UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

José Otávio Carlomagno Filho

Escalonamento Redirecionável de Código sob Restrições de Tempo Real

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Luiz Cláudio Villar dos Santos, Dr.

Florianópolis, março de 2007

Escalonamento Redirecionável de Código sob Restrições de Tempo Real

José Otávio Carlomagno Filho

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração completa e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Rogério Cid Bastos, Dr.

Banca Examinadora

Luiz Cláudio Villar dos Santos, Dr.

Olinto José Varela Furtado, Dr.

Luís Fernando Friedrich, Dr.

Rômulo Silva de Oliveira, Dr.

"If I have seen farther,
it is by standing on the shoulders of giants."
Isaac Newton

Aos meus pais, Doralysa e José Otávio, e à Marina.

Agradecimentos

Agradeço primeiramente a meus pais, Doralysa e José Otávio, a quem devo tudo que sou hoje. À minha namorada, Marina, pelo companheirismo, apoio e incentivo constantes.

Ao professor Luiz Cláudio pelo excelente trabalho de orientação e pelas oportunidades proporcionadas desde a graduação.

Aos colegas do LAPS que contribuíram direta ou indiretamente neste trabalho. Ao Alexandro Baldassin e Sandro Riggo, da UNICAMP, pelo suporte técnico.

Agradeço as contribuições de todos os membros da banca e por terem aceitado o convite.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Acrônimos

ADL	Architecture Description Language
ASIP	Application-Specific Instruction Set Processor
BB	Bloco Básico
CFG	Control Flow Graph
CI	Circuito Integrado
CPU	Central Processing Unit
DFG	Data Flow Graph
DSP	Digital Signal Processor
IP	Intellectual Property
ISA	Instruction Set Architecture
NRE	Non-recurring Engineering
PV	Programmer's View
PVT	Programmer's View plus Timing
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RT	Register-Transfer
RTL	Register-Transfer Level
SoC	System-on-Chip
SWPG	Scheduled Weighted Precedence Graph
TLM	Transaction Level Modeling
VLIW	Very Long Instruction Word

WPG Weighted Precedence Graph

Resumo

A evolução dos sistemas computacionais deu origem aos *systems-on-chip* ou SoCs, onde diversos componentes (como memória, barramentos e processador(es)) estão presentes em um único circuito integrado. Os SoCs possivelmente contêm múltiplos processadores de diferentes tipos, portanto a exploração de seu espaço de projeto requer ferramentas redirecionáveis. O aumento da complexidade de tais sistemas, juntamente com a diminuição do *time-to-market* e a necessidade de iniciar-se o desenvolvimento do *software* embarcado o mais cedo possível, deu origem à modelagem no nível de transações ou TLM (*transaction-level modeling*). O projeto inicia-se com um modelo TLM atemporal, mas a posterior anotação de restrições temporais exige que o *software* embarcado seja revisado, sendo úteis ferramentas de análise de restrições temporais pós-compilação.

Esta dissertação descreve uma técnica automaticamente redirecionável que combina análise de restrições temporais e escalonamento de código *assembly*. A técnica baseia-se na extração de informações específicas da arquitetura-alvo através de uma descrição formal do processador e na codificação de restrições temporais e de precedência em uma representação unificada usando grafos. Resultados experimentais mostram que a técnica não somente lida eficientemente com restrições temporais, mas também as explora para guiar as otimizações. São apresentados resultados para os processadores MIPS, PowerPC e SPARC, onde acelerações na execução do código de até 1,3 vezes foram obtidas em relação ao código pré-otimizado. Este trabalho aborda ainda um estudo da viabilidade de se integrar a técnica proposta em um tradutor binário, contribuindo para que, ao se traduzir código compilado de uma arquitetura para outra, o código traduzido resulte otimizado. Resultados preliminares são apresentados como um forte indício de viabilidade.

Abstract

The evolution of computing systems gave rise to the systems-on-chip or SoCs, where different components (such as memory, buses and processor(s)) are present in a single chip. SoCs possibly contain multiple processors of different types, therefore their design space exploration requires retargetable tools. The growth in the complexity of these systems, along with a shorter time-to-market and the need to launch embedded software development as early as possible, gave rise to transaction-level modeling (TLM). System design starts with an untimed TLM model, but the later annotation of timing constraints requires the revision of the embedded software, where post-compiling timing constraint analysis tools are useful.

This dissertation describes an automatically retargetable technique which combines time-constraint analysis and assembly code scheduling. The technique relies on the extraction of target-dependent information from a formal description of the target-CPU and on the encoding of time and precedence constraints on a unified graph representation. Experimental results show that the technique not only handles time-constraints efficiently, but also exploits them to guide code optimizations. Results are presented for the MIPS, PowerPC and SPARC processors, where speed-ups of 1.3 over pre-optimized code were obtained. This work also describes a study of the viability of integrating the proposed technique in a binary translator so that, when code is translated from one architecture to another, the translated code results optimized. Preliminary results are presented as a strong indication of viability.

Capítulo 1

Introdução

A constante evolução dos sistemas eletrônicos nas últimas décadas deu-se de forma muito acelerada, tanto do ponto de vista do desempenho como do ponto de vista da complexidade. Além do impacto óbvio nos computadores de uso geral, os sistemas eletrônicos estão hoje embarcados em equipamentos de uso cotidiano como automóveis, telefones celulares e outros dispositivos móveis, eletrodomésticos e aparelhos eletrônicos de entretenimento (jogos, TV, CD, DVD, etc.).

Os primeiros sistemas eram compostos por vários circuitos integrados ou CIs, que por sua vez eram compostos por algumas dezenas ou centenas de transistores. Em 1965, Gordon Moore declarou que o número de transistores que poderiam ser incorporados em um CI dobraria a cada dezoito meses, previsão que realmente se concretizou e ficou conhecida como Lei de Moore, valendo por mais de três décadas. A redução no tamanho dos transistores tem provocado melhoria de desempenho e aumento da complexidade dos circuitos integrados. Tal evolução deu origem, em meados dos anos 90, a uma nova tecnologia, o *System-on-chip* ou SoC. Um SoC é um CI contendo todo um sistema computacional composto de vários componentes como memória, barramentos, processador(es) e periféricos.

1.1 SoCs e plataformas

Ao mesmo tempo em que a evolução dos SoCs abriu novos caminhos para a indústria de semicondutores, ela trouxe também novos desafios. O rápido crescimento da complexidade dos SoCs fez com que houvesse também um aumento nos custos de projeto de tais sistemas. Destacam-se aqui os custos das máscaras dos CIs e de engenharia não-recorrente ou NRE (*non-recurring engineering*), que são custos envolvidos no processo de criação de um novo SoC (desde a pesquisa e o projeto inicial até o teste deste novo sistema). Tais aumentos nos custos de projeto tornam crítica a necessidade de se obter sucesso na primeira tentativa de produção de um SoC, ou seja, erros no projeto das funcionalidades e desempenho não são tolerados [GHE 05]. Além disto, ao mesmo tempo que aumenta a complexidade e os custos de projeto, o *time-to-market* vem diminuindo significativamente, ou seja, é preciso colocar o produto no mercado cada vez mais cedo para obter uma vantagem sobre os competidores, mas o produto vem ficando cada vez mais complexo.

Nos primeiros projetos de SoCs, o desenvolvimento do hardware e do software dependente do hardware eram feitos separadamente. Criava-se um modelo do hardware no nível de descrição *Register-Transfer Level* (RTL)[1] e depois de este modelo estar pronto eram feitas simulações para co-validar o hardware e o software dependente do hardware. Como o desenvolvimento do modelo RTL pode ser muito demorado, havia um atraso na validação do software. Além disto, durante a validação deste software eram descobertos erros no projeto do hardware, fazendo com que fosse necessária uma nova iteração neste processo. Ademais, a simulação em nível RTL pode tornar-se extremamente lenta para sistemas muito complexos.

O aumento crescente nos custos de projeto de SoCs combinado com um *time-to-market* cada vez menor fez com que a indústria buscasse outros paradigmas de projeto. Uma tentativa foi a de promover o re-uso de blocos de propriedade intelectual ou IPs

[1] A descrição em nível RTL baseia-se na noção de transferência entre registradores ou RT (*register-transfer*). Uma RT descreve o consumo de operandos armazenados em registrador, a execução de uma operação e o armazenamento do restulado também em registrador.

(*intellectual property*) ao longo de diferentes projetos visando economizar tempo e custo de desenvolvimento, o que em vários casos mostrou-se pouco prático devido à dificuldade em entender o funcionamento e integrar IPs de terceiros [GHE 05]. Outra alternativa seria o chamado *co-projeto hardware/software*, onde o desenvolvimento e validação do hardware e do software seriam feitos de forma simultânea. Apesar de ter benefícios, esta abordagem ainda utilizava modelos RTL do hardware, o que acabava atrasando o início do processo de co-verificação.

Outro caminho proposto por parte da indústria foi o de elevar o nível de abstração para acima do RTL. Inicialmente, tentou-se a criação de modelos funcionais do hardware com precisão de ciclos, mas além da dificuldade na captura das informações temporais a serem incluídas neste modelo, o seu tempo de simulação era muito alto, apenas uma ordem de magnitude acima dos modelos RTL [GHE 05].

Surgiu então a abordagem de *projeto orientado a plataformas* [VIN 02], onde propõe-se que em cada estágio do fluxo de projeto do sistema seja criada uma "plataforma". Uma plataforma é definida como "um circuito integrado flexível onde a customização para uma aplicação em particular é conseguida programando-se um ou mais componentes do chip" [VIN 02]. Estas plataformas são representadas em diferentes níveis de abstração do sistema (cada plataforma é refinada para dar origem a uma nova plataforma descrita no nível de abstração imediatamente inferior). A intenção é que estas plataformas possam ser mais facilmente estendidas e personalizadas, podendo assim ser utilizadas em diferentes projetos que pertençam a uma mesma classe de aplicações.

Dentro da abordagem de projeto orientado a plataformas, foi proposta a *modelagem no nível de transações* ou TLM (*transaction level modeling*) [GHE 05]. Este tipo de modelagem baseia-se no desenvolvimento de um modelo do hardware onde os componentes do SoC são descritos por "módulos", os quais se comunicam através de "transações" utilizando "portas" e "interconexões" [2]. O modelo é desenvolvido com uma linguagem de descrição de sistemas como, por exemplo, SystemC [SYS 06].

[2] Uma transação descreve o transporte de dados entre dois módulos através de um meio de comunicação abstrato e possivelmente atemporal, onde comunicação e funcionalidade são ortogonais. Um módulo representa o encapsulamento de um modelo de um IP.

Inicialmente, este modelo não inclui nenhuma informação de temporização e é utilizado para verificação funcional, sendo chamado de visão do programador ou PV (*programmer's view*). Este modelo, apesar de inicialmente não incluir precisão de ciclos e nem detalhes de nível mais baixo da arquitetura, é ainda assim bastante preciso pois já inclui dados como a definição dos tipos dos barramentos, hierarquia de memória, mapeamento de entrada e saída em memória, etc. Informações essenciais sobre restrições temporais podem ser anotadas mais tarde no modelo PV para gerar um novo modelo TLM, chamado de visão do programador mais tempo ou PVT (*programmer's view plus timing*) [GHE 05]. As duas principais vantagens da modelagem TLM são:

- A simulação fica muito mais rápida, já que o modelo inicialmente não inclui informações sobre precisão de ciclos, diminuindo também o tempo para desenvolvê-lo.

- Pode-se iniciar mais cedo o desenvolvimento do software dependente do hardware, pois ele pode ser validado sobre o modelo executável do hardware. Isso acarreta a diminuição do tempo de projeto, aumentando também as chances de sucesso na primeira tentativa de desenvolvimento.

1.2 A necessidade de técnicas redirecionáveis

Como foi dito anteriormente, um SoC é basicamente composto de um ou mais processadores, memória e periféricos. A escolha de IPs considera aspectos tais como o consumo de potência e o tamanho do código e busca desempenho suficiente para satisfazer restrições de tempo real. Isto requer uma exploração do espaço de projeto. Em particular, a escolha de processadores é bastante desafiante.

Como há uma grande variedade de CPUs e os SoCs muitas vezes são arquiteturas heterogêneas compostas por diferentes tipos de CPU, durante o processo de exploração do espaço de projeto várias alternativas diferentes devem ser testadas, desde processadores de propósitos gerais, DSPs (*digital signal processors*), até processadores específicos para uma aplicação, os ASIPs (*application-specific instruction-set processors*).

Para cada alternativa de CPU, precisa-se de um conjunto de ferramentas para dar suporte à geração, inspeção e otimização de código tais como compilador, montador, ligador, simulador, depurador, escalonador, etc. Para processadores de propósitos gerais este conjunto de ferramentas normalmente já está disponível, mas este nem sempre é o caso quando CPUs dedicadas como, por exemplo, os DSPs e os ASIPs, são usadas. Portanto, é essencial que estas ferramentas sejam facilmente redirecionáveis para as diferentes alternativas de arquiteturas sendo testadas.

Um bom ponto de partida para redirecionar tais ferramentas é uma descrição do processador feita através de uma linguagem de descrição de arquiteturas ou ADL (*architecture description language*). Com uma ADL, é possível descrever a arquitetura do conjunto de instruções de um processador (além de outras características como a hierarquia de memória, estrutura do *pipeline*, etc.) e, a partir desta descrição, gerar ferramentas de suporte à geração, inspeção e otimização de código, ou ainda gerar um modelo executável do hardware (simulador).

1.3 A necessidade da captura de restrições temporais

O projeto contemporâneo de sistemas inicia-se pela criação de uma descrição em TLM sem incluir informações sobre restrições de tempo, o modelo PV [GHE 05]. A principal vantagem desta metodologia é que esta abstração leva muito menos tempo para ser elaborada do que uma descrição RTL e resulta em menores tempos de simulação, permitindo iniciar mais cedo o desenvolvimento do software dependente de hardware.

No entanto, é necessária a posterior inserção de restrições temporais no modelo, gerando o modelo PVT. Isto faz com que o software dependente de hardware, desenvolvido e validado com o modelo atemporal, precise ser revisado e adaptado para atender às restrições temporais do modelo PVT.

Como as restrições temporais são impostas ao código já compilado, são desejáveis técnicas de otimização pós-compilação que levem em conta tais restrições, direcionando as otimizações no código a partir delas e verificando se o código efetivamente as satisfaz.

1.4 A pragmaticidade da otimização pós-compilação

Compiladores bastante populares, como o GCC [GCC 03], possuem suporte para geração de código para vários processadores (normalmente de propósitos gerais). Estes compiladores geralmente incluem técnicas avançadas de escalonamento de código. Justamente por gerarem código para CPUs de propósitos gerais, estas técnicas buscam otimizar apenas o desempenho médio, não levando em conta o tempo de execução no pior caso. Além disto, estes compiladores têm uma limitação prática: como geram código para uma variedade de CPUs, costuma-se limitar o suporte de otimizações dependentes da arquitetura-alvo. Portanto, quando restrições de tempo real devem ser levadas em consideração, um escalonador de código convencional fica limitado ao esquema de tentativa e erro. Da mesma forma, a necessidade de rápida compilação para que sejam exploradas diferentes alternativas de CPU também limita o uso de otimizações dependentes de máquina mais agressivas.

Quando há restrições temporais às quais um programa deve satisfazer, alguns poucos ciclos desperdiçados em trechos críticos do código podem ser cruciais. Segundo [LEU 01], seria desejável que um compilador pudesse acionar técnicas de otimização mais agressivas quando estivesse lidando com tais trechos de código, conhecidas como *hot spots* (por exemplo, laços que são repetidos muitas vezes e consomem muito tempo). Como isto não é suportado em compiladores convencionais, o desenvolvedor é obrigado a fazer otimizações manuais em trechos críticos do código fonte ou do código compilado, um processo que obviamente torna-se impraticável sob a pressão do *time-to-market*, quando deve ser repetido para várias CPUs diferentes.

No entanto, apesar das limitações das otimizações feitas por compiladores de uso geral, a geração de novos compiladores torna-se muito custosa, sendo que o re-uso da infra-estrutura já existente (por exemplo, GNU) mostra-se bastante pragmático. Desta forma, para obter um código de qualidade e ainda assim não abrir mão desta infra-estrutura pré-estabelecida, as técnicas de otimização após a compilação representam uma oportunidade a ser explorada.

1.5 A contribuição desta dissertação

Esta dissertação tem como principal produto de trabalho uma ferramenta de otimização pós-compilação, redirecionável e apta a tratar restrições temporais, a qual escalona o código *assembly* e faz a alocação dos registradores para gerar código otimizado. A técnica utilizada é automaticamente redirecionável e combina análise de restrições temporais e escalonamento de código *assembly*. Esta técnica é baseada em dois mecanismos principais:

- A extração automática de informações dependentes da arquitetura a partir de uma descrição formal de um processador-alvo arbitrário, o que torna a ferramenta redirecionável.
- A codificação de restrições temporais e de precedência das instruções em uma representação unificada utilizando-se grafos, o que provê a base para a análise de restrições temporais e de factibilidade do escalonamento.

Um subproduto desta dissertação é a integração do otimizador em um tradutor binário, dando origem a uma ferramenta não somente capaz de traduzir código compilado de uma arquitetura para outra, mas também apto a gerar o código otimizado para a arquitetura-alvo.

1.6 A organização desta dissertação

Esta dissertação está organizada da seguinte maneira: o Capítulo 2 apresenta uma revisão dos trabalhos correlatos na literatura, nas áreas de ferramentas redirecionáveis, análise de restrições temporais, otimização de código após a compilação e tradução binária. O Capítulo 3 explica como é feita a modelagem das CPUs utilizando-se uma ADL, bem como a geração da representação intermediária do código através de grafos, nos quais é feita a codificação das restrições temporais e de precedência. O Capítulo 4 descreve o otimizador de código redirecionável, mostrando a estrutura da ferramenta, seus principais componentes, fases de processamento, algoritmos e resultados dos experimentos. O

Capítulo 5 mostra como o escalonador redirecionável proposto nesta dissertação pode ser integrado a um tradutor binário, melhorando a qualidade do código traduzido. No Capítulo 6, são apresentadas as conclusões e apontadas perspectivas de trabalhos futuros.

O trabalho de pesquisa reportado nesta dissertação foi desenvolvido sob fomento parcial do Programa Nacional de Cooperação Acadêmica (PROCAD) da CAPES, no âmbito do Projeto n. 0326054, intitulado "Automação de Projeto de Sistemas Dedicados Usando uma Linguagem de Descrição de Arquiteturas", que norteia e formaliza as atividades de cooperação científica entre o Programa de Pós-Graduação em Ciência da Computação (PPGCC) da UFSC e o Instituto de Computação da UNICAMP. O projeto foi executado no Laboratório de Automação do Projeto de Sistemas (LAPS) da UFSC (http://www.laps.inf.ufsc.br/).

O desenvolvimento deste trabalho foi parcialmente amparado por bolsa de mestrado oferecida pela Motorola Industrial, no âmbito da rede Brazil Test Center (BTC), mediante contrapartida de prestação de serviços ao Projeto Test Automation no Laboratório de Desenvolvimento de Software (LabSoft) da UFSC (http://www.labsoft.ufsc.br).

Capítulo 2

Trabalhos Correlatos

2.1 Ferramentas redirecionáveis

O uso de processadores dedicados a uma classe de aplicações (os ASIPs) nos SoCs trouxe a necessidade de que conjuntos de ferramentas de suporte à geração de código pudessem ser geradas ou redirecionadas automaticamente para estas novas arquiteturas. Vários trabalhos na literatura abordam este tema, focando no redirecionamento de ferramentas como compiladores e montadores a partir de descrições do processador feitas com uma ADL.

Em [HAN 98] é apresentada a ferramenta AVIV, um gerador de código redirecionável. Esta ferramenta recebe como entrada o código fonte escrito em C ou C++ e primeiramente faz otimizações independentes da arquitetura-alvo, gerando um formato intermediário de código. A partir deste, o *back-end* do compilador baseia-se em uma descrição da arquitetura-alvo feita em ISDL [HAD 97] para gerar o código *assembly*. Esta descrição em ISDL também é usada para a geração automática de um montador.

O trabalho relatado em [HAN 98] aborda a geração do código *assembly* para a arquitetura-alvo. A ferramenta AVIV usa como *front-end* os compiladores SUIF [Sta 94] e SPAM [SPA 97] para a geração do formato intermediário do código. Este formato consiste em grafos de fluxo de controle e de dados que são usados para explorar as possibilidades de geração de código dos blocos básicos. São feitas transformações sobre estes

grafos e utilizadas heurísticas para realizar, a partir da descrição ISDL, a associação de unidades funcionais a instruções, seleção de instruções, alocação de registradores e escalonamento. O *back-end* do compilador AVIV foca na minimização do tamanho do código gerado.

O compilador redirecionável EXPRESS é apresentado em [HAL 01]. Ele utiliza a ADL EXPRESSION [HAL 99] para ser redirecionado para diferentes arquiteturas e recebe como entrada programas escritos em linguagem C. O *front-end* do compilador EXPRESS é baseado no *front-end* do compilador GCC e faz algumas otimizações independentes da arquitetura-alvo, enquanto que seu *back-end* faz a seleção de instruções e alocação de registradores. O *back-end* desse compilador incorpora um *framework* chamado *Transmutations*, o qual busca o melhor ordenamento das fases de processamento do compilador de acordo com as características do código e dos recursos disponíveis, procurando ainda customizar o compilador para diferentes classes de arquiteturas.

A geração automática de um escalonador de instruções a partir da ADL LISA [PEE 99] foi relatada em [WAH 03]. Neste trabalho, foi utilizada a ferramenta de desenvolvimento de compiladores CoSy, a qual é capaz de redirecionar o *back-end* do compilador para diferentes arquiteturas, com base em arquivos que descrevem as fases de geração de código.

A partir da descrição do processador feita em LISA, gera-se o arquivo contendo a descrição do escalonador de instruções do compilador a ser gerado pelo CoSy, o qual é combinado com outros arquivos (gerados manualmente) que descrevem a seleção de instruções e alocação de registradores. No entanto, apesar de em [HOH 04] os autores mencionarem que a geração do escalonador é totalmente automática a partir da descrição LISA, isto não fica totalmente explícito já que em [WAH 03] é reportado que a ferramenta gera apenas a parte que descreve os *hazards* estruturais, ou seja, os potenciais conflitos de recursos da arquitetura-alvo. As informações sobre as latências entre instruções não são extraídas automaticamente da descrição LISA; portanto, essa descrição deve ser elaborada manualmente.

Os trabalhos em [TAG 05b], [TAG 05a] e [BAL 05a] abordam a geração automática de montadores a partir de uma descrição do processador feita utilizando-se a ADL ArchC

[RIG 04].

Em [TAG 05b] e [TAG 05a] as informações específicas da arquitetura são extraídas da descrição ArchC e armazenadas em estruturas de dados, as quais são combinadas a um *parser* do código *assembly* da arquitetura-alvo, que também é gerado a partir da descrição do processador. Em um arquivo separado, são descritas informações sobre pseudo-instruções e sobre o cálculo dos endereços-alvo dos desvios, e este arquivo é utilizado para gerar um pré-processador, o qual é integrado ao montador no final do processo.

Já a abordagem em [BAL 05a] baseia-se nas ferramentas do pacote GNU Binutils [BIN 05] para redirecionar o montador GNU (*gas*) para uma determinada arquitetura-alvo. Primeiramente, foi feita uma extensão da linguagem ArchC, adicionando-se novas construções que permitissem a inclusão de informações necessárias à geração automática do montador, como informações sobre diferentes sintaxes *assembly* e decodificação de operandos das instruções [BAL 05b]. Com estas informações extraídas da descrição ArchC, são gerados automaticamente os arquivos do pacote GNU Binutils que são dependentes da arquitetura-alvo, enquanto os arquivos e bibliotecas independentes são re-utilizados.

Em [CAS 06] as ferramentas do pacote GNU Binutils também foram utilizadas, mas desta vez na geração automática de linkeditores a partir de uma descrição ArchC, redirecionando o linkeditor GNU (*ld*). As informações sobre relocações são incluídas na descrição ArchC re-utilizando ao máximo as construções já existentes na linguagem, mas foi feita ainda uma extensão da ferramenta descrita em [BAL 05a] para que os montadores por ela gerados fossem capazes de inserir informações de relocação para serem usadas como insumo para o linkeditor.

Uma revisão abrangente sobre compiladores redirecionáveis e técnicas de redirecionamento de ferramentas pode ser encontrada em [LEU 01].

2.2 Análise de restrições temporais

Na área de compiladores, os algoritmos clássicos de escalonamento não abordam diretamente a factibilidade sob restrições temporais, pois são focados em otimizar o de-

sempenho médio. No entanto, a análise de restrições temporais no escalonamento foi abordada em trabalhos nas áreas de síntese comportamental e geração de código para DSPs.

Um grafo orientado ponderado foi proposto em [MIC 94] como forma de se unificar a representação de restrições temporais e dependências de dados entre instruções. Com as relações de precedência e de tempo sendo codificadas como arestas no grafo, este modelo trata o problema do escalonamento sob restrições temporais como um problema de cálculo de caminho mais longo, permitindo assim que seja feita uma análise de factibilidade do escalonamento utilizando-se algoritmos clássicos como o de Bellman-Ford [COR 90].

Como o modelo descrito em [MIC 94] assume que os recursos da arquitetura-alvo são ilimitados, ele é de pouco uso prático, caso não seja combinado com técnicas que assumam restrições de recursos.

Em [MES 97] foi proposta uma extensão deste modelo onde todas as restrições (de tempo, de precedência e de recursos) são modeladas no mesmo grafo. Os vértices do grafo representam as operações, enquanto que as arestas representam as restrições entre elas. Por exemplo, se duas operações competem por um mesmo recurso, este conflito é evitado através da inserção de uma aresta ponderada entre elas, cujo peso é o atraso do recurso em questão. Da mesma forma, cada vez que uma operação é escalonada são inseridas novas arestas. A cada nova operação escalonada, um analisador é invocado para verificar se o escalonamento satisfaz as restrições codificadas no grafo. Como a inserção destas novas arestas reduz as possibilidades de ordenamento das instruções, o analisador acelera a convergência para um escalonamento factível, ou seja, quanto mais rígidas forem as restrições, mais rapidamente a solução é encontrada. Como esta abordagem assume que a arquitetura-alvo é um processador DSP com conjunto de instruções VLIW (*very long instruction word*), ela não é totalmente adequada para a exploração do espaço de projeto em SoCs, já que durante este processo devem ser levadas em consideração diferentes arquiteturas como ASIPs e processadores de propósitos gerais.

2.3 Otimização pós-compilação

Apesar de muitos compiladores realizarem otimizações durante o processo de geração de código objeto, estas melhorias visam normalmente otimizar o desempenho médio e baseiam-se em otimizações independentes da arquitetura para a qual se está compilando. Na literatura, entre os trabalhos sobre otimizações feitas após a compilação destacam-se o projeto SALTO [SAL 06] e PROPAN [KäS 00].

A técnica descrita em [SAL 06] baseia-se em um *framework* em que o usuário pode implementar ferramentas de otimização de código compilado. O *framework* baseia-se em uma descrição formal da arquitetura-alvo através de uma ADL similar à linguagem LISP. Esta descrição é lida juntamente com o código a ser otimizado, e este código é transformado em uma representação intermediária. Através de uma interface, o usuário pode manipular esta representação intermediária. Dados como tabelas de recursos, de dependências e de latências entre instruções podem ser acessados, e o código pode ser modificado através da inserção ou remoção de instruções. O usuário tem a opção de implementar ferramentas para a otimização ou para o *profiling* do código, para isto tendo que prover os algoritmos adequados. A ferramenta SALTO foca principalmente em arquiteturas do tipo VLIW.

Já a ferramenta PROPAN [KäS 00] tem como foco principal os DSPs. Ela toma como entrada uma descrição da arquitetura-alvo através da linguagem TDL [KäS 03], gerando um *parser* do código *assembly* e armazenando informações dependentes da arquitetura em estruturas de dados internas. Após a leitura da descrição da arquitetura-alvo, PROPAN renomeia os registradores para eliminar dependências de nomes. Em seguida, são aplicadas otimizações específicas da arquitetura-alvo, as quais são baseadas em técnicas que levam em conta "superblocos"[1] de código. Apesar de alcançar bons resultados quanto à redução de código quando comparada com outras heurísticas, esta técnica tem um tempo de computação significativamente alto [LEU 01].

[1] Superblocos são trechos de código que envolvem instruções pertencentes a múltiplos blocos básicos, com o objetivo de explicitar mais paralelismo e com isso aumentar as chances de otimização.

2.4 Tradução binária

Além de realizar otimizações no código gerado pelo compilador, levando em conta restrições temporais, a ferramenta proposta nesta dissertação pode ser integrada a um tradutor binário, permitindo assim que um código *assembly* seja traduzido de uma determinada arquitetura para outra e seja em seguida otimizado para esta arquitetura alvo. Por isso, cabe aqui uma breve análise dos trabalhos em tradução binária mais relevantes para a discussão do Capítulo 5.

Os trabalhos em [CIF 99] e [CIF 00] apresentam a ferramenta UQBT, um tradutor binário capaz de gerar código para arquiteturas baseadas em registradores ou em pilha. A ferramenta re-utiliza informações independentes da arquitetura-alvo, enquanto que as informações dependentes da arquitetura (incluindo sintaxe e semântica das instruções, além da indicação de quais instruções causam transferência de controle quando executadas) devem ser fornecidas através de descrições em diferentes linguagens. O código a ser traduzido é transformado em uma representação intermediária similar à de uma descrição RTL, a qual descreve a maneira como as instruções da arquitetura-fonte interagem com os registradores. Esta representação é transformada em uma de mais alto nível e independente da arquitetura, a qual descreve transferências de controle elementares (desvios condicionais, incondicionais, chamadas de procedimento, etc.) e supõe um número infinito de registradores. Levando em consideração os parâmetros da arquitetura-alvo, esta descrição é então transformada novamente na descrição similar à de RTL, a qual é usada para gerar código. Isto permite, por exemplo, que seja feita a tradução de código de uma arquitetura baseada em pilha para outra baseada em registradores.

Enquanto o UQBT aborda a tradução binária estática, o trabalho apresentado em [CIF 02] descreve um *framework*, chamado *Walkabout*, focado na tradução binária dinâmica, o qual é inspirado no UQBT. Similarmente ao UQBT, a ferramenta Walkabout utiliza descrições das sintaxes e semântica das instruções para obter redirecionabilidade. No processo de tradução, o código-fonte é primeiramente carregado em uma memória virtual e interpretado até que seja detectado um trecho recorrente ou um *hot path* [2]. Este

[2] Trecho de código executado com bastante freqüência.

trecho é então traduzido e submetido a algumas otimizações simples. O código resultante é colocado em uma memória *cache* de instruções traduzidas. Os interpretadores e a ferramenta que identifica os *hot paths* são gerados automaticamente a partir das descrições da sintaxe e semântica das instruções, feitas com as linguagens SLED [RAM 97] e SSL [CIF 98] respectivamente.

Capítulo 3

Fundamentação Teórica

3.1 Modelagem de CPUs

A descrição de um processador através de uma ADL é um bom ponto de partida para a geração de ferramentas redirecionáveis. O escalonador redirecionável a ser descrito no Capítulo 4 requer que as propriedades da CPU-alvo sejam representadas através de um modelo formal. Para fins de implementação, os modelos de CPUs são descritos através da ADL ArchC [RIG 04]. Embora qualquer ADL capaz de descrever as propriedades necessárias ao escalonamento pudesse ser usada, a adoção de ArchC - sem perda de generalidade - deve-se à sua disponibilidade sob licença GPL (*General Public License*) e ao fato de gerar modelos em SystemC adequados ao estilo TLM. Nesta seção, primeiramente a estrutura de um modelo ArchC é ilustrada através de exemplos e, em seguida, algumas extensões são propostas na linguagem para permitir a descrição de propriedades específicas utilizadas pelo escalonador.

3.1.1 Estrutura básica do modelo

Basicamente, um modelo descrito em ArchC é dividido em três partes principais, que residem em arquivos separados:

- *Parâmetros da arquitetura:* neste arquivo são definidos os recursos da arquitetura como *endian* e os tamanhos do banco de registradores, da memória cache e da

palavra de dados. Nos modelos com precisão de ciclos, aqui são também declarados os estágios do *pipeline*, bem como registradores específicos utilizados no transporte de informações entre um estágio e outro. Além disto, este arquivo referencia o arquivo que contém a descrição da arquitetura do conjunto de instruções ou ISA (*Instruction Set Architecture*). A Figura 3.1 mostra um exemplo de descrição dos parâmetros da arquitetura MIPS.

```
AC_ARCH(mips1)
{
    ac_wordsize 32;             // Tamanho da palavra
    ac_cache   DM:5M;           // Tamanho da memória cache
    ac_regbank RB:34;           // Tamanho do banco de registradores

    ARCH_CTOR(mips1)
    {
        // Referência ao arquivo que descreve a arquitetura do conjunto de instruções
        ac_isa("mips1_isa.ac");
        set_endian("big");      // Declaração arquitetura como 'big endian'
    };
};
```

Figura 3.1: Descrição dos parâmetros da arquitetura em ArchC

- *Arquitetura do conjunto de instruções:* este arquivo é dividido em duas seções principais: a descrição das características do conjunto de instruções e a descrição das propriedades das instruções da ISA. Na primeira seção, são descritos os diferentes formatos de instrução da arquitetura, bem como a lista de instruções, associando-as a seus respectivos formatos. É feita ainda a declaração de nomes simbólicos dos registradores, que são usados pelo compilador ao gerar o código *assembly*. Na segunda seção, são declaradas propriedades específicas de cada instrução como, por exemplo, as suas diferentes sintaxes *assembly*, valores dos campos de código operacional utilizados na sua decodificação, a maneira como é efetuado o cálculo do endereço-alvo de instruções de desvio, etc. A Figura 3.2 mostra um fragmento da descrição da ISA do MIPS.

- *Comportamento das instruções:* neste arquivo é descrito o comportamento de cada instrução. É possível descrever comportamentos específicos para cada instrução,

```
AC_ISA(mips1)
{
    // Formatos de instrução
    ac_format Type_R  = "%op:6 %rs:5 %rt:5 %rd:5 %shamt:5 %func:6";
    ac_format Type_I  = "%op:6 %rs:5 %rt:5 %imm:16:s";
    ac_format Type_J  = "%op:6 %addr:26";

    // Instruções associadas aos seus formatos
    ac_instr<Type_I> beq;
    ac_instr<Type_R> add;

    // Mapeamento de nomes dos registradores
    ac_asm_map reg
    {
        "$"[0..31] = [0..31];
        "$zero" = 0;
        "$at" = 1;
        "$kt"[0..1] = [26..27];
        "$gp" = 28;
        "$sp" = 29;
        "$fp" = 30;
        "$ra" = 31;
    }

    // Propriedades das instruções
    ISA_CTOR(mips1)
    {
        add.set_asm("add %reg, %reg, %reg", rd, rs, rt);
        add.set_decoder(op=0x00, func=0x20);

        beq.set_asm("beq %reg, %reg, %expR4A", rs, rt, imm);
        beq.set_asm("b %expR4A", imm, rs=0, rt=0);
        beq.set_asm("beqz %reg, %expR4A", rs, imm, rt=0);
        beq.set_decoder(op=0x04);
        beq.is_branch((ac_pc+4) + (imm<<2));
        beq.cond(RB[rs] == RB[rt]);
        beq.delay(1)
    };
};
```

Figura 3.2: Descrição da arquitetura do conjunto de instruções em ArchC

comportamentos comuns a todas as instruções de um determinado formato ou ainda um comportamento que é comum a todas as instruções da ISA (por exemplo, incrementar o *program counter*). A descrição dos comportamentos é feita em código C++. A Figura 3.3 mostra um exemplo de descrição contendo um comportamento comum a todas as instruções do MIPS e o comportamento de uma instrução específica.

Apesar de a ADL ArchC prover uma variedade de opções para se descrever um modelo de uma arquitetura, há algumas informações requeridas pelo escalonamento de código cuja descrição não era originalmente suportada nos modelos ArchC. Em face das limitações da ADL, algumas extensões foram propostas para viabilizar o escalonamento

```
#include  "mips1-isa.H"
#include  "ac_isa_init.cpp"

// Comportamento geral de todas as instruções
void ac_behavior( instruction )
{
    #ifndef NO_NEED_PC_UPDATE
        ac_pc = ac_pc +4;
    #endif
};

// Comportamento da instrução lb
void ac_behavior( lb )
{
    char byte;
    byte = DM.read_byte(RB[rs]+ imm);
    RB[rt] = (ac_Sword)byte ;
};
```

Figura 3.3: Descrição do comportamento de instruções em ArchC

redirecionável pós-compilação, conforme descrito nas Seções 3.1.2 e 3.1.3.

3.1.2 Limitações da modelagem ArchC

Como o escalonador redirecionável opera após o programa-fonte ter sido compilado, ele deve ser capaz de determinar as dependências de dados a partir do código em linguagem de montagem. Infelizmente, em sua versão original, as construções da ADL ArchC não capturavam informações que permitissem diretamente a análise de fluxo de dados, como explicado abaixo:

- **Operandos fonte e destino não explicitados:** Para determinar dependências de dados através de registradores, é necessário detectar se o operando-destino de uma instrução é o operando-fonte de uma outra. As construções ArchC permitem descrever os campos que codificam operandos, bem como associar os operandos da sintaxe *assembly* a seus respectivos campos nos formatos de instrução, mas não permitem distinguir quais campos codificam operandos-fonte e quais campos codificam operandos-destino.

- **Campos que determinam endereço efetivo não explicitados:** Para determinar dependências de dados através da memória e para fazer desambiguação estática de memória no programa, é necessário identificar os campos que determinam o ende-

reço efetivo de memória usado por instruções *load* ou *store*. As construções ArchC permitem descrever os campos usados para compor endereços (registrador-base e imediato), mas não permitem identificar quais campos são efetivamente utilizados para determinar o endereço de memória.

- **Latência entre instruções não explicitadas:** Para possibilitar que um escalonador estático possa reordenar as instruções sem a introdução de bolhas no *pipeline* devido a conflitos de dados, é preciso codificar as latências entre instruções, ou seja, o número de ciclos que se precisa intercalar entre duas instruções para garantir que o dado seja produzido pela primeira instrução antes que a segunda tente consumi-lo. As construções ArchC não permitem descrever as latências entre instruções. Em princípio, em uma descrição funcional com precisão de ciclos é possível determinar as latências através de uma análise da estrutura do *pipeline* e dos caminhos de *bypass-forwarding*, codificados no arquivo de comportamentos. Entretanto, a implementação automática de tal análise para extrair as latências seria ineficiente, uma vez que aquele arquivo permite vários estilos de descrição. Ademais, a extração de latências a partir do arquivo de comportamentos não permitiria que o escalonador pudesse ser usado sobre descrições puramente funcionais (sem precisão de ciclos).

Como o conceito de latência é fundamental para a discussão a ser feita na Seção 3.3, sua definição formal é apresentada a seguir.

Definição 3.1 - A latência entre duas instruções é o número de ciclos que se precisa intercalar entre elas para garantir que o dado produzido pela primeira instrução esteja disponível antes que a segunda possa consumi-lo.

Felizmente, os desenvolvedores de ArchC disponibilizam um mecanismo genérico para a extensão dos modelos com informações necessárias a novas aplicações. Este mecanismo baseia-se na construção `set_property`. Desta forma, não é necessário criar uma nova palavra-chave para cada nova propriedade que se precise descrever no modelo ArchC.

Primeiramente, cada nova propriedade deve ser declarada utilizando-se uma construção do tipo `ac_property`, que tem a seguinte estrutura básica:

ac_property{propriedade_1, propriedade_2, ..., propriedade_n}

Em seguida, deve-se associar cada propriedade aos elementos da descrição. Tal associação é feita através da seguinte declaração:

<dono da propriedade>.**set_property**(<propriedade>, <descrição>);

Os elementos de tal declaração são os seguintes:

- <dono da propriedade>: pode ser uma instrução, um formato ou ainda um mapeamento de nomes declarado pela construção `ac_asm_map`, ou seja, uma propriedade pode ser associada a uma instrução específica ou a todas as instruções de um mesmo formato, ou então, por exemplo, a um grupo de registradores (para se declarar quais são usados como registradores de propósito geral).

- <propriedade>: nome da propriedade a ser associada; este nome deve ter sido declarado utilizando-se `ac_property`.

- <descrição>: a descrição específica da propriedade sendo associada. Por exemplo, se queremos declarar qual é o operando destino de uma instrução, a descrição da propriedade seria o nome deste operando.

O *parser* do modelo ArchC se encarrega de verificar se todas as propriedades referenciadas em `set_property` foram declaradas na construção `ac_property`, e monta uma lista que relaciona todas propriedades, a quem foram associadas e qual a descrição de cada uma. Cada ferramenta que utiliza o *parser* do modelo se encarrega de tratar esta lista, validando as propriedades e suas descrições.

A seguir, são listadas as propriedades a serem declaradas para viabilizar o escalonamento de código, bem como a sintaxe adotada para a declaração dessas propriedades.

3.1.3 Extensões propostas para viabilizar o escalonamento de código

As extensões a seguir sugeridas contornam as limitações descritas na Seção 3.1.2, explicitando os campos que descrevem operandos-fonte e destino e endereço efetivo, ex-

plicitando as latências entre instruções, além de explicitar quais registradores serão considerados de uso geral para fins de alocação de registradores.

- **Propriedade *source*:** esta propriedade define quais são os operandos fonte de uma instrução. A sua descrição deve conter uma lista com nomes de campos do formato da instrução, declarados com `ac_format`. Pode-se ainda explicitar que uma instrução tem como operando-fonte um valor da memória, sendo que para isso declara-se uma lista, entre colchetes, dos campos que compõe o endereço da memória onde este valor será lido. A Figura 3.4 mostra exemplos do uso de `set_property` para explicitar quais são os operandos-fonte de uma instrução.

```
// Todas as instruções do tipo R têm como fontes os operandos "rs" e "rt".
Type_R.set_property(source,(rs,rt));

// Instrução "sll" tem como fontes os operandos "rt" e "shamt".
sll.set_property(source,(rt,shamt));

// Instrução "sw" tem como fontes os operandos "rt", "imm" e "rs".
sw.set_property(source,(rt,imm,rs));

// Instrução "lw" tem como fontes os operandos "imm", "rs" e o valor na
// posição de memória composta por "imm" e "rs".
lw.set_property(source,(imm,rs,[imm,rs]));
```

Figura 3.4: Usando `set_property` para explicitar operandos fonte

- **Propriedade *dest*:** esta propriedade define os operandos destino de uma instrução. Similarmente à propriedade *source*, a sua descrição deve conter uma lista com nomes de campos do formato da instrução, declarados com `ac_format`, ou uma lista de campos entre colchetes para indicar que a instrução tem como destino uma posição na memória. A Figura 3.5 mostra exemplos do uso de `set_property` para explicitar quais são os operandos destino de uma instrução.

- **Propriedade *efa*:** com esta propriedade, declaram-se quais campos codificam os componentes utilizados no cálculo do endereço efetivo de memória a ser lido ou escrito pela instrução. A descrição da propriedade *efa* deve conter uma lista, entre colchetes, destes campos (declarados com `ac_format`). A Figura 3.6 mostra

```
// Todas as instruções do tipo R têm como destino o operando "rd".
Type_R.set_property(dest,rd);

// Instrução "sw" tem como destino a posição de memória determinada
// pela composição dos campos "imm" e "rs".
sw.set_property(dest,[imm,rs]);

// Instrução "lw" tem como destino o operando "rt".
lw.set_property(dest,rt);
```

Figura 3.5: Usando `set_property` para explicitar operandos destino

exemplos do uso de `set_property` para explicitar quais são os campos que compõe o endereço efetivo de memória acessado por uma instrução.

```
// Para as instruções "sw" e "lw", os campos que compõem o endereço
// efetivo de memória por elas acessado são "imm" e "rs".
sw.set_property(efa,[imm,rs]);
lw.set_property(efa,[imm,rs]);
```

Figura 3.6: Usando `set_property` para declarar campos do endereço de memória

- **Propriedade *latency*:** é usada para declarar a latência entre duas instruções. A descrição desta propriedade contém três elementos: a instrução em relação à qual estamos declarando a latência (a instrução sucessora e que consome o dado produzido pela instrução corrente), o nome do operando-fonte da instrução sucessora que conterá o valor produzido pela instrução corrente e o número de ciclos de latência a serem intercalados. O segundo elemento pode ser tanto um campo da instrução (declarado com `ac_format`) ou uma concatenação de campos que indicam uma posição de memória (declarada utilizando uma lista de campos entre colchetes). O número de ciclos de latência entre diferentes pares de instruções é uma informação usualmente disponível nos manuais dos processadores. A Figura 3.7 mostra exemplos do uso de `set_property` para declarar latências entre instruções.

- **Propriedade *gpr*:** esta é uma propriedade que deve ser associada a um grupo de mapeamentos feitos com `ac_asm_map`, quando este grupo representa o mapeamento dos nomes dos registradores (nomes usados pelos compiladores na geração

```
// Quando um "lw" produz um valor a ser consumido por um
// "add", há uma latência de 1 ciclo (independentemente
// do campo - "rs" ou "rt" - utilizado para codificar o
// registrador onde o valor será armazenado).
lw.set_property(latency, (add,rt,1));
lw.set_property(latency, (add,rs,1));

// Quando um "lw" produz um valor a ser consumido por um
// "sw", as latências são de 0 ou 1 ciclo, conforme o
// valor seja referenciado no campo "rs" e "rt", respectivamente.
lw.set_property(latency, (sw,rs,0));
lw.set_property(latency, (sw,rt,1));

// Quando um "sw" escreve um valor na memória em um
// endereço, composto pelos campos "imm" e "rs", onde um
// "lw" lerá um valor, há uma latência de 1 ciclo.
sw.set_property(latency, (lw,[imm,rs],1));
```

Figura 3.7: Usando `set_property` para declarar latências entre instruções

do código *assembly*) aos seus números no banco de registradores real, para indicar quais registradores são considerados de propósitos gerais para fins de alocação de registradores. A descrição desta propriedade contém uma lista com os nomes dos registradores. A Figura 3.8 mostra exemplos do uso de `set_property` para declarar a lista de registradores de propósitos gerais da arquitetura.

```
// Dentre os registradores declarados com "ac_asm_map",
// especifica a lista de registradores de propósitos gerais.
reg.set_property(gpr,("$8","$9","$10","$11","$12","$13","$14"));
```

Figura 3.8: Usando `set_property` para declarar registradores de propósitos gerais

3.2 Representação intermediária do código

Para garantir redirecionabilidade ao escalonador, a representação do programa a ser escalonado deve ser independente da arquitetura-alvo.

Assim, uma vez que o modelo ArchC tenha capturado as propriedades descritas nas seções anteriores, uma representação intermediária do código pode ser obtida através das análises de fluxo controle e de dados, conforme descrito a seguir.

3.2.1 Grafo de fluxo de controle

Um programa escrito em linguagem de montagem pode ser decomposto em trechos de código sem ramificações denominados *blocos básicos*.

Definição 3.2 - Um bloco básico (BB) é a mais longa seqüência de instruções (i, i+1, ..., n) tal que:

- Se a instrução *i* é executada, então *n* será também executada;
- Se a instrução *n* é executada, então *i* foi também executada.

Um BB pode ser identificado pelo seguinte conjunto de regras [AHO 95]:

Regra 1 - Iniciam um bloco básico:

- A primeira instrução do código.
- Instruções que são alvo de um desvio condicional ou incondicional.
- Instruções precedidas por uma instrução de desvio condicional ou incondicional.

Regra 2 - Encerram um bloco básico:

- Instruções de desvio condicional ou incondicional.
- A última instrução do código.

Uma vez dividido em blocos básicos, o código pode ser representado através de um *grafo de fluxo de controle*, definido a seguir:

Definição 3.3: Um *grafo de fluxo de controle* CFG(V, E) é um grafo orientado onde cada vértice b_i representa um BB e cada aresta (b_i, b_j) representa a ordem de precedência entre dois BBs no fluxo de execução do programa.

Como conseqüência dessa definição, as arestas de um CFG podem ser identificadas pelo seguinte conjunto de regras:

Regra 3 - Se um BB b_i termina em desvio condicional, b_i tem duas arestas emergentes (b_i, b_j) e (b_i, b_k), cada uma representando um fluxo de execução alternativo: um para o caso de desvio tomado, outro para o caso de desvio não tomado.

Regra 4 - Se um BB b_i termina em desvio incondicional, b_i tem uma única aresta emergente (b_i, b_j), onde b_j é o BB que contém a instrução-alvo daquele desvio.

Regra 5 - Se um BB b_i não termina com instrução de desvio, b_i tem uma única aresta emergente (b_i, b_j), tal que a última instrução de b_i é predecessora imediata da primeira instrução de b_j na seqüência do código.

A Figura 3.9 ilustra um CFG obtido a partir de um segmento de código através da aplicação das Regras 1 a 5.

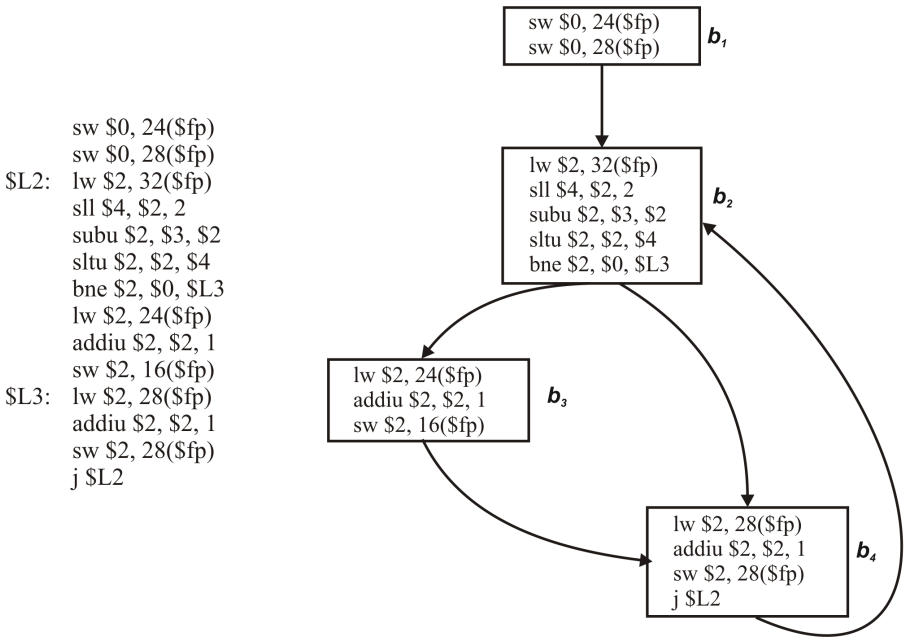

Figura 3.9: Exemplo de grafo de fluxo de controle

3.2.2 Grafo de fluxo de dados

Enquanto o CFG captura o fluxo de execução do programa, a relação de precedência entre as instruções de um dado BB pode ser representada através de um grafo de fluxo de dados, como formalizado a seguir:

Definição 3.4: Um *grafo de fluxo de dados* DFG(V, E) é um grafo polar orientado onde cada vértice v_i representa uma instrução e cada aresta (v_i, v_j) representa o fluxo de um valor produzido por v_i e consumido por v_j. Os pólos do grafo, v_0 e v_n, são chamados

source e *sink* respectivamente e representam os valores iniciais de entrada e os valores finais de saída.

As restrições de precedência são impostas pelas dependências de dados entre as instruções do bloco básico, ou seja, a produção de um dado deve preceder seu consumo.

Cada vértice do CFG será associado a um DFG. A Figura 3.10 mostra o DFG correspondente ao bloco b_2 do grafo da Figura 3.9.

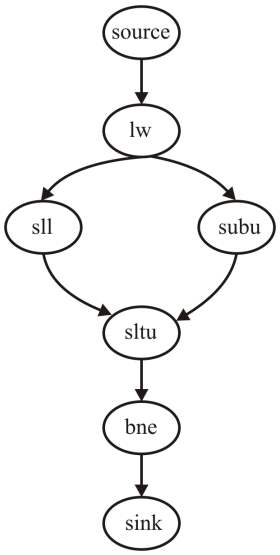

Figura 3.10: Exemplo de grafo de fluxo de dados

3.3 Modelagem de restrições temporais

Como visto na seção anterior, o DFG representa as instruções de um bloco básico e suas dependências de dados. No entanto, para escalonar instruções é necessário capturar as restrições temporais, tanto aquelas devidas às latências entre as instruções como aquelas devido a restrições de tempo real impostas a um trecho de código. A técnica aqui descrita adota a modelagem de restrições temporais proposta em [MIC 94]. Isto leva a uma representação mais genérica para as relações entre as instruções de um BB, como formalizado a seguir:

Definição 3.5: Um *grafo ponderado de precedência* WPG(V, E, W) é um grafo polar orientado, onde cada vértice v_i representa uma instrução, cada aresta representa uma restrição de precedência, e cada peso $w_{ij} \in Z$ representa o atraso relativo entre os tempos de início das instruções v_i e v_j.

As arestas do WPG, além de representarem as dependências de dados entre as instruções, também encapsulam o número de ciclos que uma instrução deve aguardar para consumir um dado produzido por outra. No entanto, uma aresta do WPG pode também representar apenas uma restrição temporal entre duas instruções, conforme as definições 3.6, 3.7 e 3.8 a seguir:

Definição 3.6: Sejam um WPG(V, E, W) e um número $k \in Z^+$. Um atraso mínimo de k ciclos entre os tempos de início de duas instruções v_i e $v_j \in V$ é representado por uma aresta $(v_i, v_j) \in E$ com peso w_{ij} = +k, desta forma restringindo a instrução v_j a iniciar sua execução pelo menos k ciclos após a instrução v_i ter iniciado a sua.

Definição 3.7: Sejam um WPG(V, E, W) e um número $k \in Z^+$. Um atraso máximo de k ciclos entre os tempos de início de duas instruções v_i e $v_j \in V$ é representado por uma aresta $(v_j, v_i) \in E$ com peso w_{ji} = -k, desta forma restringindo a instrução v_j a iniciar sua execução no máximo k ciclos após a instrução v_i ter iniciado a sua.

Definição 3.8: Sejam um WPG(V, E, W) e um número $k \in Z^+$. Um atraso exato de k ciclos entre os tempos de início de duas instruções v_i e $v_j \in V$ é representado por uma aresta $(v_i, v_j) \in E$ com peso w_{ij} = +k e uma aresta $(v_j, v_i) \in E$ com peso w_{ji} = -k, desta forma restringindo a instrução v_j a iniciar sua execução exatamente k ciclos após a instrução v_i ter iniciado a sua.

3.3.1 Modelagem de atrasos, latências e *deadlines*

A Figura 3.11 mostra a incorporação de restrições temporais às arestas do grafo da Figura 3.10.

Note que, como conseqüência da Definição 3.6, quando há uma dependência de dados entre duas instruções especifica-se que há um atraso mínimo entre produtor e consumidor através de um peso positivo atribuído à aresta que representa a restrição de prece-

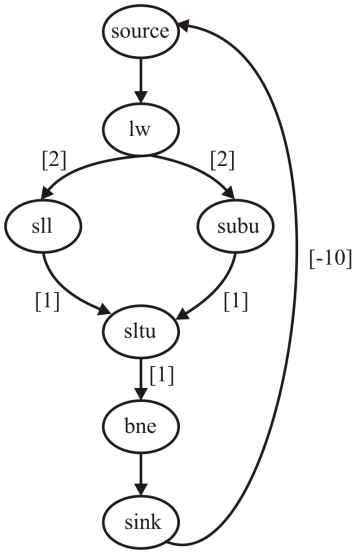

Figura 3.11: Exemplo de grafo ponderado de precedência

dência. O peso desta aresta é igual ao valor da latência entre as duas instruções (conforme a Definição 3.1) mais um. Assim, a modelagem de latências é englobada pela Definição 3.6.

Observe também que um atraso máximo de 10 ciclos foi imposto entre os vértices *source* e *sink*. Isto significa que todas as instruções compreendidas naquele BB precisam iniciar sua execução dentro deste intervalo de tempo. Portanto, a modelagem de *deadlines* é capturada pela Definição 3.7.

3.3.2 Modelagem de outros efeitos de temporização

Esta seção aborda a modelagem de vários efeitos de temporização associados a arquiteturas contemporâneas e discute as limitações da modelagem proposta no contexto de tempo real.

3.3.2.1 Desvios com atraso

As arestas ponderadas podem também ser usadas para a modelagem de desvios com atraso, que utilizam os chamados *delay slots* [PAT 04]. Em arquiteturas RISC bem conhe-

cidas, como o MIPS e o SPARC, uma instrução pode ser colocada neste *slot* e desta forma ser executada antes de o desvio ser realizado. Caso nenhuma instrução seja colocada no *slot*, uma instrução de "nop" (*no operation*) deve ser inserida ali (compiladores como o gcc normalmente inserem este "nop" ao gerar o código *assembly*).

Há ainda a possibilidade, em algumas arquiteturas, de serem usados desvios com anulação do *slot*. Nestes casos, a instrução que reside no *slot* não será executada em alguns casos específicos.

Como as informações descritas acima podem ser codificadas no modelo ArchC utilizando-se construções como `delay` e `delay_cond`, tais restrições de temporização podem ser modeladas da seguinte forma:

Regra 6 - A uma instrução de desvio com anulação do *slot* atribui-se um atraso exato de 1 ciclo em relação ao vértice *sink* do WPG, mas a qualquer outra instrução do mesmo BB é atribuído um atraso mínimo de 2 ciclos em relação a *sink*.

A Figura 3.12 mostra o WPG correspondente a um trecho de código terminado em uma instrução de desvio com anulação do *delay slot*, onde a aplicação da Regra 6 resulta na inserção das arestas pontilhadas.

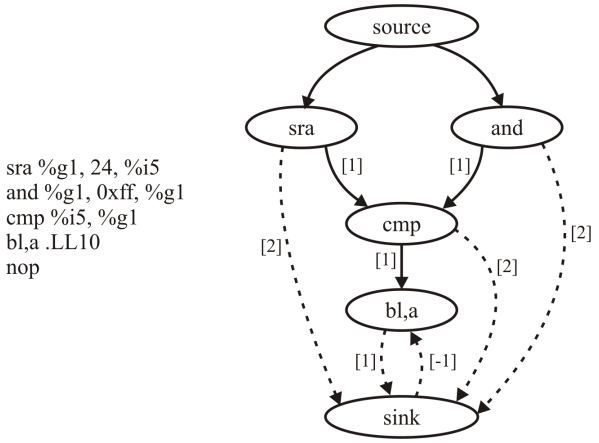

Figura 3.12: Modelagem de instrução de desvio com anulação do *slot*

Regra 7 - A uma instrução de desvio sem anulação do *slot* atribui-se um atraso exato de 1 ciclo em relação ao vértice *sink* do WPG e não há restrição alguma de atraso

mínimo para as demais instruções do BB (que podem ser candidatas a escalonamento no *slot*).

A Figura 3.13 mostra o WPG correspondente a um trecho de código terminado em uma instrução de desvio sem anulação do *delay slot*. A inserção das arestas pontilhadas é o resultado da aplicação da Regra 7.

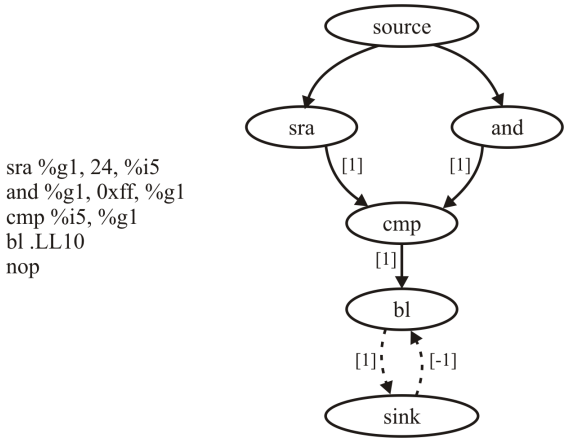

Figura 3.13: Modelagem de instrução de desvio sem anulação do *slot*

Deve-se ressaltar que as Regras 6 e 7 deliberadamente restringem o escalonamento de código no âmbito de blocos básicos. A discussão da pragmaticidade e adequação desta decisão é feita na Seção 4.1.

Desta forma, no primeiro caso obriga-se a instrução de desvio a ser a última a ser escalonada no BB (e o escalonador colocará um "nop" no *slot*, pois a instrução no *slot* correria o risco de não ser executada, violando a noção de bloco básico), enquanto que, no segundo caso, obriga-se a instrução de desvio a ser a penúltima, pois assim outra instrução do mesmo BB pode residir no *slot*.

3.3.2.2 Latências do subsistema de memória

A modelagem proposta é suficientemente genérica para capturar as restrições temporais impostas às instruções *load* e *store* devidas às latências do subsistema de memória. Entretanto, deve-se ter em mente que, para fins de análise de tempo real, não faz sentido

modelar-se o caso médio, mas o intervalo de latências a que estão suscetíveis as instruções de acesso à memória, conforme discutido a seguir.

Basicamente, o subsistema de memória pode ser implementado por dois tipos de memória: memórias com seleção através de endereço (RAM, ROM, Flash) ou memórias com seleção através de *tags* (memórias associativas). Além disso, ele pode ser organizado com hierarquia de memória (por exemplo, memórias *cache* [PAT 04]) ou sem hierarquia (por exemplo, *scratch pads* [MAR 03]).

Seja t_a o tempo de acesso a um subsistema de memória sem hierarquia. A modelagem desta latência de memória é feita conforme descrito nas Regras 8 e 9.

Regra 8 - Em um sistema *sem* hierarquia de memória, entre uma instrução *load* v_i produtora e uma instrução arbitrária v_j consumidora devem ser inseridas no WPG as seguintes arestas:

- (v_i, v_j) tal que $w_{ij} = t_a$;
- (v_j, v_i) tal que $w_{ji} = -t_a$.

Regra 9 - Em um sistema *sem* hierarquia de memória, entre uma instrução arbitrária v_i produtora e uma instrução *store* v_j consumidora devem ser inseridas no WPG as seguintes arestas:

- (v_i, v_j) tal que $w_{ij} = t_a$;
- (v_j, v_i) tal que $w_{ji} = -t_a$.

Sejam agora t_{hit} e t_{miss} os tempos de acesso quando um dado é encontrado no nível inferior ou no nível superior da hierarquia, respectivamente. A modelagem destas latências é feita de acordo com a Regras 10 e 11.

Regra 10 - Em um sistema *com* hierarquia de memória, entre uma instrução *load* v_i produtora e uma instrução arbitrária v_j consumidora devem ser inseridas no WPG as seguintes arestas:

- (v_i, v_j) tal que $w_{ij} = t_{hit}$;
- (v_j, v_i) tal que $w_{ji} = -t_{miss}$.

Regra 11 - Em um sistema *com* hierarquia de memória, sejam uma instrução arbitrária v_i produtora e uma instrução *store* v_j consumidora. São inseridas duas arestas no WPG:

- (v_i, v_j) tal que $w_{ij} = t_{hit}$;
- (v_j, v_i) tal que $w_{ji} = -t_{miss}$.

Vale ressaltar que, como os tempos t_{hit} e t_{miss} podem diferir de uma ou mais ordens de magnitude, muitos sistemas de tempo real utilizam a técnica de *scratch pad* para acelerar o acesso a dados ao invés de recorrer à estrutura hierárquica de memória.

3.3.2.3 Efeitos dinâmicos de execução

Várias arquiteturas contemporâneas desenvolvidas originalmente para o mercado de computação de uso geral são utilizadas em sistemas embarcados, possivelmente com requisitos de tempo real.

Ora, várias técnicas dinâmicas para exploração de paralelismo de baixa granularidade foram introduzidas para melhorar a execução do caso médio, tais como previsão dinâmica de desvios, escalonamento dinâmico de código, execução especulativa dinâmica [PAT 06]. Essas técnicas visam permitir a execução fora de ordem (*out-of-order execution*) do código para otimizar o caso médio, mas não oferecem garantia alguma sobre a factibilidade de restrições de tempo real, uma vez que elas deterioram a previsibilidade do pior caso de execução, inviabilizando a análise de factibilidade em tempo de compilação. Por isso, arquiteturas destituídas de técnicas dinâmicas são muitas vezes adotadas sob restrições severas de tempo real como, por exemplo, as arquiteturas VLIW subjacentes a vários processadores DSP.

Portanto, como a modelagem aqui proposta visa a análise de factibilidade de restrições de tempo real, não faz sentido empregá-la para a modelagem de efeitos dinâmicos.

3.3.3 Verificação da factibilidade do escalonamento

Após terem sido construídos WPGs para todos os blocos básicos, cada BB é submetido ao escalonador, o qual associará cada instrução a um tempo de início respeitando as restrições de precedência entre as instruções devido às dependências de dados, conflitos de recursos e restrições temporais. A função que faz este mapeamento é descrita pela Definição 3.9.

Definição 3.9: Seja $\tau(v_i, t)$ uma função que associa cada instrução v_i no tempo t a um tipo de recurso necessário à sua execução. Seja a_r o número de recursos do tipo r no processador-alvo. Uma função chamada *schedule* $\varphi : V \rightarrow N$ mapeia cada instrução v_i a um tempo de início $\varphi(v_i)$ tal que:

- $\forall(v_i, v_j) \in E: \varphi(v_j) \geq \varphi(v_i) + w_{ij}$;

- $\forall t$ em $[\varphi(v_i), \varphi(v_i) + w_{ij}] : |\{v_k \in V: [\tau(v_k, t) = r] \wedge [t = \varphi(v_k)]\}| \leq a_r$.

Entretanto, é preciso verificar se o escalonamento produzido pela função *schedule* é factível quando levamos em conta as restrições temporais e de precedência. Em [MES 97] e [MIC 94] mostra-se que um algoritmo de cálculo de caminho mais longo em grafos, como o algoritmo de Bellman-Ford [COR 90], pode induzir um escalonamento desde que as restrições de recursos estejam corretamente codificadas no grafo.

Em [COR 90] e [MIC 94] mostra-se que o algoritmo de Bellman-Ford pode ser usado em grafos polares com apenas uma fonte (neste caso, o vértice *source*) para calcular o caminho mais longo desde a fonte a cada um dos outros vértices. Se o algoritmo não convergir em um número finito de iterações, então há um ciclo positivo[1] no grafo, o que significa que o conjunto de arestas representando restrições temporais é inconsistente. Esta é a chave da análise de factibilidade do escalonamento, a qual será descrita em maiores detalhes no Capítulo 4.

Primeiramente, deve-se definir a noção de distância entre dois vértices no grafo:

Definição 3.10: Seja um WPG(V, E, W) e um caminho p tal que um vértice v_i alcança um vértice v_j através de p. A distância de v_i a v_j através de p, escrita $d(v_i, v_j)$, é

[1] Um ciclo positivo ocorre quando o valor do caminho mais longo da fonte a um ou mais vértices cresce a cada nova iteração do algoritmo, nunca convergindo para um valor estável.

a soma dos pesos de todas as arestas que pertencem a p.

Definição 3.11: Seja um WPG(V, E, W) e dois vértices arbitrários v_i e v_j, a maior distância $\lambda(v_i, v_j)$ entre v_i e v_j é dada por:

$\forall p \mid v_i$ alcança v_j através de p : Max $d(v_i, v_j)$.

Seja λ_i uma notação simplificada para a maior distância entre o vértice v_0 (*source*) e um vértice arbitrário v_i, ou seja, $\lambda_i = \lambda(v_0, v_i)$. O valor de λ_i será determinado através de sucessivas estimativas reduzidas iterativamente. Seja λ_i^k a estimativa de λ_i na k-ésima iteração. O Algoritmo 1 descreve o cálculo dos caminhos mais longos.

Algoritmo 1 Cálculo dos caminhos mais longos

Procedure: `BellmanFord`(G(V, E, W))

```
1:  λ_0^1 = 0;
2:  for i = 1 to n do
3:      if ∃(v_0, v_i) then
4:          λ_i^1 = w_0i;
5:      else
6:          λ_i^1 = -∞;
7:      end if
8:  end for
9:  for j = 1 to n do
10:     for i = 1 to n do
11:         for k = 1 to n do
12:             if k ≠ i then
13:                 λ_i^{j+1} = MAX{λ_i^j, (λ_i^k + w_ki)};
14:             end if
15:         end for
16:     end for
17:     if λ_i^{j+1} == λ_i^j ∀i then
18:         return TRUE;
19:     end if
20: end for
21: return FALSE;
```

As linhas de 1 a 7 descrevem como as estimativas de distância máxima são inicializadas com os pesos das arestas adjacentes ao vértice *source*. As linhas de 9 a 16 mostram a relaxação dos valores das distâncias máximas. As linhas de 17 a 21 detectam a convergência (ou não convergência) das estimativas.

Capítulo 4

Otimizador de Código Redirecionável

Esta seção descreve a ferramenta que é o principal produto de trabalho desta dissertação, o otimizador de código redirecionável. Primeiramente, são explicitadas algumas restrições simplificadoras assumidas durante o desenvolvimento e, em seguida, são apresentadas a estrutura da ferramenta, seus principais componentes, as fases de processamento do código *assembly* a ser otimizado e os principais algoritmos utilizados.

4.1 Restrições simplificadoras

A grande variedade de arquiteturas disponíveis, desde processadores de propósito geral a ASIPs, com diferentes características quanto à arquitetura do conjunto de instruções, modos de endereçamento, tratamento de desvios etc., torna o desenvolvimento de uma ferramenta totalmente genérica quase inviável em termos de custos e tempo. Portanto, algumas hipóteses simplificadoras são assumidas a fim de restringir as características das arquiteturas-alvo suportadas e simplificar a geração do código otimizado, mas ao mesmo tempo prover uma ferramenta redirecionável e capaz de fazer otimizações de impacto em pontos críticos do código.

Restrição 1 - O escalonamento de código restringe-se ao reordenamento de instruções dentro de seus BBs originais.

Em outras palavras, a Restrição 1 descarta técnicas de escalonamento global de

código, tais como *code motion* [dS 98] e execução especulativa [PAT 06], o que simplifica significativamente a implementação do escalonador [1]. Uma discussão sobre o impacto de se relaxar a Restrição 1 é deixada para o Capítulo 6.

Note que a Restrição 1 não se refere à alocação de registradores, nem à análise de restrições de tempo real, que não se restringem às fronteiras dos BBs.

Também se assume algumas hipóteses que restrigem as arquiteturas-alvo suportadas pela ferramenta, conforme descrito abaixo.

Restrição 2 - A CPU-alvo é uma máquina *load/store*.

A vasta maioria das CPUs concebidas a partir de 1990 obedece à Restrição 2. Entretanto, como há ainda algumas máquinas remanescentes da era pré-RISC (tais como o processador Motorola Coldfire e o microcontrolador Intel 8051), umas poucas arquiteturas em uso em sistemas embarcados não são suportadas.

Restrição 3 - Todas as instruções ocupam cada estágio do *pipeline* durante um único ciclo de relógio.

Embora a maior parte das instruções inteiras de processadores modernos satisfaçam à Restrição 3, há instruções de ponto flutuante que nela não se enquadram. Esta é uma restrição que merece relaxamento em futuras extensões da ferramenta, pois é uma mera questão de implementação.

Restrição 4 - A cada ciclo de relógio, apenas uma instrução inicia a sua execução.

Em outras palavras, a Restrição 4 descarta arquiteturas com emissão múltipla como as superescalares e VLIW [PAT 06]. Esta é também uma restrição que merece prioridade em futuras extensões da ferramenta, pois muitas das CPUs usadas em sistemas embarcados apresentam emissão múltipla.

4.2 Estrutura da ferramenta

Esta seção descreve os principais componentes do otimizador de código.

[1] Técnicas globais exigem uma complexa análise da viabilidade de se mover instruções entre blocos e mecanismos complexos de compensação de código para preservar a semântica original do programa [dS 98].

- ***Parser* do Modelo**: é o componente responsável por extrair do modelo escrito em ArchC as informações específicas da arquitetura-alvo. Como visto na Seção 3.1, o modelo contém diversas informações das instruções da ISA que são necessárias à ferramenta, especialmente as suas sintaxes *assembly*, os campos fonte e destino e as latências entre instruções, assim como a declaração de quais registradores da arquitetura são de uso geral. Estas informações extraídas do modelo ArchC são armazenadas em estruturas de dados que serão mais tarde manipuladas por outros componentes da ferramenta, em especial o *Parser* do Código *Assembly* e o Gerador de Código (descritos a seguir).

- ***Parser* do Código *Assembly***: este componente lê o código *assembly* de entrada e, utilizando as informações extraídas pelo *parser* do modelo ArchC, gera o CFG e os WPGs correspondentes a cada bloco básico.

- **Escalonador e Analisador**: recebem como entrada cada WPG gerado pelo *parser* do código e buscam um escalonamento factível (se houver algum) para cada bloco básico, inserindo arestas no grafo para induzir um ordenamento das instruções.

- **Gerador de Código**: após o escalonador ter gerado os WPGs escalonados, este componente percorre cada grafo visitando os seus vértices na ordem induzida pelo escalonador e gera o código *assembly* otimizado. Durante este processo é feita também a alocação de registradores, onde os nomes simbólicos atribuídos pelo *parser* do código serão mapeados para nomes reais de registradores.

São apresentados a seguir o fluxo da ferramenta e o detalhamento de suas fases de processamento, mostrando como os componentes operam sobre o código *assembly*.

4.3 Fases de processamento

A Figura 4.1 mostra o fluxo de execução da ferramenta. A seguir, cada fase de processamento do fluxo será descrita, desde o código *assembly* original até a geração do código *assembly* otimizado.

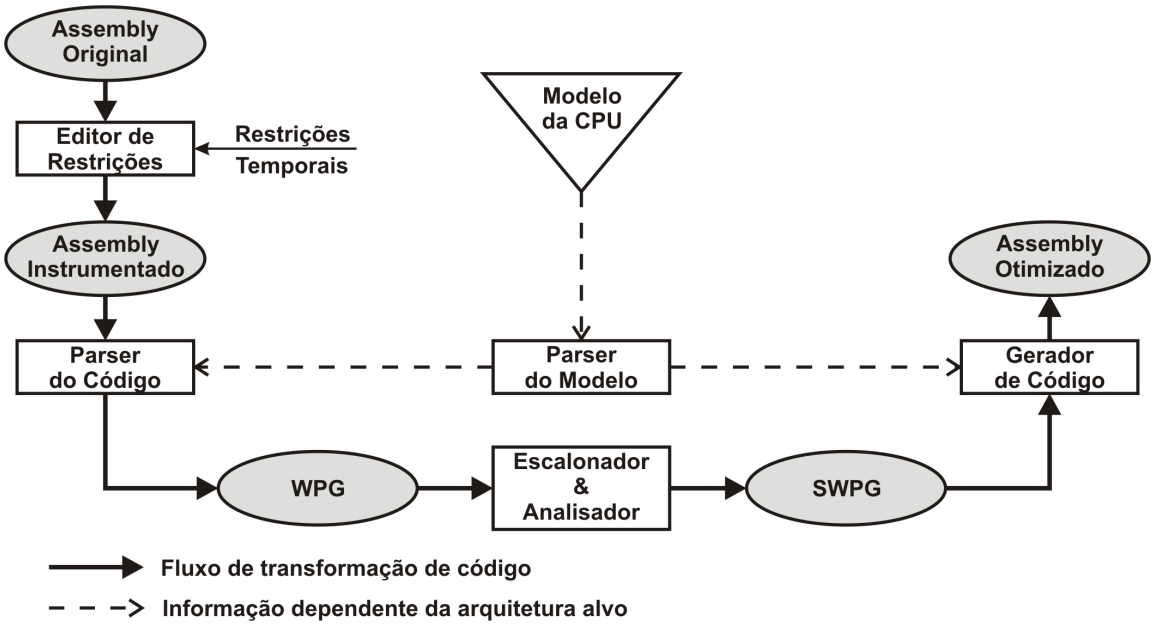

Figura 4.1: Fluxo de otimização do código

Fase 1: Inserção de restrições temporais

Entrada: código *assembly* original, gerado pelo compilador.

Saída: código *assembly* instrumentado (contendo as restrições temporais).

Procedimento: Como um compilador convencional não captura restrições temporais, elas são inseridas no código através de um editor, gerando assim o *assembly* instrumentado. As restrições são representadas por pares de pseudo-instruções que englobam trechos de código. Por exemplo, um par de pseudo-instruções [MIN k, rotulo] e [/MIN rotulo] representa um atraso mínimo de k ciclos de relógio ao trecho de código englobado por ele. De forma similar, atrasos exatos e máximos podem ser inseridos com os pares de pseudo-instruções [EXACT k, rotulo] e [/EXACT rotulo], e [MAX k, rotulo] e [/MAX rotulo].

A Figura 4.2 mostra um exemplo da transformação de um trecho de código *assembly* gerado por um compilador convencional para a arquitetura MIPS em código *assembly* instrumentado. Note que uma restrição de atraso máximo de 11 ciclos foi imposta ao segmento de código.

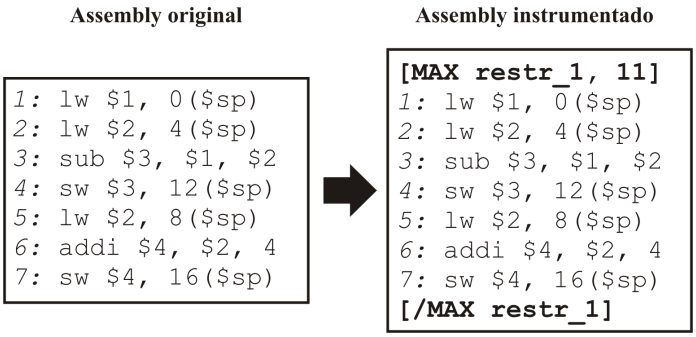

Figura 4.2: Transformação do assembly original em instrumentado

Fase 2: Análise do fluxo de dados

Entrada: código *assembly* instrumentado.

Saída: grafo ponderado de precedência (WPG).

Procedimento: Nesta fase, o *parser* do código *assembly* gera um grafo de fluxo de controle e os grafos ponderados de precedência que representam os blocos básicos.

O WPG gerado aqui contém um vértice para cada instrução do BB e arestas codificando tanto as restrições de precedência como as restrições temporais, de acordo com as definições apresentadas nas Seções 3.2 e 3.3. Para cada instrução cujo valor de destino é o valor fonte de outra instrução, o *parser* do código insere uma aresta no WPG. O peso desta aresta é determinado pela latência da instrução consumidora em relação à instrução produtora (conforme a Definição 3.1) mais um, valor que é lido do modelo do processador. O *parser* do código trata também as restrições temporais inseridas na fase anterior, inserindo arestas ponderadas para cada pseudo instrução [MIN k, rotulo], [MAX k, rotulo] e [EXACT k, rotulo] de acordo com as Definições 3.6, 3.7 e 3.8.

A Figura 4.3 mostra o WPG correspondente ao código mostrado na Figura 4.2. Note que a restrição temporal de atraso máximo foi convertida na aresta (*sink*, *source*) conforme a Definição 3.7.

Note que as arestas codificam as restrições de precedência devido às dependências de dados entre as instruções, enquanto que os seus pesos indicam quantos ciclos depois da instrução produtora começar sua execução a instrução consumidora pode iniciar a sua.

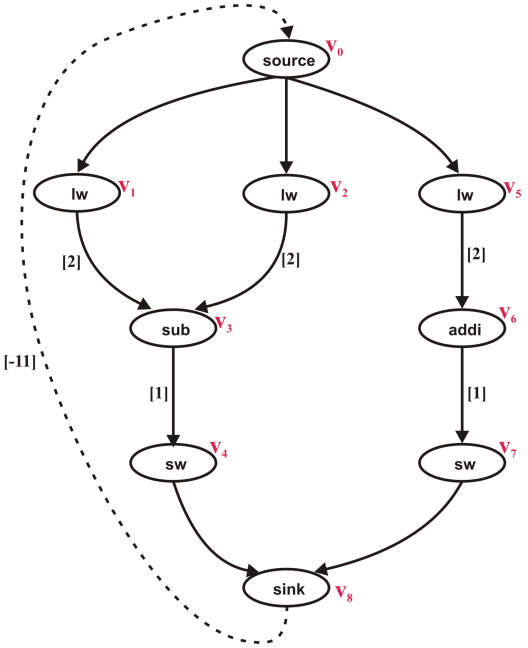

Figura 4.3: WPG gerado a partir do *assembly* instrumentado

Note também que o código da Figura 4.2 contém uma falsa dependência entre a instrução na linha 2 e a instrução na linha 5: elas escrevem no mesmo registrador e se o código for mantido assim a instrução 5 não poderia ser executada, por exemplo, antes da instrução 2 ou antes da 3, já que a instrução 3 consome o valor produzido pela 2 (neste caso, seria vantajoso executar a instrução 5 antes da instrução 3 para preencher a latência que existe entre as instruções 2 e 3). Mas, analisando-se o fluxo de dados, percebe-se que estas instruções na verdade são independentes, por isso o WPG é gerado de forma a eliminar tais dependências de nome entre as instruções e assim expor o paralelismo entre elas.

Fase 3: Escalonamento e análise de factibilidade

Entrada: WPG.

Saída: WPG escalonado (SWPG).

Procedimento: Cada vértice do WPG cujos predecessores já foram escalonados é marcado como tal. Cada vez que um vértice v_i é escalonado, insere-se um par de arestas (v_0, v_i) com w_{0i} = k e (v_i, v_0) com w_{i0} = -k para designar que v_i foi escalonado no k-ésimo

ciclo em relação ao vértice-fonte. Após a inserção de cada par de arestas, invoca-se o analisador de restrições temporais, cujo algoritmo foi discutido na Seção 3.3.3.

Como o algoritmo de escalonamento será apresentado somente na Seção 4.4, seu efeito é ilustrado através de um exemplo que usa o WPG da Figura 4.3 como ponto de partida. Primeiramente, o escalonador utiliza o algoritmo de Bellman-Ford para estimar a distância máxima entre o vértice *source* e cada um dos outros vértices do grafo. Isto equivale a uma estimativa inicial de distância máxima (λ_i) entre o vértice v_0 e um vértice arbitrário v_i. As estimativas iniciais resultantes são:

$\lambda_1 = 0$
$\lambda_2 = 0$
$\lambda_3 = 2$
$\lambda_4 = 3$
$\lambda_5 = 0$
$\lambda_6 = 2$
$\lambda_7 = 3$

Após esta inicialização, o escalonador verifica que, no início da execução (o ciclo "zero"), há três instruções que podem iniciar a sua execução (as instruções representadas pelos vértices v_1, v_2 e v_5), e escolhe uma delas para ser escalonada no ciclo atual. Esta escolha é feita por uma função de priorização [2]. Por simplicidade mas sem perda de generalidade, adota-se a ordem original das instruções no código *assembly* como função de prioridade. Portanto, a primeira instrução a ser escalonada é a representada pelo vértice v_1.

O escalonamento de v_1 é registrado no WPG através das arestas (v_0, v_1) com w_{01} = 0 e (v_1, v_0) com w_{10} = 0, conforme ilustra a Figura 4.4. As duas arestas inseridas forçam a instrução do vértice v_1 a iniciar sua execução no ciclo zero.

Após serem inseridas estas arestas, novamente é executado o Algoritmo 1 para verificar se o escalonamento atual é ainda factível. Os pesos das arestas adjacentes a *source*

[2] Uma função genérica de priorização foi concebida de forma a ser configurada para implementar diferentes heurísticas de priorização.

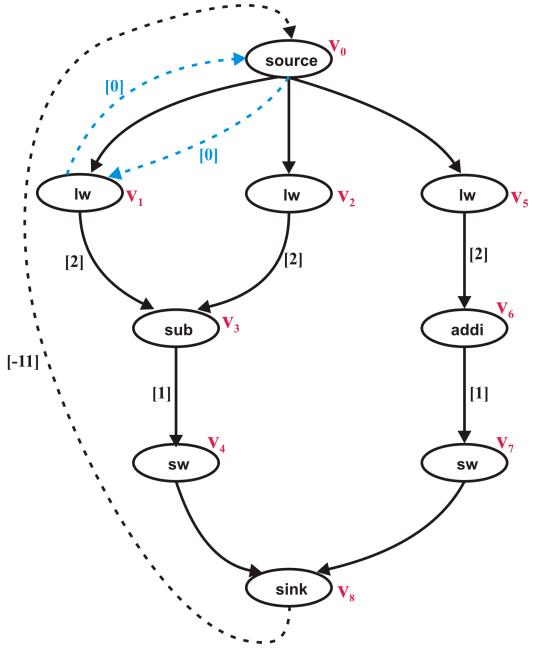

Figura 4.4: Escalonamento da primeira instrução

e que apontam para vértices não escalonados são incrementados em um, para que o Algoritmo 1 possa calcular corretamente as novas estimativas de distâncias máximas. O vetor de pesos atualizado ficará da seguinte forma:

$\lambda_1 = 0$ (peso fixo, vértice v_1 já escalonado)

$\lambda_2 = 1$

$\lambda_3 = 3$

$\lambda_4 = 4$

$\lambda_5 = 1$

$\lambda_6 = 3$

$\lambda_7 = 4$

Assim, no ciclo 1, dois vértices estão disponíveis para escalonamento, v_2 e v_5. Pela função de prioridade adotada, o vértice v_2 será escalonado no ciclo 1 através da inserção das arestas (v_0, v_2) com $w_{02} = 1$ e (v_2, v_0) com $w_{20} = -1$, conforme mostra a Figura 4.5.

Novamente o algoritmo de Bellman-Ford será invocado para analisar a factibilidade do escalonamento sendo gerado e o vetor de pesos será atualizado da seguinte maneira:

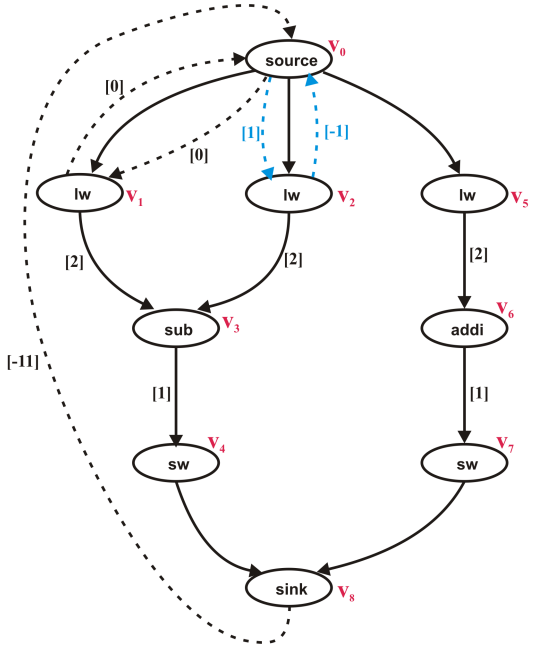

Figura 4.5: Escalonamento da segunda instrução

$\lambda_1 = 0$ (peso fixo, vértice v_1 já escalonado)

$\lambda_2 = 1$ (peso fixo, vértice v_2 já escalonado)

$\lambda_3 = 3$

$\lambda_4 = 4$

$\lambda_5 = 2$

$\lambda_6 = 4$

$\lambda_7 = 5$

Observe que os pesos dos caminhos que passam pelos vértices v_3 e v_4 não são incrementados em um. Isto ocorre pois os caminhos mais longos do vértice *source* até eles passam somente pelos vértices v_1 e v_2, que já foram escalonados (ou seja, os valores de λ_1 e λ_2 não serão mais alterados).

No ciclo 2 apenas o vértice v_5 estará disponível para ser escalonado. Os passos descritos anteriormente serão repetidos a cada ciclo até que todas as instruções tenham sido escalonadas (ou até que fosse detectada a infactibilidade do escalonamento). O novo WPG, com as arestas que induzem a ordem das instruções, é o WPG escalonado (SWPG).

O SWPG final pode ser visto na Figura 4.6. Para facilitar a visualização, o grafo da Figura 4.6 contém apenas as arestas que codificam as restrições temporais e as decisões do escalonador (as arestas representando precedência foram removidas).

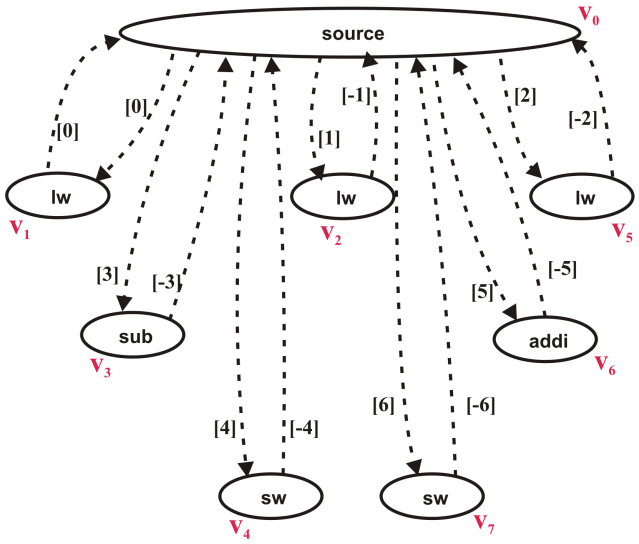

Figura 4.6: WPG escalonado (SWPG)

Note que os atrasos exatos impostos a cada um dos vértices induzem a ordem linear $(v_0, v_1, v_5, v_3, v_4, v_6, v_7)$.

Fase 4: Geração de código e alocação de registradores

Entrada: SWPG.

Saída: Código *assembly* otimizado.

Procedimento: O gerador de código percorre o SWPG na ordem induzida pelas arestas inseridas durante o escalonamento e, utilizando as informações sintáticas extraídas pelo *parser* do modelo ArchC, gera o novo código *assembly*. O código gerado não contém as pseudo-instruções do *assembly* instrumentado e é isento de dependências de nome.

Nesta fase faz-se também a alocação de registradores [AHO 95]. Este processo consiste em realizar o cálculo do tempo de vida de cada valor e construir um grafo de conflito. Obtido este grafo, executa-se um algoritmo de coloração de vértices (denominado de algoritmo da borda esquerda) para determinar o número mínimo de cores (registradores)

necessárias. Em seguida, a cada cor distinta é associado um registrador real, e se o número de cores exceder o número de registradores disponíveis serão geradas instruções que armazenam na memória os valores não mapeados em registradores reais. A Seção 4.5 descreverá o algoritmo de alocação de registradores.

A Figura 4.7 mostra o código *assembly* gerado a partir da ordem linear induzida no SWPG da Figura 4.6.

```
1: lw $1, 0($sp)
2: lw $2, 4($sp)
3: lw $3, 8($sp)
4: sub $1, $1, $2
5: sw $1, 12($sp)
6: addi $1, $3, 4
7: sw $1, 16($sp)
```

Figura 4.7: Código otimizado

Observe que a latência de 1 ciclo que existe entre a segunda instrução `lw` e a instrução `sub` foi preenchida pela terceira instrução `lw`, o que também eliminou a latência de 1 ciclo entre esta instrução e a instrução `addi`, neste caso diminuindo em dois ciclos o tempo de execução deste trecho.

Note que, nesse exemplo, a restrição temporal de atraso máximo de 11 ciclos foi imposta dentro dos limites do bloco básico. Entretanto, a modelagem de análise de restrições temporais proposta não se restringe apenas a blocos básicos individuais. A ferramenta dá suporte à inserção de restrições que englobam múltiplos BBs, para os casos em que o trecho de código a ser escalonado inclui desvios. Neste caso, as restrições deverão ser decompostas entre os BBs de acordo com os possíveis fluxos de execução descritos pelo CFG.

Como descrito anteriormente, cada vértice do CFG será visitado pelo escalonador, que fará o escalonamento e análise de factibilidade do BB que o vértice representa. Como o escalonador visita os vértices na ordem imposta pelas arestas do CFG (seguindo assim os possíveis fluxos de execução do código), a idéia é fazer a decomposição da restrição temporal ao longo deste processo, impondo a restrição ao primeiro bloco e propagando-a

aos blocos sucessores conforme seja calculado o número de ciclos necessários ao escalonamento de cada BB.

Para ilustrar o funcionamento desta técnica, suponha que uma restrição temporal fosse imposta ao código *assembly* apresentado na Figura 3.9 (Seção 3.2.1), transformando-o em *assembly* instrumentado, conforme mostra a Figura 4.8.

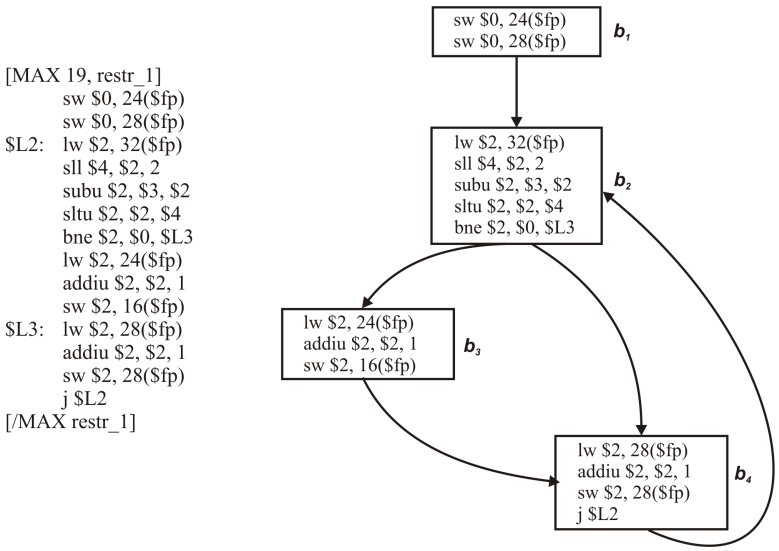

Figura 4.8: Impondo restrição temporal ao código da Figura 3.9

Suponhamos que os vértices do CFG sejam visitados seguindo a ordem b_1 -> b_2 -> b_3 -> b_4. Como o primeiro bloco a ser visitado será b_1 e a restrição "restr_1" engloba todos os BBs, antes de escalonar este bloco codificaremos nele a restrição do tipo MAX com o seu valor inteiro, ou seja, 19 ciclos, conforme mostra a Figura 4.9.

Após o bloco b_1 ter sido escalonado, conclui-se que para ele são necessários 2 ciclos. Portanto, a restrição "restr_1" deverá ser propagada a todos os sucessores do BB atual (no caso, apenas o bloco b_2) com o seu valor original menos o número de ciclos necessários para o escalonamento do BB atual. Neste caso, uma restrição do tipo MAX de 17 ciclos será imposta ao bloco b_2, conforme ilustrado na Figura 4.10.

De forma similar, o bloco b_2 é escalonado e a restrição deve ser novamente propagada aos seus sucessores. Digamos que o número de ciclos calculado pelo escalonador para b_2 foi 7, portanto a restrição imposta ao próximo BB a ser visitado (pela ordem, b_3)

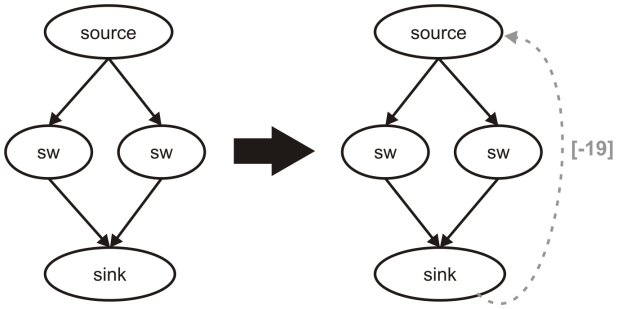

Figura 4.9: Impondo restrição “restr_1” ao bloco b_1

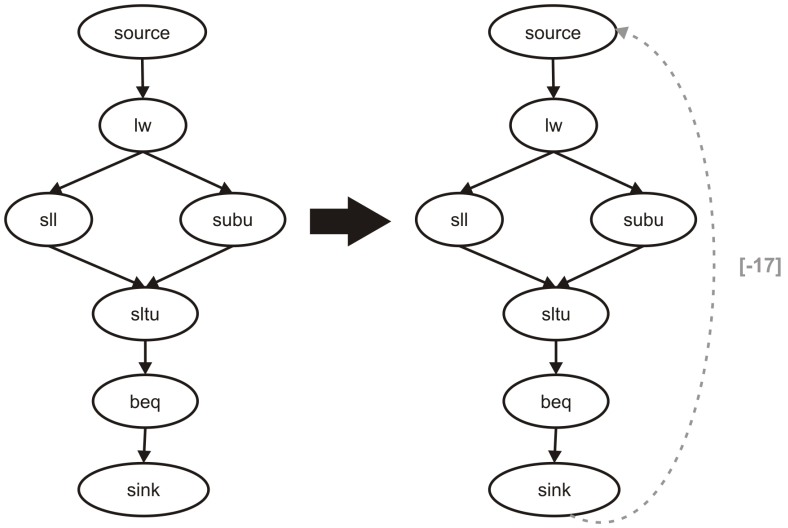

Figura 4.10: Propagando a restrição “restr_1” de b_1 para b_2

será de 10 ciclos. Finalmente, a restrição deve ser propagada de b_3 para b_4 de acordo com o número de ciclos calculado para b_3.

Note que, inicialmente, a restrição temporal não é muito rígida, mas vai ficando cada vez mais apertada conforme seguimos o fluxo de execução do código e os BBs que compões este fluxo vão sendo escalonados. Desta forma, assim como no escalonamento dentro do BB, a infactibilidade pode vir a ser detectada mais cedo, caso a restrição seja bastante severa já de início.

Observe ainda que o CFG da Figura 4.8 contém dois fluxos possíveis de execução:

b_1 -> b_2 -> b_3 -> b_4 e b_1 -> b_2 -> b_4. Desta forma é possível que, para uma certa restrição temporal imposta a todo o código, o escalonamento fosse factível quando apenas o fluxo mais curto é levado em consideração, mas não fosse factível levando-se em consideração o fluxo mais longo. Utilizando-se a técnica apresentada acima, a análise de restrições temporais levará em conta sempre o fluxo de execução mais longo (ou seja, o pior caso) para fazer a verificação de factibilidade.

4.4 Algoritmo de escalonamento de código

O Algoritmo 2 descreve o escalonamento de um WPG. Este algoritmo integra as funcionalidades do escalonador e analisador de restrições, através da invocação do Algoritmo 1 (linhas 1 e 22).

O algoritmo consiste de um procedimento principal (`Schedule`) e procedimentos auxiliares (`Infeasible`, `ScheduleStep` e `FindAvailableInstructions`) por ele invocados , os quais serão descritos a seguir.

A análise de factibilidade do escalonamento é feita pelo procedimento `Infeasible`, o qual invoca o algoritmo de Bellman-Ford para analisar se as restrições temporais atualmente codificadas no WPG são consistentes. Caso o algoritmo retorne *FALSE*, o procedimento `Infeasible` retornará *TRUE*.

O procedimento `ScheduleStep` invoca o procedimento `SelectInstruction` (linhas 2 e 11) para selecionar a instrução com a maior prioridade dentre as disponíveis para escalonamento no ciclo t. Apesar de a ferramenta aqui descrita conter um mecanismo de escolha de qual instrução será escalonada similar ao *list scheduling* [MIC 94], a função que define a prioridade é genérica e pode ser alterada para capturar diferentes heurísticas. Quando uma instrução v_i é escolhida, ela é associada ao ciclo t, ou seja, são inseridas duas arestas ponderadas no WPG: a aresta (v_0, v_i) com peso t, e a aresta (v_i, v_0) com peso -t (linhas 4 a 7). Isto significa que esta instrução deve iniciar a sua execução exatamente t ciclos após a referência inicial de tempo (representada pelo vértice v_0, "source"), de acordo com a Definição 3.8 da Seção 3.3. A cada decisão tomada pelo escalonador, o procedimento `Infeasible` (linha 8) é chamado para analisar se as restrições temporais

Algoritmo 2 Escalonamento de código

```
Procedure: Infeasible(WPG(V, E, W))
 1: return (¬ BellmanFord(WPG(V, E, W))); // Invoca o Algoritmo 1
Procedure: ScheduleStep(t, A)
 2: v_i = SelectInstruction(A);
 3: while v_i ≠ none do
 4:    E = E (v_0, v_i);
 5:    w_0i = +t;
 6:    E = E (v_i, v_0);
 7:    w_i0= −t;
 8:    if Infeasible(WPG(V, E, W)) == TRUE then
 9:       return (FALSE);
10:    end if
11:    v_i = SelectInstruction(A);
12: end while
13: return TRUE;
Procedure: FindAvailableInstructions(t)
14: A = ∅;
15: for k = 1 to n do
16:    if (v_0, v_k) == t then
17:       A = A ∪ v_k;
18:    end if
19: end for
20: return (A);
Procedure: Schedule( )
21: t = 0;
22: W = BellmanFord(WPG(V, E, W)); // Inicializa o vetor de pesos invocando o Algoritmo 1
23: A = FindAvailableInstructions (t);
24: while A ≠ ∅ do
25:    if ScheduleStep(t, A) == FALSE then
26:       return(∞);
27:    end if
28:    t = t + 1;
29:    A = FindAvailableInstructions (t);
30: end while
31: return (t);
```

atualmente codificadas no WPG são consistentes.

Já o procedimento `FindAvailableInstructions` constrói o conjunto de instruções que estão disponíveis para serem escalonadas em um dado ciclo t, ou seja, aquelas cuja distância máxima ao vértice *source* é igual a t (significando que todos os seus operandos estão prontos para serem consumidos neste ciclo).

No procedimento principal `Schedule` (linhas 21 a 31), inicialmente, as estimativas de distância máxima do vértice *source* para todos os outros vértices do grafo são calculadas utilizando-se o algoritmo de Bellman-Ford (linha 22). Em seguida, dado um ciclo de relógio t, o procedimento `FindAvailableInstructions` (linha 23) é invocado e retorna o conjunto de instruções que estão disponíveis para serem escalonadas

neste ciclo. Construído o conjunto de instruções inicialmente disponíveis, invoca-se o procedimento `ScheduleStep` (linha 25) durante sucessivas iterações até que todas as instruções tenham sido escalonadas de acordo com a Definição 3.9 da Seção 3.3, ou até que seja detectada a infactibilidade do escalonamento.

4.5 Algoritmo de alocação de registradores

O mecanismo de alocação de registradores subjacente ao otimizador foi objeto de trabalho de pesquisa correlato em cuja monografia [SAN 07] pode ser encontrado o algoritmo de alocação de registradores utilizado.

4.6 Validação experimental

A seguir será apresentada a metodologia utilizada para a validação da ferramenta descrita neste capítulo. Serão apresentados resultados para dois conjuntos diferentes de *benchmarks* e, em seguida, serão explicados os critérios usados para verificar se os resultados gerados pela ferramenta estão corretos.

Para demonstrar a redirecionabilidade da ferramenta, foram adotadas três CPUs-alvo: MIPS R2000, PowerPC 405 e SPARCV8. O compilador adotado foi o gcc [GCC 03].

Todos os experimentos foram feitos em um computador com processador Pentium 4 rodando a 3 GHz, com 1 GB de memória principal.

4.6.1 Experimentos sem código condicional

O primeiro conjunto de experimentos foi realizado utilizando-se trechos de código que não contêm desvios. Estes trechos foram extraídos do conjunto de *benchmarks* Mibench [MIB 06]. Buscaram-se blocos básicos pertencentes a partes críticas dos programas, como corpos de laços repetidos muitas vezes.

4.6.1.1 Configuração experimental

A Tabela 4.1 mostra a caracterização dos *benchmarks* para cada CPU-alvo. A primeira coluna mostra o nome dos segmentos de código, enquanto que a segunda indica os programas de onde eles foram extraídos. Para cada CPU-alvo há um par de colunas mostrando o tamanho do WPG resultante em termos de número de vértices (|V|) e de arestas (|E|).

Segmento	**Benchmark**	*MIPS*		*PowerPC*		*SPARC*	
		|V|	|E|	|V|	|E|	|V|	|E|
Isqrt	basicmath	22	52	20	63	21	51
Bitarray	bitcount	12	29	26	70	25	67
Bitstrng	bitcount	20	54	22	64	19	55
Qsort	qsort	67	137	43	97	70	154
Jdcolor1	jpeg	86	197	52	130	83	198
Jdcolor2	jpeg	58	126	58	150	51	122
Rdbmp1	jpeg	35	90	36	95	36	93
Rdbmp2	jpeg	83	185	110	340	110	278
SHA	sha	41	98	37	94	41	96
Timing	adpcm	19	42	12	28	22	53

Tabela 4.1: Caracterização dos *benchmarks* sem código condicional

4.6.1.2 Resultados

Para demonstrar como o escalonador lida com restrições temporais efetivamente, cada segmento de código foi submetido a diferentes restrições, desde a mais rígida até a mais relaxada, e foi calculada a porcentagem total de soluções factíveis para todo o conjunto de segmentos usado nos experimentos.

Para gerar tais restrições, primeiramente adotou-se uma restrição temporal base para cada segmento de código, a qual foi progressivamente relaxada. Esta base, chamada ϵ, assume que a cada ciclo uma nova instrução inicia a sua execução, ignorando assim possíveis conflitos que possam causar paradas no *pipeline*. Note que esta é uma estimativa otimista e que provavelmente acarretará em um escalonamento infactível, assegurando que o experimento inicie com restrições rígidas.

A Figura 4.11 mostra a porcentagem de soluções factíveis sob restrições de atraso máximo de 1,1ϵ, 1,2ϵ e 1,3ϵ.

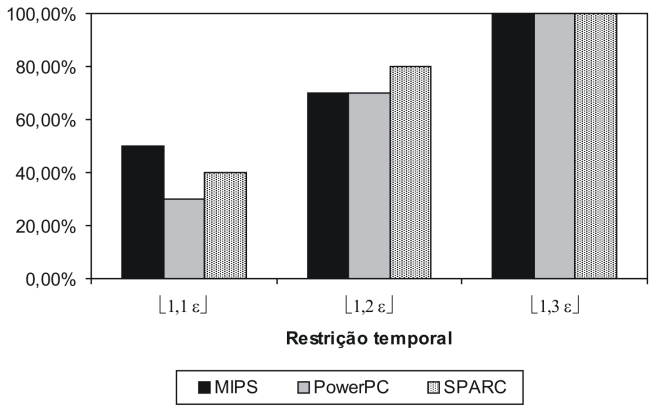

Figura 4.11: Porcentagem de soluções factíveis

Observe que, com uma variação de 30% em relação à base ϵ, todos os segmentos de código satisfazem as restrições temporais. No entanto, com uma variação de 10%, metade dos segmentos satisfazem as restrições para o MIPS, um terço para o PowerPC e aproximadamente metade para o SPARC. Isto demonstra a dificuldade em se satisfazerem restrições temporais sem se utilizar um escalonador que delas esteja ciente.

Os fluxos de compiladores desenvolvidos para gerarem código para sistemas embarcados geralmente toleram tempos maiores de otimização do que fluxos de compiladores convencionais. No entanto, como a ferramenta aqui apresentada executa um algoritmo de cálculo de caminho mais longo para cada instrução escalonada, o *overhead* introduzido devido à análise de restrições temporais deve ser verificado.

As Figuras 4.12, 4.13 e 4.14 mostram o tempo de execução do escalonador (expresso em segundos na escala da direita) correlacionado ao tamanho do WPG (expresso em número de vértices e arestas na escala da esquerda).

Observe que para todas as CPUs-alvo, em média, o crescimento do número de vértices parece estabelecer um limite superior para o crescimento do tempo de execução. Isto indica que a técnica de análise de restrições temporais aqui descrita tem baixo *overhead*.

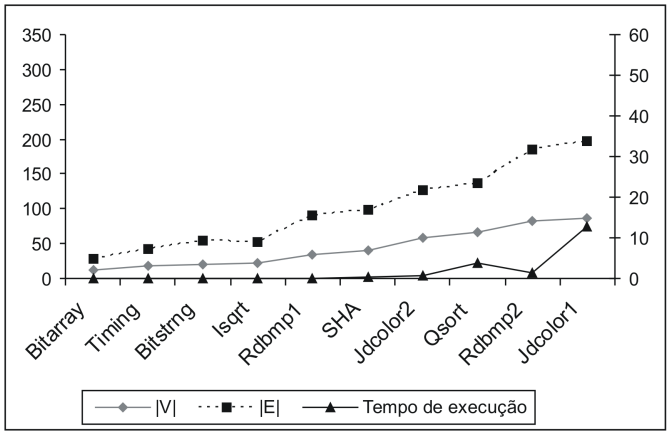

Figura 4.12: Correlação do tamanho do WPG e tempo de execução (MIPS)

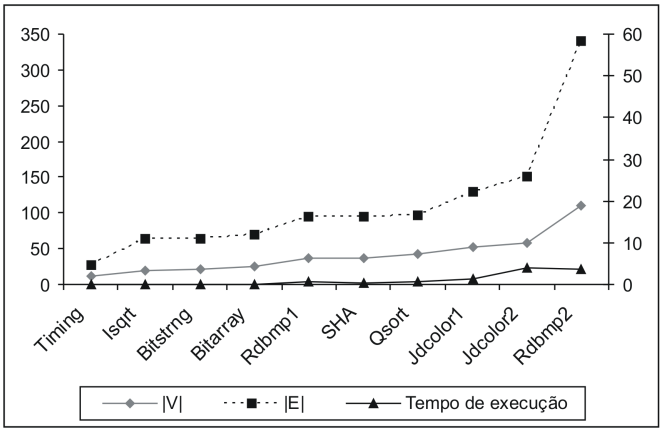

Figura 4.13: Correlação do tamanho do WPG e tempo de execução (PowerPC)

Ao contrário dos compiladores convencionais, a técnica descrita neste trabalho explora as restrições temporais para fins de otimização. Otimizações que um compilador convencional possivelmente deixa passar são induzidas para garantir que as restrições temporais sejam satisfeitas em trechos críticos de código.

Para verificar como as restrições temporais são exploradas, comparou-se o tamanho do escalonamento de um segmento de código (sem restrições temporais) com o tamanho do escalonamento do segmento otimizado equivalente obtido sob restrições temporais

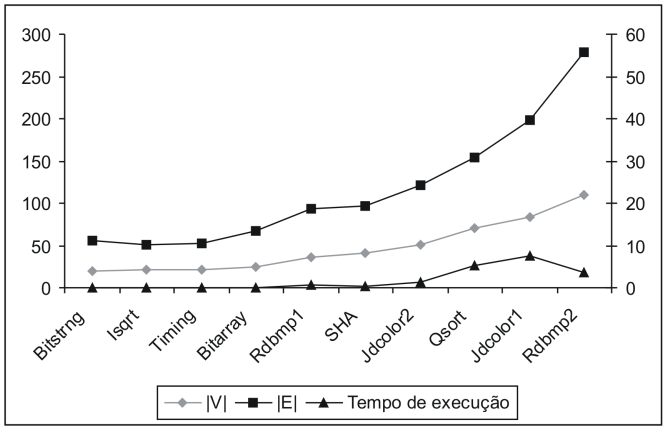

Figura 4.14: Correlação do tamanho do WPG e tempo de execução (SPARC)

rígidas. Nos experimentos, um atraso máximo de 1,3ϵ foi imposto a cada trecho de código.

A Figura 4.15 plota a razão entre os comprimentos dos escalonamentos dos códigos originais e otimizados para cada CPU-alvo, ou seja, a aceleração obtida na execução no código. Os *benchmarks* são mostrados em ordem crescente de tamanho da esquerda para a direita.

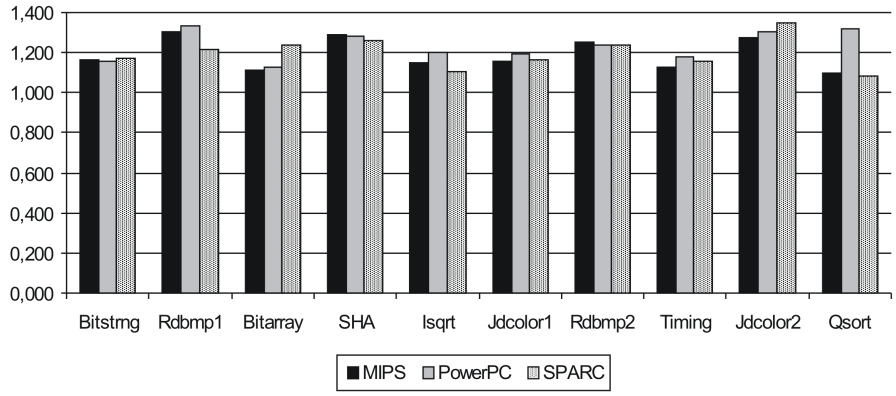

Figura 4.15: Aceleração da execução do código após a otimização

A aceleração média foi de 1,18 para os cinco menores segmentos de código e 1,23 para os cinco maiores. O aumento foi um pouco maior para os *benchmarks* com BBs

maiores, pois estes provêm maiores oportunidades de otimização.

Note que o *assembly* original é gerado por um compilador convencional, o qual introduz algumas otimizações. Assim, a aceleração foi medida em relação a trechos de código pré-otimizado. Portanto, os resultados experimentais indicam que o compilador realmente deixou passar algumas oportunidades de otimização, o que pode ser explicado pelo fato de as restrições temporais não serem capturadas em suas heurísticas de escalonamento. A ferramenta descrita aqui converte as restrições temporais em restrições de precedência, as quais invalidam algumas instruções como candidatas já que a seleção destas levaria a um escalonamento infactível. Ou seja, a ferramenta sobrescreve as heurísticas originais do compilador para favorecer a satisfação das restrições temporais.

Como acelerações significativas em relação a código já pré-otimizado foram alcançadas com baixo *overhead*, há evidências suficientes de que é vantajosa a otimização e análise de restrições temporais feitas após a compilação.

4.6.2 Experimentos com código condicional

Nesta segunda etapa de experimentação, os trechos de código utilizados contêm desvios. Novamente os trechos foram extraídos do conjunto de *benchmarks* Mibench e, assim como nos experimentos sem código condicional, buscaram-se trechos executados com freqüência.

4.6.2.1 Configuração experimental

A Tabela 4.2 mostra a caracterização dos *benchmarks* para cada CPU-alvo. Similarmente à Tabela 4.1, as colunas indicam os nomes dos *benchmarks*, os programas de onde foram extraídos, o número de vértices e o número de arestas do WPG resultante. Uma nova coluna foi adicionada para indicar o número de blocos básicos (|BB|) de cada *benchmark* para cada CPU-alvo.

Segmento	Benchmark	*MIPS*			*PowerPC*			*SPARC*		
		|V|	|E|	|BB|	|V|	|E|	|BB|	|V|	|E|	|BB|
Isqrt	basicmath	57	151	8	61	184	8	50	138	8
Bitarray	bitcount	78	199	10	45	117	4	47	114	4
Bitstrng	bitcount	68	175	8	73	221	8	66	179	8
Jdcolor	jpeg	130	328	6	135	375	6	125	322	6
Rdbmp	jpeg	56	147	4	59	163	4	56	151	4
Sq	ispell	82	193	13	84	236	13	89	210	13
Bmhasrch	stringsearch	118	254	18	129	360	18	133	294	18
SHATransform	sha	288	694	19	283	755	19	286	711	19

Tabela 4.2: Caracterização dos *benchmarks* com código condicional

4.6.2.2 Resultados

Como nesta etapa dos experimentos os *benchmarks* contém código condicional, nem sempre todas as instruções de um segmento de código são executadas em todas as vezes em que ele é executado.

Conforme exposto na Seção 4.3, a técnica de análise de restrições temporais aqui descrita leva em consideração sempre o fluxo mais longo de execução. Desta forma, a aplicação de restrições temporais aos *benchmarks* com código condicional deve ser feita de forma diferente. Nos *benchmarks* sem código condicional, a estimativa base ϵ levava em conta o número total de instruções, mas neste caso a estimativa inicial deve levar em conta o número de instruções no fluxo mais longo de execução.

A Figura 4.16 mostra a porcentagem de soluções factíveis sob restrições de atraso máximo de 1,1ϵ, 1,2ϵ e 1,3ϵ.

Observe que, com uma variação de 30% em relação à base ϵ, todos os segmentos de código satisfazem as restrições temporais. No entanto, com uma variação de 10%, cerca de um terço satisfazem as restrições para o MIPS, aproximadamente dois terços para o PowerPC e metade para o SPARC.

O uso de *benchmarks* com código condicional requer também que a verificação do *overhead* introduzido devido à análise de restrições temporais seja feita de forma diferente. Como o maior *overhead* está relacionado à execução do algoritmo de Bellman-Ford sucessivas vezes durante o escalonamento de cada bloco básico, a comparação do tempo

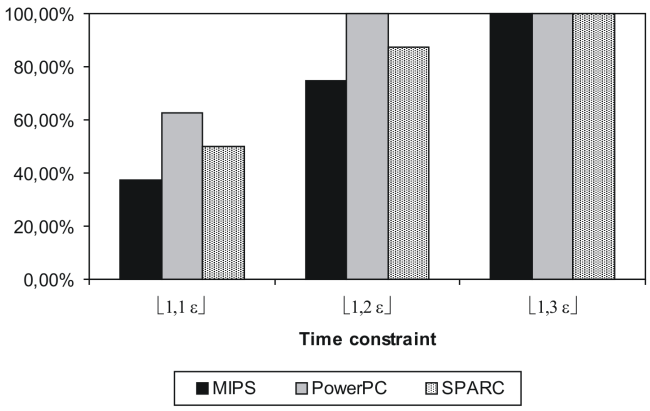

Figura 4.16: Porcentagem de soluções factíveis

de execução do escalonador deve ser feita em relação ao tamanho médio dos blocos básicos do *benchmark*.

As Figuras 4.17, 4.18 e 4.19 mostram o tempo de execução do escalonador (expresso em segundos na escala da direita) correlacionado ao tamanho médio dos WPGs que representam os blocos básicos (expresso pelos números de vértices e de arestas divididos pelo número de BBs, na escala da esquerda).

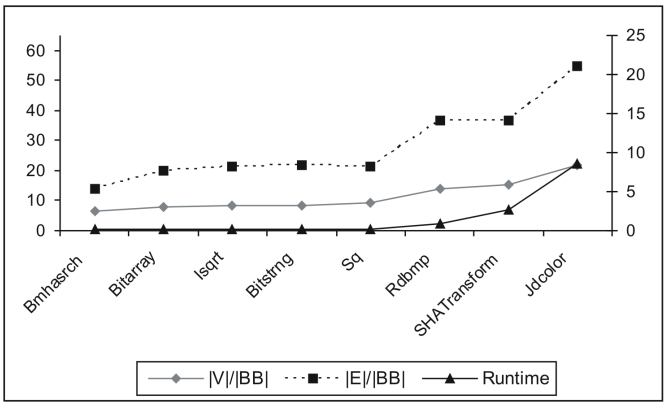

Figura 4.17: Correlação do tamanho médio do WPG e tempo de execução (MIPS)

Novamente os resultados indicam que o *overhead* é aceitável, já que o crescimento do tamanho médio dos blocos básicos parece estabelecer um limite superior para o cres-

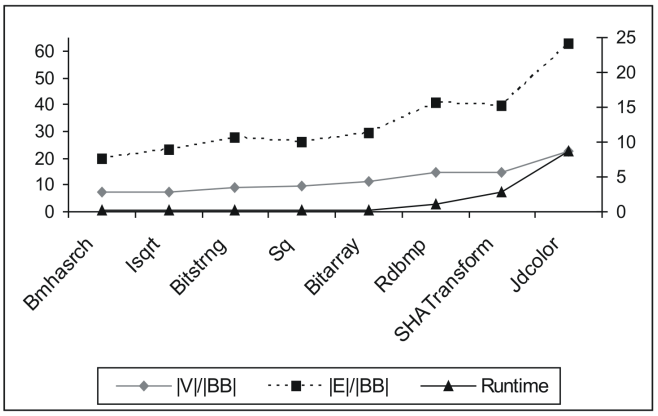

Figura 4.18: Correlação do tamanho médio do WPG e tempo de execução (PowerPC)

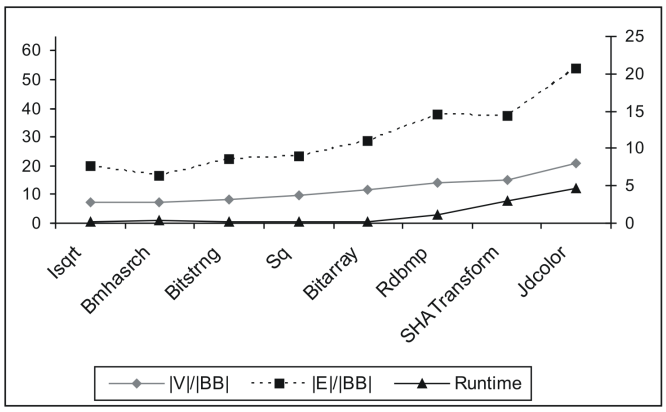

Figura 4.19: Correlação do tamanho médio do WPG e tempo de execução (SPARC)

cimento do tempo de execução.

A Figura 4.20 plota a razão entre os comprimentos dos escalonamentos dos fluxos mais longos de execução dos códigos originais e otimizados para cada CPU-alvo, ou seja, a aceleração obtida entre dois cenários de pior caso de execução. Os *benchmarks* são mostrados em ordem crescente de tamanho médio dos blocos básicos da esquerda para a direita.

A aceleração média foi de 1,13 para os quatro menores segmentos de código e 1,25 para os quatro maiores. Assim como nos experimentos sem código condicional, o

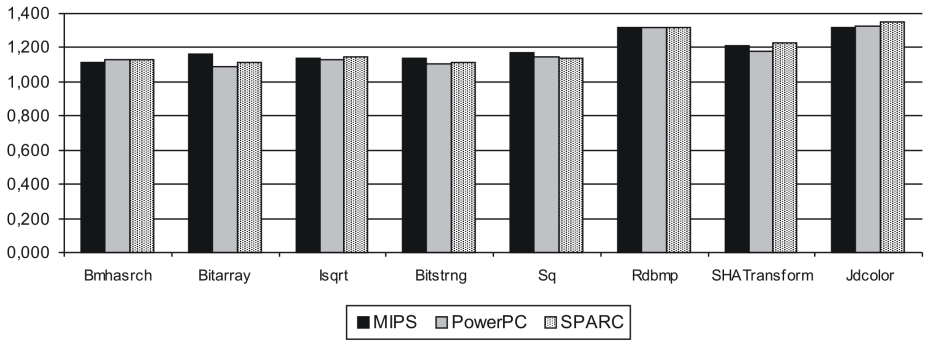

Figura 4.20: Aceleração da execução do código após a otimização

aumento foi um pouco maior para os *benchmarks* com BBs maiores devido ao fato de proverem maiores oportunidades de otimização, e acelerações acima de 1,3 foram obtidas em relação a código já pré-otimizado.

4.6.3 Critérios de validação

O foco principal dos experimentos relatados nas Seções 4.6.1 e 4.6.2 foi avaliar o impacto das otimizações no código *assembly* juntamente com o *overhead* necessário à execução da ferramenta.

No entanto, para completar-se a validação do protótipo da ferramenta é necessário ainda verificar se a semântica do código original é mantida, ou seja, se o código otimizado gerado pela ferramenta é equivalente ao código original gerado pelo compilador. Uma forma de se fazer tal verificação é executar os dois trechos de código e comparar os resultados. Como a ferramenta é baseada em uma descrição do processador feita através do ArchC, esta mesma descrição pode ser usada para gerar um simulador onde esta validação pode ser feita.

A partir de cada descrição em ArchC foi gerado o simulador correspondente ao processador-alvo desejado, utilizando-se a opção para que sejam impressos na tela o conteúdo da memória e dos registradores. Os resultados gerados na execução do código original (gerado pelo compilador) foram iguais àqueles gerados na execução do código

otimizado, verificando-se assim que a semântica do código foi mantida.

Capítulo 5

O Papel da Otimização Redirecionável na Tradução Binária

Este capítulo propõe uma técnica para tradução binária, a qual é baseada na geração automática de ferramentas a partir de modelos de CPUs descritos através de uma ADL. A motivação, a proposta e o estudo experimental de viabilidade dessa técnica foram realizados cooperativamente pelo autor e dois outros mestrandos cujas dissertações abordam tópicos também relevantes para a tradução binária [CAS 07] [dOS 07]. Por essa razão, o texto das Seções 5.1, 5.2 e 5.4 é deliberadamente comum às três dissertações de mestrado. Entretanto, como a implementação do protótipo contou com contribuições distintas e complementares, a Seção 5.3 descreve a contribuição específica do autor para esse trabalho conjunto.

5.1 Motivação

Como discutido no Capítulo 1, durante a exploração do espaço de projeto de um SoC pode-se ter que avaliar o impacto de várias CPUs alternativas até que os requisitos sejam satisfeitos. Isso requer a rápida geração de código executável para cada uma das CPUs exploradas, os quais podem ser obtidos de três maneiras distintas:

- Alternativa 1 - Disponibilidade de compiladores convencionais portados para cada

uma das CPUs candidatas;

- Alternativa 2 - Disponibilidade de um compilador redirecionável;
- Alternativa 3 - Disponibilidade de um compilador convencional para uma das CPUs e de um tradutor binário capaz de gerar código executável para as demais CPUs.

A Alternativa 1 restringe o espaço de soluções exploradas ao uso de CPUs tradicionais, cujo uso intensivo justificou o desenvolvimento de compiladores próprios.

A Alternativa 2 é a mais genérica, mas requer o uso de compiladores redirecionáveis, os quais são invariavelmente proprietários (como é o caso do compilador LisaTek [COW 07]) e cujas licenças são caras já que sua oferta no mercado é bastante pequena.

A Alternativa 3 tem a vantagem de requerer apenas um compilador convencional portado para uma única arquitetura, cuja licença pode ser pública (como a do gcc), desde que se disponha de um tradutor binário para redirecionar o código para as demais arquiteturas a serem exploradas.

Em princípio, um tradutor binário poderia ser obtido através do encadeamento de geradores de utilitários binários (Seção 5.2). Ora, a ADL ArchC provê geradores de utilitários binários sob licença GPL. Assim, o desenvolvimento de um tal tradutor resultaria numa solução de baixo custo para a exploração de CPUs. É de se esperar que o esforço de desenvolvimento de um tradutor binário estático - restrito às necessidades de sistemas embarcados - seja inferior ao requerido para se desenvolver um compilador redirecionável.

A principal dificuldade é a de garantir que a qualidade do código traduzido não seja muito inferior à obtida através de um compilador. Supondo que o compilador tenha realizado otimizações independentes de arquitetura antes de gerar o código executável sob tradução, é de se esperar que o código traduzido tenha qualidade similar, desde que o tradutor suporte otimizações dependentes de arquitetura utilizados em compiladores-otimizadores contemporâneos, tais como seleção de instruções, escalonamento de código e alocação de registradores.

A partir da revisão de literatura realizada, não é de conhecimento do autor que exista algum trabalho de pesquisa similar utilizando geração automática de ferramentas

a partir de ADL para fins de tradução binária no contexto de exploração do espaço de projeto. Este fato, aliado à infra-estrutura disponibilizada pelo pacote ArchC, motivou a investigação da viabilidade dessa alternativa, como reportado nas próximas seções.

5.2 Proposta de estrutura de um tradutor binário

O tradutor binário proposto neste capítulo realiza a tradução estaticamente. Uma possível estrutura de um tradutor binário estático é ilustrada na Figura 5.1.

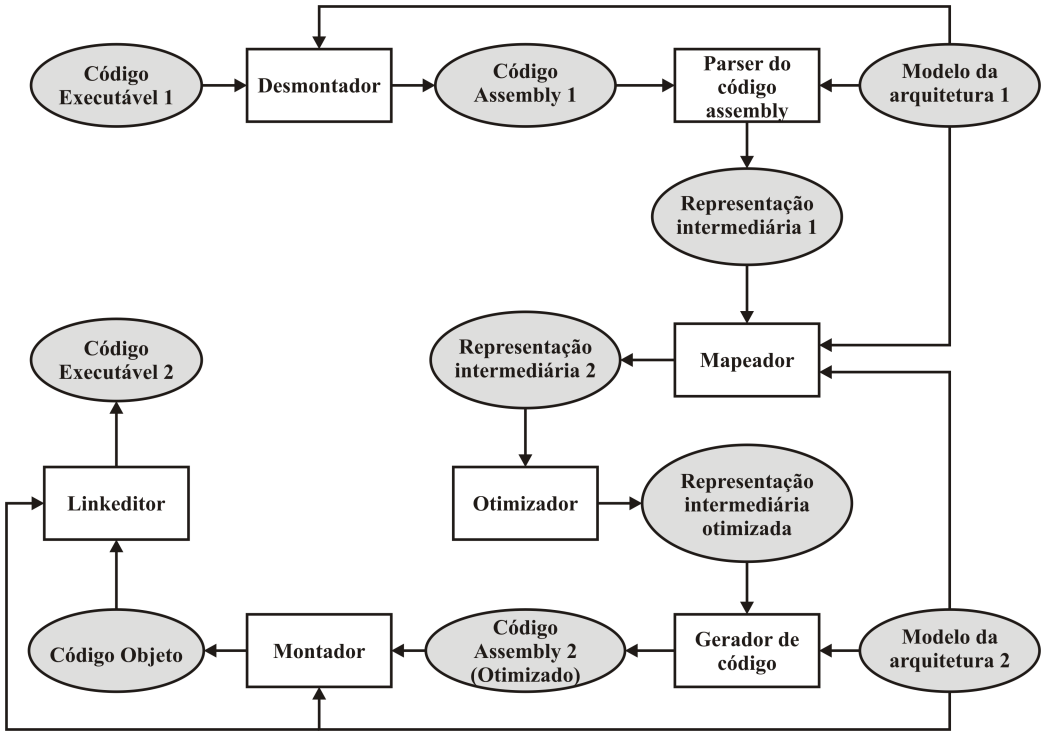

Figura 5.1: Fluxo de tradução binária

O código executável a ser traduzido deve inicialmente ser transformado em código *assembly* através de um desmontador. Em seguida, um *parser* do código *assembly* utiliza informações específicas da arquitetura para que seja gerada uma representação intermediária do código. Em geral este tipo de representação intermediária deve ser uma representação em mais alto nível das instruções do código *assembly* e deve ser independente

da arquitetura alvo.

A partir desta representação e de informações como sintaxe e semântica das instruções da arquitetura para a qual o código foi originalmente gerado, o mapeador deve operar fazendo com que ela seja traduzida em outra similar buscando instruções da arquitetura-alvo que tenham semântica equivalente. É gerada então uma nova representação que corresponde às instruções da arquitetura-alvo que sejam equivalentes às instruções da arquitetura original.

A representação resultante pode ser o ponto de entrada para um otimizador que opera antes da geração do código. A partir desta representação traduzida e otimizada, usando como insumos as informações específicas da arquitetura-alvo, a ferramenta deve gerar o código *assembly* equivalente ao original, o qual pode ser então montado e linkeditado para dar origem ao código executável traduzido.

As ferramentas usadas nos passos intermediários (desmontador, montador e linkeditor) são geradas automaticamente a partir dos modelos das arquiteturas fonte e alvo descritos através de uma ADL.

5.3 A integração do otimizador no tradutor binário

Note que a ferramenta otimizadora de código (cuja estrutura geral é mostrada na Figura 4.1 da Seção 4.3) utiliza componentes similares aos do tradutor binário proposto na Figura 5.1. Desta forma, o processo de implementação do tradutor pode ser facilitado através da re-utilização de tais componentes, como mostrado na Figura 5.2.

Neste caso, tanto a representação intermediária gerada pelo *parser* do código *assembly* como a gerada pelo tradutor são grafos ponderados de precedência (WPG1 representa as instruções do código original, WPG2 correspondente às instruções do código traduzido e SWPG ao código traduzido otimizado). O *parser* do código *assembly* e o módulo que gera código a partir do WPG obtêm as informações específicas das arquiteturas para a qual o código foi originalmente compilado e para a qual o código deve ser traduzido utilizando seus respectivos modelos, descritos através de ArchC.

O analisador de restrições de tempo real embutido no otimizador provê uma funcio-

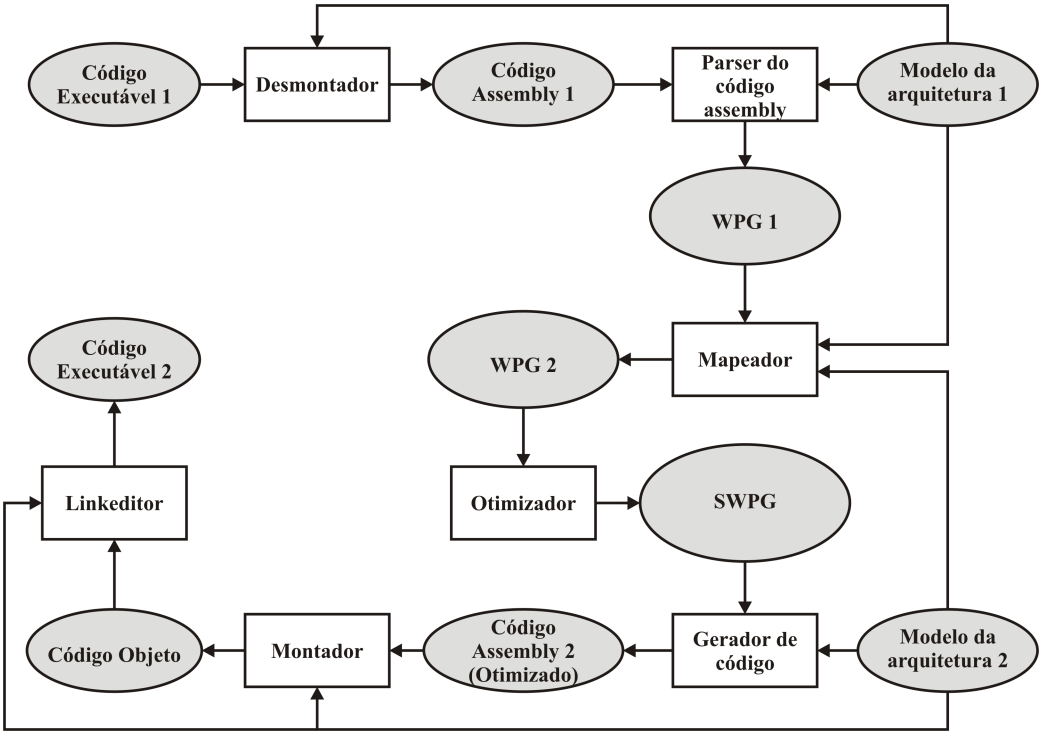

Figura 5.2: Integração do otimizador no tradutor binário

nalidade incomum nos tradutores binários. Pode-se, por exemplo, verificar a factibilidade do escalonamento de um mesmo trecho de código, sob as mesmas restrições temporais, em diferentes arquiteturas. Podem haver casos em que um segmento de código atende às restrições de tempo real quando executado em uma arquitetura, mas resulte em escalonamento infactível para uma outra arquitetura, sob as mesmas restrições temporais.

5.4 Resultados experimentais preliminares

Até o momento, o protótipo do tradutor binário proposto restringe-se à tradução binária de blocos básicos e foi validado para as CPUs MIPS e SPARC para o conjunto de *benchmarks* Dalton [DAL 06]. Contudo, o protótipo está sendo estendido para suportar código condicional.

As Tabelas 5.1 e 5.2 mostram os resultados preliminares da tradução do MIPS para

o SPARC e do SPARC para o MIPS, respectivamente, listando o número de instruções na arquitetura original e o número de instruções geradas no processo de tradução. Estes números diferem nas duas tabelas em função de algumas instruções da arquitetura de origem não possuírem uma instrução equivalente única na arquitetura destino, sendo então necessária a geração de duas ou mais instruções no processo de tradução.

Tabela 5.1: Resultados da tradução binária MIPS-SPARC

Programa	Número de instruções MIPS	Número de instruções SPARC
cast	38	34
fib	90	105
gcd	38	33
int2bin	23	26
negcnt	19	20
xram	27	35

Tabela 5.2: Resultados da tradução binária SPARC-MIPS

Programa	Número de instruções SPARC	Número de instruções MIPS
cast	31	28
fib	76	78
gcd	36	32
int2bin	18	19
negcnt	11	12
xram	36	40

A validação mais extensiva da técnica de tradução envolvendo código condicional, outras CPUs e um número maior de programas do *benchmark* será objeto de trabalho futuro (veja Capítulo 6).

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Apreciação do trabalho de pesquisa

Nesta dissertação foi proposta uma ferramenta de otimização de código pós-compilação, redirecionável e capaz de realizar uma análise de restrições temporais e fazer a alocação de registradores para a geração de código otimizado.

Para tornar a ferramenta redirecionável, as informações dependentes da arquitetura-alvo são extraídas de forma automática de uma descrição formal do processador, feita através de uma ADL. Ao mesmo tempo, restrições temporais e de precedência são codificadas em uma representação unificada utilizando-se grafos ponderados, provendo assim a base para a análise de factibilidade do escalonamento sob restrições temporais, utilizando-se algoritmos clássicos de cálculo de caminho mais longo, como o de Bellman-Ford [COR 90].

Além disso, o escalonador foi integrado a um protótipo de tradutor binário para a criação de uma ferramenta capaz de traduzir código compilado de uma arquitetura para outra, gerando código otimizado para a arquitetura-alvo.

Resultados experimentais foram apresentados para diferentes CPUs (MIPS, PowerPC e SPARC) e mostraram acelerações de até 1,3 vezes no tempo de execução de trechos de código já otimizados pelo compilador. Ao mesmo tempo, o *overhead* médio da análise de restrições temporais manteve-se abaixo de um limite superior dado pelo crescimento

do número de instruções.

A técnica descrita nesta dissertação se encaixa nos fluxos contemporâneos de projeto de sistemas embarcados pelas seguintes razões:

- **Redirecionabilidade provê suporte à exploração do espaço de projeto:** como se baseia em uma descrição formal do processador-alvo através de uma ADL, a técnica é redirecionável permitindo a avaliação de CPUs alternativas.

  A ADL utilizada foi a linguagem ArchC, o que é uma escolha pragmática visto que esta ADL é capaz de gerar modelos executáveis em SystemC, a linguagem mais promissora para a criação de modelos no estilo TLM.

- **Captura de restrições temporais facilita a extensão de modelos TLM atemporais:** a técnica se adequa à prática de adotar-se como ponto de partida no fluxo de projeto uma descrição TLM atemporal (o modelo PV), a qual é mais tarde refinada acrescentando-se informações sobre restrições temporais (o modelo PVT). Esta abordagem é ao mesmo tempo eficiente, já que guia as otimizações de modo a satisfazer as restrições temporais, e pragmática, pois preserva a infraestrutura existente de compiladores convencionais.

6.2 Contribuições técnico-científicas

A revisão da literatura apresentada no Capítulo 2 mostrou que, em geral, ferramentas baseadas em ADL provêm redirecionabilidade, mas não lidam de maneira eficiente com restrições temporais. Ao mesmo tempo, algumas abordagens tratam corretamente restrições temporais, mas não são redirecionáveis por focarem em arquiteturas específicas como, por exemplo, os DSPs. Portanto, a revisão da bibliografia não indicou a existência de trabalho anterior que combine redirecionamento automático com análise de restrições temporais para fins de escalonamento, explorando as restrições para guiar as otimizações no código.

A ferramenta aqui descrita baseia-se em descrições formais das CPUs-alvo escritas através da linguagem ArchC. Porém, algumas informações necessárias à análise de fluxo

de dados e à alocação de registradores não eram capturadas pelas construções da ADL ArchC em sua versão original. Por isto, foram propostas extensões para a linguagem de forma que ela viesse a capturar tais informações:

- Operandos fonte e destino das instruções.
- Componentes utilizados no cálculo do endereço efetivo de memória a ser lido ou escrito por instruções.
- Latências entre instruções.
- Grupos de registradores considerados de propósitos gerais para fins de alocação de registradores.

6.3 Produtos de trabalho

Os produtos resultantes deste trabalho de pesquisa foram:

- Protótipo do escalonador e analisador de restrições temporais com alocação de registradores, validado com modelos descritos em ArchC dos processadores MIPS R2000, PowerPC e SPARC V8.
- Artigo submetido ao IEEE Annual Symposium on VLSI (ISVLSI 2007) [FIL 06].
- Artigos submetidos ao IEEE International Midwest Symposium on Ciruits and Systems (MWSCAS 2007) [FIL 07b] [FIL 07a].

6.4 Tópicos para investigação futura

Ao longo desta dissertação foram levantadas algumas limitações da técnica proposta, bem como algumas restrições simplificadoras deliberadamente assumidas para fins de implementação do protótipo.

Assim, vários tópicos para continuidade deste trabalho consistem em relaxar as restrições do Capítulo 4, como segue:

- *Relaxamento da Restrição 1*: Um tópico para investigação em trabalhos futuros seria a realização de otimizações globais no código, utilizando-se técnicas como *code motion* e execução especulativa [dS 98]. Apesar de otimizações feitas no nível de blocos básicos mostrarem resultados satisfatórios, isto permitiria otimizações mais agressivas, possivelmente resultando em aumentos ainda maiores na aceleração de execução do código.

- *Relaxamento da Restrição 2*: Embora a restrição para máquinas *load*/*store* possa ser relaxada, o impacto desta generalização seria pequeno, pois apenas arquiteturas mais antigas não aderem a esta restrição.

- *Relaxamento da Restrição 3*: Outro ponto a ser investigado é o suporte a instruções que ocupam estágios do *pipeline* por mais de um ciclo. Isto permitiria que fossem feitas otimizações em programas contendo, por exemplo, instruções de ponto flutuante, que se encaixam nesta categoria, aumentando assim a aplicabilidade da ferramenta.

- *Relaxamento da Restrição 4*: O suporte a arquiteturas com emissão múltipla de instruções, como as superescalares e VLIW, também é um tópico que merece investigação em futuras implementações da ferramenta, tendo em vista que muitas CPUs utilizadas atualmente em sistemas embarcados apresentam esta característica.

Outro tópico que merece investigação futura refere-se ao subproduto desta dissertação, o tradutor binário otimizado, apresentado no Capítulo 5. Para aumentar a aplicabilidade da técnica proposta e estender a sua validação, uma versão futura da ferramenta deve suportar a tradução de programas contendo código condicional e ser capaz de traduzir código gerado para outras arquiteturas além daquelas usadas nos experimentos da Seção 5.4.

Referências Bibliográficas

[AHO 95] AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compiladores - Princípios, Técnicas e Ferramentas**. Livros Técnicos e Científicos Editora S.A., 1995.

[BAL 05a] BALDASSIN, A.; CENTODUCATTE, P. Geração automática de montadores para modelos de arquiteturas escritos em ArchC. In: PROCEEDINGS OF THE 9TH BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 2005. [s.n.], 2005. p.36–49.

[BAL 05b] BALDASSIN, A.; CENTODUCATTE, P.; RIGO, S. Extending the ArchC language for automatic generation of assemblers. In: PROCEEDINGS OF THE 17TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2005. [s.n.], 2005. p.60–67.

[BIN 05] BINUTILS, G. **The GNU Binutils Website**. Disponível em <http://www.gnu.org/software/binutils.

[CAS 06] CASAROTTO, D. C.; DOS SANTOS, L. C. V. Automatic Link Editor Generation for Embedded CPU Cores. In: NORTHEAST WORKSHOP ON CIRCUITS AND SYSTEMS - NEWCAS PROCEEDINGS, 2006. **Proceedings...** Gatineau, Canada: [s.n.], 2006.

[CAS 07] CASAROTTO, D. C. **Utilitários Binários Redirecionáveis: da Linkedição rumo à Tradução Binária**. Departamento de Informática e Estatística - UFSC, Março, 2007. Dissertação de Mestrado.

[CIF 98] CIFUENTES, C.; SENDALL, S. Specifying the semantics of machine instructions. In: IWPC '98: PROCEEDINGS OF THE 6TH INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION, 1998. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1998. p.126.

[CIF 99] CIFUENTES, C. et al. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. **Technical Committee on Computer Architecture News**, [S.l.], p.12–22, 1999.

[CIF 00] CIFUENTES, C.; EMMERIK, M. V. UQBT: Adaptable binary translation at low cost. **Computer**, Los Alamitos, CA, USA, v.33, n.3, p.60–66, 2000.

[CIF 02] CIFUENTES, C.; LEWIS, B.; UNG, D. **Walkabout - a retargetable dynamic binary translation framework**. Technical Report TR2002 -106, Sun Microsytems Laboratories.

[COR 90] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. **Introduction to Algorithms**. McGraw-Hill, 1990.

[COW 07] COWARE. **LisaTek**. Disponível em <http://www.coware.com>. Acesso em: January.

[DAL 06] DALTON. **The UCR Dalton Project**. Disponível em <http://www.cs.ucr.edu/ dalton/.

[dOS 07] DE OLIVEIRA SCHULTZ, M. R. **Geração Automática de Ferramentas de Inspeção de Código para Processadores Especificados em ADL**. Departamento de Informática e Estatística - UFSC, Março, 2007. Dissertação de Mestrado.

[dS 98] DOS SANTOS, L. C. V. **Exploiting instruction-level parallelism: a constructive approach**. Electrical Engineering department, Eindhoven University of Technology, 1998. Tese de Doutorado.

[FIL 06] FILHO, J. O. C.; SANTOS, L. F. P.; DOS SANTOS, L. C. V. **An Automatically-Retargetable Time-Constraint Driven Instruction Scheduler for Post-Compiling Optimization of Embedded Code**. Submetido ao ISVLSI 2007: IEEE Annual Symposium on VLSI.

[FIL 07a] FILHO, J. O. C. et al. **Extending the Design Exploration of Embedded Cores with Binary Translation**. Submetido ao MWSCAS 2007: IEEE International Midwest Symposium on Circuits and Systems.

[FIL 07b] FILHO, J. O. C.; SANTOS, L. F. P.; DOS SANTOS, L. C. V. **An Automatically-Retargetable Time-Constraint Driven Instruction Scheduler for Post-Compiling Optimization of Embedded Code**. Submetido ao MWSCAS 2007: IEEE International Midwest Symposium on Circuits and Systems.

[GCC 03] GCC. **The GNU Compiler Collection Website**. Disponível em <http://gcc.gnu.org.

[GHE 05] GHENASSIA, F. **Transaction-level Modeling with SystemC - TLM Concepts and Applications for Embedded Systems**. Springer, 2005.

[HAD 97] HADJIYIANNIS, G.; HANONO, S.; DEVADAS, S. ISDL: An instruction set description language for retargetability. In: PROCEEDINGS OF THE 34TH ANNUAL CONFERENCE ON DESIGN AUTOMATION, 1997. ACM Press, 1997. p.299–302.

[HAL 99] HALAMBI, A. et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 1999. [s.n.], 1999. p.485–490.

[HAL 01] HALAMBI, A. et al. A customizable compiler framework for embedded systems. In: INTERNATIONAL WORKSHOP ON SOFTWARE AND COMPILERS FOR EMBEDDED PROCESSORS, 2001. [s.n.], 2001.

[HAN 98] HANONO, S.; DEVADAS, S. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In: PROCEEDINGS OF THE 35TH ANNUAL CONFERENCE ON DESIGN AUTOMATION, 1998. ACM Press, 1998. p.510–515.

[HOH 04] HOHENAUER, M. et al. A methodology and tool suite for c compiler generation from adl processor models. In: DATE '04: PROCEEDINGS OF THE CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2004. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2004. p.21276.

[KäS 00] KäSTNER, D. PROPAN: A retargetable system for postpass optimizations and analyses. In: LCTES '00: PROCEEDINGS OF THE ACM SIGPLAN WORKSHOP ON LANGUAGES, COMPILERS, AND TOOLS FOR EMBEDDED SYSTEMS, 2000. **Proceedings...** London, UK: Springer-Verlag, 2000. p.63–80.

[KäS 03] KäSTNER, D. TDL: A hardware description language for retargetable postpass optimizations and analyses. In: GPCE '03: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING, 2003. **Proceedings...** New York, NY, USA: Springer-Verlag New York, Inc., 2003. p.18–36.

[LEU 01] LEUPERS, R.; MARWEDEL, P. **Retargetable Compiler Technology for Embedded Systems: Tools and Applications**. Norwell, MA, USA: Kluwer Academic Publishers, 2001.

[MAR 03] MARWEDEL, P. **Embedded System Design**. Kluwer Academic Publishers, 2003.

[MES 97] MESMAN, B. et al. Constraint analysis for DSP code generation. In: INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, 1997. [s.n.], 1997. p.33–40.

[MIB 06] MIBENCH. **MiBench Benchmark Suite**. Disponível em <http://www.eecs.umich.edu/mibench/.

[MIC 94] MICHELI, G. D. **Synthesis and Optimization of Digital Circuits**. McGraw-Hill Higher Education, 1994.

[PAT 04] PATTERSON, D.; HENNESSY, J. **Computer Organization and Design: The Hardware/Software Interface**. 3. ed. Morgan Kaufmann, 2004.

[PAT 06] PATTERSON, D.; HENNESSY, J. **Computer Organization and Design: The Hardware/Software Interface**. 3. ed. Morgan Kaufmann, 2006.

[PEE 99] PEES, S. et al. LISA — machine description language for cycle-accurate models of programmable DSP architectures. In: DESIGN AUTOMATION CONFERENCE, 1999. [s.n.], 1999. p.933–938.

[RAM 97] RAMSEY, N.; FERNáNDEZ, M. F. Specifying representations of machine instructions. **ACM Transactions on Programming Languages and Systems**, New York, NY, USA, v.19, n.3, p.492–524, 1997.

[RIG 04] RIGO, S. **ArchC: Uma linguagem de descrição de arquiteturas**. Instituto de Computação, UNICAMP, Campinas, Julho, 2004. Tese de Doutorado.

[SAL 06] SALTO. **The Salto Project**. Disponível em <http://www.irisa.fr/caps/projects/Salto/.

[SAN 07] SANTOS, L. F. P. **Alocação de Registradores para um Otimizador de Código Redirecionável**. Monografia de trabalho de conclusão de curso.

[SPA 97] SPAM Research Group. **SPAM Compiler User's Manual**, 1.0. ed., 1997.

[Sta 94] Stanford Compiler Group. **The SUIF Library**, 1.0. ed., 1994.

[SYS 06] SYSTEMC. **The SystemC Website**. Disponível em <http://www.systemc.org.

[TAG 05a] TAGLIETTI, L. et al. Automatic ADL-Based Assembler Generation for ASIP Programming Support. **Lecture Notes in Computer Science**, [S.l.], v.3553/2005, p.262–268, 2005.

[TAG 05b] TAGLIETTI, L. et al. Automatically Retargetable Pre-Processor and Assembler Generation For ASIPs. In: THE 3RD INTERNATIONAL IEEE-NEWCAS CONFERENCE PROCEEDINGS, 2005. [s.n.], 2005. p.215–218.

[VIN 02] VINCENTELLI, A. S. Defining platform-based design. **EEDesign of EETimes**, [S.l.], February, 2002.

[WAH 03] WAHLEN, O. et al. Instruction scheduler generation for retargetable compilation. **IEEE Des. Test**, Los Alamitos, CA, USA, v.20, n.1, p.34–41, 2003.