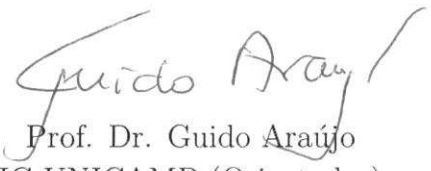
ISAMAP
Tradução Binária Dinâmica
Orientada a Mapeamento de Instruções

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Maxwell Monteiro Andrade de Souza e aprovada pela Banca Examinadora.

Campinas, Fevereiro 2008.



Prof. Dr. Guido Araújo
IC-UNICAMP (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues

Souza, Maxwell Monteiro Andrade de

So89i ISAMAP tradução binária dinâmica orientada a mapeamento de instruções / Maxwell Monteiro Andrade de Souza -- Campinas, [S.P. :s.n.], 2008.

Orientador : Guido Araújo

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Tradução binária dinâmica. 2. Processador PowerPC. 3. Processador X86. 4. Mapeamento de instruções. I. Araújo, Guido. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: ISAMAP instruction mapping driven dynamic binary translation.

Palavras-chave em inglês (Keywords): 1. Dynamic binary translation. 2. PowerPC microprocessors. 3. X86 processors. 4. Instruction mapping.

Área de concentração: Geração Dinâmica de Código

Titulação: Mestre em Ciência da Computação

Banca examinadora:

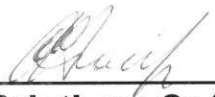
Prof. Dr. Guido Araújo (IC-UNICAMP)
Prof. Dr. Cristiano Araújo (Cin-UFPE)
Prof. Dr. Sandro Rigo (IC-UNICAMP)
Prof. Dr. Rodolfo Azevedo (IC-UNICAMP)

Data da defesa: 10-03-2008

Programa de pós-graduação: Mestrado em Ciência da Computação

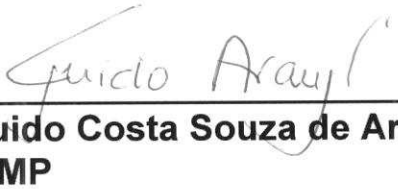
TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 10 de março de 2008, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Cristiano Coêlho Araújo
CI – UFPE

Prof. Dr. Sandro Rigo
IC – UNICAMP



Prof. Dr. Guido Costa Souza de Araújo
IC – UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

ISAMAP
Tradução Binária Dinâmica Orientada a Mapeamento de Instruções

Maxwell Monteiro Andrade de Souza[1]

Fevereiro de 2008

Banca Examinadora:

- Prof. Dr. Guido Araújo
  IC-UNICAMP (Orientador)
- Prof. Dr. Cristiano Araújo
  CIn-UFPE
- Prof. Dr. Sandro Rigo
  IC-UNICAMP
- Prof. Dr. Rodolfo Azevedo
  IC-UNICAMP (Suplente)

[1]Suporte financeiro de: Bolsa do CAPES 2006–2008

Resumo

Tradução binária dinâmica consiste em permitir que programas originalmente compilados para uma determinada arquitetura, executem sobre um nova arquitetura sem a necessidade de recompilação. Esta técnica pode ser usada como ferramenta de migração de aplicações entre arquiteturas ou até mesmo para permitir que uma aplicação execute sobre várias arquiteturas de forma transparente. A tradução binária dinâmica também permite que otimizações, não possíveis em tempo de compilação, sejam feitas em tempo de execução. ISAMAP é um sistema de tradução binária orientado a especificações de mapeamento de instruções entre um Conjunto de Instruções (ISA) origem e um ISA alvo. Em ISAMAP seqüências de instruções da ISA alvo são associadas à instruções da ISA origem, permitindo um mapeamento rápido e otimizado. Atualmente o ISAMAP realiza tradução binária de código PowerPC 32 para código x86.

Abstract

The main role of Dynamic Binary Translation is the capability of running applications compiled for a specific architecture over a totally diferent one without sources recompiling. This technique can be used neither in legacy code migration or in a transparent run-time environment to run applications of different arquitectures. Dynamic Binary Translation also offers otimizations possibilities once informations about application run-time behaviour are available. The ISAMAP is a mapping instructions driven dynamic binary translation system that makes able a mapping between two differents arquitectures. Instructions sequence of the source ISA are mapped to target ISA instructions, providing a fast and optimized mapping. In the current state ISAMAP translates PowerPC 32 binary code to x86 binary.

Agradecimentos

Eu gostaria de agradecer a todos que contribuiram diretamente e indiretamente com a realização desse trabalho.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Com as novas tendências para utilização de processadores *multi-core* com *Instruction Set EM64T* [1], *AMD64* [2] e similares, surge a necessidade de tornar possível que programas compilados para arquiteturas legadas de 32 bits ou com *Instruction Set* diferentes, sejam executados pelas novas arquiteturas. Em processadores com vários núcleos o processo de tradução binária pode ser dedicado a um desses núcleos enquanto os outros executam o código traduzido, reduzindo assim o *overhead* provocado pela tradução, otimização, *profiling*, etc [12] [17] [19].

A princípio pensa-se que o problema poderia ser resolvido recompilando os antigos programas para as novas arquiteturas [21]. No entanto, fazer recompilação de código legado pode ser bem mais cara do que proporcionar um ambiente de execução de aplicações antigas nas novas arquiteturas. A recompilação pode até mesmo ser impossível, em alguns casos, quando não se tem o código fonte da aplicação. Pode-se acrescentar também o fato dos usuários não gostarem ou não poderem parar suas aplicações para adaptá-las à nova plataforma [21].

A tradução binária torna-se uma alternativa, bastante interessante, aos problemas citados acima, pois permite a execução de programas antigos em uma nova plataforma de forma transparente para o usuário. No entanto o desempenho pode ser um problema, devido ao *overhead* gerado pelo mecanismo tradução binária e a qualidade do código gerado.

Esse trabalho apresenta um tradutor binário dinâmico orientado a mapeamento de instruções. As instruções são decodificadas, codificadas e mapeadas de acordo com as descrições dos conjuntos de instruções das arquiteturas envolvidas. Essas descrições permitem especificação de mapeamentos eficientes, pois são feitas em uma linguagem similar ao

[1] *EM64T (Extended Memory 64-bit Technology)* é uma extensão de 64-bits da arquitetura IA-32 desenvolvida pela Intel Corp. Permite computações de 64 bits em diversas plataformas (desktops, workstation/server) quando combinado com suporte de software [1].

[2] *AMD64* Arquitetura de 64 bits desenvolvida pela AMD, com suporte a instruções x86.

assembly da arquitetura. No caso, tradução de código PowerPC para código x86. Em testes realizados o ISAMAP consegui ganhos de quase 80% quando comparado ao tradutor binário QEMU [3] em *benchmarks* de ponto flutuante.

Este trabalho está organizado da seguinte forma:

- Introdução: Apresenta problemas e motivações para a realização do trabalho e o que é o ISAMAP;
- Tradutores Binários: Apresenta conceitos básicos sobre tradutores binários e tipos de tradução;
- ISAMAP: Descreve o tradutor binário ISAMAP, sua organização, sub-sistemas e modelos de mapeamento utilizados;
- Extensão na linguagem ArchC: Mostra as extensões feitas na linguagem ArchC para adapta-la ao escopo de tradução binária;
- Resultados: Apresenta resultados comparativos de testes feitos com o ISAMAP;
- Conclusão: Conclusões finais sobre o trabalho e proposições para trabalhos futuros.

Capítulo 2

Tradutores Binários

Em geral os tradutores binários existentes na literatura são estruturados em módulos de interpretação, tradução e interface com o sistema operacional [5]. O módulo de interpretação decodifica as instruções do programa origem e executa funções que emulam o comportamento da instrução decodificada. A tradução é feita em apenas algumas partes do código do programa, geralmente laços e *traces* mais executados, que são regiões onde a aplicação de otimização e tradução resulta em ganhos de desempenho significativos [5]. A interface com o sistema operacional é responsável pela gerência de chamadas ao sistema, tratamento de exceções, I/O, etc.

2.1 Interpretação

Inicialmente os tradutores binários começam lendo instrução por instrução do programa original, decodificando-as e executando funções para simular seu comportamento. Em geral, esse processo é feito de forma mecânica com o auxílio de tabelas de equivalência entre as instruções do programa origem e as implementações das instruções da arquitetura alvo. Isto torna o processo simples, mas no entanto, lento. Principalmente quando as instruções interpretadas estão contidas em laços, demandando sucessivas interpretações de um mesmo trecho de código.

Uma outra abordagem é a geração de funções que representam o comportamento de uma determinada instrução origem a partir de uma especificação, como no *framework* Yirr-Ma [9]. As funções que representam o comportamento de cada instrução são descritas em uma sintax *C-like*. O tradutor, objetivo deste projeto, usa uma estratégia semelhante. Para cada instrução original são definidas uma ou mais instruções alvo utilizando uma sintax semelhante ao *assembly* da arquitetura.

2.2 Tradução Binária

O processo de tradução binária é um processo mais sofisticado que o de interpretação. Para cada instrução decodificada é gerado código binário da arquitetura alvo. Isto permite maior desempenho do sistema quando este processo é auxiliado por uma cache de código traduzido.

Durante o processo de interpretação, informações sobre o fluxo de controle e dados do código interpretado podem ser coletadas, assim como outras informações pertinentes. Essa técnica de coleta de informações é chamada de *profiling* e grande parte dos tradutores binários a utilizam: FX!32 [6], Dynamo [4], IA-32 EL [5] entre outros. Em geral as informações mais aproveitadas são o número de execuções de um determinado trecho de código e endereços alvos de instruções de desvio ou chamadas a procedimentos. Regiões de código que foram interpretadas muitas vezes são sérias candidatas a serem traduzidas para código nativo e armazenada em uma *cache* para posterior execução [5].

As informações coletadas também são utilizadas no processo de otimização do código traduzido. Um exemplo de otimização que pode ser realizada, é a realocação de registradores, que pode resultar em ganho de desempenho caso a arquitetura alvo tenha um número maior de registradores. Algumas arquiteturas chegam a implementar instruções específicas para realocação de registradores, como é o caso do Itanium®2 que possui a instrução *alloc* [13].

2.2.1 Tradução Binária Dinâmica

Quando o processo de tradução é feito durante a execução do programa, tem-se uma tradução binária dinâmica. Geralmente, a tradução dinâmica é feita em apenas algumas partes do código, as mais executadas, onde os resultados da tradução e das possíveis otimizações trazem maiores ganhos de desempenho. Exemplos de tradutores binários dinâmicos são Aries [21], IA-32 EL [5], CMS [7], Dynamo [4].

2.2.2 Tradução Binária Estática

Uma outra alternativa é utilizar as informações obtidas durante a interpretação do código origem para fazer uma tradução quase integral do código quando este não estiver sendo executado. Geralmente o programa traduzido é armazenado em uma base de dados e invocado no lugar do programa original quando requisitado. Este é o processo de tradução usado no FX!32 [6].

2.3 Run-Time

Quando alguma exceção ocorre na execução do código 64 bits não se pode simplesmente retransmitir a exceção para o programa 32 bits original. É necessário converter a exceção e definir um estado que represente corretamente a exceção gerada em uma arquitetura 32 bits [5]. Chamadas ao sistema, I/O, interrupções, gerenciamento de memória, também devem ser tratados para que a comunicação entre programas de plataformas diferentes seja feita de forma transparente e correta. Para tentar contornar esses problemas os tradutores binários implementam um sistema de *Run-Time*, que tem como objetivo gerenciar esta comunicação entre o código origem e o sistema operacional.

Capítulo 3

Trabalhos Relacionados

Há vários tradutores binários na literatura. Abaixo estão listados os mais conhecidos.

3.1 Aries

Em Aries [21] é feita tradução de instruções PA-RISC para IA-64 utilizando mecanismos de interpretação rápida juntamente com tradução binária dinâmica. Quando o kernel do sistema HP-UX identifica que uma aplicação, preste a ser executada, é uma aplicação PA-RISC, o kernel inicia o módulo *Start-Up* do Aries. O módulo *Start-Up* invoca o módulo de *Run-Time* o qual inicia o interpretador que começa a interpretar o código PA-RISC. O *Run-Time* gerencia o número de vezes que cada bloco do código é executado e a partir do momento que o número de execuções de cada bloco atinge um limiar, o tradutor dinâmico é executado para traduzir o bloco em código nativo (IA-64). Depois que um bloco é traduzido, todas as subseqüentes chamadas ao bloco executaram o código nativo traduzido. O módulo *Enviroment Emulation* é responsável por gerenciar as chamadas de sistema, sinais e exceções entre o código PA-RISC e o sistema IA-64. O Aries consegue executar com sucesso quase todas as aplicações PA-RISC existentes, com apenas algumas poucas exceções. A não emulação do "debugador" HP-UX/PA-RISC e de programas que utilizam bibliotecas compartilhadas PA-RISC e IA-64 são exemplos de tais exceções.

3.2 FX!32

Digital FX!32 [6] realiza execução de programas x86 em um sistema Alpha, utilizando emulação e tradução binária estática. O Digital FX!32 é uma DLL que intercepta chamadas a rotina de sistema Windows `CreateProcess`. Se a imagem especificada na chamada de `CreateProcess` é um código x86, o Agente FX!32 invoca o módulo de *Run-Time* para

executar a imagem. O código é emulado e informações sobre sua execução são coletadas. Assim que a imagem x86 não está sendo mais executada, o módulo tradutor traduz o código da imagem que foi emulada. As informações coletadas são utilizadas para fazer as devidas otimizações. O código traduzido é então armazenado em um *DataBase*, o qual será executado da próxima vez que a imagem for requisitada para execução. O Digital FX!32 é um sistema de tradução binária estático totalmente transparente para o usuário. No entanto tem o desempenho bastante degradado quando executa um programa x86 pela primeira vez. Durante a primeira execução o código x86 é interpretado pelo emulador. Por exemplo, o FX!32 executa um código x86 em um sistema Windows NT Alpha com um processador 500Mhz com o mesmo desempenho do código x86 executando nativamente em um sistema Windows NT com processador 200Mhz.

3.3 IA-32EL

No IA-32 EL[5] a execução de código IA-32 ocorre em um sistema Itanium utilizando interpretação e tradução binária dinâmica. Foi projetado para trabalhar com sistemas operacionais baseados no Itanium. O que é obtido separando-se os mecanismos de tradução em uma biblioteca independente do sistema que se comunica com um biblioteca dependente do sistema. Há dois modos de tradução, o *Cold Code* que simplesmente interpreta o código IA-32 e o *Hot Code* que traduz um conjunto de blocos (um super-bloco ou *trace*), se o número de execuções do trecho de código ultrapassar um determinado limiar. Os blocos traduzidos são armazenados em uma cache e executados sempre que requeridos. Para o tratamento de exceções o IA-32 EL fornece um mecanismo no qual, quando uma exceção é gerada o sistema operacional transmite esta exceção para o IA-32 EL ao invés de transmiti-la para a aplicação. Assim o IA-32 EL se encarrega de converter a exceção de uma arquitetura 64 bits para um exceção padrão IA-32. Em benchmarks de inteiros o IA-32 EL alcançou um nível de desempenho médio de 65% em relação ao desempenho de códigos nativos.

3.4 Dynamo

Dynamo [4] opera com um sistema de interpretação de instruções PA-8000 executando sobre um processador PA-8000. O código original é interpretado até que uma região "quente" (*trace*, laços, etc...) seja encontrada. Quando tal região é identificada, a otimização da região é feita é armazenada em uma cache, onde posteriormente o código otimizado será buscado para execução quando necessário. Em alguns casos de teste o Dynamo consegui obter speeds-up de 22%, já em outros casos é obtido speed-ups negativos

de -4%.

3.5 CMS

Em CMS (*Code Morphing Software*) [7] é apresentado o processador Transmeta's Crusoe que é uma completa implementação a nível de sistema de uma arquitetura x86, o qual consiste de um processador VLIW nativo com uma camada de software (CMS). Ao contrário dos outros tradutores apresentados, que funcionam a nível de aplicação, o CMS funciona a nível de sistema. O sistema CMS se baseia em um paradigma de especulação agressiva e um mecanismo de recuperação *rollback-commit* suportado pelo hardware. Assim como a maioria dos tradutores binários dinâmicos, o CMS inicialmente interpreta o código x86 e assim que o número de execuções de uma determinada região do código alcança um limiar, o tradutor é invocado. O tradutor traduz e otimiza um determinado trecho de código e o guarda em uma cache, onde o mesmo possa ser executado quando requisitado. O mecanismo de recuperação do CMS é suportado pelo hardware através da existência de registradores *shadow*. Para cada registrador que guarda um estado x86 (registradores de trabalho) existe um outro registrador *shadow*. Na ocorrência de alguma exceção ou tomada de algum *branch* indevidamente, uma operação de *rollback* substitui os valores dos registradores de trabalho pelo conteúdo dos registradores *shadow*. O conteúdo dos registradores *shadow* é definido pelo último *commit* feito. Em uma operação de *commit* o conteúdo dos registradores de trabalho é copiado para os registradores *shadow*. O CMS é um sistema comercial disponível que prove alto desempenho e uma implementação totalmente compatível de um conjunto de instruções de arquitetura x86 em um conjunto de instruções de uma arquitetura diferente, no caso arquitetura VLIW.

3.6 UQDBT

UQDBT [18] é um *framework* de tradução binária dinâmica que suporta diversas arquiteturas como origem e alvo. Isto é obtido através de um conjunto de especificações que definem formato de arquivos binários, sintaxe das instruções e semântica das instruções. A unidade de tradução utilizada é um bloco básico. As instruções do programa origem são decodificadas e transformadas em uma abstração independente da arquitetura da máquina. A partir desta representação intermediária é feito o processo de tradução e emissão de código assembly e a posterior codificação em código objeto. Se o número de execuções de um conjunto de blocos alcança um limiar especificado. Otimizações dinâmicas são feitas sobre esse trecho de código (*hot path*). Vários *hot paths* podem ser transformados em um único *hot path* se as ligações entre estes seguirem um determinado

padrão. Depois que os *hot paths* são determinados, o conjunto de blocos que formam o *hot path* podem ser traduzidos e executados como se fossem um único bloco. Com o sistema de *profiling* desligado o UQDBT executa programas de 2.5 a 7 vezes mais lentos que programas nativos. Quando o *profiling* é ativo, tornando possível a otimização, há um ganho de desempenho de até 15% em relação ao desempenho com profiling desligado.

3.7 ADORE

ADORE [13] é um *framework* de otimização binária dinâmica que utiliza suporte do hardware para avaliar o comportamento do código executado. O Itanium 2 possui registradores que são utilizados para medir a performance dos programas que estão sendo executados. São 4 contadores que monitoram centenas de métricas de desempenho, possibilitando uma visão completa da execução dos programas. O sistema de otimização do ADORE consiste em tentar identificar *traces* (super-blocos) que são freqüentemente utilizados (cerca de 2% do código). Os *traces* são definidos a partir de registradores específicos que guardam os endereços alvo dos quatro últimos *branchs* tomados. Depois que o *trace* é identificado, várias otimizações são feitas, como alocação dinâmica de registradores utilizando a instrução IA-64 `alloc`, *data cache prefetching* e *trace layout*. O resultado deste mecanismo de monitoramento com suporte de hardware é um *overhead* de apenas 2% para executar *profiling* do código executado. As otimização feitas sobre os *traces* dos programas podem levar a *speed-ups* de até 156%. Considerando que o speed-up vária conforme o tipo da aplicação..

3.8 DAISY

DAISY [8] é um sistema de emulação para executar aplicações de diversas arquiteturas em uma arquitetura VLIW. O processo de tradução é feito com a decodificação das instruções origem em instruções RISC (Se a arquitetura origem for CISC). A cada instrução RISC produzida é encontrada uma instrução VLIW na qual a instrução RISC possa ser colocada, enquanto ao mesmo tempo é feito o escalonamento global de instruções VLIW. O DAISY usa um sistema de reordenação de instruções agressivo, ou seja, fazendo especulações. Isto é possível através de suporte de hardware, onde existem registradores *non-architected* que são usados nas especulações. Se o resultado de uma especulação não é o previsto, nenhuma exceção será gerada. O Mapeamento de memória é feito dividindo-se a memória do sistema VLIW em três partes: uma sendo a memória utilizada pelo programa base, outra sendo a memória do *Virtual Machine Monitor*, responsável pela tradução do código. E por último a memória destinada a cache de código traduzido. Os resultados demonstram a obtenção

de um grau de paralelismo por volta de 2 instruções para uma arquitetura VLIW com instruções de largura 4 (2 ALU, 2 *Load/Store*, 1 *Branch*).

3.9 SIND

SIND [16] é um *framework* para o desenvolvimento de aplicações e técnicas de tradução binária focado em flexibilidade. O SIND é composto de vários módulos interligados. O *bootstrapper* carrega e interrompe a execução de código antes da execução da linha de código principal. Toda comunicação entre os diversos módulos é feita pelo *dispatcher*, o qual é responsável pela inicialização do gerenciador de memória e o interpretador. Isto permite que informações sejam coletadas ao invés de se fazer instrumentação do código. Assim que um *trace* interessante é encontrado, o mesmo é traduzido e armazenado em uma cache (*code fragment cache*). Quando o *trace* traduzido for encontrado novamente, a tradução armazenada na cache será utilizada. Eliminando assim retraduções desnecessárias de código. O código disponível na cache é persistido para permitir que em uma possível nova execução do programa, já se tenha trechos do código traduzido. Esta ação também permite que otimização mais demoradas sejam feitas sobre o código traduzido enquanto este não é executado. As otimizações dinâmicas realizadas pelo SIND se concentram nos *traces* encontrados. As principais otimização realizadas são: substituição de *indirect branchs* por *direct branchs* e *constant folding*. Semelhante ao FX!32, o SIND combina técnicas de otimização binária estática com otimizações e tradução binária dinâmicas. A sua estrutura modular proporciona flexibilidade para adição de novos interpretadores e tradutores para diferentes conjuntos de instruções.

3.10 Yirr-Ma

Yirr-Ma [9] é um *framework* de Tradução Binária Dinâmica que utiliza instrumentações de código customizadas. Como na maioria dos tradutores binários dinâmicos, informações sobre a emulação do código origem são coletadas a fim de auxiliar o processo de tradução e otimização. Yirr-Ma utiliza o *framework WalkAbout* (gerador de emuladores a partir de especificações em SLED e SSL) para gerar emuladores customizáveis para diferentes arquiteturas *host*. Para cada instrução na arquitetura host, a ser emulada, é gerada uma função. O Yirr-Ma então adiciona código de instrumentação para cada instrução do código original (host). Através da coleta de informações sobre a emulação do código, os blocos freqüentemente executados são identificados e classificados como *hot blocks*. Para cada *hot block* é gerada uma função em código nativo que implementa a semântica do bloco. Posteriormente essas funções são armazenadas na cache do Yirr-Ma para possíveis

reutilizações futuras e se houver a possibilidade, as novas funções vão sendo ligadas as funções já existentes na cache (*trace*). O sistema de detecção de *hotspots* implementado pelo Yirr-Ma é bem simples. Este consiste em um contador atrelado a cada bloco básico, o qual é incrementado a cada execução do bloco. Quando o contador alcança um limiar estabelecido, o bloco básico é traduzido em código nativo. A portabilidade do Yirr-Ma é obtida através do framework *WalkAbout*. O Yirr-Ma encapsula o código nativo, a ser emitido, em objetos de código relocáveis e estes são manipulados pelo *framework* como uma caixa preta. Por se tratar de um *framework* ainda em estágio inicial de desenvolvimento. O Yirr-Ma apresenta grandes *overheads* na emulação de programas, principalmente devido a dois fatores. Primeiro o *overhead* causado pelo sistema de *blockkeeping*. Segundo, o comportamento das aplicações. Aplicações que possuem poucos laços não têm chances de ter seu *overhead* amortizado pela execução de laços traduzidos em código nativo.

3.11 QEMU

QEMU [3] é um emulador *open source* de alto desempenho que utiliza tradução binária para executar programas PowerPc, x86, ARM, Sparc, MIPS, m68k em processadores x86, x86_64 e PowerPC. Considerado o mais rápido tradutor binário *open source* PowerPC - x86. Os mapeamentos de instruções é feito por meio de macros escritas em linguagem C, semelhante ao utilizado no ArchC [15]. Essas macros são compiladas durantes a compilação do sistema de tradução binária e o arquivo objeto contendo os mapaeamento compilados é ligado ao código objeto do tradutor. Durante o processo de tradução o QEMU decodifica a instrução e cópia o trecho de código objeto correspondente a seu mapeamento para um buffer, e assim sucessivamente até montar todo um bloco traduzido, não existe codificação de instruções. QEMU possui *cache* de código traduzido, faz ligação de blocos traduzidos e mapeamento de *system calls*. O desempenho do QEMU fica entre 4 a 10 vezes mais lento que o código nativo x86, o que é considerado rápido para um emulador.

3.12 Análise Comparativa

Na tabela 3.1 abaixo são listadas as principais características dos emuladores e tradutores binários apresentados anteriormente. São utilizadas cinco características comuns a tradutores binários no comparativo. Origem-Destino: mostra qual conjunto de instruções o sistema aceita como entrada e para qual conjunto de instruções o código é traduzido. Traces: indica se o sistema em questão implementa mecanismos de construção de regiões de código bastante executadas. Otimizações: mostra se o sistema implementa ou não

otimizações de código. Orientação: indica se o sistema faz o mapeamento de instruções orientado por modelos ou o mapeamento é implementado no próprio código fonte do sistema de tradução. Tradução: indica o tipo de tradução utilizada pelo sistema, dinâmica, estática ou híbrida.

Tabela 3.1: Comparativo

	Origem-Destino	Traces	Otimizações	Orientação	Tradução
Aries	PA-RISC - IA-64	Não	Não	Implementação	Dinâmica
FX!32	x86 - Alpha	Não	Não	Implementação	Estática
IA-32 EL	IA32 - Itanium	Sim	Não	Implementação	Dinâmica
Dynamo	PA-8000 - PA-8000	Sim	Sim	Implementação	Dinâmica
UQDBT	x86-Sparc e Sparc-Sparc	Sim	Sim	Modelos	Dinâmica
ADORE	Itanium - Itanium	Sim	Sim	Implementação	Dinâmica
DAISY	PowerPC - VLIW	Não	Sim	Implementação	Dinâmica
SIND	Sparc, PowerPC - Sparc	Sim	Sim	Implementação	Híbrida
Yirr-Ma	Sparc-x86 e Sparc-Sparc	Sim	Não	Modelos	Dinâmica
QEMU	PowerPC, x86, ARM, Sparc, MIPS, m68k - x86, x86_64, PowerPC	Não	Não	Implementação	Dinâmica
ISAMAP	PowerPC - x86	Não	Não	Modelos	Dinâmica

Capítulo 4

ISAMAP

O ISAMAP é um tradutor binário dinâmico orientado a modelos de descrição. A linguagem utilizada na confecção dos modelos é uma variação da linguagem ArchC [15] [1]. No ISAMAP são utilizados três modelos, um para definição da arquitetura origem, um para arquitetura alvo e um terceiro descrevendo o mapeamento entre os *Intruction Set* das duas arquiteturas. A partir desses modelos é gerado o código para o tradutor binário. Nesses modelos são definidos os formatos das instruções, registradores, operandos e forma de codificação das instruções. Na figura 4.1 é mostrado um exemplo de modelo para o PowerPC e na figura 4.2 é mostrado um exemplo de modelo para x86.

```
ISA(powerpc) {
  isa_format XO1 = "%opcd:6 %rt:5 %ra:5 %rb:5 %oe:1 %xos:9 %rc:1";
  isa_instr <XO1> add, subf;
  isa_regbank r:32 = [0..31];
  ISA_CTOR(powerpc) {
    add.set_operands("%reg %reg %reg", rt, ra, rb);
    add.set_decoder(opcd=31, oe=0, xos=266, rc=0);
    subf.set_operands("%reg %reg %reg", rt, ra, rb);
    subf.set_decoder(opcd=31, oe=0, xos=40, rc=0);
  }
}
```

Figura 4.1: Exemplo de modelo PowerPC.

[1] ArchC é uma linguagem para descrição de arquiteturas de processadores baseada na linguagem SystemC. Com ArchC é possível descrever processadores tanto na forma comportamental (apenas indicando o que cada instrução faz) quanto com precisão de ciclos (indicando o que cada instrução faz em cada ciclo do pipeline). Com base nessa descrição, as ferramentas da linguagem são capazes de gerar simuladores para os processadores.

```
1  ISA(x86) {
2    isa_format op1b_r32 = "%op1b:8 %mod:2 %regop:3 %rm:3";
3    isa_instr <op1b_r32> add_r32_r32, mov_r32_r32;
4    isa_reg eax = 0;
5    isa_reg ecx = 1;
6    ...
7    isa_reg edi = 7;
8    ISA_CTOR(x86) {
9      add_r32_r32.set_operands("%reg %reg", rm, regop);
10     add_r32_r32.set_encoder(op1b=0x01, mod=0x3);
11     mov_r32_r32.set_operands("%reg %reg", rm, regop);
12     mov_r32_r32.set_encoder(op1b=0x89, mod=0x3);
13   }
14 }
```

Figura 4.2: Exemplo de modelo x86.

4.1 Modelos

As principais estruturas encontradas nos modelos que descrevem o conjunto de instruções das duas arquiteturas envolvidas são listadas abaixo:

- `isa_format`: determina a declaração de um formato de instrução que consiste nos campos que compõem o formato e seus respectivos tamanhos em número de bits;
- `isa_instr`: associa instruções aos seus respectivos formatos;
- `isa_reg`: define nomes e códigos dos registradores;
- `isa_regbank`: define um banco de registradores e o intervalo dos códigos dos registradores que compõem o banco;
- `set_operands`: define os operandos da instrução, seus tipos e a que campos do formato se associam. Está disponível três tipos de operandos: `reg`: registrador, `addr`: endereço e `imm`: imediato.
- `set_encoder` e `set_decoder`: especificam quais campos e valores identificam unicamente uma instrução. Estas informações são usadas no processo de decodificação e codificação de instruções.

A figura 4.2 exemplifica a aplicação dos comandos citados acima. Na linha 2 `isa_format` define um formato (`op1b_r32`) com quatro campos com os respectivos tamanhos em bits: 8, 2, 3, 3. Na linha 3 `isa_instr` declara duas instruções (`add_r32_r32` e `mov_r32_r32`) que

são do formato `op1b_r32`. Nas linhas 4 à 7 são declarados os registradores e definidos seus códigos através do comando `isa_reg`. Nas linhas 9 e 11 são definidos os operandos das instruções `add_r32_r32` e `mov_r32_r32`. Nos dois casos o campo `rm` é o primeiro operando e o campo `regop` o segundo. Nas linhas 10 e 12 são definidos os códigos dos campos que identificam a instrução. Os campos que identificam as instruções declaradas são `op1b` e `mod`.

O mapeamento das instruções é descrito em um terceiro modelo. Para cada instrução da arquitetura origem é definida uma ou mais instruções para representar seu comportamento. A sintax da descrição é bastante semelhante à sintax *assembly*, como pode ser visto na figura 4.3.

```
isa_map_instrs {
  add %reg %reg %reg;
} = {
  mov_r32_r32 edi $1;
  add_r32_r32 edi $2;
  mov_r32_r32 $0 edi;
}
```

Figura 4.3: Exemplo de mapeamento PowerPC para x86.

Este esquema não é tão portável quanto o mapeamento descrito por função feito pelo Yirr-MA [9], no entanto, permite geração de código mais eficiente e flexibilidade para implementar diferentes mapeamentos para cada instrução da arquitetura origem. Por exemplo, o mapeamento da figura 4.3 poderia ser feito utilizando a instrução x86 `lea`, o que economizaria uma instrução.

O símbolo *edi* indica que, se a instrução for emitida, um dos operandos deve obrigatoriamente ser o registrador *edi*. Os símbolos iniciados com o caracter $ indicam uma referência a um operando da instrução na arquitetura origem, $0 referência o operando 0, $1 o operando 1 e assim sucessivamente. Se o operando referenciado é um registrador então o registrador na arquitetura alvo, que mapeia o registrador da arquitetura origem, é emitido na instrução. A figura 4.4 mostra um exemplo de um possível código emitido para o mapeamento de uma instrução `add r0, r1, r3` do PowerPc. As instruções nas linhas 0, 2, 5 são código de *spill*.

A instrução na linha 0 carrega o conteúdo do registrador R1, que é representado em memória, para EAX. Na linha 2 é carregado para EAX o registrador R3. Na linha 5 é armazenado o resultado da operação no registrador R0. Este exemplo de código pode ser melhorado utilizando-se outra especificação para a instrução `add` no modelo x86 no mapeamento. A arquitetura x86 permite que pelo menos um dos operandos de uma instrução qualquer seja um acesso a memória. Se for usado as versões com acesso a

```
0 mov eax, 0x80740504
1 mov edi, eax
2 mov eax, 0x80740508
3 add edi, eax
4 mov eax, edi
5 mov 0x80740500, eax
```

Figura 4.4: Exemplo de código emitido.

memória das instruções `add` e `mov` pode-se economizar três instruções no código emitido para o mapeamento. A figura 4.5 mostra a especificação das instruções `add` e `mov` que possuem um dos operandos em memória. A figura 4.6 mostra o novo mapeamento e a figura 4.7 mostra o código emitido para o novo mapeamento. Como pode ser observado o novo mapeamento resultou em apenas três instruções.

```
...
isa_format op1b_r32_m32disp = "%op1b:8 %mod:2 %regop:3 %rm:3 %m32disp:32";
isa_instr <op1b_r32_m32disp> add_r32_m32disp, mov_r32_m32disp;
...
add_r32_m32disp.set_operands("%reg %addr", regop, m32disp);
add_r32_m32disp.set_encoder(op1b=0x01, mod=0x0, rm=0x5);
mov_r32_m32disp.set_operands("%reg %addr", regop, m32disp);
mov_r32_m32disp.set_encoder(op1b=0x8b, mod=0x0, rm=0x5);
mov_m32disp_r32.set_operands("%addr %reg", m32disp, regop);
mov_m32disp_r32.set_encoder(op1b=0x89, mod=0x0, rm=0x5);
```

Figura 4.5: Outro exemplo de modelo x86.

```
isa_map_instrs {
  add %reg %reg %reg;
} = {
  mov_r32_m32disp edi $1;
  add_r32_m32disp edi $2;
  mov_m32disp_r32 $0 edi;
}
```

Figura 4.6: Outro exemplo de mapeamento PowerPC para x86.

Este exemplo de código pode ser melhorado utilizando-se outra especificação para a instrução `add` no modelo x86 e a utilizando no mapeamento. A figura 4.5 mostra a especificação das instruções `add mov` que possuem um dos operandos em memória. A

```
0 mov edi, 0x80740504
1 mov edi, 0x80740508
2 mov 0x80740500, edi
```

Figura 4.7: Outro exemplo de código emitido.

figura 4.6 mostra o novo mapeamento e a figura 4.7 mostra o código emitido para o novo mapeamento. Como pode ser observado o novo mapeamento resultou em apenas três instruções.

4.2 Estrutura

O estrutura do tradutor binário ISAMAP segue o padrão estrutural da maioria dos tradutores binários existentes na literatura. A figura 4.8 mostra a estrutura geral do ISAMAP.

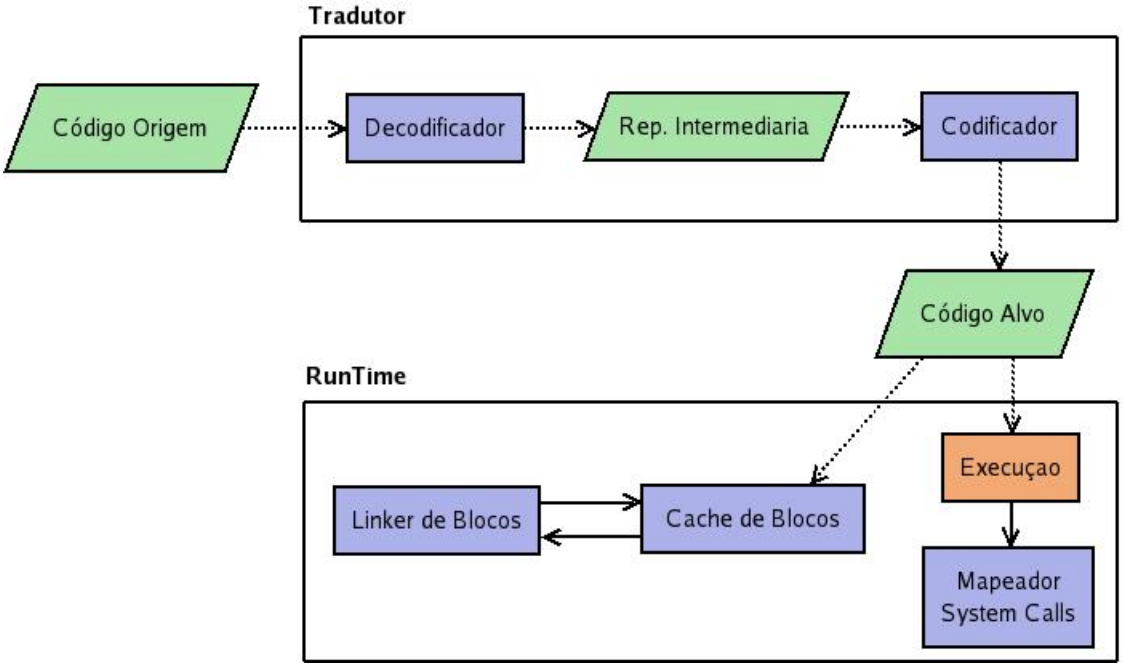


Figura 4.8: Estrutura do ISAMAP

O módulo de tradução consiste nos sub-módulos de decodificação e codificação. O módulo *Run-Time* possui os sub-módulos de `Cache de Código`, ***`Linker`*** `de Blocos` e `Mapeador de` ***`System Calls`***. A Cache de Código armazena blocos básicos de código traduzido e permite sua rápida recuperação se houver necessidade de execução. O Linker

de Blocos realiza ligação entre blocos básicos para que o controle de execução fique mais tempo no código traduzido. Por último tem-se o Mapeador de *System Calls*, que é responsável por fazer o mapeamento entre as *System Calls*, que possuem interfaces diferentes nas duas arquiteturas envolvidas na tradução.

4.3 Gerador de Tradutor

O Gerador de Tradutor recebe como entrada os três modelos de especificação e a partir deles gera o código em linguagem C para o tradutor, `tradutor.c`. O código contido em `tradutor.c` consiste basicamente de chamadas ao decodificador é uma estrutura `switch` para emitir a tradução correspondente a instrução decodificada. Outros arquivos também são gerados como:

- *ctx_switch.c*: Contém código para emissão de código binário para salvar e restaurar conteúdo de registradores da arquitetura destino;
- *isa_init.c*: Contém inicialização das estruturas de dados com informações sobre instruções, formatos e campos das instruções da arquitetura origem;
- *encode_init.c*: Contém inicialização das estruturas de dados com informações sobre instruções, formatos e campos das instruções da arquitetura destino;
- *pc_update.c*: Contém protótipos de funções para fazer a emulação das instruções de *branch*, a implementação deve ser fornecida;
- *spill.c*: Contém protótipos de funções para emitir código binário de *spill*, a implementação deve ser fornecida;
- *sys_call.c*: Contém protótipos de funções para mapeamento de *system calls*, a implementação deve ser fornecida;

As implementações do Decodificador, Codificador e utilitários são genéricas portanto não são geradas.

4.4 Tradutor

A entrada para o tradutor é um *buffer* contendo o código binário do programa da arquitetura origem, no caso código PowerPC. O código binário é carregado a partir do arquivo

ELF [2] do programa a ser traduzido. O `Decodificador` decodifica instrução por instrução até que uma instrução de *branch* seja encontrada. Na especificação da arquitetura origem instruções de *branch* são definidas através do comando `set_type("jump")`. A figura 4.9 mostra um exemplo dessa definição para uma instrução PowerPC.

```
bc.set_operands("%imm %imm %addr %imm %imm", bo, bi, bd, aa, lk);
bc.set_decoder(opcd=16);
bc.set_type("jump");
```

Figura 4.9: Especificação da instrução *branch conditional*.

As instruções da arquitetura origem são decodificadas para uma representação intermediária, e a partir dessa representação é criada a representação intermediária da instruções da arquitetura destino de acordo com o mapeamento. Por fim essa representação intermediária das instruções destino é códificada em código binário. As instruções de *branch* não têm o seu equivalente em binário emitido, isto ficará a encargo do *Linker* de Blocos. Enquanto o blocos não são ligados, as instruções de *branch* da arquitetura origem são emuladas. Esta emulação consiste nos teste de condição de salto, alteração de registradores e cálculo do endereço da próxima instrução. A implementação dessa emulação não é gerada pelo Gerador de Tradutor, portanto deve ser fornecida.

Todos os registradores da arquitetura origem são representados em memória, pois a arquitetura alvo pode não ter registradores suficientes para representar os da arquitetura origem. Logo, para cada referência a estes registradores, no mapeamento de uma instrução, é gerado código de *spill* para carregar seus conteúdos para registradores x86 e armazenar de volta em memória. Registradores que não têm seu conteúdo alterado pela instrução são considerados para leitura, portando só precisam ser carregados da memória. Registradores que têm seu conteúdo alterado são considerados de escrita, e portanto devem ser apenas gravados em memória. Quando um registrador tem seu conteúdo usado pela instrução e em seguida alterado, deve-se carregá-los e depois gravá-los em memória. Código de *spill* não é gerado pelo tradutor quando se tem mapeamentos como o da figura 4.6, onde o operando parâmetro é uma referência à memória.

A semântica de quais operandos da instrução são registradores de leitura/escrita é definida no modelo da arquitetura alvo atravéz das palavras reservadas `set_write` e `set_readwrite`. A ausência dessas implica que o operando é um registrador que não tem seu conteúdo alterado pela instrução. Um exemplo pode ser visto na figura 4.10. As linhas 0, 2 e 5 da figura 4.4 são exemplos de código gerado para carregar e gravar o conteúdo dos registradores em memória.

[2] *ELF (Executalbe and Linking Format / Extensible Linking Format)* é um padrão para arquivos executáveis, código objeto, bibliotecas compartilhadas e *core dumps*

```
add_r32_r32.set_operands("%reg %reg", rm, regop);
add_r32_r32.set_encoder(op1b=0x01, mod=0x3);
add_r32_r32.set_readwrite(rm);

mov_r32_r32.set_operands("%reg %reg", rm, regop);
mov_r32_r32.set_encoder(op1b=0x89, mod=0x3);
mov_r32_r32.set_write(rm);
```

Figura 4.10: Exemplo de definição de leitura/escrita.

4.4.1 Representação Intermediária

A representação intermediária utilizada pelo tradutor do ISAMAP é a estrura de dados utilizada pelo ArchC [15] para representar as instruções, formatos e campos com algumas modificações. Na tabela 4.1 é mostrada essa representação com a descrição de seus campos agrupados por estrutura de dados.

A estrutura `isa_op_field` foi inserida para representar os campos que são operandos de uma instrução e que tipo de operandos são (leitura, escrita, leitura/escrita). O campo `type` foi adicionado devido a necessidade de informações semânticas básicas sobre qual o tipo da instrução, pois originalmente a linguagem Archc não proporciona informações semânticas sobre as instruções descritas. A inserção do campo `format_ptr` foi feita para melhorar o desempenho do tradutor. Ao invés de realizar uma busca em uma lista encadeada pelo nome do formato, tem-se um apontador para formato da instrução. Na implementação do ISAMAP os formatos são armazenados em uma lista encadeada. No momento que a estrutura de dados de uma instrução é preechida, além do nome do formato, é atribuido ao campo `format_ptr` o endereço da estrutura correspondente ao formato. Assim ao invés de realizar uma busca em uma lista encadeada pelo nome do formato (O(n)), tem-se um apontador direto para formato da instrução na lista (O(1)).

4.5 *Endianness*

Na tradução binária entre arquiteturas com *endianness* diferentes como PowerPC (*big endian*) e x86 (*little endian*) é necessário fazer a conversão de *endianess* sempre que se busca um dado na memória. Como o PowerPC é uma arquitetura RISC, então a conversão de *endianess* se restringe a tradução das instruções de *load* e *store*.

Um outra abordagem seria copiar todo o programa para memória ao contrário. O primeiro byte do programa ficaria no endereço mais alto de memória e o último byte no endereço mais baixo e o acesso seria feito decrementalmente [14]. No entanto esta estratégia se torna bem complicada para emulação e tradução binária a nível de aplicação

Tabela 4.1: Representação Intermediária

campo	descrição
ac_dec_field	
name	nome do campo
size	tamanho do campo em bits
first_bit	posição do primeiro bit do campo
id	identificador do campo
val	valor contido no campo
sign	sinal do campo
ac_dec_format	
name	nome do formato
size	tamanho do formato em bits
fields	campos que compõem o formato
ac_dec_list	
name	nome do campo
value	valor contido no campo
isa_op_field	
field	nome do campo
writable	Se o campo representa um operando de escrita ou leitura
ac_dec_instr	
name	nome da instrução
size	tamanho da instrução em bytes
mnemonic	mnemonico da instrução
asm_str	asm da instrução
format	nome do formato da instrução
id	identificador da instrução
cycles	não usado pelo ISAMAP
min_latency	não usado pelo ISAMAP
max_latency	não usado pelo ISAMAP
dec_list	lista dos campos que identificam a instrução
cflow	não usado pelo ISAMAP
op_fields	campos que identificam a instrução
type	tipo da instrução
format_ptr	apontador para formato da instrução

devido aos recalculos de endereços de memória.

O trecho de memória com dados em formato *big endian* se resume ao segmento de dados do programa, onde estão locazidas as variáveis globais e estáticas, já que o *heap* não possue valores iniciais antes da execução de um programa e os valores iniciais da pilha são definidos pelo sistema de *Run-Time*. Como os dados podem ser copiados do segmento de dados para o *heap* ou pilha, e vice-versa, a estratégia adotada no ISAMAP é considerar todos os dados na memória como *big endian*. A conversão será sempre feita quando traduções de instruções *load/store* foram executadas, assim como é feito no QEMU [3].

O código para conversão de *endianness* é especificado no modelo de mapeamento para todas as instruções de *load/store*. Um exemplo de tal mapeamento pode ser visto na figura 4.11. A partir da arquitetura 486 existe a instrução `bswap` que inverte os bytes de uma palavra de 32 bits, o que torna eficiente a conversão de *endianess*. Para conversão de palavras de 16 bits é usada a instrução `xchg`.

```
isa_map_instrs {
  lwz %reg %imm %reg;
} = {
  mov_r32_m32 edx $1 $2;
  //Conversao Endianness
  bswap_edx;
  mov_r32_r32 $0 edx;
}
```

Figura 4.11: Exemplo de mapeamento de uma instrução *load*.

4.6 *Run-Time*

O sistema de *Run-Time* (SRT) do ISAMAP é responsável pela inicialização do ambiente necessário para execução do programa traduzido, execução de blocos traduzidos, gerenciamento da cache de código, ligação de blocos e mapeamento de *system calls*. O SRT possui implementação muito dependente da arquitetura destino tornando-o assim pouco portável. Por exemplo: As transferências de controle entre o SRT e código traduzido exigem que todos os registradores sejam salvos para preservar o contexto. O código que implementa essa transferência deve ser escrito em *assembly*, o que compromete a portabilidade para outra arquitetura.

4.6.1 Inicialização

O ambiente de execução do programa traduzido é configurado de acordo com as especificações da ABI (*Aplication Binary Interface*) da arquitetura origem, no caso a ABI PowerPC Linux. Alguns registradores devem ter um determinado valor inicial, como por exemplo o registrador `R1`, que deve conter o endereço do topo da pilha. A pilha para o programa a ser traduzido é alocada. Atualmente o ISAMAP aloca uma pilha de 512 KB, o que mostrou-se suficiente para executar os *benchmarks* do SPEC CPU 2000, com exceção do gcc que exige uma pilha de 8 MB. A pilha alocada é também inicializada conforme as especificações da ABI [2].

4.6.2 Execução de Blocos Traduzidos

Para execução do código traduzido no mesmo espaço de endereçamento do processo do tradutor binário, é necessaria criar uma *sandbox*. Isto é, isolamento entre os dois ambientes de execução para que um não interfira no outro. Sempre antes da execução de um bloco básico traduzido o conteúdo dos registradores utilizado pelo tradutor binário deve ser salvo (prólogo) e após a execução do bloco traduzido este conteúdo deve ser restaurado (epílogo). Os códigos de prólogo e epílogo são mostrados na figura 4.12. O registrador `esp` não é salvo e restaurado pois não é utilizado no código traduzido para evitar qualquer alteração indevida na pilha do sistema de tradução e pelo fato das instruções `call` e `ret` serem usadas para transferir a execução para o código traduzido.

Prólogo	Epílogo
`mov 0x80a48000, eax`	`mov eax, 0x80a48000`
`mov 0x80a48004, ecx`	`mov ecx, 0x80a48004`
`mov 0x80a48008, edx`	`mov edx, 0x80a48008`
`mov 0x80a4800c, ebx`	`mov ebx, 0x80a4800c`
`mov 0x80a48010, esi`	`mov esi, 0x80a48010`
`mov 0x80a48014, edi`	`mov edi, 0x80a48014`
`mov 0x80a48018, ebp`	`mov ebp, 0x80a48018`

Figura 4.12: Exemplo de código de prólogo e epílogo.

Ao final do código do bloco traduzido é adicionada uma instrução `ret` para retornar o controle de execução ao sistema de tradução. O controle de execução do sistema de tradução para o código traduzido é transferido por uma instrução `call` direcionada para o início do bloco de código traduzido a ser executado. Inicialmente o controle de execução é alternado entre sistema de tradução e bloco básico traduzido, até que o *Linker* de Blocos faça a ligação entre os blocos básicos.

4.6.3 Cache de Código

O processo de tradução pode ser algo muito oneroso para um sistema de tradução binária, logo não é uma boa abordagem traduzir um mesmo bloco básico para cada iteração de um laço. Toda a vez que um bloco básico acaba de ser traduzido e executado, o SRT do ISAMAP o armazena em uma cache, diferentemente do IA-32 EL [5]. Onde apenas blocos freqüentemente executados são colocados na cache. Se posteriormente houver a necessidade de executar este bloco, uma retradução não será necessária, bastando apenas que o SRT recupere o bloco na cache.

Os blocos traduzidos são identificados na *cache* pelo valor de seu endereço original no código não traduzido. O enderoço original é submetido a uma função *hash* para se obter sua chave *hash*. Então uma referência ao bloco é armazenada em uma tabela na posição indicada pela chave *hash*. As buscas na cache do ISAMAP são realizadas com o auxílio dessa tabela *hash*, o que permite uma recuperação extremamente rápida do bloco solicitado. Colisões na tabela *hash* são resolvidas fazendo-se encadeamentos na posição da tabela onde ocorreu a colisão. A figura 4.13 exemplifica a estrutura da tabela *hash* usada pela cache do ISAMAP. A função *hash* implementada no sistema de cache de código do ISAMAP é dada por **hash**(**addr**) = **addr** % **n**.

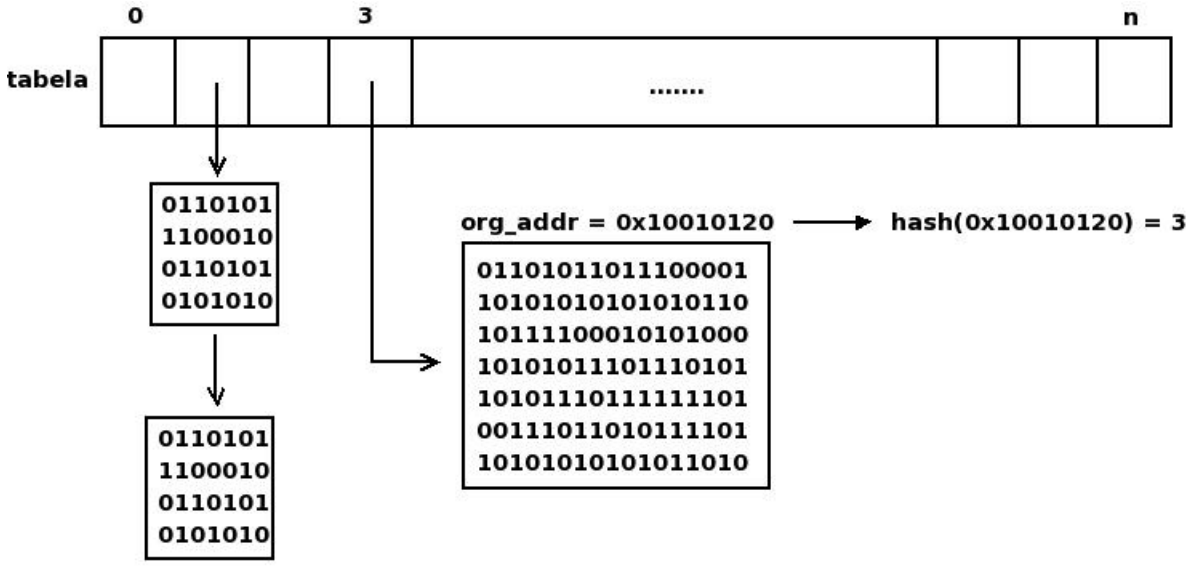


Figura 4.13: Tabela hash usada pela Cache de Código

Assim como no QEMU [3], uma região de memória de 16MB contíguos é utilizada como cache. Este tamanho mostrou-se suficiente para execução de todos *benchmarks* do SPEC CPU 2000. Uma macro `ALLOC` é utilizada para retornar o endereço inicial de uma

região da cache onde será armazenado o código do bloco traduzido. `ALLOC` sempre retorna o próximo endereço vago na cache assim blocos básicos executados em seqüência são guardados bem próximos uns dos outros na cache, proporcionando melhora na localidade do código traduzido na maioria dos casos.

A política de manutenção da cache do ISAMAP é bastante simples. Sempre que a *cache* enche, ela é totalmente esvaziada. Isto não mostrou-se um gargalo de desempenho pois 16MB são suficientes para executar todos os *benchmarks* utilizados e o tempo gasto em tradução de código não chega a 1% do tempo de execução de um *benchmark* com utilização de cache de código. Outra vantagem dessa abordagem é que ela simplifica o sistema de Ligação de Blocos, já que este não terá que efetuar o desligamento de blocos que foram removidos da *cache*.

4.6.4 *Linker* de Blocos

Para melhorar o desempenho do sistema de tradução o ISAMAP possui um módulo de ligação de blocos traduzidos assim como QEMU [3] e Yirr-MA [9]. A partir do momento que blocos traduzidos são ligados o número de vezes que o controle de execução será transferido para o SRT é reduzido drasticamente e a localidade de cache de código do processador é privilegiada.

O *Linker* de Blocos considera quatro tipos de *branchs* para fazer as ligações: *branchs* condicionais, *branchs* incondicionais, *system calls* e *branchs* indiretos. No ISAMAP instruções de *system calls* são consideradas *branchs* incondicionais. Considerando A o último bloco traduzido executado pelo SRT, este possuirá no máximo dois sucessores. B é o bloco correspondente ao alvo do *branch* de A, e C correspondente a próxima instrução após o *branch* no final de A, mais conhecido como *fallthrough*. Se o próximo bloco a ser executado for B, A e B serão ligados senão A e C serão ligados. O bloco que não foi executado, seja B ou C, será ligado a A apenas se for executado no futuro. Ou seja o ISAMAP faz ligação de blocos sob-demanda diferentemente do IA32-EL [5] que liga toda uma sub-árvore de blocos traduzidos. A desvantagem da técnica utilizada pelo IA32-EL [5] é que blocos que nunca serão executados podem ser ligados e colocados na cache. A vantagem é que vários blocos são traduzidos e ligados em uma passagem pelo SRT. A ligação de blocos é feita da seguinte forma: São colocados trechos de código binário, chamados *stubs*, no final do bloco traduzido. Estes realizam as condições de teste do salto, se for um *branch* condicional, e fazem o salto para seu sucessor traduzido ou retornam o controle de execução ao SRT se o salto é para um bloco não traduzido ainda. Abaixo é exemplificada a colocação dos *stubs* para cada tipo de *branch*.

- ***branchs* condicionais**: é adicionado um *stub* após o código do bloco A para realizar o teste da condição de salto. Em seguida são adicionados mais dois *stubs*, um fará

a ligação entre o bloco A e o bloco B, e o outro a ligação entre o blocos A e C. Se o bloco B está traduzido, o *stub* de salto para o alvo faz um salto para B, senão faz um salto para o SRT. Se o bloco C está traduzido o *stub* de salto para o *fallthrough* faz um salto para C, senão faz um salto para o SRT. A figura 4.14 exemplifica esse caso;

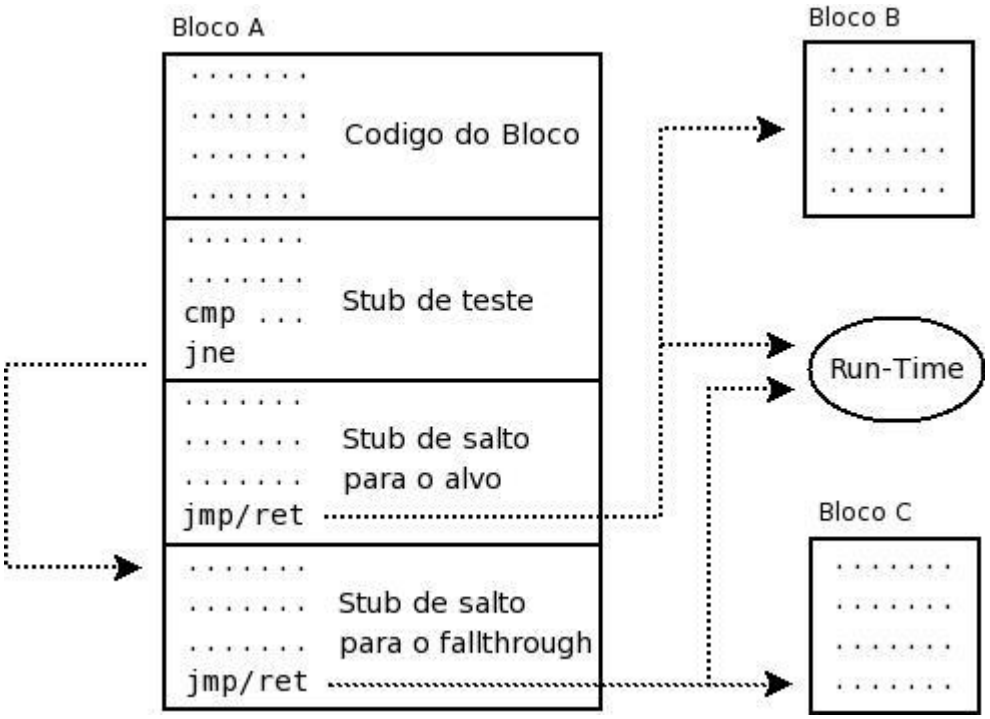


Figura 4.14: Ligação de blocos para *branchs* condicionais

- ***branchs* incondicionais**: é adicionado um stub após o código do bloco A para realizar um salto para o *stub* de salto para o alvo. O *stub* de salto para o alvo se encarregará de fazer o salto para o bloco B. A figura 4.15 exemplifica esse caso;

- ***system calls***: é adicionado um *stub* após o código do bloco A para realizar um salto para o *stub* de salto para o alvo. No *stub* de salto para o alvo é adicionada uma chamada ao sistema de mapeamento de *system calls* para executar a chamada ao sistema. Após a chamada ao sistema é feito o salto para o bloco B. A figura 4.16 exemplifica esse caso;

- ***branchs* indiretos**: é adicionado um *stub* após o código do bloco A para realizar o teste da condição de salto. Em seguida é adicionado mais dois stubs, um fará uma busca na cache pelo bloco B. Se o bloco B estiver na cache, será realizado um

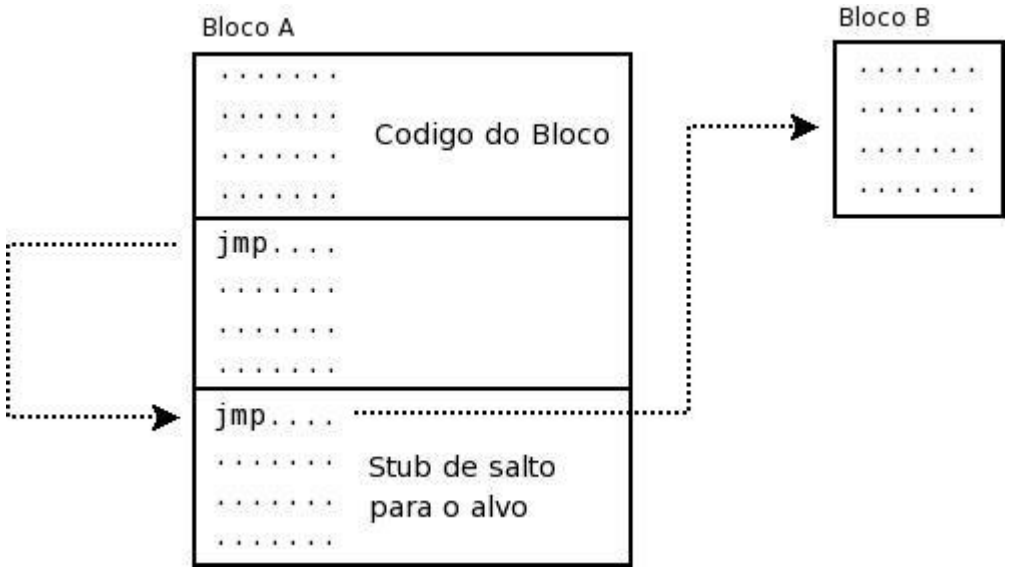


Figura 4.15: Ligação de blocos para *branchs* incondicionais

salto para B, senão um salto para o SRT. O outro stub fará a ligação dos blocos A e C, se C já foi traduzido, senão realizará um salto para o SRT. O código para busca na cache é necessário pois em *branchs* indiretos o endereço do salto talvez só seja conhecido após a execução da última instrução antes da instrução de *branch*. A figura 4.17 exemplifica esse caso.

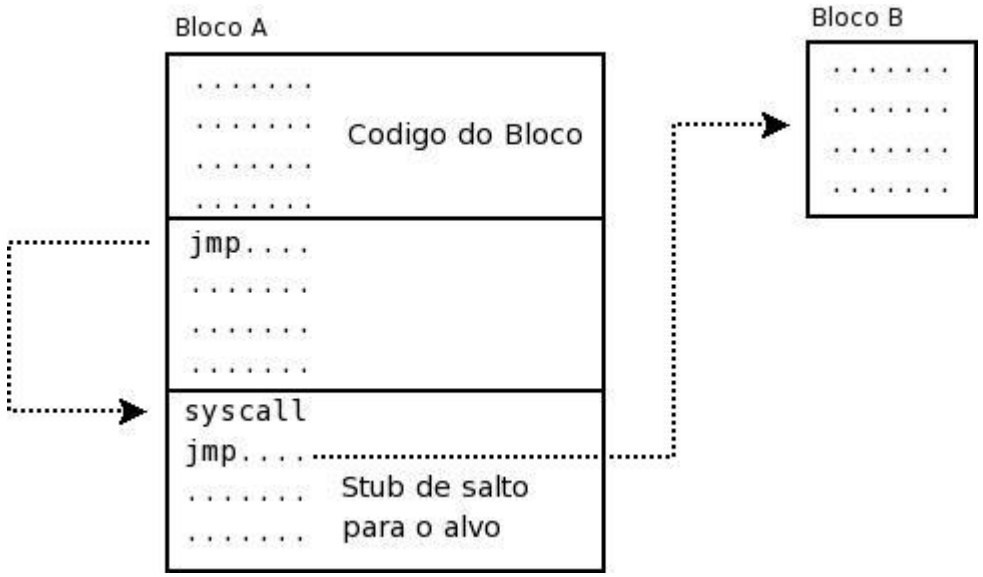


Figura 4.16: Ligação de blocos para *sytemcalls*

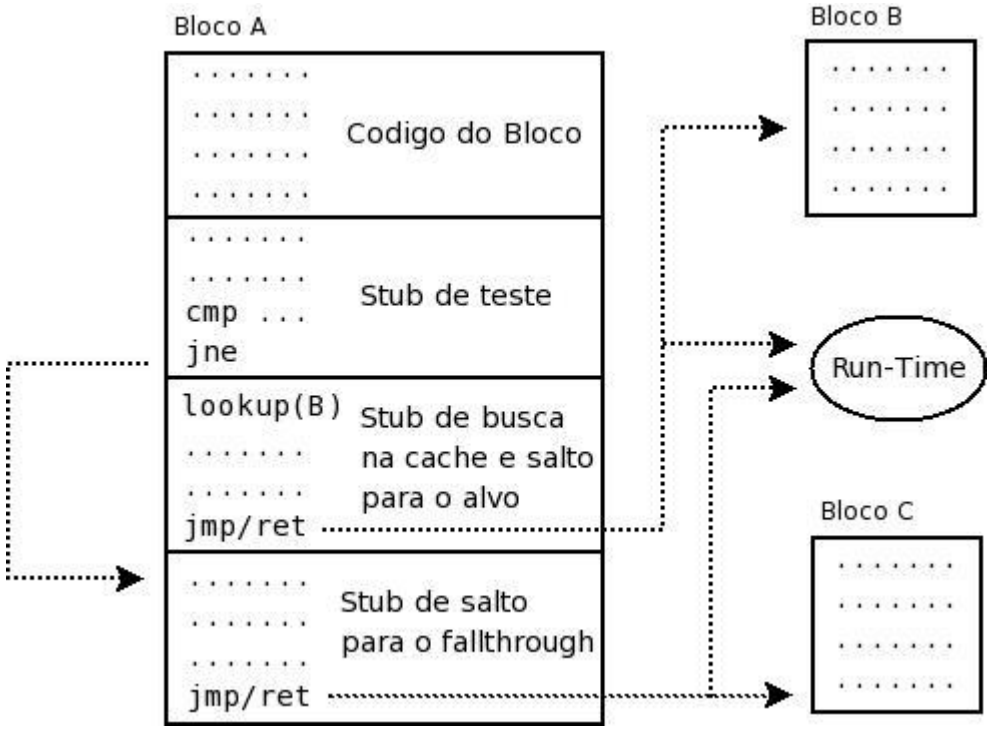


Figura 4.17: Ligação de blocos para *branchs* indiretos

4.6.5 Mapeamento de *System Calls*

As *system calls* do sistema operacional possuem algumas diferenças em sua implementação de uma arquitetura para outra. As principais diferenças são os números de identificação de algumas *system calls*, *endianness* dos parâmetros, formato das estruturas de dados e até mesmo a forma como os parâmetros são passados. No PowerPC todos os parâmetros de *system calls* são passados por registradores, enquanto que no x86 algumas *system calls* tem seus parâmetros passados em memória devido ao reduzido número de registradores.

A função do Mapeador de *System Calls* do ISAMAP é fazer as devidas conversões dos parâmetros de entrada e saída quando necessário para correta execução do programa traduzido. Em alguns casos é necessária a mudança de valores de alguns parâmetros, quando esses são constantes definidas pelo kernel, logo podendo ter valores diferentes para kernels de diferentes arquiteturas. Um exemplo de *system call* onde essa conversão é necessária é a `sys_ioctl`. Em outros casos o formato das estruturas de dados que servem como parâmentros ou retornos são diferentes, como é o caso das *system calls* `sys_fstat` e `sys_fstat64` onde as estruturas de dados `fstat` e `fstat64` possuem alinhamentos de campo diferentes em sua implementação para as arquitetura PowerPC e x86.

O Sistema de mapeamento de *System Calls* realiza processos de conversão, um anterior a chamada da *system call* e um posterior. O pré-processamento consiste em preparar o dados de entra da *system call* se necessário. O pós-processamento consiste em fazer as devidas conversões nos valores de retorno da *system call* se houver.

No caso da mapeamento entre PowerPC e x86, os seis primeiro parâmetros da *system calls* que estão no registradores R3 à R8 são copiados respectivamente para os registradores EBX, ECX, EDX, ESI, EDI, EBP. R0 que contém o número da *system call* é copiado para EAX. Após a execução da *system call* o retorno contido em EAX é copiado para R0. Para efetuar a chamada da *system call* é necessario criar uma *sandbox*, logo todos os registradores x86 envolvidos são salvos antes da chamada e restaurados logo após. Vale lembrar que os registradores PowerPc são representados em memória, de fato eles não existem como registradores reais.

4.7 Qualidade dos Mapeamentos

A forma como os mapeamentos são feitos influência drasticamento o desempenho final do código traduzido. Um exemplo, observado durante o projeto, é o mapeamento de instruções que alteram o registrador CR(*Condional Register*). Este registrador pode ser alterado por instruções aritméticas, lógicas e é sempre alterado em operações de comparação. Logo a qualidade do mapeamento de instruções que atualizam o registrador CR terá grande influência no desempenho. A figura 4.18 mostra um trecho de mapeamento

de uma instrução `cmp` que altera CR e a figura 4.19 mostra um outro mapeamento melhor para a mesma instrução.

```
0  isa_map_instrs {
1    cmp %imm %reg %reg;
2  } = {
  ...
3    mov_r32_r32 ecx src_reg(xer);
4
5    mov_r32_imm32 eax #0;
6
7    jnz_rel8 #6; // Setting CR[EQ] with ZF
8    lea_r32_disp32 eax eax #2;
9
10   jng_rel8 #6;   // Setting CR[GT] to 1 if ZF = 0 && SF = OF
11   lea_r32_disp32 eax eax #4;
12
13   jnl_rel8 #6;   // Setting CR[LT] to 1 if SF <> OF
14   lea_r32_disp32 eax eax #8;
15
16   and_r32_imm32 ecx #0x80000000;
17   jz_rel8 #6;
18   lea_r32_disp32 eax eax #1;
19
20   mov_r32_imm32 ecx #7;
21   sub_r32_imm32 ecx $0;
22   shl_r32_imm8 ecx #2;
23
24   shl_r32_cl eax;
25
26   mov_r32_imm32 esi #0x0000000f;
27   shl_r32_cl esi;
28   not_r32 esi;
29
30   mov_r32_r32 edi eax;
31
32   and_r32_r32 src_reg(cr) esi; // Reseting CR[crfD]
33   or_r32_r32 src_reg(cr) edi;
34 };
```

Figura 4.18: Exemplo de mapeamento instrução cmp.

```
 0  isa_map_instrs {
 1    cmp %imm %reg %reg;
 2  } = {
    ...
 3    mov_r32_m32disp ecx src_reg(xer);
 4
 5    jnl_rel8 #8; // L0
 6    mov_r32_imm32 eax cmpmask32($0, #0x80000000);
 7    jmp_rel8 #13; // L1
//L0:
 8    setg_r8 eax;
 9    movzx_r32_r8 eax eax;
 10   lea_r32_sib_disp8 eax eax eax #0 #2;
 11   shl_r32_imm8 eax shiftcr($0);
//L1:
 12   test_r32_imm32 ecx #0x80000000;
 13   jz_rel8 #6;
 14   or_r32_imm32 eax cmpmask32($0, #0x10000000);
 15
 16   and_r32_imm32 src_reg(cr) nniblemask32($0);
 17   or_r32_r32 src_reg(cr) eax;
 18 };
```

Figura 4.19: Exemplo de mapeamento melhorado instrução cmp.

O *Conditional Register* é dividido em 8 grupos de 4 bits. Para cada grupo de 4 bits o primeiro indica se o resultado da comparação é menor, o segundo indica se é maior, o terceiro indica se é igual e o quarto indica *overflow*. O mapeamento da figura 4.18 possui quatro instruções de jump para alterar ou não cada bit dos quatro possíveis. No entanto os valores dos três primeiros bits são mutuamente exclusivos, não há como um resultado ser maior, menor e igual ao mesmo tempo. Logo o trecho do mapeamento das linhas 7 a 18 pode ser resumido ao trecho das linhas 5 a 11 da figura 4.19. O mapeamento da figura 4.19 também possui menos instruções de salto, pois evitar ao máximo instruções de salto é uma das técnicas de otimização usada na arquitetura x86.

Outra coisa que vale notar é que a instrução `cmp` do PowerPC possui um parâmetro que indica qual dos 8 grupos do registrador CR será alterado. Este parâmentro é um valor imediato, logo sempre possui um mesmo valor para todas as execuções de uma determinada instrução. Portanto a máscara gerada pelas instruções das linhas 26 a 28 na figura 4.18 pode ser gerada durante a tradução da instrução pelo Tradutor. Para isso foi acrescentado a linguagem de descrição de mapeamentos a macro `nniblemask32()` que recebe como parâmetro o primeiro operando da instrução `cmp`. Em tempo de tradução esta macro gerará a máscara correta e a emitirá no corpo da instrução `and` da linha 16 da figura 4.19. Esta técnica permite descartar a necessidade das três instruções que construiam a máscara. No conjunto de instruções PowerPC existem outras instruções que constroem máscaras a partir de operandos imediatos, onde essa técnica também é aplicada.

4.8 Mapeamentos Condicionais

A pseudo-instrução `mr`, que faz cópia entre registradores, é implementada com uma instrução `or` onde os dois operandos fonte são o mesmo registrador (`or rx ry ry`). Um mapeamento da instrução `or` para x86 também serviria para mapear a pseudo-instrução `mr`. No entanto na arquitetura x86 existe uma instrução especifica para fazer copias de registradores, a instrução `mov`. Um mapeamento que utilize essa instrução com certeza é um mapeamento melhor do que usando a instrução `or` do x86. Então para tornar o mapeamento mais flexível, a sintaxe da linguagem de mapeamento permite especificar dois mapeamentos para uma instrução da arquitetura origem utilizando uma estrutura `if-then-else`.

A figura 4.20 mostra um exemplo de mapeamento da instrução `or` PowerPC que considera os dois usos dessa instrução. Quando os dois operando fonte são iguais o tradutor emitira o mapeamento contido no bloco `if` senão o tradutor emitira o mapeamento contido no bloco `else`. Então sempre quando for encontrado uma instrução `or` PowerPC que apenas faz cópias entre registradores, será emitido um mapeamento x86 com uma instrução a menos.

```
isa_map_instrs {
  or %reg %reg %reg;
} = {
  if(rs = rb) {
    mov_r32_m32disp edi $1;
    mov_m32disp_r32 $0 edi;
  }
  else {
    mov_r32_m32disp edi $1;
    or_r32_m32disp edi $2;
    mov_m32disp_r32 $0 edi;
  }
};
```

Figura 4.20: Mapeamento instrução `or`.

A mesma abordagem é utilizada na instrução `rlwinm`. Esta instrução rotaciona o conteúdo de um registrador e depois realiza uma operação lógica `and` com uma máscara especificada. Em alguns casos o imediato que indica em quantos bits o conteúdo do registrador deve ser rotacionado, tem valor 0. Neste caso nenhuma rotação precisa ser feita, logo possibilitando um mapeamento mais eficiente nesses casos particulares. A figura 4.21 mostra um exemplo de mapeamento para a instrução `rlwinm`.

```
isa_map_instrs {
  rlwinm %reg %reg %imm %imm %imm;
} = {
  if(sh = 0) {
    mov_r32_m32disp edi $1;
    and_r32_imm32 edi mask32($3, $4);
    mov_m32disp_r32 $0 edi;
  }
  else {
    mov_r32_m32disp edi $1;
    rol_r32_imm8 edi $2;
    and_r32_imm32 edi mask32($3, $4);
    mov_m32disp_r32 $0 edi;
  }
};
```

Figura 4.21: Mapeamento instrução `rlwinm`.

Um outro exemplo é a instrução `addi`. Quando o primeiro operando fonte dessa instrução é o registrador R0, então esta instrução apenas carrega o imediato para o registrador destino. A figura 4.22 mostra o mapeamento da instrução `addi`.

```
isa_map_instrs {
  addi %reg %reg %imm;
} = {
  if(ra = 0)
  {
    mov_m32disp_imm32 $0 $2;
  }
  else
  {
    mov_r32_m32disp edi $1;
    add_r32_imm32 edi $2;
    mov_m32disp_r32 $0 edi;
  }
};
```

Figura 4.22: Mapeamento instrução `addi`.

Este esquema de mapeamento condicional é utilizado em vários outros mapeamentos de instruções PowerPC. Em vários casos contribuindo para confecção de mapeamentos mais enxutos.

A experiência na confecção desses mapeamentos de instrução mostrou que, a qualidade do código traduzido é determinada pelo conhecimento do conjunto de instruções das arquiteturas envolvidas. Pessoas com grande conhecimento do conjunto de instruções conseguem produzir mapeamentos altamente eficientes. É necessário também, muito cuidado para elaborar mapeamentos de instruções em tradutores binários, pois um único mapeamento errado pode ocasionar a execução totalmente errônea de um programa. Encontrar erros de mapeamento de instruções isoladas é trabalhoso e demorado devido a não existência de ferramentas próprias para auxiliar a depuração de programas que executam sobre tradutores binários.

Capítulo 5

Extensão na linguagem ArchC

O sistema de geração de tradutores foi construido a partir do parse do ArchC. Portanto a sintaxe e semântica é práticamente a mesma utilizada na descrição de modelos ArchC com algumas pequenas alterações e algumas extensões. A tabela 5.1 mostra as extensões à linguagem ArchC que foram feitas para adequá-la ao proposito de tradução binária.

Tabela 5.1: Extensões na linguagem ArchC

palavra chave	função
<instr>.set_operands	define os operandos da instrução e os campos correspondentes do formato
<instr>.set_type	define uma semântica para a instrução: salto, ponto flutuante e outros.
<instr>.set_read	define campos somente leitura de uma instrução
<instr>.set_write	define campos somente escrita de uma instrução
<instr>.set_readwrite	define campos leitura e escrita de uma instrução
isa_mem	reserva uma região de memória para uso geral dos mapeamentos
isa_map_instrs	define o mapeamento de instrução entre arquitetura origem e arquitetura alvo
src_reg(<reg>)	especifica que a região de memória que representa o registrador `reg` da arquitetura deve ser referenciada na instrução
src_fpreg(<fpreg>)	o mesmo que `src_reg` para registradores de ponto flutuante
<op> << <num>	realiza uma operação de *shift left* sobre o operando imediato `op` de `num` bits

<op> >> <num>	o mesmo que <<, mas realizando operação de *shift right*
<op> + <num>	o mesmo que <<, mas realizando operação de adição
<op> − <num>	o mesmo que <<, mas realizando operação de subtração
mask32(<op1>, <op2>)	emite uma máscara de 32 bits onde os bits de op1 a op2 são 1's e o restante são 0's
nmask32(<op1>, <op2>)	emite uma máscara de 32 bits onde o resultado é o complemento de um sobre o retorno de mask32
bitmask32(<op>,	emite uma máscara de 32 bits construida pela expresão em C: 0x80000000 >> op
nbitmask32(<op>,	emite uma máscara de 32 bits construida pela expresão em C: (0x80000000 >> op)
rmask32(<op>,	emite uma máscara de 32 bits construida pela expresão em C: (0xffffffff << op)
lmask32(<op>,	emite uma máscara de 32 bits construida pela expresão em C: (0xffffffff >> op)
cmpmask32(<op>, <num>)	emite uma máscara de 32 bits construida pela expresão em C: num >> (op * 4)
niblemask32(<op>)	emite uma máscara de 32 bits construida pela expresão em C: 0xf0000000 >> (op * 4)
nniblemask32(<op>)	emite uma máscara de 32 bits construida pela expresão em C: (0xf0000000 >> (op * 4))

Abaixo segue alguns exemplos das extensões utilizadas nos modelos de descrição e mapeamento:

```
add_r32_r32.set_operands("%reg %reg", rm, regop);
add_r32_r32.set_encoder(op1b=0x01, mod=0x3);
add_r32_r32.set_readwrite(rm);
```

Figura 5.1: Exemplo set_operands, set_readwrite

```
movss_xmm_m32.set_operands("%fpreg %imm %reg", regop_5, disp32_7, rm_6);
movss_xmm_m32.set_encoder(op3b1=0xf3, op3b2=0x0f, op3b3=0x10, mod_4=0x2);
movss_xmm_m32.set_write(regop_5);
movss_xmm_m32.set_type("fp");
movss_xmm_m32.set_semantic(LOAD);
```

Figura 5.2: Exemplo set_type, set_semantic, set_write

```
isa_map_instrs {
  mflr %reg;
} = {
  mov_r32_m32disp edi src_reg(lr);
  mov_m32disp_r32 $0 edi;
};
```

Figura 5.3: Exemplo src_reg

```
isa_map_instrs {
  xoris %reg %reg %imm;
} = {
  mov_r32_imm32 edi $2 << 16;
  xor_r32_m32disp edi $1;
  mov_m32disp_r32 $0 edi;
};
  ...
  not_r32 edi;
  add_r32_imm32 edi $2 + 1;
  mov_m32disp_r32 $0 edi;
  ...
```

Figura 5.4: Exemplo operação de *shift* e adição

```
isa_map_instrs {
  mtfsb1 %imm;
} = {
  mov_r32_imm32 edi bitmask32($0);
  movd_xmm_r32 xmm7 edi;
  orpd_xmm_xmm src_fpreg(fpscr) xmm7;
};
```

Figura 5.5: Exemplo de macro para geração de máscara

Capítulo 6

Resultados

Atualmente o ISAMAP traduz e executa corretamente os seguintes *benchmarks* do SPEC CPU 2000: 164.gzip, 175.vpr, 176.gcc, 181.mcf, 186.crafty, 197.parser, 252.eon, 254.gap, 256.bzip2, 300.twolf, 168.wupwise, 171.swim, 172.mgrid, 173.applu, 177.mesa, 178.galgel, 179.art, 183.equake, 187.facerec, 188.ammp, 189.lucas, 191.fma3d, 301.apsi.

6.1 Metodologia

Os resultados são apresentados da seguinte forma: desempenho do ISAMAP comparado ao desempenho do QEMU e desempenho do ISAMAP comparado ao desempenho do código x86 nativo e em alguns casos o desempenho entre versões diferentes do ISAMAP. Valores positivos significam *speedup* e negativos *slowdown* obtidos pelo ISAMAP. Foi usando a base de dados *reference* do SPEC CPU 2000 em um computador com processador Intel Core^{TM}2 Duo 2.66 Ghz, 1 GB RAM, Fedora Core 3. O código PowerPC foi gerado com *cross compilers* gcc-3.4.5 e gfortran-4.1.0, o código x86 foi gerado com os compiladores gcc-3.4.5 e gfortran-4.1.0. Apesar da versão 3.4.5 do GCC ser antiga, esta foi a única versão que compilou corretamente todos os *benchmarks* do SPEC CPU 2000.

Os valores de *speedup* e *slowdown* são obtidos da seguinte forma: Cada benchmark é executado três vezes e calculada a média aritmética. Em um comparativo de A versus B o *speedup/slowdown* de A é definido por $\frac{Tm(B)-Tm(A)}{Tm(B)}$, onde Tm é o tempo médio.

6.2 Comparativos

As figuras 6.1 e 6.2 mostram o desempenho do ISAMAP utilizando o mapeamento mais eficiente obtido até o momento em relação ao desempenho do QEMU. Os registradores PowerPC CR e XER são representados por vetores de bytes.

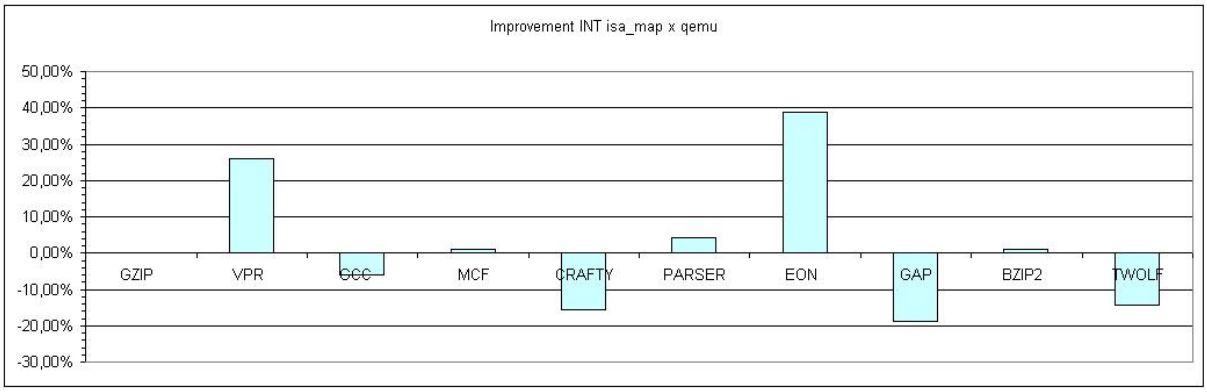


Figura 6.1: Comparação ISAMAP em relação ao QEMU SPEC INT

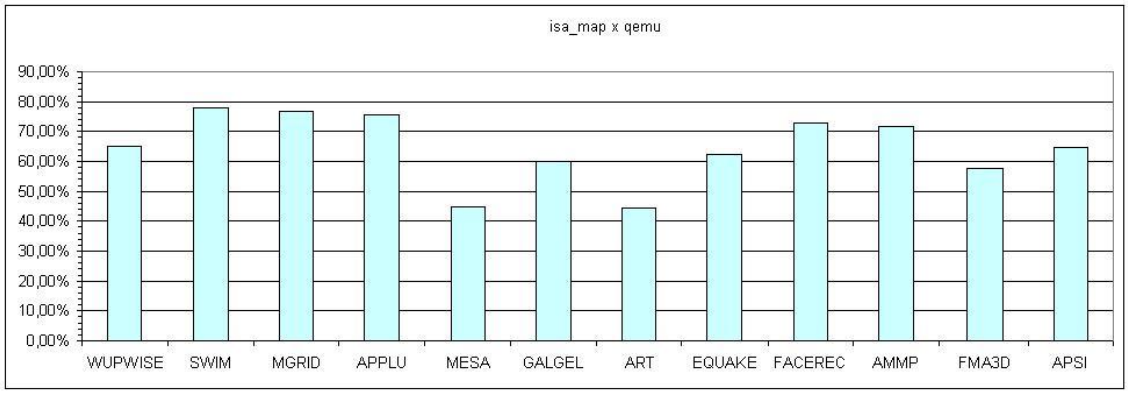


Figura 6.2: Comparação ISAMAP em relação ao QEMU SPEC FP

O *slowdown* obtido no gcc pode ser explicado pelo fato do *benchmark* gcc possuir muito código a ser traduzido, como visto na tabela B.1 nos apêndices desse trabalho. Como o tradutor utilizado pelo ISAMAP possui uma implementação genérica para possibilitar a tradução de diversas ISA, este possui desempenho inferior ao tradutor encontrado no QEMU. O sistema de tradução do QEMU possui uma implementação especificamente direcionada a tradução do conjunto de instruções do PowerPC.

O *benchmark* eon também possuem código relativamente volumoso. No entanto, esse *benchmark* possui tempo de execução bem maior que o do gcc, logo o tempo gasto em tradução é relativamente insignificante comparado ao tempo de execução do código traduzido.

Para o *slowdown* observado nos *benchmarks* crafty, gap e twolf, não se encontrou explica ção até o momento, deixando como alvo de pesquisa para trabalhos futuros. Pensava-

se que tal *slowdown* seria devido a mapeamento ineficiente de alguma determinada instrução muito executada. No entanto quando se observa os gráficos das instruções mais executadas (Apêndice A) de cada *benchmark* não encontra-se nenhuma instrução muito executada nesses *benchmarks* que não esteja em outros *benchmarks*, que não apresentam o mesmo *slowdown*. Um exemplo é o grafico de instruções do parser A.6 que possui quase as mesmas instruções que o gráfico do gap A.8, no entanto o parser apresenta um *speedup* de quase 5%.

O *speedup* obtido nos *benchmarks* vpr, eon e todos do SPECFP é explicado pelo mapeamento das instruções de ponto flutuante feito pelo ISAMAP. Os mapeamentos feitos utilizam instruções SSE para mapear as instruções de ponto flutuante do PowerPC, enquanto o QEMU utiliza instruções x87. As instruções x87 necessitam o empilhamento de operandos e ajustes no topo da pilha de registradores x87 o que torna as instruções x87 ineficientes quando comparadas as instruções SSE.

As figuras 6.3 e 6.4 mostram o desempenho do ISAMAP em relação ao desempenho do código nativo x86 com instruções SSE.

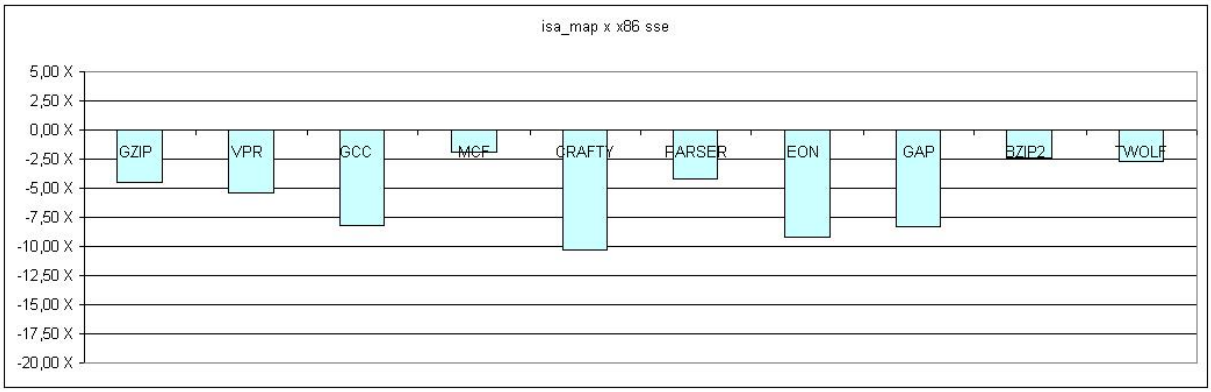


Figura 6.3: Slowdown do ISAMAP em relação a código nativo x86 SPEC INT

Nas figuras 6.5 e 6.6 é feito um comparativo entre um mapeamento do ISAMAP, que aloca estaticamente os registradores CR e XER para registradores x86, e o mapeamento original do ISAMAP sem alocação estática. A alocação estática de CR e XER é feita reservando-se os registradores x86 EBX e EBP para fazerem o papel de XER e CR. Com isso, todo referência a XER e CR será traduzida para uma referência a EBX e EBP, não necessitando acessos a memória.

A princípio acreditava-se que o fato de alocar registradores x86 para representar os registradores CR e XER do PowerPC melhoraria o desempenho, mesmo que ligeiramente. No entanto o resultado é o oposto, na maioria dos *benchmarks* o desempenho piora. Acredita-se que isso é provocado devido aos novos projetos dos processadores x86 que

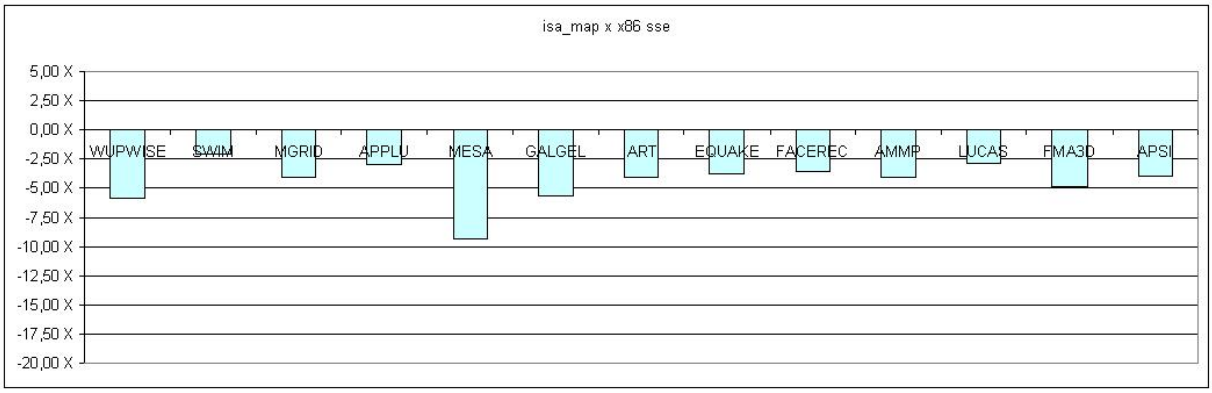


Figura 6.4: Slowdown do ISAMAP em relação a código nativo x86 SPEC FP

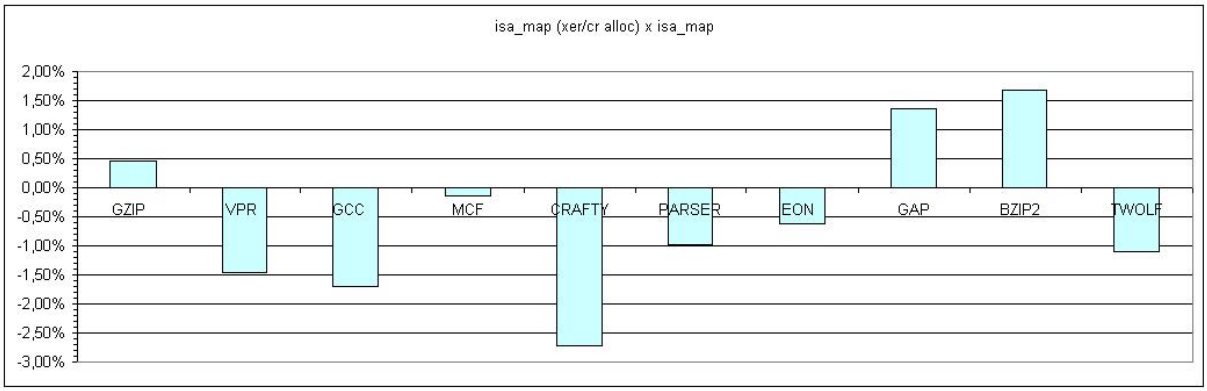


Figura 6.5: Comparação entre ISAMAP com alocação estática e sem alocação estática SPEC INT

utilizam truques arquiteturais para agilizar o acesso à memória [14] e também pela enorme quantidade de memória cache nos processadores atuais. A implementação com alocação estatica exige algumas instruções a mais, pois uma alteração nos bits do CR exige duas instruções no mínimo, um `or` e um `and`. Quando se utiliza a implementação de CR e XER como vetores a alteração dos bits é simplificada para um instrução `mov` na maioria dos mapeamentos. A figura 6.7 mostra a diferença nos mapeamentos das duas abordagens. A macro bit emite o endereço da posição 0 do vetor que representa CR. Como pode ser observado, cria-se uma dependência entre as instruções `and` e `or`, pois as duas alteram o conteúdo de EBP. Isto também contribui para o desempenho inferior do mapeamento com alocação estática.

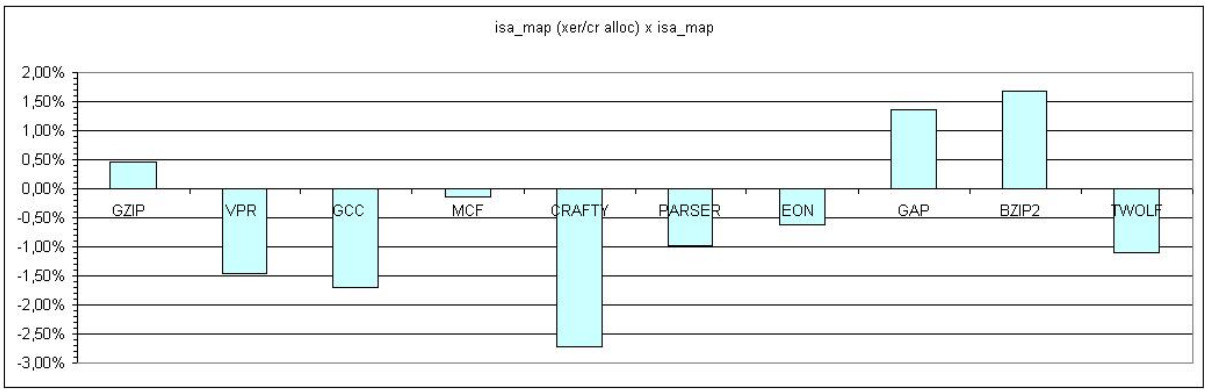


Figura 6.6: Comparação entre ISAMAP com alocação estática e sem alocação estática SPEC FP

```
Com alocação estática                Sem alocação estática

...                                  ...
and_r32_imm32 ebp #0x0fffffff;       mov_m32disp_r8 bit(cr, #0) eax;
or_r32_r32 ebp eax;                  ...
...
```

Figura 6.7: Com alocação estática versus Sem alocação estática

Capítulo 7

Conclusões

Com o surgimento de novas arquiteturas 64 bits e multi-core sistemas de tradução e emulação tornam-se ferramentas úteis na execução de código legado sobre as novas arquiteturas. O que possibilita que esses códigos utilizem os recursos a mais, disponíveis nos processadores novo. Estes sistemas de tradução binária também tornam possível a aplicação de otimizações não feitas em tempo de compilação pela falta de informações sobre o comportamento do programa, com o auxílio de técnicas de *profiling*.

Este trabalho apresenta o ISAMAP, um Sistema de Tradução Binária orientado a modelos de alto desempenho. Em *benchmarks* de ponto fluante o ISAMAP tem ganhos de desempenho de quase 80% em relação ao desempenho do QEMU [3]. O desempenho em relação a código nativo x86 fica entre 2 a 11 vezes mais lento, o que é muito bom considerando que o ISAMAP não faz otimização de código.

O mapeamento a nível de instruções mostrou-se eficiente quando feito por pessoas com considerável conhecimento do conjunto de instruções das arquiteturas desenvolvida e de rápida confecção (menos de duas semanas). Os mapeamentos são também de fácil e rápida implementação considerando a simplicidade da linguagem ArchC *like* utilizada. No entanto a abordagem utilizada perde em portabilidade se comparada as soluções adotadas por QEMU [3] e Yirr-MA [9]. Onde os mapeamentos são descritos em níveis de abstração mais elevados.

7.1 Trabalhos Futuros

Otimizações dinâmicas tem um grande potencial para melhorar o desempenho de códigos legados, principalmente com todas as informações disponíveis quando tem-se um sistema de tradução binária ou emulação executando a aplicação. Vários trabalhos como [7], [10], [13] desenvolveram implementações eficientes e com ganhos de desempenho aplicando otimizações dinâmicas. Outros trabalhos como [20] e [11] apresentam técnicas de

paralelização dinâmica que podem proporcionar grandes ganhos de desempenho.

Como trabalhos futuros pretende-se a implementação de técnicas de otimização e paralelização dinâmica em códigos gerados por tradutores binários / emuladores. Principalmente devido a popularização das arquiteturas multi-core, que exigem aplicações *multi-thread* para proporcionar o alto desempenho prometido. Outros possíveis trabalhos futuros seria a implementação de mapeamentos para outras arquiteturas e confecção de ferramentas para importar modelos de descrição de arquiteturas já existentes para ArchC. O que permitiria testar o quanto de portabilidade a técnica de tradução binária proposta nesse trabalho oferece.

Apêndice A

Frequência de Execução de instruções PowerPC

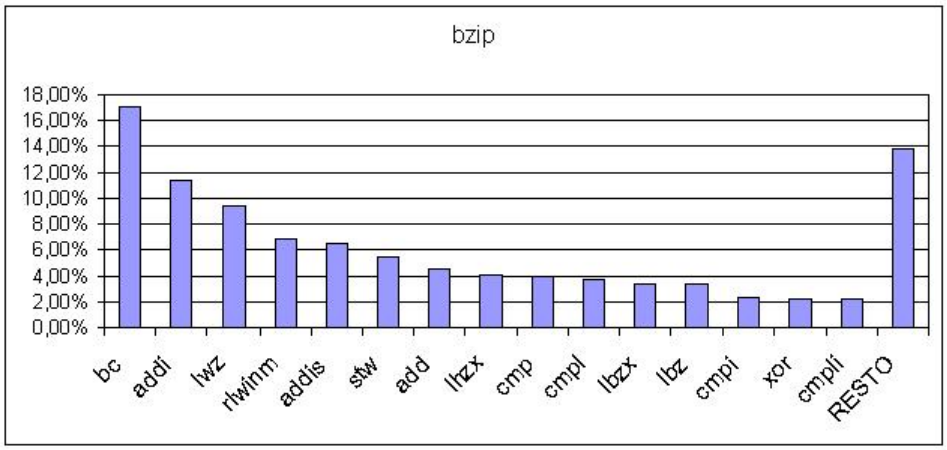


Figura A.1: gzip

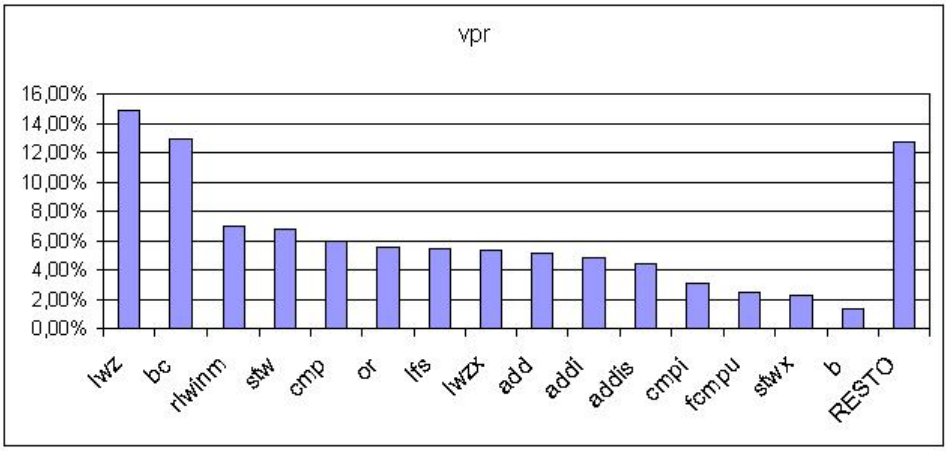


Figura A.2: vpr

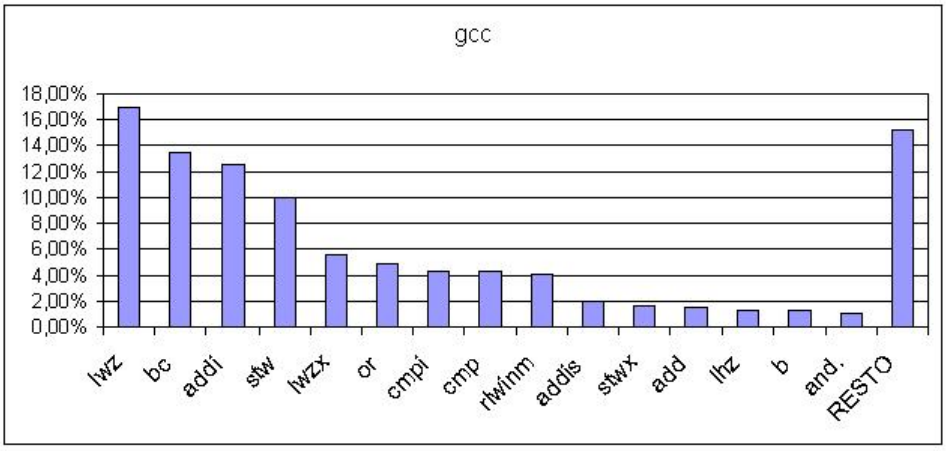


Figura A.3: gcc

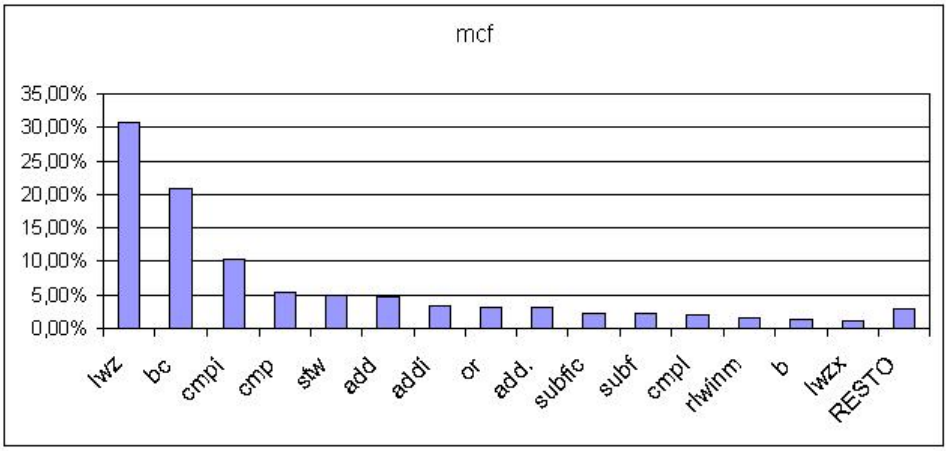


Figura A.4: mcf

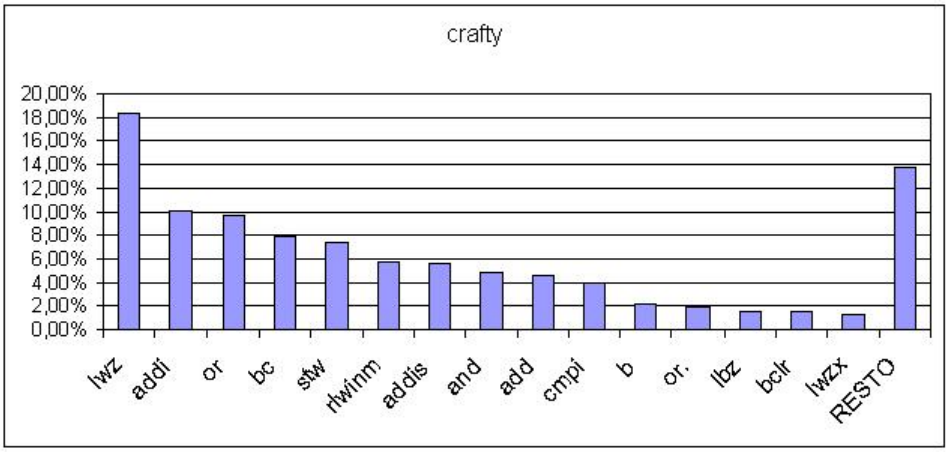


Figura A.5: crafty

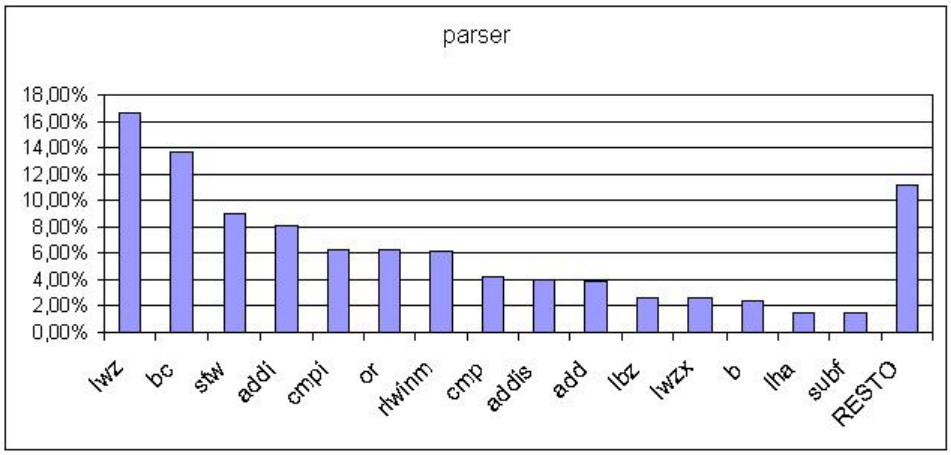


Figura A.6: parser

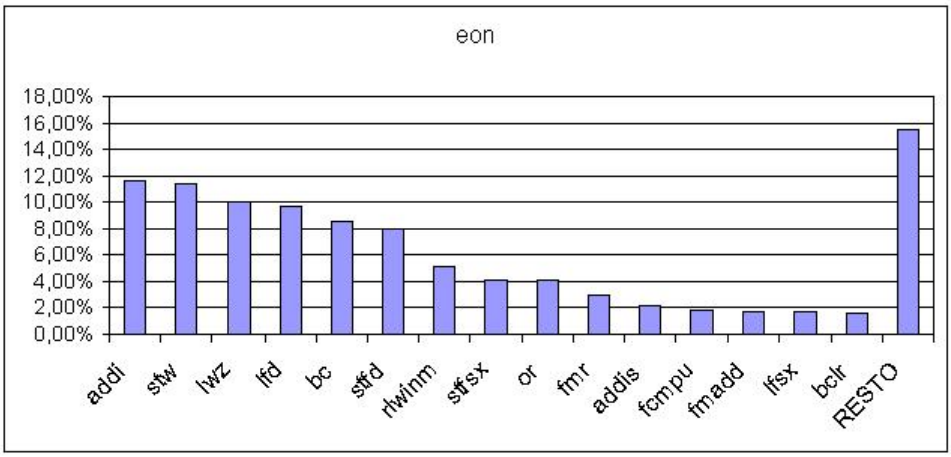


Figura A.7: eon

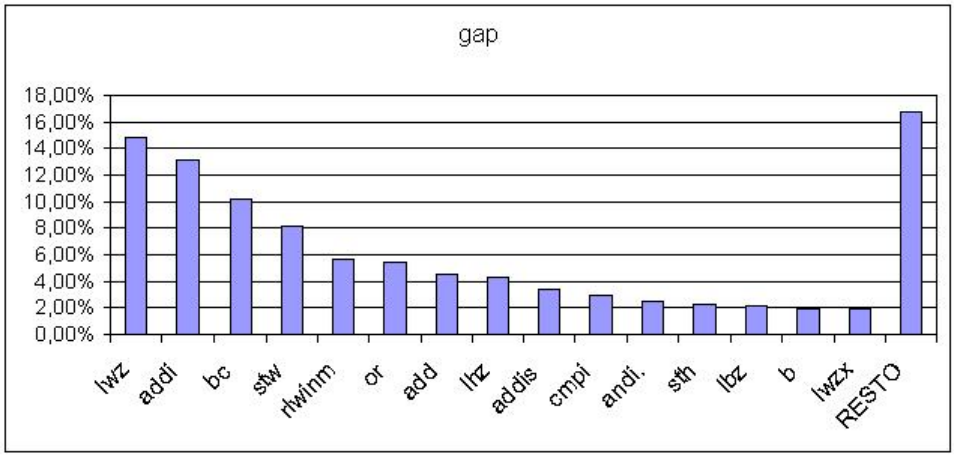


Figura A.8: gap

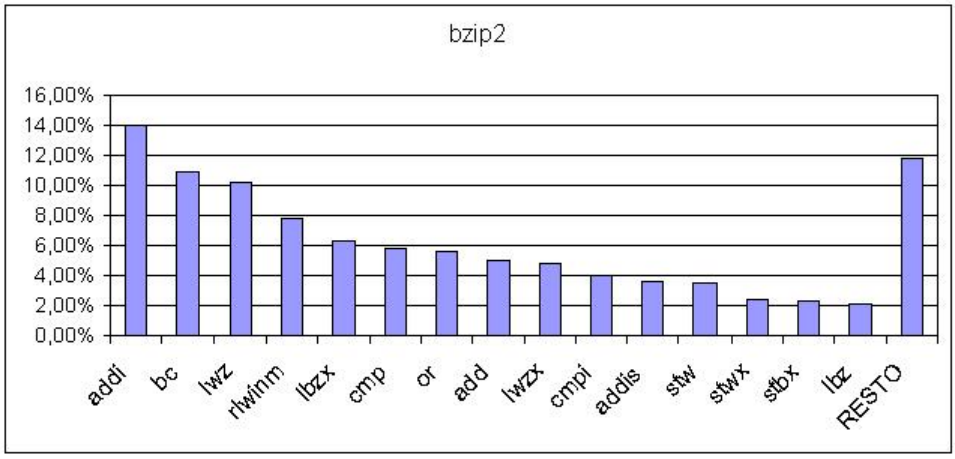


Figura A.9: bzip2

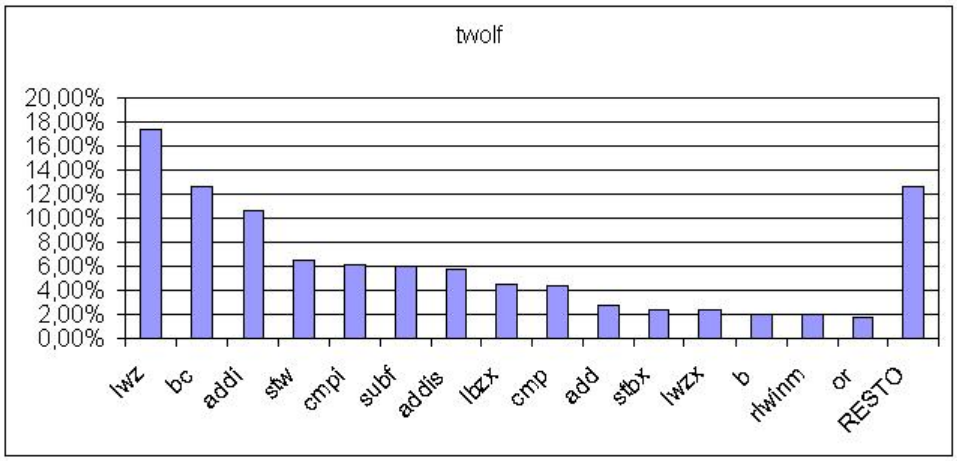


Figura A.10: twolf

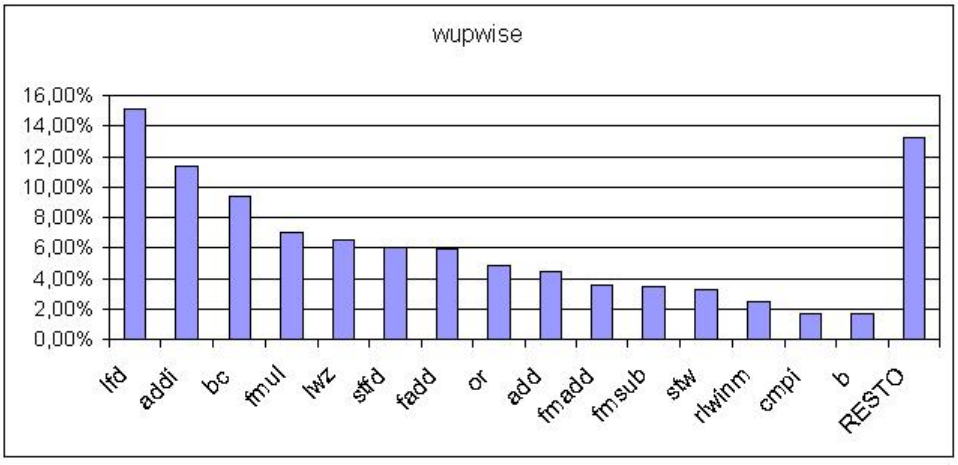


Figura A.11: wupwise

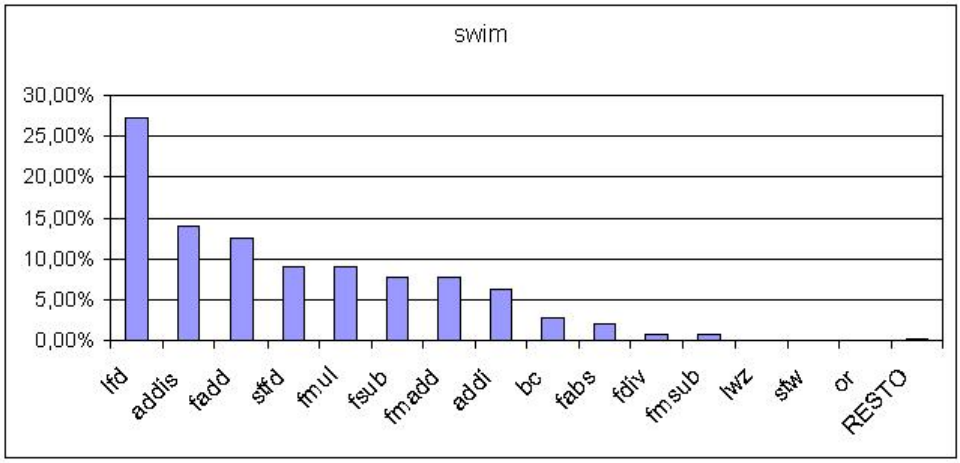


Figura A.12: swim

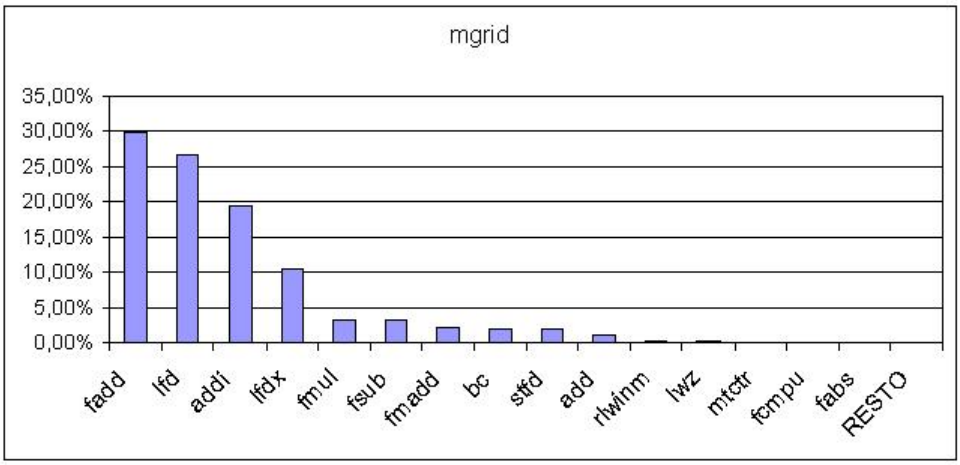


Figura A.13: mgrid

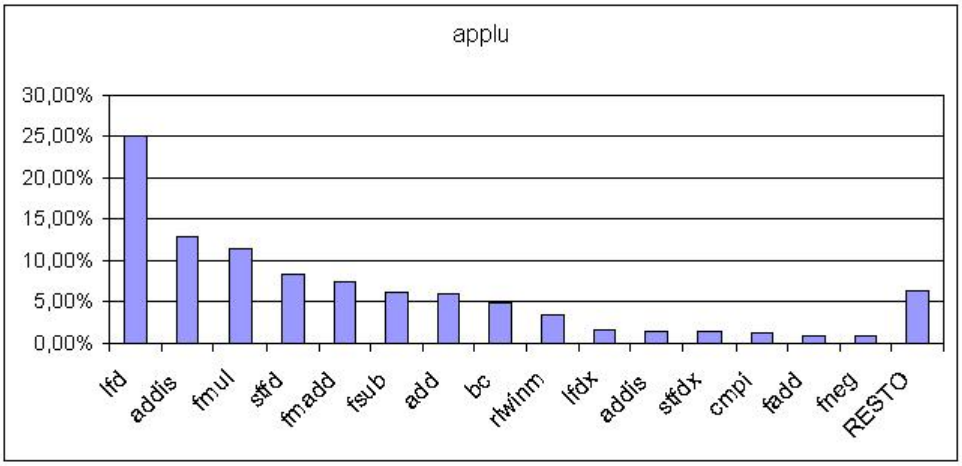


Figura A.14: applu

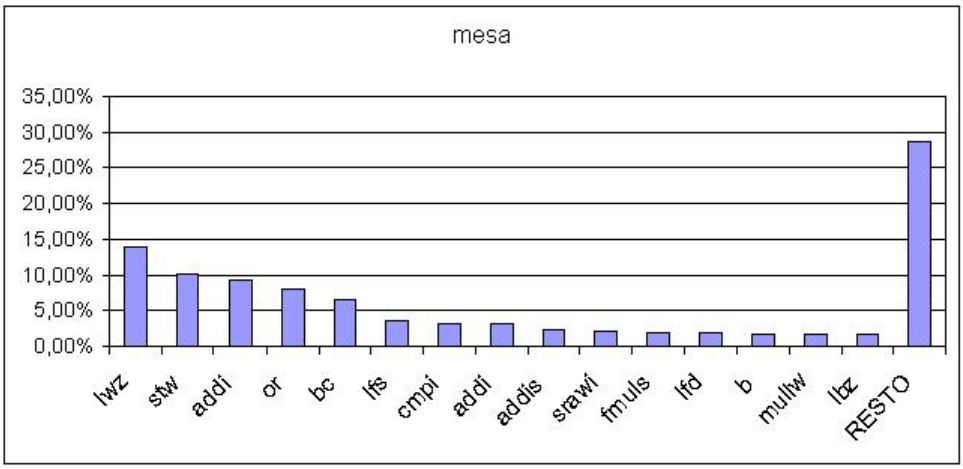


Figura A.15: mesa

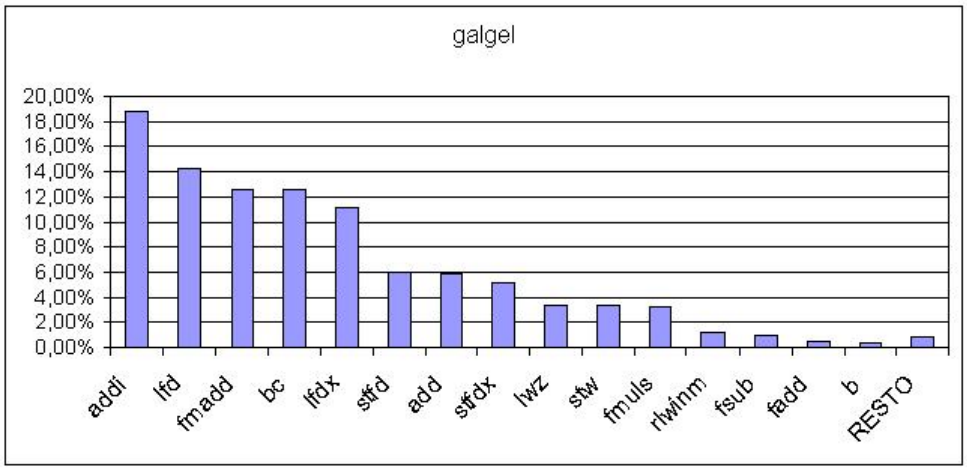


Figura A.16: galgel

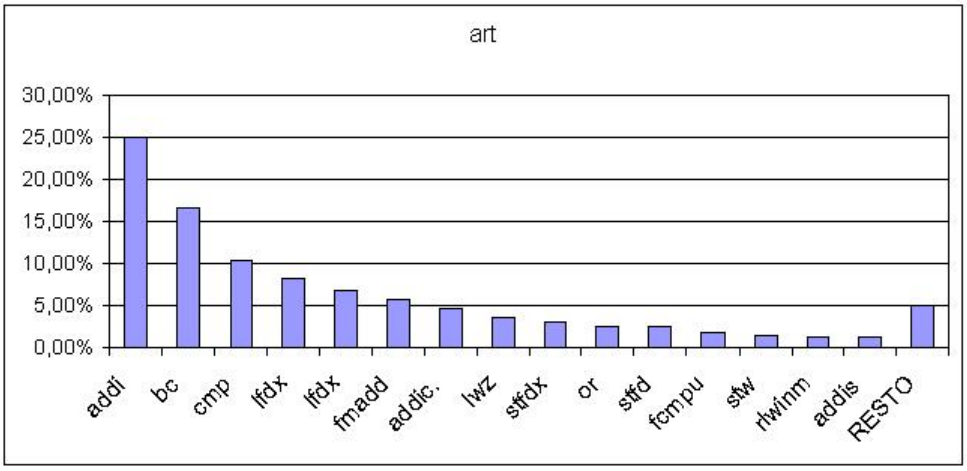


Figura A.17: art

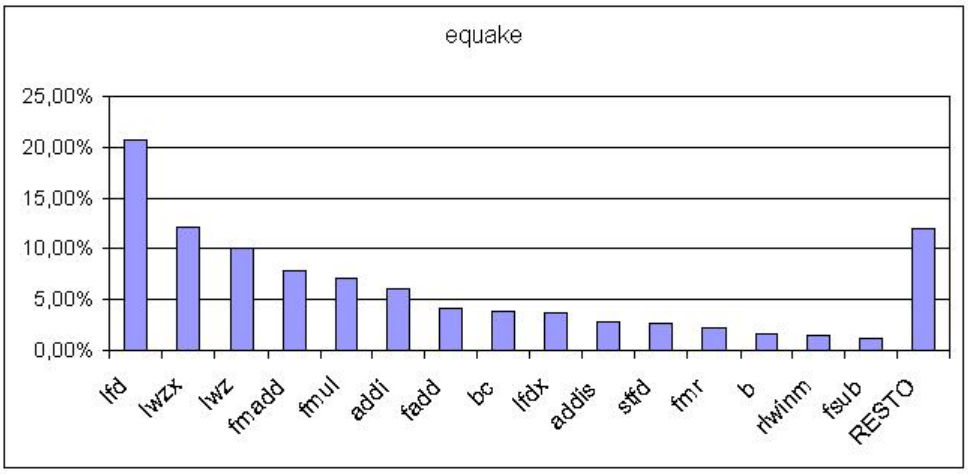


Figura A.18: equake

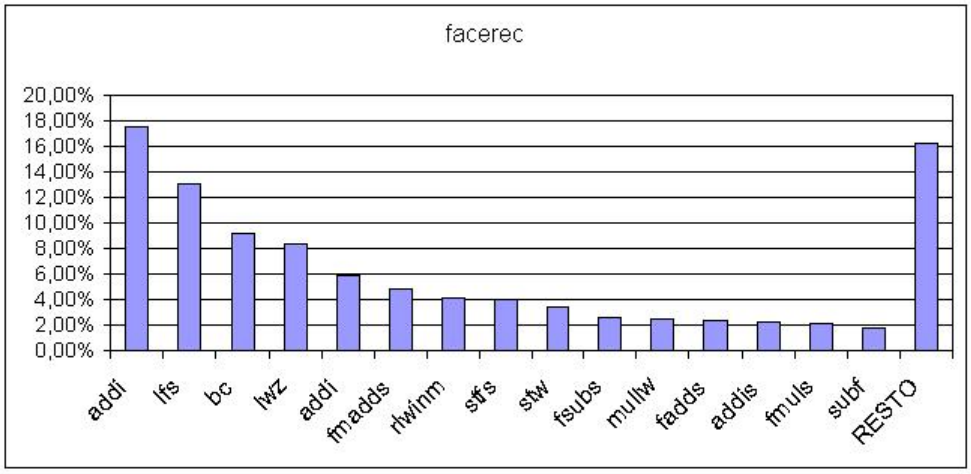


Figura A.19: facerec

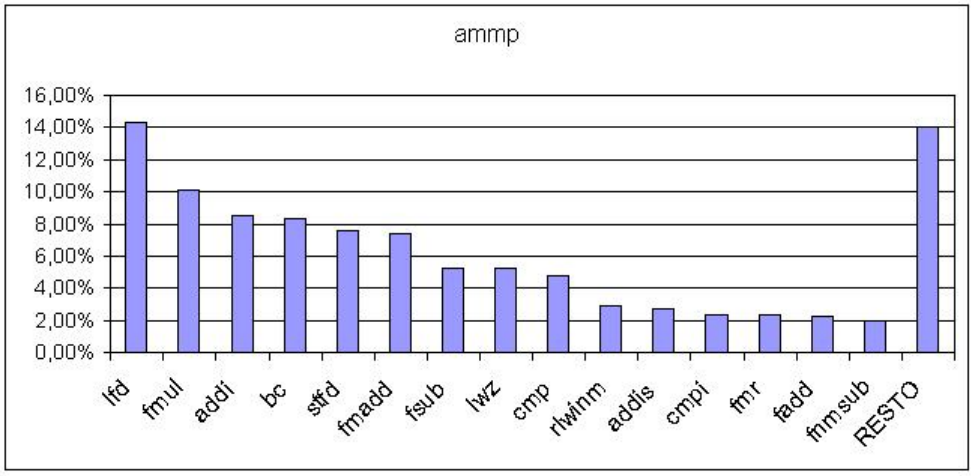


Figura A.20: ammp

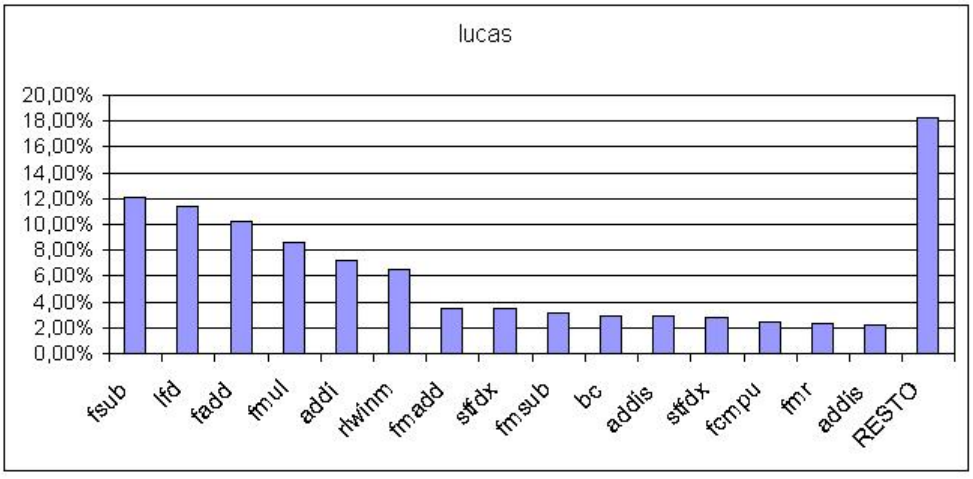


Figura A.21: lucas

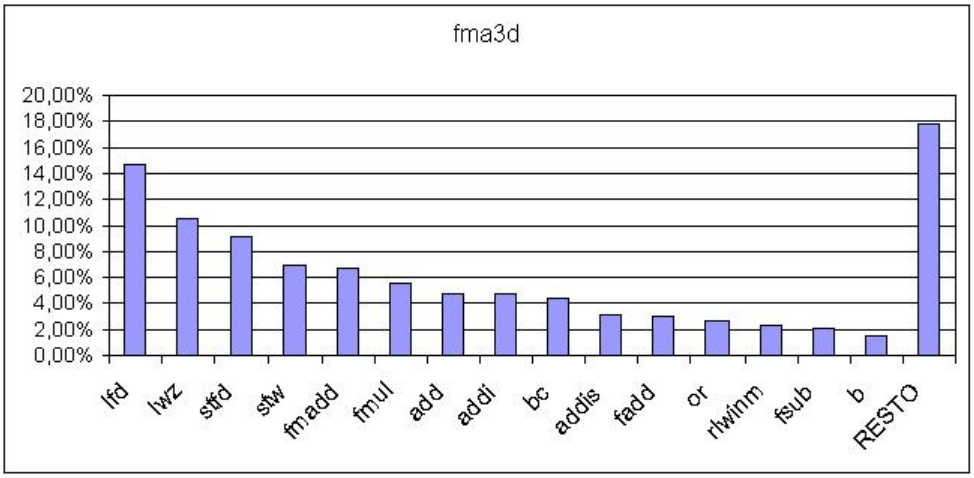


Figura A.22: fma3d

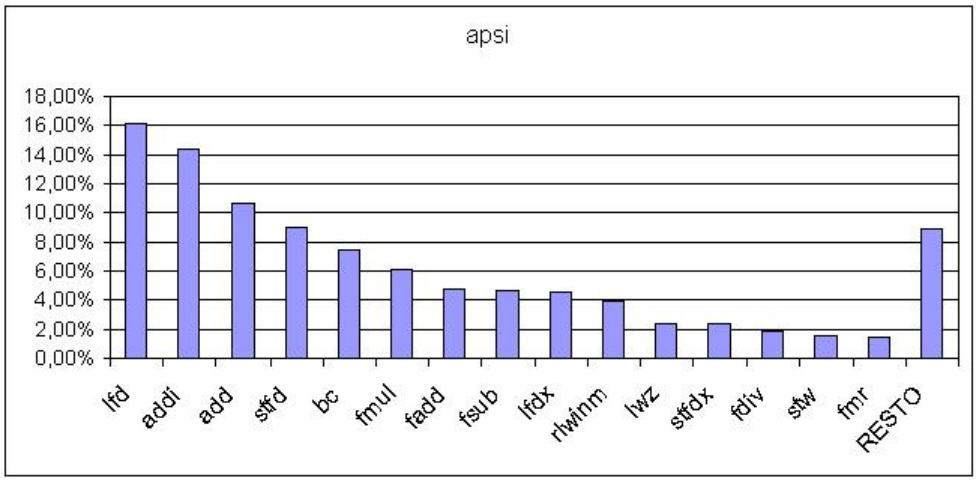


Figura A.23: apsi

Apêndice B

Tamanho do Código Traduzido

Tabela B.1: Tamanho Código Traduzido

Benchmark	Run	Tamanho do Código em bytes
gzip	1	182313
	2	176938
	3	174052
	4	175307
	5	175146
vpr	1	419393
	2	603476
gcc	1	3655746
	2	3463874
	3	3509540
	4	3281204
	5	3502833
mcf	1	170179
crafty	1	810753
parser	1	600255
eon	1	1258904
	2	1281677
	3	1274061
gap	1	866480
bzip2	1	203898
	2	203556
	3	204511
twolf	1	712699
wupwise	1	355566

Benchmark	Run	Tamanho do Código em bytes
swim	1	438684
mgrid	1	382890
applu	1	770416
mesa	1	504455
galgel	1	1032826
art	1	170370
	2	171705
equake	1	338905
facerec	1	499734
ammp	1	628487
lucas	1	591633
fma3d	1	945161
apsi	1	838278

Referências Bibliográficas

[1] Intel®Extended Memory 64 Technology. Technical report, Intel Corp. http://www.intel.com/technology/64bitextensions acessado em 23 de Agosto, 2006.

[2] Linux Standard Base Specification for the PPC32 Architecture 1.3. Technical report, Free Standards Group. http://refspecs.freestandards.org/LSB_1.3.0/PPC32/spec.html acessado em 15 de Janeiro, 2007.

[3] QEMU - Open Source Processor Emulator. Technical report. http://fabrice.bellard.free.fr/qemu/qemu-tech.html acessado em 2 de Abril, 2007.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent Dynamic Optimization System. *SIGPLAN PLDI*, pages 1–12, June 2000.

[5] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigal Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. San Diego CA, December 2003. 6th International Conference on Microarchitecture (MICRO36).

[6] A. Chernoff and R. Hookway. DIGITAL FX!32 - Running 32-Bit x86 Applications on Alpha NT. Berkeley CA, August 1997. USENIX Association, Proceedings of the USENIX Windows NT Workshop.

[7] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, , and J. Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address reallife challenges. pages 15–24. In International Symposium on Code Generation and Optimization. IEEE Computer Society, 2003.

[8] K. Ebcioglu and E. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. Denver CO, June 1997. Proc. of the 24th International Symposium on Computer Architecture (ISCA).

[9] Ptroger J. and Gough J. Fast dynamic binary translation the yirr-ma framework. In Proceedings of the 2002 Workshop on Binary Translation, 2002.

[10] Shu Xu Bo Huang Jianhui Li, Qi Zhang. Optimizing Dynamic Binary Translation for SIMD Instructions. Proceedings of the International Symposium on Code Generation and Optimization CGO 06, March 2006.

[11] Christopher Kirkham Jisheng Zhao and Ian Rogers. Lazy Interprocedural Analysis for Dynamic Loop Parallelization. Workshop on New Horizons in Compilers, Bangalore, India, December 2006.

[12] Pen-Chung Yew Jiwei Lu, Howard Chen and Wei-Chung Hsu. Design and Implementation of a Lightweight Dynamic Optimization System. AI Access Foundation and Morgan Kaufmann Publishers., 2004.

[13] J. Lu, H. Chen, P.-C. Yew, , and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. *The Journal of Instruction-Level Parallelism*, 6, 2004.

[14] Darek Mihocka. NO EXECUTE! A weekly look at personal computer tecnology issues. Technical report.

[15] Rodolfo J. Azevedo Sandro Rigo and Guido Araujo. The ArchC Architecture Description Language. Technical report, Universidade Estadual de Campinas, 2003.

[16] Palmer T., Zovi D. D., and Stefanovic D. SIND: A framework for binary translation. Tech. rep. tr-cs-2001-38, Department of Computer Science, University of New Mexico, December 2001.

[17] Vijay Janapa Reddi Dirk Grunwald Tipp Moseley, Alex Shye and Ramesh Peri. Shadow Profiling: Hiding Instrumentation Costs with Parallelism. Proceedings of the International Symposium on Code Generation and Optimization CGO 07, March 2007.

[18] D. Ung and C. Cifuentes. Optimising hot paths in a dynamic binary translator. In Workshop on Binary Translation, October 2000.

[19] Steven Wallace and Kim Hazelwood. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. Proceedings of the International Symposium on Code Generation and Optimization CGO 07, March 2007.

[20] Efe Yardimci and Michael Franz. Dynamic Parallelization and Mapping of Binary Executables on Hierarchical Platforms. Proceedings of the 3rd conference on Computing frontiers CF 06, May 2006.

[21] Cindy Zheng and Carol Thompson. PA-RISC to IA-64: Transparent Execution, No Recompilation. *IEEE Computer*, 33(3):47–53, March 2000.