UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

JAIR FAJARDO JUNIOR

Sistema de Tradução Binária de Dois Níveis para Execução Multi-ISA

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Microeletrônica

Prof. Dr. Luigi Carro
Orientador
Prof. Dr. Antonio Carlos S. Beck Filho
Co-orientador

Porto Alegre, agosto de 2011.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Fajardo Jr, Jair

Sistema de Tradução Binária de Dois Níveis para Execução Multi-ISA / Jair Fajardo Junior – Porto Alegre: Programa de Pós-Graduação em Microeletrônica, 2011.

76 p.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica. Porto Alegre, BR – RS, 2011. Orientador: Luigi Carro.

1.Tradução Binária 2.Arquiteturas Reconfiguráveis 3.Sistemas Embarcados I. Carro, Luigi. II. Beck Filho, Antonio C. S. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PGMicro: Prof. Ricardo Reis
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 5
LISTA DE FIGURAS 6
LISTA DE TABELAS 8
RESUMO 9
ABSTRACT 10
1 INTRODUÇÃO 11
1.1 Soluções Existentes 12
1.2 Motivação 13
1.3 Contribuições deste trabalho 16
1.4 Organização deste trabalho 17
2 SISTEMAS DE TRADUÇÃO BINÁRIA 19
2.1 Conceitos Básicos 20
2.1.1 Modos de execução 20
2.1.2 Software/hardware 21
2.1.3 Dinâmico/estático 22
2.1.4 Otimização de código 23
2.2 Trabalhos Correlatos 25
2.2.1 Rosetta 25
2.2.2 FX!32 26
2.2.3 Dynamo 28
2.2.4 Transmeta Crusoé 29
2.2.5 Daisy 31
2.2.6 Godson-3 GS464 33
2.3 Considerações finais 36
3 ARQUITETURAS RECONFIGURÁVEIS 38
3.1 Conceitos Básicos 39
3.1.1 Granularidade 39
3.1.2 Acoplamento 41
3.1.3 Modos de Reconfiguração 41
3.2 Trabalhos Correlatos 42
4 TRADUÇÃO BINÁRIA DE DOIS NÍVEIS 44
4.1 Funcionamento do Sistema Proposto 45
4.2 Tradução de Código x86 para MIPS (Primeiro Nível) 46
4.2.1 Unidade de Tradução 47
4.2.2 Unidade de Montagem 49
4.2.3 Unidade de PC 49

4.2.4 Unidade de Controle 49
4.3 Processador MIPS (Estendido) 49
4.3.1 Processador MIPS 49
4.3.2 Processador x86 50
4.3.3 Extensão do conjunto de instruções 52
4.4 DIM (Dynamic Instruction Merging) 54
4.4.1 Organização da Unidade Reconfigurável 54
4.4.2 Tradutor Binário (Segundo Nível) 56
4.4.3 Cache de Tradução (TCache) 56
5 METODOLOGIA E RESULTADOS 58
5.1.1 Simuladores (Software) 58
5.1.2 Ferramentas e Configuração de Simulação 60
5.2 Resultados 60
5.2.1 Desempenho 62
5.2.2 Potência e Energia 64
5.2.3 Área 66
6 ANÁLISE CRÍTICA 67
6.1 Custos de implementação de todo o conjunto de instruções x86 67
6.2 Suporte para outros conjuntos de instruções 67
6.3 Estendendo o segundo nível de tradução 68
6.4 Desempenho de outros sistemas de tradução binária 69
7 CONCLUSÕES E TRABALHOS FUTUROS 71
7.1 Trabalhos Futuros 72
7.1.1 Processador Multi ISA 72
7.1.2 Estudo do sistema utilizando multi-processadores 73
REFERÊNCIAS 74

LISTA DE ABREVIATURAS E SIGLAS

ASIP	Application Specific Instruction Set Processor
DSP	Digital Signal Processor
MP3	Moving Picture Experts Group
GPS	Global Positioning System
JPEG	Joint Pictures Expert Group
UFR	Unidade Funcional Reconfigurável
PPG	Processador de Propósito Geral
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
VHDL	VHSIC hardware description language
VLIW	Very Long Instruction Word
TB	Tradutor Binário
LDH	Linguagem de Descrição de Hardware
ISA	Instruction Set Architecture
NOC	Network On-Chip
SOC	System On-Chip
ULA	Unidade Lógica Aritmética
ILP	Instruction Level Parallelism
TLP	Thread Level Parallelism
PC	Program Counter
ROM	Read Only Memory
SRAM	Static Random Access Memory

LISTA DE FIGURAS

Figura 1.1: Declínio da lei de Moore em termos de desempenho (KIM, AUSTIN, *et al.*, 2003). 12
Figura 1.2: Gráfico de desempenho em relação às diferentes arquiteturas (BECK, 2010). 14
Figura 1.3: Diferentes cenários para a execução de código compilado. 15
Figura 1.4: Sistema de processamento com dois níveis de tradução binária. 16
Figura 1.5: Visão geral do trabalho proposto. 17
Figura 2.1: Camadas onde o tradutor binário pode ser implementado. 21
Figura 2.2: Camada ocupada pelo tradutor binário em software. 26
Figura 2.3: Tradução binária de x86 para Alpha. 27
Figura 2.4: Diagrama de funcionamento do FX!32. 28
Figura 2.5: Demonstrativo do funcionamento do sistema Dynamo (BALA, 1999). 29
Figura 2.6: Camada ocupada pelo sistema de tradução binária CMS. 30
Figura 2.7: Comparativo de temperatura entre um Crusoé (b) e um Pentium III Coppermine (a) (KLAIBER, 2000). 31
Figura 2.8: Camadas da arquitetura DAISY. 32
Figura 2.9: Tradutor binário situado na camada de interface. 34
Figura 2.10: Resultado do processo de tradução binária de instruções x86 para MIPS. 35
Figura 2.11: Desempenho do sistema ao executar diferentes aplicativos. 36
Figura 3.1: Arquitetura reconfigurável incluída em uma plataforma de processamento. 38
Figura 3.2: ARs de granularidade fina (a) e granularidade grossa (b e c). 40
Figura 3.3: Princípios da reconfiguração estática (a) e dinâmica (b). 42
Figura 4.1: Mecanismo proposto para este trabalho. 44
Figura 4.2: Visão geral do trabalho proposto. 45
Figura 4.3: Unidades que compõem o mecanismo de tradução binária. 47
Figura 4.4: Geração dos bits de configuração correspondente a instrução x86 da entrada. 48
Figura 4.5: Bits de configuração chegando na unidade de interpretação. 48
Figura 4.6: Os três formatos de instruções MIPS. 50
Figura 4.7: Possibilidades de composição de uma instrução x86. 51
Figura 4.8: Comparativo entre o número de vezes em que cada *flag* é calculada e o número de vezes em que são utilizadas. 53
Figura 4.9: Exemplo de uma configuração da Unidade Reconfigurável a partir de uma sequência definida de instruções (BECK, RUTZIG, *et al.*, 2008). 55
Figura 4.10: Estágios do Tradutor Binário anexados ao *pipeline* do processador 56
Figura 5.1: Tradução binária realizada pelo simulador. 59
Figura 5.2: Total de espaço utilizado para alocar os algoritmos na memória, compilados pelo GCC com “-static”. 61

Figura 5.3: Comparativo entre o número médio de instruções MIPS geradas a cada instrução x86. 62
Figura 5.4: Gráfico de desempenho para quatro diferentes configurações. 63
Figura 5.5: Comparativo do sistema proposto com um processador nativo da arquitetura x86 com IPC = 1. 64
Figura 5.6: Gráfico demonstrativo do consumo de potência relativo em cada componente do sistema. 65
Figura 5.7: Consumo de energia em três sistemas diferentes. 66
Figura 7.1: Processador capaz de executar diversos conjuntos de instruções. 72
Figura 7.2: Utilização de diversos núcleos em um sistema de processamento. 73

LISTA DE TABELAS

Tabela 1: Características da implementação de um TB. 22
Tabela 2: Sistemas reconfiguráveis e suas características (BECK, 2010). 43
Tabela 3: Um exemplo de tradução de instruções x86 para MIPS................................. 52
Tabela 4: Operações realizada por uma instrução complexa. 53
Tabela 5: Número de instruções executadas nos dois conjuntos de instruções............. 61
Tabela 6: Configurações utilizadas para as simulações de desempenho........................ 62
Tabela 7: Custo de área de cada componente do sistema.. 66

RESUMO

Atualmente, a adição de uma nova função implementada em hardware em um processador não deve impor nenhuma mudança no conjunto de instruções (ISA – *Instruction Set Architecture*) suportado para atingir melhorias em seu desempenho. O objetivo é manter a compatibilidade retroativa e futura de programas já compilados. Todavia, este fato se torna, muitas vezes, um fator impeditivo para o aprimoramento ou desenvolvimento de uma nova arquitetura. Desta maneira, a utilização de mecanismos de Tradução Binária abre novas oportunidades aos projetistas, já que estes mecanismos permitem a execução de programas já compilados em arquiteturas que suportam conjuntos de instruções diferentes do previsto inicialmente. Assim, para eliminar o custo adicional apresentado por estes sistemas de tradução, será proposto um novo mecanismo de tradução binária dinâmico de dois níveis. Enquanto o primeiro nível é responsável pela tradução *de facto* das instruções do conjunto nativo para instruções de uma linguagem de máquina intermediária, o segundo nível otimiza estas instruções já traduzidas para serem executadas na arquitetura alvo. O sistema é totalmente flexível, pois pode suportar a tradução de conjuntos de instruções completamente diferentes; assim como a utilização de arquiteturas de hardware com as mais diversas características. Este trabalho apresenta o primeiro esforço nesta direção: um estudo de caso onde ocorre a tradução de código x86 para MIPS (linguagem intermediária), que será otimizado para ser executado em uma arquitetura que realiza reconfiguração dinâmica. Resta demonstrado que é possível manter a compatibilidade binária, com melhoria no desempenho em torno de 45% em média e consumo de energia semelhante ao da execução nativa.

Palavras-Chave: tradução binária, sistemas embarcados, arquiteturas reconfiguráveis.

Tow-Level Binary Translation System for Multiple-ISA Execution

ABSTRACT

In these days, every new added hardware feature must not change the underlying instruction set architecture (ISA), in order to avoid adaptation or recompilation of existing code. Therefore, Binary Translation (BT) opens new possibilities for designers, previously tied to a specific ISA and all its legacy hardware issues, since it allows the execution of already compiled applications on different architectures. To overcome the BT inherent performance penalty, we propose a new mechanism based on a dynamic two-level binary translation system. While the first level is responsible for the BT *de facto* to an intermediate machine language, the second level optimizes the already translated instructions to be executed on the target architecture. The system is totally flexible, supporting the porting of radically different ISAs and the employment of different target architectures. This work presents the first effort towards this direction: it translates code implemented in the x86 ISA to MIPS assembly (the intermediate language), which will be optimized by the target architecture: a dynamically reconfigurable architecture. In this work is showed that is possible to maintain binary compatibility with performance improvements on average 45% and similar energy consumption when compared to native execution.

Keywords: binary translation, embedded systems, reconfigurable architectures.

1 INTRODUÇÃO

Nos últimos anos, houve um grande salto de crescimento no mercado de sistemas embarcados (VASSILIADIS, 2006) enquanto o consumidor final tem exigido cada vez mais poder de processamento. A tendência é a unificação de diversas funcionalidades em um mesmo dispositivo, com a mesma ou superior qualidade dos dispositivos anteriormente individualizados. Esta união de diferentes funcionalidades podem ser visualizadas em aparelhos como os *smartphones*, onde em apenas um celular foram integrados funcionalidades de áudio e vídeo, jogos, editor de texto, navegação na internet, GPS (*Global Positioning System*) e etc.

Com a Lei de Moore entrando em declínio e os processadores chegando ao seu limite de dissipação térmica (FLYNN e HUNG, 2005) (SIMA, 2004) houve uma busca maior de alternativas na área de tecnologia de sistemas computacionais. Esta realidade revela a grande necessidade das empresas na procura de novas soluções que atendam os requisitos orçamentários mínimos (alto poder de processamento e baixo consumo de energia) estipulados pelo mercado de embarcados. A Figura 1.1 demonstra que ao passar dos anos, os processadores não tem conseguido manter a linearidade de desempenho, prevista pela lei de Moore.

Um ponto muito significativo para o desenvolvimento de novas arquiteturas computacionais é a questão da compatibilidade binária entre os códigos de aplicativos já existentes e os que irão surgir. Para os desenvolvedores de hardware, existe uma grande dificuldade de lançar no mercado novos modelos de processadores, pelo fato de que existe a necessidade do reaproveitamento de ferramentas e códigos já desenvolvidos e compilados.

Outra questão importante é que no ponto de vista de uma empresa desenvolvedora de software se torna muito custoso o treinamento de funcionários para codificação em novos conjuntos de instruções. Seria muito mais simples se a informação a respeito do hardware que compõe o sistema pudesse ser abstraída, assumindo-se, desta forma, que o mesmo aplicativo poderia ser executado em componentes de hardware diferentes. Já para os usuários, os aplicativos desenvolvidos apenas em uma linguagem perdem em flexibilidade, sendo reduzida a possibilidade de utilizar o mesmo código compilado em diferentes processadores, tornando-os assim, totalmente dependentes da arquitetura utilizada.

Existem famílias de processadores muito bem difundidas e solidificadas no mercado, juntamente com uma série de aplicativos já desenvolvidos e compilados com seu conjunto de instruções proprietários. Assim, pode-se concluir que disponibilidade de

serviços para o usuário está diretamente ligada ao modelo de processador escolhida. Podem ser citados como exemplos os processadores ARM (ARM, 2011) no mercado de sistemas embarcados e os processadores x86 (INTEL, 2011), que estão no mercado há mais de 30 anos e são utilizados principalmente na computação de propósito geral.

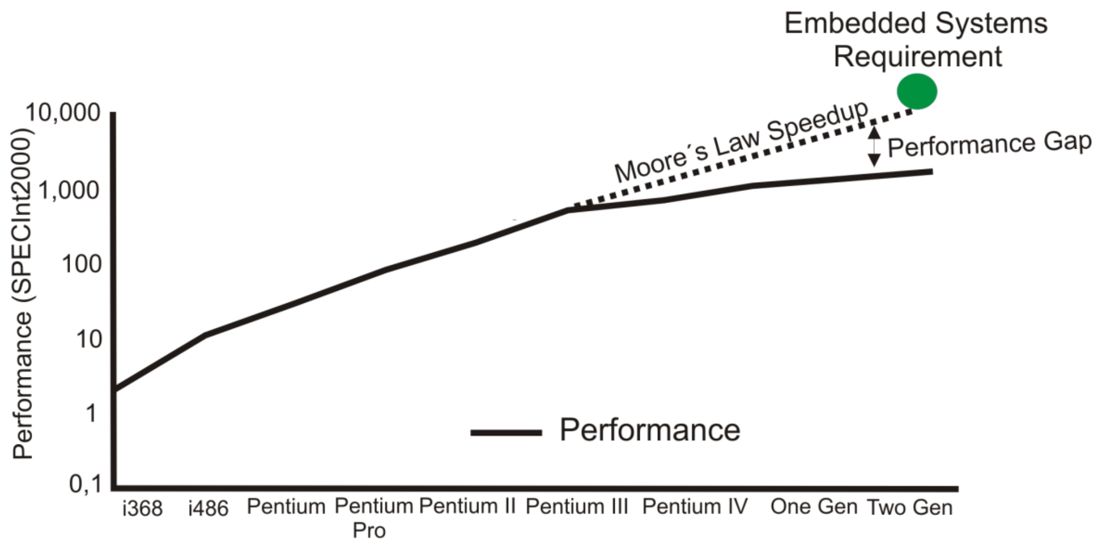


Figura 1.1: Declínio da lei de Moore em termos de desempenho (KIM, AUSTIN, *et al.*, 2003).

1.1 Soluções Existentes

Algumas técnicas (tanto em hardware como software) foram propostas como solução para este problema. Dentre elas está a interpretação por software. Esta solução permite a execução de código previamente compilado em diferentes conjuntos de instruções em apenas um único processador. Cada código executado pelo aplicativo é analisado em tempo real pelo interpretador, e em seguida, realizada uma conversão entre os conjuntos de instruções nativa do aplicativo e o novo conjunto do processador.

A utilização desta técnica traz como principal benefício a execução de um mesmo código binário em diferentes arquiteturas. Porém, existem significativas desvantagens, podendo ser citado o sistema de interpretação por software que necessita de uma camada extra de software entre o aplicativo e o processador e, sendo assim, a sua execução em muitos casos é mais lenta quando comparado com a execução do código nativo. Também pode ser mencionado a necessidade de haver um interpretador implementado no mesmo conjunto de instruções do processador alvo.

Outra solução proposta é o emulador por microcódigo (HENNESSY e PATTERSON, 2007) que realiza um processo semelhante ao interpretador de software. Todavia, este é implementado em hardware, dispensando um custo alto de processamento. Esta arquitetura é responsável pela decodificação do código do aplicativo para o novo conjunto de instruções. O mecanismo interno de funcionamento é basicamente formado por uma memória ROM (*Read Only Memory*) de alta velocidade para armazenamento do microcódigo e uma máquina de estados para realizar o controle interno do sistema. Entretanto, este sistema não é flexível pois é totalmente dependente do processador para o qual o sistema foi programado.

Geralmente a escrita do microcódigo é feita pelos engenheiros de projeto da arquitetura. Existem alguns emuladores que permitem a adição de microcódigo através

de uma memória flash ou SRAM (*Static Random Access Memory*), porém normalmente este nível é transparente para o programador comum. Uma utilidade prática desta técnica é permitir que instruções CISC (*Complex Instruction Set Computer*) sejam executadas em processadores mais simples do tipo RISC (*Reduced Instruction Set Computer*), apenas adicionando uma camada extra de hardware e mantendo assim a compatibilidade binária. Esta solução já é utilizada há muito tempo nos processadores da Intel que transformam instruções CISC x86 (complexas) em instruções RISC para serem executadas internamente (HINTON, SAGER, *et al.*, 2001).

Existe também uma terceira solução, conhecida como tradução binária. A tradução binária possui o mesmo objetivo das técnicas anteriores, ou seja, manter a portabilidade de aplicativos em diferentes conjuntos de instruções com ou sem a intervenção do usuário final. A principal diferença entre um Tradutor Binário (TB) e as técnicas citadas anteriormente é que, além de guardar as traduções para uma reutilização futura, ele pode ser implementado de diversas maneiras: pode ser dinâmico ou estático; e pode ainda ser implementado como uma camada de software ou hardware. Este tópico é o foco principal deste trabalho e será discutido com mais riqueza de detalhes nos próximos capítulos.

1.2 Motivação

Conforme o comentado na seção anterior, usuários de sistemas embarcados, como telefones celulares do tipo *smartphone*, querem abstrair qual processador está sendo utilizado em um determinado produto ou ainda qual a frequência em que o processador está computando dados. Entretanto, maior importância é dada para questões como a quantidade e a disponibilidade de software para o seu produto, a forma satisfatória de execução dos aplicativos no equipamento, e ainda se estes mesmos aplicativos poderão ser executados em novas versões do produto.

Analisando as soluções existentes citadas na seção anterior, com a tradução binária é possível, em muitos casos, atingir ganhos de desempenho quando comparada com a interpretação de software e o emulador de microcódigo. Isto ocorre porque os dois modelos anteriores não se preocupam com as otimizações do código em tempo de execução para eliminar a queda no desempenho do sistema. No primeiro caso, o tempo de computação sempre é maior devido à concorrência no processamento entre a execução do interpretador e a execução do aplicativo.

A Figura 1.2 ilustra um gráfico de desempenho, demonstrando a superioridade dos programas traduzidos em relação aos códigos interpretados ou se utilizando a solução por microcódigo. Porém, a tradução possui um desempenho menor quando comparada a execução nativa com otimizações em tempo de compilação.

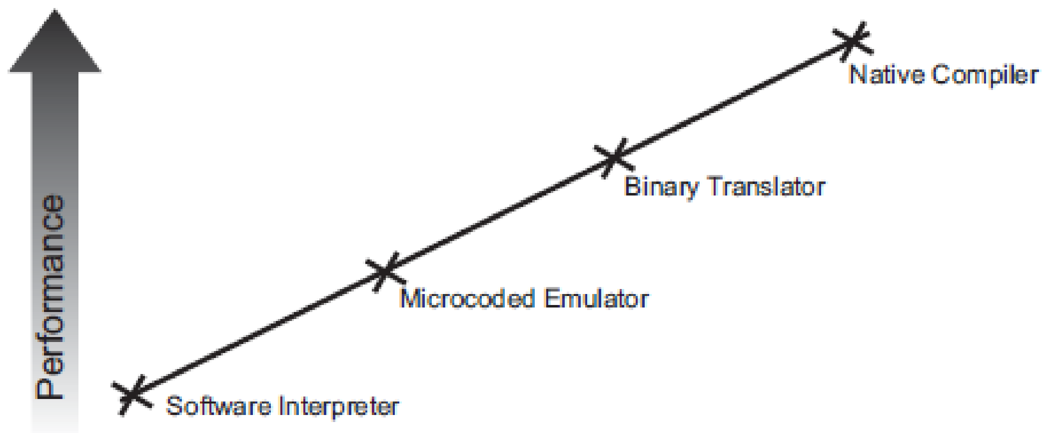
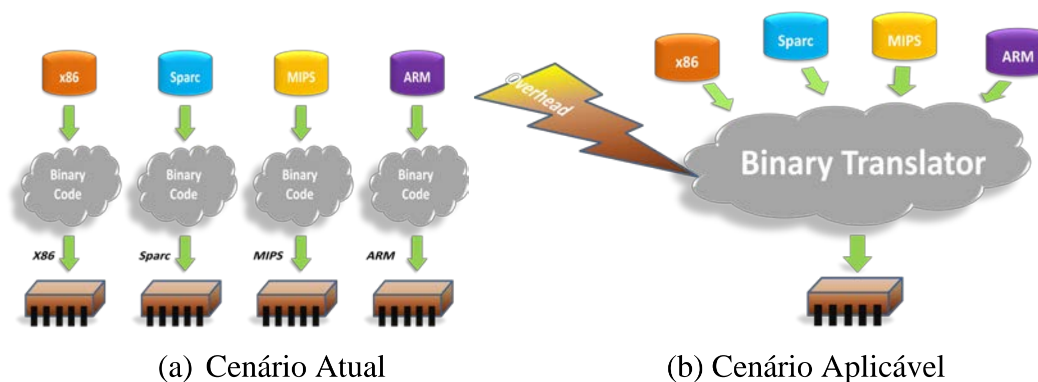


Figura 1.2: Gráfico de desempenho em relação às diferentes arquiteturas (BECK, 2010).

Para melhor visualização da problemática e baseando-se na solução da utilização de sistemas de tradução binária, existem hoje três diferentes cenários:

- **Cenário atual**: cada aplicativo compilado em um determinado conjunto de instruções deverá ser executado no seu correspondente processador. Neste cenário existe uma forte dependência entre o software e o hardware. Por exemplo, um código compilado no conjunto de instruções X86 somente poderá ser executado em um processador AMD, Intel, etc. Este cenário pode ser melhor ilustrado através da Figura 1.3 (a).
- **Cenário ideal**: de forma independente do conjunto de instruções em que o aplicativo foi compilado, esse código seria executado em um processador genérico de forma eficiente, ou seja, com desempenho igual ou melhor que o processador nativo. Por exemplo, dois códigos compilados para conjuntos de instruções diferentes (ARM e PowerPC) poderiam ser executados no mesmo processador de forma eficiente.
- **Cenário aplicável**: o aplicativo que foi desenvolvido e compilado visando a execução em um determinado processador, poderá ser executado em um processador genérico através da inserção de uma camada de tradução binária entre o caminho crítico do hardware e o software. No entanto, este cenário (exibido na Figura 1.3 (b)) impossibilita que o aplicativo seja executado com o mesmo desempenho quando se for comparado com a execução nativa.

Apesar da ideia de um tradutor binário ser simples, na prática, a complexidade deste sistema é muito alta, pois existem diversas variáveis (formato da instrução, manipulação de dados, modo de endereçamentos, *flags* de estado) a serem analisadas e que precisam ter seu comportamento emulado perfeitamente.



(a) Cenário Atual (b) Cenário Aplicável

Figura 1.3: Diferentes cenários para a execução de código compilado.

Para eliminar o impacto causado pelo tradutor binário e fazer com que o processador genérico idealizado tenha uma execução satisfatória, é necessário que este sistema (tradutor binário e processador) possibilite uma execução de código compartilhada com aceleradores em hardware. Estes aceleradores são sistemas especializados na execução de trechos específicos de código, aproveitando-se características como a exploração de paralelismo e/ou hardware otimizado. Entre os aceleradores podem ser citados: processadores VLIW (*Very Long Instruction Word*) que exploram paralelismo; processadores DSP (*Digital Signal Processor*) que, por exemplo, possuem o hardware otimizado para operações do tipo “multiplica e acumula”; e arquiteturas reconfiguráveis que permitem que o hardware se adapte de acordo com a necessidade do código.

Com a adição de um hardware acelerador, outro impasse é criado. Os aceleradores também possuem um conjunto de instruções proprietário, impossibilitando o seu uso na execução direta do código traduzido pelo primeiro nível de tradução binária. Neste caso, a ideia foi inserir um segundo nível de tradução binária, que realiza a tradução do código genérico para o código do acelerador, gerando assim um sistema de tradução binária de dois níveis. As possibilidades deste sistema podem ser melhores visualizadas na Figura 1.4, onde a primeira camada de tradução binária é responsável pela transformação do código de um determinado conjunto de instruções para um código pertencente a um conjunto de instruções genérico, enquanto a segunda camada otimiza este código para ser executado em *hardware* otimizado. Como há uma interface bem definida entre as duas camadas, o sistema é, ao mesmo tempo, flexível (o conjunto de instruções a ser traduzido pode ser modificado, assim como o *hardware* alvo, sem que hajam alterações na camada oposta) e eficiente, pois ambas as camadas são implementadas em *hardware*.

Devido as propriedades de um código compilado, onde a maior parte do tempo de computação de um aplicativo se concentra em uma pequena porção do código escrito, ou seja, na execução de laços (*for* e *while*) e rotinas, é possível melhorar o desempenho de um aplicativo se estas partes específicas do código forem executadas de forma mais eficiente. De tal modo, a queda no desempenho de execução causada pelos dois níveis de tradução poderá ser compensada com a execução rápida de partes chave do código. Esta possibilidade é a temática do trabalho e será fortemente discutida e avaliada nos próximos capítulos.

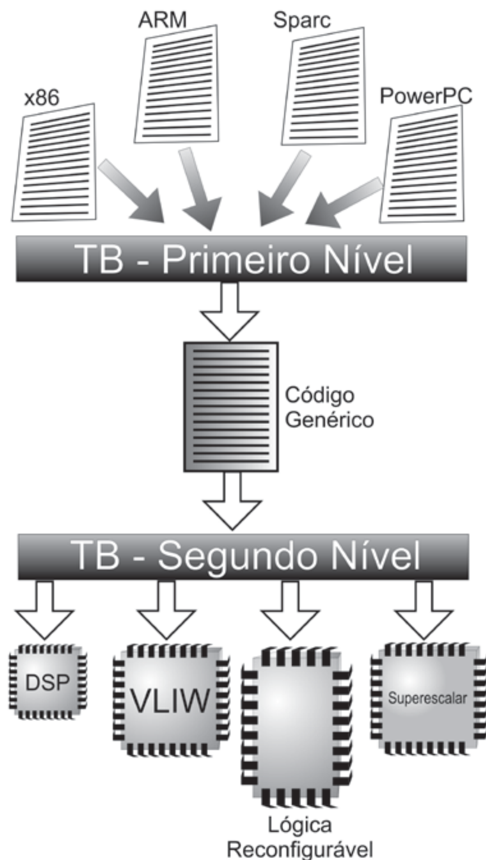


Figura 1.4: Sistema de processamento com dois níveis de tradução binária.

1.3 Contribuições deste trabalho

Este trabalho é baseado na ideia do uso de dois níveis de tradução binária para a execução satisfatória de código não nativo. A finalidade do trabalho é implementar um estudo de caso da proposta supracitada: um sistema completo que executa código compilado para o conjunto de instruções X86 em uma arquitetura reconfigurável de forma eficiente, onde este sistema é capaz de executar os dois conjuntos de instruções.

Para o primeiro nível de tradução binária, foi desenvolvido um novo mecanismo dinâmico implementado em hardware que funciona como interface entre o processador e a memória de instruções. Este primeiro nível tem a função de manter a compatibilidade binária entre dois diferentes conjuntos de instruções. Para o segundo nível de tradução binária foi utilizada uma arquitetura reconfigurável (revisão bibliográfica no Capítulo 2) já existente chamada DIM (*Dynamic Instruction Merge*) funcionando assim como acelerador de execução de código e amortizando os custos do primeiro nível.

Como já mencionado, o trabalho visa o estudo de caso, onde o primeiro nível é responsável pela tradução do código nativo x86 para MIPS (neste caso, o código genérico da Figura 1.4), enquanto o segundo nível otimiza partes do código traduzido em tempo de execução para que futuramente sejam executadas na arquitetura reconfigurável. A Figura 1.5 exibe uma visão geral do sistema de tradução com dois níveis.

O tradutor binário de primeiro nível foi desenvolvido tanto em hardware como em software (simulador) para extrair resultados mais rápidos e seguros. Com o hardware foi possível extrair dados como área, potência e energia. Já com o simulador em software, foi possível analisar o comportamento do código compilado, número e tipo de instruções, avaliar as possibilidades de melhorias no hardware, tempo de computação entre outras informações que serão discutidas posteriormente.

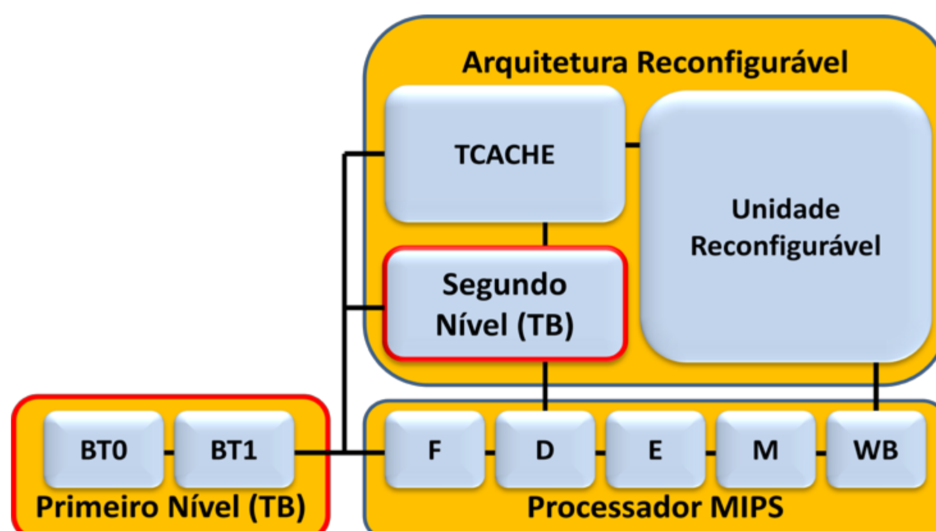


Figura 1.5: Visão geral do trabalho proposto.

As principais contribuições deste trabalho, em resumo, são:

- Implementação de uma ferramenta de simulação em linguagem C que realiza a tradução de código X86 para MIPS;
- Avaliação do comportamento do código compilado em termos de número e tipo de instruções, tempo de computação e uso de EFlags;
- Determinação das melhorias a serem feitas no hardware para otimização da execução;
- Utilização em série do simulador em software desenvolvido neste trabalho, com o simulador já desenvolvido da arquitetura reconfigurável;
- Desenvolvimento de um protótipo de hardware de tradução binária em VHDL;
- Extração dos custos de área, desempenho e energia utilizando as ferramentas da Synopsys;
- Validação do funcionamento do sistema.

1.4 Organização deste trabalho

Neste trabalho os capítulos estão organizados de forma a seguir uma sequência lógica e auto suficiente para o entendimento do sistema de tradução binária de dois níveis. Após esta introdução, onde foram discutidas as soluções existentes, foi descrita a motivação que levou a implementação do trabalho e na sequência suas contribuições. No Capítulo 2, através de uma revisão bibliográfica, serão vistos os sistemas de tradução binária mais utilizados na atualidade. Neste mesmo capítulo serão apresentados conceitos básicos a respeito de tradução binária, nomenclaturas, definições, formas de implementação e características gerais. É necessária também uma

breve explanação a respeito de cada um dos principais sistemas de tradução binária, suas funcionalidades e peculiaridades.

O capítulo 3 traz uma revisão bibliográfica a respeito de arquiteturas reconfiguráveis, o conceito de granularidade e acoplamento, além das formas de reconfiguração. Esta revisão é necessária para que o leitor se situe melhor na funcionalidade do segundo nível de tradução. Da mesma forma, o Capítulo 2 terá uma amostra sobre arquiteturas reconfiguráveis existentes, porém de forma mais sucinta, já que este tema não faz parte da discussão principal e nem foi fruto do desenvolvimento deste trabalho.

No capítulo 4 o tema deste trabalho será apresentado de forma mais aprofundada. Detalhes sobre o funcionamento de seu hardware, composição, diferenças arquiteturais e na semântica dos conjuntos de instruções x86 e MIPS, bem como as características funcionais do sistema como um todo serão exibidos.

O capítulo 5, apresenta a metodologia permitindo a repetibilidade dos experimentos e os resultados obtidos durante a fase de validação do sistema. Além disso, este capítulo irá apresentar os resultados e juntamente uma discussão a respeito dos dados apresentados.

Uma visão crítica realizada no capítulo 6, comentando a respeito dos resultados atingidos por outros trabalhos correlacionados. Neste capítulo também será discutido as possibilidades de se substituir o segundo nível de tradução binária por outro sistema equivalente.

Finalmente, no capítulo 7 serão apresentadas as devidas conclusões do trabalho, juntamente com os trabalhos futuros, mostrando as possibilidades para dar continuidade e expansão desta dissertação.

2 SISTEMAS DE TRADUÇÃO BINÁRIA

Nos últimos anos, a dependência software/hardware vem se tornando cada vez mais presente no mercado de processadores e, também, mais significativa, ao passo em que novas plataformas são desenvolvidas e tecnologias incompatíveis entre si são criadas. O mercado de processadores para sistemas embarcados é relativamente novo e existem diversas empresas tentando fixar o seu produto. Esta diversidade arquitetural torna cada vez mais comum o surgimento de aplicativos exclusivos (dependentes do conjunto de instruções). Empresas e pesquisadores há muito tempo visualizaram esta problemática. No decorrer de aproximadamente 10 anos, se estuda e se desenvolve sistemas de tradução binária como uma possível solução arquitetural para problemas de compatibilidade binária entre processadores.

Basicamente, o método de tradução binária consiste em converter um código já compilado para um determinado conjunto de instruções apontado pelo processador, no qual o código será executado de fato. Como resultado desta tradução, surge a possibilidade de executar um aplicativo que anteriormente era dependente de um hardware proprietário em outro sistema que utiliza um conjunto de instruções diferente. Para o usuário do sistema, esta tradução é abstraída e o aplicativo é executado da mesma forma como se o programa estivesse sendo utilizado no processador nativo. No entanto, poderá existir uma diferença em termos de tempo de computação e consumo de energia. Neste caso, o usuário poderia perceber, por exemplo, que seus aplicativos são executados de forma mais lenta ou ainda que o tempo de vida da sua bateria foi reduzido.

O aumento no tempo de computação causado pela execução de um tradutor binário não é desprezível. O sistema está sujeito a uma perda de desempenho em relação à execução de código em um processador nativo. Esta queda no desempenho deve-se a inserção de uma camada adicional entre o hardware e o software (i.e. tradutor binário). Esta limitação poderia tornar esta técnica impraticável, principalmente considerando o mercado de sistemas embarcados, onde qualquer custo adicional tem grande impacto ao usuário.

Além do mais, espera-se que a tradução binária, no futuro, possa trazer outros benefícios, como a possibilidade de que um código uma vez escrito possa ser executado em qualquer modelo de processador (*write once, run everywhere*) e ainda tirando proveito de toda a tecnologia que determinado hardware possa oferecer para a aceleração do software.

2.1 Conceitos Básicos

As arquiteturas de processadores apresentam muitas diferenças. Por exemplo, os processadores têm diferentes quantidades de registradores (propósito geral ou não), utilização de *flags* de estado, número de unidades funcionais, interrupções, tipos de instruções (CISC ou RISC) e até mesmo a forma como exploram o paralelismo (superescalar ou VLIW).

Entretanto, segundo (ALTMAN, KAELI e SHEFFER, 2000), partindo do princípio que todos processadores são baseados em uma máquina de Turing, apesar de suas diferenças, todos podem ser emulados de um para o outro. Porém, à medida que cresce a complexidade dos processadores, aumentam-se também as dificuldades de realizar o processo de tradução binária.

Devido à complexidade dos sistemas de tradução binária, frequentemente se emprega uma nomenclatura própria, assim, torna-se importante um breve resumo explicativo a este respeito. Algumas das principais são citadas abaixo:

- **Arquitetura nativa**: conjunto de instruções para o qual o código foi originalmente compilado;
- **Arquitetura alvo**: código o qual um tradutor binário gerou após analisar e traduzir as instruções de uma arquitetura nativa;
- **Cache de tradução (TCache)**: memória utilizada para alocar o resultado do processo de tradução binária de um trecho de código a fim de ser reaproveitado futuramente;
- **Máquina virtual de monitoramento**: conhecida também como VMM (*Virtual Machine Monitor*): realiza o controle da execução do tradutor binário e o momento correto em que um código otimizado deverá ser executado.

Nos próximos tópicos serão apresentados mais alguns conceitos que definem a semântica de um mecanismo de tradução binária, contemplando as camadas de execução de um TB e suas características. Além disso, outras estratégias possuem particularidades em relação aos benefícios e problemas que um sistema pode enfrentar. Estas características são totalmente dependentes da forma de implementação, seja ela software/hardware e dinâmico/estático ou ainda, a relação dos mecanismos de TB com os processos de otimização de código.

2.1.1 Modos de execução

Talvez uma das primeiras decisões de implementação ou característica a ser observada em um tradutor binário é o seu modo de execução. Tradutores binários podem ser executados em diferentes camadas dentro de uma arquitetura computacional. Alguns sistemas ocupam um espaço adicional no hardware, outros são executados em uma camada abaixo do sistema operacional e existem ainda arquiteturas que são executadas de forma similar a um aplicativo podendo, inclusive, ser visíveis ao usuário. Desta forma, o modo de execução refere-se à camada onde o TB será implementado e que será o foco de sua tradução. Basicamente existem três camadas de abstração (ilustradas na Figura 2.1) possíveis, onde as duas primeiras são camadas de software:

- **Camada de Aplicação**: a sua implementação é feita da mesma forma que um aplicativo comum. É dependente do sistema operacional e utiliza seus recursos (por exemplo, bibliotecas) no momento da sua execução.

Geralmente, neste caso, o tradutor binário é executado somente quando o aplicativo compilado na arquitetura nativa é chamado. Tanto o tradutor binário quanto o sistema operacional já estão compilados e preparados para a arquitetura alvo.

- **Camada de Interface**: nesta camada o tradutor binário está situado abaixo da camada do sistema operacional. Todo o software (SO e aplicativos), com exceção do TB, está compilado para uma determinada arquitetura nativa. Para o SO, o tradutor binário pode ser equiparado a uma máquina virtual, no qual somente após a tradução do código fornece acesso ao hardware de fato. Desta forma, o usuário não percebe diretamente a execução do tradutor, sendo este o primeiro a rodar na inicialização do sistema. Para tornar a sua execução menos custosa, o mecanismo de TB é desenvolvido e compilado para a arquitetura no qual o código nativo será executado de fato (arquitetura alvo).
- **Camada de Hardware**: neste caso, o tradutor passa a ter um controle interno dedicado implementado em hardware. Desta maneira, não é gasto poder de processamento da unidade central de processamento para realizar a tradução. Geralmente, neste caso, a tradução é feita com alto desempenho, onde diversos recursos que são necessários para a tradução são executados em paralelo. Entretanto, esta camada faz com que o sistema se torne pouco flexível (já que é difícil de modificar o hardware) e impactante em termos de área e potência consumida pelo processador.

A camada ocupada pelo tradutor está diretamente ligada com a forma no qual este TB irá interagir com o sistema. Por exemplo, caso o TB tenha incluído a funcionalidade de otimização de código (este tema será abortado na seção 2.1.4) e estiver na camada de aplicação, somente a execução do aplicativo será otimizada. Já nas camadas de interface e hardware, qualquer parte do software poderá ser beneficiada pelo processo de otimização que o TB provê, inclusive o sistema operacional. Entretanto, esta característica não revela qual é a camada ideal para execução, dependendo muito do objetivo para o qual este mecanismo será utilizado.

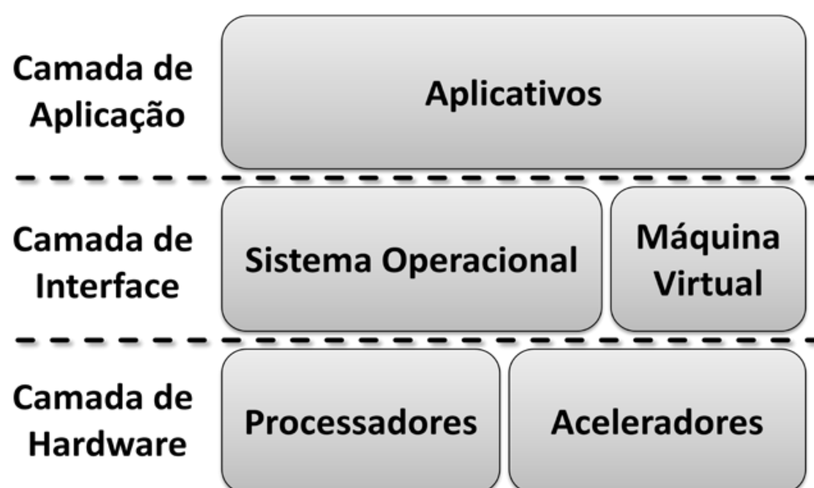


Figura 2.1: Camadas onde o tradutor binário pode ser implementado.

2.1.2 Software/hardware

A composição software/hardware de um sistema de tradução binária tem impacto significativo no funcionamento da arquitetura. Algumas características inerentes à

forma de implementação devem ser analisadas e consideradas como as vantagens e desvantagens devem ser consideradas em um tradutor binário. Na Tabela 1 podemos visualizar melhor as características de ambas.

Tabela 1: Características da implementação de um TB.

Características	Hardware	Software	Observações
Flexibilidade	Não	Sim	Utilizando um tradutor binário em software, é possível utilizá-lo em outras arquiteturas apenas recompilando o código, já em hardware qualquer modificação é impossibilitada.
Desempenho	Alto	Baixo	Em hardware a tendência do sistema é ser mais simples e seu controle é dedicado, já em software o próprio tradutor binário possui execução concorrente com o aplicativo.
Análise de Código	Baixa	Alta	As ferramentas disponíveis em hardware são limitadas. Em software as possibilidades de análise do código em termos de complexidade são muito maiores.
Recursos de Hardware	Alto	Baixo	O software tem pouco consumo de recursos do hardware, que resume-se em alocação na memória e processamento. Quando o TB é implementado em hardware, há os custos óbvios de área e potência.

Além da forma de implementação do tradutor, é possível avaliá-lo também conforme o momento em que será realizada a tradução do código. Esta análise de código pode ocorrer de forma dinâmica (em tempo de execução) ou estática (em tempo de computação).

2.1.3 Dinâmico/estático

A forma com que é feita a análise de código é definida por (ALTMAN, KAELI e SHEFFER, 2000) que classificou a arquitetura do TB em três diferentes tipos:

- **Tradutor Estático:** realiza a tradução off-line, ou seja, em tempo de compilação. Desta forma é possível que as otimizações do código sejam feitas de forma mais eficiente que o tradutor dinâmico. Este tipo de tradução também pode utilizar informações extraídas em tempo de execução do aplicativo, através de ferramentas auxiliares, para aumentar a eficiência da otimização.
- **Emulador:** realiza a interpretação do código em tempo de execução. Todavia, depois de terminada a análise por parte do tradutor binário, geralmente nenhuma tradução é salva em memória para uma futura reutilização. Geralmente sua estrutura é mais simples e baseada em acessos a tabelas que contêm as possíveis configurações da arquitetura para qual a instrução será traduzida.
- **Tradutor Dinâmico:** realiza a tradução e otimização de código em tempo de execução, com a possibilidade de guardar, em uma memória reservada, partes do código que poderão ser utilizadas novamente no futuro. Desta forma, é

> possível obter um ganho de desempenho otimizando trechos do código que são executados com maior frequência (laços e rotinas), amortizando o aumento no tempo de computação do sistema. Um exemplo muito conhecido é o Java JIT (*Just-in-Time*), que converte bytecodes JAVA (arquitetura nativa) para o conjunto de instruções da arquitetura alvo de forma dinâmica (YANG, MOON, *et al.*, 1999).

Utilizando a técnica de tradução binária estática, é necessário o uso de uma ferramenta padrão, para que o sistema torna-se visível ao usuário comum possibilitando a intervenção manual no seu processo. Esta intervenção manual pode ser proveitosa caso o usuário tenha um conhecimento mais aprofundado do hardware, já que poderão ser alterados parâmetros que podem ser melhor aproveitados segundo as características da arquitetura alvo e que tornarão o processo mais eficiente.

O emulador e o tradutor dinâmicos realizam análise somente durante a execução do programa e geralmente são transparentes tanto para o usuário final quanto para o programador, ou seja, o código pode ser executado em um processador “A” da mesma forma que seria executado no processador “B”. Esta questão é de extrema importância ao ponto que evita a necessidade do usuário final ou o programador comum de conhecer a arquitetura em que o código está sendo executado.

As três formas de implementação revelam a complexidade de um sistema de tradução e que qualquer decisão a respeito do formato ou comportamento do sistema poderá resultar em custos para o usuário, seja ele em termos de potência, queda no desempenho ou ainda transparência. Entretanto, atualmente existe uma forte tendência na utilização de TBs dinâmicos, pelo fato de que dispensam o usuário final ou o programador de um conhecimento mais profundo da arquitetura.

2.1.4 Otimização de código

O conceito de otimização de código está fortemente ligado com a técnica de Tradução Binária. Frequentemente, tradutores binários são executados em paralelo com aceleradores ou ainda possuem este mecanismo incorporado à sua implementação, otimizando o código da melhor forma possível, eliminando dependências, desenrolando laços (do tipo *for* ou *while*), montando instruções que irão tirar melhor proveito do hardware, dentre outras otimizações possíveis.

Apesar da técnica de tradução binária permitir a portabilidade de uma arquitetura à outra, é necessário também que a sua execução seja competitiva com os códigos gerados de forma nativa. Para tanto, a maioria dos sistemas de tradução também realizam análise do código em tempo real, visando otimização e redução do custo trazido pelo mecanismo de tradução. Segundo (CHERNOFF, HERDEG, *et al.*, 1998), arquiteturas de tradução binária podem se comportar como um compilador tradicional de alto desempenho. Todavia, ao contrário dos tradutores binários que têm como entrada o código objeto, os compiladores comuns iniciam a sua análise através do código de alto nível (por exemplo, na linguagem C). Desta forma, são capazes de interpretar melhor o comportamento do programa antes de realizar a otimização e a compilação de fato.

A falta de uma linguagem de alto nível (código fonte C/C++, Java, Pascal) para realizar uma análise mais profunda do seu real funcionamento, faz com que um dos maiores desafios para os sistemas de tradução seja a geração de um código correto e eficiente a partir das informações disponíveis em baixo nível. Algumas destas informações da linguagem de alto nível são perdidas no momento em que o código é

compilado. Desta maneira, muitas das otimizações de código feitas por um compilador ficam impossibilitadas.

Pode-se utilizar como exemplo um processador lançado há 2 anos e que possui a sua arquitetura baseada no conjunto de instruções x86. Um código desenvolvido, compilado e executado para este modelo de processador não terá nenhum tipo de problema de compatibilidade. Mas, se levarmos em conta que nesta família um novo processador é desenvolvido com recursos em hardware que permita também a execução de instruções multimídia (e.g. MMX, SIMD) (INTEL, 2011), o código anteriormente compilado não terá problemas de compatibilidade binária. No entanto, também não será beneficiado com a tecnologia disponibilizada pelo novo processador devido ao seu conjunto de instruções antigo.

Com o uso de um tradutor binário, surge a possibilidade de otimizar o código anteriormente escrito, atualizando partes do código que utilizavam um conjunto de instruções antigo e menos eficiente para as novas tecnologias desenvolvidas, fazendo com que um código anteriormente compilado para um modelo antigo de processador seja readequado para um novo conjunto de instruções, a fim de que tire proveito de todos os artifícios que um novo processador possa oferecer em termos de aceleração.

2.1.4.1 Profiling

Segundo (ALTMAN, EBCIOGLU, *et al.*, 2001), em diversas arquiteturas a tradução binária e a otimização de código estão incluídas no mesmo sistema. Porém, técnicas de análise de execução em tempo real como o Profiling, são realizadas separadamente do processo de tradução estática. Esta técnica realiza análise do programa em tempo de execução, extraindo informações que poderão ser utilizadas posteriormente como um guia de otimização. Durante o processamento de um aplicativo, o Profiling é executado em paralelo, verificando quais partes do código são as mais requisitadas (como o número de iterações que ocorre dentro de um laço) e instruções que podem ser executados em paralelo visando aceleração por ILP (*Instruction Level Parallelism*).

Após o Profiling gerar diversos arquivos com uma variedade de otimizações que podem ser feitas no código, o sistema de tradução analisa e associa estes arquivos em um único *profile* (ALTMAN, EBCIOGLU, *et al.*, 2001). Posteriormente o código pode ser novamente analisado, caso o sistema de Profiling altere o padrão de otimização do código. Geralmente estas informações são alocadas em um espaço reservado na memória.

2.1.4.2 Otimização dinâmica

O processo de otimização dinâmica funciona de forma concorrente à tradução binária e pode estar ligada também ao processo de profiling. Estas análises e mudanças no código em tempo de execução permitem que as melhorias para otimizar o processamento sejam sempre atualizadas caso exista mudanças no padrão de execução. Um ponto muito interessante é que dependendo da técnica utilizada, existe a possibilidade do programador comum não ter acesso ao processo de otimização. Muitas vezes, o programador não fica ciente da sua existência, devido ao fato de que o tradutor binário é executado de forma transparente.

Os compiladores realizam a tradução de código estaticamente. Já com tradutores dinâmicos surge a possibilidade de adquirir informações importantes a respeito do comportamento do programa durante sua execução como, por exemplo, com qual frequência que determinada parte do código é executada ou ainda em qual sequência

determinadas instruções podem ser executadas pra aumentar a eficiência de processamento. Estas informações permitem que se façam otimizações que são apenas possíveis por causa da análise dinâmica do programa sendo executado.

Por outro lado, existem alguns problemas enfrentados pela otimização dinâmica, como a necessidade de amortização do custo adicional de processamento. O acréscimo de mais uma camada entre o software e o hardware faz com que o conjunto perca em velocidade de execução para um compilador nativo, devido ao tempo gasto em tradução e otimização. Este atraso pode ser originado dos diversos estágios da tradução binária e até mesmo do tempo de acesso à memória.

Devido a grande geração de dados oriundos das análises realizadas pelo sistema de otimização, se faz necessária a alocação de um espaço dedicado para a arquitetura na memória principal ou a adição de uma memória física, conhecida como Cache de Tradução (TCache).

Dependendo do formato com que os dados são armazenados, pode existir um aumento no tempo de computação devido à alta quantidade de acessos à memória. Se um trecho de código otimizado possuir uma taxa alta de execução, logo serão realizados vários acessos à memória de tradução (TCache), assim o tempo de acesso a este dado na memória torna-se significante para o sistema. Uma das possíveis soluções propostas para minimizar esta perda é criar memórias menores e de rápido acesso permitindo que sequências de código mais executadas sejam alocadas neste espaço de endereçamento (BANAKAR, 2002).

Contudo, segundo (SRIDHAR, SHAPIRO e BUNGALE, 2007), a redução da sobrecarga causada pelo tradutor binário pode estar na simplificação da arquitetura, visando à utilização de algoritmos mais simples para otimização e tradução de código.

2.2 Trabalhos Correlatos

2.2.1 Rosetta

A empresa Apple, que se encontrava em um momento de transição de arquitetura de processador (substituindo PowerPCs por processadores Intel), solicitou à empresa Transitive Corporation o desenvolvimento de um software capaz de permitir que aplicativos, originalmente desenvolvidos para processadores PowerPC, fossem executados na nova família de computadores Macintosh baseados na arquitetura de processamento da Intel.

A Transitive Corporation então, desenvolveu o tradutor binário chamado Rosseta (APPLE, 2011). Apesar de portar ferramentas de otimização, Rosseta não consegue fazer com que o código traduzido seja executado com a mesma eficiência de um código nativo, devido à queda de desempenho causada pelo mecanismo de tradução. Além do mais, por problemas de compatibilidade, nem todos os aplicativos podem ser executados após a tradução. Esta compatibilidade depende do tipo de aplicação e as instruções que são utilizadas pelo programa candidato à tradução.

O Rosetta traduz instruções das versões G3, G4, e AltiVec do PowerPC. No entanto, não é compatível com instruções G5. Desta forma, aplicativos que utilizam instruções G5 devem ser modificados pelos seus desenvolvedores para serem compatíveis com os computadores Macs baseados em processadores Intel. Programas com baixa necessidade de processamento e alta interação com o usuário geralmente têm execução satisfatória após a tradução. Em contrapartida, programas com alta necessidade de

processamento, como a disponibilidade de dados em tempo real (aplicativos multimídia) e computação complexa de modelos 3-D, geralmente tornam a execução do aplicativo bastante lenta.

Durante a tradução binária, o código convertido e otimizado é alocado em uma memória de tradução (TCache), de forma transparente ao usuário final. Esta tradução, após sua devida alocação na memória, poderá ser reutilizada em execuções posteriores, permitido que o aplicativo seja acelerado devido a não necessidade de nova tradução e evitando perdas decorrentes do custo desta atividade. Desta forma, a etapa de tradução binária é desnecessária durante uma grande parte do tempo. Na Figura 2.2 é ilustrada a camada que se esta situado o sistema de tradução.

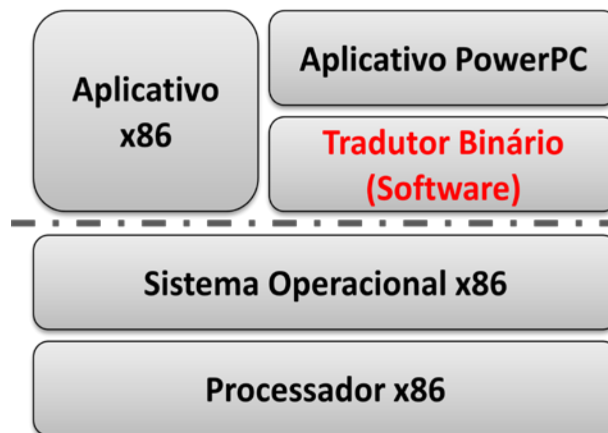


Figura 2.2: Camada ocupada pelo tradutor binário em software.

No instante em que um programa portador de um código binário não nativo é executado, o Sistema Operacional detecta esta incompatibilidade e executa o aplicativo por meio da interface Rosetta. Para o usuário que está utilizando uma máquina baseada na arquitetura Intel, é imperceptível visualmente a execução e a tradução binária dinâmica do Rosetta, devido à inexistência de qualquer tipo de interface gráfica ou ainda indicativo visual demonstrando que a aplicação está sendo traduzida. Entretanto, o usuário poderá perceber uma queda no desempenho durante a inicialização e execução do aplicativo.

2.2.2 FX!32

O sistema FX!32 (HOOKWAY, 1997) (CHERNOFF, HERDEG, *et al.*, 1998), tinha como premissa a portabilidade de código, possibilitando que qualquer programa originalmente implementado em código Intel x86, possa ser executado em microprocessadores Alpha. O Digital FX!32, é uma camada de software que combina emulador com o sistema de tradução binária do código nativo Intel x86, para o processador alvo Alpha. O emulador realiza conversão do código x86 para Alpha (que também fornece informações de *profiling)* e um sistema de tradução binária. A partir das informações do *profiling*, que é gerado em tempo de execução, o tradutor binário gera imagens otimizadas da aplicação por meio de intensivos algoritmos computacionais, com o objetivo de executar o código de forma mais eficiente. Segundo o autor, o FX!32 é o primeiro sistema a explorar esta combinação de emulação e tradução binária.

A camada de software é totalmente transparente ao usuário e permite que programas 32-bit sejam instalados e executados em uma arquitetura x86, executando o sistema operacional Windows NT 4.0, possam também ser instalados e executados em sistemas Alpha executando a mesma versão do SO (desenvolvida implementando o conjunto de instruções nativo do Alpha). As camadas deste sistema estão ilustradas na Figura 2.3.

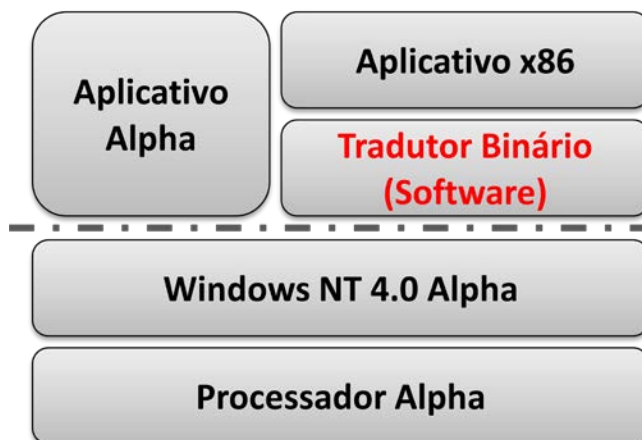


Figura 2.3: Tradução binária de x86 para Alpha.

Executando aplicativos compilados de forma nativa para a arquitetura Alpha, é dispensado o uso do emulador e do sistema de tradução binária. No momento em que o usuário comum deseja executar um aplicativo compilado para a arquitetura x86, o emulador inicia o processo de geração de código do Alpha a partir das instruções nativas x86 em tempo de execução juntamente com o processo de profiling. Devido ao fato do sistema estar sendo executado na camada de interface, a sua execução é concorrente com a execução do software ocasionando uma queda na velocidade de processamento. Para minimizar o problema de perda de desempenho, foi implementado um sistema de tradução binária em software que realiza otimização do código a partir das informações geradas pelo profiling após a sua execução, impedindo-se que esse processo consuma recursos de processamento juntamente com o aplicativo. Assim, o emulador irá executar o código x86 traduzido e otimizado somente quando for detectado uma pré tradução já existente.

As imagens e as informações de *profiling* são alocadas em um banco de dados para serem utilizadas no momento em que o usuário executar um aplicativo. O servidor coordena as otimizações, tanto em tempo de execução, como as realizadas após a execução, podendo atuar de forma padrão ou de acordo com os parâmetros especificados manualmente pelo usuário. Existe também a possibilidade do usuário controlar os recursos utilizados pelo FX!32 através de um sistema de gerenciamento.

O FX!32 é composto por diversos componentes em software conforme demonstrado na Figura 2.4 :

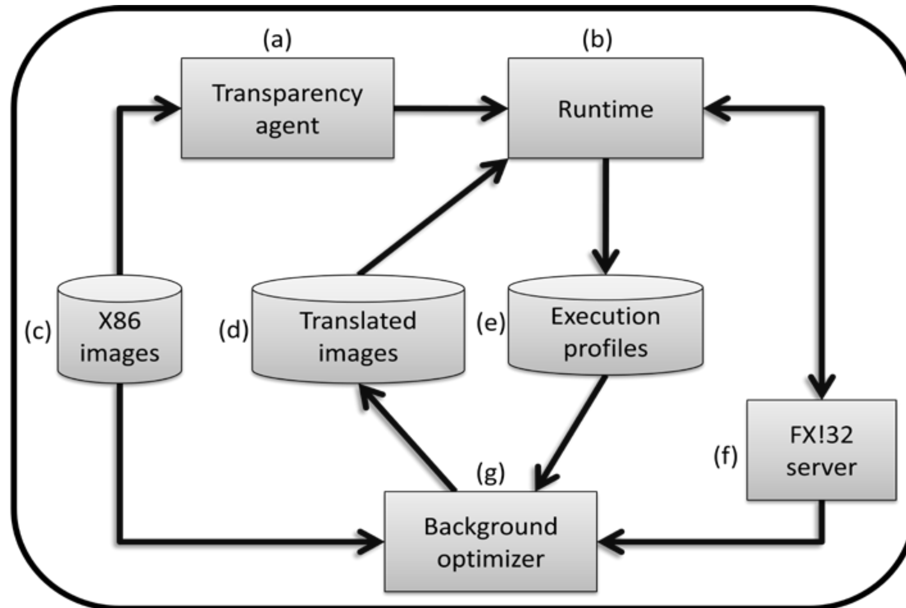


Figura 2.4: Diagrama de funcionamento do FX!32.

Transparency agent – agente (Figura 2.4a) que garante que o sistema seja invisível durante a sua execução, permitindo que os aplicativos x86 sejam executados sem a distinção de ser ou não código nativo, ou seja, de forma transparente e sem a necessidade de intervenção do usuário comum;

Runtime – carrega as imagens x86 e permite que sejam executadas em tempo real, além disso permite que o código x86 faça chamada de rotinas da API do Windows NT (Alpha) através de pequenos fragmentos de código chamados de "*Jackets*" (Figura 2.4b). Esta execução ocorre passando pelo emulador no qual disponibiliza análises da execução (Figura 2.4e) e faz uso do código já traduzido (Figura 2.4d) e otimizado (quando disponível);

Backgroud optimizer – deste bloco (Figura 2.4g) faz parte o sistema de tradução binária que realiza a otimização do código e gera imagens do código x86 (Figura 2.4d);

FX!32 Server – este servidor (Figura 2.4f) provê o serviço de alocar os dados de profiling fornecido pelo emulador e controla o momento em que o tradutor binário deve iniciar a sua execução.

Após a finalização de um determinado aplicativo x86, o servidor realiza uma comparação dos dados de um novo *profiling* com o *profiling* já existente, ambas geradas pelo emulador. Caso as informações estiverem diferentes, significa que o *profiling* atual possui informações incompletas ou desatualizadas, necessitando então, de atualização. Este processo é repetido todas as vezes que o aplicativo for executado.

2.2.3 Dynamo

O Dynamo (BALA, DUESTERWALD e BANERJIA, 2000) é um sistema baseado inteiramente em software, escrito em linguagem C e algumas partes em assembly. Este sistema foi criado por pesquisadores da Hewlett-Pachard (HP) para realizar otimização dinâmica de código. Originalmente o sistema foi implementado para PA-RISC, uma plataforma baseada no PA-8000 executando o sistema operacional HPUX. Esta arquitetura não realiza tradução binária, somente otimização em tempo de execução.

Segundo (BALA, DUESTERWALD e BANERJIA, 2000), pelo fato do Dynamo ser totalmente transparente para o usuário final, este sistema pode ser visto como um processador virtual, onde as otimizações dinâmicas do componente são implementadas na camada de software enquanto o processador é implementado em hardware. Suas otimizações são realizadas baseadas no escopo do código do programa à medida que ele é executado, ou seja, não há nenhum tipo de acesso aos arquivos executáveis.

Inicialmente, o Dynamo começa a analisar um determinado trecho de código e verifica se existe uma alta taxa de execução deste grupo de instruções. Depois de satisfeita esta condição, o sistema entra em modo de otimização de código.

As informações referentes aos trechos de instruções mais executados (*hot regions*) são armazenados em um buffer. O otimizador gera uma versão otimizada do conjunto, chamada de fragmento. Este fragmento é então alocado em uma memória reservada (*fragment cache memory*), onde estes fragmentos podem ser integrados dinamicamente através de procedimentos de chamadas. Na Figura 2.5 é apresentado um esquemático de funcionamento do sistema.

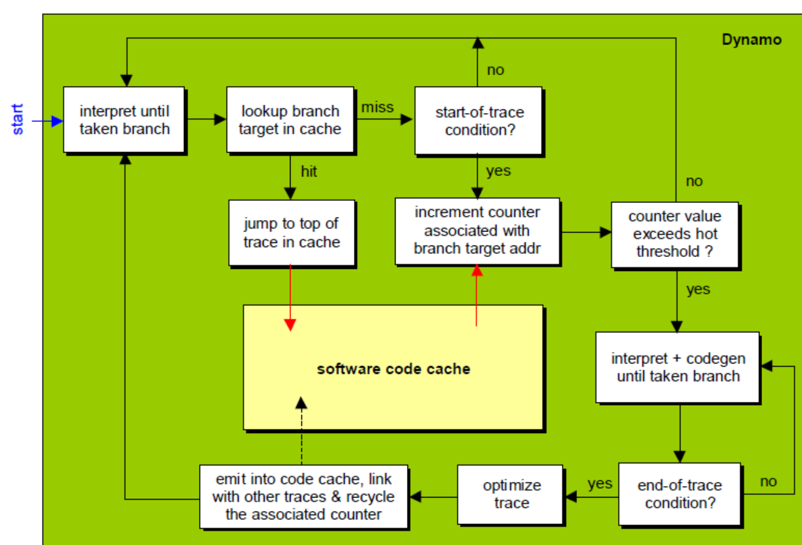


Figura 2.5: Demonstrativo do funcionamento do sistema Dynamo (BALA, 1999).

Dentre as técnicas de otimização de código estão incluídas a remoção de redundâncias no código, propagação de constantes, diminuição de instruções de leitura e *loop unrolling*, que é basicamente a reordenação das instruções contidas em um laço e para transformá-las em uma sequência contínua de código, a fim de diminuir o número de instruções de controle (BALA, 2000).

2.2.4 Transmeta Crusoé

O Crusoé fez parte de um projeto da empresa Transmeta, que visou o desenvolvimento de um microprocessador totalmente compatível com o conjunto de instruções x86, porém que apresentasse alguns fatores diferenciais dos processadores já existentes, em termos de poder de processamento e baixo consumo de energia. Internamente a sua arquitetura é compromissada a um processador do tipo VLIW (*Very Large Instruction Word*) nativo, o qual é dependente de uma camada de software chamada *Code Morphing Software* (CMS).

Cada palavra (nesta arquitetura é chamada de molécula) pode armazenar duas ou quatro instruções do tipo RISC, chamados de átomos. Estes átomos podem ser

executados por cinco unidades funcionais em paralelo: duas ULAs (unidade lógica aritmética), uma unidade de ponto flutuante, uma unidade de memória e uma unidade para tratamento de salto. Além disso, 64 registradores de propósito geral e 32 registradores de ponto flutuante.

O uso deste modelo de processador torna o sistema passível de diversos tipos de otimizações, principalmente em termos de ILP. Desta maneira, aumentam-se as chances de ocupação de todas as unidades funcionais disponíveis do processador, executando o maior número de instruções em paralelo. A família de processadores TM5000 está fortemente ligada à família TM3000. A diferença da nova geração é que a mesma obteve mudanças pela adição de átomos para melhorar a execução das instruções x86

Em software, o sistema Crusoé possui uma camada situada logo abaixo do SO, realizando a interface entre o hardware e o aplicativo CMS, como demonstrado na Figura 2.6. O CMS é a implementação de um sistema de tradução binária em software, que de forma dinâmica realiza a análise de código, otimização e tradução binária.

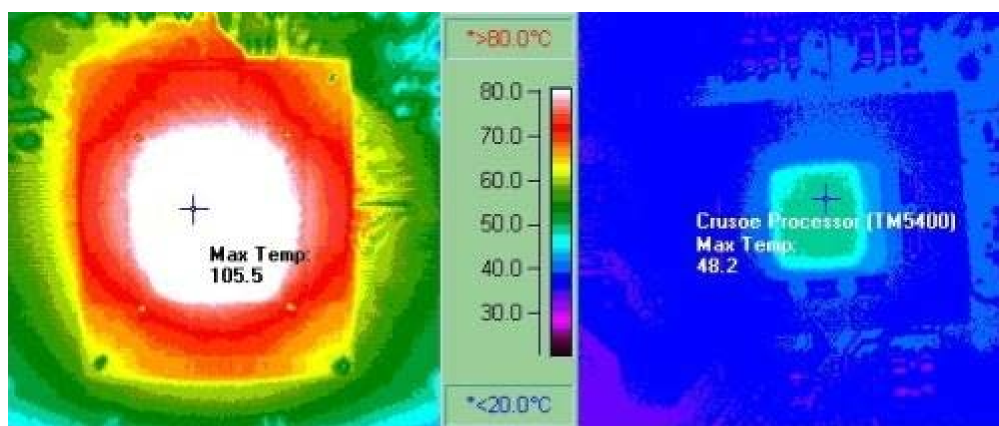
Figura 2.6: Camada ocupada pelo sistema de tradução binária CMS.

Segundo (DEHNERT, GRANT, *et al.*, 2003), os desafios naturais do CMS são:

- O tradutor binário deve implementar completamente a arquitetura x86, incluindo: mapeamento de memória das entradas e saídas, implementação de todo o conjunto de instruções, manutenção do comportamento das interrupções e suporte aos registradores da arquitetura.
- Devido ao CMS ser uma camada de sistema e não de aplicação, o software não pode interferir nas tomadas de decisões do sistema operacional, ou seja, deve ser transparente no ponto de vista de tradução e até mesmo executar instruções da BIOS, tornando-o não dependente de nenhum tipo de informação ou assistência das camadas superiores. Desta forma, o CMS não tem acesso a nenhum tipo de arquivo ou código estático, todas as instruções chegam em tempo de execução para a tradução.
- Com base na grande quantidade de diferentes tipos de aplicativos a serem executados como: jogos, processamento de áudio e vídeo, servidores e sistemas operacionais, o CMS deve prover ao sistema um bom desempenho de tradução e otimização para que seja amortizado o custo desta camada, e ainda tenha um ganho de desempenho na execução das aplicações melhorando o processamento.

Quando o computador é ligado, o Code Morphing é o primeiro programa a ser inicializado. O código do CMS é escrito no formato das instruções do Crusoé. Seu funcionamento ocorre de forma semelhante a outros tradutores: inicialmente um interpretador decodifica e executa as instruções x86 sequencialmente. Quando o número de instruções ultrapassarem uma quantidade mínima, este conjunto é otimizado e guardado com uma indexação através do PC em uma TCache em formato de instruções VLIW. Posteriormente as instruções serão executadas pelo processador quando o mesmo PC for encontrado.

A Figura 2.7a ilustra o mesmo aplicativo sendo executado no Pentium III e ao lado na Figura 2.7b a execução ocorrendo no processador Crusoé. A execução em um processador VLIW permite a combinação da aceleração por ILP juntamente com a diminuição da complexidade do hardware. Como estes fatores são significativos na redução de consumo de potência com alto poder de processamento, pode-se observar que reduzindo a frequência de operação do processador e mantendo o mesmo desempenho, os custos no consumo de potência são reduzidos de forma significante. Tornando-se a otimização do software e hardware necessários e fundamentais para que um sistema funcione de forma eficiente.



(a) Pentium III (b) Crusoé

Figura 2.7: Comparativo de temperatura entre um Crusoé (b) e um Pentium III Coppermine (a) (KLAIBER, 2000).

2.2.5 Daisy

O sistema DAISY (*Dynamically Architected Instruction Set from Yorktown*) (EBCIOGLU e ALTMAN, 1997) é muito semelhante ao Crusoé, no entanto possui propósitos diferentes quando se refere ao conjunto de instruções e algumas outras particularidades. O objetivo deste projeto é executar o conjunto de instruções do PowerPC em um processador VLIW, possibilitando um ganho de desempenho de processamento, juntamente com uma queda na frequência de operação. Assim, é possível reduzir os custos de energia, quando comparado ao processador original.

Este sistema é composto por três grandes camadas: camada de aplicação, onde está situado o sistema operacional e aplicativos; camada da máquina virtual (DAISY VMM), que realiza a interface entre o hardware e software aplicativo; e camada de hardware, formado pelo processador DAISY VLIW (as camadas estão ilustradas na Figura 2.8). Existem também diferentes tipos de memórias: a TCache que aloca os dados vindos da tradução e otimização de código; a memória do PowerPC, que aloca as instruções no

formato da arquitetura original; e memória de boot que guarda as instruções referentes à inicialização da máquina virtual (DAISY VMM).

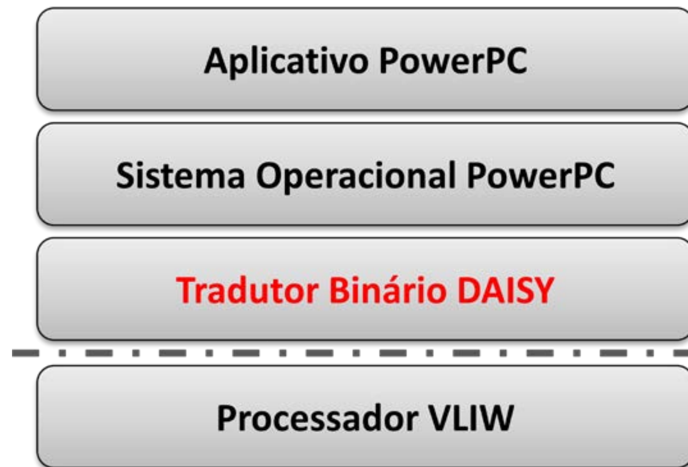


Figura 2.8: Camadas da arquitetura DAISY.

A microarquitetura do processador DAISY VLIW é composta por 4 conjuntos, conhecidos como clusters, onde cada cluster possui 4 unidades funcionais e uma ou duas unidades de *load/store*. Para a alocação dos dados resultantes das operações, o processador DAISY VLIW possui 64 registradores de números inteiros, 64 registradores de ponto flutuante e 16 registradores de condição. Este processador também possui suporte a especulação durante a execução.

O processo de tradução binária ocorre de forma totalmente transparente, ou seja, para o usuário final, o processador DAISY VLIW se comporta como se fosse um processador PowerPC. Esta tradução binária é realizada pela camada de software DAISY VMM. Esta camada de software é inicializada juntamente com o boot do sistema. De forma similar ao PowerPC nativo, o endereço da ROM de boot é o mesmo (0xfff00100). O código de boot será interpretado, traduzido e executado sob o controle da máquina virtual.

Quando o DAISY VMM encontra o primeiro fragmento de código nativo PowerPC, ele realiza a interpretação. Durante esta interpretação é realizada uma análise dos dados coletados, para que futuramente sejam utilizados na otimização deste trecho. Estas primitivas são alocadas em uma área de memória que é invisível pelo usuário.

A sequência traduzida é utilizada em chamadas do tradutor binário, para evitar que um código já presente na TCache seja novamente recompilado. Desta forma, o processador DAISY VLIW executa imediatamente a sequência de código otimizada, amortizando o tempo adicional da compilação dinâmica e elevando o grau de paralelismo na execução no nível das instruções (ILP). Cada grupo de código traduzido é vinculado ao endereço PC. Assim, quando um PC já existente for localizado na TCache, entende-se que não existe necessidade de uma nova análise.

A aceleração por ILP e a paralelização na execução do código são os principais focos da arquitetura DAISY, o que permite alcançar um alto desempenho através da adaptabilidade dinâmica de código. Esta otimizações permitem o DAISY a atingir a execução de quase 2,5 instruções por ciclo, mesmo com o custo adicional de sistema.

2.2.6 Godson-3 GS464

O processador Godson-3 (FAN, YANG, *et al.*, 2009) (HU, WANG, *et al.*, 2009) é a terceira geração de processadores Godson. O primeiro *multi-core* da família surgiu no final do ano de 2008, na China. A grande aposta desta arquitetura está na possibilidade da execução de código x86 em processadores MIPS, através de um sistema de tradução binária em software. No entanto, esta tradução de instruções x86 para MIPS não é realizada de forma simples. Diferenças importantes tanto na arquitetura como na organização dos processadores que executam estes conjuntos de instruções precisam ser consideradas, como por exemplo, flags de estado, modos de endereçamento, movimentação de dados, etc. Devido a estas diferenças, muitas vezes a tradução de uma instrução x86 requer dezenas de instruções MIPS. Para reduzir estes custos de tradução, foram desenvolvidas soluções tanto em software como em hardware, visando tornar este processo o mais eficiente possível. Na Figura 2.10 é possível visualizar exemplos de tradução do código x86 (Figura 2.10a) para MIPS (Figura 2.10b) onde se pode visualizar que cada instrução x86 é transformada em várias instruções MIPS formando um grupo equivalente.

2.2.6.1 Sistema (Hardware)

O sistema é formado por processadores superescalares MIPS R10000 modificados (64 bits) com suporte a execução fora de ordem. A comunicação entre eles é feita através de uma mescla entre redes em chip (NOC) e um sistema Crossbar, possibilitando desta forma que até 64 processadores MIPS possam ser instanciados. O modelo descrito (GS464) (HU, WANG, *et al.*, 2009) é composto por quatro processadores, onde cada processador possui duas ULAs de ponto fixo. Ambas executam operações de subtração, lógicas, deslocamento e comparação. Porém, a primeira unidade (ALU1) também executa instruções de chamadas de sistema UNIX (*trap*), movimentação condicional de dados e instruções de salto. A segunda unidade (ALU2), em contrapartida, executa operações de multiplicação e divisão. Além disso, existem mais duas unidades de ponto flutuante (FALU1 e FALU2), onde a primeira pode executar todas as instruções de ponto flutuante, e a segunda unidade executa instruções de adição, subtração e multiplicação. A memória de sistema suporta 64 bits de endereçamento virtual e 48 bits para endereçamento real. É possível acessar 128 bits (*quad word*) de dados em apenas um ciclo. Possui dois níveis de memória cache: L1 (instruções e dados) com 64 kbytes (*four-way associative*) e cache L2 de 512 kbyte.

A tradução de código x86 (CISC) para MIPS (RISC) não é eficiente por causa das diferentes características na semântica de cada conjunto de instruções. Pode ser citado como exemplo o uso de EFlags de estado, que são sinalizadores em hardware que fornecem informações a respeito do resultado de uma determinada operação (negativo, igual a zero, paridade, etc). No caso de instruções X86, a maioria das operações de ponto fixo calculam sinais de estado de acordo com o resultado da operação na ULA. Desta forma, a microarquitetura do Godson-3 foi estendida para auxiliar que a tradução de uma instrução x86 ocorra de forma mais eficiente. Podem ser citadas, além do suporte a *flags* de estado: suporte ao acesso a dados de ponto flutuante, suporte a instruções multimídia (MMX, SSE e SSE2), alterações no modo de endereçamento e mudanças no banco de registradores permitindo inserção e extração de bytes ao invés de apenas palavras.

2.2.6.2 *Tradutor binário (Software)*

O sistema de tradução binária do Godson-3 foi desenvolvido sobre um sistema operacional Linux, compilado no conjunto de instruções do Godson, ou seja, implementado com as instruções MIPS estendidas. O tradutor binário pode ser considerado intrínseco ao SO, já que provê diretamente as instruções x86 traduzidas para o processador. Desta forma, a camada do mecanismo de TB pode ser visualizada na Figura 2.9.

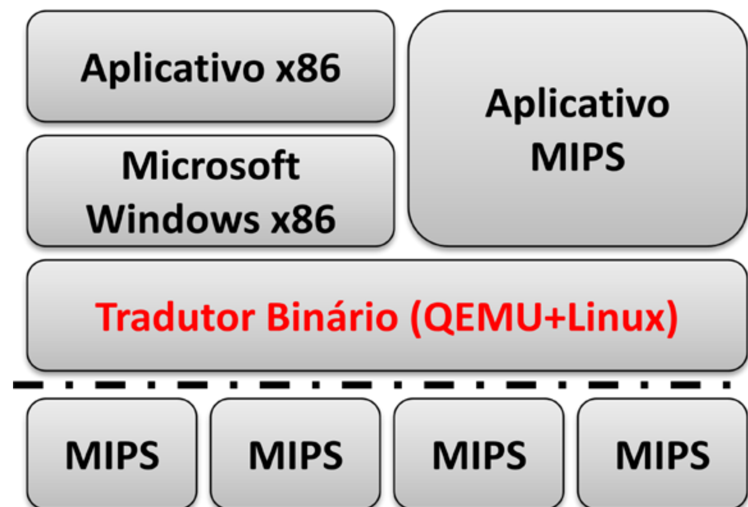


Figura 2.9: Tradutor binário situado na camada de interface.

O mecanismo de tradução é uma melhoria da máquina virtual QEMU (BELLARD, 2005) com melhorias que permitem tradução binária e otimização no código em tempo de execução. Inicialmente, a VMM interpreta as instruções x86 e as transforma em inúmeras instruções MIPS, continuamente analisando o comportamento da execução. No momento em que um trecho com alta frequência de execução é encontrado, além da tradução, são feitas também otimizações no código para a exploração de paralelismo entre instruções.

Apesar das otimizações serem muito eficazes, a utilização de um tradutor binário em software faz com que o BT seja executado de forma concorrente com o aplicativo, com tempo de computação superior à execução do mesmo código compilado para arquitetura nativa MIPS.

Number of instructions		Instruction				Comment
0		SUB	ECX	EDX		
1		JE	X86_target			

(a)

0.00		SUBU	Result	Recx	Redx	
0.01		SRL	Rsf	Result	31	/*SF=Result[31]*/
0.02		BEQ	Result	R0	L1	
0.03		ADD	Rzf	R0	R0	/*ZF=0*/
0.04		B	L2			
0.05		NOP				
0.06	L1:	ADDI	Rzf	R0	1	/*ZF=1*/
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
0.35		B	L8			
0.36		NOP				
0.37	L7:	ADDI	Rcf	R0	1	/*CF=1*/
0.38	L8:	ADD	Recx	Result	R0	
1.00		BNE	Rzf	R0	MIPS_target	
1.01		NOP				

(b)

0.0	SUBU	Result	Recx	Redx	/*Generating Sub result*/
0.1	**SETFLAG**				
0.2	**SUBU**	**Reflag**	**Recx**	**Redx**	**/*Generating EFLAGS*/**
1.0	**X86JE**	**Reflag**	**MIPS_target**		**/*Branch on EFLAGS*/**

(c)

0.0	SUB	Result	Recx	Redx	/*Generating Sub result*/
0.1	**X86SUB**	**Reflag**	**Recx**	**Redx**	**/*Generating EFLAGS*/**
1.0	**X86JE**	**Reflag**	**MIPS_target**		**/*Branch on EFLAGS*/**

(d)

Figura 2.10: Resultado do processo de tradução binária de instruções x86 para MIPS.

2.2.6.3 Desempenho do sistema

Na Figura 2.11 é demonstrado um gráfico de desempenho na execução de diversos algoritmos (compilados para x86), no qual devem ser feitas duas considerações:

- O gráfico normalizado demonstra que o valor máximo (100%) representa o algoritmo originalmente compilado para o conjunto de instruções MIPS, ou seja, com execução direta pelo hardware;
- Os dois testes realizados (com e sem suporte do hardware) demonstram a execução do mesmo algoritmo. Porém, agora, compilados para o conjunto de instruções x86. Desta forma, foi necessária a utilização do mecanismo de tradução binária.

Devido as diferenças entre os dois conjuntos de instruções, o suporte que o hardware provê um grande ganho na execução das instruções, que em média pode chegar a aproximadamente 4,5 vezes mais rápida quando comparada com a execução sem suporte em hardware. Entretanto, devido aos custos da tradução em software, nenhuma das execuções com tradução binária são equivalentes a execução do código nativo, chegando, em média, a apenas 68% do desempenho, aproximadamente.

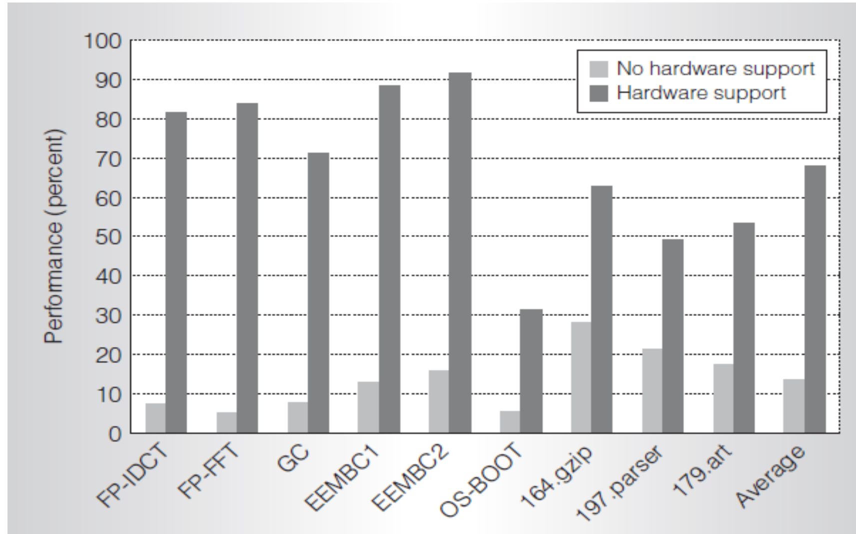


Figura 2.11: Desempenho do sistema ao executar diferentes aplicativos.

Em termos de potência, o Godson-3, que foi desenvolvido na tecnologia de 65nm utilizando 7 camadas de metal, executando a uma frequência de 1Ghz, teve um consumo entre 5 e 10W, dependendo da aplicação.

2.3 Considerações finais

Outro sistema de tradução binária bem conhecido é o JIT (*Just-in-time*) compiler (YANG, MOON, *et al.*, 1999) das máquinas virtuais Java. O JIT possui um mecanismo de tradução binária em software e aloca partes de código traduzidos em memória para reduzir os custos de desempenho. Podem ser citados também o sistema VEST (KIRK, 1993), que permite a tradução de imagens VAX e Ultrix MIPS para serem executadas em computadores Alpha AXP.

Visualizando todos os sistemas citados nesta capítulo, observam-se características interessantes em cada processo. Pode ser citado como característica comum, a existência de um sistema de otimização, podendo ela ocorrer de forma estática ou ainda de forma dinâmica. Independentemente da forma de otimização, a sua necessidade junto aos dispositivos de tradução binária permite que o código traduzido seja executado de forma aceitável ao usuário, apesar de que geralmente haja degradação de quando comparado à execução de código nativo em um processador nativo.

Outra questão diz respeito as camadas no qual cada tradutor binário é executado. O processo de tradução binária em software é mais flexível devido a possibilidade de executar o mesmo tradutor binário em outros processadores, somente recompilando o código fonte.

A implementação do TB em hardware reduz a queda do desempenho durante a execução de algoritmos, que pode ser até de apenas alguns ciclos por instrução traduzida. Entretanto, a flexibilidade do TB fortemente reduzida: o hardware não permite modificações, o que impossibilita atualizar ou até mesmo migrar para um novo conjunto de instruções.

A proposta deste trabalho possui uma abordagem diferente. Apesar de ser totalmente implementado em hardware (e com a premissa de tradução binária com alto desempenho), são utilizadas duas camadas de tradução binária. O primeiro nível é responsável por traduzir o código binário nativo para um código comum, enquanto o

segundo nível é responsável pela otimização do código a ser executado na arquitetura alvo. Em termos de flexibilidade, devido a interface bem definida entre ambos tradutores binários, existe a possibilidade de migrar de arquitetura nativa simplesmente trocando o primeiro nível de tradução, possibilitando que seja garantida a compatibilidade binária entre outros conjuntos de instruções. Da mesma maneira, é possível trocar de arquitetura alvo apenas mudando o segundo nível da tradução. Comparando a técnica proposta com outros sistemas de tradução, as maiores contribuições deste trabalho são:

- Custo de tradução reduzido, já que sua implementação é totalmente em hardware;
- Ganhos em termos de desempenho quando comparado com a execução do código nativo compilado para a arquitetura alvo original;
- Flexibilidade através da utilização da tradução binária de dois níveis, tornando simples migrar para uma nova versão do conjunto de instruções a ser traduzido, ou ainda migrar para uma nova arquitetura alvo.

3 ARQUITETURAS RECONFIGURÁVEIS

A descontinuidade da lei de Moore tem levado as empresas desenvolvedoras de processadores comerciais a mudarem as suas perspectivas em relação ao simples aumento da frequência de seus sistemas para obter ganhos substanciais em termos de desempenho. Os limites futuros no aumento da densidade de transistores no silício (que já se encontram próximos ao seu limite de miniaturização) é um fator impeditivo que implica no surgimento de novas tecnologias. O excesso de complexidade nos processadores, a adição de mais hardware e consequentemente o aumento de área em um chip, tem forçado a indústria a investir em pesquisas alternativas para a execução de código de uma forma mais eficiente.

Um ramo de pesquisa que visa substituir ou melhorar o desempenho dos métodos atuais de computação são as Arquiteturas Reconfiguráveis (AR). Basicamente estes sistemas objetivam criar componentes de hardware mutáveis, de forma que o mecanismo de execução se adapte de forma mais eficiente de acordo com as necessidades do software (Figura 3.1). Esta abordagem permite a exploração mais profunda de ILP, visando a computação paralela e combinacional de instruções de um determinado código. Esta possibilidade traz ganhos de desempenho do sistema (HENKEL e ERNST, 1997) (VENKATARAMANI, 2001), e por consequência, reduzindo também o consumo de energia (STITT e VAHID, 2002). Entretanto, tem-se um alto custo em termos de área ocupada pela organização do sistema em uma pastilha de silício.

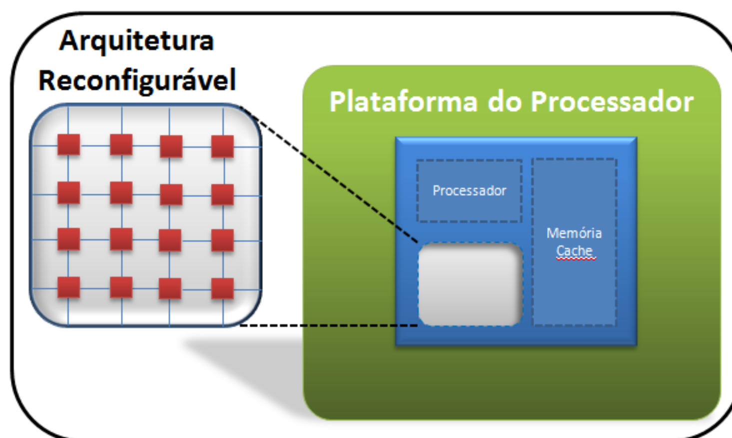


Figura 3.1: Arquitetura reconfigurável incluída em uma plataforma de processamento.

Atualmente as arquiteturas reconfiguráveis também estão abordando a execução eficiente de instruções baseada na exploração de TLP, utilizando mais de um mecanismo de processamento ou diferentes recursos de hardware. Entretanto, como este trabalho é baseado na melhoria de desempenho apenas em nível de ILP, neste capítulo serão descritas arquiteturas que fazem otimizações neste nível, para fins de contextualizar o uso de uma Arquitetura Reconfigurável destinada à otimização de execução de código no trabalho proposto. Deve ser considerado que este capítulo será apenas uma leitura complementar, já que o sistema reconfigurável não faz parte da contribuição deste trabalho.

3.1 Conceitos Básicos

Como descrito na introdução deste capítulo, Arquiteturas Reconfiguráveis são mecanismos inteligentes capazes de se adaptar conforme a necessidade do software. Em um dado aplicativo a ser otimizado, alguns dos quesitos que importantes a serem observados em um algoritmo são: grau de dependência entre instruções, número e tamanho *basic blocks*, e quantidade distinta de hot spots (pedaços de código que são executados diversas vezes durante o tempo de vida do programa). Todos estes quesitos são interessantes para definir a forma com que uma AR poderá se beneficiar de acordo com a organização interna de seu mecanismo.

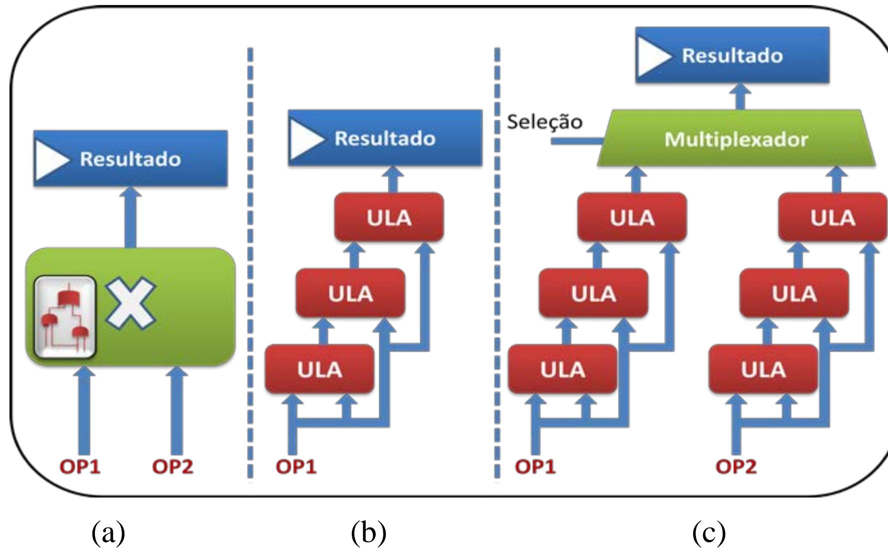
Deve-se observar que estes sistemas devem ser equilibrados em termos de:

Maximização do desempenho – O hardware deve ser projetado de modo que explore ao máximo as possibilidades de paralelismo e execução combinacional do software.

Minimização da complexidade – O sistema proposto deve manter-se simples o bastante para reduzir os custos de reconfiguração do sistema. Muitas vezes o tempo de reconfiguração de um determinado sistema se torna tão alto devido a sua complexidade, que os ganhos com a sua computação otimizada não amortiza o custo perdido com a sua. Os conceitos apresentados a seguir são baseados em (COMPTON e HAUCK, 2002).

3.1.1 Granularidade

A granularidade de reconfiguração está intimamente ligada a complexidade de reconfiguração. Basicamente existem duas formas de classificação de granularidade: grossa ou fina (Figura 3.2). Entretanto, esta classificação pode se torna relativa de acordo com os mecanismos a serem comparados.



(a) (b) (c)

Figura 3.2: ARs de granularidade fina (a) e granularidade grossa (b e c).

3.1.1.1 Granularidade Fina

Arquiteturas Reconfiguráveis de granularidade fina propõem mecanismos de reconfiguração no nível das portas lógicas, permitindo que o código seja altamente otimizado. Isso se deve ao fato de que o hardware da arquitetura possibilita que o software seja paralelizado ao menor nível possível, que é o bit. Por exemplo, na execução de um algoritmo que realiza uma multiplicação através de um algoritmo por somas sucessivas, a AR poderia identificar este algoritmo e reconfigurar o seu hardware para gerar um multiplicador dedicado a esta operação que seria executado rapidamente (Figura 3.2a). Este trecho de código, ao ser executado em um processador comum, poderia, dependendo dos operadores, levar centenas de ciclos para realizar esta operação. Podem ser citados como exemplos de componentes de hardware utilizados em um sistema de reconfiguração fina: PLAs e FPGAs.

Para este tipo de granularidade, pode-se afirmar que a complexidade tanto do algoritmo de reconfiguração quanto o algoritmo de otimização são altas. Além disso, exigem uma grande memória de contexto para alocar as configurações, tornando estas memórias consequentemente mais lentas e caras. Desta forma, deve-se ter o cuidado para que os ganhos atingidos com o alto desempenho de execução amortizem os custos decorrentes da arquitetura, caso contrário o sistema se tornará ineficiente.

3.1.1.2 Granularidade Grossa

Um mecanismo de reconfiguração de granularidade grossa define sistemas reconfiguráveis no nível das unidades lógicas, ou ainda, componentes de hardware mais complexos prontos e instanciáveis. Para esta granularidade, a reconfiguração pode ser feita a partir de operações completas. Estas operações geralmente são baseadas no código pertencente ao mesmo conjunto de instruções do processador no qual a AR está acoplada (mais detalhes no tópico 3.1.2).

Utilizando o mesmo exemplo do algoritmo para multiplicação por somas sucessivas, podemos adicionar ao algoritmo duas multiplicações com valores independentes. As otimizações no nível do hardware poderiam ser feitas de duas formas: adicionar unidades lógicas combinacionais em série para reduzir o tempo de computação (Figura 3.2b), ou ainda, adicionar unidades lógicas em paralelo para explorar paralelismo entre instruções (Figura 3.2c). Pode-se observar que para este exemplo, como o nível de

reconfiguração é mais alto, é necessário otimizar a execução das instruções do software. Já na granularidade fina seria possível montar um hardware dedicado e específico para a execução deste algoritmo. Entretanto, a reconfiguração de sistemas de granularidade grossa é muito mais simples, e define um sistema mais robusto em termos de tempo de reconfiguração. O grão deste tipo de reconfiguração pode ser, por exemplo, unidades funcionais ou até diversos processadores completos na mesma arquitetura.

3.1.2 Acoplamento

Outra característica das ARs é o grau de acoplamento do mecanismo de reconfiguração. Frequentemente, AR fortemente acopladas compartilham, por exemplo, banco de registradores, unidades lógicas ou ainda até mesmo parte do controle do núcleo. Estes sistemas podem ser fracamente acoplados e independentes de um processador, ou seja, não há nenhum tipo de compartilhamento direto de recursos de hardware.

Ambas as características inferem no grau de flexibilidade em termos de alteração do projeto em hardware. As arquiteturas do primeiro tipo explicadas anteriormente são fortemente acopladas, e possuem alta dependência com o processador. Desta maneira, interferem diretamente em diversas características, como: o caminho crítico, frequência de operação, tamanho da palavra de operação, quantidade de registradores, etc. O segundo tipo, fracamente acopladas, tem como principais defeitos o alto tempo de comunicação entre processador e a unidade reconfigurável, que implica também em altos custos em potência dissipada.

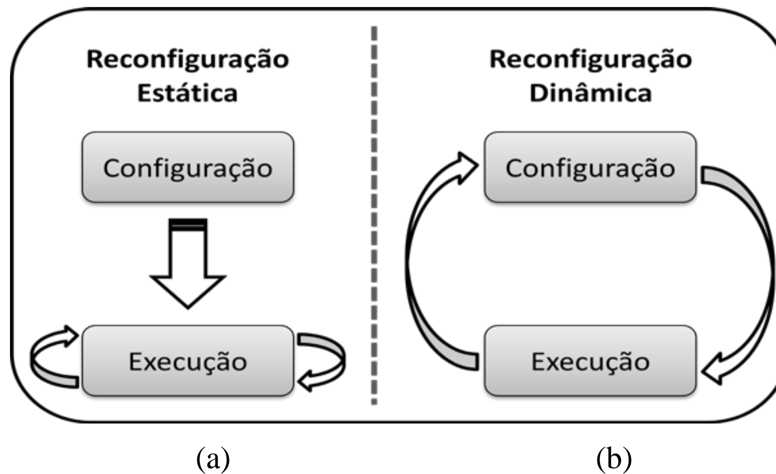
3.1.3 Modos de Reconfiguração

Existem duas características importantes para a reconfiguração. A primeira refere-se a possibilidade da reconfiguração ocorrer de forma estática ou dinâmica, já a segunda característica refere-se a capacidade do mecanismo se reconfigurar completamente ou somente uma parte (parcialmente) em sua organização em hardware.

3.1.3.1 Reconfiguração Estática x Dinâmica

Reconfiguração estática - Permite com que o hardware se adapte apenas em um único instante, isto é, anteriormente ao seu uso para executar uma aplicação. Este tipo de técnica é utilizada para aplicações mais específicas, muitas vezes bem comportadas e com certa regularidade no seu controle. Por exemplo, em FPGAs, a reconfiguração do seu hardware pode ser realizada mais de uma vez. No entanto, geralmente isto ocorre somente no momento em que a aplicação será alterada ou modificada. Anteriormente à execução a configuração deste hardware deverá ser feita estaticamente, gravada em uma memória e enviada para o FPGA. Este comportamento é ilustrado na Figura 3.3a.

Reconfiguração Dinâmica – Ocorre em tempo de execução do aplicativo. Desta forma, o hardware reconfigurável poderá se adequar as necessidades do software durante a sua execução. Existem otimizações que podem ser feitas somente em tempo de execução. E tempo de execução é possível, por exemplo, identificar quais sãos os trechos de código mais processados, e utilizar diferentes configurações do hardware para otimizar a execução destes trechos. O comportamento da reconfiguração dinâmica é ilustrado na Figura 3.3b.



(a) (b)

Figura 3.3: Princípios da reconfiguração estática (a) e dinâmica (b).

3.1.3.2 Reconfiguração Parcial

Esta característica se refere a capacidade de determinada arquitetura reconfigurar o seu hardware totalmente ou de forma parcial. Muitas vezes durante a execução de um algoritmo, parte de uma configuração pode ser reaproveitada devido a mudanças sutis no comportamento do software. Para se reduzir o tempo de reconfiguração, apenas partes da arquitetura são modificadas a fim de melhorar a execução. Por exemplo, o trabalho de (STITT e VAHID, 2002) permite que blocos complexos de hardware sejam instanciados de forma dinâmica e parcial na arquitetura, variando de acordo com a necessidade do software. O processo de reconfiguração de parcial do hardware pode ocorrer sem prejudicar outra configuração que está sendo executada. A grande vantagem da reconfiguração parcial é a amortização do tempo de reconfiguração.

3.2 Trabalhos Correlatos

Podem ser citados, como exemplos de arquiteturas reconfiguráveis, os sistemas Chimaera (HAUCK, FRY, *et al.*, 1997) e ConCISe (RAZDAN e SMITH, 1994). Ambas as arquiteturas são fortemente acopladas ao processador e possuem uma unidade de reconfiguração de grão fino, formada por lógica totalmente combinacional. O primeiro possui uma unidade lógica semelhante ao FPGA, o segundo, possui lógica semelhante a um CPLD. Estas unidades compartilham os recursos do processador adicionando mais um estágio no *pipeline* da sua organização. O reaproveitamento de hardware torna o controle da unidade simples, diminuindo os custos (em desempenho e potência) de comunicação entre a unidade reconfigurável e o restante do sistema.

Outro exemplo a ser citado é o processador TRIPS (SANKARALINGAM, 2004), que impôs mudanças no paradigma de programação, implicando o desenvolvimento de novas ferramentas e compiladores. A sua organização é formada por diversas unidades de granularidade grossa, com a premissa de executar trechos de código através de lógica combinacional. A sua composição através de diversos nodos, memória com operados e um roteador permite que sejam agrupados e configurados para explorar, de forma mais eficiente, diferentes tipos de paralelismo, desde o nível de dados até o nível de *threads*. Esta análise e alocação de trechos do código nos núcleos é realizada em tempo de compilação. Partilhando do mesmo conceito, pode-se citar, também, o processador Wavescalar (SWANSON, 2007).

Existe, ainda, o processador GARP (HAUSER e WAWRZYNEK, 1997), que possui uma unidade reconfigurável fracamente acoplada a um processador baseado na arquitetura MIPS. A comunicação entre o processador e a unidade reconfigurável é realizada através de instruções especiais e dedicadas *move*. Já a arquitetura Molen (VASSILIADIS, WONG, *et al.*, 2004), que possui uma unidade reconfigurável externa, utilizada para fazer otimizações nos núcleos principais do programa. Na Tabela 2 tem um resumo de diversas arquiteturas reconfiguráveis encontradas atualmente.

Tabela 2: Sistemas reconfiguráveis e suas características (BECK, 2010).

Name	GPP	Coupling	Granularity	Code Analysis/ Transformation
Chimaera	MIPS R400	Tightly	Fine	Hand-coded/modified GCC
GARP	MIPS II	Loosely	Fine	Hand-coded/modified GCC + special language
REMARC	MIPS	Loosely	Fine	Hand-coded/modified GCC + special language
Rapid	Standalone	-	Coarse	Modified C language
Piperench	-	Loosely	Coarse	Parameterized Compiler
Molen	PowerPC	Loosely	Fine	Hand-coded/modified GCC
Morphosys	TinyRISC	Loosely	Coarse	Hand-coded/Special Compiler
ADRES	VLIW	Tightly	Coarse	Framework (IMPACT + VLIW compiler)
Concise	MIPS-like RISC	Tightly	Fine	Automated search for hotspots/ C Translator to HDL
PACT-XPP	Standalone	-	Coarse	Special hardware language (NML)/Modified C
RAW	Standalone	-	Coarse	Modified C Compiler
Onechip	MIPS	Tightly	Fine	Modified C Compiler
PRISM II	AMD Am29050	Loosely	Fine	Hand-coded/modified GCC
Chess	Standalone	-	Coarse	Hand-coded
Nano	nP Core	Tightly	Fine	Hand-coded
DISC	-	Loosely	Fine	Hand-coded
Montium	Standalone	-	Coarse	Automated modified C to Montium Design Flow
XiRISC	RISC VLIW	Tightly	Fine	Dedicated Framework/GCC
Pleiades	ARM8	Loosely	Mixed	Hand-coded
ReRISC	RISC	Tightly	Coarse	Automated Search/Transformation
Napa	Compact RISC	Loosely	Fine	Dedicated Framework/special language based on C
Splash	Standalone	-	Fine	Hand-coded
DPGA	Standalone	-	Fine	Hand-coded
Colt	Standalone	-	Coarse	Hand-coded
Matrix	Standalone	-	Coarse	Hand-coded
DreAM	Standalone	-	Coarse	Hand-coded
Chamaleon	ARC Processor	Loosely	Coarse	Specific Framework/C-Based design
KressArray	Standalone	-	Coarse	Special Compiler/Framework

4 TRADUÇÃO BINÁRIA DE DOIS NÍVEIS

A proposta deste trabalho é baseada em um sistema dinâmico de tradução binária de dois níveis, que permite que a compatibilidade binária seja mantida e ainda que os custos relativos a tradução sejam amortizados. Basicamente, o TB de primeiro nível é responsável por traduzir o código nativo para um código comum (ou genérico), da mesma forma que um tradutor binário convencional. O segundo nível de tradução é responsável por otimizar e traduzir trechos ou sequências de instruções deste código comum para a arquitetura alvo (já traduzidas pelo primeiro nível). Neste trabalho foi realizado um estudo de caso onde o primeiro nível realiza tradução de instruções inicialmente compiladas para o conjunto de instruções x86 para o conjunto de instruções MIPS (adotado como código comum). Este código é, por sua vez, traduzido para ser executado em uma arquitetura reconfigurável, a qual é composta por um processador MIPS fortemente acoplado com um segundo nível de tradução binária e uma unidade reconfigurável fortemente acoplada de granularidade grossa (BECK, RUTZIG, *et al.*, 2008). Os trechos traduzidos são chamados agora de "configurações", e são alocados na TCache para serem executados novamente de acordo com o fluxo do programa, porém agora na unidade de reconfigurável. A Figura 4.1 ilustra uma visão geral da proposta do trabalho, e como a partir desta foi desenvolvido o estudo de caso.

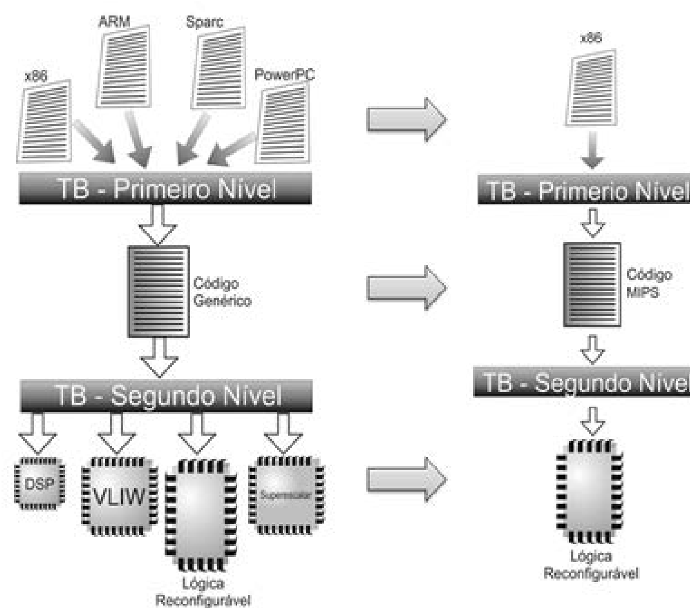


Figura 4.1: Mecanismo proposto para este trabalho.

4.1 Funcionamento do Sistema Proposto

Para um melhor entendimento da operação da arquitetura, pode-se considerar um aplicativo compilado para o conjunto de instruções x86 e que será executado pela primeira vez. Inicialmente, a unidade de tradução (primeiro nível) busca instruções na memória, de forma similar a um processador comum. Cada instrução x86 buscada na memória é traduzida pelo TB gerando, então, uma ou mais instruções MIPS equivalentes. Este conjunto gerado de instruções MIPS do tipo RISC realiza a operação equivalente a instrução CISC x86 original. No momento em que as instruções MIPS traduzidas estão sendo enviadas para o processador, o TB de primeiro nível calcula de forma paralela o novo endereço (PC) da memória de instruções. No primeiro nível de tradução, não é salvo nenhum dado para ser reutilizado no futuro: todos os dados são processados em tempo de execução de forma a minimizar os custos de memória. Estas instruções geradas pelo primeiro nível do tradutor binário são disponibilizadas ao processador MIPS de forma sequencial.

Ao mesmo tempo em que o processador MIPS executa cada instrução disponibilizada pelo TB de primeiro nível, o TB de segundo nível realiza a análise e otimização do código (que já fora traduzido para MIPS) para ser executado de forma eficiente na unidade reconfigurável. Ao finalizar este processo, é montada uma configuração indexada ao endereço (PC) da primeira instrução do trecho do código (*hotspot*) traduzido, e então guardada na TCache para um futuro reuso. No momento em que o mesmo trecho de código for encontrado pela segunda vez para ser executado, será encontrada na TCache uma configuração com o mesmo endereço PC, com código já traduzido pelo primeiro nível e otimizado pelo segundo nível. Desta forma, ocorre a execução direta do trecho na unidade reconfigurável, dispensado o uso do processador MIPS e dos dois níveis de tradução binária.

Considerando que, em média, cerca de 80% do tempo de computação de um aplicativo é baseado na execução de laços (KLAIBER, 2000), a grande vantagem de utilizar esta proposta é que no momento em que uma determinada sequência de código passa através dos dois níveis de tradução binária, a próxima vez em que o mesma sequência for executada novamente, um código traduzido pelo primeiro nível e otimizado pelo segundo nível já estará presente na TCache e, desta forma, os dois níveis de tradução não serão necessários para aquela sequência, ocorrendo a execução direta na unidade reconfigurável, conforme exibido na Figura 4.2.

Figura 4.2: Visão geral do trabalho proposto.

Desta maneira, os ganhos em termos de desempenho são decorrentes de:

- Ambos os mecanismos de TB são completamente implementados em hardware, ou seja, o controle das unidades são realizados em paralelo com suas funções. Desta forma, a tradução binária é rápida e com mínimo custo em termos de desempenho.
- O código, uma vez otimizado e traduzido, não fará uso do primeiro nível do TB, do processador MIPS, nem do segundo nível do TB quando da sua próxima execução. Estes recursos passam uma boa parte do tempo de computação desativadas, sendo o código executado somente na arquitetura reconfigurável.
- A unidade reconfigurável é utilizada para a execução do código otimizado da forma mais paralela o possível, sendo bem mais rápida quando comparada à execução em um processador MIPS padrão.

Além das vantagens ao executar código x86 de forma eficiente tanto em termos de desempenho quanto em energia, existe ainda a possibilidade de esta arquitetura executar código nativo MIPS. Isto é feito pela desativação completa do primeiro nível de tradução binária. Cada componente do sistema será melhor discutido nas próximas subseções.

4.2 Tradução de Código x86 para MIPS (Primeiro Nível)

Conforme citado anteriormente, o mecanismo de tradução binária de primeiro nível realiza a interpretação de instruções x86 e as traduz em uma ou mais instruções MIPS, funcionando como uma interface entre a memória e o processador. As instruções x86 (CISC) são buscadas da memória e as instruções equivalentes para o processador MIPS (RISC) são disponibilizadas. A implementação deste trabalho permite que sejam traduzidas cerca de 50 instruções diferentes, no total de 150 que faz parte do conjunto de instruções IA32. Porém, se for levado em conta todas as suas variações (modos de endereçamento, número de bytes de dados, códigos de repetição), o número de combinações possíveis sobe de dezenas para centenas de operações diferentes. Dentre as recursos não implementados estão as interrupções, operações de ponto flutuante e instruções multimídia (como MMX, SIMD e SSEx). Ressaltando que as instruções implementadas neste trabalho foram suficientes para a execução de todos os *benchmarks* testados.

O componente de hardware do primeiro nível do tradutor binário é composto por quatro diferentes unidades: Tradução, Montagem, PC (*Program Counter*) e finalmente a unidade de Controle, formando dois estágios de *pipeline* Estas unidades interagem de forma sincronizada entre si e de forma paralela, ou seja, a tradução de cada instrução x86 pode ser realizada ao mesmo tempo em que é calculado o novo endereço da próxima instrução e/ou juntamente com o processo de despacho de instruções MIPS para o processador. Na Figura 4.3 é proposta uma visão geral do mecanismo de tradução.

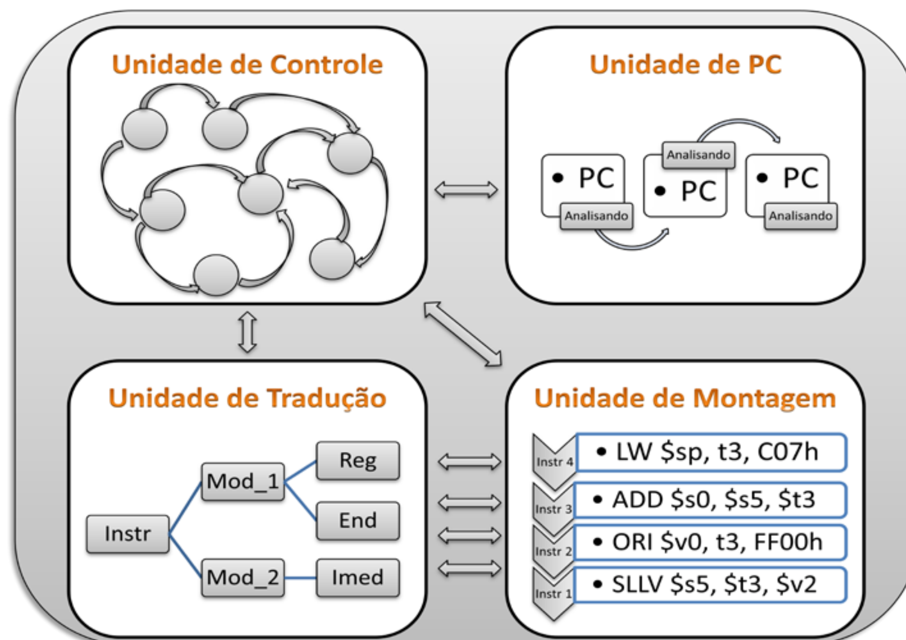


Figura 4.3: Unidades que compõem o mecanismo de tradução binária.

4.2.1 Unidade de Tradução

Esta unidade é o principal componente deste mecanismo. A Unidade de Tradução é responsável por buscar as instruções x86 da memória (uma instrução por vez), analisar o seu formato, quantidade de bytes que a compõe e classificá-la de acordo com a operação, operadores e modo de endereçamento, gerando então instruções equivalentes MIPS. Nesta unidade é realizado principalmente operações de acesso a tabelas. Sendo assim, seu hardware é composto principalmente por memórias do tipo ROM. Estas memórias guardam todas as instruções MIPS que possam ser utilizadas para gerar o conjunto equivalente de uma determinada instrução complexa. Desta forma esta unidade concentra a maior parte da área deste mecanismo.

Além de realizar a tradução, esta unidade deve fornecer informações para as unidades auxiliares como: a quantidade total de bytes no qual a instrução x86 é formada (importante para o cálculo do endereço da próxima instrução), o número de instruções MIPS geradas para verificar a disponibilidade na fila de instruções da unidade de montagem, e ainda informar o tipo de instrução (operação lógica, saltos condicionais ou incondicionais, flags de estado que poderão ser modificadas pela instrução, etc).

Esta unidade possui 4 diferentes tabelas e um hardware especial para manipular valores imediatos. Estas tabelas estão diretamente relacionadas com os campos encontrados em uma instrução x86. Estes campos são melhor ilustrados no tópico 4.3.2.

Os dados de cada campo (prefixo, opCode, modo de endereçamento, SIB) são entradas para as quatro tabelas, enquanto o conteúdo imediato (descolamento e dados) são enviados para o hardware de manipulação de imediatos. Em cada uma destas tabelas e hardware de imediatos, terá como saída bits de configuração para serem utilizadas posteriormente. Dependendo da instrução, algumas destas tabelas irão retornar valores nulos, ou seja, algumas tabelas não são utilizadas e dependem do formato da instrução. As cinco saídas com os bits de configuração são enviadas para o componente de hardware que analisa estes bits de configuração. Este componente é responsável por

descartar bits de configuração que não serão utilizados e os coloca em ordem correta. Alguns bits são descartados devido ao processamento das instruções serem geradas em paralelo considerando somente o campo de instruções, mas não a instrução propriamente dita. A Figura 4.4 ilustra este processo com maior detalhes. Após a geração das instruções equivalentes, a tabela de OPcode também aloca o tamanho de cada instrução que será enviada para a Unidade de PC.

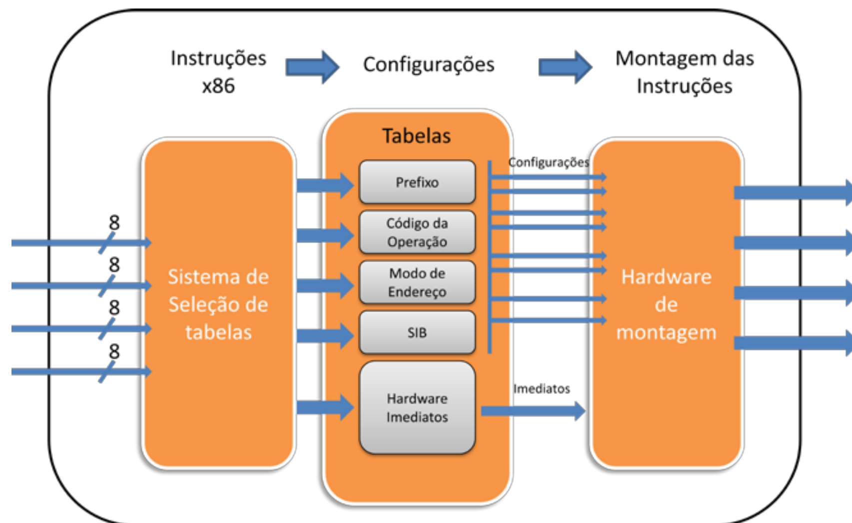


Figura 4.4: Geração dos bits de configuração correspondente a instrução x86 da entrada.

Neste instante, são conhecidos: a operação da instrução e os operadores que compõem determinada instrução x86. Neste momento esta informação está em um formato intermediário composto por bits de configuração. Na sequência, os bits de configuração são enviados para a unidade que decodifica esta informação e a partir dela monta a instrução no formado do conjunto MIPS. As instruções MIPS são geradas dependendo da sua operação: nos formatos R, I ou J (uma melhor análise consta no tópico 4.3.1), como ilustrado na Figura 4.5.

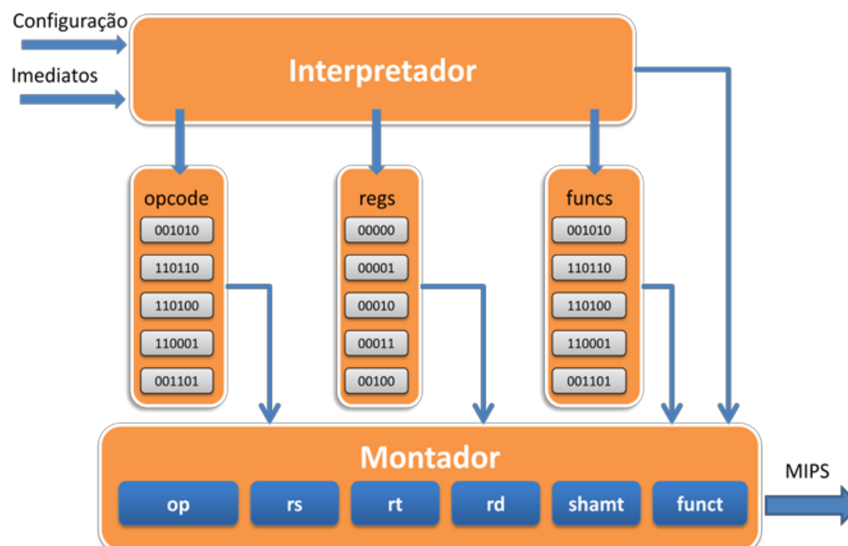


Figura 4.5: Bits de configuração chegando na unidade de interpretação.

4.2.2 Unidade de Montagem

O processador MIPS utilizado busca novas instruções vindas do TB de forma sequencial, da mesma forma que o processador original buscasse suas instruções de uma memória regular. Desta forma, a principal funcionalidade da Unidade de Montagem é disponibilizar as instruções equivalentes MIPS de forma sequencial e transparente para o processador como se fosse uma memória comum. A sua unidade é formada por uma fila de registradores de 32 bits, nos quais as instruções provenientes da Unidade de Tradução são alocadas. Além disso, a cada ciclo é enviado para a Unidade de Controle a quantidade de instruções que estão ocupando os registradores da fila, de forma a garantir que não existirá descontinuidade no processo de fornecimento de instruções MIPS para o processador, reduzindo eventuais bolhas ou paradas na execução do código que possam surgir.

4.2.3 Unidade de PC

Ao contrário do conjunto de instruções MIPS que possuem um formato padrão em termos de número de bytes que compõe cada instrução (4 bytes), as instruções x86 possuem tamanhos variáveis, impossibilitado o reaproveitamento do hardware do processador MIPS para o cálculo do próximo endereço na busca de novas instruções. Desta forma, foi implementada a Unidade de PC, que calcula o próximo endereço da nova instrução a ser buscada na memória, baseando-se na quantidade de bytes em que é composta a instrução x86 corrente, e somando esse número ao endereço atual, ou ainda calculando o endereço através de um valor imediato contido em uma instrução de salto incondicional.

4.2.4 Unidade de Controle

A Unidade de Controle permite que seja mantida a sincronia e a consistência da informação entre as unidades de Tradução, Montagem e de PC. Cada unidade possui sinalizadores, informando a Unidade de Controle o seu estado atual. A unidade de controle pode repassar informações como: o momento em que a Unidade de Tradução pode buscar uma nova instrução na memória ou ainda a forma e o instante em que deve ser calculado o endereço de PC. Esta unidade é baseada em uma máquina de estados finitos.

4.3 Processador MIPS (Estendido)

Como citado anteriormente, neste trabalho foi utilizado o processador MIPS R3000, tendo em vista que a grande vantagem do uso do conjunto de instruções MIPS é a regularidade do código bem como o seu conhecido comportamento. Desta forma, se torna simples realizar a tradução de outro conjunto de instruções para esta. Entretanto, a tradução de um conjunto de instruções complexo, como foi feito neste trabalho, em muitos casos para gerar um código equivalente a uma instrução CISC x86, são necessárias diversas instruções MIPS (HU, WANG, *et al.*, 2009).

Estas diferenças, tanto na semântica do processador quanto na estrutura do código, poderão ser melhor analisadas após uma breve revisão nos dois tópicos apresentados a seguir, que discutem particularidades das arquiteturas MIPS (RISC) e x86 (CISC).

4.3.1 Processador MIPS

O primeiro processador MIPS foi desenvolvido na universidade de Stanford, em 1981, por John Hennessy. A premissa era que um processador do tipo RISC seria mais

rápido que os processadores CISC devido a sua simplicidade, tanto na arquitetura como no conjunto de instruções. O processador MIPS, por ser desenvolvido sob uma plataforma RISC, possui um conjunto de instruções "bem comportado" no que diz respeito a sua organização, formato e tamanho de instrução. A sua memória é baseada na arquitetura Harvard, onde os dados e as instruções ficam armazenados em memórias separadas, facilitando o acesso simultâneo das informações. Neste trabalho, o processador base será o MIPS R3000, com arquitetura de 32 bits e 32 registradores de propósito geral. O grande número de registradores (quando comparado a arquitetura x86) se deve ao fato que o tempo de acesso a operandos em registradores é muito menor do que o tempo de acesso a memória (HENNESSY e PATTERSON, 2007). Além disso, como a sua arquitetura é baseada em instruções de leitura/escrita (*load/store*) de dados na memória, necessita-se de uma grande quantidade de registradores para alocar os dados a serem computados.

Suas instruções são baseadas em três tipos: instruções do tipo J, R e I. Cada tipo possui um formato diferente. No entanto, o tamanho da palavra da instrução é sempre o mesmo: 32 bits. Desta forma, para encontrar a próxima instrução dentro de um bloco base, somente é necessário deslocar 4 bytes em relação ao endereço da instrução atual. A Figura 4.6 exibe o formato básico das instruções MIPS de 32 bits.

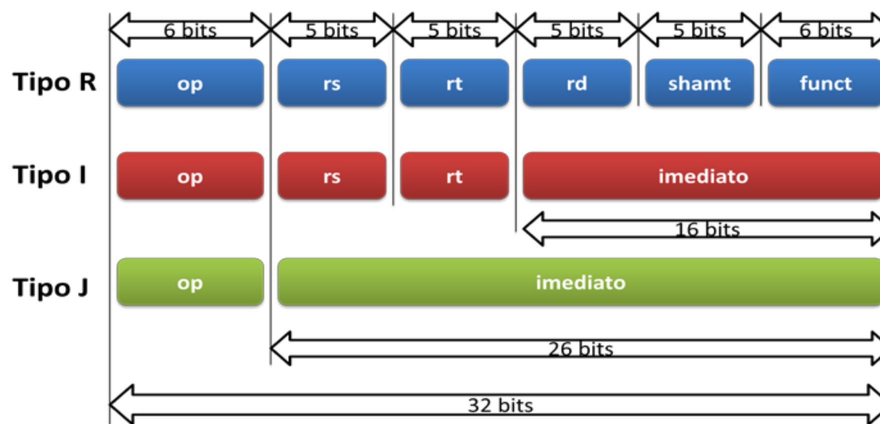


Figura 4.6: Os três formatos de instruções MIPS.

O processador MIPS também não é baseado em Flags para a execução de instruções de salto. As operações de comparação entre dois operadores, muito comuns em instruções de salto, são realizadas com instruções específicas (e.g.: slt – *set on less than*). Já em processadores baseados em Flags, bits de estado são gerados (e.g. se o resultado foi igual a zero, negativo, com *carry*, se ocorreu overflow, etc) de forma automática sempre que instruções aritméticas são executadas. Estes flags serão posteriormente utilizados para a verificação de que se uma instrução de desvio é verdadeira ou não.

4.3.2 Processador x86

Este conjunto de instruções foi implementado inicialmente pela (INTEL, 2011) no processador 8086, porém diversas empresas (Cyrix, AMD, VIA e outras) seguiram o padrão para competir pelo mesmo mercado. Uma grande parte dos computadores (desktop e laptop), assim como servidores e estações foram desenvolvidas baseados no conjunto de instruções x86. Uma das maiores vantagens deste conjunto de instrução é a grande quantidade de aplicativos e sistemas operacionais existentes. Por exemplo, o MS-DOS, Windows, Linux, Solaris foram desenvolvidos para esta plataforma.

O conjunto de instruções x86 já está presente no mercado há mais de 30 anos e neste período houve severas modificações. O número de instruções especializadas aumentou com o passar dos anos, porém sempre com a preocupação de que compatibilidade retroativa de código com as arquiteturas mais antigas fosse mantida. Por serem instruções do tipo CISC, o seu formato não segue um padrão definido de formatação, bem como a quantidade de bits que as compõe. Consequentemente, o deslocamento do PC (*Program Counter*), responsável por informar o endereço na memória da próxima instrução, também é variável.

Os desvios condicionais desta arquitetura são baseados em *EFlags* que compõem registradores de condição. Como já explicado, estes registradores são alterados a todo o instante durante a execução do código. Em grande parte da execução, estas *flags* são muito utilizadas para comparar o valor de determinado resultado. A vantagem nesta técnica é que é mais fácil testar estes registradores do que comparar resultados com conteúdos da memória. No entanto, este fato traz prejuízo para o processamento devido ao aumento no tempo de realização da operação (computar e atualizar *flags*). Além disso, grande parte destas operações de cálculo de flags não são utilizadas, sendo sobrescritas pela próxima operação.

Existem diversas regras e padrões do conjunto de instruções x86 que definem exatamente a função de determinada instrução. Na Figura 4.7, é possível visualizar algumas definições do código da instruções.

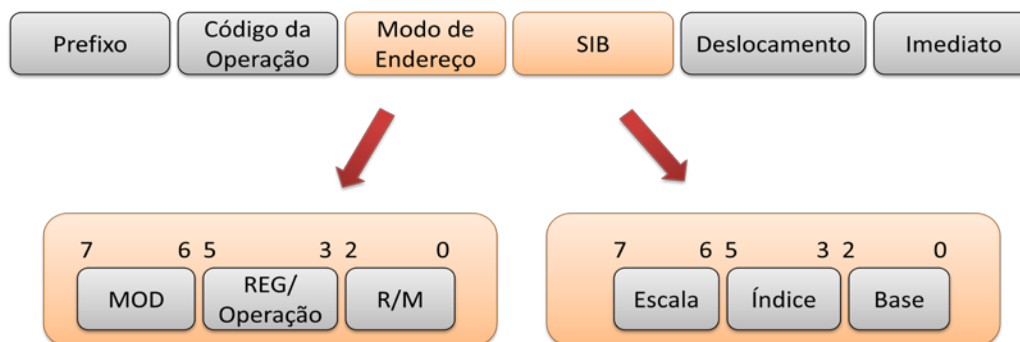


Figura 4.7: Possibilidades de composição de uma instrução x86.

Tendo em vista esta alta complexidade na montagem das instruções, o cálculo do PC para a próxima instrução direta (sem saltos) é baseado no valor atual do endereço somado a quantidade de bytes da instrução anterior. Houve uma grande mudança no conjunto de instruções x86 de 16 bits para 32 bits. Além da alteração do tamanho da palavra, foi adicionado um byte adicional de endereçamento chamado SIB (do inglês, *Scale Index Base*) que possibilita agrupar em uma instrução a realização de operações adicionais nos operadores. Desta forma, uma instrução x86 pode variar o código da instrução entre 1 a 15 bytes. As diretivas relacionadas às instruções podem ser definidas nos 4 primeiros possíveis componentes de uma instrução:

- *Prefixo* – 1 byte (se necessário);
- *Código da operação* – 1 ou 2 bytes de código;
- *Modo de endereçamento* – 1 byte (se necessário);
- *SIB* – 1 byte (se necessário);
- *Deslocamento* – 1, 2 ou 4 bytes (se necessário);
- *Imediato* – 1, 2 ou 4 bytes (se necessário).

Outra observação interessante a ser feita da arquitetura x86 é a possibilidade de manipular variáveis de 8, 16 e 32 bits. Porém, esta característica não é exclusividade de arquiteturas CISC, sendo encontrada também em arquiteturas RISC como a ARM (ARM, 2011).

4.3.3 Extensão do conjunto de instruções

Devido as grandes diferenças arquiteturais ente os conjuntos de instruções x86 e MIPS, foi necessário realizar modificações no hardware que permitem com que o conjunto de instruções MIPS padrão seja estendido e o processo de tradução binária seja mais eficiente. A Tabela 3 demonstra um exemplo de uma sequência de instruções x86 traduzidas para o código MIPS. Neste caso 18 instruções MIPS foram geradas de apenas 5 instruções x86.

Tabela 3: Um exemplo de tradução de instruções x86 para MIPS.

x86 Instructions	**Translated MIPS Instructions**	**Operation**
ADD EAX, B340	ADDIU r8, r0, 40	r0=0 r8=GP0 r9=GP1 r16=EAX
	ADDIU r9, r0, B3	
	SLL r9, r9, 15	
	ADDU r9, r8, r9	
	ADD r16, r16, r9	
PUSH ES	SUBI R20, R20, 1	r8=GP0 r20=SP
	ADD R8, R8, R20	
	SW R7, R8(0)	
OR EDX, EAX	OR r19, r19, r16	r16=EAX r19=EDX
XOR AL, C2	ADDIU r9, r0, C2	r16=EAX r9=GP1 r0=0
	XOR r16, r16, r9	
AND EAX, [48F6 + BP]	ADDIU r8, r0, F6	r16=EAX r21=BP r8=GP0 r9=GP1
	ADDIU r9, r0, 48	
	SLL r9, r9, 15	
	ADDU r9, r8, r9	
	ADDU r9, r9, r21	
	LW r8, r9(0)	
	AND r16, r16, r8	
GPx - General Purpose Register		

Como já explicado, existem na arquitetura x86, por exemplo, registradores de sinalização (*flags*) de estado, nos quais são automaticamente atualizados em muitas operações lógicas, aritméticas e utilizados na maior parte por instruções de salto. Entretanto, este tipo de operação não é suportada pela arquitetura MIPS. Neste caso, dezenas de instruções adicionais são necessárias por instrução x86 para que estes registradores de estado sejam emulados em um processador MIPS padrão. Além disso, com instruções x86, é possível utilizar um determinado dado contido na memória como

operador em instruções aritméticas, ao contrário das instruções RISC, no qual são necessárias duas instruções para realizar a mesma operação (leitura e operação). Isto ocorre devido aos modos de endereçamento segmentados, além de muitas outras possibilidades de construção de uma instrução complexa. Um exemplo pode ser visualizado na Tabela 4.

Tabela 4: Operações realizada por uma instrução complexa.

Instrução	Operação
ADD EAX, [0x1234]	Busca o conteúdo do dado na memória através do endereço imediato;
	Realiza uma soma entre o conteúdo da memória e o conteúdo do registrador EAX;
	Guarda o valor do resultado no registrador EAX;
	Realiza o cálculo de flags.

Para reduzir estes custos de tradução, foi necessário estender o conjunto de instruções do MIPS fazendo uma adaptação ao hardware existente e disponibilizando um suporte também em hardware para o cálculo das variáveis de estado (*EFlags*), mantendo a compatibilidade com o conjunto de instruções padrão MIPS. Foram realizados testes em ambiente de simulação a respeito do cálculo e do uso das *EFlags*. Os resultados restaram demonstrados no gráfico da Figura 4.8, onde se exibe a quantidade de vezes em que cada *flag* é calculada e faz um comparativo com a quantidade de vezes em que estes sinais são utilizados de fato. Conclui-se que existe uma ineficiência em relação ao cálculos e o uso das *Eflags* de uma forma geral. Se for levado em conta que para o cálculo de cada *flag* for gerada diversas instruções MIPS equivalentes, percebe-se que a maior parte da execução destas instruções serão desnecessárias, já que estes valores muitas vezes são sobrescritos.

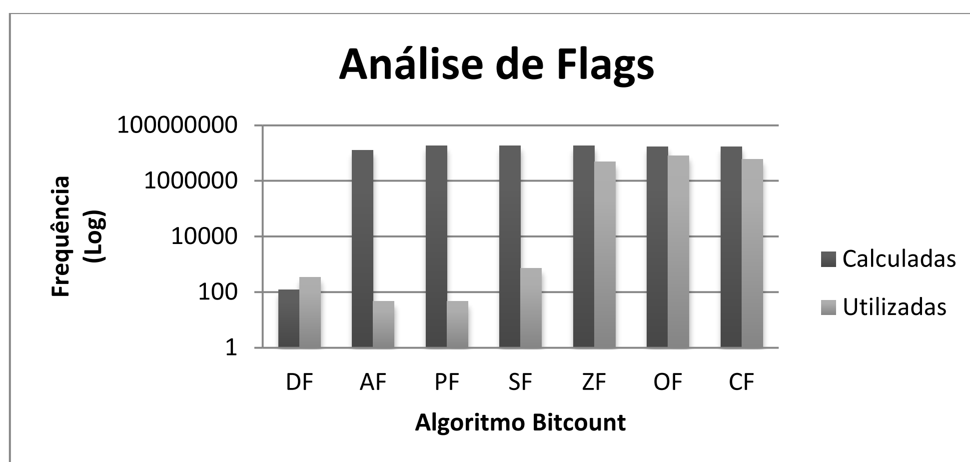


Figura 4.8: Comparativo entre o número de vezes em que cada *flag é* calculada e o número de vezes em que são utilizadas.

As modificações feitas na microarquitetura do MIPS são identificadas abaixo:

Suporte a Sinais de Estado – Foi adicionado um hardware que gera o valor dos sinais de Estado a partir dos operadores e do resultado de forma similar a um processador x86 nativo, onde estes valores são escritos em um registrador mapeado dentro do banco de registradores no processador MIPS.

Manipulação de byte – Diversas operações que ocorrem no código x86 são baseados em manipulação de variáveis de 8, 16 e 32 bits em um registrador. Pelo fato de que a arquitetura do processador MIPS manipula somente variáveis de 32 e 16 bits (em alguns casos), o conjunto de instruções MIPS foi estendido para suportar este tipo de operação. Em hardware foram feitas modificações no banco de registradores para manipular variáveis menores que 32 bits.

Modo de Endereçamento – O processador MIPS é constituído por uma arquitetura de leitura e escrita (*load/store*). Em contraste com o conjunto de instruções x86, no qual suporta diversos modos de endereçamento, o processador MIPS suporta apenas modo de endereçamento do tipo Base (registrador) + Deslocamento (imediato). Para reduzir esta diferença e reduzir os custo de tradução, foi inserido dois novos modos de endereçamento para instruções de leitura e escrita: Base (registrador) + Deslocamento (registrador) e Base + Deslocamento (registrador) + Imediato (byte).

4.4 DIM (Dynamic Instruction Merging)

De acordo com o comentado no Capítulo 1, o primeiro nível de tradução binária é acoplado a uma arquitetura reconfigurável já implementada (BECK. e CARRO, 2005) (BECK, RUTZIG, *et al.*, 2008). A arquitetura DIM é composta por um sistema de tradução binária (neste trabalho, este TB é definido como o segundo nível de tradução), uma TCache onde são alocadas as configurações geradas pelo TB e uma unidade reconfigurável responsável pela execução das configurações.

4.4.1 Organização da Unidade Reconfigurável

A unidade reconfigurável (BECK, RUTZIG, *et al.*, 2008) é basicamente uma matriz dinâmica de grão grosso, ou seja, a sua reconfiguração ocorre em unidades funcionais completas. Esta unidade é fortemente acoplada ao processador (COMPTON e HAUCK, 2002), que é uma abordagem similar a utilizada no Chimaera (HAUCK, FRY, *et al.*, 1997). Na fase de execução do código, a unidade reconfigurável funciona como uma extensão do estágio de execução do processador, mas de forma totalmente independente e sem a necessidade nenhum tipo de acesso externo (no ponto de vista do processador).

Uma visão geral da organização do sistema é exibida na Figura 4.9. A matriz de unidades funcionais possui duas dimensões, onde cada instrução é alocada na interseção entre uma linha e uma coluna. As unidade funcionais alocadas lado a lado podem executar operações em paralelo, já as unidades alocadas na mesma coluna se caracterizam por executar as operações de forma combinacional, uma após a outra. Desta forma, se duas instruções não possuem nenhum tipo de dependência de dados entre si, existe a possibilidade de serem alocadas na mesma linha, ou seja, com execução paralela. Caso contrário, serão obrigatoriamente alocadas em linhas diferentes.

Cada coluna é homogênea, isto é, possui um número determinado de unidades funcionais idênticas, podendo ser elas: ULAs, multiplicadores, etc. Dependendo do tempo de execução de cada unidade funcional (caminho crítico), mais de uma operação pode ser executada no período equivalente a um ciclo do processador. Por outro lado, as operações mais complexas, como por exemplo a multiplicação, demoram mais tempo para a execução de cada operação (dependendo da tecnologia e a forma de implementação do hardware). Já as operações de leitura e escrita de dados na memória (*Load/Store*) pertencem um grupo diferente da Unidade Reconfigurável. O número de unidades deste tipo disponíveis em paralelo depende da quantidade portas disponíveis na memória de dados. Nesta versão da Unidade Reconfigurável, não está disponível as

operações de ponto flutuante. Nota-se que TB de primeiro nível também ainda não suporta este tipo de instruções. No exemplo demonstrado na Figura 4.9, o primeiro grupo de unidades funcionais suporta a execução de até 4 operações em paralelo de unidades lógicas/aritméticas, já no segundo grupo foram alocadas duas unidades de leitura/escrita em paralelo. Já no terceiro grupo, apenas uma unidade de multiplicação é suportada.

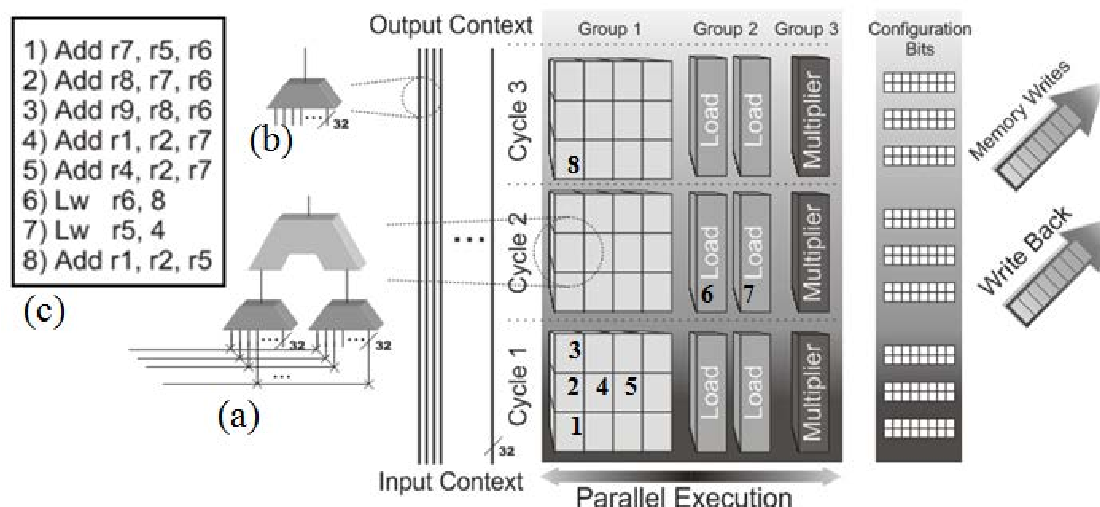


Figura 4.9: Exemplo de uma configuração da Unidade Reconfigurável a partir de uma sequência definida de instruções (BECK, RUTZIG, *et al.*, 2008).

O processo de reconfiguração funciona da seguinte forma: primeiramente, os valores contidos no banco de registradores são enviados para um barramento de operadores, no qual cada linha do barramento é conectado à todas as unidades funcionais por meio de multiplexadores. Eles são configurados para que o valor da linha correta do barramento seja enviado para a unidade funcional, como podemos observar na Figura 4.9a. Estes multiplexadores são chamados de "multiplexadores de entrada". Existem também, na sequência de cada unidade funcional, os chamados "multiplexadores de saída", ilustrados na Figura 4.9b. Após cada operação, os multiplexadores de saída selecionam em qual linha do barramento será destinado o resultado da operação.

Considerando o exemplo da Figura 4.9, vemos como foi alocado uma determinada sequência de instruções na Unidade Reconfigurável. No quadro da Figura 4.9c, observamos que a primeira instrução é alocada normalmente na primeira unidade lógica, guardando o resultado da operação no registrador "r7". A segunda instrução foi alocada na linha seguinte, pois ela terá que operar com o registrador "r7". Este dado deve estar atualizado com o resultado da primeira instrução. De forma similar, a terceira instrução também é dependente do resultado da segunda instrução (registrador "r8"). Assim, é novamente alocada na próxima linha. A quarta e a quinta instruções possuem apenas dependência verdadeira de dados com a primeira instrução (registrador "r7"), deste modo, poderão ser executadas antes da terceira instrução e em paralelo com a segunda. As instruções 6 e 7 possuem dependência de dados verdadeira com várias instruções (1, 2 e 3) através dos registradores "r6" e "r5". As duas instruções foram alocadas no ciclo seguinte, após a execução das duas operações de leitura (6 e 7). Enfim, a instrução 8 possui dependência direta com a instrução 7 através do registrador "r5" impossibilitando a alocação desta instrução no mesmo ciclo que as instruções anteriores, sendo alocada no próximo ciclo (ou nível) apenas.

4.4.2 Tradutor Binário (Segundo Nível)

O tradutor binário de segundo nível foi entendido de uma versão anterior (BECK, RUTZIG, *et al.*, 2008) e adaptado para este projeto. A sua execução se resume na análise e otimização de blocos base. Os blocos básicos (*basic blocks*) são sequências de código situadas entre saltos, desvios ou chamadas de função. Deste modo, a execução do TB inicia na primeira instrução logo após uma instrução de desvio, e finaliza a tradução ao encontrar uma instrução não suportada (instruções de ponto flutuante, por exemplo) ou ainda outra instrução de desvio (quando especulação é atualizada e é atingido o seu limite). A cada instante em que a matriz de unidades funcionais é reconfigurada, existe um custo, que deve ser amortizado através a execução mais eficiente das instruções. Portanto, um número mínimo de 3 instruções foi definido para que uma configuração seja gerada e alocada na Tcache para uma possível reutilização no futuro.

4.4.2.1 Estrutura do hardware

A implementação em hardware do tradutor binário possui quatro estágios, com as seguintes funcionalidades: decodificação, análise de dependências, atualização de tabelas, montagem de configuração. Um conjunto de tabelas são utilizados para manter as informações a respeito da sequência de instruções que estão sendo processadas, sendo que algumas tabelas intermediárias são utilizadas apenas na fase de montagem da configuração e não são salvas posteriormente na TCache (são desnecessárias durante a fase de reconfiguração). Cada tabela é responsável por armazenar dados como: o configuração dos multiplexadores de entrada e saída, tipo de operação realizada pelas ULAs, mapa binário de alocação de cada instrução na matriz, tabela de dados a serem buscados no banco de registradores, etc. Mais detalhes podem ser visto em (RUTZIG, S. e CARRO, 2008) (BECK, RUTZIG, *et al.*, 2008). Podemos observar na Figura 4.10 que todo o processo de tradução ocorre de forma paralela ao processador, sem afetar o caminho crítico do sistema.

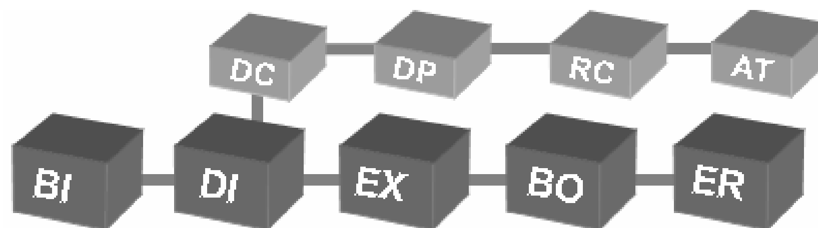


Figura 4.10: Estágios do Tradutor Binário anexados ao *pipeline* do processador

A instrução é decodificada no primeiro estágio, fornecendo para o segundo estágio informações a respeito da instrução, como: tipo de operação, dados imediatos, operadores, etc. O segundo estágio inicialmente verifica as dependências (falsas e verdadeiras) para manter a coerência dos dados durante a execução. O terceiro estágio atualiza as tabelas de configurações com informações a respeito da nova instrução que foi adicionada ao conjunto no ciclo anterior. Finalmente, no último estágio, quando houver uma quebra de configuração (causada por uma instrução de salto, instrução não suportada ou ainda por falta de recursos em hardware), todas as tabelas são organizadas e uma configuração é montada para ser guardada na TCache.

4.4.3 Cache de Tradução (TCache)

A TCache é uma memória baseada no algoritmo de substituição FIFO (do inglês *first in, first out*) e implementada utilizando o modelo totalmente associativo. Esta cache

especial é utilizada para alocar todas as configurações geradas pelo tradutor binário para uma futuro reutilização. As configurações alocadas são compostas por bits de reconfiguração da Unidade Reconfigurável. Como já explicado, estes bits são responsáveis pelo controle dos multiplexadores de entrada e saída, além dos bits que controla a operação a ser realizada por cada unidade funcional e os dados imediatos contidos em determinadas instruções.

Uma importante modificação realizada neste trabalho é em relação a indexação do PC. Em trabalhos anteriores, o PC era indexado através das instruções MIPS. Devido ao acréscimo do primeiro nível de tradução, a indexação é realizada através do PC x86 gerado pelo TB. Desta forma, ao se encontrada uma configuração pronta na TCache, os dois níveis de TB podem ser desativados temporariamente, como demonstrado na Figura 4.2).

5 METODOLOGIA E RESULTADOS

Neste capítulo serão apresentados todos os métodos e configurações do trabalho proposto, bem como as ferramentas utilizadas neste processo. No próximo tópico será exibida uma breve descrição de cada ferramenta utilizada no processo de simulação. Na sequência serão apresentados os resultados, fruto da interpretação dos dados extraídos das simulações.

5.1.1 Simuladores (Software)

Para extrair os resultados contidos neste trabalho, foi utilizado uma sequência de três simuladores em cadeia: Simics (MAGNUSSON, CHRISTENSSON, *et al.*, 2002), simulador comercial, com licença gratuita para estudantes), simulador do TB de primeiro nível (contribuição deste trabalho), simulador da arquitetura reconfigurável DIM (RUTZIG, S. e CARRO, 2008) (já existente de trabalhos anteriores).

5.1.1.1 SIMICS

O Simics é um simulador de plataformas com precisão no nível de instruções, permitindo até que sistemas operacionais completos (Linux, Solaris, Windows, etc) sejam emulados. Além disso, é capaz de simular diversas arquiteturas (x86, Sparc, MIPS, etc) e sistemas multiprocessados. Para este trabalho foi utilizada uma instância do Simics que simula uma máquina de núcleo simples x86 (Pentium II) executando em seu ambiente virtual o sistema operacional Linux Red Hat 6.2.

Através de scripts é possível realizar a chamada de execução dos algoritmos em ambiente simulado, podendo ser extraída desta simulação o traço de execução do programa. Para tanto, foi necessário modificar o código fonte dos algoritmos inserindo as chamadas "*magic instructions*" (instruções mágicas) antes da compilação dos algoritmos. As instruções mágicas tem como finalidade determinar o início e o fim da análise e extração do traço de código de um algoritmo. Neste caso, estas instruções especiais foram inseridas no início e fim da função principal do código (*main*) a ser analisado, permitindo que as instruções que posteriormente serão analisadas pertençam apenas a aquela aplicação. Outra importante característica é a possibilidade de separar as instruções de aplicativo das instruções de kernel do sistema simulado (Linux), utilizando um componente do Simics chamado "Tracker".

A medida em que o algoritmo é executado, o simulador Simics disponibiliza um arquivo com o todas as instruções que foram encaminhadas para o processador virtual. Neste arquivo há informações importantes de cada instrução a ser executada como: o valor do endereço do PC, quantidade de instruções executadas até o momento, código

em hexadecimal da instrução e descrição da instrução em formato de texto. Estas informações são disponibilizadas em uma mkfifo (arquivo de dados do tipo FIFO que permite a alocação e liberação de espaço em memória de forma dinâmica, na medida em que os dados são escritos e lidos) para serem consumidas pelo simulador x86.

5.1.1.2 Tradução x86 para MIPS

A implementação do tradutor binário de primeiro nível é umas das principais contribuições deste trabalho. O simulador em software foi desenvolvido em linguagem C e objetiva a geração de resultados de forma mais rápida e com precisão a nível de instrução. Este algoritmo realiza o tratamento de cada instrução que o simulador Simics provê, através da extração do traço (*trace*) do código em tempo de execução. A análise de cada instrução é feita através da leitura de uma mkfifo onde as informações a respeito de cada instrução é disponibilizada. O seu conteúdo em formato de texto é analisado e o código binário da instrução decodificado, levando em consideração o modo de endereçamento, operadores, EFlags, etc. Basicamente, a decodificação de cada instrução é baseada na análise byte a byte do código binário, funcionando como uma máquina de estados. Na Figura 5.1 ilustra o funcionamento do simulador do TB de primeiro nível.

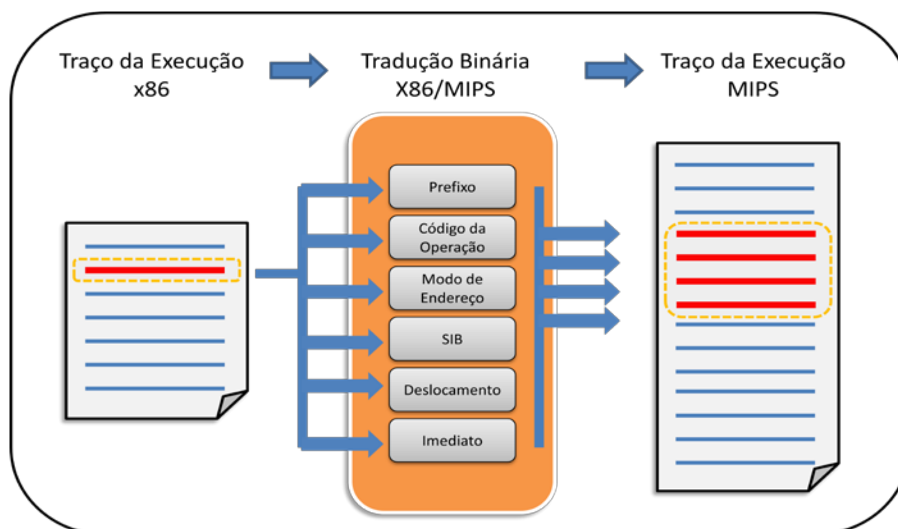


Figura 5.1: Tradução binária realizada pelo simulador.

Como resultado desta tradução, é gerado outra mkfifo, porém desta vez para ser consumida pelo simulador da arquitetura reconfigurável. Além disso, são fornecidas informações numéricas a respeito do comportamento do código executado, incluindo: número médio de instruções MIPS/x86, número de flags executadas e utilizadas, quantidade de instruções, etc.

Além da implementação do simulador em software, também foi desenvolvido um protótipo do tradutor em hardware. Sua organização foi descrita em linguagem VHDL com a finalidade de avaliar melhor o seu funcionamento ao atender as características comportamentais do circuito de tradução. Além disso, esta descrição foi interessante para avaliar interferência do sistema de tradução no caminho crítico do processador, onde nos testes avaliados a frequência de operação foi superior à frequência de operação do banco de registradores do MIPS (caminho crítico do processador)

5.1.1.3 Execução no DIM

Para a execução no DIM, foi utilizado a ferramenta de simulação ARISE (do inglês *Automatic Resources Investigation System based on application Execution*) que, de forma automática, explora a execução de uma aplicação e fornece o exato número de recursos necessários para acelerá-la, diminuindo a área ocupada e a potência consumida do sistema reconfigurável (RUTZIG, 2008).

Esta ferramenta realiza a simulação da execução de instruções MIPS na arquitetura reconfigurável (DIM) a partir de uma entrada de dados, onde neste trabalho, é fornecida pelo simulador do TB de primeiro nível. Este simulador, que funciona também com precisão de ciclo, fornece dados como: desempenho, energia, porcentagem de execução do código na Unidade Reconfigurável, etc. A grande vantagem da sua utilização é a possibilidade de modificar o hardware rapidamente e explorar o seu comportamento de forma precisa.

5.1.2 Ferramentas e Configuração de Simulação

Neste trabalho foi definida uma configuração ideal para testes, objetivando apenas um estudo de caso para exploração de possibilidades futuras no uso da tradução binária de dois níveis. Em todos os testes realizados foi utilizado o processador MIPS R3000 com memória cache de instruções e dados unificados. A unidade reconfigurável possui 48 colunas e 16 linhas; cada coluna possui 8 ULAs, 6 unidades de leitura/escrita e 2 multiplicadores. A TCache utilizada tem capacidade para alocar 512 configurações. Segundo (BECK, RUTZIG, *et al.*, 2008), esta configuração do DIM mostrou ser a melhor razão custo/benefício em termos de área de ocupação do sistema em um chip e ganhos de desempenho.

Os algoritmos executados nos testes pertencem ao conjunto Mibench (GUTHAUS, RINGENBERG, *et al.*, 2001) que disponibiliza algoritmos de código aberto para aplicação no domínio de sistemas embarcados. Como citado anteriormente na seção 5.1.1.1, todos os aplicativos foram executados em uma plataforma virtual (Simics), em paralelo com um sistema operacional Linux. Em todos os casos as aplicações fora compiladas utilizando o compilador GCC, com as bibliotecas ligadas de forma estática e o código otimizado através com a opção "-O3" (maior nível de otimização possível oferecido pelo GCC atualmente).

Para a avaliação de área foi utilizado o software Mentor Leonardo Spectrum (SPECTRUM, 2011) com a biblioteca TSMC 90nm e versões em VHDL do processador MIPS (MINIMIPS, 2008), da arquitetura reconfigurável e o protótipo do TB de primeiro nível. Nenhum dos componentes em hardware estão no caminho crítico do processador MIPS, no qual opera na freqüência de 600MHz.

5.2 Resultados

Inicialmente foram apenas feitas considerações a respeito do código compilado para os conjuntos de instrução x86 e MIPS. Analisando a taxa de ocupação da memória de instruções, na média, considerando todo o conjunto de *benchmarks*, o compilador GCC para MIPS gerou 26,69% mais código que o mesmo algoritmo compilado para o conjunto de instruções x86. Além disso como pode ser observado na Tabela 5, o processador MIPS executa em média, 36,63% mais instruções que um processador x86 executaria, considerando a execução do mesmo algoritmo.

Tabela 5: Número de instruções executadas nos dois conjuntos de instruções.

Benchmark	MIPS	x86
String Search	279,725	199,362
Sha	15,976,677	12,274,689
Bitcount	59,810,191	41,334,546
Qsort	51,695,224	27,386,935
Gsme	30,578,227	16,975,259
Gsmd	13,896,515	11,038,642

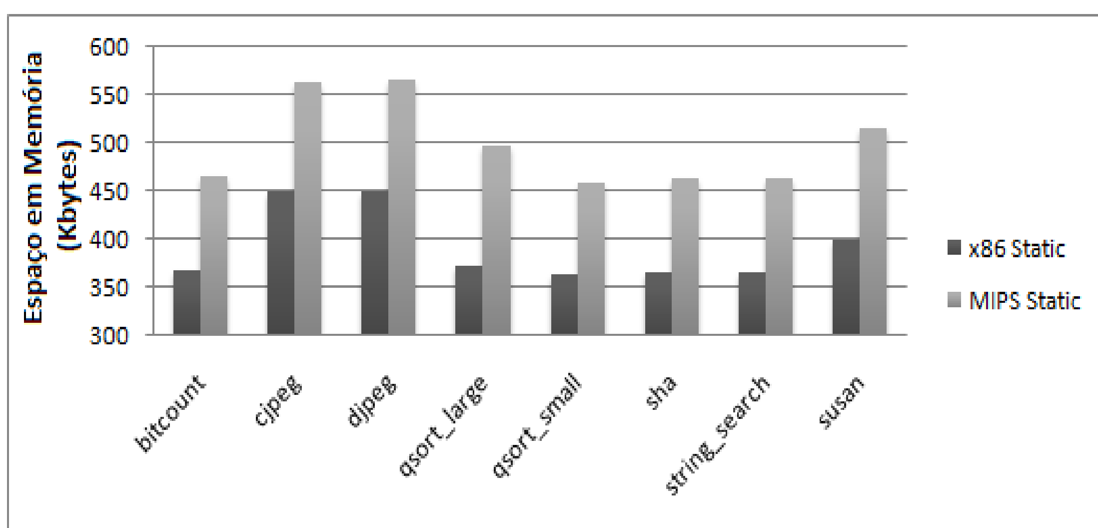


Figura 5.2: Total de espaço utilizado para alocar os algoritmos na memória, compilados pelo GCC com "-static".

Conforme o já explicado, o processador MIPS foi estendido com modificações básicas na sua organização e componentes extras de hardware para suportar novas instruções que facilitarão o processo de tradução binária. Entretanto, mesmo com estas modificações, ainda são geradas, em média, mais de uma instrução MIPS para cada instrução x86. Para ilustrar melhor a problemática e a melhoria do sistema com as modificações no processador, a Figura 5.3 exibe o número médio de instruções MIPS geradas para cada instrução x86, considerando cada algoritmo separadamente, em três casos:

Hardware original – Neste caso foi utilizado um processador MIPS comum, ou seja, sem nenhum tipo de alteração na sua organização e arquitetura;

Suporte para EFlags – Com o suporte à EFlags em hardware, não precisará ser gerado em software instruções para os cálculos das *flags* durante a tradução;

MIPS Estendido – Neste caso foi utilizado o recurso de suporte em hardware para EFlags juntamente com as outras extensões do processador, como já explicado anteriormente.

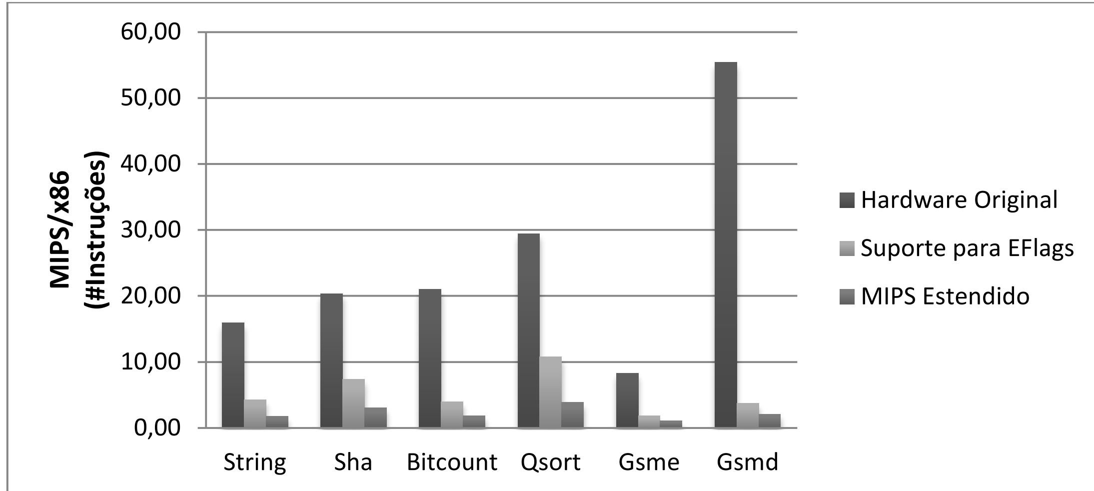


Figura 5.3: Comparativo entre o número médio de instruções MIPS geradas a cada instrução x86.

As Figura 5.2, Figura 5.3 e a Tabela 5, demonstram claramente a diferença em relação ao número de instruções executadas entre as arquiteturas (x86 e MIPS) e ilustram o alto custo que a tradução binária de primeiro nível provê. Este custo reflete na queda de desempenho na execução dos algoritmos e também no custo em termos de energia, devido o aumento do tempo de computação sem a redução da frequência, deixando claro os desafios que o sistema deve superar.

5.2.1 Desempenho

No gráfico da Figura 5.4, que demonstra o comportamento do sistema em termos de desempenho, foram realizados testes executando todos os algoritmos considerando quatro cenários diferentes, demonstrados na Tabela 6.

Tabela 6: Configurações utilizadas para as simulações de desempenho.

Configuração	Compilação	Execução	Observações
MIPS	MIPS	MIPS	Execução de código nativo MIPS em um processador MIPS padrão.
MIPS + DIM	MIPS	MIPS	Execução de código nativo MIPS utilizando um processador MIPS padrão juntamente com a arquitetura reconfigurável (DIM). Neste caso somente o TB de segundo nível é utilizado.
TB + MIPS	x86	MIPS++	Execução de código nativo x86 através do TB de primeiro nível e o processador MIPS estendido. Neste caso não existe otimização de código pela unidade reconfigurável.
TB + MIPS + DIM	x86	MIPS++	Execução de código nativo x86 utilizando a solução proposta por este trabalho: Tradução binária de dois níveis (tradução e otimização).

Nos dois primeiros casos, o algoritmo foi compilado para o conjunto de instruções do MIPS. A execução do código nativo em um processador padrão MIPS foi normalizado como 100%. O segundo teste apresentado no gráfico (código MIPS nativo + DIM) apresenta um ganho de desempenho em mais de duas vezes na média de todos os algoritmos. Podem ser citados como exemplos o algoritmo *Sha*, que apresenta ganhos

de 3,43 vezes, o *Bitcount*, que apresentou ganhos de 2,42 vezes e o *GSM Encoder* que apresenta um desempenho de apenas 1,53 vezes, no qual é considerado o pior caso em termos de desempenho ao ser avaliado o subconjunto de algoritmos do Mibench. De forma similar, resultados semelhantes são encontrados utilizando outras arquiteturas reconfiguráveis (GOLDSTEIN, SCHMIT, *et al.*, 2000) (CLARK, KUDLUR, *et al.*, 2004).

Considerando o mesmo código C dos algoritmos citados anteriormente, mas compilados agora para o conjunto de instruções x86, mais dois testes foram feitos: o primeiro utilizando apenas o primeiro nível de tradução binária, sem a otimização do DIM e o segundo teste executando o código no sistema completo, com os dois níveis de tradução binária. Como era esperado, no primeiro caso houve uma queda no desempenho do sistema devido ao custo da inserção do mecanismo de tradução binária. No algoritmo *GSM Decoder*, houve uma queda no desempenho de aproximadamente 2 vezes em relação a execução nativa do mesmo algoritmo compilado para um processador MIPS padrão. Entretanto, no segundo teste de execução com o código compilado para x86 (quarto cenário da tabela), o sistema completo (proposta deste trabalho) executou o código com melhor desempenho quando comparada a execução do código MIPS em um processador padrão, amortizando totalmente os custos em termos de desempenho causados pelo TB de primeiro nível. Os ganhos de desempenho (apresentados na Figura 5.4) em todos os benchmarks varia entre 1,11 e 1,96 vezes para o pior e melhor caso, respectivamente. Em média, os ganhos de desempenho foram de 45%.

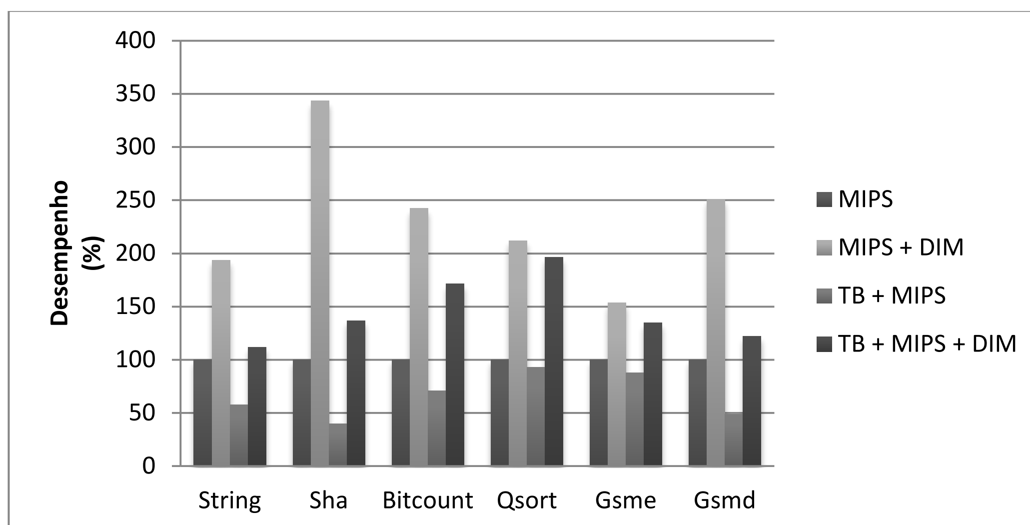


Figura 5.4: Gráfico de desempenho para quatro diferentes configurações.

Os resultados apresentados na Figura 5.4 podem ser considerados satisfatórios, tendo em vista que em sistemas onde somente o processo de virtualização (QEMU) é usada, sem a utilização do mecanismo de tradução binária (isto é, é utilizada uma máquina virtual que executa do mesmo conjunto de aplicativos do *host*), o a execução nativa das instruções x86 (CLARK, KUDLUR, *et al.*, 2004) é 4 vezes mais lenta. O processador Godson-3 (HU, WANG, *et al.*, 2009), ao realizar a tradução do código x86 para MIPS somente utilizando a interface do QEMU, apresenta em média uma queda no desempenho que chega a ser 6 vezes menor que a execução nativa MIPS. Entretanto, com um suporte para a tradução binária provido pelo hardware, o Godson-3 reduz esta queda de desempenho para apenas 1,42 vezes. É importante ressaltar que a intenção

deste trabalho não é realizar um comparativo direto com o processador Godson-3, já que este processador suporta todo o conjunto de instruções x86, incluindo interrupções e memória virtual. É possível visualizar que o processo de tradução binária de um conjunto CISC para um conjunto RISC possui um alto custo em termos de desempenho e que se torna inviável a sua utilização caso o sistema não possua nenhum tipo de otimização.

Além disso, pode-se fazer ainda um comparativo entre a execução do sistema proposto com uma execução em um processador nativo x86. Apesar de não ser o intuito deste trabalho (i.e. construir um sistema com desempenho superior a execução nativa de instruções x86), foi considerado que um processador nativo x86 qualquer executasse uma instrução complexa por ciclo. É possível visualizar na Figura 5.5, seu desempenho frente ao trabalho proposto. É de se observar que, dependendo do grau de paralelismo apresentado pela aplicação o sistema proposto consegue executar o aplicativo com um desempenho maior.

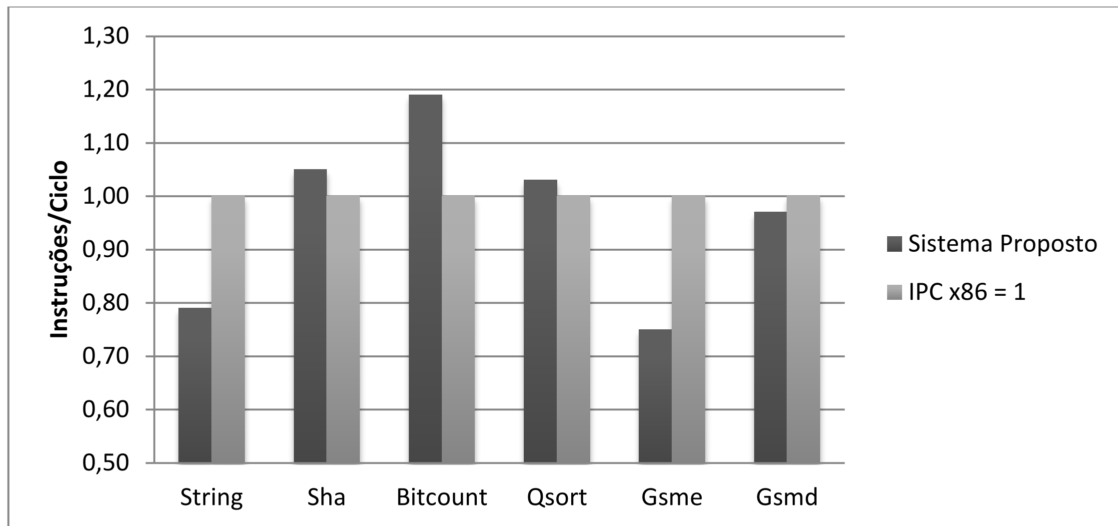


Figura 5.5: Comparativo do sistema proposto com um processador nativo da arquitetura x86 com IPC = 1.

5.2.2 Potência e Energia

Considerando o primeiro nível de TB, a Unidade de Tradução é responsável pela maior parte do consumo de potência. Durante a sua fase de execução ocorre a potência de pico devido ao momento de máximo chaveamento de transistores. Entretanto, a Unidade de Tradução não está ativa em todos os momentos. Ela entra em atividade somente quando uma instrução x86 é buscada na memória e é necessário gerar novas instruções equivalentes MIPS. Posteriormente, o código gerado é encaminhado à Unidade de Montagem, a qual leva um ou mais ciclos para disponibilizar as instruções ao processador. A sua execução será requisitada novamente somente no momento em que a Unidade de Controle permitir a busca uma nova instrução x86.

O consumo de potência considerando, o sistema completo (dois níveis de tradução binária) é de 78mW, enquanto a potencia de pico fica em torno de 1,7W. Na Figura 5.6, como pode ser observado, o primeiro nível tem um consumo um pouco maior que o processador MIPS, enquanto o segundo nível possui um consumo próximo ao do MIPS. O componente que mais consome potencia é a memória de dados, a tabela de PC

indexado e a Cache de contexto, utilizada para guardar as configurações da arquitetura reconfigurável.

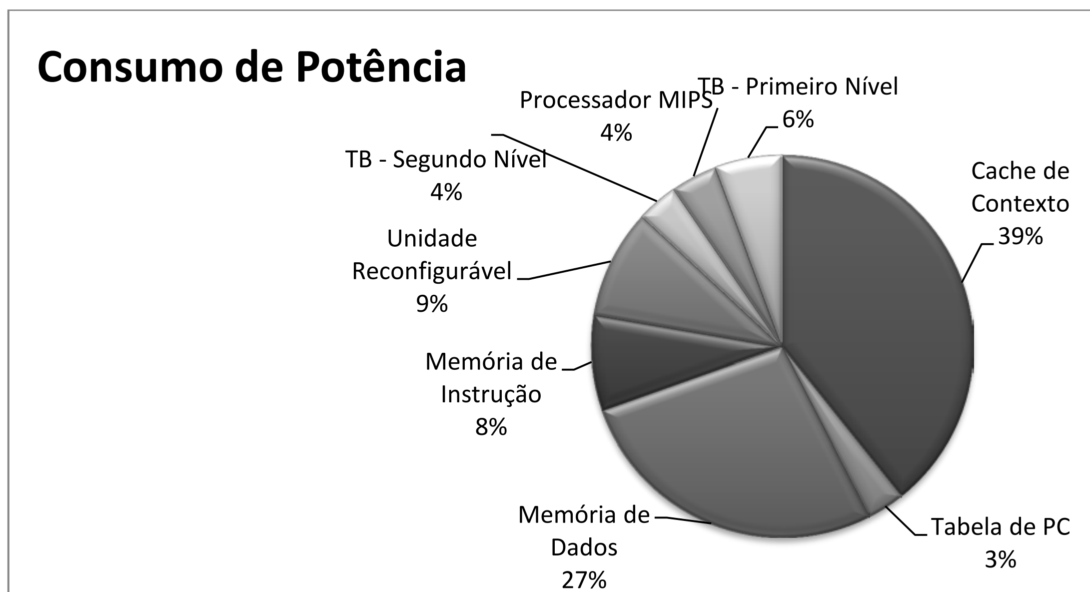


Figura 5.6: Gráfico demonstrativo do consumo de potência relativo em cada componente do sistema.

Na Figura 5.7 está ilustrado o consumo de energia do sistema, considerado: processador MIPS padrão, processador juntamente com a unidade reconfigurável (MIPS + DIM) e a proposta do trabalho com dois níveis de tradução (BT + MIPS estendido + DIM) executando o mesmo benchmark apresentado anteriormente. Note-se que o sistema proposto executando os algoritmos SHA, GSME e GSMD possui aproximadamente o mesmo consumo que o processador MIPS padrão. Apesar de o sistema completo ter um consumo maior por ciclo, o tempo de computação é reduzido, ou seja, menos ciclos são necessários para computar o mesmo algoritmo. Reduzindo-se assim, significativamente o consumo de energia do sistema.

A quantidade de energia economizada pelo sistema depende diretamente do quanto está sendo utilizado a Unidade Reconfigurável, que por sua vez, é dependente também do algoritmo a ser processado. Quanto maior o grau de paralelismo entre instruções de um programa, maior a otimização que a arquitetura reconfigurável irá prover e, consequentemente, amortizar os custos de energia. Isto se dá principalmente devido a três fatores:

- Embora o consumo de potência seja maior com a adição do mecanismo de tradução binária (primeiro nível) e o DIM, após a tradução de vários trechos do código com alta taxa de execução (*hotspots*) e que serão guardados já em forma otimizada na TCache, novas traduções não serão mais necessárias, (os dois mecanismos de tradução binária não serão utilizados) assim como a busca das instruções destes trechos na memória de instruções.
- Após um determinado trecho de código passar através dos dois níveis de tradução, no instante em que ocorre a repetição da sua execução, esta será feita de maneira muito mais eficiente, utilizando lógica combinacional da Unidade Reconfigurável, ao invés da lógica padrão de estágios de *pipeline* em um processador comum.

- A execução paralela de instruções na matriz lógica e a não necessidade de repetir análises de paralelismo repetidamente para o mesmo trecho de código.

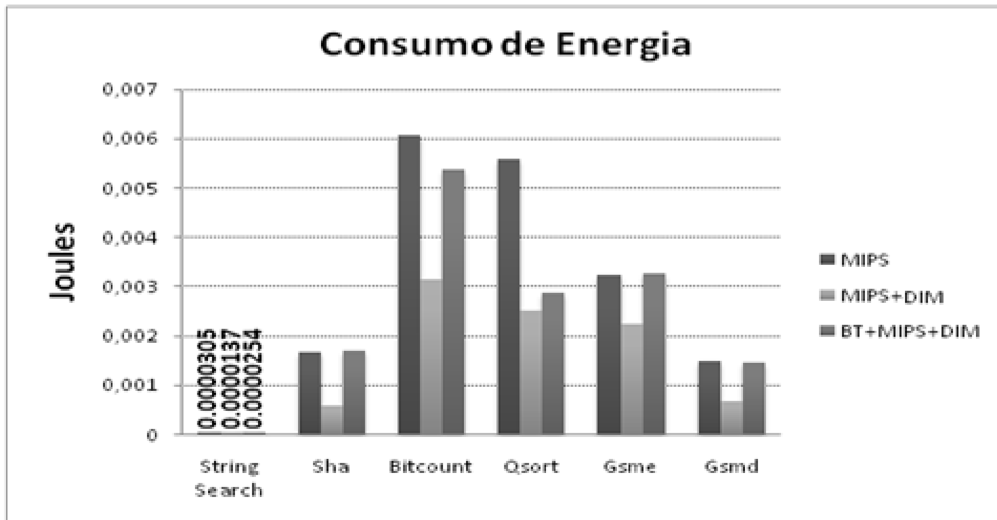


Figura 5.7: Consumo de energia em três sistemas diferentes.

5.2.3 Área

A Tabela 7 demonstra os valores de área de cada componente em número de portas lógicas (*gates*). Através desta tabela pode-se observar que o TB de primeiro nível corresponde a somente 2% da área total do sistema. Considerando que cada porta é composto por 4 transistores, o sistema completo teria um total de 4,87 milhões de transistores. Outra observação importante é em relação ao espaço de memória. Se compararmos o sistema proposto, que executa instruções x86, com um processador padrão MIPS, existe uma significante redução no espaço de memória de instruções (conforme demonstrado na Figura 5.2), o que amortizaria a área ocupada em hardware pelo sistema proposto.

Tabela 7: Custo de área de cada componente do sistema.

Unit	Área (Portas)	Porcentagem
First-Level BT	22.406	1,83%
MIPS R3000	26.866	2,2%
Second-Level BT	15.264	1,25%
Rec. Array	1.017.620	94,72%
Total	**1.219.535**	**100%**

Somente para se ter uma noção de área, como já foi comentado, o processador Godson-3 é composto por 4 núcleos superescalares MIPS R10000. De acordo com (YEAGER, 1996), cada MIPS R10000 possui uma área de aproximadamente 2,4 milhões de portas lógicas. Assumindo que cada porta possui 4 transistores (da mesma forma que anteriormente), seriam necessários 9,6 milhões de transistores para implementar do processador Godson-3, ocupando cerca de 2 vezes mais área que o sistema aqui proposto.

6 ANÁLISE CRÍTICA

6.1 Custos de implementação de todo o conjunto de instruções x86

O conjunto de instruções x86 é composto por aproximadamente 190 instruções (sem contabilizar as instruções de ponto flutuante). De acordo com os estudos realizados seriam necessários endereços adicionais para dar suporte ao conjunto x86 completo. Entretanto, os custos seriam limitados somente por este espaço extra em tabelas, pois este sistema já dá suporte a todos os sinais de *flags* e modos de endereçamento e o número de acessos nas tabelas continuariam os mesmos. Por consequência do aumento das tabelas, o acesso a esses dados iriam levar mais tempo para ser adquiridos. De acordo com os resultados em testes utilizando a linguagem VHDL, este tempo de acesso aos dados seria apenas 10% maior. Assim, a unidade de tradução ainda iria continuar sem afetar o caminho crítico do processador.

Além disso, a Unidade de Montagem iria continuar praticamente a mesma e termos de controle: somente o tamanho do buffer seria ampliado (tamanho suficiente para não gerar espaços vazios no *pipeline* do processador MIPS). O mesmo ocorrerá com a Unidade de PC: seu *buffer* deverá ser aumentado de 11 para 18 bytes (o tamanho máximo das instruções x86 é de 15 bytes). O PC deverá ser calculado de acordo com o número de bytes da instrução (esse dado é fornecido pela Unidade de Tradução) para gerar o endereço da próxima instrução, assim, este valor depende da instrução atual, onde o hardware que provê esta informação já esta implementado. Para o novo controle das instruções irá ocorrer o mesmo: com a adição das novas entradas na unidade de Controle, o hardware para o controle de fluxo das instruções já está implementado sendo necessário apenas a adição de estados extras em sua máquina de estados.

6.2 Suporte para outros conjuntos de instruções

Conforme o já explanado, cada instrução x86 pode ser formada por até 15 bytes. Estas instruções possuem diversas formas de endereçamento e *flags* que devem ser atualizadas a cada instrução executada. Além disso, o conjunto x86 possui instruções muito complexas no qual podem levar de um ciclo para finalizar a sua execução. Por outro lado, conjuntos de instruções PowerPC, Sparc V8 e ARM, são significativamente menos complexas: estes conjuntos implementam poucas ou nenhum tipo de *flag* de estado; o tamanho de suas instruções possuem tamanho fixo ou pouca variação (no caso das instruções normais e *thumbs* no processador ARM); e existem menos modos de endereçamento que o conjunto de instruções x86.

Estes conjuntos são tipicamente máquinas RISC (apesar de alguns conjuntos estarem se tornando cada vez mais complexos). Desta forma, o conjunto de instruções x86 pode ser considerado de alta complexidade.

Por consequência da alta complexidade, a implementação de um TB de primeiro nível capaz de traduzir outro conjunto de instruções provavelmente será mais simples. Em termos de implementação a estrutura básica em hardware do TB de primeiro nível não necessita mudar. Os quatro componentes em hardware poderiam ser facilmente adaptados e suas interconexões iriam permanecer praticamente as mesmas. Considerando os conjuntos de instruções anteriormente mencionados, a Unidade de PC também seria mais simples. A implementação do TB para máquinas RISC, como o tamanho da instrução é fixo, permite com que um valor fixo seja somado ao endereço atual. Além disso instruções de desvio poderiam ser calculadas praticamente da mesma forma com que já eram calculadas anteriormente. Entretanto, no lugar de flags, os valores dos registradores (normalmente resultados de instruções como *slt*) seriam utilizados.

Como o processador MIPS também é uma máquina RISC, a maioria das instruções dos processadores PowerPC e Sparc V8 poderiam ser traduzidas diretamente em apenas uma instrução MIPS. Consequentemente, as tabelas da Unidade de Tradução iriam ser reduzidas e por consequência, também se reduziriam o número de registradores que formam o buffer na Unidade de PC e o tamanho da fila na Unidade de Montagem. Por fim, devido o número diminuído de modos de endereçamentos, a Unidade de Controle teria seus estados da *FSM* reduzidos significativamente. Já a arquitetura ARM, por ela ser mais complexa que as citadas anteriormente, seria um pouco mais complicada a sua implementação, mas ainda continuaria a ser mais simples que a implementação deste trabalho (conjunto de instruções x86). O processador ARM suporta predição e instruções do tipo Thumb (um formato de instruções compactas para economia de espaço em memória. Neste caso, um multiplexador poderia ser utilizado para escolher qual o conjunto de instruções ARM a ser empregado (original ou modo thumb), de acordo com o valor de uma flag salva em um registrador de estados no qual é utilizado na atual arquitetura ARM.

6.3 Estendendo o segundo nível de tradução

Neste trabalho foi apresentado um caso de estudo onde o TB de segundo nível é particularmente usado para uma arquitetura reconfigurável específica. O segundo nível de tradução é fortemente acoplado ao processador MIPS, desse modo, se o usuário optar por utilizar uma arquitetura alvo diferente, outra implementação deverá ser feita. Nesta subseção será discutida outras técnicas existentes que poderiam ser utilizadas como segundo nível de tradução. Deve ser levado em conta neste trabalho que devido o uso do conjunto de instruções MIPS como código intermediário, qualquer adaptação seria simplificada já que a maioria dos mecanismos de otimização existentes se utiliza do código de arquiteturas RISC. Considerando somente a arquitetura reconfigurável como o hardware de otimização, três arquiteturas poderiam ser adaptadas e inseridas neste sistema como segundo nível de tradução.

A primeira arquitetura a ser considerada é a *Warp Processing System* (VAHID, STITT e LYSECKY, 2001) (LYSECKY, 2006). Esta arquitetura é baseado em um SOC (System On-Chip) composto por um microprocessador para executar a aplicação do software, outro microprocessador onde um algoritmo CAD é executado, memória local e um FPGA dedicado. Primeiramente, o microprocessador executa o código binário

original enquanto um sistema de monitoramento detecta as regiões críticas. Na sequência, o software CAD gera a partir da aplicação um grafo de controle de fluxo e após uma síntese destes dados, é realizado um mapeamento deste circuito dentro de uma estrutura FPGA simplificada. A partir de código binário é gerado blocos de hardware. Neste caso, o software de CAD poderia ser utilizado como o TB de segundo nível, otimizando o código já traduzido para o conjunto de instruções intermediárias MIPS vindas do primeiro nível de tradução e otimizando-o para ser executado em hardware no FPGA. No trabalho original, um processador PowerPc e um processador ARM são utilizados como caso de estudo na arquitetura nativa. Desta forma, o software de CAD deveria ser adaptado para interpretar instruções MIPS.

O sistema CCA (*Configurable Compute Array*) (CLARK, KUDLUR, *et al.*, 2004) também pode ser uma alternativa para o segundo nível de tradução. O CCA é baseado em um array de grão grosso, composto por unidades funcionais simples e fortemente acoplado em um processador ARM. O funcionamento do sistema ocorre em dois passos: primeiramente é verificado quais são os trechos de código a serem executados no CCA e na sequência estes trechos são substituídos por micro operações. Neste trabalho duas abordagens são apresentadas: uma estática e outra dinâmica. A análise estática realiza a busca dos trechos de código em tempo de compilação, já na dinâmica, este processo é realizado em tempo de execução. É de se considerar, que somente a forma dinâmica poderia ser utilizada como segundo nível de tradução binária, assumindo o uso de uma memória cache para alocar as instruções já traduzidas e permitir a busca de trechos de código para serem substituídos por micro operações. Este sistema utiliza uma técnica muito complexa de análise de grafos baseada em rePlay (PATEL e LUMETTA, 2001).

Outra possibilidade é o processador RISPP (BAUER, SHAFIQUE e HENKEL, 2008). Nesta arquitetura, os trechos de instruções (átomos) são sintetizados em tempo de compilação. Em tempo de execução é decidido quais os átomos que serão alocados em partes (containers) de um FPGA e combinadas para compor moléculas. Uma ou mais moléculas possuem diferentes implementações de uma SI (*Special Instruction*), no qual irá substituir parte do programa no momento da execução. Esta técnica utiliza um FPGA acoplado em um processador MIPS ou um SparcV8. Entretanto, o processador RISPP não é uma técnica totalmente transparente (o código deve ser alterado antes da execução).

Outras técnicas baseadas em repetição de instruções também podem ser adaptadas como segundo nível de tradução binária, ou seja, mantendo a ideia de que as instruções com os mesmos operandos sejam repetidos por um grande número de vezes durante a execução de um programa.

6.4 Desempenho de outros sistemas de tradução binária

Tendo em vista a dificuldade no acesso à outros sistemas de tradução binária em executar novamente os algoritmos devido a falta de informações ao seu respeito e a falta de acesso ao seus códigos fonte, neste tópico será realizada uma breve discussão a respeito do comportamento dos sistemas citados no tópico 2.2 (Trabalhos Correlatos). Entretanto, grande parte dos sistemas mencionados anteriormente não apresentam dados a respeito de área, memória, potência e consumo de energia, desta formar restringindo a discussão a respeito de medidas de desempenho.

A arquitetura Daisy (DAISY e EBCIOGLU, 1996), no qual traduz instruções PowerPC para serem executadas em um processador VLIW gera em média uma instrução VLIW a cada 2,5 instruções PowerPC (no total podem ser incluídas até 8 instruções PowerPC em uma instrução VLIW), considerando a execução do SPECInt95. O mesmo trabalho também mostra que são necessárias até 4000 instruções para traduzir apenas uma instrução para a arquitetura alvo. Entretanto este custo é reduzido devido a alta taxa de reutilização do código traduzido.

O Dynamo (BALA, 2000) não traduz o código binário para ser executado em outra arquitetura, seu propósito maior é otimizar a execução para torná-la mais eficiente (segundo o autor este sistema realizada uma otimização de nativa para nativa). O benchmark utilizado inclui o SPECInt95 e uma aplicação comercial chamada Deltablue. As medidas de desempenho são baseadas em tempo de computação, considerando a arquitetura do HP PA-8000. O sistema Dynamo alcança consideráveis ganhos de desempenho em alguns casos (até 22%) e na média 9%, quando a otimização no código binário é compilado com nível +O2 (compilador específico da HP). Este sistema de otimização consegue atingir ganhos até mesmo quando o código é compilado com o nível +O4 ao executar os códigos anteriormente mencionados.

Outro sistema de tradução é o FX!32 (CHERNOFF, HERDEG, *et al.*, 1998), quando executa aplicações x86 traduzidas (algoritmos do benchmark BYTEmark) no processador Alpha 21164A a 500MHz, atingindo 4 pontos, se torna aproximadamente duas vezes mais rápido do que um Pentium Pro a 200MHz, rodando a mesma aplicação nativa (neste caso sem a tradução), para aplicações com operadores inteiros. Entretanto, é mais lenta quando é considerado aplicações com ponto flutuante (o Pentium chega a 3,2 pontos, enquanto o processador Alpha chega somente a 2,1). Para se ter uma noção de custos em torno do processo de tradução, utilizando a mesma aplicação compilada de forma nativa para ser executada no mesmo processador Alpha, chega-se de 6 a quase 8 pontos considerando aplicações com valores inteiros e de ponto flutuante respectivamente.

Analisando o Godson3, somente o processo de virtualização (se o mecanismo de tradução binária) da maquina virtual Qemu é 4 vezes mais lenta do que a execução nativa de instruções x86 (BELLARD, 2005). Devido a este custo o processador Godson3, quando traduz o código de x86 para MIPS usando o Qemu e sem suporte do hardware para a tradução, a execução é na média 6 vezes mais lenta que a execução do mesmo código compilado para o processador MIPS nativo (HU, WANG, *et al.*, 2009). Com o suporte do hardware para o mecanismo de tradução, o Godson3 reduz este custo para apenas 1,42 vezes mais lento que a execução do código nativo MIPS. Entretanto, este trabalho não pode ser diretamente comparado com o Godson3, já que este processador executa todo o conjunto de instruções x86, incluindo interrupções e memória virtual.

Este tópico demonstra as dificuldades e os altos custo para se realizar o processo de tradução binária. Além disso, uma das grandes vantagens deste trabalho quando se realizada um comparativo com os demais trabalhos comentados anteriormente, é a possibilidade de ser transparente ao usuário final e atingir significativos ganhos de desempenho, onde o primeiro nível de tradução binária apresenta custos das mesma forma como é apresentado pelos outros sistemas, porém, a segunda camada de tradução binária é capaz de abstrair os custos da primeira camada e ainda apresentar ganhos de desempenho.

7 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi demonstrado um sistema de processamento composto por dois níveis de tradução binária, através de um estudo de caso onde aplicativos compilados de forma nativa para a arquitetura x86 (CISC) foram executados em um processador com uma arquitetura MIPS (RISC). Os níveis de tradução binária foram implementados em hardware e de forma totalmente transparente ao usuário final. Ao ser executado um aplicativo compilado para a arquitetura x86 na arquitetura proposta, a sua execução ocorrerá de forma imperceptível ao usuário. Ressaltando-se que o mesmo acontecerá com o desenvolvedor. Aplicativos poderão ser desenvolvidos diretamente para a arquitetura x86 e da mesma forma serão executados no sistema proposto neste trabalho.

O fator motivante deste trabalho foi a possibilidade de se manter a compatibilidade binária entre dois processadores completamente diferentes, consequentemente aumentando a gama de aplicativos a serem reutilizados na mesma organização. Além disso, o sistema deveria ser competitivo ao ponto de ser comparado com a execução nativa de um código em uma arquitetura nativa e atingisse desempenho semelhante ou ainda com poucos custos.

Com o auxílio do simulador do primeiro nível de tradução, juntamente com o simulador já existente de segundo nível e a unidade reconfigurável, foi realizado o comparativo de execução. O sistema de maneira geral apresentou ganhos de desempenho de 45% em relação à execução nativa. Além disso, foi realizado ainda o comparativo em termos energia, onde foi obtido praticamente o mesmo consumo de energia para quatro aplicativos (String Search, Sha, Gsme e Gsmd). Para a execução de dois aplicativos (Bitcount e Qsort) o sistema proposto teve um consumo menor que a execução nativa. Constatou-se que essas diferenças energéticas se devem ao grau de dependência entre instruções. Aplicativos mais passíveis à execução paralela de instruções, fazem com que seja melhorada a eficiência da execução da arquitetura reconfigurável na qual explora principalmente esta característica em um código. Podemos observar na Figura 5.7 que os dois aplicativos citados anteriormente apresentaram o melhor desempenho entre os algoritmos testados.

Apesar de os dados de desempenho serem satisfatórios e os dados de energia condizentes com o formato do sistema, houve um custo considerável em área. A adição de hardware para auxiliar o processo computacional. Entretanto, o sistema não interferiu no caminho crítico do processador, sendo possível manter a mesma frequência de 600MHz do processador MIPS R3000 na tecnologia TSMC 90nm do sistema original.

7.1 Trabalhos Futuros

A apresentação desta pesquisa foi apenas um estudo de caso, podendo desta forma, originar a partir dela, outros caminhos e novas linhas de pesquisas. O estudo de caso contido neste trabalho ainda pode ser melhorado, adicionando e dando suporte a novas instruções como:

- Adicionar suporte a instruções de ponto flutuante;
- Permitir a execução de instruções do tipo SIMD (do inglês *Single Instruction Multiple Data*);
- Implementar o suporte à interrupções.

Além de melhorias no sistema aqui discutido, pode-se também realizar mudanças mais significativas no hardware como as propostas apresentadas nos próximos tópicos.

7.1.1 Processador Multi ISA

A tradução binária de dois níveis mostrou-se muito eficiente através do estudo de caso realizado neste trabalho. A possibilidade de executar código nativo x86 em uma arquitetura completamente diferente e de forma eficiente em termos de energia e desempenho, mostra também uma grande possibilidade de reaproveitamento de código de diversas arquiteturas em um único processador.

O sistema proposto é capaz de executar tanto aplicativos x86 quanto aplicativos MIPS. No entanto, através do desenvolvimento de variações no primeiro nível de tradução binária, é possível a implementação de um mecanismo onde um único processador seria capaz de executar diversas outras arquiteturas. Na Figura 7.1 é ilustrado uma possível implementação dos sistema de tradução binária em paralelo, possibilitando a compatibilidade binária com diversas arquiteturas.

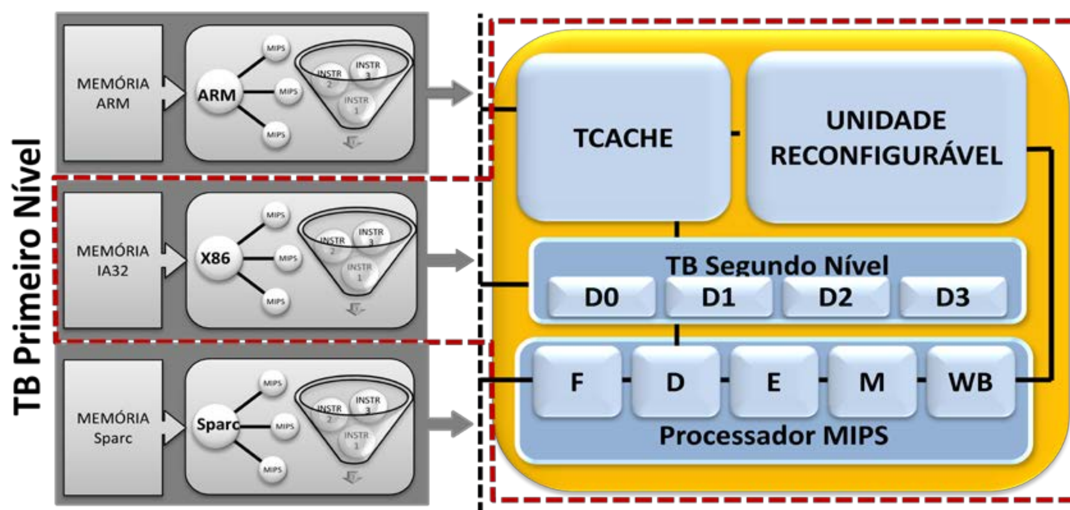


Figura 7.1: Processador capaz de executar diversos conjuntos de instruções.

Esta mudança de arquitetura poderia ocorrer a nível de leiaute da arquitetura antes da fabricação, definindo neste momento qual será a arquitetura utilizada, ou ainda, ocorrer em tempo de execução, onde vários mecanismos de tradução (primeiro nível) estão implementados.

7.1.2 Estudo do sistema utilizando multi-processadores

Outra tendência é a utilização vários processadores em um único chip. Esta tendência anteriormente era vista somente em sistemas de processamento de propósito geral, hoje este está sendo observada no mercado de sistemas embarcados. Da mesma forma que funciona hoje um processador com diversos núcleos, poderia existir diversos núcleos com um sistema de tradução binária. Desta forma além de explorar paralelismo a nível de instrução (ILP), pode-se atingir um ganho maior explorando paralelismo a nível de *threads* (TLP) como pode ser visto na Figura 7.2.

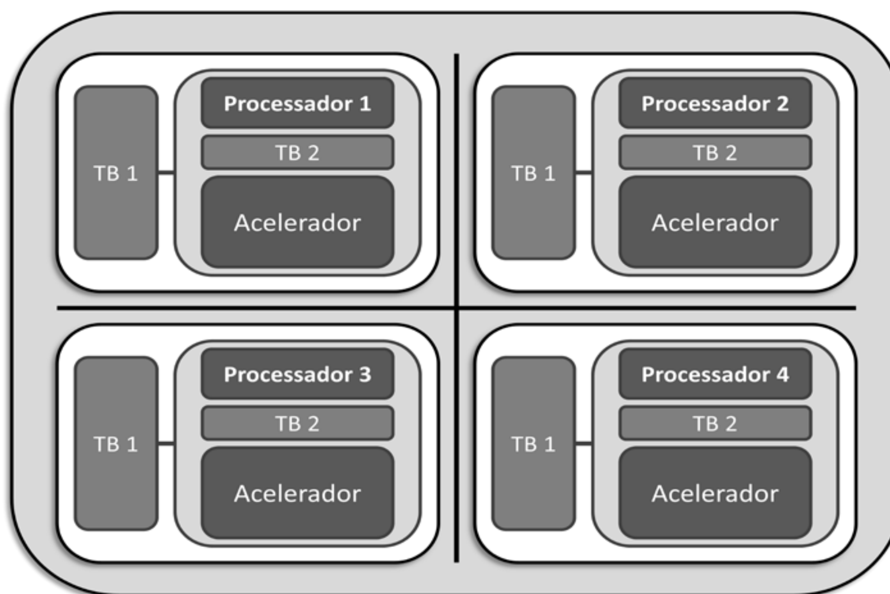


Figura 7.2: Utilização de diversos núcleos em um sistema de processamento.

Além disso, pode-se pensar em multi-processadores heterogêneos, onde apenas um conjunto de instruções é processado. Neste caso, o sistema de tradução binária poderia selecionar qual o melhor processador para enviar as instruções conforme o comportamento do código e tipo de instrução.

REFERÊNCIAS

ALTMAN, E. R. et al. Advances and future challenges in binary translation and optimization. **Proceedings of the IEEE**, v. 89, n. 11, p. 1710-1722, 2001.

ALTMAN, E. R.; KAELI, D.; SHEFFER, Y. Welcome to the opportunities of binary translation. **Computer**, v. 33, n. 3, p. 40-45, 2000.

APPLE, 2011. Disponivel em: <http://www.apple.com/asia/rosetta/>. Acesso em: 17 abril 2011.

ARM, 2011. Disponivel em: <www.arm.com>. Acesso em: 12 março 2011.

BALA, V. Dynamo (Panel Session): a transparent, dynamic, native binary optimizer. **SIGPLAN Not.**, v. 35, p. 75--, 2000. Chairman-Cytron, Ron.

BALA, V.; DUESTERWALD, E.; BANERJIA, S. Dynamo: a transparent dynamic optimization system. **SIGPLAN Not.**, v. 35, p. 1-12, 2000.

BECK, A. C. S. . L. C. **Dynamic Reconfigurable Architecture and Transparent Optimization Techniques**. 1st Edition. ed. [S.l.]: Springer, 2010.

BECK, A. C. S. et al. **Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications**. [S.l.]: [s.n.]. 2008. p. 1208-1213.

BECK., A. C. S.; CARRO, L. **Dynamic reconfiguration with binary translation:** breaking the ILP barrier with software compatibility. [S.l.]: [s.n.]. 2005. p. 732-737.

BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. **USENIX Annual Technical Conference, Freenix Track**, Anaheim, CA, Abril 2005. 10-15.

CHERNOFF, A. et al. FX!32 a profile-directed binary translator. **IEEE MICRO**, v. 18, n. 2, p. 56-64, 1998.

CLARK, N. et al. **Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization**. [S.l.]: [s.n.]. 2004. p. 30-40.

COMPTON, K.; HAUCK, S. Reconfigurable Computing: A survey of systems and software. **In ACM Computing Surveys**, v. 34, n. 2, p. 171-210, June 2002.

DEHNERT, J. C. et al. **The Transmeta Code Morphing™ Software:** using speculation, recovery, and adaptive retranslation to address real-life challenges. [S.l.]: [s.n.]. 2003. p. 15-24.

EBCIOGLU, K.; ALTMAN, E. R. **Daisy:** Dynamic Compilation For 100 Architectural Compatibility. [S.l.]: [s.n.]. 1997. p. 26-37.

FAN, B. et al. **The implementation and design methodology of a quad-core version Godson-3 microprocessor**. [S.l.]: [s.n.]. 2009. p. 1167-1170.

FLYNN, M. J.; HUNG, P. Microprocessor design issues: thoughts on the road ahead. **IEEE MICRO**, v. 25, n. 3, p. 16-31, 2005.

GOLDSTEIN, S. C. et al. PipeRench: a reconfigurable architecture and compiler. **Computer**, v. 33, n. 4, p. 70-77, 2000.

GUTHAUS, M. R. et al. **MiBench:** A free, commercially representative embedded benchmark suite. [S.l.]: [s.n.]. 2001. p. 3-14.

HAUCK, S. et al. The Chimaera reconfigurable functional unit. **In Proc. IEEE Symp. FPGAs For Custom Computing Machines,** Napa VAlley, CA, p. 87-96, 1997.

HAUSER, J. R.; WAWRZYNEK, J. Garp: A MIPS Processor with a Reconfigurable Coprocessor. **In: FPGA-Based Custom Computing Machines, Napa Valley. Proceedings.**, Washington: IEEE Computer Society., 1997. 12-21.

HENKEL, J.; ERNST, R. **A Hardware/software Partitioned Using A Dynamically Determined Granularity**. [S.l.]: [s.n.]. 1997. p. 691-696.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture - A Quantitative Approach (4. ed.)**. [S.l.]: Morgan Kaufmann, 2007.

HINTON, G. et al. The Microarchitecture of the Pentium® 4 Processor. **Intel Technology Journal**, v. 1, p. 2001, 2001.

HOOKWAY, R. **DIGITAL FX!32 running 32-Bit x86 applications on Alpha NT**. [S.l.]: [s.n.]. 1997. p. 37-42.

HU, W. et al. Godson-3: A Scalable Multicore RISC Processor with x86 Emulation. **IEEE MICRO**, v. 29, n. 2, p. 17-29, 2009.

INTEL, 2011. Disponivel em: <www.intel.com>. Acesso em: 12 março 2011.

KIM, N. S. et al. Leakage current: Moore's law meets static power. **Computer**, v. 36, n. 12, p. 68-75, 2003.

KIRK, C. A. Binary Translation. **Communications of the ACM**, v. 36, n. 2, p. 69-81, 1993.

KLAIBER, A. The Technology Behind Crusoe Processors. **Transmeta Corporation**, January 2000. Disponivel em: <http://www.transmeta.com>.

MAGNUSSON, P. S. et al. Simics: A full system simulation platform. **Computer**, v. 35, n. 2, p. 50-58, 2002.

MINIMIPS. **VHDL Opencores**, 2008. Disponivel em: <http://www.opencores.org>. Acesso em: 2008.

RAZDAN, R.; SMITH, M. D. A High-performance Microarchitecture with Hardware-programmable Functional Units. **MICRO 27 Proceedings of the 27th Annual International Symposium on Microarchitecture**, San Jose, California. ACM, New York, NY, 1994. 172-180.

RUTZIG, M. B.; S., A. C.; CARRO, L. **Balancing reconfigurable data path resources according to application requirements**. [S.l.]: [s.n.]. 2008. p. 1-8.

SANKARALINGAM, K. et al. Trips: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP. **ACM Transactions on Architecture an Code Optimization**, New York, v. 1, n. 1, Maio 2004. 62-93.

SIMA, D. Decisive aspects in the evolution of microprocessors. **Proceedings of the IEEE**, v. 92, n. 12, p. 1896-1926, 2004.

SPECTRUM, L., 2011. Disponivel em: <http://www.mentor.com>. Acesso em: Outubro 2010.

SRIDHAR, S.; SHAPIRO, J. S.; BUNGALE, P. P. HDTrans: a low-overhead dynamic translator. **SIGARCH Comput. Archit. News**, v. 35, p. 135-140, 2007.

STITT, G.; VAHID, F. The Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic. **IEEE Design an Test of Computers**, Los Alamitos, v. 19, n. 6, November 2002. 36-43.

SWANSON, S. et all, The WaveScalar Architecture. **ACM Transactions on Computer Systems, New York, v. 25, n. 2**, Maio 2007.

VASSILIADIS, S. **The Hipeac Embedded Systems**. [S.l.]: Apresentação oral no Hipeac Compilation Architecture, 2006.

VASSILIADIS, S. et al. The MOLEN polymorphic processor. **Transactions on Computers**, v. 53, n. 11, p. 1363-1375, 2004.

VENKATARAMANI, G. et al. A Compiler Framework for Mapping Applications to a Coarse-Grained Reconfigurable Computer Architecture. **International Conference on Compilers, Architecture and Synthesis for Embedded Systems**, Atlanta. Proceedings. New York: ACM Press, 2001. 116-125.

YANG, B.-S. et al. **LaTTe:** a Java VM just-in-time compiler with fast and efficient register allocation. [S.l.]: [s.n.]. 1999. p. 128-138.

YEAGER, K. C. The Mips R10000 superscalar microprocessor. **IEEE MICRO**, v. 16, n. 2, p. 28-41, 1996.