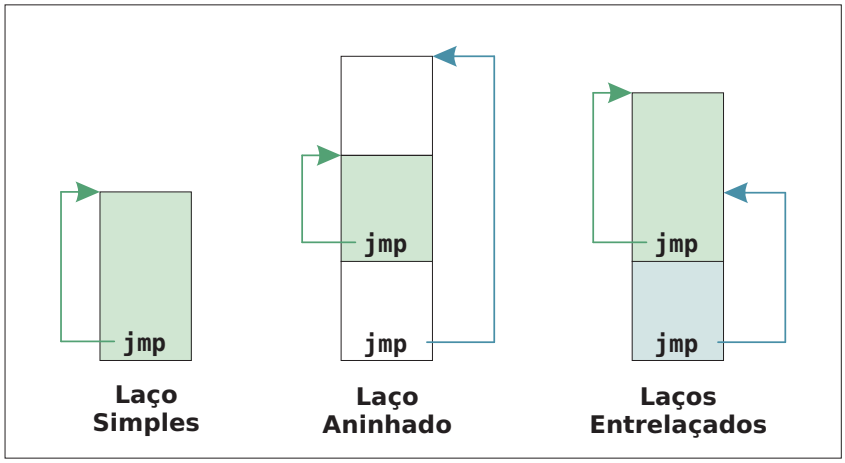


Figura 3.1: Tipos de laços encontrados na instrumentação

Na entrada do laço é inserida uma instrumentação para iniciar a medição do laço (em termos de tempo de execução). Na outra extremidade, isto é, na saída do laço é inserida uma instrumentação para encerrar a medição.

As entradas e saídas adicionais são ilustradas na figura 3.2 e referem-se a saltos para dentro e para fora do laço, respectivamente. Elas podem implicar na existência de laços entrelaçados, sendo tratadas como tais. No caso em que uma entrada ou saída adicional é um salto para frente, ela é armazenada em uma lista de saltos para frente. Antes de uma entrada adicional, insere-se instrumentação para iniciar ou continuar a medição do laço. Analogamente, antes de uma saída adicional, insere-se instrumentação para parar ou encerrrar a medição.

Além dos saltos, a execução do programa pode ser desviada do corpo do laço analisado através de chamadas de função. Antes de uma chamada `CALL`, é inserida uma instrumentação para parar a medição do laço e, assim, não considerar o tempo de execução da função chamada. Após a instrução `CALL`, uma nova instrumentação reinicia a medição do laço.

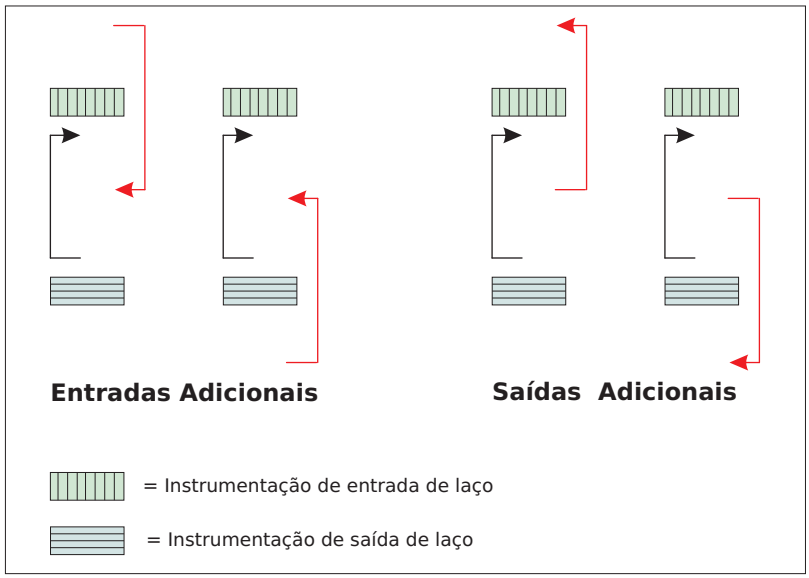


Figura 3.2: Entradas e saídas adicionais de laços de repetição

Após a definição dos pontos de instrumentação do laço, cria-se um vetor (armazenado em uma área de memória compartilhada entre o Paradyn e a aplicação) contendo todo o código da função mais o laço com as instrumentações, com os deslocamentos das instruções de salto e chamadas de função devidamente corrigidos. Em seguida, cria-se um trampolim da função original da aplicação para o novo código instrumentado. Após a realização das medições, o código original da função é restaurado. Para cada laço da função, cria-se uma nova função relocada com as instrumentações.

Nas seções seguintes, todo esse processo será descrito mais detalhadamente, abordando os aspectos de arquitetura x86 envolvidos na instrumentação.

3.1.2 Identificação das modificações no código original do Paradyn e de trechos dependentes de arquitetura

Primeiramente identificaram-se todas as inclusões e alterações efetuadas no código original do Paradyn pelo projeto anterior. Essas modificações e inclusões foram adicionadas ao código do Paradyn, sendo necessário alterar apenas os trechos de código que eram específicos da arquitetura Sparc, como máscaras para identificação de instruções, métodos para geração do código instrumentado, entre outros, de modo a viabilizar sua compilação

e execução na arquitetura x86.

As máscaras de identificação são aplicadas ao conjunto de *bytes* referentes a uma instrução, revelando padrões que permitem identificar os tipos de instruções de interesse (salto, chamada de função, retorno de função, entre outros). Uma vez que as tarefas de identificação de laços e definição dos pontos de instrumentação são feitas em código de máquina, as máscaras de instruções são totalmente dependentes da arquitetura.

Como as arquiteturas Sparc e x86 especificam diferentes formatos de instrução, precisou-se também alterar o modo como os métodos identificavam e manipulavam as instruções de máquina, o que consequentemente afetou os métodos responsáveis pela geração de código instrumentado e pela correção de deslocamento de instruções de salto e de chamadas de função.

3.1.3 Decodificação e manipulação de instruções x86

O Paradyn possui um conjunto de métodos que permitem obter, em código de máquina, todos os *bytes* de uma função sem nenhuma instrumentação por ele efetuada. A partir desses métodos, é possível construir uma estrutura que contenha todas as instruções da função, cujos laços de repetição serão analisados.

Na arquitetura Sparc, todas as instruções são de tamanho fixo (4 *bytes*), sendo necessário conhecer apenas os endereços inicial e final para obter suas instruções, percorrendo esse espaço de endereços de 4 em 4 *bytes*. Em x86, esse procedimento não é viável, visto que as instruções têm tamanho variável, podendo ter de 1 a 15 *bytes*.

Para obter as instruções x86, utilizou-se a classe `instruction`, que possui métodos que analisam sintaticamente e reconhecem instruções, seu tipo (chamada de procedimento, salto, retorno de procedimento, entre outros) e tamanho em *bytes*, a partir da posição inicial em memória. Portanto, diferentemente da implementação voltada para a arquitetura Sparc, utilizou-se essa classe para resolver o problema provocado por instruções de tamanhos distintos, evitando uma implementação extra para a identificação e armazenamento de instruções x86.

3.1.4 Identificação dos pontos de instrumentação

O algoritmo para a identificação dos pontos de instrumentação não sofreu grandes mudanças. Esse procedimento é realizado pelo método `defInstPoint` da classe `loopAnalyzer`.

Este método identifica quais pontos são entrada ou saída de um laço ou função. Identifica ainda entradas e saídas adicionais de laços. Essas informações são armazenadas em listas que permitem definir a quantidade de instrumentações de entrada e saída até a posição da n-ésima instrução da função.

A figura 3.3 ilustra os pontos de instrumentação definidos para um laço de repetição em uma função gargalo.

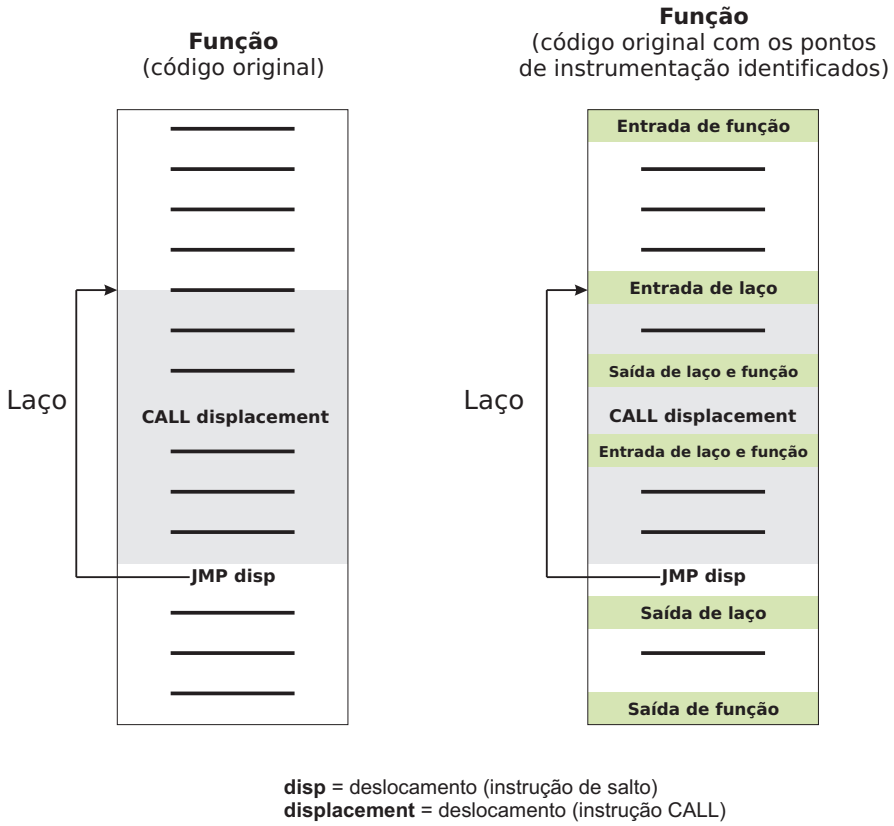


Figura 3.3: Identificação dos pontos de instrumentação

3.1.5 Correção de deslocamento e tamanho da instrução de salto

Uma vez identificados os pontos de instrumentação para todos os laços da função gargalo, é preciso tratar o problema provocado pela inserção de instrumentações no código original. Esse problema consiste na necessidade de se alterar o deslocamento das instruções de salto e de chamadas de função. O alvo dessas instruções é calculado com base em três fatores: deslocamento (um operando imediato da instrução), contador de programa e tamanho da instrução. A figura 3.4 mostra como o alvo de uma instrução de salto é calculado.

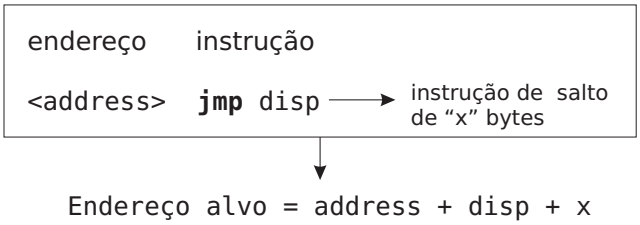


Figura 3.4: Cálculo do endereço alvo de uma instrução de saltos

Assim, toda instrução de salto deverá ter seu valor de deslocamento corrigido, caso haja instrumentações no intervalo do salto, sendo necessário incrementar esse valor para que o salto seja feito para a mesma instrução que faria originalmente. No caso do salto ser realizado no interior do laço, com um endereço de destino externo ao espaço do bloco de repetição, seu deslocamento deverá ser corrigido, de forma que o seu destino seja uma instrumentação de saída de laço.

Em Sparc a correção de deslocamento não afeta o formato nem o tamanho da instrução de salto, tornando sua implementação mais simples, como mostrado na figura 3.5.

Em x86, no entanto, há instruções de salto de 2 a 6 *bytes*, utilizadas de acordo com o valor de deslocamento. Logo, ao se alterar o deslocamento da instrução, deve-se avaliar se ela é capaz de armazenar o novo valor de deslocamento. Caso for capaz, o código de operação (*opcode*) da instrução é mantido, alterando-se apenas o deslocamento. Caso contrário, identifica-se uma instrução de salto equivalente (que executa o mesmo teste condicional), mas com maior espaço para o deslocamento, para substituí-la. Essa alteração da instrução de salto provoca um aumento no tamanho do código entre a origem e o destino

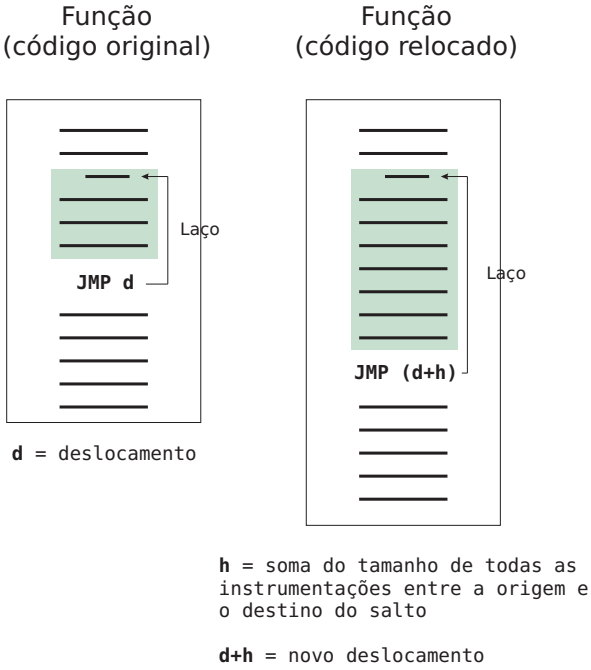


Figura 3.5: Correção de deslocamento de salto na arquitetura Sparc

de um salto. Logo, uma nova correção de deslocamento precisou ser feita, considerando as instruções de salto que foram substituídas entre esses dois pontos. A figura 3.6 exemplifica essa nova correção de deslocamento, com modificação do *opcode* da instrução de salto.

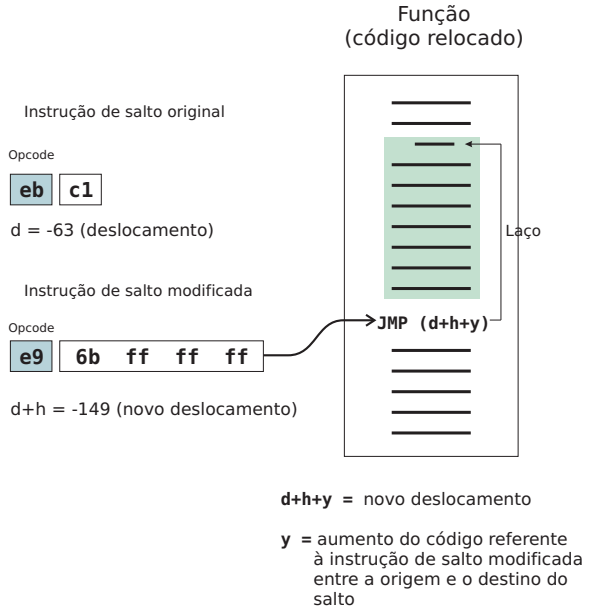


Figura 3.6: Correção de deslocamento na arquitetura x86

Instruções de chamada de função (CALL) também utilizam um valor de deslocamento para calcular o endereço do seu alvo. Logo, deve-se corrigir o deslocamento das instruções CALL da função gargalo, caso instrumentações sejam inseridas no intervalo entre a instrução CALL e o seu alvo. O pseudocódigo da correção de deslocamento da instrução CALL, ilustrado na figura 3.7, mostra que a correção depende basicamente da relação entre os endereços da instrução de chamada de função e de seu alvo.

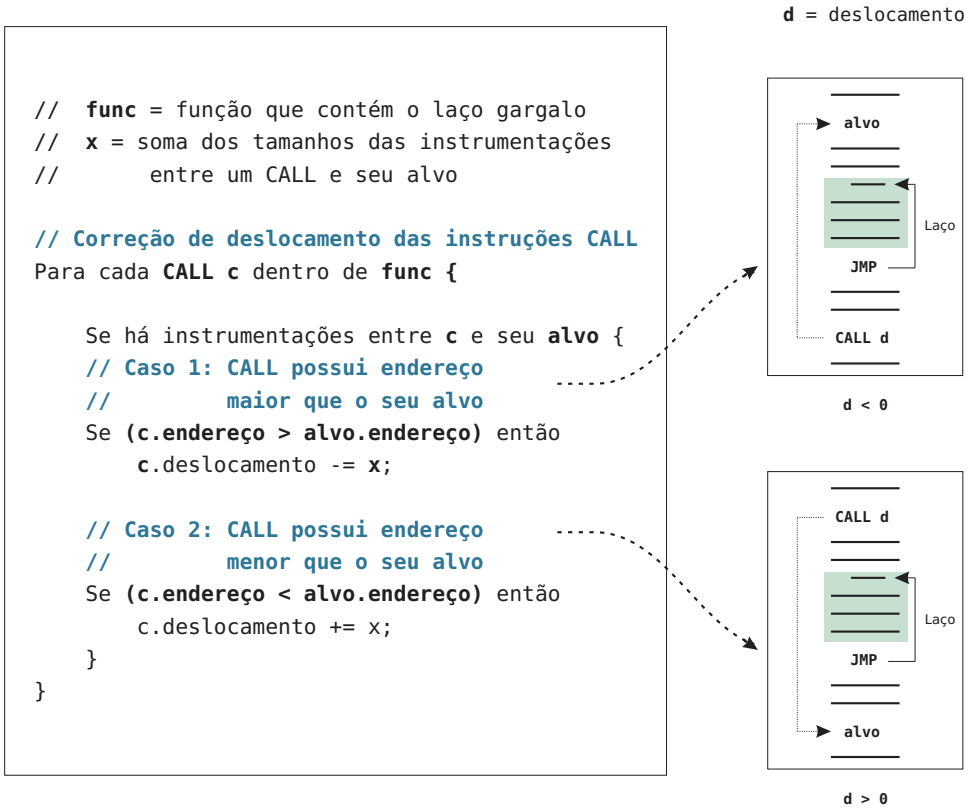


Figura 3.7: Pseudocódigo da correção de deslocamento da instrução CALL

3.1.6 Instrumentação de laços

O método de instrumentação de laços contém a maioria do código dependente de arquitetura e é responsável pela instrumentação de laços e armazenamento do novo código da função em um vetor de instruções (objetos da classe `instruction`). Neste ponto, todas as formas de instrumentação em Sparc são convertidas para as correspondentes em x86. Identificamos também um caso não tratado anteriormente, que é o de saltos para chamadas

de função dentro de um laço. A implementação anterior não efetuava a correta correção de deslocamento do salto, que apontava diretamente para a instrução de chamada de função (`CALL`). Na nova correção, a instrução de salto passou a apontar para as instrumentações de saída de laço e de função, conforme pode ser visto na figura 3.8, interrompendo o contador de tempo do laço, que é novamente ativado pelas instrumentações de entrada de laço e função após o retorno da função chamada.

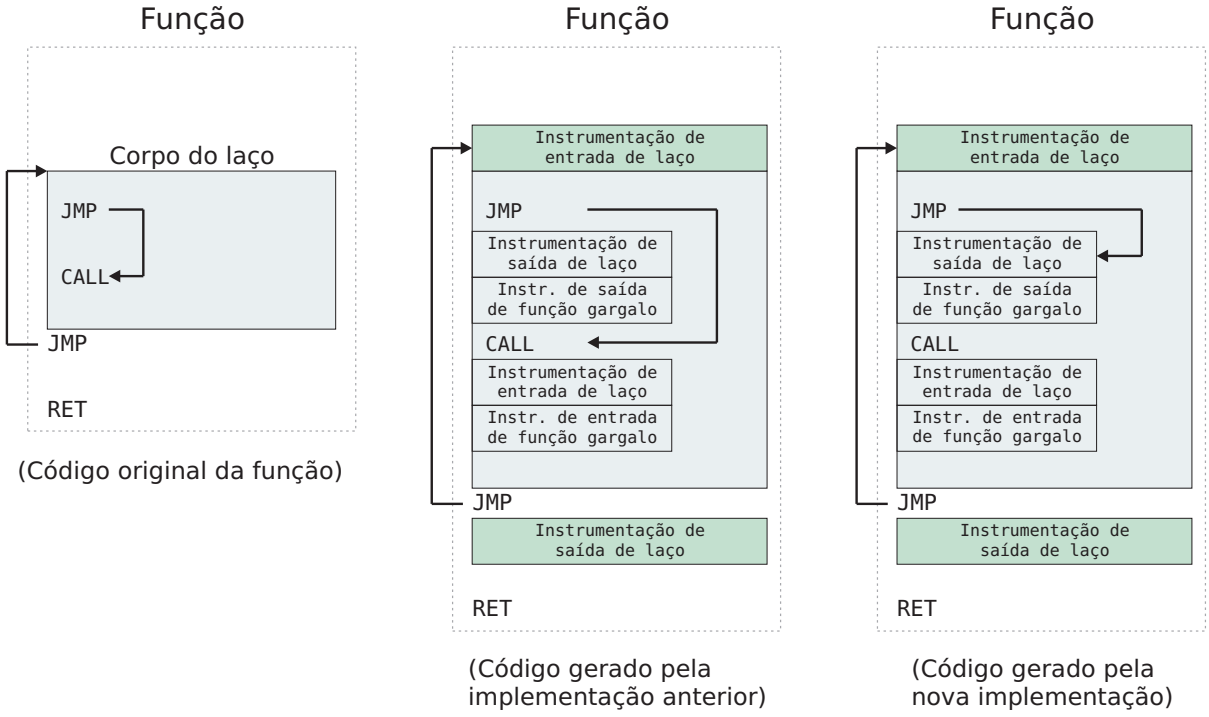


Figura 3.8: Ilustração do tratamento do caso não previsto no projeto anterior

Uma vez terminado o processo de criação do novo código com as instrumentações adequadas, cria-se um vetor de *bytes* contendo a função relocada. Esse novo vetor é armazenado em uma área de memória compartilhada entre o *daemon* do Paradyn e a aplicação. As primeiras instruções da função gargalo são então alteradas para saltar para o início da função relocada. Uma função relocada é criada para cada laço da função original. A figura 3.9 exemplifica o procedimento de relocação da função gargalo.

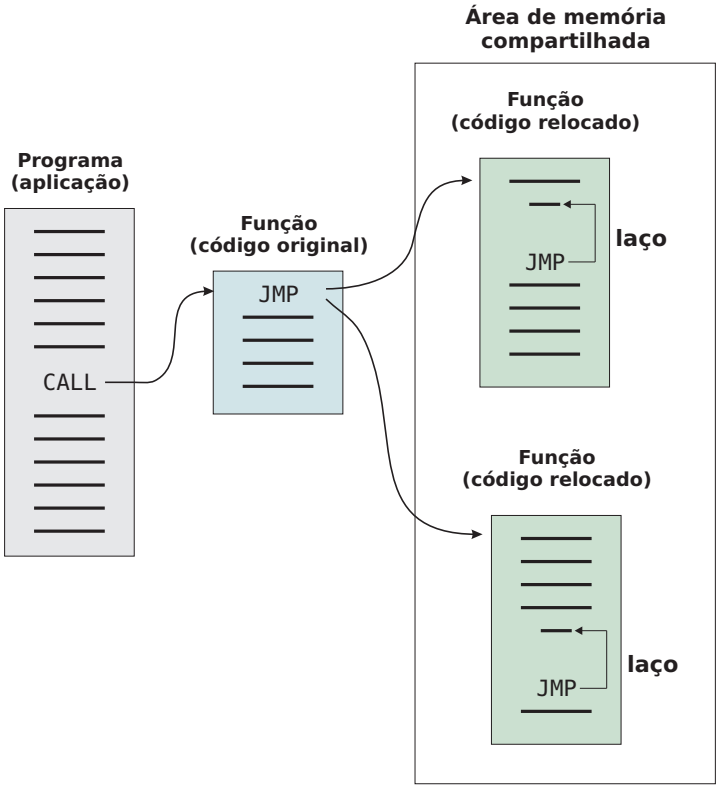


Figura 3.9: Relocação de código

3.1.7 Medição dos laços instrumentados

Após relocar uma função, o *daemon* do Paradyn é bloqueado temporariamente para que sejam feitas as medições do laço instrumentado, ou seja, para que haja tempo suficiente para que as medições sejam aferidas. Após ser desbloqueado, o *daemon* altera novamente as primeiras instruções da função original para apontar para uma nova função relocada, correspondente ao novo laço a ser analisado. Por meio das medições, é possível calcular o tempo de execução do laço em relação ao tempo total de execução da função. Com isso, pode-se classificar o laço como gargalo ou não.

3.2 Módulo otimizador

As otimizações dos laços de repetição identificados como gargalos são feitas através da aplicação de alinhamento de funções nas chamadas de função dentro desses laços, seguida

de otimizações como propagação de constantes e eliminação de instruções `LOAD` e `RESTORE` redundantes.

Estas três técnicas foram implementadas no projeto, automatizando a inserção dessas otimizações no código binário executado pela ferramenta. Elas foram escolhidas, dentre as várias técnicas investigadas, por questões de viabilidade de implementação.

O desdobramento de laços, por sua vez, não foi implementado. A verificação de dependência de dados entre as instruções de um laço, para que o desdobramento de laços seja aplicado de forma eficiente, mostrou ser uma tarefa muito complexa. Esta complexidade inviabilizou o desenvolvimento de um módulo de aplicação automática desta técnica.

A seguir estão descritos todas as etapas necessárias para a aplicação das três técnicas de otimização em um código de máquina e todas as correções necessárias a serem feitas no código da função com a aplicação dessas técnicas. A partir da identificação dessas etapas, um módulo de descompilação do bloco gargalo foi implementado, permitindo a criação de parâmetros que serão utilizados pelo otimizador. Esse processo é descrito para cada uma das técnicas e se encontra dentro dos tópicos correspondentes às otimizações, que são: alinhamento de funções (3.2.2), propagação de constantes (3.2.3) e eliminação de `LOAD/RESTORE` redundantes (3.2.4).

3.2.1 Considerações quanto ao processo de descompilação do bloco gargalo para a aplicação de otimizações

Após a etapa de identificação de laços gargalos, passamos para o procedimento de formulação dos parâmetros a serem passados para o otimizador. Estes parâmetros são construídos através de uma investigação no código da função, a partir do qual uma série de padrões são identificados. Eles mantêm informacões relevantes para a aplicação de otimizações em trechos específicos no código.

Uma vez que o alinhamento de uma função habilita as demais otimizações, os parâmetros a serem passados para a propagação de constantes e para a remoção de `LOAD/RESTORE` redundantes estão relacionados com as intruções `CALL` marcadas para o alinhamento. Esta relação é mantida em listas, onde um conjunto de informações, como o endereço de uma

instrução ou o nome da função alvo de uma instrução `CALL`, são utilizadas para guiar otimizações em trechos específicos. O uso de otimizações em locais específicos possibilita a verificação de desempenho das otimizações isoladamente, permitindo a aplicação no código binário apenas daquelas que trouxerem um ganho considerável de desempenho.

Para iniciar este procedimento é necessário delimitar a área em que as otimizações serão aplicadas. O primeiro passo é obter o endereços de inicio e fim de todos os laços identificados como gargalo. Este endereço é utilizado como a entrada do processo de descompilação do bloco, a partir do qual será construída a estrutura que guiará as otimizações.

Esta etapa consiste em identificar, através do uso de máscaras de instruções e de recursos do próprio Paradyn, instruções que serão utilizadas e modificadas durante o processo de otimização. Essas informações são armazenadas em listas que consistem, basicamente, em objetos que contém informações isoladas para cada instrução `CALL` do laço gargalo. Juntamente com as informações das instruções `CALL`, informações adicionais, para que correções sejam feitas, são adicionadas. Os procedimentos referentes à cada otimização são mostrados nos tópicos de construção de parâmetros. A figura 3.10 ilustra o procedimento de relocação de função que é feito após as otimizações serem aplicadas em pontos específicos do código da função gargalo.

As informações nas listas que identificam uma determinada instrução é ou o seu endereço ou a sua posição em relação ao inicio da função na qual ela pertence. Elas são armazenadas dessa forma pois o Paradyn possui uma classe que contém métodos que obtém todas as informacões de um instrução x86 a partir da sua posição ou endereço. Como esta classe é responsável por, a partir destas informações e do código de máquina da função, identificar todas as partes de uma instrução e permitir sua modificação através de uma forma fácil de manipular, não foi necessário obter informações específicas de cada instrução durante o processo de descompilação do bloco gargalo.

3.2.2 Alinhamento de funções

As transformações do alinhamento de funções são implementadas para produzir um código muito mais eficiente, o que é feito com a substituição da chamada `CALL` no laço da função gargalo com uma instância da própria função. Este objetivo é alcançado através de

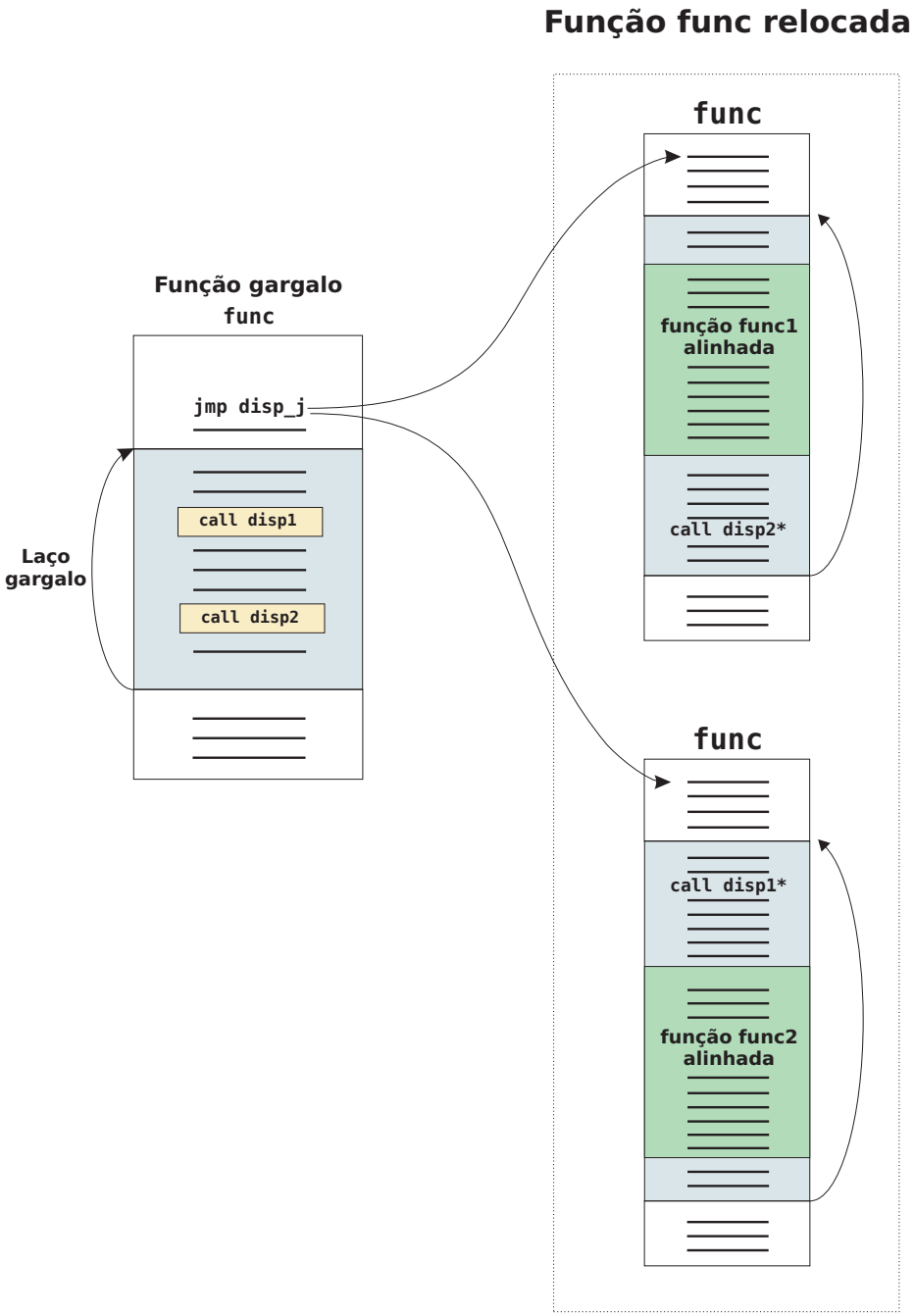


Figura 3.10: Relocação da função otimizada em trechos específicos

quatro etapas.

A primeira simplesmente aplica a função da forma como está, substituindo a instrução `CALL` na função chamadora. Já a segunda etapa consiste em corrigir o deslocamento nas

instruções de salto e nas chamadas de função da função chamadora. Como o número de instruções na função chamadora aumenta com o alinhamento da função, as instruções de salto e as instruções de chamada de função devem ser reajustadas para apontarem para os mesmos locais de antes do alinhamento. A terceira etapa, por sua vez, efetua a correção de deslocamento nas chamadas de função da função chamada. Nesta etapa não é necessário corrigir o deslocamento dos saltos, pois os mesmo continuarão a apontar para os mesmos locais dentro da função. Por fim, a quarta etapa remove a instrução de retorno `RET` da função alinhada.

A figura 3.11 ilustra o alinhamento de uma função `g()` no corpo da função chamadora `f()` e as consequentes correções dos deslocamentos das instruções de salto e de chamada de função. Os deslocamentos são ajustados para que a instrução de salto tenha como alvo a mesma instrução de antes do alinhamento.

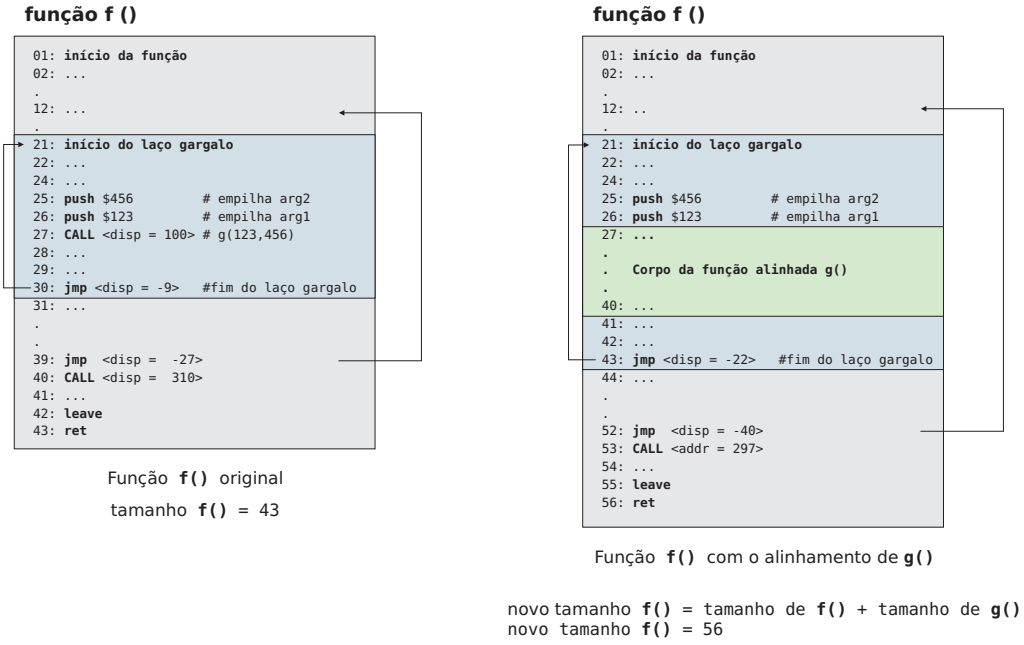


Figura 3.11: Exemplo da aplicação do alinhamento de função

A exemplo do que ocorre na inserção de instrumentações de laços, o alinhamento de funções provoca o aumento no código da função gargalo. Assim, deve-se relocar a função chamadora, contendo o código da função alinhada. Isso é tratado da mesma maneira que a instrumentação de laços: cria-se um trampolim da função original para a função relocada

em uma área de memória compartilhada entre o *daemon* do Paradyn e a aplicação.

Parâmetros para o alinhamento de funções

Como a primeira técnica a ser aplicada é o alinhamento de funções, esta etapa efetua, inicialmente, uma busca por instruções `CALL` no código do laço gargalo. Essa busca é feita através do uso da classe `instruction` do Paradyn, que possui métodos específicos para a identificação de instruções. Cada instrução, a partir da posição inicial do laço, é verificada através do método `isCall()` desta classe. Este processo continua até que se atinja o endereço final do laço para todos os laços gargalos do programa. Para cada instrução `CALL` encontrada, uma informação é armazenada em uma lista, chamada `listaCALL`, contendo algumas informações como o nome da função à qual instrução pertence, seu endereço, seu tamanho e o endereço da função alvo, calculado a partir de seu deslocamento (figura 3.12).

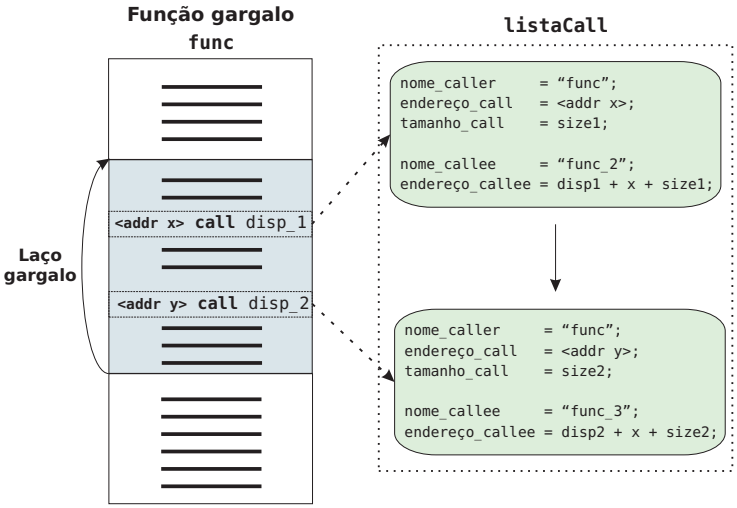


Figura 3.12: Construção da `listaCALL`

Em seguida, todas as instruções de salto da função são identificadas através do método `isJump()` e seus respectivos alvos são calculados. No caso de existir alguma instrução `CALL`, armazenada na `listaCALL`, entre a origem e o destino de um salto, esta é armazenada em uma nova lista, chamada `listaJUMPD`, indicando as instruções que deverão ter seu deslocamento corrigido. Caso contrário, a instrução de salto é descartada. Este processo é ilustrado na figura 3.13.

Por fim, as demais instruções `CALL` da função (aquelas que não estão na `listaCALL`)

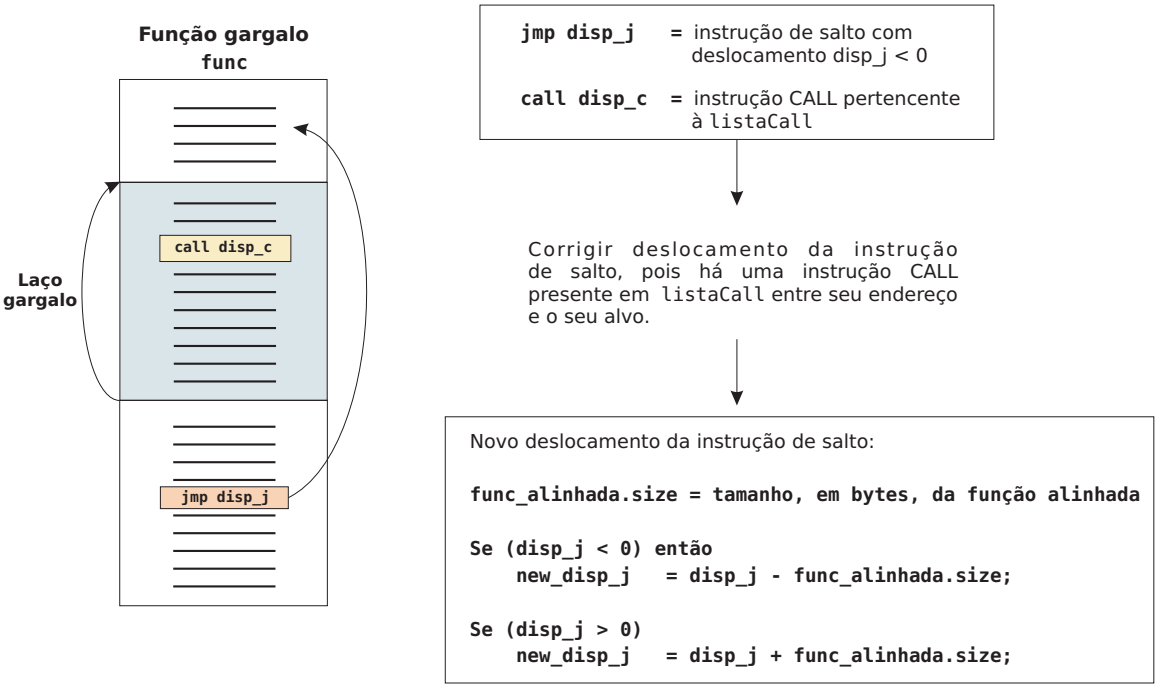


Figura 3.13: Correção de deslocamento em saltos após o alinhamento de uma função

são identificadas. No caso de existir alguma instrução `CALL`, armazenada na `listaCALL`, entre a origem e o destino dessas instruções, estas são armazenadas em uma nova lista, chamada `listaCALLD`, também indicando as instruções que deverão ter seu deslocamento corrigido. Esse processo é ilustrado na figura 3.14.

Porém, as listas de deslocamento de `CALL` e `JUMP` (`listaCALLD` e `listaJUMPD`) ficam armazenadas dentro da `listaCALL` para cada objeto, com as informações referentes às instruções `CALL` específicas. Essa estrutura possibilita a implementação de um módulo que efetua o alinhamento de funções em chamadas de funções específicas do laço gargalo, sem que haja a necessidade de obter informações e efetuar cálculos específicos de instruções.

3.2.3 Propagação de constantes

A propagação de constantes é realizada após o alinhamento de funções e visa propagar os valores constantes passados pela função chamadora como argumentos da função chamada. A propagação de constantes permite, assim, uma redução no número de acessos à pilha. Em outras palavras, essa técnica reduz o número de acessos à memória, pois troca

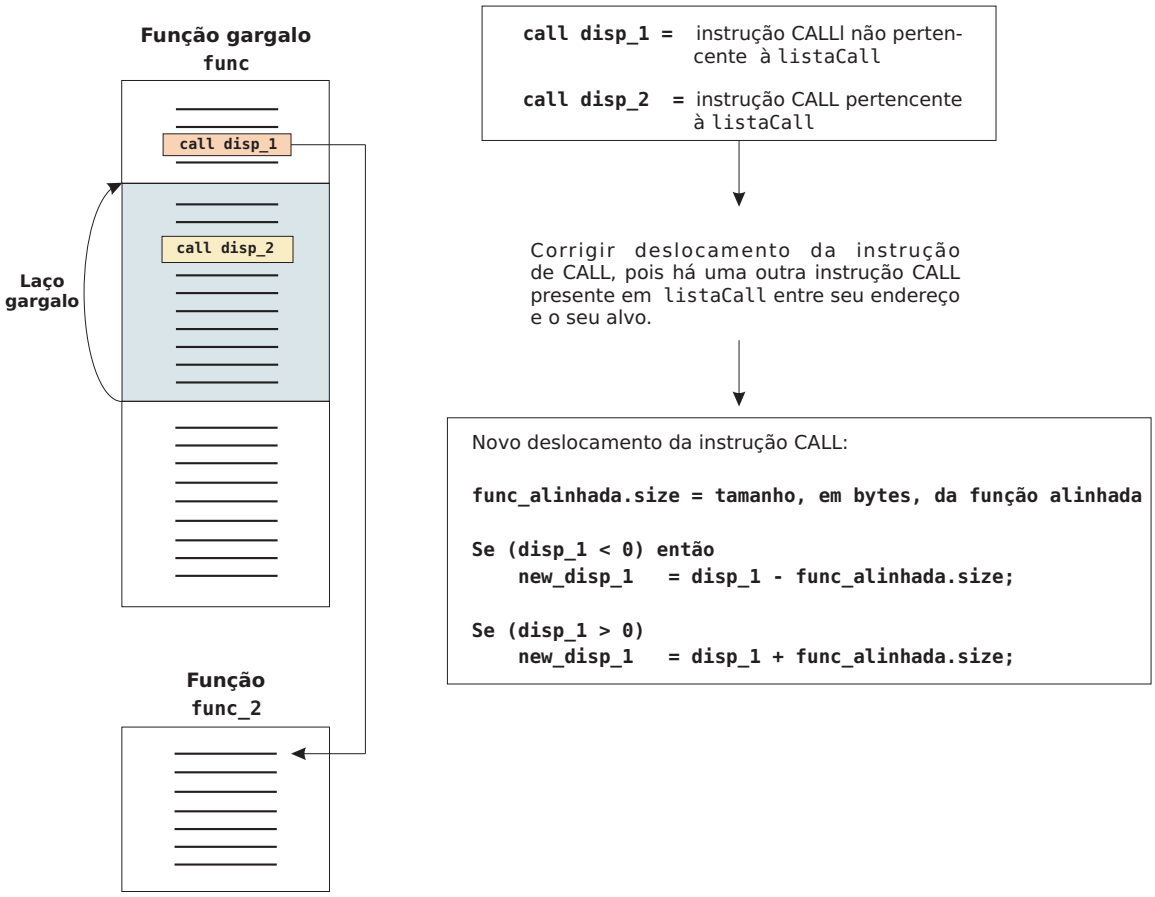


Figura 3.14: Correção de deslocamento em instruções `CALL` após o alinhamento de uma função

instruções que referenciam valores da pilha por outras com argumentos constantes.

A passagem de parâmetros para funções, na arquitetura x86, é feita por meio da pilha, que é uma estrutura de dados armazenada em memória. Todos os argumentos podem ser vistos, então, como valores em memória. Na função chamada, cada argumento pode ser usado por meio de duas formas distintas:

- Referência à posição da pilha em que o argumento se encontra.
- Copia-se o valor do argumento para um registrador de uso geral. A partir daí, todas as operações sobre o argumento serão realizadas utilizando-se esse registrador.

A figura 3.15 ilustra a passagem de argumentos e a manipulação deles por meio das

formas apresentadas anteriormente.

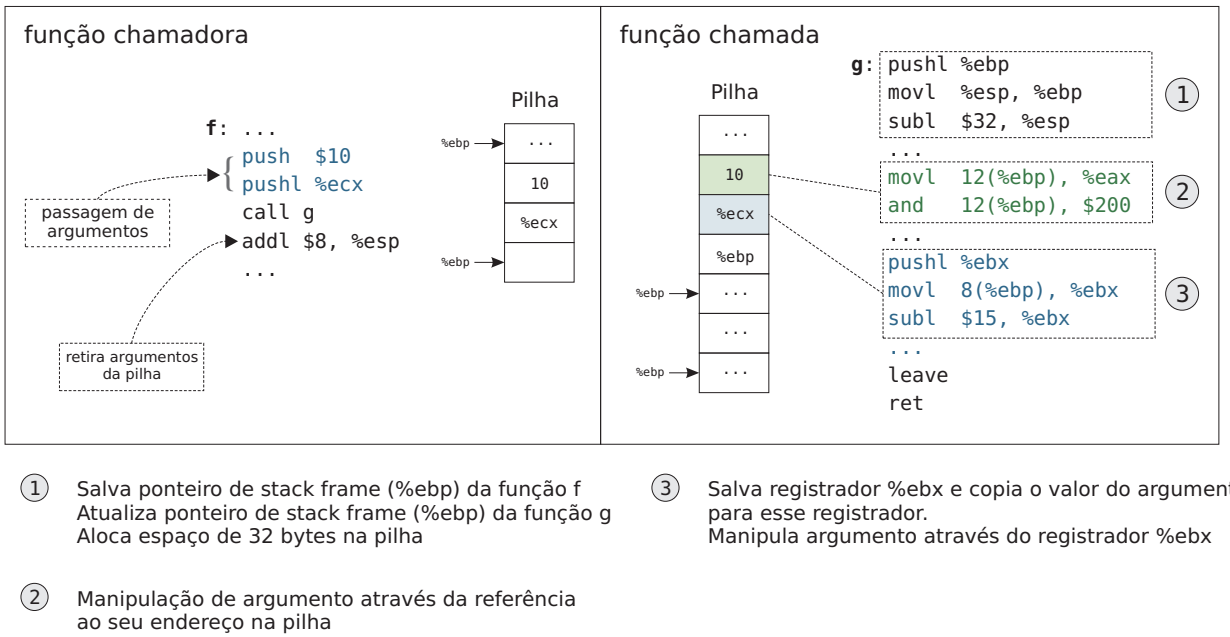


Figura 3.15: Exemplo de passagem de argumentos entre funções na arquitetura x86 e das formas de utilização

Independentemente da forma como o argumento é utilizado pela função chamada, é possível aplicar a propagação de constantes. Essa técnica foi implementada nesse projeto, sendo realizada em 3 etapas:

1. Identifica-se os argumentos passados pela função chamadora para a função chamada, e é verificado se esses valores são literais (constantes).

2. Para cada argumento constante, verifica-se se alguma instrução da função chamada o altera. Embora o argumento seja passado como um valor literal, alguma instrução pode modificá-lo.

3. Para os argumentos identificados como imutáveis, no corpo da função chamada, troca-se todas as referências a ele, por meio de seu endereço na pilha, pelo seu valor.

Deve-se ressaltar que um argumento passado para a função chamada, na forma de um valor literal, não necessariamente permanecerá constante ao longo do corpo da função

chamada. Pode haver alguma função que altere o conteúdo da posição da pilha onde o valor está armazenado.

A figura 3.16 mostra um exemplo com uma função `f()`, que chama uma função `g()`, passando argumentos constantes.

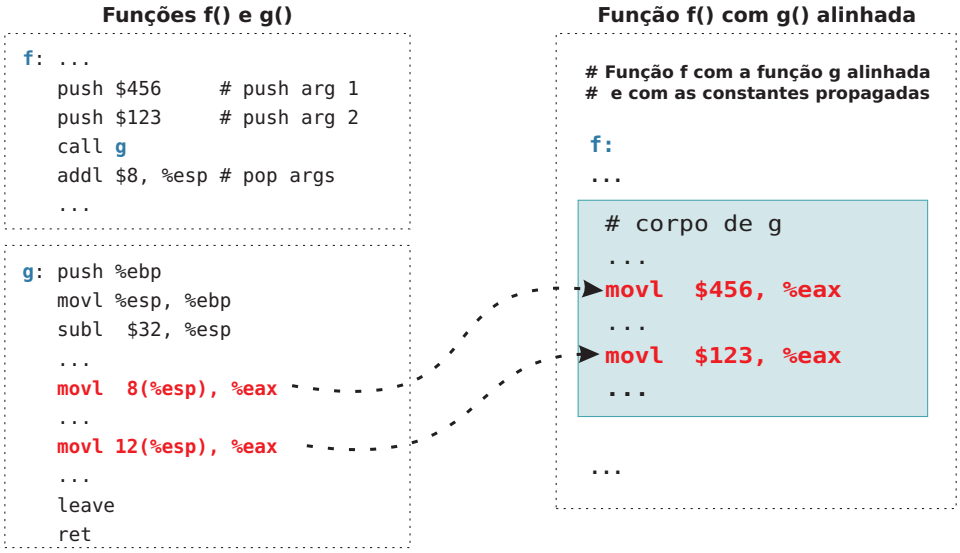


Figura 3.16: Aplicação da propagação de constantes após o alinhamento da função

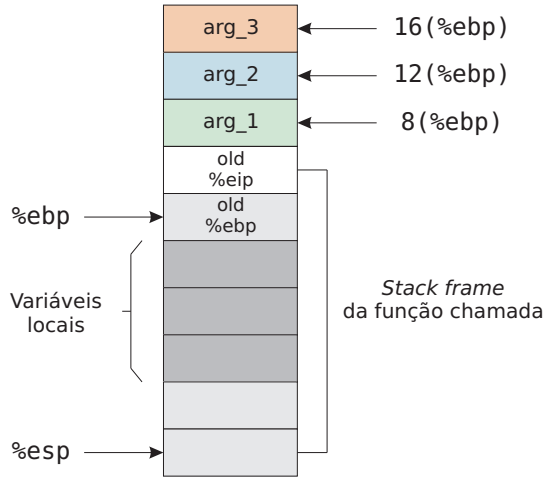
A propagação de constantes que foi implementada é limitada a apenas propagar constantes passadas como argumento da função chamada e que são utilizados por meio da referência à sua posicao na pilha. No caso dessas constantes serem passadas para um registrador e este continuar imutável, as constantes não serão novamente propagadas.

Parâmetros para a propagação de constantes

A identificação dos parâmetros para que a propagação de constantes seja aplicada está relacionada apenas com a função alinhada e com os argumentos que estão sendo passados para ela, uma vez que ela é aplicada após o alinhamento de funções. Esses parâmetros são utilizados pelo otimizador para identificar as instruções da função alinhada, que acessam os dados constantes na pilha, e colocar os argumentos constantes diretamente no código da função alinhada.

Esses argumentos são acessados dentro da função através do ponteiro base da pilha

(%ebp), sendo 8(%ebp) o primeiro argumento, 12(%ebp) o segundo argumento, 16(%ebp) o terceiro argumento, continuando assim, de 4 em 4 *bytes*, até o último argumento, como mostrado na figura 3.17.



16(%ebp)	argumento 3
12(%ebp)	argumento 2
8(%ebp)	argumento 1
4(%ebp)	old %eip = endereço de retorno
0(%ebp)	old %ebp = ponteiro base da função chamadora
-4(%ebp)	primeira variável local
-8(%ebp)	segunda variável local

Figura 3.17: Posição dos argumentos na pilha

Assim, todas as instruções dentro da função alinhada que fazem acesso à esses locais na pilha são identificadas, sendo armazenados em uma lista que contém a posição da instrução na função (`listaINSTR`). Da mesma maneira, as instruções da função chamadora, que estão imediatamente antes da instrução `CALL` e que são responsáveis por colocar os valores constantes, parâmetros da função chamada, na pilha, são também armazenadas. Essas instruções, por sua vez, estão armazenadas em uma lista distinta (`listaARG`).

Como a propagação de constantes está vinculada ao alinhamento de uma função, a `listaCALL` é modificada, mantendo cada uma dessas listas para um único objeto. A figura 3.18 mostra essas listas dentro de um objeto da `listaCALL` e a forma como elas são construídas.

Essa estrutura, assim como no alinhamento de funções, permite a aplicação da otimização em trechos específicos. Nela, o otimizador obterá os endereços das instruções que passam os argumentos para a função e das instruções que acessam os argumentos, pertencentes à função. A partir desses endereços, as instruções são acessadas através de métodos da classe `instruction`, pertencente ao Paradyn, podendo ser facilmente modificada.

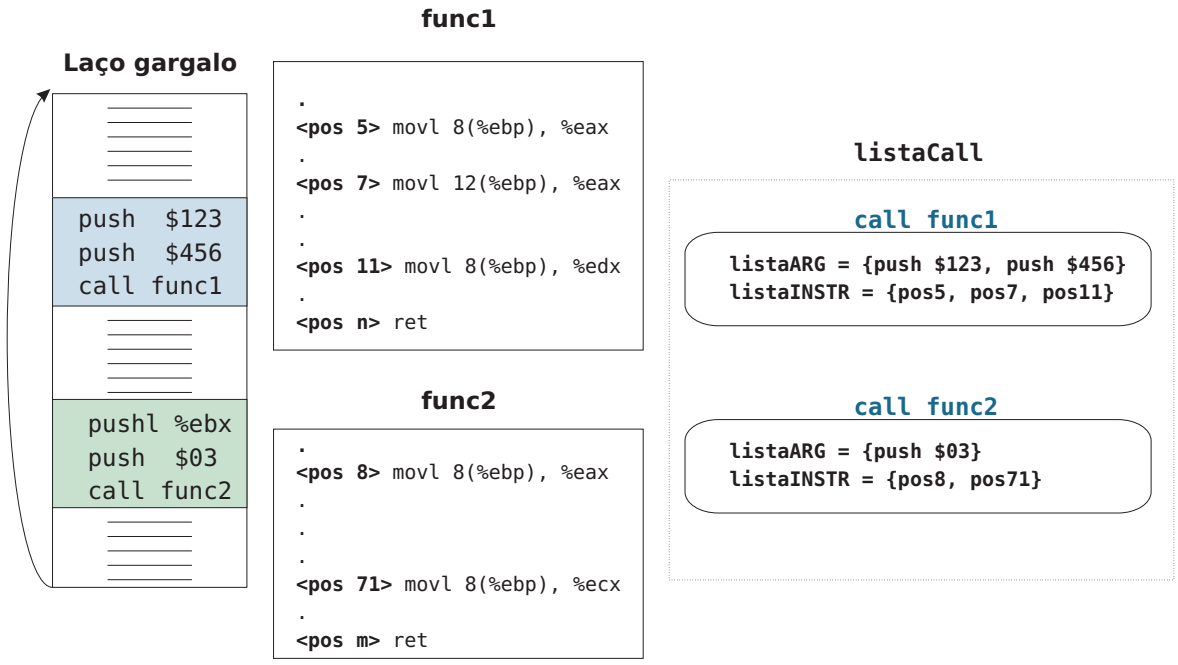


Figura 3.18: Construção da `listaINSTR` e da `listaARG` para cada objeto da `listaCALL`

3.2.4 Remoção de `LOAD/RESTORE` redundantes

A remoção de instruções `LOAD/RESTORE` redundantes é um tipo de otimização *peephole*, que é aplicada em código objeto. Um dos seus objetivos é identificar instruções `LOAD` de um valor em memória para um registrador, seguidas de uma instrução `RESTORE` desse registrador para o mesmo local em memória, sem que o valor inicialmente obtido da memória tenha sido alterado, para posteriormente eliminar a instrução `RESTORE`. Essa remoção pode ser realizada, visto que, nessas condições, o valor gravado pelo `RESTORE` é idêntico ao valor lido pelo `LOAD`, e a leitura e gravação são realizadas na mesma posição de memória.

Essa técnica evita, assim, que instruções `RESTORE` desnecessárias sejam executadas, minimizando o número de acessos à memória. A figura 3.19 mostra um exemplo de instruções `LOAD/RESTORE` redundantes e a figura 3.20 mostra a remoção da instrução `RESTORE` desnecessária.

No projeto, essa técnica segue a aplicação das técnicas de alinhamento de função e propagação de constantes. Ela é aplicada não apenas no laço gargalo, mas também em todo o resto do corpo da função. Para isso, são consideradas as seguintes etapas:

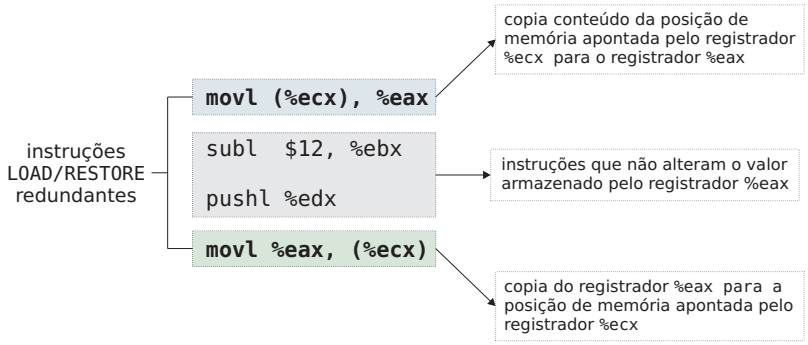


Figura 3.19: Exemplo de instruções LOAD/RESTORE redundantes

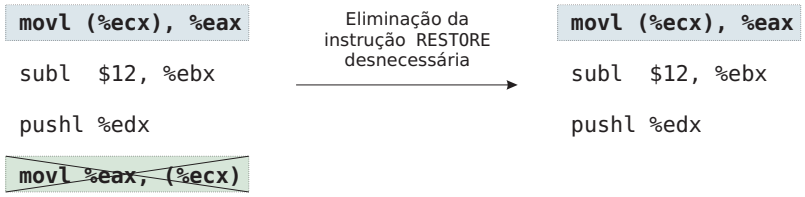


Figura 3.20: Exemplo de remoção de instruções LOAD/RESTORE redundantes

1. Para cada instrução do tipo LOAD na função gargalo, que grava um valor em um registrador reg, determina-se, caso exista, a posição da instrução RESTORE que grava o valor de reg na mesma posição de memória.

2. No intervalo entre essas duas instruções, é verificado se há alguma instrução que altere o valor do registrador reg. Se não houver, a instrução RESTORE é desnecessária, sendo eliminada.

O segundo objetivo dessa otimização é identificar todos os pontos do código que contém uma instruções LOAD seguida de uma instução RESTORE, no formato descrito na figura 3.21. Esta situação é muito comum de ocorrer em um código x86, pois trata-se da atribuição de uma variável por outra variável.

Neste caso, a redundância pode acontecer devido a possibilidade de existirem dois pares que utilizem a mesma posição de memória, sendo um na função chamadora e o outro na função chamada, que foi alinhada. Dessa forma, dois casos de redundância são possíveis. O primeiro é a ocorrência de uma atribuição $var_2 = var_1$, na função chamadora, seguida de uma atribuição $var_1 = var_2$, na função alinhada, sem que var_1 e var_2 sejam modificadas entre as duas atribuições. O segundo caso é a ocorrência de uma atribuição $var_1 = var_2$,

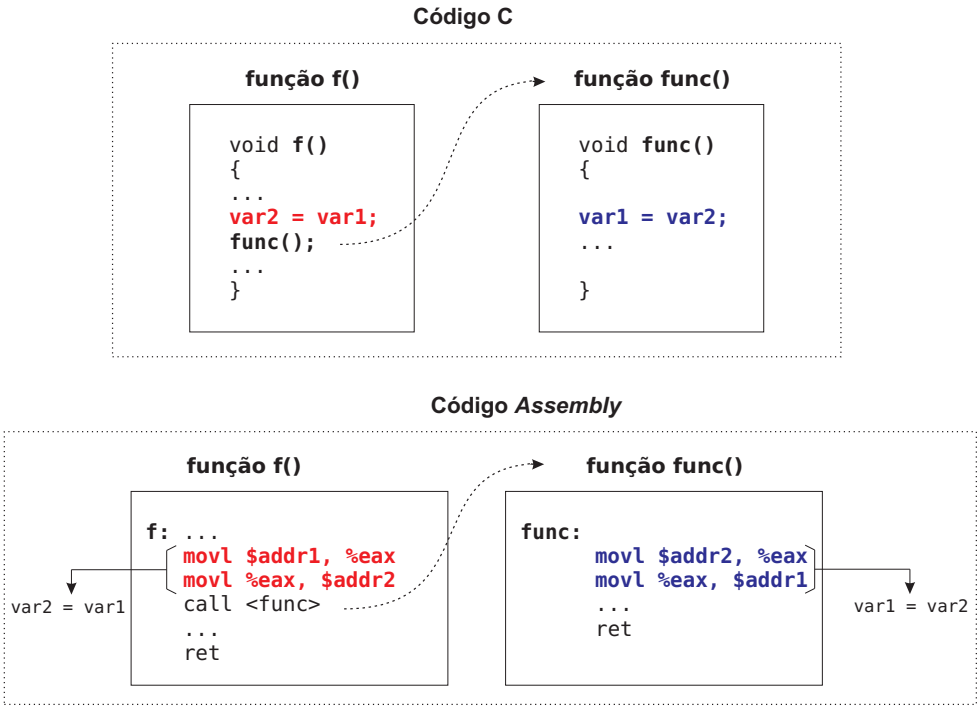


Figura 3.21: Instruções LOAD/RESTORE correspondentes à uma atribuição entre variáveis

na função chamadora, seguida de uma atribuição $var_1 = var_2$, na função alinhada, sem que var_1 e var_2 sejam modificadas entre as duas atribuições.

A figura 3.21 mostra a redundância com a ocorrência do primeiro caso antes do alinhamento e a figura 3.22 mostra a redundância com a ocorrência do segundo caso após o alinhamento.

A solução proposta é verificar se há instruções RESTORE entre os dois pares que modificam os conteúdos das posições de memória referenciadas pelas instruções do par. Se não houver, então o segundo par pode ser removido sem prejuízo à aplicação.

Parâmetros para a remoção de LOAD/RESTORE redundantes

A remoção de LOAD/RESTORE é aplicada em todo o código da função para identificar as possíveis redundâncias surgidas com o alinhamento de uma função. Para isso, é necessário identificar instruções que movem dados da memória para um registrador (LOAD) e de um registrador para a memória (RESTORE). Essa identificação é feita na função chamadora e

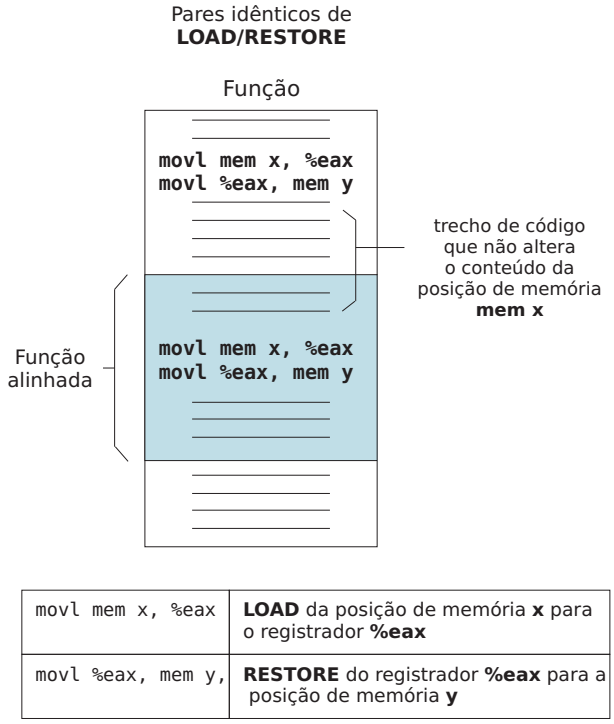


Figura 3.22: Redundância de instruções LOAD/RESTORE após o alinhamento

no bloco referente à função alinhada. Entretanto, como durante a fase de identificação dos parâmetros a função não está alinhada, é necessário identificar instruções de LOAD e RESTORE tanto nas funções que serão alinhadas quanto nas funções chamadoras.

A solução para este caso foi manter, para todas as funções do programa, uma lista independente da listaCALL que contém a posição das instruções LOAD e RESTORE para cada função. Essa estrutura é mantida através de uma nova lista de objetos que contêm o nome e uma lista com as posições das instruções LOAD e RESTORE da função (figura 3.23).

Assim, a remoção de LOAD/RESTORE redundantes é feita pelo otimizador após obter as instruções LOAD e RESTORE na função chamadora e na função alinhada. Como essa otimização é feita após o alinhamento de uma função específica, as informações da listaCALL são utilizadas para acessar os objetos corretos na listaLR, como mostrado na figura 3.23. Essas informações são o nome da função chamadora e o nome da função chamada. A partir dessas informações, todas as instruções de LOAD e RESTORE da função são identificadas, permitindo ao módulo otimizador verificar se há redundâncias e, no caso de existirem, removê-las do código da função.

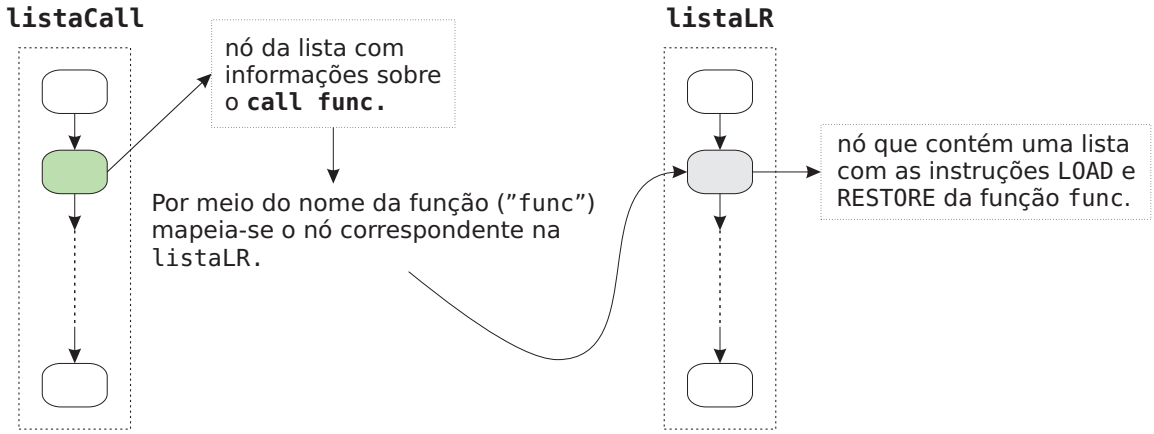


Figura 3.23: Armazenamento das instruções LOAD e RESTORE

3.3 Considerações Finais

Este capítulo apresentou as principais etapas e metodologias seguidas para o desenvolvimento do projeto, descrevendo os módulos e funcionalidades implementadas para a obtenção de um protótipo que atendesse as principais especificações do projeto.

Os testes realizados para a verificação da eficiência e corretude da instrumentação de laços e do processo de descompilação do bloco gargalo, bem como os resultados obtidos com os mesmos, encontram-se descritos no próximo capítulo.

4 Testes e Resultados

Para os testes, utilizamos alguns programas criados em C que fazem uso de bibliotecas MPI (*Message Passing Interface*). A seguir estão descrito os testes realizados, os programas utilizados em cada teste e seus respectivos resultados.

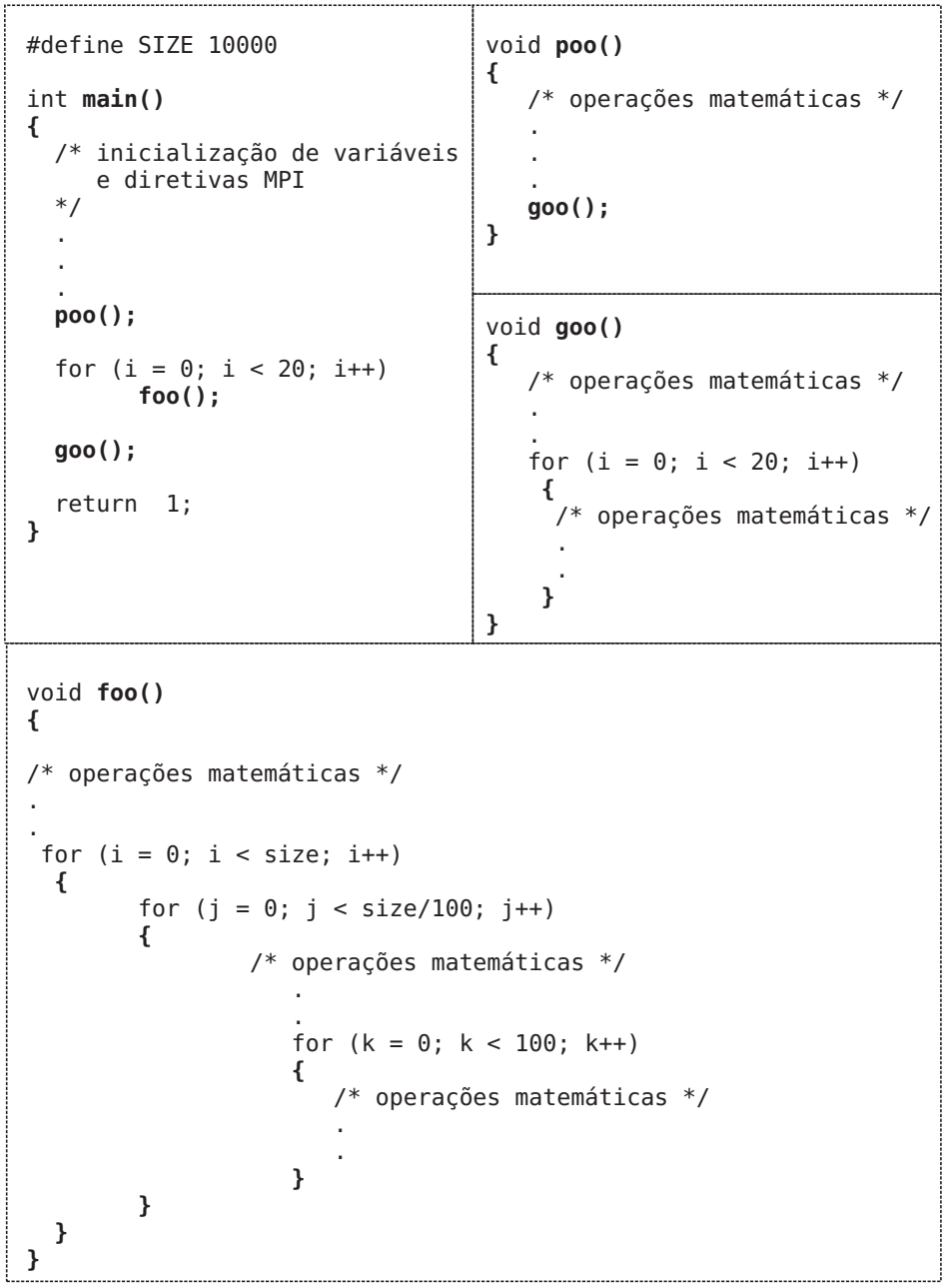
4.1 Instrumentação de laços

O programa utilizado para a realização do teste da instrumentação de laços está mostrado na figura 4.1.

Após a função `foo` ser reconhecida como gargalo, inicia a etapa de instrumentação de laços. Esta etapa inclui a modificação no processo original - geração da instrução de salto para a função relocada - e a restauração do processo modificado para o original. Para cada função relocada, o processo do Paradyn espera um tempo no qual são feitas as medições adequadas para cada laço.

Este programa faz chamada a três funções que foram inicialmente analisadas pelo Paradyn. Essas funções (`poo`, `goo` e `foo`) são chamadas pela função principal (`main`) na seguinte forma:

- Após as inicializações, a função `poo` é chamada uma única vez. Esta função acomoda um conjunto de instruções que efetuam cálculos matemáticos e uma chamada à função `goo`. Tais operações possuem um custo computacional relativamente baixo em relação às demais funções. Esta função não foi identificada como gargalo pelo Paradyn.
- A próxima função (`foo`) é mantida em um laço que itera vinte vezes. Tal função



```
#define SIZE 10000

int main()
{
  /* inicialização de variáveis
     e diretivas MPI
  */
  .
  .
  .
  poo();

  for (i = 0; i < 20; i++)
        foo();

  goo();

  return  1;
}
```

```
void poo()
{
   /* operações matemáticas */
   .
   .
   .
   goo();
}
```

```
void goo()
{
   /* operações matemáticas */
   .
   .
   for (i = 0; i < 20; i++)
    {
     /* operações matemáticas */
      .
      .
    }
}
```

```
void foo()
{

/* operações matemáticas */
.
.
 for (i = 0; i < size; i++)
  {
        for (j = 0; j < size/100; j++)
        {
                /* operações matemáticas */
                   .
                   .
                   for (k = 0; k < 100; k++)
                   {
                      /* operações matemáticas */
                      .
                      .
                   }
        }
  }
}
```

Figura 4.1: Ilustração do programa utilizado para teste.

possui um tempo de execução relativamente maior que as outras duas funções. Ela é composta de três laços aninhados efetuando um conjunto de operações matemáticas complexas. Foi a única identificada como gargalo pelo Paradyn.

- A última função (`goo`) é composta apenas de instruções que realizam cálculos mate-

máticos com um único laço de repetição. Seu custo computacional é maior que o da primeira função, porém muito menor que o da segunda. Também não foi identificada como gargalo pelo Paradyn.

A seguir, mostra-se os resultados obtidos para os três laços da função gargalo.

```
Nome da funcao: foo

Loop Endereco Inicial 402852ff
Loop Endereco Final 40285469
Loop Irmao -1
Loop Filho 1
TempoLoop 3692.000000
TempoFuncao 3796.000000
Gargalo = 1 ocupando 97.26%

Loop Endereco Inicial 40285314
Loop Endereco Final 4028545f
Loop Irmao -1
Loop Filho 2
TempoLoop 3486.000000
TempoFuncao 3744.000000
Gargalo = 1 ocupando 93.10%

Loop Endereco Inicial 4028541b
Loop Endereco Final 40285458
Loop Irmao -1
Loop Filho -1
TempoLoop 3260.000000
TempoFuncao 3731.000000
Gargalo = 1 ocupando 87.37%
```

Nesta função, todos os laços foram reconhecidos como gargalo.

4.2 Descompilação do bloco gargalo

A etapa de testes deste procedimento consistiu em verificar se as estruturas para as otimizações foram corretamente construídas. Ela foi realizada com um programa constituido de três funções: `main()`, `foo()` e `func()`. A função `main` chama a função `foo` que, por sua vez, chama a função `func`. As funções `foo` e `func` são ilustradas na figura 4.2.

```
void foo()
{

/* operações matemáticas */
.
.
 for (i = 0; i < size; i++)
  {
        for (j = 0; j < size/100; j++)
        {
                /* operações matemáticas */
                   .
                   .
                   for (k = 0; k < 100; k++)
                   {
                      /* operações matemáticas */
                      .
                      .
                      var_1 = 10;
                      var_2 = var_1;      <-- var_1 e var_2 são
                                              variáveis globais

                      func (1,2);
                   }
        }
  }

func_t();

}
```

```
void func (int a, int b)
{
var_2 = var_1;

/* operações matemáticas
   usando os argumentos "a" e "b"
*/
}
```

Figura 4.2: Código fonte das funções `foo` e `func`.

A função `foo` foi a única identificada como gargalo pelo Paradyn. Ela é composta de três laços, como ilustrado na figura 4.3, sendo que o laço mais interno foi o único identificado como gargalo, através da instrumentação de laços. Como este laço contém uma chamada de função, as otimizações alinhamento de função e propagação de constantes puderam ser

realizadas. Para a remoção de `LOAD/RESTORE` redundantes utilizamos as variáveis globais `var1` e `var2`. Como o par de instruções `LOAD/RESTORE` faz parte do escopo do laço gargalo e é repetido dentro da função alinhada, sem que `var1` e `var2` sejam alteradas entre os pares, o segundo par é marcado como redundante, sendo eliminado. Dessa forma, a remoção de `LOAD/RESTORE` redundante também pôde ser realizada.

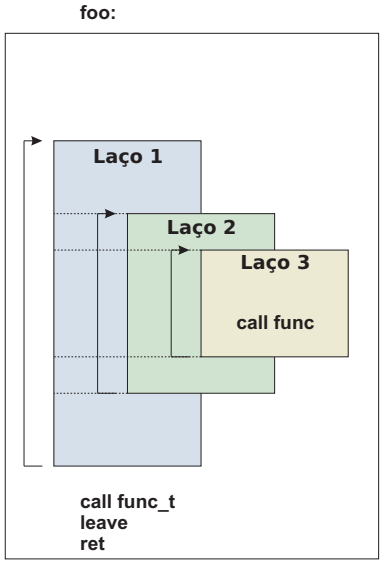


Figura 4.3: Estrutura simplificada da função `foo`.

A figura 4.4 mostra o código em *Assembly* da função `func`.

`func:`

Endereço	Código de máquina	Código *Assembly*
4029bf4c <func+0>	55	push %ebp
...	...	...
4029BF52 <func+6>	a1 1c ab 09 08	mov 0x809ab1c,%eax
4029BF57 <func+11>	a3 14 ab 09 08	mov %eax,0x809ab14
4029BF5c <func+16>	-	mov 0x8(%ebp),%eax
...	...	...
4029BF63 <func+23>	-	mov 0xc(%ebp),%eax
...	...	...
4029BF6a <func+30>	c9	leave
4029BF6b <func+31>	c3	ret

LOAD/RESTORE (listaLR)
%eax = var_1;
var_2 = %eax;

referência a argumento constante na pilha (listaINSTR)

referência a argumento constante na pilha (listaINSTR)

Figura 4.4: Código *Assembly* da função `func`.

Esta função, por ser chamada dentro do laço gargalo, será alinhada. Nesta figura são indicadas as instruções que realizam `LOAD/RESTORE`. Também são indicadas as instruções

de acesso aos argumentos constantes (via pilha).

O código em *Assembly* da função gargalo `foo` é mostrado na figura 4.5.

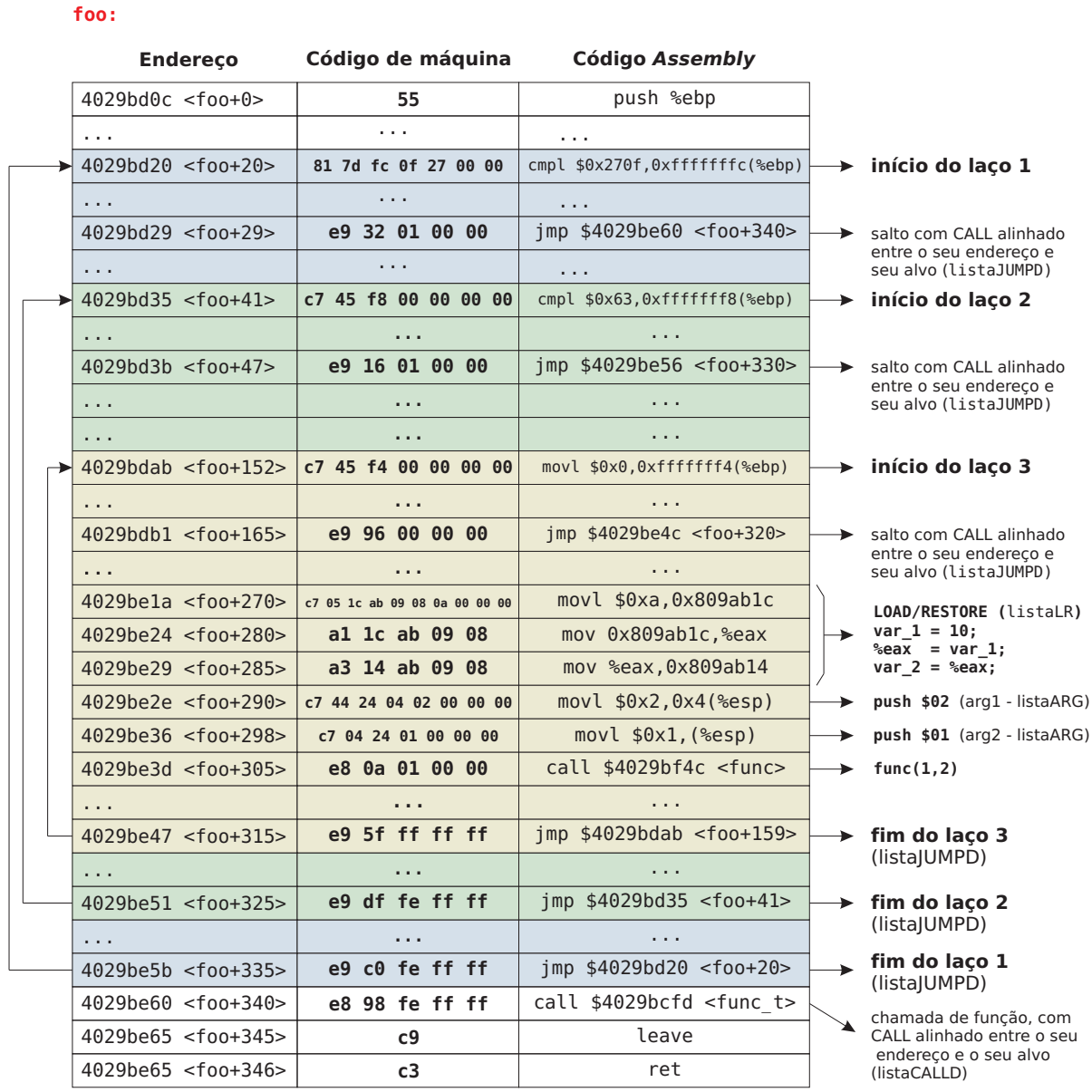


Figura 4.5: Código *Assembly* da função `foo`.

Esta figura mostra apenas as instruções de interesse à passagem de parâmetros para o módulo de otimização.

A partir dos códigos das funções `func` e `foo` é possível determinar os parâmetros para a aplicação das técnicas de otimização. Como dito na sessão anterior, esses parâmetros

encontram-se nas seguintes listas: `listaCALL` (alinhamento de funções e propagação de constantes) e `listaLR` (remoção de `LOAD/RESTORE` redundantes). Os dados da `listaCALL`, obtidos com a execução do programa apresentado, estão mostrados na figura 4.6.

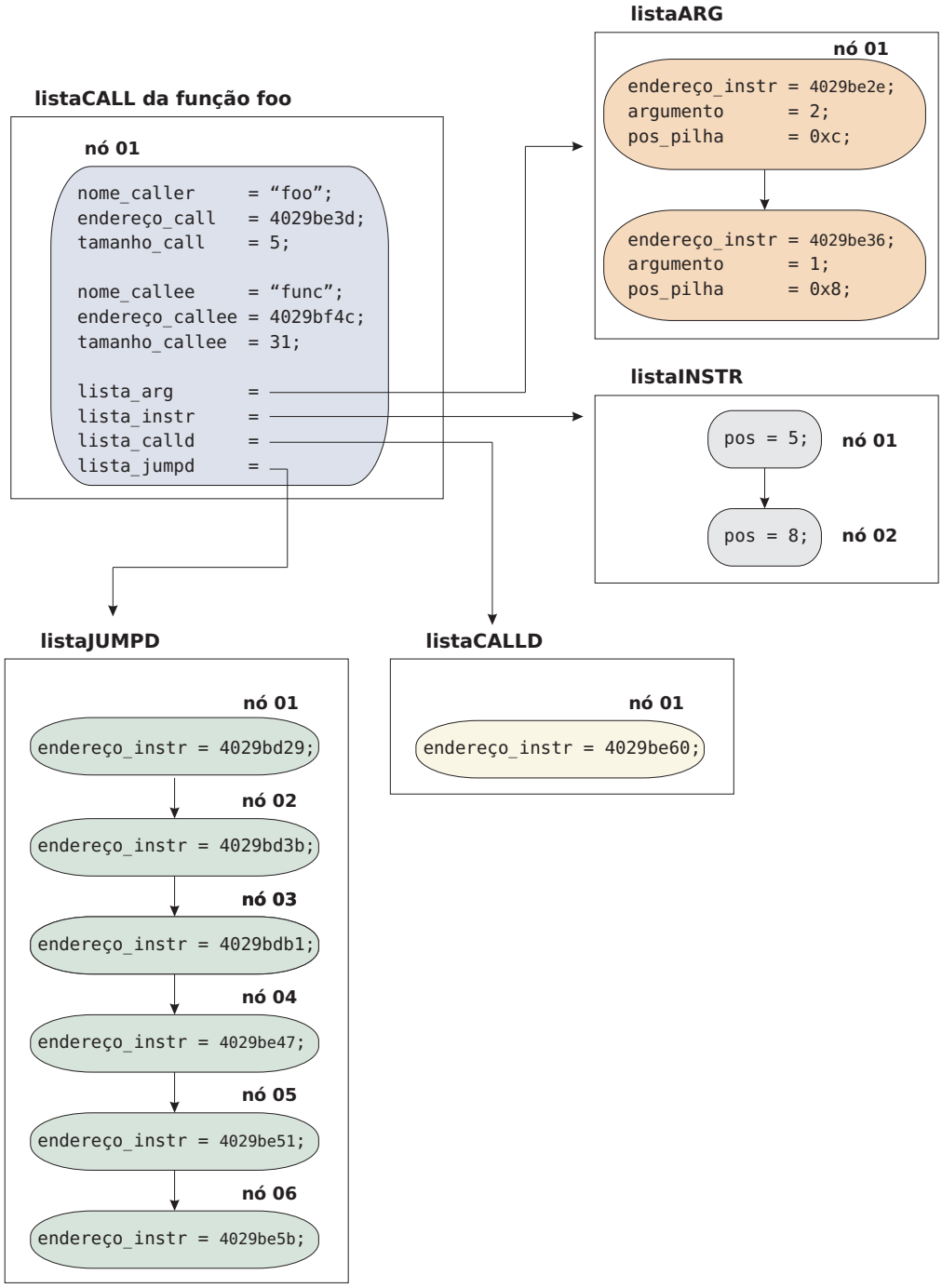


Figura 4.6: Dados da `listaCALL` obtidos com a execução do programa.

Os dados da `listaLR`, por sua vez, são mostrados na figura 4.7.

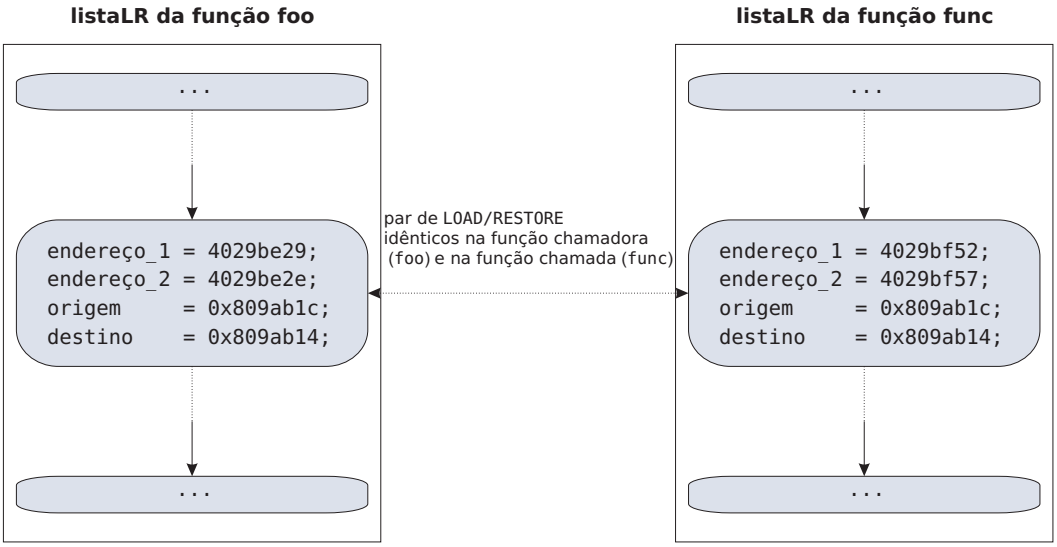


Figura 4.7: Dados da `listaLR` obtidos com a execução do programa.

Como pode ser visto, os valores das listas correspondem aos pontos de otimização, que estão mostrados no código *Assembly* nas figuras 4.4 e 4.5. Deste modo, as estruturas obtidas com a execução do programa teste estão de acordo com o esperado. Com os parâmetros identificados e as estruturas construídas, inicia-se o processo de otimização.

4.3 Aplicação das otimizações

Com as informações coletadas pelo processo de descompilação do bloco gargalo, é possível automatizar a aplicação das técnicas de otimização abordadas nesse projeto. A aplicação das técnicas implementadas permitiu medir o aumento de desempenho e sugerir ao usuário a aplicação delas em alguns pontos específicos do programa.

Os testes foram realizados utilizando-se três programas construídos com a biblioteca *MPI* com diferentes características, mas que possuem em comum os seguintes aspectos:

1. Há um único laço gargalo no programa, localizado dentro da função `foo`.
2. Dentro desse laço há uma chamada de função (`func`) - viabilizando a aplicação do alinhamento de função. Para a função `func` são passados argumentos literais, que

não são alterados dentro dela - o que permite explorar a propagação de constantes.

3. Foram inseridas atribuições entre variáveis dentro da função chamadora e da função chamada para testar a ocorrência de `LOAD/RESTORE` redundantes.

A estrutura geral dos programas utilizados nos testes do módulo de otimização - é mostrado na figura 4.8.

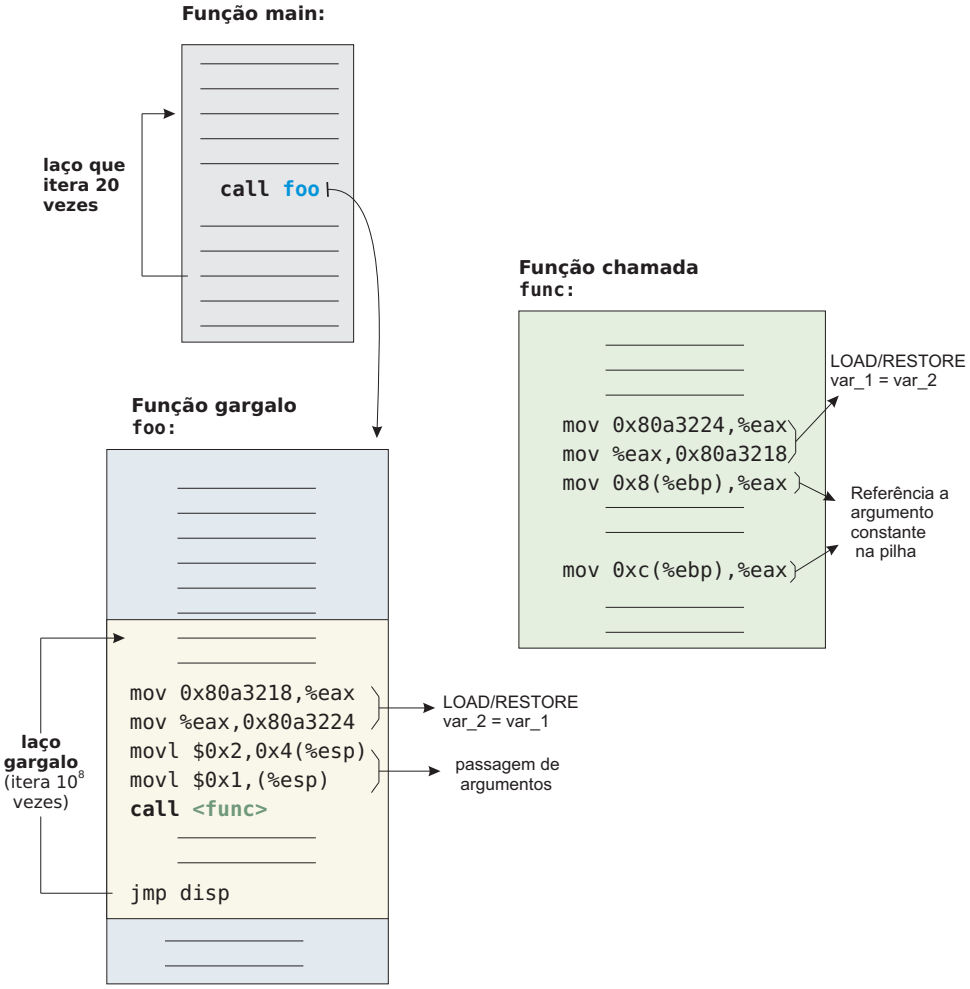


Figura 4.8: Estrutura geral do programa usado nos testes.

Uma vez que o laço dentro da função `foo` é identificado como gargalo, o módulo otimizador cria uma função relocada de `foo` para cada possibilidade:

- F_1: função `foo` original, sem otimização.

- F_2: função `foo` com o alinhamento da função `func`.
- F_3: função `foo` com o alinhamento de função `func` e propagação de constantes.
- F_4: Função `foo` com o alinhamento de função `func`, propagação de constantes e remoção de `LOAD/RESTORE` redundantes.

Cada uma desses casos é avaliado individualmente pelo módulo otimizador. Deve-se ressaltar que dentro de cada função relocada são adicionadas instrumentações para medir o tempo de execução do laço. A seguir, serão apresentados cada um desses programas e os respectivos resultados.

4.3.1 Primeiro teste

Para esse teste, foi utilizado um programa paralelo com as características descritas anteriormente, ou seja, com um único laço gargalo reconhecido pelo Paradyn. Nesse programa, o laço gargalo itera cem milhões de vezes para cada vez que a função gargalo (`foo`) que o contém é chamada. Dentro desse laço há apenas duas operações: atribuição entre variáveis (operações de `LOAD/RESTORE`) e uma chamada da função `func`, à qual são passados argumentos constantes. A função chamada, por sua vez, possui uma operação de atribuição e algumas poucas operações matemáticas. mil A figura 4.9 mostra os tamanhos das funções `foo` e `func` e do laço gargalo, em *bytes*. Como pode ser visto pelas figuras 4.8 e 4.9, a função `foo` possui um laço com um pequeno número de instruções, que itera um número relativamente grande de vezes. A função chamada, por sua vez, também é pequena.

Tamanhos dos blocos e funções

```
Função foo:    656 bytes
Laço gargalo:  46  bytes
Função func:   31  bytes
```

Figura 4.9: Tamanho das funções chamadora e chamada e do laço gargalo utilizados no primeiro teste.

Esse programa foi executado com o Paradyn, que apontou a função `foo` como gargalo. O módulo de instrumentação de laços identificou o único laço dessa função também como

gargalo, ocupando 92.3% do tempo de execução da função. Uma vez que o programa foi construído de modo a viabilizar a aplicação das três técnicas de otimização, mediu-se o tempo de execução do laço gargalo para cada uma das situações descritas anteriormente. A medição do laço gargalo sem nenhuma otimização é necessária para permitir verificar se os resultados das otimizações representam ou não algum ganho de desempenho.

A tabela 4.1, mostrada a seguir apresenta os dados obtidos pelo módulo otimizador, que mediu o tempo de execução do laço para cada uma das 4 situações descritas anteriormente.

	F_1	F_2	F_3	F_4
Tempo total do laço	2.1604s	2.6732s	1.6749s	0.9766s
N^o de iterações do laço	$3x10^8$	$4x10^8$	$3x10^8$	$2x10^8$
Tempo médio de execução do laço	0.7201s	0.6683s	0.5583s	0.4883s
Diferença em relação ao programa original	0%	-7.19%	-22.47%	-32.19%

Tabela 4.1: Resultados da otimização no primeiro teste.

Nessa tabela o tempo total do laço representa o tempo medido pela instrumentação do laço gargalo ao longo das execuções da função relocada. O número de iterações é o número de vezes que o laço gargalo foi executado, podendo variar entre as funções relocadas. O tempo médio de execução indica o tempo médio de execução do laço por função relocada. E por fim, apresenta-se o ganho de desempenho (em termos de diminuição do tempo de execução) obtido em cada caso avaliado.

A partir da tabela 4.1, pode-se observar, que para este programa, a aplicação das técnicas de otimização resultaram em um ganho de desempenho significativo. Deve-se ressaltar, também, que as otimizações habilitadas pelo alinhamento da função `func` (propagação de constantes e remoção de `LOAD/RESTORE` redundantes) são as principais responsáveis pelo ganho de desempenho apresentado.

4.3.2 Segundo teste

No segundo teste foi empregado um programa semelhante ao usado no primeiro teste, ou seja, possui um único laço gargalo que itera cem milhões de vezes, localizado na função `foo`, que faz chamada a uma função `func`. Assim como no teste anterior, o programa foi construído de forma a viabilizar a aplicação das três técnicas de otimização.

A diferença entre ambos reside no tamanho do laço gargalo. Neste exemplo, o laço gargalo é significativamente maior. Com isso, espera-se que os efeitos das otimizações sejam menores nesse exemplo, como pode ser observado nos resultados apresentados a seguir. A figura 4.10 ilustra as diferenças entre os tamanhos dos laços nos dois programas.

A execução do programa com o Paradyn permitiu identificar o laço da função `foo` como gargalo, ocupando 96.7% do tempo de execução da função. Após a fase de instrumentação, o módulo otimizador foi chamado para obter o tempo de execução do laço em cada um dos 4 casos. O resultado dessa medição pode ser visto na tabela 4.2.

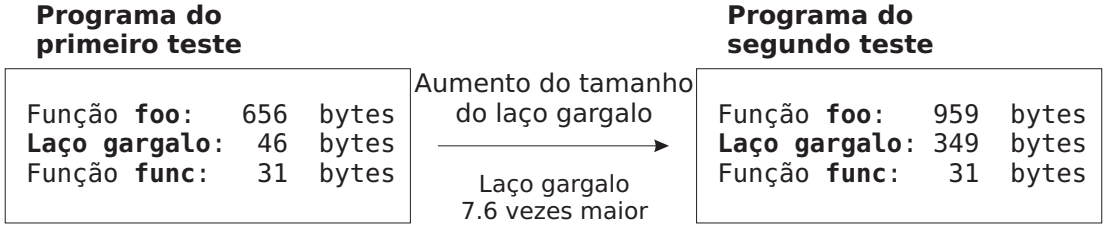


Figura 4.10: Diferença entre os tamanhos dos laços dos programas do primeiro e segundo testes.

	F_1	F_2	F_3	F_4
Tempo total do laço	3.3186s	7.915s	4.7418s	4.7214s
N^o de iterações do laço	$2x10^8$	$5x10^8$	$3x10^8$	$3x10^8$
Tempo médio de execução do laço	1.6593s	1.5830s	1.5806s	1.5738s
Diferença em relação ao programa original	0%	-4.6%	-4.74%	-5.15%

Tabela 4.2: Resultados da otimização no segundo teste.

Os resultados obtidos mostram um pequeno ganho de desempenho - em torno de 5%-, quando comparados com o ganho de desempenho alcançado no teste anterior, e confirmam a hipótese de que com o aumento do tamanho do laço, o impacto da otimização é minimizado. Como a maior parte do laço é composto por trechos de código que não são alterados pela otimização aplicada, a influência dos trechos otimizados sobre o resultado final é menor.

4.3.3 Terceiro teste

Esse teste também utilizou um programa semelhante ao do primeiro teste. A diferença entre ambos está no tamanho da função `func`, chamada dentro do laço gargalo e alvo da

técnica de alinhamento de função. Neste teste buscou-se verificar o impacto do aumento da função alinhada nos resultados da otimização. Na figura 4.11, pode-se observar a diferença entre os tamanhos das funções `func` utilizadas nos dois programas.

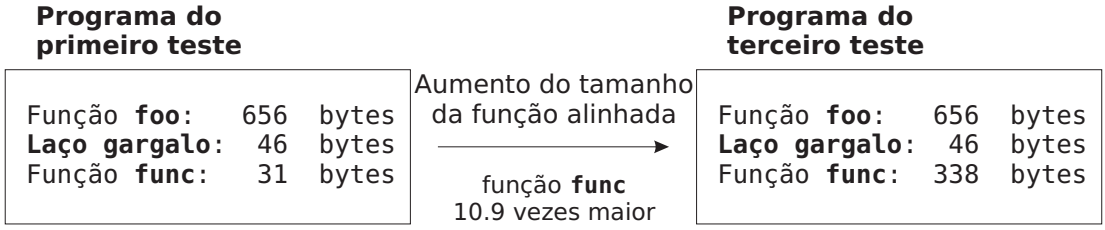


Figura 4.11: Diferença entre os tamanhos dos laços dos programas do primeiro e segundo testes.

O programa foi executado com o Paradyn, que apontou o laço da função `foo` como gargalo, ocupando 97.1% do tempo de execução da função. O módulo otimizador gerou o código da função `foo` relocada para cada um dos 4 casos e obteve os resultados descritos na tabela 4.3

	F_1	F_2	F_3	F_4
Tempo total do laço	5.7828s	2.878s	2.8830s	4.3158s
N^o de iterações	$3x10^8$	$2x10^8$	$2x10^8$	$3x10^8$
Tempo médio de execução do laço	1.4457s	1.4391s	1.4415s	1.4386s
Diferença em relação ao programa original	0%	-0.46%	-0.29%	-0.49%

Tabela 4.3: Resultados da otimização no terceiro teste.

Os resultados obtidos revelam um ganho de desempenho (em torno de 0.5%) quase inexpressivo. A provável causa da pequena otimização é a diminuição do impacto do *overhead* de chamada e retorno de função provocado pelo aumento significativo da função alinhada. Observa-se, assim, uma diminuição do efeito da otimização com o aumento da função que será alinhada dentro do laço gargalo.

5 Conclusões e trabalhos futuros

Este capítulo descreve inicialmente as conclusões referentes ao projeto e suas experiências profissionais e acadêmicas obtidas pelos alunos (5.1). Em seguida, possíveis trabalhos futuros, que possam ser desenvolvidos na linha deste projeto, e que foram identificados ao longo da execução do mesmo, são descritos no tópico 5.2.

5.1 Conclusões

As atividades de conceituação teórica foram uma parte concebidas durante a primeira fase do projeto e outra parte durante a segunda fase do projeto. O estudo realizado na primeira parte centrou-se na ferramenta utilizada - o Paradyn - e nas arquiteturas Sparc e x86. Na segunda parte o estudo caracterizou-se pela investigação das técnicas possíveis de serem aplicadas no laço gargalo. Muitas publicações referentes a otimização dinâmica de código para arquitetura x86 foram utilizadas.

Conseqüentemente, a etapa de desenvolvimento também esteve presente nas duas partes do projeto. Na parte inicial, realizou-se o processo de porte da aplicação da arquitetura Sparc para a arquitetura x86. Durante esta etapa, alguns problemas e casos não tratados na implementação em Sparc foram identificados e corrigidos. Na segunda parte, primeiramente implementou-se o descompilador de bloco gargalo, responsável por construir uma estrutura, com todas as informacões relevantes, para que as otimizações sejam feitas. Em seguida, implementou-se a parte do módulo otimizador responsável por aplicar as técnicas no código da aplicação e medir o tempo de execução dos laços gargalos em que foram efetuadas as otimizações.

Quanto à contribuição deste projeto para o desenvolvimento intelectual e acadêmico dos alunos, pode-se considerá-la significativa. O desenvolvimento do projeto exigiu árduo estudo da codicaficação da ferramenta, das técnicas de instrumentação de laços e de otimizações possíveis de serem analisadas em um código binário. Isto demandou análise de código de linguagem de programação C++, estudo das arquiteturas Sparc e x86 e um estudo aprofundado para decidir quais otimizações são melhor aplicáveis neste contexto e como implementá-las.

Uma vez que trabalho mostrou ser relevante a sistemas computacionais de alto desempenho, os problemas defrontados, juntamente com a complexidade do projeto, trouxeram-nos uma experiência muito gratificante.

Por fim, a concessão de bolsas de iniciação científica, fornecidas pela FAPESP, bem como a estrutura disponível para que as implementações e os testes pudessem ser feitos, foram de grande incentivo para os alunos.

5.1.1 Conclusões do Projeto

Os resultados obtidos na fase de porte de instrumentação mostraram que o porte da ferramenta para a arquitetura x86 foi realizado adequadamente. A seguir são listadas as implicações da adição do código da instrumentação, que foram corretamente tratadas durante a primeira etapa do projeto.

- Correção de deslocamento das instruções de salto e de chamada de função;
- Ajuste do tamanho das instruções de salto;
- Relocação da função contendo as instrumentações.

A instrumentação de laços nessa etapa viabilizou a aplicação das técnicas de otimização, ao mostrar de forma evidente os trechos que demandam alto tempo de execução. A partir dos testes realizados pode-se perceber que a instrumentação é realizada de forma precisa e eficiente.

Na fase de implementação do módulo do otimizador foi possível reunir um conjunto de técnicas possíveis de serem aplicadas em código de máquina e que permitem explorar

possibilidades de otimização não abordadas por compiladores. Os resultados obtidos com o módulo implementado permitiram observar que:

- Há um ganho de desempenho maior em casos em que um laço gargalo pequeno itera um grande número de vezes e chama um função pequena;
- Com o aumento do laço gargalo ou da função chamada por ele, os efeitos da otimização sobre o tempo de execução do laço diminuem;
- Se a função chamada dentro do laço - a ser alinhada - for grande, os efeitos da otimização sobre o tempo de execução do laço também diminuem.

As implicações da aplicação dessas técnicas, como a necessidade de correção de deslocamento das instruções de salto e de chamada de função também foram tratadas.

5.2 Trabalhos Futuros

Possíveis continuações deste trabalho envolvem a alteração automática do código binário original e a indicação dos pontos do código fonte passíveis de alteração. A alteração automática não é possível nesta fase pois o Paradyn faz suas medições sobre imagens do código binário, não sendo capaz de alterá-lo concretamente. A indicação dos pontos de alteração no código fonte depende de um mapeamento mais preciso entre fonte e executável, o que é difícil em função de otimizações previamente feitas pelo compilador. O que se faz atualmente é a indicação de quais chamadas de função devem receber alinhamento.

Referências Bibliográficas

1 CHANG, P. P.; MAHLKE, S. A.; HWU, W. mei W. Using profile information to assist classic code optimizations. In: . New York, NY, USA: John Wiley & Sons, Inc., 1991. v. 21, n. 12, p. 1301–1321. ISSN 0038-0644.

2 MILLER, B. P. et al. The paradyn parallel performance measurement tool. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 28, n. 11, p. 37–46, 1995. ISSN 0018-9162.

3 BOCARDO, D. R. et al. Medição de desempenho de programas paralelos - estendendo o paradyn. *Projeto Final de Graduação*, 2005.

4 PARADYN PROJECT. *Paradyn Developer's Guide*. [S.l.], May 2003.

5 ZHAO, P.; AMARAL, J. N. To inline or not to inline ? enhanced inlining decisions. *16th Workshop on Languages and Compilers for Parallel Computing*, October 2003.

6 MCFARLING, S. *Procedure Merging with Instructions Caches*. [S.l.], 1991.

7 BALL, J. E. *Program Improvement by the Selective Integration of Procedure Calls*. Tese (Doutorado) — University of Rochester, 1983.

8 DAVIDSON, J. W.; HOLLER, A. M. *Subprogram Inlining: A Study of its Effects on Program Execution Time*. Charlottesville, VA, USA, 1989.

9 DAVIDSON, J. W.; HOLLER, A. M. *A study of a C function inliner*. 775–790 p. Tese (Doutorado), New York, NY, USA, 1988.

10 STEENKISTE, P. *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization*. Tese (Doutorado) — Stanford University, 1987.

11 CHANG, P. P.; HWU, W.-W. Inline function expansion for compiling c programs. In: *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 1989. p. 246–257. ISBN 0-89791-306-X.

12 CHANG, P. *Aggressive code improving techniques based on control flow analysis*. Dissertação (Mestrado) — University of Illinois, 1987.

13 TANENBAUM, A. S.; STAVEREN, H. van; STEVENSON, J. W. Using peephole optimization on intermediate code. *ACM Trans. Program. Lang. Syst.*, ACM Press, New York, NY, USA, v. 4, n. 1, p. 21–36, 1982. ISSN 0164-0925.

14 SPINELLIS, D. Declarative peephole optimization using string pattern matching. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 34, n. 2, p. 47–50, 1999. ISSN 0362-1340.

15 BANSAL, S.; AIKEN, A. Automatic generation of peephole superoptimizers. In: *Conference on Architectural Support for Programming Languages and Operating Systems*. [S.l.: s.n.], 2006.

16 DEBOSSCHERE, K.; DEBRAY, S. *alto: A Link-Time Optimizer for the DEC Alpha*. Tucson, AZ, USA, 1996.

17 SCHWARZ, B. W. *Post Link-Time Optimization on the Intel IA-32 Architecture*. [S.l.], 2005.

18 ROLAZ, L. *An Implementation of Sparse Conditional Constant Propagation for Machine SUIF*. [S.l.], March 2003.

19 TRIANTAFYLLIS, S. et al. A framework for unrestricted whole-program optimization. In: *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2006. p. 61–71. ISBN 1-59593-320-4.

20 ITO, S. A. Arquiteturas vliw: Uma alternativa para exploração de paralelismo a nível de instrução. *http://www.inf.ufrgs.br/procpar/disc/cmp134/trabs/T1/981/VLIW/vliw10.html (visitado em 21/09/2006)*, Junho 1998.

21 DONGARRA, J.; HINDS, A. R. Unrolling loops in fortran. *Software-Practice and Experience*, v. 9(3), p. 219–226, 1979.

22 WEISS, S.; J.E., S. A study of scalar compilation techniques for pipelined supercomputers. *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 105–109, Outubro 1987.

23 HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. [S.l.]: Morgan Kaufmann Publishers, 1990.

24 DAVIDSON, J. W.; JINTURKAR, S. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. IEEE Computer Society Press, Los Alamitos, CA, USA, p. 125–132, 1995.

25 WEAVER, D. L.; GERMOND, T. *The SPARC Architecture Manual - Version 9*. [S.l.], 1994.

26 INTEL CORPORATION. *IA-32 Intel Architecture Software Developer's Manual - Volume 1: Basic Architecture*. [S.l.], June 2006.

APÊNDICE A -- Arquitetura x86

A primeira etapa de implementação do projeto consistiu no porte da instrumentação de laços (originalmente elaborada para a arquitetura Sparc[25]) para a arquitetura x86[26]. A instrumentação dinâmica da aplicação pelo Paradyn envolve a inserção de código na imagem binária da aplicação. A otimização de código, por sua vez, também envolve a manipulação do código executável da aplicação, o que torna a implementação tanto da instrumentação quanto do módulo de otimização dependentes da arquitetura.

Nesse tópico serão apresentadas brevemente algumas das características da arquitetura x86, focando os elementos da arquitetura envolvidos no porte da instrumentação e no desenvolvimento do módulo de otimização.

A.1 Características da Arquitetura x86

A arquitetura x86 engloba uma famíla de processadores iniciada pelo 8086, e inclui processadores de 16, 32 e 64 bits. Ela foi concebida como uma arquitetura CISC (*Complex Instruction Set Architecture*) e enfatiza a compatibilidade entre as variações de processadores x86, ou seja, um código criado para um processador x86 de 16 bits deve executar normalmente em processadores x86 de 32 e 64 bits.

Neste trabalho será enfatizada a arquitetura IA-32, nome dado à famíla de processadores x86 de 32 bits, uma vez que o *cluster* utilizado para a implementação e testes deste trabalho é composto por processadores de 32 bits. Apesar disso, a implementação realizada é válida e funcional para quaisquer processadores x86.

A seguir, são destacadas as principais características da arquitetura IA-32:

- Arquitetura superscalar, executando instruções através de dois *pipelines* (u e v) de 5 estágios, permitindo executar até 2 instruções por ciclo de *clock*, isto é, permite a execução paralela de instruções;
- Número relativamente grande de instruções;
- Apenas 8 registradores de uso geral. Todos são de 32 bits, mas podem ser usados como registradores de 8 ou 16 bits;
- Instruções apresentam o seguinte formato básico:

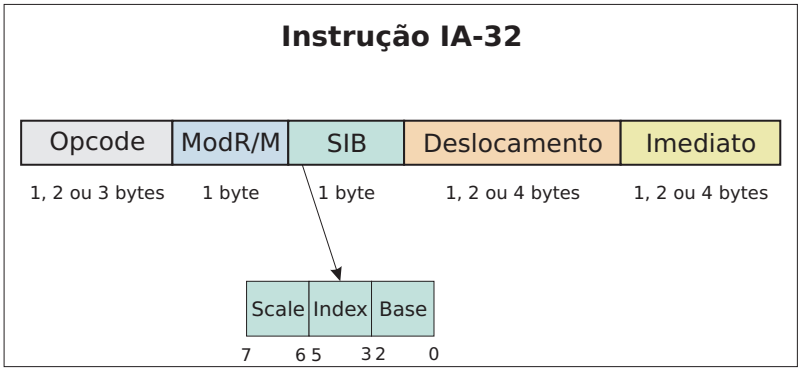


Figura A.1: Formato de instrução IA-32

Alguns campos da instrução, como ModR/M, SIB, Deslocamento e Operando Imediato são opcionais. Apenas o código de operação (*opcode*) é obrigatório. O ModR/M é um *byte* utilizado para especificar o modo de endereçamento e pode ser complementado pelo *byte* SIB (*Scale-Index-Base*) que especifica a forma como um endereço de memória é calculado ($Endereco = RegBase + RegIndex * 2^{scale}$), sendo útil no endereçamento de arranjos. O deslocamento também altera a forma como o operando é calculado, sendo usado quando o endereçamento é relativo. O campo "Imediato" especifica o valor de um operando imediato (codificado na própria instrução), quando existente;

- Instruções de tamanho variável, podendo ter de 1 a 15 *bytes*;
- Grande variedade de formas de endereçamento: indireto, imediato, de registrador, indireto por registrador, por base, indexado, de *string*, relativo e direto através de porta de E/S;

- Unidade de Predição de Desvios, utilizada para tratar instruções de desvio condicional, que prejudicam o *pipeline*. Quando uma instrução de desvio condicional é buscada na memória, essa unidade antecipa se o desvio será tomado ou não, por meio do histórico de execução da instrução.
- Possibilidade de uso de modelo de memória plano (*flat)*) ou segmentado. O primeiro modelo trata todo endereço de memória como um valor de 32 bits. O segundo, por sua vez, divide o endereço em duas partes: um segmento e um deslocamento dentro do segmento.

A.2 Instruções de *Branch*

As instruções de desvio são especialmente importantes para o porte de instrumentação e para o módulo de otimização, pois a partir delas consegue-se identificar laços de repetição e chamadas `CALL` no código. A identificação de laços de repetição é essencial para a instrumentação desse tipo de bloco básico. A identificação de instruções `CALL` é útil tanto para a instrumentação de laços quanto para a técnica de otimização *function inlining*.

A.2.1 Instruções de Salto

As instruções de salto condicional e incondicional têm de 2 a 6 *bytes* e caracterizam-se pelo fato de o endereço alvo do desvio ser calculado através de um valor de deslocamento presente na instrução. Na figura A.2, mostra-se o formato basico da instrução de salto na arquitetura x86 e o cálculo para se obter o endereço alvo do salto.

A.2.2 Instruções de Chamada e Retorno de Procedimento

A instrução de *branch* `CALL` basicamente invoca uma função ou procedimento e possui de 2 a 5 *bytes*. Assim como as instruções de salto, a instrução `CALL` apresenta um valor de deslocamento, por meio do qual calcula o endereço inicial da função chamada. Além de desviar o fluxo de execução do programa, essa instrução automaticamente coloca na pilha do procedimento corrente (*stack frame*) o endereço de retorno, que será o da primeira

Figura A.2: Instrução de salto na arquitetura x86/IA-32

instrução após o `CALL`.

A instrução de retorno de procedimento, por sua vez, não apresenta operandos. Consiste em um código de operação de 1 *byte*. Ela retira da pilha o endereço de retorno (empilhado pelo `CALL`) e altera o valor do registrador EIP (ponteiro de instruções) para esse valor.

A figura A.3 ilustra o formato da instrução `CALL` e o modo como o endereço da função alvo é calculado.

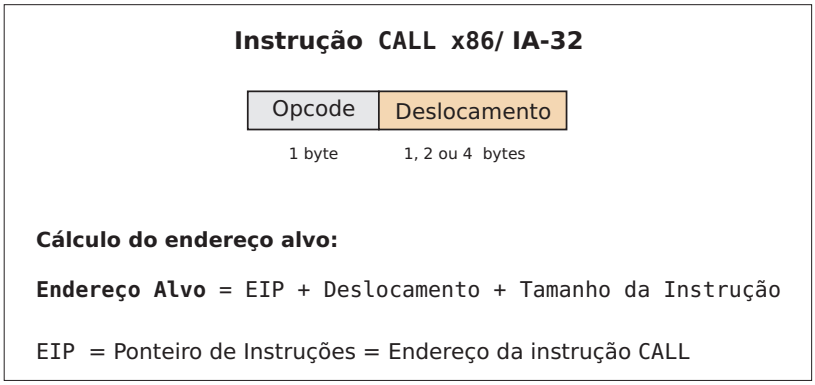


Figura A.3: Instrução CALL na arquitetura x86/IA-32

Já a figura A.4 exemplifica as operações realizadas pelas instruções de chamada e retorno de procedimento sobre a pilha.

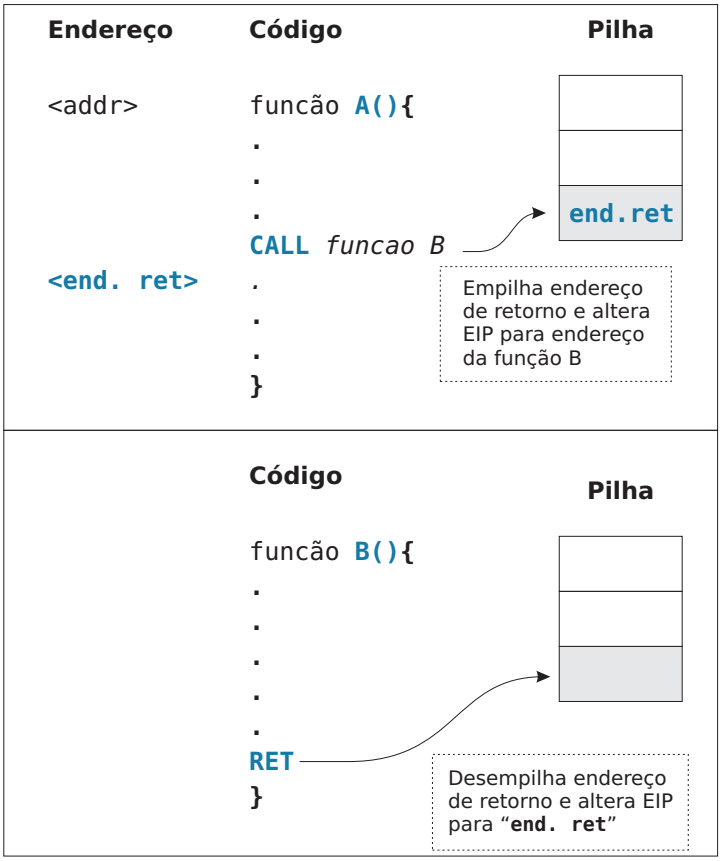


Figura A.4: Alterações realizadas na pilha pelas instruções `CALL` e `RET`

A.3 Instruções de Manipulação da Pilha

As instruções de manipulação direta da pilha são `PUSH/PUSHL` e `POP/POPL`, que respectivamente inserem e removem um valor de 2, 4 ou 8 *bytes* da pilha. Essas instruções são executadas automaticamente por outras instruções, como `CALL` e `RET`, descritas anteriormente.

O estudo dessas instruções e do funcionamento da pilha é importante no estudo da técnica de propagação de constantes entre procedimentos.

Essa técnica, descrita na seção 2.2.3, necessita saber quais são os parâmetros passados para uma função invocada por meio de uma chamada `CALL`. Os parâmetros passados por uma função são colocados na pilha pela função chamadora e acessados pela função chamada. Após a execução da função invocada, os parâmetros são removidos da pilha.

Para um melhor entendimento sobre essas instruções, deve-se, portanto, mostrar funcionamento básico da pilha na arquitetura IA-32/x86.

A.3.1 Pilha na arquitetura IA-32

A pilha consiste em uma estrutura de dados mantida em memória, que trabalha sob a disciplina LIFO (*Last In, First Out*) e que cresce dos endereços mais altos para os mais baixos. Cada função ou procedimento de um programa, se necessário, reserva um espaço na pilha denominado *stack frame*, o qual é delimitado pelos valores dos registradores `ebp` e `esp`. Esses registradores na arquitetura IA-32 são reservados para auxiliar as operações na pilha:

- `esp`: trata-se do *stack pointer*, que armazena o endereço do topo da pilha (primeira posição livre da pilha);
- `ebp`: trata-se do *base pointer*, que armazena o endereço inicial do *stack frame*.

Em seu *stack frame*, cada função armazena suas variáveis locais e informações para o retorno da função e reestabelecimento do *stack frame* da função chamadora.

A figura A.5 explica as alterações provocadas na pilha pela instrução `PUSH/PUSHL`.

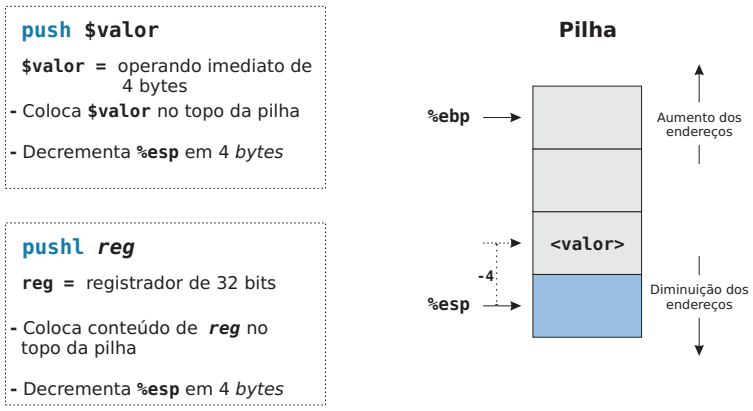


Figura A.5: Instrução `PUSH/PUSHL` e alterações na pilha

A figura A.6 explica as alterações provocadas pela instrução `POP/POPL`, que são basicamente inversas àquelas realizadas pela instrução `PUSH/PUSHL`.

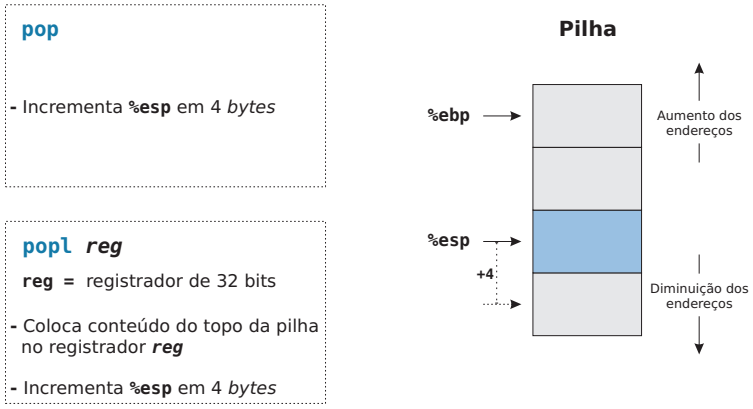


Figura A.6: Instrução `POP/POPL` e alterações na pilha

Com base nesses conceitos, as instruções `CALL` e `RET` podem ser representadas da seguinte forma:

```
CALL func  =  push end. retorno
              movl $end.func, %eip

   RET     =  popl reg
              movl %reg, %eip
```

Instrução `Leave`

Outra instrução associada ao retorno de função é o `LEAVE`, que é responsável por desalocar o *stack frame* da função chamada. Se a função invocada aloca espaço na pilha para armazenar suas variáveis locais, então o uso da instrução `LEAVE` torna-se necessário para que o *stack frame* da função chamadora seja corretamente restaurado após a execução do `RET`.

Antes de alocar o seu *stack frame* a função chamadora coloca na pilha o valor corrente do registrador `ebp` (que corresponde ao *base pointer* da função chamada), para que o `ebp` da função chamadora possa ser recuperado depois.

A instrução `LEAVE` basicamente ajusta o *stack pointer* (`esp`) para o valor corrente do *base pointer* (`ebp`) e então atribui o valor presente no topo da pilha ao *base pointer*. A figura A.7 demonstra melhor essa operação.

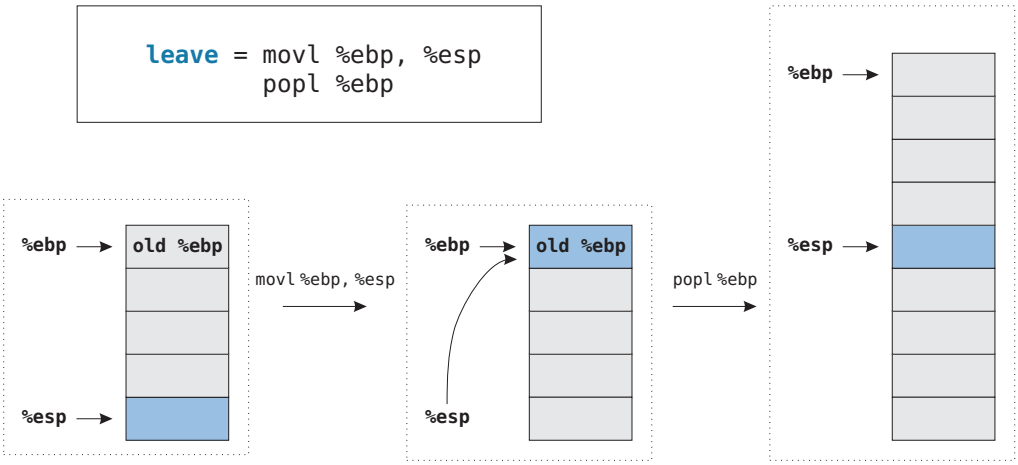


Figura A.7: Operações realizadas pela instrução `LEAVE`