Instituto de Computação
Universidade Estadual de Campinas

# Geração Automática de *Backend* de Compiladores Baseada em ADLs

**Rafael Auler**

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rafael Auler e aprovada pela Banca Examinadora.

Campinas, 6 de dezembro de 2011.

Prof. Dr. Paulo Cesar Centoducatte
(Orientador)

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

FICHA CATALOGRÁFICA ELABORADA POR ANA REGINA MACHADO – CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA – UNICAMP

Au51g

Auler, Rafael, 1986-
Geração automática de backend de compiladores baseada em ADLs / Rafael Auler. – Campinas, SP : [s.n.], 2011.

Orientador: Paulo Cesar Centoducatte.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Compiladores (Computadores). 2. Arquitetura de computador. 3. Sistemas de computação.
4. Programação automática (Computação).
I. Centoducatte, Paulo Cesar, 1957-. II. Universidade Estadual de Campinas. Instituto de Computação.
III. Título.

Informações para Biblioteca Digital

**Título em inglês:** ADL based automatic compiler backend generation
**Palavras-chave em inglês:**
Compiling (Electronic computers)
Computer architecture
Computer systems
Automatic programming (Computer science)
**Área de concentração:** Ciência da Computação
**Titulação:** Mestre em Ciência da Computação
**Banca examinadora:**
Paulo Cesar Centoducatte [Orientador]
Olinto José Varela Furtado
Rodolfo Jardim de Azevedo
**Data da defesa:** 27-09-2011
**Programa de Pós-Graduação:** Ciência da Computação

# TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 27 de setembro de 2011, pela Banca examinadora composta pelos Professores Doutores:

**Prof. Dr. Olinto José Varela Furtado**
**INE / UFSC**

**Prof. Dr. Rodolfo Jardim de Azevedo**
**IC / UNICAMP**

**Prof. Dr. Paulo Cesar Centoducatte**
**IC / UNICAMP**

Instituto de Computação
Universidade Estadual de Campinas

# Geração Automática de *Backend* de Compiladores Baseada em ADLs

**Rafael Auler**[1]

Dezembro de 2011

**Banca Examinadora:**

- Prof. Dr. Paulo Cesar Centoducatte (Orientador)
- Prof. Dr. Olinto José Varela Furtado - UFSC (Titular)
- Prof. Dr. Rodolfo Jardim de Azevedo - UNICAMP (Titular)
- Prof. Dr. Luiz Cláudio V. Santos - UFSC (Suplente)
- Prof. Dr. Sandro Rigo - UNICAMP (Suplente)

[1]Suporte financeiro de: Bolsa CAPES por intermédio do Instituto de Computação da UNICAMP 2009–2010 (processo 01 P-04388/2010) e Bolsa FAPESP (processo 2010/02230-5) 2010–2011.

# Resumo

O processo de automatização da criação de *backends* de compiladores, isto é, do componente responsável pela tradução final para código de máquina, é perseguido desde o surgimento dos primeiros compiladores. A separação entre os algoritmos empregados no *backend* e a descrição formal da máquina, que requer conhecimento sobre a arquitetura alvo, é uma característica bastante desejada, uma vez que propicia a criação de novos *backends* sem a necessidade de conhecer o projeto do compilador, mas apenas do processador. Por esse motivo, um esforço natural para manter o desenvolvimento simples e intuitivo é a concentração do conhecimento sobre a máquina alvo em uma forma concisa de descrição, a partir da qual seja possível especializar algoritmos genéricos de compilação para este alvo específico. Uma linguagem de descrição de arquiteturas (ADL) permite a especificação das características arquiteturais de processadores, incluindo o seu conjunto de instruções (ISA). Neste trabalho, um estudo de mecanismos para gerar *backend* de compiladores através de descrições arquiteturais de processadores é apresentado, com ênfase no estudo de caso da ADL ArchC com o compilador LLVM. Um protótipo de um gerador de *backends* para LLVM a partir de uma descrição em ArchC foi desenvolvido, e *backends* para as arquiteturas ARM, MIPS, SPARC e PowerPC foram gerados com sucesso. Para alcançar este objetivo, foi usado um algoritmo de busca para resolver o problema da programação automática e inferir como implementar fragmentos pré-selecionados da linguagem intermediária LLVM utilizando instruções de uma arquitetura alvo arbitrária. Quatro técnicas para aumentar a velocidade deste algoritmo são apresentadas, de forma a não somente viabilizar uma solução para a geração automática de *backends*, mas também concluir o processo em menos de 20 segundos para três das quatro arquiteturas testadas. Programas do *benchmark* Mibench foram compilados com os compiladores gerados, executados com simuladores ArchC e os resultados comparados com aqueles gerados a partir dos mesmos programas compilados com os compiladores gcc e LLVM original, validando os novos *backends*. A qualidade do código gerado pode ser comparada com a de compiladores consagrados, caso seja utilizado um otimizador *peephole* para realizar substituições simples de algumas sequências ineficientes.

# Abstract

Researchers pursue the automation of compiler backend generation, the component responsible for final translation from intermediate language to machine code, since the begining of compilers theory creation. The separation between the algorithms used in the backend and formal machine description, which encompasses knowledge about the target architecture, is an important feature, since it facilitates the creation of new backends without the need for deep understanding of the compiler project. For this reason, an effort to maintain the development natural, simple and intuitive must concentrate the knowledge of the target machine in a concise description in a way it is possible to specialize generic algorithms to this target. An architecture description language (ADL) allows the specification of architectural features of processors, comprising the instruction set architecture available. This work presents a study of mechanisms for generating compiler backend through architectural descriptions of processors, with emphasis on a case study of the ArchC ADL with the LLVM compiler. We developed an automatic backend generator prototype for LLVM backends based on ArchC and successfully generated backends for the architectures ARM, MIPS, PowerPC and SPARC. To achieve this, we used a search algorithm to solve the problem of automatic programming and to infer how to implement pre-selected fragments of LLVM intermediate language using instructions of an arbitrary target architecture. We present four techniques to increase the speed of this algorithm which not only enables a solution for the automatic generation of backends, but also completes the process in less than 20 seconds for three of four architectures tested. Test compilation of Mibench benchmark programs attested the proper functioning of the backend and revealed that the quality of the generated code can compare with that of existing compilers, if a peephole optimizer were used to perform some simple substitutions of inefficient sequences.

# Sumário

# Capítulo 1

# Introdução

Decorrente da crescente integração dos dispositivos de hardware em um único chip, a indústria tem demonstrado uma grande aceitação de sistemas computacionais completos na forma de SoC (*System on Chip*). Esta tendência, associada à necessidade de redução nos tempos de projetos, criou uma demanda por ferramentas sofisticadas para simulação do sistema completo em alto nível. Para abordar este desafio, surgiu uma alternativa para a descrição de hardware denominada ADL (Architecture Description Language), capaz de descrever o projeto com elevado grau de abstração e flexível o suficiente para permitir o estudo do impacto no software que será executado nesta plataforma devido a decisões no projeto da arquitetura (*software/hardware co-design*).

Características importantes necessárias às ferramentas baseadas em ADLs não se restringem à capacidade de geração de simuladores comportamentais, mas englobam também a síntese de ferramentas para suporte à criação e depuração de software para a plataforma em desenvolvimento. Estas ferramentas compreendem desde montadores, ligadores e depuradores até compiladores. Somente com estes recursos o projetista conta de fato com um suporte adequado ao estudo do comportamento, em alto nível, do processador e do sistema que estão sendo projetados, pois um simulador é pouco útil se faltam meios para se criar softwares para serem simulados de maneira rápida. Rapidez é exatamente a ênfase buscada nas etapas iniciais dos projetos de plataformas, em que a equipe de hardware deve fornecer um simulador funcional e ferramentas para que a equipe de software possa iniciar o seu desenvolvimento desde o início do projeto.

ArchC [8] fornece diferentes ferramentas que, de posse das informações contidas em um modelo ADL de um processador, produzem simuladores (com ou sem precisão de ciclo), montadores, ligadores e depuradores. Os simuladores são produzidos com uma descrição SystemC do processador, permitindo que o módulo do processador possa ser facilmente integrado como parte de uma plataforma maior. A extensão do ArchC para geração automática de montadores e demais ferramentas binárias, denominada *acbingen* [11], en-

riqueceu a capacidade de descrição de um modelo arquitetural, com a possibilidade da descrição das sintaxes da linguagem de montagem específica para o conjunto de instruções do modelo. Deste modo, a ferramenta *acbingen* é capaz de extrair as informações necessárias e produzir automaticamente um montador para a arquitetura alvo. Entretanto, ArchC ainda necessita de ferramentas capazes de extrair informações do modelo ADL e redirecionar um compilador base para o processador descrito pelo modelo.

Os chamados compiladores redirecionáveis são projetados para serem estendidos com a capacidade de geração de código para novos alvos e novas linguagens de forma incremental. A arquitetura típica de um compilador pode ser vista como um *pipeline* em que se aplica um conjunto de transformações nos dados a cada estágio do processo de compilação. Para o compilador redirecionável, é interessante dividir as etapas em a) *frontend*, responsável por traduzir um programa escrito em uma determinada linguagem de programação para estruturas internas (representação intermediária); b) otimizações independentes de arquitetura e c) *backend*, responsável pela tradução da representação intermediária otimizada para código ou linguagem de montagem da máquina. É importante notar que, geralmente, o *frontend* é composto pelo léxico, *parser* e analisador semântico, ao passo que o *backend* é composto pelo seletor de instruções, escalonador e alocador de registradores.

A área de compiladores possui uma teoria sólida e estabelecida para a geração automática dos componentes léxicos e *parsers*. A tarefa de codificação manual destes módulos é tediosa e repetitiva, uma vez que se deve realizar análise de muitos casos que ocorrem na linguagem de programação que o *frontend* atende de forma a garantir que qualquer programa válido seja traduzido. Mostrou-se que este processo poderia ser automatizado, de forma que o código seja sintetizado a partir de uma descrição da linguagem de programação suportada pelo *frontend* [5], na forma de gramáticas livre de contexto. Esta formalização foi muito benéfica para o desenvolvedor de compiladores já que, basicamente, é exigido apenas conhecimento sobre a linguagem de programação suportada para gerar os módulos responsáveis pelas etapas iniciais necessários à compilação.

Em contraste à abordagem do *frontend*, a construção do gerador de código do compilador (formado pelo seletor de instruções, escalonador e alocador de registradores) continuou sendo predominantemente realizada de forma manual. O desenvolvimento manual desta etapa é longo e não trivial, uma vez que requer do desenvolvedor conhecimento profundo sobre a máquina alvo (para a qual o código será gerado) e sobre a linguagem intermediária do compilador, tornando o projeto desafiador. Por esse motivo, deu-se início a uma busca pela formalização teórica das técnicas de geração de código, na esperança de tornar o desenvolvimento do compilador mais intuitivo e eficiente. Com o surgimento de descrições formais de algoritmos para a seleção de código [2, 30, 61], logo surgiu a ideia de que seria possível a geração automática destes componentes, através dos chamados *code generator generators* (CGG).

Os algoritmos genéricos de seleção de código e alocação de registradores devem ser isolados da descrição dependente de máquina que especializa o código, e esta é a ideia principal dos CGGs [3, 20, 27, 28]. Entretanto, esta descrição da máquina também é dependente da linguagem intermediária usada pelo compilador, uma vez que se trata de um tradutor de linguagem intermediária para código de máquina. A formulação do problema é de fato semelhante ao do *frontend*, pois ambos são traduções entre diferentes formas de se representar um programa, e isto levou alguns pesquisadores a investigarem se *parsers* LR (*Left to right*, *Rightmost derivation*) usados nos *frontends*, quando modificados, não poderiam ser usados também para geração de código [30], onde uma gramática sintetiza a forma como as construções são agrupadas e ações de redução emitem código, isto é, quando uma instância da regra da gramática é reconhecida.

É possível criar geradores de código a partir de modelos de uma máquina que não dependem de informações sobre a linguagem intermediária do compilador. Deste modo, a descrição da máquina conta com a vantagem de ser consideravelmente mais fácil de ser escrita. A informação ausente sobre como relacionar as instruções da máquina com padrões de um compilador específico deve ser inferida pelo gerador de gerador de código através da análise das instruções da máquina e dos padrões do compilador. Infelizmente, o problema de mapear estas informações é intratável. A melhor abordagem, portanto, consiste em utilizar heurísticas de tempo polinomial que não garantem que uma solução será encontrada, mas que, na prática, alcança uma solução para a maioria das instâncias do problema. O uso de modelos de uma máquina que não dependem de informações sobre a linguagem intermediária é a maneira ideal para redirecionamento de um compilador baseado em um modelo ADL, como o oferecido por ArchC, pois o criador do modelo precisa apenas descrever o significado semântico das instruções do processador sem se preocupar em relacionar suas instruções com padrões de uma linguagem intermediária de um compilador específico. Essa é a abordagem usada neste trabalho.

## 1.1 Visão Geral do Fluxo ArchC e Motivação

A Figura 1.1 mostra um diagrama com o fluxo para produção de software e teste de uma plataforma de hardware descrita em ArchC, viabilizando o desenvolvimento conjunto do software e do hardware. No diagrama, caixas inclinadas representam entradas e saídas, ao passo que retângulos representam os componentes utilizados no processo. Aqueles em negrito representam ferramentas de síntese oferecidas por ArchC, com destaque para o sistema `acllvmbe`, desenvolvido neste trabalho. As flechas indicam tanto comunicação entre os componentes quanto transformações sucessivas da entrada até que se obtenha o resultado desejado. O software é representado pela caixa *Código C*, supondo que a linguagem de desenvolvimento seja C. O hardware deve ser descrito por um modelo ArchC

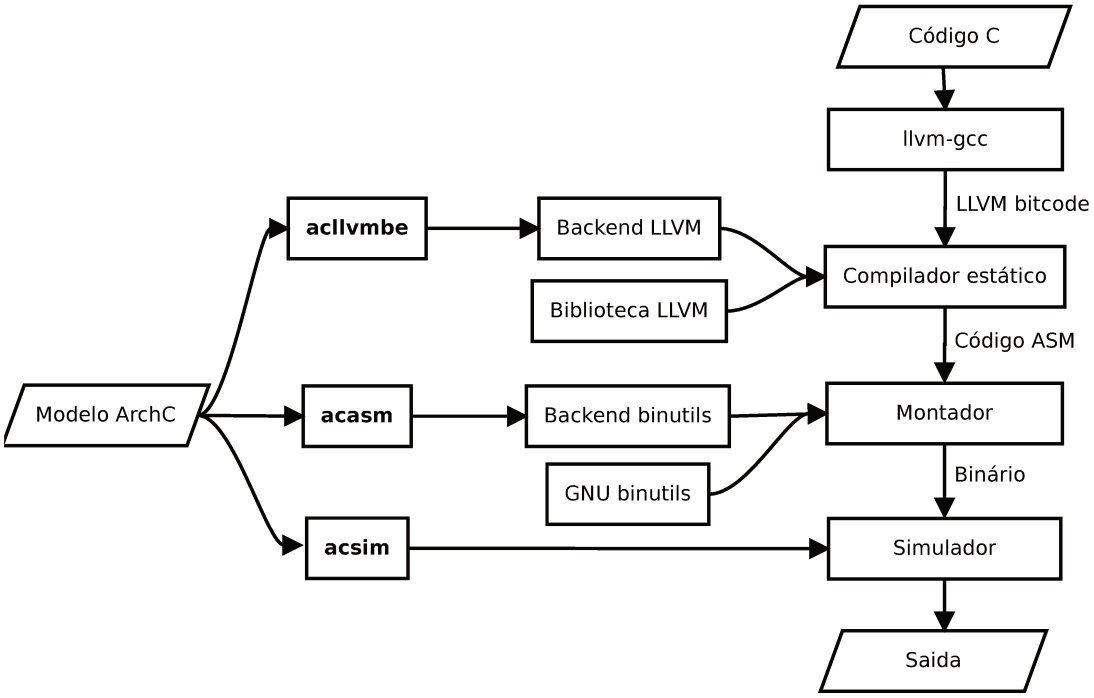



Figura 1.1: Visão geral.

e também aparece no diagrama como entrada do processo. Ao final, a saída produzida pode ser apresentada em diversas formas para auferir informações sobre o desempenho da plataforma e do software, ou simplesmente imprimir a saída do programa que foi executado na plataforma, para fins de verificação e validação do conjunto plataforma-software.

Os componentes básicos usados nesta dissertação foram o *Backend LLVM*, a *Biblioteca LLVM* e o sistema desenvolvido `acllvmbe`. As demais ferramentas do fluxo foram concluídas em projetos passados. `acsim` é o gerador de simuladores para a plataforma descrita com ArchC e foi a razão para a criação da ADL [8]. `acasm` ou `acbingen` [11], o gerador de ferramentas binárias que inclui o montador, ligador e depurador, deu origem ao desenvolvimento de ferramentas ArchC para a produção de software para uma plataforma projetada com ArchC, que antes era limitada ao estudo do hardware. Todavia, contando com apenas montadores e ligadores para a produção de software, o desenvolvimento é lento e tedioso, ou mesmo impraticável para programas maiores. A disponibilização de um compilador não só é crucial para o desenvolvimento de softwares, mas também para execução de testes de verificação da plataforma. A existência de um compilador propicia a automação da verificação da plataforma por meio da comparação da saída produzida por um mesmo programa em diferentes plataformas.

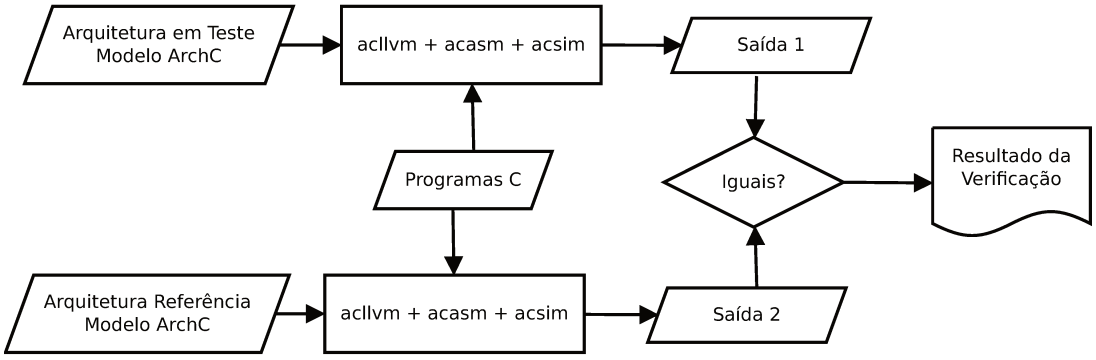



Figura 1.2: Diagrama de verificação de uma plataforma, utilizando um modelo ArchC correto como referência e o conjunto de ferramentas `acllvmbe`, discutida nesta dissertação, `acasm` e `acsim`.

Um exemplo de fluxo de verificação é apresentado no diagrama da Figura 1.2. Dispondo de uma arquitetura de referência, na forma de modelo ArchC, que produza resultados corretos e de programas escritos na linguagem C que irão executar na plataforma, é possível seguir o fluxo para verificar se uma arquitetura em teste produz o mesmo resultado, demonstrando sua corretude para os casos exercitados pelos programas utilizados na verificação.

## 1.2 Contribuições deste Trabalho

Em um compilador redirecionável, a etapa do *backend* é responsável pela seleção de instruções, alocação de registradores e escalonamento de instruções. O elemento crucial para a existência do gerador de código é certamente o seletor de instruções. As duas outras tarefas podem ser tratadas como uma forma especial de otimização (portanto, inicialmente dispensáveis). De fato, é possível realizar estas duas tarefas de forma trivial, sacrificando o desempenho do código gerado. As contribuições deste trabalho são:

- O projeto e desenvolvimento de um sintetizador de seletores de instrução para a construção de um *backend* funcional para o LLVM [45], uma infraestrutura de compilador redirecionável;
- Extensão da linguagem ArchC para incluir na descrição de um modelo de processador informações de semântica das instruções, para que o comportamento das mesmas possa ser capturado, bem como detalhamento da *Application Binary Interface* (ABI) do sistema, para que o *backend* do compilador gerado seja capaz de obedecer a restrições de uso dos recursos da máquina alvo;

- Definição de um esqueleto de código para um *backend* mínimo para o LLVM, a ser utilizado pelo gerador de *backends*, mas que pode também ser usado por um programador interessado em criar um novo *backend* manualmente;

- O projeto e desenvolvimento de uma infraestrutura para extração de informações de modelos ArchC para, em conjunto com os itens anteriores, gerar automaticamente um *backend* de uma máquina alvo descrita com ArchC para o LLVM, possibilitando derivar um compilador capaz de traduzir código C em código para a máquina cuja descrição em ArchC esteja disponível;

- Estudo e aperfeiçoamento do algoritmo de busca utilizado para resolver o problema da programação automática necessário para gerar automaticamente uma implementação para fragmentos da linguagem intermediária da infraestrutura LLVM utilizando instruções da máquina alvo. Neste contexto, quatro técnicas foram desenvolvidas para viabilizar a execução do algoritmo de busca em tempo hábil e, ainda, fazer com que todo o processo de geração de um *backend* fosse concluído em menos de 20 segundos para 3 das 4 arquiteturas testadas.

## 1.3 Organização desta Dissertação

Este trabalho foi estruturado da seguinte forma:

- O Capítulo 2 contém uma análise das primeiras abordagens para o problema da seleção de instruções, mostrando como a área evoluiu para geradores automáticos e qual papel as ADLs desempenham neste processo de automação.

- O Capítulo 3 introduz a infraestrutura LLVM e os detalhes relacionados ao backend do compilador estático LLVM, ou seja, o componente responsável por traduzir linguagem intermediária LLVM em código de máquina, cuja geração automática é tema deste trabalho.

- O Capítulo 4 apresenta as extensões propostas à linguagem ArchC para que um modelo contenha as informações necessárias para a geração automática de compiladores.

- O Capítulo 5 apresenta a ferramenta `acllvmbe` desenvolvida neste trabalho, bem como suas funcionalidades.

- O Capítulo 6 contém os resultados experimentais, a forma de análise e a análise dos resultados em si.

- O Capítulo 7 contém as conclusões obtidas no desenvolvimento deste trabalho.
- O Apêndice A contém listagens de todos os operadores da árvore de semântica, a linguagem de representação de semântica introduzida no Capítulo 4, bem como das regras de transformação utilizadas e dos padrões que cobrem a linguagem intermediária do LLVM.

# Capítulo 2

# Trabalhos Relacionados

## 2.1 Primeiras Abordagens para Seleção de Instruções

Na década de 70 já se reconhecia a necessidade de formalizar as abordagens para a geração de código [20] e muitas pesquisas abordaram este tema. A grande contribuição desta fase foi o desenvolvimento de formalismo teórico necessário para lidar com *straight line code* [4] (código que representa um bloco básico, sem desvios) e do desenvolvimento de soluções eficientes, neste escopo simplificado, para alocar registradores [61] e realizar otimizações básicas [4]. O tema que aborda a geração de geradores de código também foi discutido [20], porém de forma incipiente e bastante limitada. Nesta época apareceram soluções elegantes para o problema de seleção de código de modo a possibilitar que um mesmo algoritmo fosse aplicado para vários tipos de máquinas, utilizando programação dinâmica [2].

A programação dinâmica, desde cedo, mostrou o poder da geração de código baseada em casamento de padrões de árvores. O algoritmo de programação dinâmica apresentado [2] está acompanhado de provas formais de seu funcionamento e é mais completo do que o utilizado atualmente, pois resolve também o problema da alocação de registradores. Mais tarde [3], o uso do algoritmo de programação dinâmica foi limitado à etapa de seleção de instruções e o problema de alocação de registradores tratado separadamente, uma vez que para máquinas de registradores homogêneos não há necessidade de integrar estas duas tarefas (posteriormente, entretanto, o problema voltaria a ser integrado nas abordagens para DSPs [6]). No final da década de 70, Susan Graham e Steven Glanville deram origem a uma abordagem diferente, utilizando *parsers* LR modificados para geração de código [30]. A linguagem intermediária linearizada em notação prefixa era o objeto do *parsing*, que deveria contar com informações da máquina alvo, condensada na forma de uma gramática livre de contexto. O resultado das reduções das produções da gramática dá origem ao código alvo. Esta abordagem foi, com o tempo, perdendo relevância no cenário de geração de código, uma vez que as restrições impostas pela des-

crição da geração de código em uma gramática LR (modificada) e as idiossincrasias do método mostraram-se em desvantagem com relação aos geradores baseados em casamento de padrões.

## 2.2 Seletores de Instruções Eficientes e Geradores de Gerador de Código

No final da década de 80 e início da década de 90 surgem vários sistemas (TWIG [3], BEG [22], BURG [28], IBURG [27] e Olive [66]) como alternativas diferentes para a implementação de BUPMs (*bottom up pattern matchers*) [63], isto é, geradores de geradores de código que usam uma descrição dependente de máquina na forma de padrões de árvore intermediária. A geração de código é realizada pelo casamento destes padrões em um percurso *bottom-up* e emissão das instruções correspondentes. TWIG mostra-se como um caso ligeiramente diferente, uma vez que realiza um percurso *top-down*.

TWIG é a tentativa de colocar na prática parte do algoritmo originalmente apresentado em [2], sem alocação de registradores, para máquinas com registradores homogêneos. Entretanto, o algoritmo de programação dinâmica antigo não especifica exatamente como o casamento de padrões deve ser feito, pois este é, por si só, um tema de pesquisa à parte. Dessa forma, TWIG utiliza um casamento baseado em strings e na construção de um DFA (*Deterministic Finite Automaton*), similar ao utilizado no léxico do *frontend* de um compilador. TWIG também formaliza sua linguagem para descrição de padrões de árvores, determinação do custo de um padrão e ações de emissão de código, de forma semelhante àquela apresenta neste trabalho. A diferença da abordagem adotada neste trabalho, contudo, é que o projetista deve descrever as instruções do processador, não padrões e nem ações de emissão de código, pois estes são transparentes para o projetista e decididos pelo sistema.

BEG utiliza uma linguagem mais simples e casamento de padrões diretamente codificado no gerador de código sintetizado (método *naïve*). BURG é levemente baseado na teoria BURS [52, 54, 56] para realizar programação dinâmica em tempo de geração do gerador de código, criando um seletor de instruções muito mais veloz, porém mais restrito (dependendo de definições estáticas de custo). BURG explora o compromisso espaço-tempo no sentido de pré-computar cálculos de programação dinâmica e gerar um autômato com tais informações, de modo que o gerador de código se limita a percorrer estados desta estrutura (mais veloz, porém com maior consumo de memória).

IBURG é uma implementação que reconhece as mesmas descrições de padrões de árvore que o BURG, sendo portanto facilmente comparável com este último. Todavia, é mais flexível, uma vez que realiza programação dinâmica em tempo de geração de código

(perdendo desempenho em relação ao BURG) e mais simples do que o TWIG pois realiza casamento de padrões de forma *naïve* (semelhante ao BEG).

Na década de 90, Stanford criou seu próprio modelo de compilador redirecionável com objetivo de facilitar o redirecionamento para processadores DSP emergentes na época. O seletor de instruções e sua linguagem para descrição da máquina foram batizados de Olive [66]. Esta solução é apresentada como sucessora do TWIG. O sistema de geração é baseado no IBURG.

Na área de algoritmos de seleção de instruções, também foram desenvolvidos novos métodos, explorando a geração a partir de *Directed Acyclic Graphs* (DAGs) [43] ao invés de árvores de expressão (a maneira tradicional). Isto é particularmente importante para o seletor de instruções utilizado no compilador LLVM, usado neste trabalho, pois sua linguagem intermediária é convertida para um DAG para então aplicar o algoritmo de seleção de instruções em DAG. Este processo é detalhado no Capítulo 3. Outros métodos para a seleção de instruções incluem o uso de diagramas *trellis* [67] e seleção integrada à alocação de registradores [6]. Estes métodos concorrem com abordagens tradicionais de seleção em árvores e também alternativas integradas de seleção e otimização com o uso de otimizações *peephole*.

## 2.3 Geradores de Código Sintetizados a partir de Descrição Declarativa de Máquina

De forma paralela aos esforços para construir geradores de gerador de código eficientes baseados em descrição de gramática de árvore simples (uma definição deste tipo de gramática pode ser encontrada em [52]), alguns pesquisadores [14, 19, 39] focaram seus esforços na geração de *backend* baseada em uma descrição declarativa da máquina. Esta descrição não contém informações sobre como realizar o casamento de instruções com padrões da árvore da linguagem intermediária, sendo que, para que o gerador de código seja corretamente sintetizado, é necessário inferir estes dados a partir da descrição das instruções da máquina. Conforme comentado na introdução desta dissertação, esta abordagem conta com uma qualidade apreciável para nosso trabalho, na qual o usuário que descreve a máquina não precisa conhecer o compilador e sua linguagem intermediária, mas apenas a semântica das instruções da arquitetura. Esta é a abordagem que mais se adapta ao contexto de ADLs, em que uma descrição declarativa de máquina informa as características da máquina e as ferramentas de síntese extraem informações para gerar programas.

Um dos primeiros trabalhos com este enfoque foi devido a Cattel [14]. O seletor de instruções é gerado completamente a partir de informações sobre a semântica de cada ins-

trução. A descrição desta semântica é realizada com uma árvore que expressa as operações realizadas, com a possibilidade de especificar os efeitos colaterais da instrução. Com estas informações, Cattel resolve o problema de encontrar como realizar a computação de uma certa expressão utilizando tais instruções. Para isso, um conjunto de transformações aritméticas é empregado para manipular a expressão a se computar até que se prove que a expressão equivale à semântica de uma instrução do alvo, ou a um conjunto de instruções do alvo. Esta solução pode ser vista como um provador de teoremas simples, que busca demonstrar equivalências. Com isso, é possível *inferir* como calcular uma expressão solicitada pela linguagem intermediária, de posse apenas do comportamento das instruções da máquina alvo. Este mecanismo é suficiente para gerar um tradutor de linguagem intermediária para instruções da máquina alvo, limitado pela capacidade de busca do provador de teoremas.

## 2.4 Comparadores Semânticos e Otimizadores de Código Redirecionáveis com Descrição Declarativa

Os conceitos discutidos na seção anterior podem ser generalizados de modo que sejam aplicados não apenas a problemas de geração automática de seletores de instrução, mas também em otimizadores de código. O conceito principal é a capacidade de sintetizar uma sequência de instruções que realiza uma dada computação. Se houver informações sobre o desempenho das instruções, é possível atribuir custo a essas sequências de instruções. Dessa maneira, é possível construir um *peephole optimizer* baseado em *comparador semântico*, que verifica se duas sequências de código realizam a mesma computação e avalia qual delas melhor satisfaz a alguma função objetivo. Tais métodos são importantes para criar otimizadores de código para arquiteturas arbitrárias, descritas de forma declarativa e podem ser usados em um trabalho futuro com o objetivo de se construir *backends* otimizantes.

Fraser et al. [17, 18, 26] apresenta sua versão de um compilador redirecionável e otimizante. De forma adiantada no processo de compilação, o código é traduzido para uma forma de representação intermediária de baixo nível, muito próxima ao código de fato da máquina. Esta tradução é realizada por um *expansor de código*. Entretanto, o código produzido por esta etapa é de baixa qualidade, gerado por um seletor de instruções trivial que não conta com garantia de otimalidade local. A seguir, um *peephole optimizer* redirecionável é utilizado para sucessivamente substituir sequências de instruções de baixo desempenho por sequências equivalentes otimizadas. Este tipo de otimizador utiliza uma descrição das instruções da máquina baseada em *register transfer* e garante que não existe, no código otimizado, uma sequência de 2 ou 3 instruções que possa ser substituída por

uma instrução que realiza a mesma computação de forma otimizada, pois o objetivo do otimizador é realizar todas essas substituições possíveis. A abordagem mais veloz utiliza um conjunto de treinamento, em tempo de compilação do compilador, para identificar sequências que podem ser otimizadas e então construir um *peephole optimizer* guloso baseado nestas informações.

Kessler [42] desenvolveu um otimizador mais próximo dos conceitos de Cattel, redirecionável por descrição do comportamento das instruções. O objetivo é substituir alguma sequência de código de tamanho arbitrário por uma única instrução capaz de realizar a mesma computação. Esta comparação é realizada por uma versão simplificada de um comparador semântico criada por Kessler.

Conforme afirmado na introdução, o problema de se encontrar uma sequência de instruções que representa um comportamento arbitrário é intratável. Entretanto, é possível utilizar heurísticas que ofereçam boas soluções em tempos razoáveis. Um exemplo é a busca baseada em transformações algébricas de Cattel. Já *superoptimizer* [50] apresenta uma abordagem de busca exaustiva para encontrar uma sequência de instruções potencialmente não trivial que implemente o mesmo comportamento de um programa a se otimizar. O algoritmo não é capaz de operar com muitas instruções nem otimizar programas com mais de 10 instruções de máquina, devido ao tempo necessário para realizar a busca exaustiva.

Hoover [39] apresenta a sua plataforma para redirecionar um compilador completamente, incluindo todas as otimizações, chamado TOAST. Este é baseado fortemente em comparadores semânticos para concluir diversos aspectos sobre a arquitetura, como por exemplo, se é proveitoso realizar *strength reduction* para uma dada expressão. Para responder a tais perguntas dependentes da arquitetura, TOAST consulta seu comparador semântico para inferir se uma otimização leva a resultados melhores. Com esta abordagem, TOAST consegue redirecionar um compilador que leva em consideração a máquina alvo em todas as fases de otimização (não existe etapas de otimização independentes de máquina). TOAST não realiza seleção de instruções baseada em casamento de padrões em árvores, mas utiliza uma abordagem semelhante a Fraser [18], dependendo de um *peephole optimizer* guloso para melhorar a qualidade do código. O comparador semântico utilizado é mais elaborado do que aquele apresentado no artigo de 16 anos atrás, de Cattel [14].

Dias [19] realiza um desenvolvimento formal sobre o problema de geração automática de seletor de instruções baseada em descrição declarativa de máquina e apresenta os resultados para o seu sistema, alegando ter facilmente redirecionado um compilador para processadores das arquiteturas ARM, x86 e PowerPC. O trabalho é inspirado em Cattel, uma vez que a heurística de busca também é baseada em transformações algébricas. A diferença está no nível de formalismo empregado e no uso de políticas menos restritivas do espaço de busca, melhorando a qualidade dos geradores de código.

### 2.4.1 Superotimizadores

O termo *superotimização* passou a ser empregado com o mesmo sentido que o termo *otimização* possui quando tecnicamente correto, isto é, a maneira ótima de se calcular uma expressão, dadas algumas restrições. Esta mudança na terminologia ocorreu porque o termo *otimização*, na área de compiladores, virou sinônimo de melhoria, ao invés de busca pelo resultado ótimo.

A ideia de superotimização introduzida por Massalin [50] deu início a um novo tema de pesquisa ligado ao de otimizações baseadas em descrições declarativas relacionado ao problema de programação automática da área de inteligência artificial. O problema da programação automática consiste em encontrar uma sequência de instruções de máquina que realize uma determinada computação. Se adicionarmos a restrição de que esta sequência é ótima de acordo com algum critério, o problema do superotimizador é definido. O trabalho de Massalin é inviável de ser usado na prática, uma vez que a abordagem apresentada utiliza busca exaustiva para construir um programa solução dispensando qualquer heurística ou artifício para reduzir o espaço de busca. O algoritmo, portanto, é genérico demais e pouco acrescenta para o comunidade científica em termos práticos. Todavia, é um trabalho notável por primeiro explorar a superotimização.

A ideia de superotimização foi refinada e melhor explorada mais tarde. Denali-2 [40] é o estudo teórico do protótipo de um superotimizador de código. Para alcançar este objetivo, a formalização do problema a ser resolvido, a especificação dos objetivos e instâncias do problema muito se assemelham ao projeto desta dissertação e ao de Cattel [14], aplicando também o conceito de axiomas e regras de transformação que preservam equivalência da semântica de uma expressão. O problema resolvido é formalmente definido como *Straight-line Automatic Programming* e corresponde ao mesmo problema abordado por esta dissertação para alcançar a geração automática de um seletor de instruções. Programação automática é utilizada para concluir como implementar um padrão específico do tradicional algoritmo de casamento de padrões para geração de código. Contudo, a forma de se resolver o problema não utiliza sistemas de reescrita de termos, como o proposto nesta dissertação, mas faz uso de uma estrutura de dados chamada *E-graphs*. A abordagem de Denali é muito mais cara em recursos computacionais, porém impõe menos restrições na descrição das regras e axiomas.

Apesar dos problemas de geradores de otimização baseados em descrição declarativa de máquina e o de geração de seletores de código poderem ser resolvidos com programação automática e compartilharem diversos aspectos em comum, é importante ressaltar que a solução do último não necessariamente deve gerar um programa ótimo. O foco está em encontrar um programa que compute uma expressão, mas não necessariamente otimizar, uma vez que a otimização propriamente dita é realizada pelo *peephole optimizer* que se apresenta como último passo antes da emissão de código em si. Por este motivo, o gerador

de seletor de código pode utilizar diversas heurísticas para aumentar sua velocidade e evitar buscas desnecessárias por programas mais curtos. O gerador do seletor de instruções é apenas um dos aspectos da geração de *backend*. Maior atenção deve ser dispendida na implementação de regras da interface binária de aplicação, ou *Application Binary Interface* (ABI).

## 2.5 ADLs e Compiladores Redirecionáveis

Atualmente, os desafios impostos pelas arquiteturas emergentes e que necessitam de compiladores redirecionáveis são derivados de processadores integrados em SoCs que não são de propósito geral, ou seja, que possuem um conjunto de instruções para computação específica e são classificados como ASIPs (*application specific instruction-set processors*). Esta categoria também pode incluir DSPs e demais arquiteturas particulares. Como o desenvolvimento destes processadores pode ser auxiliado por ADLs, a solução natural é integrar a descrição necessária para redirecionar um compilador diretamente na descrição ADL da arquitetura do processador.

O problema de redirecionar um compilador com base em uma descrição ADL da máquina foi estudado no contexto dos projetos LISA [37], Flexware CodeSyn [47, 48], AVIV [36], RECORD [46], CHESS [44], EXPRESSION [35] e MADL [57].

No caso da ADL LISA 2.0, a primeira tentativa [38] de redirecionamento de um compilador C consistiu em utilizar ajuda do projetista do modelo (através de uma GUI, ou *Graphical User Interface*) para especificar manualmente como implementar um padrão da árvore IR com instruções da máquina. Apesar de oferecer grande flexibilidade para que o projetista criasse o gerador de código, este método não é automatizado e pode se tornar facilmente uma das etapas mais trabalhosas na criação do modelo do processador. Para corrigir estes problemas, um trabalho subsequente [15] realizou a geração automática de sequências de instruções para estes padrões, utilizando técnicas de transformação algébrica semelhantes às descritas na seção anterior. O artigo não fornece detalhes do método empregado nem aponta referências sobre o algoritmo utilizado, mas é possível concluir que há uso de transformações algébricas, uma vez que o texto comenta sobre a possibilidade do usuário alterar um arquivo com as transformações possíveis e estendê-lo com a possibilidade de suporte a arquiteturas não previstas.

A solução do gerador CodeSyn da plataforma Flexware é redirecionar um compilador através de informações sobre padrões de computação que são implementados com instruções do alvo e, portanto, de forma manual. Ainda, uma alternativa baseada nos moldes do compilador redirecionável de Fraser [18] (baseado no PO - *peephole optimizer*) que funciona basicamente com substituições gulosas, foi estudada para a plataforma Flexware [48], alegando a capacidade de redirecionar compiladores mais facilmente e de

forma rápida. Contudo, se a abordagem de gerar automaticamente código para os padrões fosse utilizada, a alternativa inicial ficaria ainda mais fácil para o projetista.

AVIV é um sistema de geração de *backend* para o compilador SPAM [32], processando a descrição ISDL [34] de uma máquina. Este trabalho apresenta uma solução diferente para a geração de *backend*, pois trata os problemas de seleção de código, alocação de registradores e escalonamento ao mesmo tempo. Para viabilizar esta análise, um *Split-Node DAG* é criado. Trata-se da representação de um bloco básico em que todas as maneiras de realizar a computação deste bloco com recursos da máquina alvo estão inclusos. Utilizando as informações neste grafo, AVIV não só seleciona código, mas também realiza escalonamento das instruções e considera informações preliminares sobre a alocação de registradores, aumentando a qualidade do código gerado. Em uma segunda etapa, a alocação de registradores de fato é realizada e um sistema de *peephole optimization* procura melhorar porções ineficientes do código final. A publicação deixa implícito que um mapeamento da descrição de instrução em ISDL para operações básicas da linguagem intermediária SUIF (*Stanford University Intermediate Format*) é realizado automaticamente e, portanto, recorrendo a técnicas de automatização deste processo.

EXPRESSION é uma linguagem de descrição de arquiteturas voltada para exploração dos compromissos do projeto de um processador. Conceitualmente, isto é alcançado através da geração de simuladores e compiladores. Entretanto, o compilador gerado depende de construções explícitas na linguagem que mapeiam linguagem intermediária para instruções e, assim, não é um processo automatizado, mas sim deixado ao projetista.

CHESS e CBC [24] são duas abordagens diferentes para geração de gerador de código baseado em uma descrição de máquina em nML [25], criados por dois grupos de pesquisa distintos. nML é uma linguagem de descrição de arquitetura que mistura descrição estrutural (para elementos que armazenam estado) e comportamental (para operações do *datapath*). O objetivo primário da linguagem é a descrição de DSPs de ponto fixo. Ambos CHESS e CBC extraem, a partir da descrição nML, um grafo bipartido em nós que representam estado e nós que representam operações, chamado de grafo do conjunto de instruções (ISG). Contudo, CBC utiliza o grafo para derivar padrões que alimentam um seletor IBURG, ao passo que CHESS não gera padrões, mas utiliza um algoritmo particular de análise *ad-hoc* do ISG, na tentativa de mapear o grafo de fluxo de dados do programa de entrada nas restrições impostas pelo ISG. Esta última abordagem pode ser consideravelmente mais lenta do que simples casamento de padrões. Ainda, diferentemente da solução de seleção de instruções e alocação de registradores integrada do AVIV, o sistema de geração de código é separado em fases distintas.

RECORD é um gerador de especificações em gramática de árvore para IBURG (que por sua vez gera um gerador de código), a partir de um modelo de processador em MIMOLA [49], semelhante ao CBC para nML. Utiliza, para tanto, uma abordagem auto-

matizada que extrai um conjunto de instruções a partir de uma especificação estrutural do processador (no caso do MIMOLA, tecnicamente um subconjunto de uma linguagem de descrição de hardware). Esta abordagem não é necessária para as ADLs ArchC, EXPRESSION, nML e LISA, que contam com descrição explícita do conjunto de instruções. Cada instrução extraída forma um padrão, que mais tarde é expandido para englobar um conjunto capaz de cobrir qualquer árvore de representação intermediária. O método de extensão é implícito e não é comentado na publicação científica. Exploração de paralelismo é realizado em uma fase distinta após a seleção de instruções, diferentemente de AVIV.

Farfeleder et al. [23] definem uma nova linguagem de descrição de arquiteturas baseada em XML. O foco é geração de simuladores e redirecionamento de compiladores. *Open compiler environment* da Altair é a infraestrutura de compilação utilizada. Nesta ADL, o projetista precisa definir padrões de casamento de árvore para redirecionar o compilador. Contudo, apenas os padrões básicos (que garantem cobertura de todos os casos) são definidos. Os demais padrões são inferidos através de regras aritméticas simples (o mecanismo não é tão elaborado quanto de Cattel, por exemplo). As fases de escalonamento e alocação de registradores são distintas. O mérito desta publicação, entretanto, é a sua clareza e especificação de todos componentes importantes do sistema, ao contrário de outros sistemas comerciais em que o mecanismo exato utilizado é obscurecido. Do mesmo grupo de pesquisa, Brandner et al. [13], um ano mais tarde, apresentam uma nova definição de ADL baseada em XML, que dessa vez utiliza descrição estrutural da arquitetura (como MIMOLA). Os padrões do seletor de instruções são agora derivados de forma completamente automática, apenas com a definição do comportamento das *unidades* do projeto (computam algum valor baseado nos elementos de estado).

MADL é uma linguagem de descrição de arquiteturas focada na cobertura de diversos tipos de processadores, como escalar, superescalar, VLIW e *multithreaded*. O objetivo da linguagem é o auxílio a geração automática de ferramentas como simuladores e compiladores. Para isso, a linguagem utiliza um modelo em duas camadas. A abstração do modelo de processador adotada, de modo a não restringir arquiteturas diferentes, é uma máquina de estados de operações. Recursos são modelados como *tokens*, e um protocolo básico de *tokens* é utilizado pela máquina de estados para modelar situações de uso e liberação dos recursos (registradores e unidades funcionais). A segunda camada do modelo é a de anotação, contendo informações sujeitas a interpretação das ferramentas que a utilizarão (tornando assim a ADL facilmente extensível). Este modelo é substancialmente diferente do utilizado pelo ArchC, que é uma ADL híbrida em que o comportamento das instruções são descritas com funções SystemC.

# Capítulo 3

# A Infraestrutura LLVM

Como infraestrutura de apoio para compilação e geração de código para uma máquina arbitrária foi usado neste trabalho o projeto LLVM.

A linguagem de representação intermediária LLVM [45] utiliza a forma SSA (*static single assignment*) e, portanto, conta com a função $\phi$ para construir código SSA e um conjunto virtual de instruções para expressar a lógica de um programa. O código gerado por LLVM dessa maneira pode ser lido por humanos na forma de linguagem de montagem, ou armazenado em arquivos em seu formato comprimido chamado *bitcode*.

A infraestrutura para construção de compiladores LLVM propicia o desenvolvimento de componentes independentes que realizam otimizações no código, análises e tradução para código de máquina específico. O LLVM pode ser visto também como um compilador estático completo (otimizações intraprocedurais e interprocedurais) se construído com os componentes necessários para isto. Pode também ser usado como um compilador em tempo de execução que utiliza informações disponíveis apenas em tempo de execução para realizar otimizações mais agressivas.

O projeto LLVM provê uma estratégia de compilação com otimizações por toda a vida útil de um programa. Especificamente, há um grande conjunto de otimizações disponíveis na infraestrutura LLVM, seja em tempo de criação de objetos (intraprocedural), tempo de ligação de objetos (interprocedural), tempo de execução (dinâmico) e também quando não está executando, após instalado. Esta última compreende otimizações caras que utilizam informações de *profiling* de execuções passadas e que são realizadas quando o usuário não está utilizando tempo de processamento de seu computador.

## 3.1 Redirecionando LLVM com um Novo Backend

Seguindo o paradigma original do projeto LLVM, o *backend* é escrito em linguagem C++, com algumas informações contidas em arquivos utilizando a linguagem particu-

lar do LLVM, chamada TableGen, cujo objetivo é capturar informações declarativas, isto é, que explicitam informações de domínio e não de como resolver um problema em particular. Ainda que a ideia geral do projeto de *backend* do LLVM seja de descrevê-lo completamente utilizando TableGen, na prática, muitas informações precisam ser codificadas em C++. Como o projeto desta dissertação, `acllvmbe`, possui o objetivo de gerar automaticamente um *backend* para o LLVM, não faz diferença prática se o *backend* gerado utiliza C++ ou TableGen. Isto acontece porque as informações em TableGen visam automatizar alguns trechos de escrita do *backend* que podem ser tediosos se escritos em C++. Entretanto, como o *backend* não é escrito por um ser humano, mas sim gerado automaticamente, o uso de TableGen perde seu apelo.

Na verdade, torna-se até desvantajoso utilizar TableGen, uma vez que este impõe limitações na expressividade do *backend*, ao passo que um *backend* codificado em C++ possui máxima flexibilidade e facilita o desenvolvimento do gerador. Por este motivo, no que diz respeito ao *backend* gerado, apenas informações básicas de nomes de registradores e instruções são codificados em arquivos TableGen. A vantagem desta abordagem é que o código gerado fica mais legível para o programador de *backends* LLVM que já está acostumado com TableGen. Contudo, para tarefas mais complexas que expõem a falta de expressividade da linguagem TableGen, como a especificação do seletor de instruções, o *backend* gerado utiliza código C++ e abandona a sintaxe TableGen.

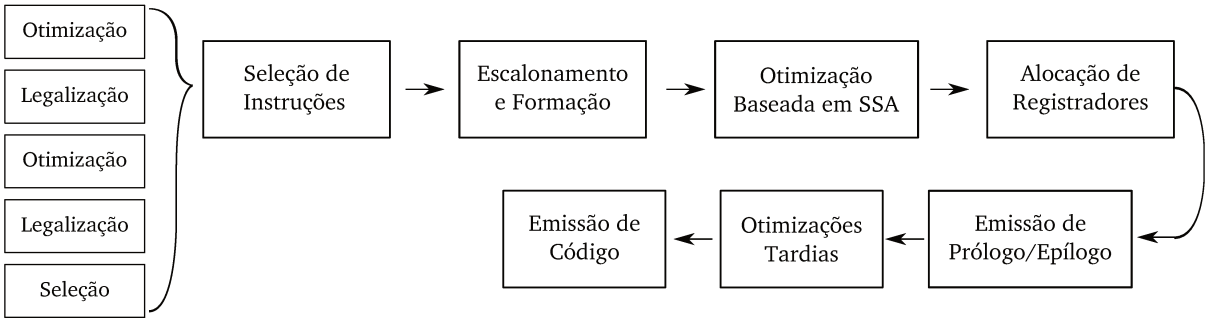



Figura 3.1: Sequência de passos seguidos no *backend* LLVM para traduzir código IR em código da máquina alvo.

A Figura 3.1 mostra a arquitetura geral de um *backend* LLVM, que consiste em um conjunto de passos que gradativamente transformam o programa em linguagem intermediária LLVM para instruções de máquina. Esta arquitetura é conhecida como arquitetura de repositório, em que os passos se comunicam por uma representação comum do dado. No caso do LLVM, esta estrutura é uma instância da classe `SelectionDAG`, uma representação do programa em um grafo direcionado acíclico.

O processo de geração de código é por função, ou seja, cada função é emitida separadamente e completará o fluxo de geração de código de maneira independente das outras. O primeiro passo consiste em simplesmente construir o DAG inicial diretamente a partir do código LLVM *intermediate representation* (IR). Em seguida, um passo de otimizações simples é aplicado, sucedido pelo passo de legalização do DAG. Este passo representa uma importante decisão de projeto do *backend* LLVM. Ao invés de exigir que o seletor de instruções seja complicado o suficiente para abordar todas as construções possíveis da linguagem intermediária LLVM, a etapa de seleção de instruções é precedida pelo passo de legalização, que consiste em transformar casos de código não previstos pelo seletor de código em casos tratáveis. Esta abordagem simplifica o desenvolvimento do seletor de instruções. Vários passos de legalização sucedidos por otimização são aplicados, até que o seletor de instruções propriamente dito (que utilizará um algoritmo de casamento de padrões no DAG) irá finalmente converter os nós restantes da linguagem intermediária LLVM em nós que representam instruções da máquina alvo. A seleção de instruções é sucedida pelo escalonador de instruções, que atribui uma ordem linear ao grafo, que pode ser linearizado de diversas formas.

Após o escalonamento, o alocador de registradores irá atribuir registradores reais, eliminando instâncias de registradores virtuais do código final. Somente após conhecido o programa com registradores reais e o número exato de posições na pilha (que pode aumentar devido a falta de registradores reais na arquitetura) é emitido o prólogo e epílogo da função em processamento, ajustando corretamente a pilha. Finalmente, otimizações do tipo *peephole* podem ser aplicadas e o emissor de código em linguagem de montagem é acionado para que o arquivo de saída seja escrito.

A infraestrutura LLVM faz uso extensivo de orientação a objetos e o projeto do *backend* não é exceção. O componente do sistema conhecido como gerador de código independente de máquina compreende um conjunto de classes base para qualquer *backend*. Estas classes contêm código pronto para lidar com tarefas rotineiras que qualquer *backend* LLVM, independentemente da arquitetura, precisa tratar. O objetivo é fazer com que o projeto de um *backend* específico se limite a definir subclasses daquelas já existentes no gerador de código independente de máquina e adaptar apenas algumas funções membro que respondem pela máquina alvo. Este conjunto de classes básicas que todo *backend* LLVM deve derivar será descrito nas próximas subseções. Paralelamente à apresentação da biblioteca de *backend* LLVM, também será delineado o esqueleto de código mínimo utilizado pelo gerador `acllvmbe` como modelo para gerar automaticamente um *backend* para LLVM.

## 3.2 Arquitetura do *Backend* LLVM

### 3.2.1 TargetMachine

A classe `TargetMachine` constitui o *façade* do componente que representa um *backend* LLVM. Possui funções membro para solicitar qualquer tipo de informação ou ação necessária para a geração de código. Isto é realizado delegando o trabalho para outras classes do *backend*, com funções `get*Info()`, que retornam instâncias para as respectivas classes responsáveis por um aspecto específico da tarefa. Por exemplo, `getRegisterInfo()`, que retornará uma instância de `TargetRegisterInfo`, responsável por fornecer informações e lidar com registradores da arquitetura alvo.

| Nome | Descrição |
|---|---|
| `getInstrInfo()` | Fornece acesso a uma instância da classe $\xi$`InstrInfo` |
| `getFrameInfo()` | Fornece acesso a uma instância da classe `TargetFrameInfo` |
| `getRegisterInfo()` | Fornece acesso a uma instância da classe $\xi$`RegisterInfo` |
| `getTargetLowering()` | Fornece acesso a uma instância da classe $\xi$`TargetLowering` |
| `addInstSelector()` | Adiciona ao *pipeline* LLVM o passo de seleção de instruções |
| `addAssemblyEmitter()` | Adiciona ao *pipeline* LLVM o passo de emissão de código |

Tabela 3.1: Principais funções membro da classe $\xi$`TargetMachine`, em que $\xi$ é o nome da arquitetura alvo. A implementação específica destas funções é gerada automaticamente por `acllvmbe`.

Esta classe deve ser necessariamente especializada por um *backend*, se o *backend* pretende usar a biblioteca do gerador de código independente de máquina oferecido pelo LLVM. No caso do *backend* gerado por `acllvmbe`, uma subclasse de `TargetMachine` chamada $\xi$`TargetMachine` é criada, em que $\xi$ é o nome da arquitetura alvo.

### 3.2.2 TargetLowering

`TargetLowering` é a classe que captura o importante conceito que a fase de legalização representa na seleção de instruções do LLVM. Para decidir se um `SelectionDAG` é legal ou não, uma instância de `TargetLowering` é consultada. Ainda, se alguma instrução da linguagem intermediária LLVM deve ser traduzida de maneira customizada por código C++ específico, a subclasse que deriva `TargetLowering` na máquina alvo, $\xi$`TargetLowering`, deverá conter o código responsável por esta tarefa.

A superclasse `TargetLowering` contém diversas simplificações genéricas, úteis para todos os *backends*, para ajudar na conversão de linguagem intermediária LLVM. Se a

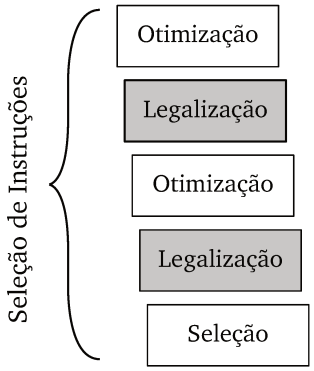



Figura 3.2: Passos executados na fase de seleção de instruções do backend LLVM destacando-se as etapas em que a classe `TargetLowering` participa diretamente.

máquina alvo não lida com inteiros de 16 bits, basta que a subclasse ξ`TargetLowering` informe que inteiros de 16 bits devem ser promovidos para 32 bits, por exemplo. Além de simples conversão de tipos, `TargetLowering` também realiza conversão de operações que não são diretamente suportadas pelo *backend*. A Figura 3.2 esquematiza os passos executados na fase de seleção de instruções do *backend* LLVM e destaca em quais etapas a classe `TargetLowering` atua.

| **Nome** | **Descrição** |
|---|---|
| `LowerOperation()` | Legaliza uma operação da linguagem intermediária LLVM que foi sinalizada para ser transformada com código C++ customizado. |
| ξ`TargetLowering()` | O construtor da classe contém o código mais importante, pois é onde o registro de todos os tipos e operações legais é realizado e, para os tipos e operações ilegais, qual ação tomar para providenciar a legalização. |

Tabela 3.2: Principais funções membro da classe ξ`TargetLowering`. A implementação específica destas funções é gerada automaticamente por `acllvmbe`.

### 3.2.3 TargetRegisterInfo

Esta classe, bem como sua subclasse específica de *backend*, ξ`RegisterInfo`, são utilizadas para encapsular todas as informações relacionadas ao banco de registradores da máquina

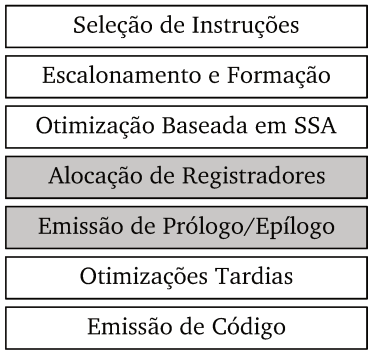



Figura 3.3: A arquitetura geral de um backend LLVM destacando-se as etapas em que a classe TargetRegisterInfo participa diretamente.

alvo. Para ajudar o programador, o LLVM dispõe da possibilidade de se codificar parte desta informação em arquivos TableGen. No momento da compilação do projeto, a ferramenta `tablegen` é chamada e um arquivo com código C++ é gerado automaticamente a partir da descrição dos registradores em TableGen, contida no arquivo ξ`RegisterInfo.td`.

O trecho de informação que deve ser escrito em C++ desta parte do *backend* diz respeito a funções de auxílio ao alocador de registradores, que precisa de algoritmos para manipular diretamente a estrutura de dados do LLVM que representa instruções de máquina (`MachineInstr`), bem como as funções membro para emissão de prólogo e epílogo, que são cruciais para a conclusão da geração de código pelo *backend*. No caso do *backend* gerado por `acllvmbe`, o código C++ para emissão de prólogo e epílogo é gerado automaticamente utilizando informações sobre instruções da máquina alvo para ajustar a pilha. Repare que esta classe codifica majoritariamente informações de domínio da ABI da máquina alvo. A Figura 3.3 esquematiza a arquitetura geral de um *backend* LLVM destacando os componentes onde a classe `TargetRegisterInfo` atua.

| Nome | Descrição |
|---|---|
| `getCalleeSavedRegs()` | Informa quais registradores devem ser salvos pela função chamada. |
| `getReservedRegs()` | Informa quais registradores não devem ser usados pelo alocador de registradores. |
| `eliminateFrameIndex()` | Contém código para transformar um índice de pilha para instruções de máquina que calculam exatamente o endereço da posição no *frame*. |
| `emitPrologue()` | Chamada após a alocação de registradores, deve emitir instruções de máquina relativas ao prólogo da função. |
| `emitEpilogue()` | Chamada após a alocação de registradores, deve emitir instruções de máquina relativas ao epílogo da função. |

Tabela 3.3: Principais funções membro da classe $\xi$`RegisterInfo`. A implementação específica destas funções é gerada automaticamente por `acllvmbe`.

### 3.2.4 TargetInstrInfo


Figura 3.4: A arquitetura geral de um backend LLVM destacando-se as etapas em que a classe `TargetInstrInfo` participa diretamente.

De maneira similar a `TargetRegisterInfo`, as informações contidas nesta classe também são divididas em arquivos C++ e um arquivo TableGen. No *backend* gerado, este arquivo é `ξInstrInfo.td` e possui o objetivo ambicioso de conter uma descrição de todas as instruções da máquina de maneira a gerar completamente o seletor de instruções. Na prática, devido a limitações na expressividade da linguagem TableGen, algumas transformações precisam ser realizadas com código C++. É importante ressaltar que este conceito de *design* do LLVM que visa expressar as informações necessárias para o seletor de instruções em um arquivo com informações declarativas TableGen difere do objetivo desta dissertação em diversos graus. Ainda que o objetivo deste trabalho inclua a geração automática do seletor de instruções baseada em uma descrição declarativa das instruções, a abordagem do LLVM envolve apenas casamento de padrões diretos e, nos casos em que esta abordagem falha, o programador deve escrever código C++ para tratar a situação.

A abordagem desta dissertação envolve gerar completamente o seletor de instruções sem intervenção do usuário, bem como todas as outras partes do *backend* necessárias para se obter um compilador C funcional para a arquitetura alvo. Por este motivo, é razoável que o gerador `acllvmbe` não utilize TableGen nesta etapa, mas que emita código C++ diretamente, para que o projeto não fique restrito às limitações que TableGen possui.

A Figura 3.4 apresenta os componentes afetados por `TargetInstrInfo` na arquitetura geral do *backend*. A subclasse do *backend* gerado `ξInstrInfo` está dividida nos arquivos `ξInstrInfo.cpp`, `ξInstrInfo.h` e `ξISelDAGToDAG.cpp`. Este último é particularmente importante, pois hospeda todo o código do seletor de instruções automaticamente gerado por `acllvmbe` para a arquitetura alvo. Os dois primeiros arquivos contêm código para analisar uma estrutura de lista de `MachineInstr` e identificar alguns padrões, bem como

emitir código específico. Por exemplo, a função membro `isMoveInstr()` deve ser implementada para identificar quando o código está movendo valores entre registradores, ao passo que a função membro `copyRegToReg()` deve ser implementada para gerar código de cópia de valores entre registradores. Como pode ser visto na Tabela 3.4, estas funções não são numerosas e tem o objetivo de apenas auxiliar o alocador de registradores e algumas pequenas otimizações, mas não o de gerar todo tipo possível de código. Esta tarefa é delegada ao algoritmo de casamento de padrões no `SelectionDAG`, que transforma padrões de nós que contêm linguagem intermediária LLVM para nós que contêm instruções da máquina alvo.

O arquivo `ξInstrInfo.td` deve conter a declaração de todas as instruções da máquina e, no caso do projeto `acllvmbe`, é completamente gerado a partir das instruções declaradas no modelo ArchC da arquitetura.

| **Nome** | **Descrição** |
|---|---|
| `isMoveInstr()` | Informa se o código implementa movimentação de valores entre registradores |
| `isLoadFromStackSlot()` | Informa se o código implementa carga de um valor da pilha |
| `isStoreToStackSlot()` | Informa se o código salva um valor na pilha |
| `InsertBranch()` | Insere código para realizar um salto |
| `copyRegToReg()` | Insere código para copiar valores entre registradores |
| `storeRegToStackSlot()` | Insere código para salvar um valor na pilha |
| `storeRegToAddr()` | Insere código para salvar um valor em posição arbitrária de memória |
| `loadRegFromStackSlot()` | Insere código para carregar um valor da pilha |
| `loadRegFromAddr()` | Insere código para carregar um valor de uma posição arbitrária de memória |

Tabela 3.4: Principais funções membro da classe `ξInstrInfo`. A implementação específica destas funções é gerada automaticamente por `acllvmbe`.

### 3.2.5 AsmPrinter

Este componente do *backend* representa um passo separado a ser executado no fluxo LLVM. Este passo deve utilizar uma lista de instruções de máquina para imprimir um arquivo com o código correspondente em linguagem de máquina. Sua função em comparação com a visão geral do *backend* é ilustrada na Figura 3.5, destacando o componente como o último passo para encerrar a geração de código.

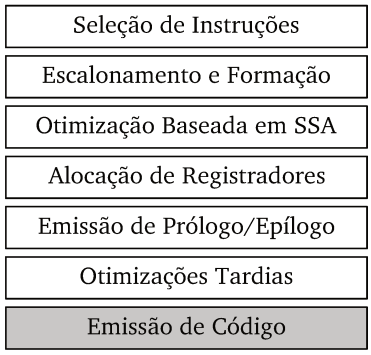



Figura 3.5: A arquitetura geral de um backend LLVM destacando-se as etapas em que a classe `AsmPrinter` participa diretamente.

A classe contém código para lidar com a impressão de maneira genérica e satisfatória para todos os *backends*, pois a sintaxe em linguagem de montagem para cada instrução da máquina está disponível no arquivo $\xi$`InstrInfo.td`. Os *backends* devem derivar uma subclasse, $\xi$`AsmPrinter` no caso do sistema `acllvmbe`, para implementar funções específicas da máquina e lidar com operandos especiais.

No caso do *backend* gerado por `acllvmbe`, este componente trabalha de maneira muito semelhante ao montador gerado automaticamente por `acasm` [11], utilizando as mesmas informações contidas no modelo ArchC para escrever o código em linguagem de montagem da máquina corretamente. Mais tarde, para completar o fluxo de criação de programas das ferramentas ArchC, este código em linguagem de montagem deverá ser processado pelo montador também gerado por uma ferramenta ArchC para finalmente produzir código objeto para a máquina alvo. O *backend* gerado no trabalho desta dissertação se limita a gerar código em linguagem de montagem, e não linguagem de máquina diretamente, uma vez que o montador já está disponível e pode ser gerado automaticamente a partir de um modelo ArchC.

# Capítulo 4

# Extensão da Linguagem ArchC

Este capítulo discute a extensão da linguagem ArchC necessária para adicionar capacidade de expressão à sua gramática para que um modelo possa conter as informações requeridas pela geração automática de *backends* do projeto `acllvmbe`. A descrição destas informações em um modelo ArchC exige maior esforço do projetista. Porém, isto permite alcançar o objetivo principal deste trabalho, mas não limitado a este, que é a geração automática de um *backend* para o compilador LLVM. A extensão da linguagem também permite a atuação de outras ferramentas, como por exemplo o gerador automático de programas em linguagem de montagem para testar cada instrução separadamente. Em cada um desses programas, uma única instrução é repetida diversas vezes, mas com operandos diferentes. Estes podem ser usados em um *testbench* para verificação do simulador da máquina alvo, ou, por exemplo, para estimativa do consumo de energia por instrução, se uma infraestrutura para este fim estiver disponível.

## 4.1 Princípios de Projeto

Dois compromissos importantes foram adotados quanto à elaboração da extensão da linguagem. O primeiro é o de que, se a extensão visa principalmente o suporte a geração de compiladores para a arquitetura alvo, o esforço necessário para adicionar estas informações ao modelo deve ser consideravelmente menor do que aquele necessário para criar um compilador manualmente. Do contrário, não há sentido em utilizar uma ferramenta ArchC para gerar o compilador.

O segundo compromisso diz respeito ao nível de abstração e tipo de informações exigidas do projetista do modelo. Por um lado, parece vantajoso incluir no modelo ArchC informações de uso específico para a criação de um *backend* LLVM. Por exemplo, definir uma notação em que o usuário descreva como um padrão de linguagem intermediária LLVM específico deve ser mapeado para instruções da máquina alvo. Esta abordagem

oferece bastante flexibilidade ao usuário, uma vez que ele pode inspecionar diretamente como o seletor de instruções funciona e decidir exatamente quais instruções serão usadas, garantindo maior controle do usuário sobre o *backend* LLVM criado. Por outro lado, existem desvantagens nesta abordagem. O usuário precisa ter conhecimento específico sobre a linguagem intermediária do LLVM para entender os padrões e, todo o esforço de engenharia do seletor de instruções é manual. De fato, o trabalho para descrição dos padrões seria comparável ao de escrever um *backend* em C++ manualmente, ou utilizando TableGen, como discutido no Capítulo 3. Portanto, seria questionável se o esforço em utilizar a linguagem ArchC e, por conseguinte, uma nova camada de abstração, é compensado. Esta abordagem de descrição de padrões foi adotada inicialmente pela ADL LISA [38], mas foi substituída um ano mais tarde [15].

Outra desvantagem importante de incluir no modelo ArchC informações de uso específico para a geração do *backend* para o LLVM é a própria restrição imposta de dependência do projeto do LLVM. A reusabilidade destas informações seria questionável, uma vez que são específicas para o LLVM e isto limitaria severamente o número de projetos futuros que poderiam usufruir da disponibilização destas informações. O segundo compromisso, então, visa a expressão de informações genéricas sobre a arquitetura alvo, sem nenhum viés para alguma ferramenta. Estas informações devem ser declarativas e enunciar as características da arquitetura alvo. Por exemplo, o modelo não deve conter código C++ que será embutido no *backend* para fins de prover alguma funcionalidade necessária no compilador alvo. É de responsabilidade da ferramenta de geração de *backend* realizar uma série de decisões de projeto de algoritmos a serem utilizados no *backend* e codificá-los em C++, com base apenas nas características descritas no modelo.

Os dois compromissos de projeto adotados são complementares. Ao exigir menos informação específica no modelo e deixá-lo isento de conhecimento específico do domínio do problema a ser resolvido (englobando compiladores e seletores de instrução), o modelo também fica consideravelmente mais fácil de ser escrito. Isto acontece porque o projetista não precisa ter conhecimento específico sobre um compilador para criar um novo *backend*. Ele apenas deve descrever características da arquitetura alvo e usar a ferramenta `acllvmbe`. Desta forma, há um grande atrativo para os usuários da ferramenta, que é a completa automação da geração do compilador. O grau de flexibilidade e customização do comportamento do compilador é certamente comprometido, mas isso é uma consequência natural da programação automática que deve ser levada em consideração pelo time de desenvolvimento das ferramentas de suporte a criação de software da plataforma alvo.

## 4.2 Informações Necessárias no Modelo

A lista a seguir contém, em alto nível, as informações de uma máquina alvo arbitrária necessárias para a criação de um *backend* de compilador.

**Formato de codificação das instruções:** Para a emissão de código de máquina, é necessário conhecer como as instruções são codificadas em formato binário. Esta informação não é usada diretamente pelo projeto `acllvmbe`, pois o produto final da compilação é um arquivo em linguagem de montagem. Entretanto, o montador gerado pela ferramenta ArchC `acasm` [11] utiliza esta informação e preenche a lacuna necessária para traduzir código em linguagem de montagem, gerado pelo *backend* LLVM, para código de máquina executável pelos simuladores ArchC.

**Identificação das instruções:** Informações básicas como um identificador único para cada instrução, número de operandos e sintaxe da linguagem de montagem orientando como usar a instrução em um programa escrito em linguagem de montagem da máquina alvo. Desta maneira, há informação suficiente para emitir código em linguagem de montagem a partir de uma lista identificando as instruções contempladas na arquitetura.

**Semântica das instruções:** A informação de semântica deve capturar o comportamento da instrução e tem por objetivo explicitar qual expressão é calculada com o uso da instrução. É através desta informação que o gerador de *backend* tira conclusões sobre quais instruções devem ser usadas para construir um pequeno trecho de programa da máquina alvo. Este trecho de programa deve ter uma especificação formal no mesmo formato da semântica das instruções.

**Layout de dados:** Informações sobre *endianess*, tamanho do inteiro usado para endereçar a memória, bem como tamanho e alinhamento de todos tipos fundamentais conforme especificado pela ABI.

**Registradores:** É necessário atribuir nomes a cada registrador para que seja possível usá-lo como operando das instruções. Ainda, é necessário também o conhecimento sobre convenções da ABI: quais registradores são reservados, quais são salvos pela função chamada, qual registrador é reservado para guardar o endereço da pilha e qual registrador é usado para guardar o endereço do retorno de funções.

**Convenção de chamadas:** De acordo com a ABI, deve ser especificado como os parâmetros de uma função devem ser passados, isto é, por meio da pilha, de registradores, ou um misto das duas abordagens. O mesmo vale para a maneira como uma função retorna um valor.

## 4.3 Informações Existentes

Algumas das informações necessárias enunciadas na Seção 4.2 estão disponíveis em um modelo ArchC não estendido, ao passo que outras precisam ser incluídas.

A organização básica de um modelo de arquitetura descrito em ArchC separa-o em informações estruturais, isto é, no formato de uma lista de recursos disponíveis em hardware, e informações da arquitetura do conjunto de instruções, ou *instruction set architecture* (ISA). Contidas na seção ISA do modelo ArchC estão as descrições do comportamento de cada instrução, que consiste em código C++ que manipula o estado do processador (seus elementos estruturais) realizando alguma computação útil.

As informações necessárias já disponíveis em modelos ArchC não estendido estão todas contidas na seção ISA. Primeiramente, a codificação das instruções é necessária para a geração de simuladores para a arquitetura [8] (mais especificamente, o decodificador) e para a geração de montadores [9] e, portanto, já está disponível no modelo ArchC. Além disso, o modelo anterior do ArchC já contém uma forma de identificação e enumeração de todas as instruções da arquitetura, bem como especificação da sintaxe da instrução em linguagem de montagem alvo. A Figura 4.1 contém um exemplo de especificação de alguns formatos e instruções para a arquitetura MIPS [41].

```
1 ac_format tipoR = "%op:6 %rs:5 %rt:5 %rd:5 %shamt:5 %funct:6";
2 ac_instr <tipoR> add, addu;
3
4 add.set_asm("add %reg, %reg, %reg", rd, rs, rt);
5 add.set_decoder(op=0x00, funct=0x20);
6
7 addu.set_asm("addu %reg, %reg, %reg", rd, rs, rt);
8 addu.set_decoder(op=0x00, funct=0x21);
```

Figura 4.1: Trecho de exemplo de um modelo ArchC para MIPS, seção ISA.

O exemplo da Figura 4.1 sugere como a sintaxe em linguagem de montagem, a codificação e a identificação de cada instrução podem ser extraídas. Para isto, basta utilizar a biblioteca atual das ferramentas ArchC que contém um pré-processador para construir a árvore de sintaxe abstrata, ou *abstract syntax tree* (AST). É importante atentar ao fato de que a semântica da instrução, conforme especificada na Seção 4.2, poderia ser extraída do arquivo C++ do modelo ISA ArchC que contém código que define o comportamento da instrução em relação a mudança de estado do processador. Contudo, a linguagem ArchC não restringe a implementação do comportamento das instruções em C++. Como o projetista possui liberdade para utilizar qualquer construção C++ para definir o comportamento da instrução, na prática, é inviável derivar uma expressão de

semântica da instrução a partir do código C++ que descreve o seu comportamento. A Figura 4.2 apresenta um exemplo de definição do comportamento da instrução `add` da arquitetura MIPS.

```
1 void ac_behavior( add )
2 {
3   RB[rd] = RB[rs] + RB[rt];
4 };
```

Figura 4.2: Exemplo de descrição do comportamento da instrução `add` do MIPS em ArchC. O símbolo `RB` deve ser previamente declarado no modelo estrutural como o banco de 32 registradores da arquitetura MIPS. `rd`, `rs` e `rt` são operandos da instrução e aparecem declarados na linha 1 da Figura 4.1.

O projeto para viabilização de síntese de processadores em hardware a partir de um modelo ArchC [31] definiu um subconjunto de construções C++ que podem ser utilizadas para a descrição de instruções de forma que elas sejam sintetizáveis em hardware. Esta abordagem não foi adotada no projeto `acllvmbe`. Ao invés disso, informações de árvores de expressão adicionais são necessárias para expressar o comportamento de cada instrução. Todavia, não há impedimentos teóricos para se realizar a extração de semântica através da tradução do código C++ que descreve o comportamento de uma instrução para a representação de semântica adotada. Apesar de ser tecnicamente desafiante realizar esta tradução, uma vez que diversas simplificações seriam necessárias para extrair uma árvore de expressão simples o suficiente para ser útil na geração de código, ainda seria possível gerar a tradução parcialmente e oferecê-la ao projetista como um ponto de partida para descrever o comportamento das instruções com árvores de expressão. Esta tarefa pode ser futuramente implementada, diminuindo ainda mais a quantidade de informações extras necessárias no modelo ArchC para a geração de *backend* de compilador C.

## 4.4 Novas Construções

### 4.4.1 Árvore Semântica

Um importante conceito criado para adicionar à linguagem ArchC a capacidade de capturar a semântica das instruções (veja definição da semântica na Seção 4.2) é o de árvore semântica. Trata-se de uma árvore em que os nós intermediários são chamados de operadores e são identificados por um nome e pelo número de filhos, chamados operandos. Os nós folha correspondem a um elemento armazenador de estado da plataforma (registrador ou memória), constantes ou imediatos. Utilizando uma representação linearizada da árvore,

o projetista é capaz de descrever sucintamente qual o efeito de uma instrução no estado do processador. Esta informação é vital para a síntese do seletor de instruções do compilador. A ideia principal é de que qualquer modificação que uma instrução provoque em um elemento de seu estado possa ser completamente especificada por uma árvore semântica. Portanto, se uma instrução modifica mais de um elemento, uma floresta semântica é suficiente para especificar completamente a instrução ao se utilizar uma árvore para cada elemento modificado. Este conceito foi baseado na notação *register transfer lists* (RTL).

No contexto de árvores de semântica, os operadores representam uma manipulação dos valores contidos em seus filhos. Se o operador não é raiz da árvore, produzirá um resultado para o nó pai. Toda árvore de semântica necessariamente deve expressar uma mudança de fluxo ou transferência de um valor para um elemento de estado do processador. A Figura 4.3 ilustra um exemplo de árvore de semântica para a instrução `add` da arquitetura MIPS. Os operadores estão representados com elipses de linha contínua, ao passo que os nós folha estão representados com elipses de linha tracejada. Neste caso, o operador `add` é uma soma de seus 2 operandos, retornando o resultado para o nó pai, o operador `transfer`, que é o nó raiz e possui o comportamento de copiar o valor do filho à direita para o filho à esquerda. Os nós folha possuem nomes sufixados por números. Este número diz respeito à posição do operando na sintaxe de linguagem de montagem (operando 1 é o primeiro operando que aparece na sintaxe, lida da esquerda para a direita, e assim por diante). Ainda, os nós folha são do tipo `GPR` (*general purpose register*), ou seja, esta árvore representa uma soma que opera sobre dois registradores e armazena o resultado em um terceiro registrador.

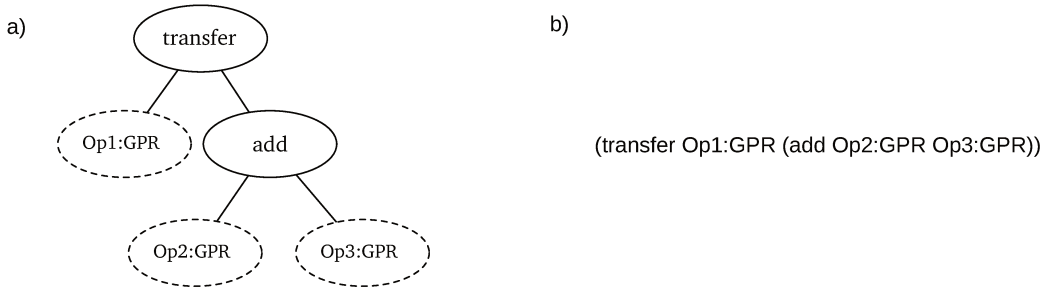



Figura 4.3: A árvore semântica da instrução `add` da arquitetura MIPS: (a) Representação gráfica; (b) Forma linearizada.

### Definição de Operadores e Tipos de Operandos

A árvore semântica é intrinsecamente extensível em sua representação, pois não impõe restrições sobre quantos tipos diferentes de operadores ou nós folha pode ter. A sintaxe para definir novos tipos de operadores para a árvore exige apenas o nome e o número de filhos, para viabilizar a análise de consistência da árvore. Nenhum operador é definido *a priori* mas, na prática, o projeto `acllvmbe` conta com um repositório de regras com a definição de diversos operadores e tipos, bem como regras para manipular estes operadores.

```
define operator X as arity N;
define operand Y as size M;
```

Figura 4.4: Sintaxe para definir novo operador e novo operando.

A Figura 4.4 apresenta a sintaxe para definir novos operadores e tipos de nós folha, em que `X` é o nome do novo operador, `N` o número de filhos, `Y` o nome do novo tipo de nó folha e `M` o seu tamanho em bits.

### Operadores Especiais

Alguns nomes de operadores são utilizados para sinalizar o uso de um operador especial. Estes operadores são necessários para prover uma forma canônica de se descrever algumas expressões. Dessa maneira, a análise de casos para lidar com diferentes arquiteturas é simplificada se há concordância entre diferentes modelos em usar os mesmos nomes de operadores para representar uma determinada operação. A demonstração mais óbvia desta necessidade é a definição do operador `transfer`. Toda vez que uma movimentação de dados é necessária, um operador binário com este mesmo nome deve ser usado. Um outro exemplo é o operador especial `memref` unário. O seu único operando deve avaliar um endereço de memória a ser acessado por `memref`. Caso este operador seja usado como primeiro filho de `transfer`, a expressão representa salvar um valor na memória. Caso o operador seja usado em qualquer outra situação, trata-se de uma carga da memória. Alguns exemplos são mostrados na Tabela 4.1. A listagem completa encontra-se no Apêndice A.

### Tipos de Nós Folha

Os nós folha de uma árvore semântica representam a unidade de informação que é utilizada como entrada ou saída de uma expressão calculada. Dada uma função de transição de estado do processador, o subconjunto de elementos do estado que é lido e que é alterado deve necessariamente ser representado como nó folha. Contudo, o nó folha não se restringe

| Nome | Tipo | Função |
|---|---|---|
| transfer | Binário | Copia o valor do filho direito para o filho esquerdo. |
| memref | Unário | Acesso à memória. |
| pcrelative | Unário | Acesso à memória relativo ao *program counter* (código independente de posição). |
| call | Unário | Salta para um endereço, salvando o próximo endereço para retorno. |
| ret | Folha | Salta para endereço de retorno. |
| jump | Unário | Salta para um endereço. |

Tabela 4.1: Alguns operadores com funcionalidades pré-determinadas na árvore semântica.

a representar apenas elementos armazenadores de estado do processador. Um nó folha pode ser também um número imediato que é armazenado diretamente na codificação da instrução ou ainda um número constante que não é um operando da instrução, mas apenas agrega significado a um operador pai. A Tabela 4.2 sumariza a sintaxe para representar diferentes tipos de nós folha.

| Tipo | Sintaxe |
|---|---|
| Registrador | `Nome:Classe do registrador` |
| Imediato | `imm:Nome:Tipo` |
| Constante | `const:Tipo:Valor` |

Tabela 4.2: Sintaxe para definir diferentes tipos de nós folha da árvore semântica.

### 4.4.2 Registradores e Classes de Registradores

A especificação anterior da linguagem ArchC previa a declaração de elementos estruturais como banco de registradores. Entretanto, não possuía mecanismos para nomeação explícita dos registradores individuais nem para especificação dos tipos de dados armazenados em tais registradores (como, por exemplo, para a distinção entre ponto flutuante e ponto fixo). Para resolver o problema da nomeação dos registradores, a extensão da linguagem ArchC para o projeto `acasm` [10] adicionou o conceito de mapas de símbolos. Isto é, para que o projetista descreva a sintaxe em linguagem de montagem de uma ins-

trução que poderia admitir um registrador como operando, basta que defina um *mapa* com símbolos que representam os nomes dos registradores.

Todavia, esta extensão ainda é insuficiente para os propósitos listados na Seção 4.2 referente a registradores. Para atender a estes objetivos, criou-se uma sintaxe para descrição dos registradores da arquitetura baseada em classes. Uma classe de registradores é um conjunto de registradores que compartilham o mesmo tipo. Este tipo pode ser qualquer um definido obedecendo a sintaxe apresentada na Seção 4.4.1.

```
1 define registers GPR:int as (
2   $0 $1 $2 $3 $4 $5 $6 $7 $8 $9 $10
3   $10 $11 $12 $13 $14 $15 $16 $17 $18 $19 $20
4   $20 $21 $22 $23 $24 $25 $26 $27 $28 $29 $30
5   $31 );
```

Figura 4.5: Exemplo de construção para definir classes de registradores para a arquitetura MIPS.

A Figura 4.5 apresenta um exemplo de definição da classe de registradores `GPR` da arquitetura MIPS. Neste exemplo, a classe `GPR` foi definida como pertencente ao tipo `int`, que necessariamente deve estar declarado previamente. Os símbolos contidos entre parênteses, como por exemplo `$17`, definem os nomes dos registradores pertences à classe `GPR`. Após esta declaração, uma árvore semântica pode utilizar nós folha do tipo `GPR`, ou o nome da classe de registradores criada, para representar uma instrução que manipula diretamente estes registradores. Um exemplo está ilustrado na Figura 4.3 com a árvore semântica da instrução `add`.

### 4.4.3 Informações Dependentes de ABI

Para estender o modelo com capacidade para expressar diversos aspectos da ABI de uma plataforma alvo, criou-se uma única construção com o objetivo de englobar todos esses dados. A Figura 4.6 apresenta um exemplo completo para a arquitetura ARM. Cada construção de definição da ABI será discutida em detalhes nas próximas subseções.

#### Registradores Salvos pela Função Chamada

A linha 2 da Figura 4.6 mostra como é feita a definição dos registradores que, segundo a convenção da ABI adotada, devem ser salvos por uma função antes de serem utilizados. Estes são também conhecidos simplesmente como registradores salvos, ou não temporários. Repare que os símbolos `r4`, `r5`, `r6` e `r7` devem ter sido declarados previamente em uma construção do tipo *classe de registradores* (Seção 4.4.2). Esta é uma restrição válida

```
1 define abi as (
2   define callee save registers as (r4 r5 r6 r7);
3   define reserved registers as    (r9 r10 r13 r14 r15);
4   define auxiliar registers as    (r9 r10);
5   define calling convention for int as (r0 r1 r2 r3);
6   define calling convention for int as stack size 4 alignment 4;
7   define return convention for int as (r0 r1);
8   define stackpointer register as r13;
9   define framepointer register as r11;
10  define return register as r14;
11  define stack grows down alignment 8;
12  define pcoffset -8;
13 );
```

Figura 4.6: Exemplo de construção para definir informações ligadas à ABI da arquitetura ARM.

também para todas as outras definições de ABI que referenciam nomes de registradores da arquitetura alvo.

### Registradores Reservados

Seguindo o exemplo, a linha 3 da Figura 4.6 define, para a arquitetura ARM, o nome dos registradores que são reservados. Estes registradores são excluídos do algoritmo de alocação de registradores. No caso do ARM, os registradores excluídos correspondem ao registrador de pilha `r13`, o registrador que contém o endereço da instrução em execução *program counter*, `r15`, o registrador que contém o endereço de retorno da função `r14` e os dois registradores auxiliares `r9` e `r10` que serão apresentados a seguir.

### Registradores Auxiliares

A linha 4 da Figura 4.6 explicita a notação para declarar uma lista de registradores auxiliares. Estes registradores auxiliares são excluídos do algoritmo de alocação de registradores e são reservados para uso de funções específicas do *backend*. Devem ser de caráter temporário (não salvos). Como são registradores que não são alocados, estão sempre livre para serem utilizados pelo *backend* em passos que ocorrem após a alocação de registradores como, por exemplo, na emissão do prólogo ou epílogo da função. Como o *backend* não pode emitir código com registradores virtuais após a fase de alocação de registradores, a lista de registradores auxiliares é utilizada para prover registradores reais para os trechos de código que utilizam valores intermediários. A Seção 6.1.2 apresenta mais detalhes sobre a utilização de registradores auxiliares.

### Convenção de Chamada

A maneira de se expressar a convenção de chamadas da ABI possui algumas características específicas. Um exemplo aparece nas linhas 5 a 7 da Figura 4.6. Em particular, a ordem com que estas sentenças aparecem tem importância e define a prioridade do recurso a ser utilizado para armazenar os parâmetros da função. A construção `define calling convention` especifica onde os parâmetros da função serão passados, enquanto que a construção `define return convention` especifica onde o valor de retorno da função será armazenado.

A ocorrência de mais de uma construção do tipo *calling convention* especifica que, caso o recurso especificado pela sentença anterior se esgote, o recurso da sentença atual deverá ser utilizado. No exemplo da arquitetura ARM, as linhas 5 a 7 definem que os parâmetros para funções devem ser passados primeiramente através dos registradores `r0`, `r1`, `r2` e `r3`. Caso a função tenha mais de 4 parâmetros, a linha 6 define que o restante dos parâmetros devem ser passados pela pilha em unidades de 4 bytes em posições alinhadas também a 4 bytes. Finalmente, a linha 7 atribui aos registradores `r0` e `r1` a tarefa de carregar o valor de retorno da função, com prioridade para `r0`.

### Atribuição de Registradores Especiais

Alguns registradores possuem um papel específico sobre como devem ser utilizados pelo software e esta é puramente uma convenção da ABI. A construção ABI é capaz de especificar 3 papéis específicos: o registrador responsável por armazenar o ponteiro da pilha, o ponteiro para o *frame* da função atual e o endereço de retorno da função atual. Esta especificação, na Figura 4.6, acontece nas linhas 8 a 10 e atribui os papéis a, respectivamente, `r13`, `r11` e `r14`.

Note que o *backend* gerado precisa conhecer estas informações para utilizar os registradores adequados quando produzir trechos de programa que manipulam a pilha ou retornam de uma função.

### Características da Pilha

Também de competência da ABI é a especificação de qual tipo de pilha deve ser implementada pelo software. A construção de ABI criada consegue expressar pilhas que crescem para cima ou para baixo, bem como o seu alinhamento. A linha 11 da Figura 4.6 informa que a pilha ARM cresce para baixo e alinha-se em limites de 8 bytes.

**Endereçamento Relativo ao PC**

Finalmente, a última informação encapsulada na construção de ABI é o deslocamento aplicado ao *program counter* no momento de calcular um endereço relativo ao `PC`. Esta informação não é necessariamente da ABI, mas natural da arquitetura alvo. A linha 12 da Figura 4.6 informa que, após consultado o valor do *program counter*, deve-se somar um deslocamento de -8 para obter o valor real do endereço da instrução atual.

### 4.4.4 Comportamento de Instruções

Conforme discutido na Seção 4.4.1, o comportamento da instrução deve ser expresso por uma floresta semântica. Supondo que todos os tipos, operandos e classes de registradores referenciados por uma floresta semântica específica já tenham sido declarados anteriormente, a definição da floresta semântica de uma instrução em particular acontece como no exemplo da Figura 4.7. O nome `addu` relaciona-se diretamente com uma, e apenas uma, construção `set_asm` para a instrução `addu` do modelo ISA ArchC do MIPS.

```
1 define instruction addu semantic as (
2   (transfer Op1:GPR (+ Op2:GPR Op3:GPR));
3 ) cost 1;
```

Figura 4.7: Exemplo de definição da floresta semântica da instrução `addu` para a arquitetura MIPS.

Uma única instrução ArchC pode ter mais de uma definição `set_asm`, segundo o mecanismo de *sobrecarga de sintaxe* [10]. Isto acontece quando uma instrução possui mais de uma sintaxe em linguagem de montagem, ou compartilha construções com formatos iguais mas que diferem em poucos campos na maneira de codificar. O *backend* precisa saber exatamente qual dessas sintaxes utilizar e, por este motivo, uma definição de semântica, conforme apresentada na Figura 4.7, corresponde a apenas uma sintaxe. A sintaxe exata a qual a semântica corresponde é determinada por ordem de posição no código e, portanto, mais de uma semântica pode existir com o mesmo nome de instrução. Dessa forma, a segunda semântica com o nome `addu`, por exemplo, corresponderia à segunda sintaxe de linguagem de montagem para a instrução `addu` do modelo ArchC MIPS. Isto é ilustrado na Figura 4.8.

A abordagem ideal seria atrelar cada floresta semântica a uma sintaxe de linguagem de montagem e não a uma instrução. Mas como a linguagem ArchC não dá nomes a cada sintaxe, mas sim a cada instrução, o mecanismo de relacionamento posicional foi adotado. Observe que na Figura 4.7 está definido também um *custo* para esta floresta semântica. Este custo deve refletir o esforço computacional da plataforma para executar

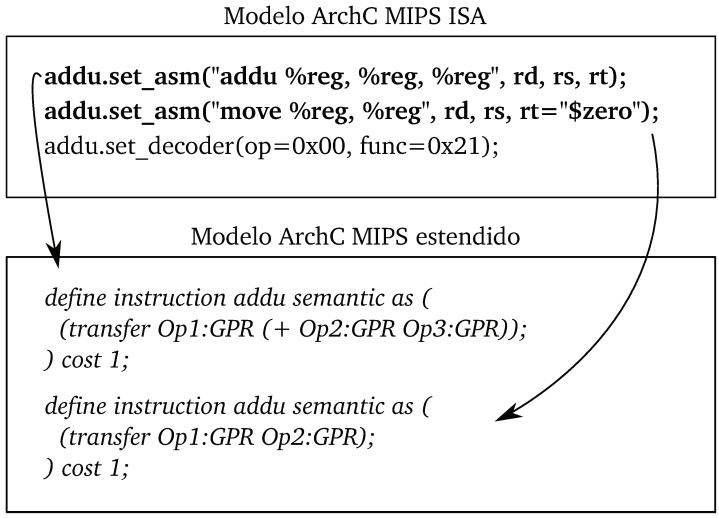



Figura 4.8: A correspondência de especificação de floresta semântica para cada sintaxe de linguagem de montagem obedece a ordem de posição.

esta instrução. No momento de geração de código pelo *backend*, a busca ocorre por uma combinação de instruções que ofereçam o menor custo.

### Atribuição de Operandos Literais

Caso uma árvore semântica específica não manipule diretamente todos os operandos disponíveis na sintaxe de linguagem de montagem de uma determinada instrução, o *backend* gerado pode não saber qual valor utilizar para o operando desconhecido. Para este fim, deve-se utilizar uma construção para atrelar literais a operandos que não são referenciados na árvore semântica. Na Figura 4.9, por exemplo, o operando 1 não é mencionado em nenhum nó folha da árvore semântica `(jumpnz $14:GPR imm:Op2:int)`. O único operando referenciado é o de número 2, que é um inteiro codificado como imediato na instrução. O outro nó folha faz uma referência direta ao registrador `$14` especificamente, mas não diz que este deve ser o operando 1. Para isso, a construção `let Op1 = ¨$14¨` é utilizada para atrelar a literal ao operando 1. Note que a literal é uma string qualquer e não necessariamente deve fazer referência a um símbolo, como o nome de registrador previamente declarado.

### Fragmentos de Árvore Semântica

Em casos onde diversas instruções compartilham um subgrafo de árvore semântica em comum, é possível declarar este subgrafo como um *fragmento* de semântica e instanciá-lo

```
1 define instruction bgtz semantic as (
2   let Op1 = "$14" in
3     (jumpnz $14:GPR imm:Op2:int);
4 ) cost 1, has_delay_slot;
```

Figura 4.9: Definição da floresta semântica da instrução `bgtz` do MIPS.

em diferentes florestas semânticas utilizando apenas o nome do fragmento. Este recurso é útil para especificar modos de endereçamento e evitar a tarefa tediosa e passível de erros que ocorre ao reescrever o mesmo trecho diversas vezes. No modelo ARM, por exemplo, um subgrafo recorrente é aquele que expressa um operando registrador que sofre deslocamento à esquerda por um número de casas especificado por um imediato. A construção exata de como declarar este subgrafo como um fragmento é ilustrada na Figura 4.10. Na figura também aparece um exemplo de uso do fragmento com a especificação da floresta semântica da instrução `and1` do modelo ARM. Neste caso, para interpretar corretamente a árvore semântica com o comportamento da instrução, basta substituir o nó folha `RegShifterAddrMode:fragment` pela árvore `(shl Op5:GPR imm:Op7:tgtimm)`.

```
1 define semantic fragment RegShiftedAddrMode as (
2   (shl Op5:GPR imm:Op7:tgtimm);
3 );
4
5 define instruction and1 semantic as (
6   let Op1 = "", Op2 = "" in
7     (transfer Op3:GPR (and Op4:GPR RegShiftedAddrMode:fragment));
8 ) cost 1;
```

Figura 4.10: Definição de fragmento de árvore semântica no modelo ARM estendido e exemplo de uso do fragmento na especificação da floresta semântica da instrução `and1`.

### 4.4.5 Regras de Transformação da Árvore Semântica

Da mesma maneira que árvores semântica são usadas para especificar as instruções sob o ponto de vista do compilador, regras de transformações especificam os operadores da árvore. Suponha, por exemplo, que um novo operador seja definido segundo a sintaxe apresentada na Seção 4.4.1. Toda a informação disponível é um rótulo atribuído a este operador e o número de filhos (operandos) que este deve ter. Todavia, isso não é suficiente para as necessidades do sistema `acllvmbe` que precisa realizar comparações entre duas expressões e concluir se elas são equivalentes. Ainda que uma comparação estrutural

de duas árvores possa determinar se elas são idênticas, está é uma restrição muito forte para se determinar a equivalência. Por esse motivo, regras que especificam quando uma árvore A pode ser transformada em uma árvore B sem que o resultado do programa seja alterado agregam informações valiosas para especificar exatamente o que os operadores significam e como podem ser transformados para provar equivalência entre duas árvores estruturalmente diferentes e distingui-lo dos demais nós.

Uma diferença importante das árvores semântica usadas para especificar regras das árvores semântica usadas para especificar instruções é em relação aos nós folha. Na árvore semântica que especifica o comportamento de uma instrução, o nome do nó folha é usado para fazer a correspondência com os operandos usados na sintaxe de linguagem de montagem (Seção 4.4.1). No caso das regras, o nome dos operandos é um nome arbitrário usado para fazer a correspondência entre árvores diferentes que aparecem na mesma regra. Ainda, a sintaxe do nó registrador é usada mas, no lugar do nome da classe de registrador, um tipo de operando genérico é utilizado, para especificar que aquele nó casa com qualquer nó daquele tipo específico. O motivo desta diferença é que a árvore semântica usada para especificar regras não representa o comportamento de uma instrução, mas representa um padrão que casa com inúmeras árvores e oferece a possibilidade de substituir esta subárvore casada pelo padrão por outra subárvore equivalente.

A Figura 4.11 ilustra o conceito de regras de transformação. No caso, o item (a) define graficamente duas árvores equivalentes, ao passo que o item (b) faz isso textualmente, já usando a sintaxe apresentada na Figura 4.12. Diz-se que uma regra *casa* com uma árvore de entrada $\tau$ quando uma de suas duas árvores $T$ é encontrada como subárvore em $\tau$, respeitando regras especiais para comparação dos nós folhas, isto é, o nó folha da regra pode casar com qualquer nó daquele tipo - veja também a próxima subseção para a apresentação de um nó folha que pode ser substituído por qualquer subárvore. O item (c) apresenta um exemplo de árvore a qual a regra casa, bem como o resultado de sua aplicação. A aplicação ocorre quando a subárvore $T$ é substituída pela equivalente.

### Nó Especial Curinga

A Figura 4.12 apresenta um exemplo de definição de uma regra de transformação que adiciona conhecimento a respeito do operator *rotate left* (rótulo `rol`). Nesta caso, esta regra pode ser chamada de *regra de expansão*, uma vez que a árvore a esquerda contém apenas um operador e a árvore a direita contém vários operadores e, portanto, ensina o sistema a expandir o nó `rol` em operações mais simples. Este exemplo também ilustra o uso de um novo tipo de nó folha, exclusivo para definição de regras, o nó curinga (`any`). Para usá-lo, uma sintaxe semelhante a um nó registrador é usado, seguindo a Tabela 4.2, mas ao invés do nome da classe do registrador, utiliza-se a palavra chave `any`. Para uma regra, isto significa que qualquer subárvore semântica casa com um nó curinga. A única

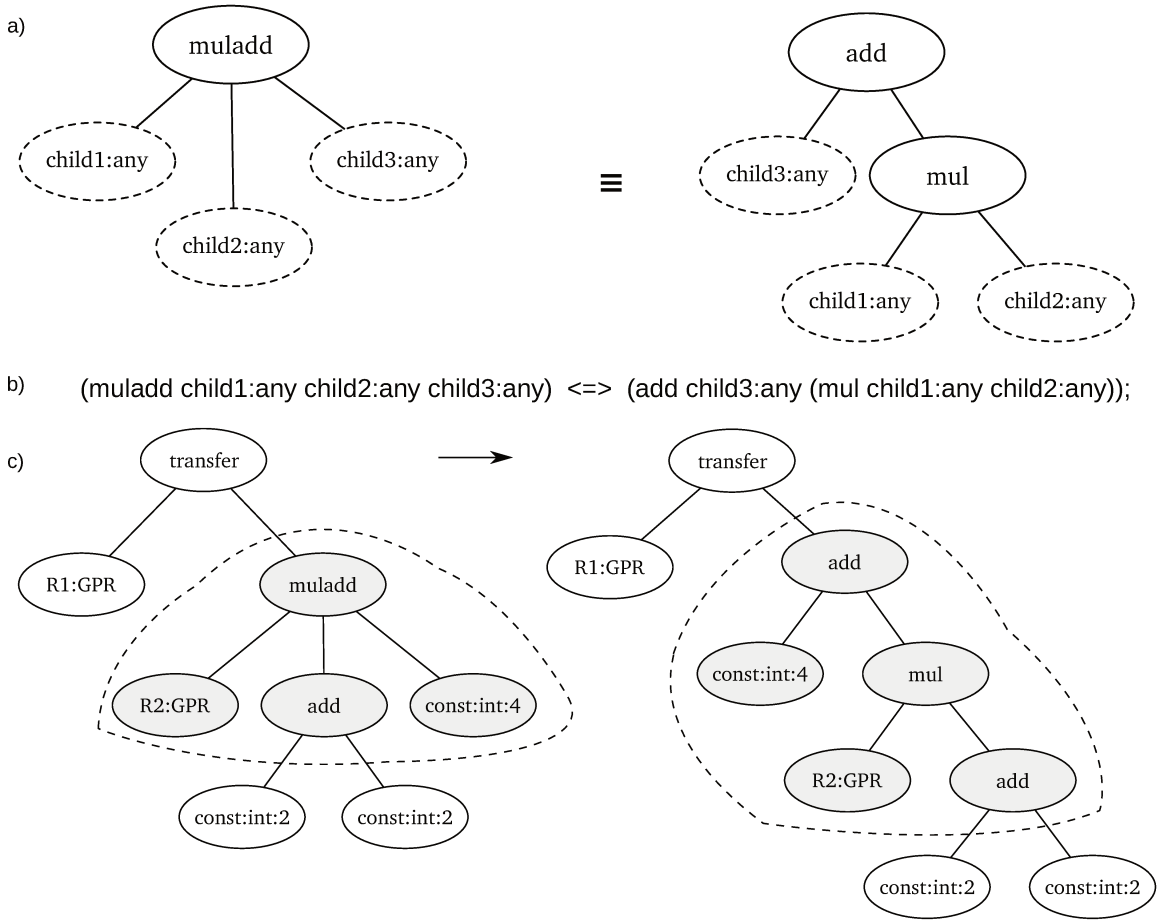



Figura 4.11: Ilustração gráfica da especificação de uma regra de equivalência (a), a sua representação textual (b) e um exemplo de casamento da regra e aplicação para transformar uma árvore semântica (c).

exceção está no filho esquerdo do operador `transfer` (o destino de uma movimentação de dados), que nunca é casado com um nó curinga por representar um caso especial. O destino da transferência determina um aspecto importante do operador de transferência (por exemplo, se a árvore representa um *load* ou *store*) e não pode ser separado do pai. Mais detalhes serão apresentados na próxima subseção.

Ainda, vale destacar que uma regra pode ser definida no modelo estendido ArchC ou em um arquivo de regras interno do sistema `acllvmbe`. As regras internas valem para todos os modelos e especificam características de operadores pré-definidos, como por exemplo, que um operador `add` é comutativo. As regras específicas de modelos podem ser usadas para introduzir novos operadores exclusivos para descrever uma determinada

```
1 (rol child1:any child2:any) <=>
2   (or (shl child1:any child2:any) (shr child1:any
3     (- const:byte:32 child2:any)));
```

Figura 4.12: Exemplo de regra de transformação que mostra que um nó `rol` (*rotate left*) é semanticamente equivalente a uma árvore que desloca o operando para esquerda e para direita e combina o resultado com um *ou* lógico.

máquina.

### O Operador Especial `dec`

O operador `dec` (de *decomposição* ou *sequenciador*) possui uma função especial na aplicação de regras de transformação. O operador `dec` possui a restrição de ser o nó raiz de uma árvore semântica ou filho de outro operador `dec`. A sua função é de, após a aplicação da regra, particionar uma árvore semântica em diferentes subárvores. Dessa forma, cada filho de um operador de decomposição, após a aplicação da regra, se tornará uma árvore semântica independente em termos estruturais. Contudo, a semântica é preservada pela noção de sequencialidade: são árvores separadas que devem ser avaliadas em sequência, começando pelo filho esquerdo, para se obter a mesma transformação de estado no processador que a árvore semântica original (não decomposta) expressa. Este elemento sequenciador é crucial no sistema `acllvmbe` para provar que determinada expressão deve ser implementada com mais de uma instrução de uma máquina alvo.

A Figura 4.13 apresenta a principal regra de decomposição utilizada no sistema `acllvmbe`. Esta regra especifica que qualquer nó (casado pelo nó curinga) pode ser transformado em um registrador qualquer, desde que uma outra árvore independente mova o resultado produzido pelo nó para este registrador. Neste momento é importante a restrição de que o nó curinga não case com o filho esquerdo do operador *transfer*, pois, caso contrário, uma árvore que expresse um *store* seria descaracterizada (pela decomposição).

```
1 AnyOperator:any =>
2   (dec (transfer AReg:regs^ AnyOperator:any) AReg:regs^);
```

Figura 4.13: A principal regra de decomposição utilizada no sistema `acllvmbe`.

Note o uso do caractere `^` no nó folha que especifica o registrador novo a ser usado para armazenar temporariamente o valor. Esta construção impõe a restrição de que o nó do tipo `regs` não deve casar com referências específica as registradores, mas apenas com referências genéricas (que especificam a liberdade de que qualquer registrador pode ser usado na árvore semântica).

O funcionamento interno do sistema `acllvmbe` e a maneira exata como as regras de transformação são empregadas para construir o *backend* de compilador são abordados na Seção 5.2.3. O Apêndice A apresenta todas as regras utilizadas no sistema. Um exemplo gráfico do uso de regras de decomposição aparece na Figura 4.14, que apresenta a expressão $a_1 = a_2 + a_3 + a_4$, sendo que $a_4$ é um número imediato e todos os outros são registradores. Suponha que queremos implementar esta expressão utilizando instruções de uma máquina arbitrária. Se esta máquina possuir uma instrução que realiza uma soma de 3 parcelas, sendo que a última é um imediato, basta comparar a expressão com a árvore semântica desta instrução. No caso mais comum em que a máquina possui uma instrução para somar dois registradores e outra para somar um registrador a um imediato, convém aplicar a regra de decomposição mostrada na Figura 4.13 e obter as duas árvores mostradas na Figura 4.14 como resultado. Deste modo, cada árvore corresponde a uma instrução de soma, e podemos mostrar que esta expressão pode ser implementada pelo encadeamento de uma instrução de soma de registrador com imediato, seguida por uma instrução de soma de dois registrados, utilizando a introdução de um registrador temporário chamado, neste exemplo, de *NewReg*[1].

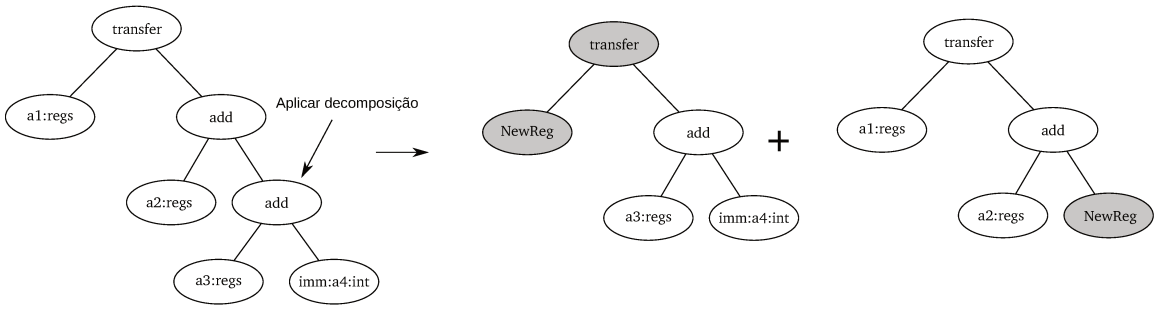



Figura 4.14: Ilustração gráfica de um exemplo de aplicação da regra de decomposição da Figura 4.13 utilizada para decompor uma árvore com 2 somas em 2 árvores com 1 soma cada, facilitando a inferência da necessidade de se implementar tal expressão com o encadeamento de 2 instruções diferentes. Os nós sombreados foram adicionados pela regra de decomposição.

[1]Na prática, este será um registrador virtual a ser atribuído para um registrador real pelo alocador de registradores.

# Capítulo 5

# `acllvmbe`: O Gerador Automático de *Backends* LLVM

Este capítulo é dedicado à discussão do projeto e implementação do sistema `acllvmbe`, que concretiza o ideal de geração de um *backend* LLVM de maneira não supervisionada e, deste modo, permite que o projetista de um modelo ArchC construa um compilador para a arquitetura do modelo. Com o *backend* pronto e utilizando as demais ferramentas ArchC, esta construção, por sua vez, é uma simples tarefa de unir componentes. Consulte a Figura 6.1 para observar o protótipo de compilador utilizado para validação de `acllvmbe`, como exemplo. Basta eleger um *frontend* capaz de converter código C para código de linguagem intermediária LLVM e então usar o *backend* para converter linguagem intermediária LLVM para código em linguagem de montagem da máquina alvo. Caso não haja um montador e ligador disponíveis para montar um executável, utiliza-se a ferramenta `acbingen`/`acasm`, capaz de gerar automaticamente montadores e ligadores. Esse fluxo de geração de ferramentas de criação de software e teste da plataforma está ilustrado na Figura 1.1.

Vale destacar que até o momento da escrita desta dissertação, estão disponíveis duas opções para *frontend*: `llvm-gcc`, que consiste no compilador GNU gcc redirecionado para gerar código LLVM e o `clang`, um novo projeto de *frontend* do time de desenvolvedores do LLVM que é independente do gcc.

## 5.1 Metas

No escopo deste projeto, utiliza-se a infraestrutura para construção de compiladores LLVM com foco na síntese de um compilador redirecionado para uma máquina alvo específica. Esta máquina alvo específica deve possuir uma descrição declarativa. Mais especificamente, uma descrição nos moldes da ADL ArchC que faz uso das extensões pro-

postas no capítulo anterior. A partir da descrição declarativa de diversas características da arquitetura e máquina alvo envolvidas, o objetivo principal deste projeto é automatizar o processo de redirecionamento do compilador LLVM.

> **Meta 1**: Redirecionar o compilador LLVM para uma máquina descrita com a ADL ArchC.

Para alcançar esse objetivo, é essencial a realização de algumas modificações na linguagem ArchC. Isto ocorre porque, no desenvolvimento inicial da linguagem, o objetivo de derivar *backends* de compiladores não foi contemplado nas decisões de projeto. Logo, informações tais como qual convenção é usada para as chamadas de funções e os quais tamanhos dos tipos básicos da linguagem não são fornecidos no momento da criação de um modelo ArchC. Consulte o Capítulo 4 para uma discussão mais aprofundada sobre as extensões da linguagem ArchC propostas e implementadas como parte deste trabalho.

> **Meta 2**: Estender a linguagem ArchC com construções que permitam a declaração das informações necessárias.

Um compilador completo, porém incapaz de gerar código eficiente, é pouco útil para avaliar um processador. O principal motivo para o projeto de um ASIP (*application specific instruction-set processor*) é implementar e avaliar o impacto de instruções específicas, que aceleram um determinado processamento importante para um nicho de aplicações. Se o compilador não é capaz de utilizar tais instruções, muito da praticidade almejada é perdida.

> **Meta 3**: O compilador sintetizado deve entender e utilizar instruções de propósito específico e não se restringir a um pequeno subconjunto de instruções gerais.

Pelos mesmos motivos do terceiro princípio, também é desejável um compilador competitivo, capaz de aplicar otimizações específicas de máquina e que seja capaz de extrair o maior desempenho da máquina modelada.

> **Meta 4**: O *backend* sintetizado deve ser capaz de aplicar otimizações específicas da máquina, aumentando a qualidade do código gerado.

## 5.2 Geração do Seletor de Código

O principal componente do *backend* do compilador certamente é o seletor de instruções, uma vez que abrange a fase em que grande parte da linguagem intermediária é traduzida

para linguagem de máquina. Neste trabalho, separa-se da seleção de instruções as tarefas de alocação de registradores e escalonamento de instruções. A alocação de registradores é abordada parcialmente pela biblioteca do gerador de código independente de máquina do LLVM e a parte dependente de máquina é trivial e gerada por `acllvmbe`.

### 5.2.1 Métodos para Criação do Seletor de Código

A bibliografia relacionada da Seção 2.2 indica diversos algoritmos disponíveis para seleção de instruções. Especificamente, este projeto utiliza o algoritmo *Maximal Munch* presente no LLVM (conforme identificado por Koes et al. [43]).

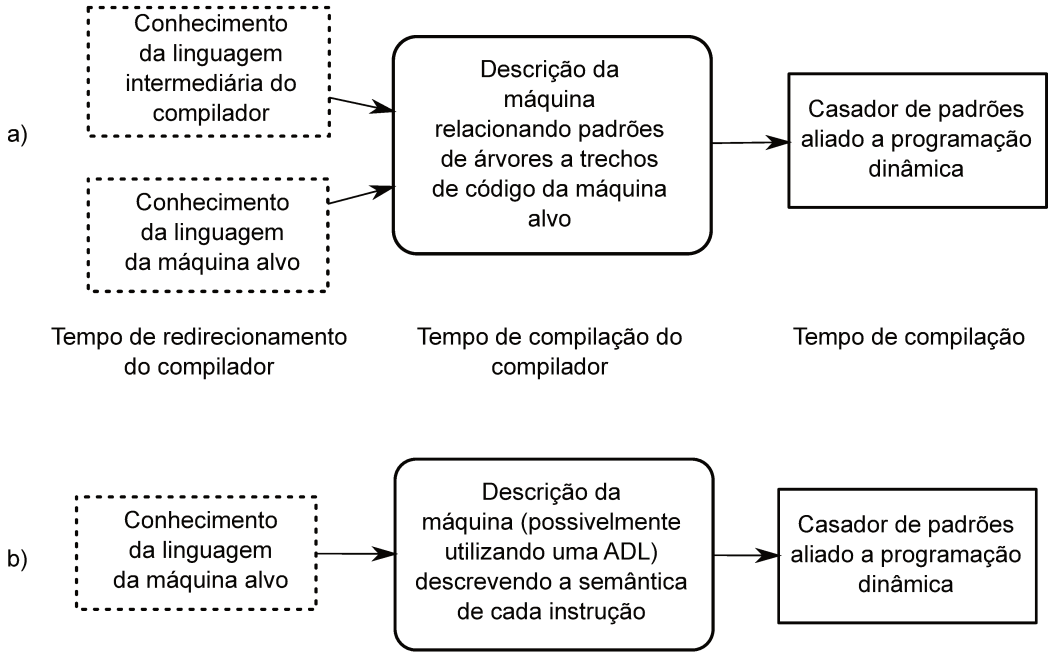



Figura 5.1: Comparação entre abordagem clássica para o redirecionamento de seletores de instrução (a) e abordagem proposta neste trabalho (b). Como há muito menos informações disponíveis neste último, algoritmos de busca são necessários para gerar automaticamente os padrões de árvores.

Para gerar os padrões de casamento com a linguagem de representação intermediária do LLVM, uma abordagem baseada na solução descrita na seção 2.3, de Cattel, foi adotada. Esta técnica se encaixa no fluxograma (b) da Figura 5.1. Dessa maneira, as informações necessárias no modelo ArchC para gerar o *backend* foram reduzidas consideravelmente, bastando uma maneira de expressar a semântica da instrução em RTL (*Register Transfer Lists* [58]). Esta forma de expressão é apresentada no Capítulo 4.

### 5.2.2 Descrição do Problema

Para formular o problema da geração automática de um seletor de instruções, considera-se apenas programas sem desvios, chamados *straigth line programs*. A seleção de instruções no LLVM particiona o programa em blocos básicos [5], um conjunto de *straigh line programs*, e resolve o problema da seleção de instruções em cada bloco básico individualmente.

Seja $P = \{E_0, ...E_{r-1}\}$ um programa sem desvios representado por uma sequência de árvores semântica (ver Seção 4.4.1) $E_i$, $0 \leq i < r$, onde $r$ é o tamanho do programa. Seja $M = \{I_0, ...I_{s-1}\}$ o conjunto de instruções $I_k$, $0 \leq k < s$ de uma máquina alvo arbitrária, onde $s$ é o número de instruções. Cada instrução $I_k$ corresponde a uma floresta semântica que representa a alteração no estado do processador que ocorre quando esta é executada. Considera-se que dois programas são equivalentes se a mudança de estados do processador provocada pela execução de um deles é idêntica àquela provocada pelo outro. Então:

> O problema da programação automática sem desvios consiste em, dado um programa sem desvios $P$ qualquer e um conjunto de instruções $M$, encontrar um programa $P' = \{E'_0, ...E'_{r'-1}\}$ equivalente tal que $\forall E'_i \in P'$, $E'_i \in M$.

Ao resolver uma instância do problema da programação automática sem desvios, consegue-se, para qualquer programa sem desvios, uma sequência de instruções de uma máquina alvo que implementa o programa, desde que as florestas semântica de todas as instruções da máquina alvo estejam disponíveis.

Vamos mostrar agora como o problema da geração automática do seletor de instruções para árvores de expressão se reduz ao problema da programação automática sem desvios. Para isso, note que uma árvore semântica é uma árvore de expressão e resolva o problema da seleção de instruções com qualquer algoritmo baseado em casamento de padrões em árvores de expressão como, por exemplo, o algoritmo de programação dinâmica proposto por Aho [3]. Para isso, é necessário especificar um conjunto de padrões da árvore de expressão que cobrem todos os casos que aparecem na linguagem intermediária que está sendo traduzida [5]. O conjunto de padrões pode ser o mesmo para qualquer máquina, desde que a linguagem intermediária seja a mesma e que este conjunto cubra todos os casos.

Porém, é necessário, para cada padrão, especificar como este é mapeado em um conjunto de instruções da máquina alvo. Se o padrão é uma árvore de expressão, este pode ser expresso como um programa sem desvios $P$. Juntamente com a máquina de conjunto de instruções $M$, configura-se uma instância do problema da programação automática sem desvios. A solução trará uma sequência de instruções da máquina alvo para implementar um padrão. Ao se resolver instâncias do problema para todos os padrões, obtém-se então, através do algoritmo de casamento de padrões escolhido, um seletor de instruções capaz de

traduzir uma linguagem intermediária cujos padrões estejam pré-determinados para uma sequência de instruções de uma máquina alvo cuja floresta semântica de cada instrução seja conhecida.

Desta maneira, para obter o seletor de instruções, um usuário precisa conhecer apenas a máquina alvo (o conjunto de instruções $M$) e não mais a linguagem intermediária do compilador alvo, como acontece na abordagem tradicional de criação de *backends* ao descrever padrões e seu mapeamento [3,5,22,27,28,66]. Esta situação corresponde ao fluxo (b) da Figura 5.1 e obedece aos princípios de projeto discutidos na Seção 4.1, evitando que a ADL ArchC dependa diretamente de informações específicas da linguagem intermediária LLVM.

Uma vantagem adicional de modelar o problema dessa forma é que a busca por uma implementação $P'$ pode ser voltada também para otimização segundo critérios de desempenho da máquina alvo, se a notação for estendida para incluir custos às instruções. De fato, diversos trabalhos [12,17,18,26,39,40,42,50] exploram algoritmos superotimizadores e *peephole optimizers* para programas sem desvios, sendo redirecionáveis com um modelo da máquina alvo. Assim, os princípios de projeto 3 e 4 relacionados a otimização discutidos na Seção 5.1 também podem ser atendidos, dependendo do algoritmo utilizado para resolver o problema da programação automática sem desvios.

### 5.2.3 Abordagem Utilizada

Dado o problema da Seção 5.2.2, se o objetivo é encontrar $P'$, a abordagem mais ingênua é utilizar um algoritmo de busca para encontrar, para um dado número $r'$ de instruções na solução, todas as combinações possíveis de instruções pertencentes a $M$. Isto exige um algoritmo cuja complexidade computacional é $\Omega(s^{r'})$, isto é, o limitante inferior do número de tentativas cresce exponencialmente com o tamanho da solução. Note que este limitante não contempla ainda dois fatores multiplicativos importantes: a grande variação de operandos que cada instrução pode ter e a complexidade do algoritmo que verifica se a solução candidata é uma solução final.

Para realizar esta análise, suponha que $X$ seja a variável aleatória que expressa o número de diferentes operandos a se testar para uma instrução. Então o número médio de soluções tentativas testadas será de $E[\sum_{i=1}^{s^{r'}} X] = s^{r'} E[X]$. Ainda, quanto ao segundo fator multiplicativo, deve-se atentar que a cada uma das tentativas, um *comparador semântico* é utilizado para verificar se a solução candidata obedece à restrição de ser equivalente ao programa $P$ original. Para isso, o programa pode, por exemplo, ser simulado e seu comportamento observado para verificar se é o mesmo do programa original. Como uma expressão, em geral, aceita como operando muitas possibilidades e a simulação precisaria testar cada combinação de operandos para verificar se a expressão é idêntica, utiliza-

se apenas algumas combinações como um método heurístico. O objetivo da heurística é realizar uma triagem para descartar candidatos e, em uma segunda etapa, verificar formalmente se as duas expressões são equivalentes. Para a verificação formal, pode ser usado um provador de teoremas para provar que $P \equiv P'$.

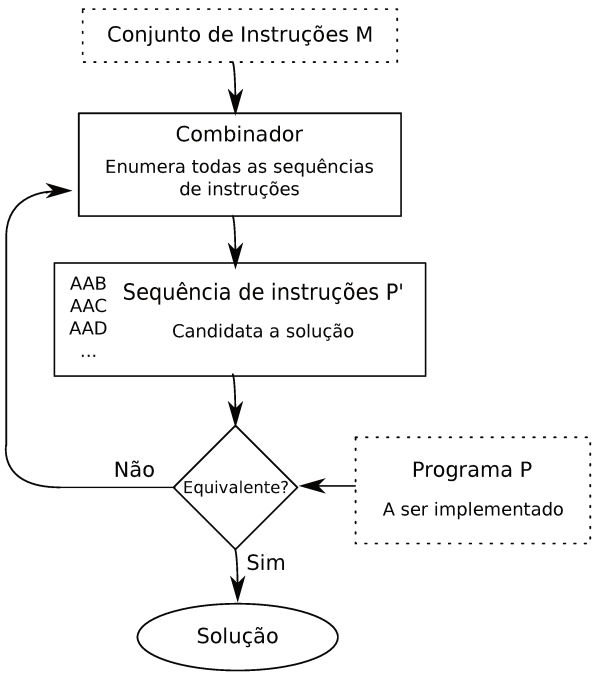



Figura 5.2: Ilustração da ideia geral da abordagem ingênua apresentada nesta seção. O comparador *Equivalente?* na verdade é um comparador semântico que verifica se a sequência de instruções implementa o programa P.

A maneira mais sensata de implementar esta abordagem é tentar sucessivamente sequencias maiores de instruções para $P'$, incrementando $r'$ a cada busca malsucedida. Repare que, se a métrica de desempenho utilizada for o número de instruções do programa, este algoritmo encontrará o programa ótimo que computa $P$. A abordagem descrita foi originalmente proposta por Massalin [50] que cunhou o termo superotimização (veja Seção 2.4.1), pois, por se tratar de uma busca exaustiva de todas as combinações de instruções possíveis para se computar uma expressão (limitada apenas pelo tamanho da sequência de instruções), prometia encontrar de fato um programa ótimo e produziu resultados experimentais surpreendentes ao encontrar trechos de código não triviais mesmo para um ser humano especialista na máquina alvo. Na abordagem de Massalin, contudo, a verificação final de que a solução de fato implementa um programa equivalente é deixada a um humano.

Mesmo que o método economize tempo ao descartar verificações formais (deixando esta tarefa ao usuário), como o método utilizado envolve busca exaustiva, o algoritmo

não escala para instâncias reais (máquinas com centenas de instruções e padrões que possivelmente necessitam de dezenas de instruções para sua implementação). No artigo original, publicado na década de 80, o algoritmo trabalhou com um subconjunto das instruções do processador 68020.

### Melhorando o projeto do algoritmo

Note que a ideia geral do método descrito envolve partir das instruções disponíveis no conjunto $M$ e tentar combiná-las para produzir um programa $P'$ tal que $P' \equiv P$. Isto está ilustrado na Figura 5.2. Outra abordagem consiste em partir da expressão $E = P$ e realizar uma série de transformações que preservam a equivalência $E \equiv P$ até que a condição de que $E$ deva ser uma sequência de instruções de $M$ seja satisfeita, concluindo que $P' = E$. A visão geral deste conceito está ilustrada, de maneira informal, na Figura 5.3. Uma invariante importante neste algoritmo é que as transformações não alteram a semântica final da sequência $P'$, de modo que todas as soluções candidatas, neste caso, já implementam o programa original $P$. Entretanto, dessa vez, há um laço que realiza comparações para se certificar de que a solução candidata pode ser implementada com as instruções $M$, ou seja, de que de fato, com as transformações, obteve-se uma sequência de instruções em $M$. Esta ideia foi utilizada no sistema `acllvmbe`, baseada no trabalho de Cattel [14] que também inspirou vários outros sistemas similares [19, 40, 65].

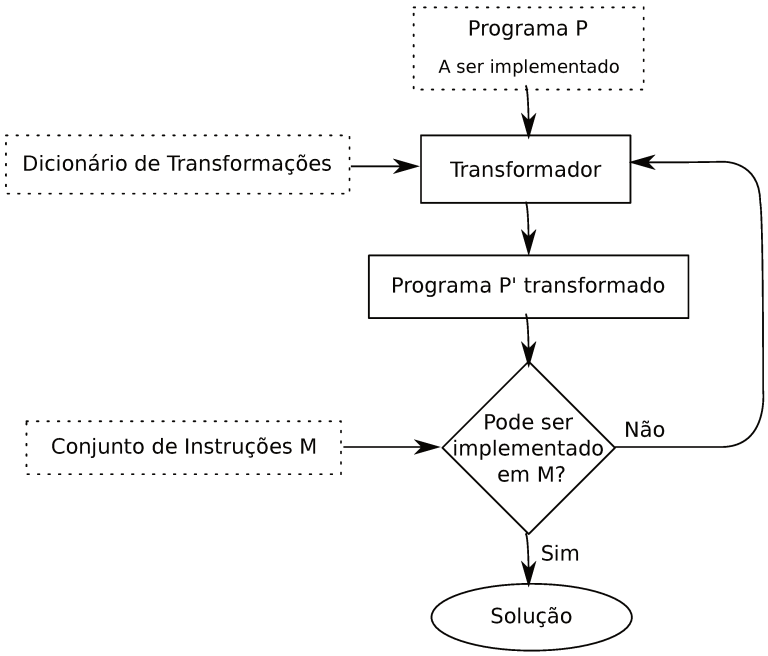



Figura 5.3: Diagrama da visão geral do algoritmo de busca baseado em transformações de equivalência.

No caso da abordagem apresentada na Figura 5.2, o espaço de busca é muito grande, uma vez que compreende todas as sequências de instruções de determinado tamanho. Já na abordagem utilizada no sistema `acllvmbe`, a vantagem é que o espaço de busca é consideravelmente reduzido. Primeiro, é necessário considerar que estamos partindo da expressão objetivo e transformando-a para encontrar instruções que a implementem. Ou seja, neste caso, a busca possui mais informações do que anteriormente (mais precisamente, a expressão inicial e as instruções são conhecidas e utilizadas para guiar a busca). Em segundo lugar, o espaço de estados é reduzido, uma vez que apenas expressões equivalentes ao programa original são pesquisadas e a busca é limitada pelo número de transformações conhecidas. Por outro lado, pode ser que exista um mapeamento do programa em instruções que nunca seja encontrado. Contudo, para fins de geração dos padrões do seletor de instruções, não é crucial evitar a perda de um mapeamento possível, se este poderá ser encontrado mais tarde com uma sequência mais cara. Este ponto é mais importante para um superotimizador, que precisa encontrar programas realmente não triviais, que podem não ser alcançados por um conjunto de transformações conhecidas.

## 5.3 Um Modelo Genérico de *Backend* e os Padrões LLVM

Os métodos de busca discutidos correspondem às técnicas e metodologia utilizada para construir um importante componente do sistema, responsável por encontrar uma determinada sequência de instruções e seus operandos, ou seja, uma ordem específica para se executar instruções da máquina alvo que garantidamente irá produzir o efeito semântico do objetivo informado no momento da busca. Este objetivo é informado na forma de uma árvore semântica. Por exemplo, se o objetivo corresponde a uma árvore cujo nó raiz é uma operação de soma e seus operandos são registradores, o resultado da busca irá provavelmente ser uma instrução da máquina alvo capaz de realizar a soma cujos operandos são dois registradores de propósito geral da arquitetura alvo. Diz-se que a árvore que representa o comportamento desta instrução de soma é *semanticamente equivalente* à árvore do objetivo.

De posse deste componente, um computador é capaz de tomar decisões sobre como implementar várias partes do *backend* do compilador, que exigem normalmente raciocínio analítico e conhecimento profundo sobre a arquitetura alvo por parte de um ser humano. A ideia geral da geração automática de *backend* implementada possui o seguinte ponto de partida: Há um modelo genérico de *backend*, do qual derivam-se todas as arquiteturas. Em toda situação onde é necessário emitir algum código, este modelo codifica o problema com uma floresta semântica. Como exemplo, suponha que o *backend*, escrito

em linguagem C++, implemente uma função membro de nome `EmitPrologue` da classe `RegisterInfo`. A especificação desta função membro diz que, conhecido o número de objetos da pilha de uma determinada função da máquina, o *backend* deve emitir código de máquina correspondente ao prólogo da função, cuja principal tarefa é alocar espaço suficiente na pilha para que todos os objetos sejam armazenados sem problemas.

Para resolver isto, o modelo genérico possui uma floresta semântica objetivo relacionada à função `EmitPrologue`, que possui nós expressando a natureza da operação. Se a arquitetura alvo possui pilha cheia decrescente, o modelo genérico irá utilizar a floresta semântica cuja operação expressa uma subtração no registrador responsável por armazenar o endereço atual do topo da pilha. Então, no momento da geração do *backend* para esta arquitetura alvo, o componente de busca será questionado sobre quais instruções deverão ser utilizadas para subtrair o registrador de pilha, e a resposta será armazenada em memória como uma lista de instruções.

Para prover completude ao exemplo ilustrado aqui, será descrito o processo completo, após o resultado de busca ser computado pelo algoritmo. O leitor não deve se preocupar em entender os pormenores, pois este tópico será revisitado na Seção 5.4.5. Esta lista de instruções armazenada em memória é um objeto da classe `SearchResult`. A ação de tradução é coordenada pela classe `TemplateManager`, responsável por especializar o modelo genérico para um modelo específico, que será o *backend* gerado para a máquina alvo. De posse do resultado da busca, `TemplateManager` delega à classe `PatternTranslator` a tarefa de traduzir esta lista de instruções, um objeto `SearchResult`, em código C++. Este código, quando colocado no contexto da função membro `EmitPrologue`, irá emitir exatamente a lista de instruções alvo informadas. Note que, para isso, o código utiliza a API (*Application Programming Interface*) do gerador de código LLVM.

Nesta etapa tardia de emissão de código de prólogo, o programa, no LLVM, é representado como uma lista encadeada de objetos `MachineInstr` e o código gerado irá utilizar a função membro `BuildMI`. O código da Figura 5.4 ilustra um exemplo real de um trecho de código gerado para o *backend* da arquitetura ARM, na função membro `emitPrologue`. Note que a classe `PatternTranslator` teve apenas o trabalho de organizar os operandos corretos, neste caso, utilizar o registrador décimo terceiro, o ponteiro de pilha da arquitetura ARM. Contudo, o conhecimento de que a instrução `Armv5e::sub1_1` deveria ser utilizada é fruto do algoritmo de busca. O nome `sub1_1` é formado pela composição do nome da instrução descrita no modelo ArchC para ARM, `sub1` e do número que corresponde à posição da sintaxe de linguagem de montagem a ser utilizada (neste exemplo, a primeira sintaxe, dando origem ao sufixo `_1`). O objeto `TII` corresponde a uma instância de `TargetInstrInfo`, uma classe do *backend* LLVM que encapsula informações sobre todas as instruções da arquitetura alvo. `MBB` é o bloco básico em construção, ao passo que as funções membro `addImm` e `addReg` apenas adicionam operandos à instrução recém-criada.

```
unsigned a1 = Armv5e::r13;
unsigned a3 = Armv5e::r13;
BuildMI(MBB, I, TII.get(Armv5e::sub1_1), a3).addImm(1).
  addImm(1).addReg(a1).addReg(a2);
```

Figura 5.4: Exemplo de código gerado para o *backend* da arquitetura ARM para compor a função membro `emitPrologue` do *backend* LLVM, apresentada na Tabela 3.3 do Capítulo 3.

Através do exemplo anterior, é possível se ter uma visão geral sobre como um modelo genérico é especializado através do algoritmo de busca e da classe `PatternTranslator`. Todavia, o seletor de instruções do gerador de código sintetizado deve ser capaz, ainda, de traduzir diretamente linguagem intermediária LLVM em linguagem de máquina alvo. Este problema é resolvido de maneira similar. O modelo genérico segue a convenção dos demais *backends* LLVM e utiliza um algoritmo guloso de casamento de padrões para selecionar as instruções. Entretanto, cada um dos *backends* LLVM programados por humanos possuem seus próprios padrões. Em nosso caso, o modelo genérico possui padrões pré-escolhidos e fixos, com a garantia de que esses padrões sempre são capazes de cobrir a árvore em linguagem intermediária LLVM.

No processo de leitura das informações específicas de compilador de uma máquina alvo descrita em ArchC, um arquivo de descrição do modelo genérico é utilizado como cabeçalho. Neste arquivo, há 57 padrões pré-escolhidos. Um usuário experiente no sistema pode adicionar padrões livremente, mas não é nosso objetivo que o usuário precise estudar a arquitetura interna do sistema. Por esse motivo, o conjunto mínimo provido garante que o seletor de instruções gerado será capaz de selecionar qualquer código. A maneira como esses padrões são descritos no arquivo cabeçalho consiste em relacionar trechos de árvore de linguagem intermediária LLVM com uma descrição, no formato de floresta semântica do sistema `acllvmbe` de seu significado semântico, expressando sua função. Desta maneira, para sintetizar o seletor de instruções do modelo especializado, a classe `TemplateManager` itera sobre todos os padrões descritos no cabeçalho e, para cada um deles, utiliza sua floresta semântica como objetivo do algoritmo de busca. A listagem completa dos padrões escolhidos, na ordem em que são casados, encontra-se no Apêndice A, bem como uma listagem das regras de transformação utilizadas no algoritmo de busca. O algoritmo de busca, então, fornece um objeto `SearchResult` com a lista de instruções da máquina que implementa aquele padrão LLVM. Esta lista é repassada à classe `PatternTranslator`, que é responsável por traduzi-la para código C++ que utiliza a API LLVM. A etapa de seleção de instruções ocorre antes da emissão de prólogo (ver Capítulo 3) e, nesta fase, o programa em compilação ainda é representado como um DAG (*Directed Acyclic Graph*),

diferentemente do caso da função membro `emitPrologue` descrita anteriormente. Neste caso, `PatternTranslator` traduz a lista de instruções resultado da busca em um DAG. A partir deste DAG, o código C++ para criá-lo é gerado e embutido no *backend* gerado.

Um exemplo deste código, que segue as convenções e API LLVM, aparece na Figura 5.5. Neste exemplo, o padrão que foi casado corresponde ao armazenamento em memória de um valor de 16 bits. Na linguagem intermediária LLVM, isto é codificado com uma operação `truncstorei16` com 2 operandos: o valor a ser armazenado e o endereço onde será armazenado. O código C++ gerado por `PatternTranslator` para o ARM simplesmente transforma o nó `truncstorei16` em um nó `Armv5e::strh_1`, com operandos próprios que codificam como esta instrução do ARM deve ser usada. Esta ação de *transformar* um nó, ao invés de remover e adicionar um novo, é uma estratégia comum na seleção de instruções LLVM para melhorar o desempenho. Note que, conforme descrito no Capítulo 3, a seleção de instruções no LLVM corresponde apenas a uma transformação dos nós do DAG. Ao final do processo, o DAG que antes possuía apenas nós com operações da linguagem intermediária LLVM, deve possuir apenas nós que correspondem diretamente a instruções da máquina alvo.

```
// Código que define o padrão casado, do modelo genérico:
// define pattern STORETRUNC16 as (
//   "(truncstorei16 i32:$src, i32:$addr)";
//   (transfer (memref addr:regs) (trunc src:regs const:num:16));
//);

// Código que emite este padrão, para a arquitetura ARM:
SDValue N0 = CurDAG->getTargetConstant(1ULL, MVT::i32);
SDValue N1 = N.getOperand(1);
SDValue N2 = N.getOperand(2);
SDValue OpsN[] = {N0, N1, N2, N.getOperand(0)};
return CurDAG->SelectNodeTo(N.getNode(), Armv5e::strh_1,
                            MVT::Other, OpsN, 4);
```

Figura 5.5: Exemplo de código gerado para o *backend* da arquitetura ARM para realizar a seleção da instrução `store` de meia palavra.

Portanto, pode-se concluir que as pré-condições necessárias para que `acllvmbe` complete a geração de um *backend* não se restringem apenas à descrição completa das informações da máquina alvo conforme apresentadas no Capítulo 4. A conclusão ou não do processo de geração depende diretamente do sucesso do algoritmo de busca em encontrar uma implementação para os 57 padrões LLVM pré-definidos (se estes não forem estendi-

dos pelo usuário). Contudo, caso o algoritmo de busca encontre uma lista de instruções de máquina que implemente cada um dos padrões, garante-se que o usuário obterá um *backend* funcional, uma vez que este conjunto de padrões foi escolhido para cobrir toda a árvore de representação intermediária LLVM.

## 5.4 Visão Geral da Arquitetura do Sistema

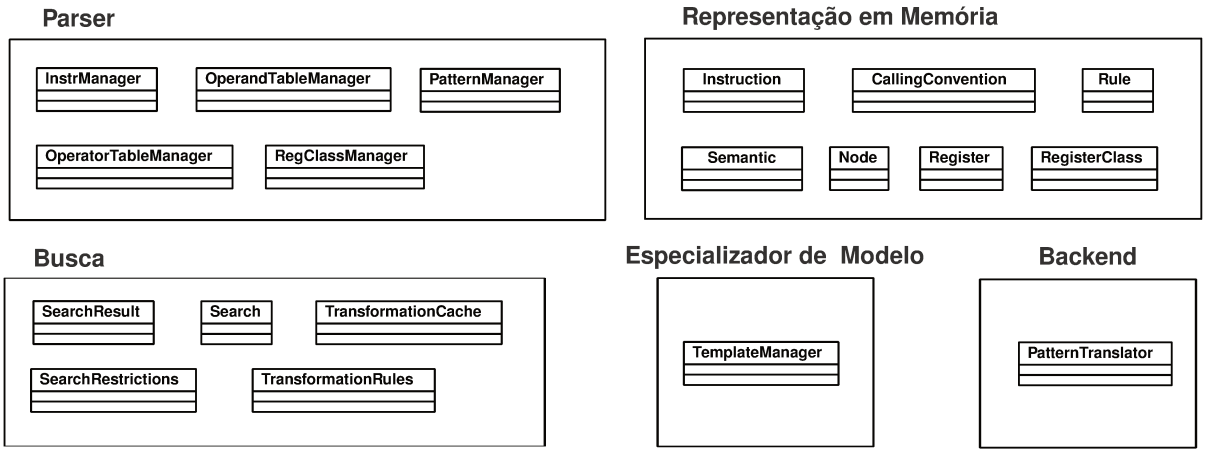



Figura 5.6: Diagrama UML simplificado com uma visão geral dos componentes que compõem o sistema `acllvmbe`.

O sistema `acllvmbe` foi construído na linguagem C++ e, portanto, projetado utilizando-se conceitos de orientação a objetos. Sua arquitetura de sistema lembra um compilador, uma vez que possui um *parser* para transformar as informações em arquivos texto para representação em memória, manipular estes dados e então utilizar um *backend* para transformar estes dados em representação intermediária no produto final, que é o *backend* LLVM gerado. O ponto onde difere de um compilador tradicional é no uso do algoritmo de busca para tomada de decisões sem supervisão do usuário. Neste ponto, as estratégias se assemelham àquelas utilizadas na área de inteligência artificial. Como ferramenta auxiliar, são utilizados o gerador de *parsers* `GNU Flex` [53] e `GNU Bison` [21] e como biblioteca de apoio, `libboost` [1] para manipular expressões regulares para permitir que o código existente da árvore LLVM seja facilmente manipulado para a inclusão de um novo *backend* através da aplicação de *patches*. Ao todo, o sistema `acllvmbe` foi implementado utilizando aproximadamente 15 mil linhas de código C++.

O projeto pode ser dividido em 5 componentes principais, de acordo com a Figura 5.6. O primeiro componente é o *Parser*. Este foi produzido com o auxílio de `GNU Flex` e

`GNU Bison` e é responsável por ler as informações de um modelo ArchC e o cabeçalho do modelo genérico discutido na Seção 5.3 e codificá-los em uma representação em memória. O segundo componente, *Busca*, contém todas as classes que implementam o algoritmo de busca e, como resultado, geram uma representação intermediária da solução em memória. O terceiro componente, *Representação em Memória*, consiste também em classes utilizadas para a representação intermediária em memória da solução oferecida pelo algoritmo de busca e também por diversas informações sobre o modelo ArchC colhidas pelo *parser*. Este componente é responsável por conectar todos os demais componentes do sistema, provendo um meio de comunicação entre eles (o formato de dados em memória). O quarto componente, *Especializador de Modelo*, abrange as classes de controle que sabem como utilizar as informações do *parser* e as funções do *backend acllvmbe* para especializar um modelo genérico e criar um *backend* LLVM. O modelo genérico está embutido no quarto componente. O último componente é o *Backend* `acllvmbe`. Para evitar confusão com o *backend* LLVM que é o produto da síntese, sempre que o último componente for referenciado neste texto, o nome *backend acllvmbe* será utilizado. Este componente traduz a representação intermediária da solução do algoritmo de busca em código C++ que realiza as ações necessárias para que um *backend* LLVM para a arquitetura alvo gere código.

### 5.4.1 O Parser

O parser é composto pelo léxico descrito utilizando `GNU Flex`, do *parser* de gramática descrito utilizando `GNU Bison` e da biblioteca preexistente do projeto ArchC `acpp`, responsável pelo pré-processamento de informações gerais de um modelo ArchC. Através desta biblioteca é possível extrair informações gerais contidas em um modelo ArchC e através dos componentes próprios de `acllvmbe` é possível extrair informações específicas para o gerador de compiladores. Contudo, a biblioteca `acpp` é programada em C. Ainda que ela já forneça uma representação intermediária das informações contidas no modelo, estas informações são traduzidas para a representação intermediária própria `acllvmbe` em classes apresentadas na Seção 5.4.3.

O arquivo com as informações do modelo ArchC específicas de `acllvmbe` conforme descritas no Capítulo 4 é separado dos arquivos que descrevem as outras informações e, por esse motivo, existem dois *parsers* distintos: o de `acllvmbe` e o do pré-processador ArchC. O arquivo contendo as informações específicas na árvore de arquivos do modelo é denominado `compiler_info.ac`. Os arquivos que compõem um modelo ArchC completo estão ilustrados na Tabela 5.1. Onde se lê *modelo*, deve-se supor o nome do modelo da arquitetura descrita, por exemplo, `mips`, `arm`, `powerpc` etc.

| **Nome do arquivo** | **Descrição** | **Utilizado por acllvmbe?** |
|---|---|---|
| ac_rtld.relmap | Informações sobre tradução de relocações para o ligador estático e dinâmico gerado por acbingen. | Não |
| modelo.ac | Arquivo principal do modelo ArchC, contém declaração de elementos estruturais armazenadores de estado. | Não |
| modelo_isa.ac | Arquivo com informações sobre o ISA(*Instruction Set Architecture*) do modelo. Contém a lista de todas as instruções da arquitetura, bem como sua codificação e sintaxe de linguagem de montagem. Este arquivo é utilizado por acllvmbe, lido pelo pré-processador acpp. | Sim |
| modelo_isa.cpp | Implementação em C++ do comportamento de cada instrução da arquitetura. | Não |
| modelo_syscall.cpp | Adaptador utilizado no auxílio da tradução de chamadas de sistema da máquina hóspede para a máquina hospedeira, em uma simulação. | Não |
| compiler_info.ac | Contém todas as informações de uso exclusivo do sistema acllvmbe, descritas no Capítulo 4. | Sim |
| dynamic_info.ac | Informações relacionadas ao ligador e carregador de tempo de execução e ligador dinâmico. | Não |
| dynamic_patch.ac | Informações de auxílio à construção do ligador dinâmico. | Não |

Tabela 5.1: Arquivos que compõem a árvore de um modelo ArchC.

No momento de realizar o *parsing* das informações específicas, ou seja, que dependem do *parser* de `acllvmbe`, o arquivo `compiler_info.ac` é precedido pelo cabeçalho do modelo genérico, e então estas informações são processadas. O cabeçalho contém todas as regras de transformação a serem utilizadas pelo algoritmo de busca, utilizando a sintaxe descrita na Seção 4.4.5. Também contém a descrição de todos os padrões LLVM, conforme explicado na Seção 5.3.

### 5.4.2 Componente de Busca

Este componente contém uma implementação do algoritmo de busca previamente discutido. Há 6 classes que compõem este componente: `Search`, `SearchResult`, `SearchRestrictions`, `TransformationCache` `Rule` e `TransformationRules`.

A classe `Search` contém não somente a implementação do algoritmo de busca, mas também define alguns parâmetros que podem alterar substancialmente as características da busca e da árvore de espaço de estados associada. O primeiro parâmetro importante é `MaxDepth`, a profundidade máxima que a árvore de busca pode atingir, ou seja, o máximo grau de recursão. Isto influencia diretamente no número de transformações aplicadas na árvore para se chegar ao objetivo.

Outro parâmetro é a chave `ExtensiveSearch`, que desliga uma heurística utilizada para reduzir bastante o espaço de busca ao impedir que seja aplicada mais do que uma transformação por nível da árvore de expressão. Por exemplo, considere uma instância do problema em que desejamos realizar transformações para provar que o nó `add` pode ser equivalente ao nó `sub`, desde que seus operandos sejam alterados. Contudo, suponha que existam apenas duas regras de transformação no dicionário: transformação de `add` em `X` e de `X` em `sub`. É fácil ver que existe uma relação de transitividade que implica `add` ser transformável em `sub`. Contudo, a heurística da restrição de uma transformação por nível iria impedir que esta solução fosse alcançada. No exemplo, o algoritmo iria aplicar a transformação `add` para `X` e então caminhar recursivamente tentando casar os filhos de `X`, evitando aplicar mais uma transformação no pai `X`, pois este já sofreu uma transformação. Para impedir que estes tipos de solução sejam ignorados, se a chave `ExtensiveSearch` estiver ligada, o algoritmo é capaz de encadear quantas transformações forem possíveis em um único nó, limitado apenas pelo parâmetro de máxima profundidade. Note que, sem o parâmetro de máxima profundidade, o algoritmo poderia facilmente entrar em um laço infinito caso exista um circuito no grafo que representa o encadeamento de transformações. O uso desta chave aumenta consideravelmente o tempo de busca.

Outro parâmetro importante da classe `Search` é o desligamento ou não do uso da *cache* de transformações, que é representada pela classe `TransformationCache`. É importante lembrar que o algoritmo de busca é orientado pelo objetivo, isto é, uma vez determinado

que vale a pena tentar implementar a árvore A com uma instrução cuja árvore semântica é B, inicia-se uma busca para encontrar o conjunto de transformações que prova que A é equivalente a B. A busca irá realizar transformações até que duas árvores sejam estruturalmente idênticas e, devido à natureza da busca realizada, caminhando recursivamente pelos filhos assim que o nó atual é transformado ou é idêntico ao objetivo, frequentemente a mesma pergunta é realizada diversas vezes: "Existe uma maneira de se transformar a subárvore A em B utilizando o dicionário de transformações disponíveis?". Caso uma busca sem sucesso já tenha concluído que o subproblema não possui solução, esta informação é anotada na *cache*. Como exemplo, considere o problema de determinar se um nó folha do tipo `imediato` pode ser transformado em um nó folha do tipo `registrador`. Neste problema especificamente, a resposta é positiva caso haja na arquitetura uma instrução para mover um imediato para um registrador e, no contexto da busca, isto será concluído ao se aplicar uma regra de decomposição. Entretanto, caso não exista esta opção nesta máquina específica, a busca irá tentar todas as transformações possíveis até concluir que não há como mostrar que `imediato` pode ser equivalente a `registrador`. Ainda, esta pergunta poderá ser feita inúmeras vezes durante a busca de padrões. Com o uso da *cache* que armazena as transformações que não são possíveis, esta conclusão é computada apenas uma vez e a árvore do espaço de busca é podada, reduzindo bastante o tempo de execução. Uma discussão detalhada sobre a importância da *cache* de transformações no desempenho do algoritmo é apresentada no Capítulo 6.

O último, mas não menos importante, parâmetro da classe `Search` aborda a sua execução em arquiteturas modernas CMP (*Chip Multi-Processor*). Fazendo uso da biblioteca OpenMP [16] para sua paralelização, é possível realizar a busca pela implementação de mais de um padrão LLVM ao mesmo tempo. Note que o laço principal da geração de *backend* consiste em uma iteração sobre todos os padrões LLVM a se implementar, conforme exposto na Seção 5.3. Entretanto, para que isso seja possível, é necessário que não haja variáveis compartilhadas entre duas ou mais *threads* de execução. Neste caso, entre duas ou mais instâncias de execução do algoritmo de busca.

Frequentemente, os parâmetros de uma instância do problema da busca não se limitam apenas à árvore semântica objetivo, que é representada por um objeto do tipo `Tree` (ver Seção 5.4.3). Podem existir pedidos de restrições na busca com relação aos nós folha da árvore semântica, como por exemplo, de que um determinado nó folha do tipo registrador seja necessariamente da classe de registradores de propósito geral. Esse tipo de restrição surge no processo de busca. Ou seja, as restrições aparecem como parâmetros de um subproblema que está sendo resolvido recursivamente e nunca como parâmetro inicial da busca. Isto ocorre na aplicação de regras de decomposição, quando efetivamente se particiona a árvore em duas, e uma delas vira instância de um subproblema a ser resolvido recursivamente. Contudo, muitas vezes, esta partição da árvore em duas não as

separa completamente, de modo que uma árvore ainda possui dependências implícitas em relação à outra. Se uma árvore foi casada com uma instrução que utiliza registradores `X` para implementar um nó folha do tipo registrador, esta informação precisa ser passada para a outra árvore, de que um de seus nós folha, de mesmo nome daquela casada em outra instância da busca, precisa ser o registrador `X` também. Esta comunicação que atrela dois subproblemas para resolver um problema maior é realizada pela classe `SearchRestrictions`.

Por fim, o dicionário de transformações e suas regras são representados pelas classes `TransformationRules` e `Rule`, respectivamente. Os algoritmos que implementam as funções membro dessas classes realizam a especialização de uma regra geral para uma instância de árvore semântica específica. Suponha, a título de exemplificação, que exista uma regra que transforme um nó `add` em um nó `sub`, preservando os mesmos operandos na mesma ordem. Neste caso, a função membro `ForwardMatch()` irá informar se esta regra pode ser aplicada em uma dada árvore semântica utilizada como parâmetro, isto é, se o seu nó raiz é um operador do tipo `add`. Caso positivo, a função membro `Apply()` irá extrair as duas subárvores filhas, excluir o nó `add`, criar um nó `sub` e configurar seus filhos com os filhos anteriores de `add`. A função para aplicação de regras que decompõem a árvore trata este caso especial e funciona de maneira análoga, com a diferença de que, como resultado, produz uma floresta semântica, ao invés de apenas uma árvore semântica.

### 5.4.3 Representação em Memória

A representação em memória dos dados colhidos pelo *parser* e também processados ao longo do fluxo de execução `acllvmbe` precisa contemplar, principalmente, a floresta semântica descrita no Capítulo 4. Com relação à floresta semântica, deve-se notar que a representação não deve seguir exatamente os mesmos princípios de projeto de uma linguagem intermediária de compilador, que precisa abranger muitos casos e construções particulares, sob pena de perder expressividade para armazenar uma operação que alguma das máquinas para a qual o compilador possui um *backend* realiza nativamente (encarecendo o custo da otimização). Uma árvore semântica deve, sobretudo, privilegiar o funcionamento do algoritmo de busca, pois será primariamente utilizada para este fim. Deve, entretanto, ser extensível e inicialmente abranger pelo menos a diversidade encontrada nos diferentes tipos de arquitetura possíveis de serem expressados com ArchC.

A formalização da máquina sobre a qual a expressão de semântica opera é mais abrangente do que aquela utilizada por ArchC. A adoção de uma formalização menos restritiva simplifica o projeto. De forma similar a Cattel [14], é utilizada no projeto de `acllvmbe` uma floresta de expressões que representam a semântica de uma instrução. Tais árvores expressam manipulação de dados de forma concomitante, como ocorre com a descrição

comportamental natural em linguagens HDL (*Hardware Description Languages*). Isto é, não existe ordem nem sequenciamento da execução de duas árvores de uma floresta semântica.

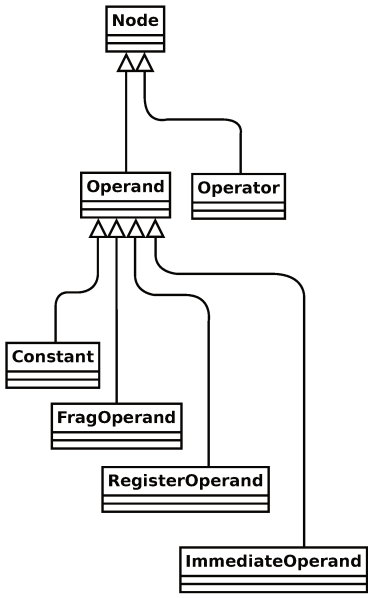



Figura 5.7: Visão da hierarquia das classes que representam uma árvore semântica. A classe pai `Node` possui também o sinônimo `Tree`.

O diagrama de classes mostrado na Figura 5.7 apresenta as classes criadas para expressar uma floresta semântica. Note que estas classes estão diretamente relacionadas com a linearização de uma árvore em notação textual apresentada no Capítulo 4.

As ações semânticas do *parser* descrito na Seção 5.4.1 dão origem a uma árvore semântica em memória. Porém, muitas outras informações obtidas pelo *parser* precisam ser representadas em memória, além da floresta semântica. Esta é a tarefa das classes `Register`, `RegisterClass`, `CallingConvention`, `RegClassManager`, `FragmentManager`, `PatternElement` e `PatternManager`, que não serão discutidas em detalhes nesta dissertação por possuírem implementação trivial para encapsular os respectivos objetos e conceitos já apresentados.

Ainda, as classes `Instruction` e `InstrManager` encapsulam as informações, em C++, dos conceitos já modelados pelo pré-processador ArchC, mas em linguagem C. De maneira geral, todas as classes com sufixo `Manager` representam um banco, conjunto ou dicionário que armazena um determinado elemento. A classe `InstrManager` armazena o conjunto de todas as instruções da máquina alvo e, para cada instrução, a classe `Instruction` contém a

floresta semântica que descreve o comportamento desta instrução. É importante lembrar, conforme exposto no Capítulo 4, que `acllvmbe` não trabalha com a mesma granularidade de ArchC para instruções. Isto é, uma instrução descrita com ArchC pode dar origem a várias instruções diferentes em `acllvmbe`, pois cada sintaxe de linguagem de montagem diferente para uma mesma instrução exige uma floresta semântica diferente (e um objeto `Instruction` diferente), uma vez que a floresta semântica tem a característica de que seus nós folha são diretamente relacionados aos operandos da sintaxe em linguagem de montagem.

Como o tempo de execução do algoritmo de busca pode ser grande e para evitar que cada execução de `acllvmbe` incorra em refazer todo o processo de busca para cada padrão LLVM, uma camada de persistência foi criada com a classe `SaveAgent`. Para cada implementação de padrão descoberta pelo algoritmo de busca, o objeto `SearchResult` é serializado e armazenado na forma de texto, em disco. Caso o usuário deseje interromper a busca pela metade e retomá-la mais tarde, os padrões já descobertos não serão revisitados, mas extraídos diretamente do disco. Para esses padrões, a busca é acionada novamente apenas no caso de uma mudança nos arquivos `compiler_info.ac` ou cabeçalho do modelo genérico ser detectada.

### 5.4.4 Especializador de Modelo

O componente especializador de modelo articula a execução de todos os outros de maneira a gerar um *backend* LLVM para a arquitetura alvo. Ele é representado por uma única classe destinada a esse fim, chamada `TemplateManager`. O nome *gerenciador de modelos* foi utilizado porque esta classe contém código para especializar os diversos modelos de arquivos do *backend* LLVM. Mais especificamente, cada classe mencionada no Capítulo 3 possui um arquivo relacionado e foi construído um modelo genérico com o esqueleto das funções e código genérico utilizado em comum a todos os *backends* gerados.

A tarefa de `TemplateManager`, então, é a de receber as informações coletadas pelo *parser*, executar o componente de busca para cada padrão LLVM informado no arquivo cabeçalho do modelo genérico (de forma paralelizada ou não, conforme descrito na Seção 5.4.2), executar a busca também para inferir como realizar algumas tarefas específicas do modelo, como a função membro `emitPrologue()` ilustrada como exemplo na Seção 5.3, traduzir todos esses resultados de busca em código C++ utilizando o *backend* *acllvmbe* da Seção 5.4.5 e inserir este código para preencher o esqueleto das funções do *backend* LLVM e, dessa forma, gerar uma árvore de arquivos completa que corresponde a um *backend* funcional LLVM, desde que esta seja inserida no processo de compilação do projeto LLVM versão 2.5. Para isso, `TemplateManager` ainda utiliza um conjunto de expressões regulares para analisar uma árvore de código-fonte LLVM 2.5 e alterar os

arquivos necessários (*build scripts*) para informar a existência de um novo *backend*. A partir de então, basta que o usuário compile o projeto LLVM para derivar um compilador funcional para uma arquitetura alvo descrita com ArchC.

A árvore de arquivos código-fonte do esqueleto do modelo genérico inclui todas as funções membro descritas no Capítulo 3. Para cada região onde o arquivo deve ser especializado por código C++ gerado por `TemplateManager`, o arquivo possui um marcador especial, como `__marcador__`. Dessa maneira, basta que `TemplateManager` monte um *script* do processador de macros GNU m4 [60] e realize uma chamada de linha de comando para que o processador de macros substitua os marcadores pelo código C++ gerado.

### 5.4.5 Backend `acllvmbe`

O componente *backend acllvmbe* é composto pelas classes `PatternTranslator`, `LLVMDAGInfo` e `AsmProfileGen`, dentre as quais a primeira é a mais importante delas. Sua função é a de especializar os resultados de busca, encapsulados em objetos `SearchResult`, para aplicação na geração de *backends* LLVM. O nome deste componente é *backend acllvmbe* pois, em nenhum outro componente do sistema `acllvmbe`, há a menção de que o objetivo é a geração específica de *backends* LLVM. Portanto, o *backend acllvmbe* atual concentra as informações específicas do projeto LLVM e pode ser substituído por outro, que irá interpretar os resultados de busca de maneira diferente, para que *backends* para outros compiladores sejam produzidos.

Porém, a infraestrutura `acllvmbe` construída não se limita apenas à geração de *backends* de compiladores. Para ilustrar isso, a classe `AsmProfileGen` foi construída e é acionada quando o objetivo é gerar automaticamente arquivos em linguagem de montagem da arquitetura alvo para testar as instruções e validar uma plataforma instrução por instrução. Neste contexto, `AsmProfileGen` gera, para cada instrução da máquina alvo a ser testada, um programa em linguagem de montagem que consiste em um laço de repetição em um número pré-determinado de vezes. No corpo do laço encontra-se a instrução a ser testada, utilizada diversas vezes seguidamente, alternando apenas os operandos possíveis. O corpo do laço pode ser gerado de maneira trivial, instanciando a instrução diversas vezes com operandos aleatórios. Contudo, para gerar o laço, é necessário utilizar o algoritmo de busca do componente da Seção 5.4.2. Ocorre, então, uma busca por uma árvore semântica que expressa o incremento de uma variável de controle e o salto condicional. Caso o algoritmo de busca encontre, para a arquitetura alvo, uma maneira de implementar o laço, a lista de instruções é fornecida como um objeto `SearchResult`, e a classe `AsmProfileGen` é responsável por traduzir esta lista e imprimi-la no arquivo que contém o código em linguagem de montagem para realizar o teste da máquina alvo. Este algoritmo foi utilizado com sucesso em um projeto do Laboratório de

Sistemas de Computação (LSC) do Instituto de Computação da UNICAMP para gerar testes automaticamente com o objetivo de estimar a energia dissipada com a execução de cada instrução. A plataforma completa conta, também, com um mecanismo para estimar a potência através da simulação dos programas gerados por `acllvmbe`.

Existem três tipos de algoritmos das funções membro de `PatternTranslator`. Os primeiros concernem a geração de código C++ para gerar[1] código de máquina na fase de seleção de instruções. Nesta fase, o código de máquina na API LLVM é representado como um DAG. Por esse motivo, o algoritmo precisa converter a lista encadeada de instruções e seus respectivos operandos, disponível em um objeto `SearchResult`, em um DAG. Em memória, o DAG é representado por objetos da classe `SDNode` (`SelectionDAG` *nodes*). Este é o mesmo nome utilizado na API LLVM, mas não deve ser confundindo com o mesmo, uma vez que a versão utilizada no sistema `acllvmbe` é uma simplificação daquela disponível no LLVM, apenas para fins de tradução intermediária.

Para gerar o DAG, um percurso na lista encadeada de instruções é realizado e seus operandos são inspecionados. As arestas do grafo codificam uma relação de uso. Por exemplo, considere a lista de instruções a seguir, de uma arquitetura fictícia. Suponha que se trata de uma arquitetura que utiliza sempre três endereços como operandos das instruções (quádruplas), em que o primeiro é o destino. A Figura 5.8 ilustra um processo de tradução que passa a lista encadeada de instruções para um DAG.

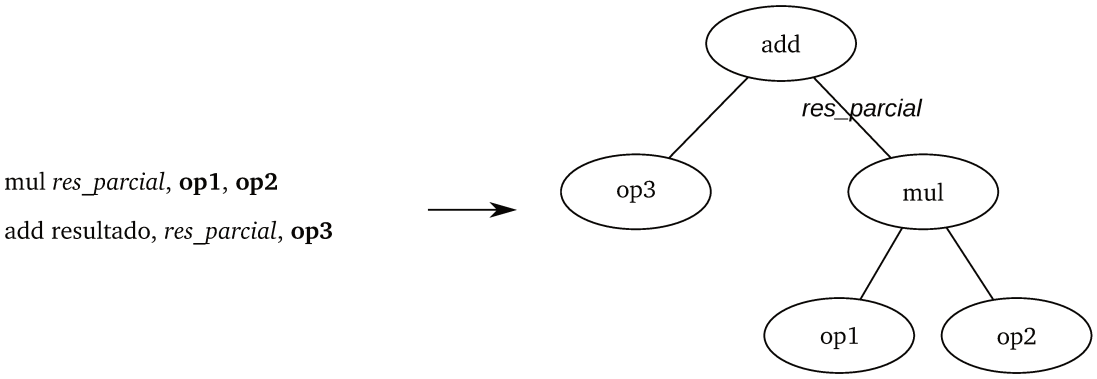



Figura 5.8: Exemplo da tradução de quádruplas para DAGs, etapa intermediária realizada pela classe `PatternTranslator`.

Desse modo, note que a instrução `add` usa o resultado da instrução `mul` através do registrador intermediário `res_parcial`. Ao percorrer a primeira instrução, o algoritmo de tradução de representação intermediária em quádruplas para DAGs [5] armazena a

[1]Note que trata-se de um gerador de gerador de código, ou *code generator generator* (CGG) na literatura.

definição de `res_parcial` como sendo do nó DAG criado por esta mesma instrução. No momento de analisar os operandos da segunda instrução, `add`, para cada um deles, uma consulta a um dicionário de resultados intermediários é realizada. Caso exista, como no caso de `res_parcial` do exemplo, o nó que define este resultado é usado como filho do nó em análise, criando uma aresta que codifica a relação de **uso** do resultado.

Após essa etapa, `PatternTranslator` realiza um percurso em profundidade no DAG gerado e, para cada nó encontrado, emite o código C++ correspondente que utiliza a API LLVM para instanciar este nó no *backend* LLVM, conforme exposto na Seção 5.3 e ilustrado na Figura 5.5.

O segundo tipo de algoritmo é responsável por gerar a emissão de código em fase tardia, após o escalonamento de instruções. Nesta fase, o algoritmo genérico escalonador de instruções do LLVM já transformou o DAG de seleção de instruções em uma lista encadeada de objetos do tipo `MachineInstr`. Ao realizar esta transformação de DAG para lista encadeada, uma ordem precisa ser atribuída, uma vez que no DAG, não está explícito qual operando de um nó deve ser avaliado primeiro. Os códigos relacionados a emissão de prólogo de função, epílogo de função e endereçamento de objeto na pilha são todos realizados em fase tardia. Neste caso, o algoritmo utilizado em `PatternTranslator` para traduzir um objeto `SearchResult` em código C++ é ligeiramente mais simples. O motivo é que o objeto `SearchResult` já apresenta o resultado em formato de lista encadeada de instruções e não é necessário transformar o formato. Basta percorrer a lista e gerar código C++ que utiliza a API LLVM para emitir diretamente as instruções necessárias para realizar uma dada tarefa.

A última classe de algoritmos utilizada em `PatternTranslator` realiza uma tarefa menos trivial. A partir de um padrão descrito com um objeto `SearchResult`, a tarefa consiste em gerar código C++ que utiliza a API LLVM para identificar este padrão em um bloco básico de máquina no *backend* LLVM em fase tardia. Uma vez identificadas as instruções que compõem este padrão, oferece a opção para remover estas instruções. A necessidade deste código está em trocas de decisão em fase tardia do *backend* LLVM. Por exemplo, caso seja necessário utilizar um operando de 32 bits ao invés de um operando menor, cujo código já está emitido no bloco básico, a única forma é a de procurar o padrão que implementa o operando menor, removê-lo e adicionar um novo padrão que carrega o operando de 32 bits.

# Capítulo 6

# Resultados Experimentais

Os experimentos realizados no âmbito do projeto `acllvmbe` podem ser divididos em três grupos. O primeiro conjunto possui a função de validar o sistema, isto é, concluir se o *backend* gerado leva a construção de um compilador capaz de gerar código correto para uma dada arquitetura alvo. Neste quesito, foram testadas 4 arquiteturas: ARM [29], SPARC [62], MIPS [41] e PowerPC [51]. O segundo conjunto de testes destina-se a avaliar o desempenho do algoritmo de busca em diferentes condições e diferentes implementações. O último conjunto de testes compara a qualidade do código gerado pelos *backends* gerados por `acllvmbe` com os gerados por compiladores consagrados no mercado para as arquiteturas correspondentes.

## 6.1 Validação e Análise da Qualidade do Código Gerado pelos Backends

### 6.1.1 ARM

O fluxo adotado para verificação da corretude do compilador sintetizado para a arquitetura ARM foi: a) compilar programas do Mibench [33], um *benchmark* para sistemas embarcados; b) executá-los usando o simulador ARM criado por ArchC e c) comparar a saída com um modelo de referência.

A Figura 6.1 ilustra o fluxo de validação utilizado para a arquitetura ARM. O objetivo final é o de produzir um compilador da linguagem C para a arquitetura ARM e atestar o seu funcionamento. O primeiro passo é utilizar o *frontend* `llvm-gcc`, que irá traduzir o programa escrito em linguagem C para *bitcode* LLVM. Esta é a forma de armazenamento em disco para um programa convertido para linguagem intermediária da infraestrutura LLVM. Em seguida, `llc`, o programa responsável por traduzir linguagem intermediária da

infraestrutura LLVM em código de linguagem de montagem para uma máquina alvo específica é utilizado. Este é o componente gerado automaticamente pelo projeto `acllvmbe`. O objetivo do fluxograma desenvolvido é testar este componente. Todavia, o programa em linguagem de montagem produzido pelo *backend* ainda não é um programa completo. É necessário utilizar um montador e um ligador (do pacote binutils [55]) e possuir uma biblioteca C pronta para a arquitetura alvo.

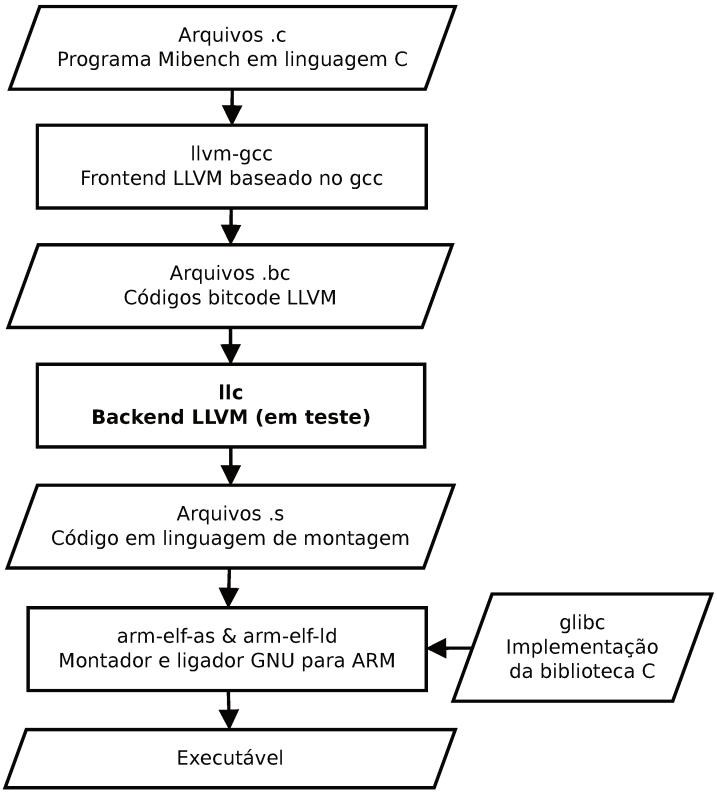



Figura 6.1: Diagrama do fluxo de validação utilizado para a arquitetura ARM.

Os testes para a arquitetura ARM são mais completos (utilizando Mibench) e fazem uso da biblioteca C. Para isso, é utilizado o pacote *cross-compiler* `gcc` [64] versão 3.4.5 para prover o montador e ligador para receber os arquivos em linguagem de montagem produzidos por `llc` (em teste) e ligar à biblioteca C `glibc` versão 2.3.6. O resultado, um programa executável ARM, é executado por um simulador criado com ArchC versão 2.1 a partir do modelo ARM versão 0.7. A saída produzida pelo simulador corresponde ao que o programa ARM imprime na saída padrão (em sistemas UNIX, escritas no arquivo de índice 1) e foi utilizada como meio para determinar se o *backend* `llc` gerado traduziu corretamente o programa. As entradas para o programa (obtidas por meio de leitura

do arquivo de índice 0, entrada padrão) correspondem aos casos de teste propostos pelo *benchmark* Mibench. Cada programa possui dois casos de teste: pequeno e grande. Alguns programas precisam ser recompilados para alternar entre os dois casos. Para outros, basta fornecer uma entrada diferente, especificada pelo *benchmark*. A Tabela 6.1 sumariza as versões dos componentes utilizados. O sistema `acllvmbe` gera *backends* para a versão 2.5 da infraestrutura LLVM e, por esse motivo, esta foi a versão utilizada para `llc`. O *frontend* `llvm-gcc` 4.2.1 possui a versão compatível com a infraestrutura LLVM 2.5.

| **Nome do componente** | **Versão** |
|---|---|
| `llvm-gcc` | gcc 4.2.1 (Baseado em Apple Inc. build 5636) |
| `llc` | LLVM 2.5 |
| `arm-elf-gcc` | gcc 3.4.5 com glibc 2.3.6 |
| `acsim` | ArchC 2.1 com modelo ARM 0.7 |

Tabela 6.1: Versão dos componentes utilizados para a validação da arquitetura ARM.

### Programas Testados

A Tabela 6.2 apresenta os programas que foram compilados utilizando o *backend* gerado automaticamente. Para cada programa, a tabela apresenta o número de instruções executadas pelo simulador ao executar com carga pequena e grande. Para fins de comparação com um compilador amadurecido, o número de instruções para execução dos mesmos programas compilados com gcc versão 3.4.5 é apresentado também. Ambos utilizam a mesma biblioteca dinâmica compartilhada glibc versão 2.3.6. É importante atentar ao fato de que o código compilado pela infraestrutura de compilador gerada corresponde apenas ao código da aplicação, excluindo-se o código da biblioteca C, que foi compilado com o compilador gcc versão 3.4.5 para ARM. Portanto, em um traço de execução que executa chamadas a funções da biblioteca padrão C, certamente haverá código idêntico aos dois casos (programa compilado com infraestrutura gerada LLVM e programa compilado com gcc), pois este código está presente na biblioteca C.

### Qualidade do Código Gerado

A Tabela 6.3 também inclui o resultado produzido para o caso em que o programa foi compilado utilizando o *backend* ARM LLVM 2.5 codificado manualmente, sem o auxílio da automação do projeto `acllvmbe`. A comparação com o compilador gcc para ARM não reflete diretamente a qualidade do *backend* gerado automaticamente, uma vez que o

| Programa | Descrição |
|---|---|
| Quick Sort | Ordenação utilizando a função `qsort` da biblioteca padrão C. |
| Bit Count | Teste de algoritmos de contagem de bits. |
| CRC32 | Algoritmo CRC para cálculo de resumo. |
| ADPCM | Modulação de sinal utilizado na área de telecomunicações. |
| Dijkstra | Algoritmo clássico de busca do menor caminho em um grafo com pesos positivos nas arestas. |
| Patricia | Implementação da estrutura de dados Patricia trie para armazenar informações que são identificadas por um conjunto esparso de valores de chave. |
| Rijndael | Algoritmo *Advanced Encryption Standard* (AES) de criptografia. |
| SHA | Função segura de *hash* utilizada em criptografia. |

Tabela 6.2: Programas Mibench testados.

problema pode estar nas etapas anteriores ao *backend*, independente de máquina. Mas oferece uma estimativa útil da qualidade de um compilador consagrado. Por outro lado, a comparação com o *backend* ARM preexistente no projeto LLVM indica diretamente a diferença de qualidade de código produzida pelo *backend* gerado automaticamente e um *backend* codificado manualmente. Para destacar este aspecto da comparação, o compilador construído com o *backend* ARM manual utilizou exatamente a mesma infraestrutura da Figura 6.1, de modo que apenas o *backend* `llc` foi substituído. Dessa maneira, qualquer perda de desempenho sofrida pelo código produzido pelo *backend* automático em relação ao código produzido pelo *backend* manual pode ser atribuída exclusivamente à qualidade do *backend* gerado, sem interferência de outros fatores.

Contudo, para que esta comparação fosse possível, o *backend* manual ARM teve que utilizar o mesmo *frontend* utilizado pelo *backend* gerado automaticamente, que corresponde ao *frontend* `llvm-gcc`. Os *backends* criados manualmente, por vezes, alteram também o *frontend* para que algumas características básicas da linguagem C sejam obedecidas. O `acllvmbe` não customiza estes aspectos e utiliza um *frontend* padrão x86, traduzindo com sucesso o código para uma arquitetura arbitrária. O *backend* manual ARM, entretanto, não produziu o resultado correto em alguns programas, devido à idiossincrasias do *frontend*. Esses poucos casos foram desconsiderados e a contagem de instruções omitida na tabela.

Na Tabela 6.3, a terceira linha mostra, por exemplo, que o programa CRC32 do Mibench, quando simulado no simulador para ARM gerado por ArchC, com uma arquivo de entrada pequeno, executou 106.827.226 instruções quando compilado com o gcc para

| Programa | Carga pequena gcc | LLVM | | acllvmbe | |
|---|---|---|---|---|---|
| Quick Sort | 43.108.593 | 40.355.153 | (-6%) | 41.498.004 | (-4%) |
| Bit Count | 89.791.275 | - | | 110.958.423 | (24%) |
| CRC32 | 106.827.226 | 99.983.461 | (-6%) | 117.778.730 | (10%) |
| ADPCM Coder | 76.801.887 | 65.233.864 | (-15%) | 99.265.918 | (29%) |
| ADPCM Decoder | 58.811.342 | 48.665.141 | (-17%) | 86.540.442 | (47%) |
| Dijkstra | 91.220.665 | - | | 170.376.345 | (87%) |
| Patricia | 89.911.550 | 87.653.621 | (-3%) | 89.134.698 | (-1%) |
| Rijndael Encode | 60.963.920 | 36.678.563 | (-40%) | 93.495.495 | (53%) |
| Rijndael Decode | 59.345.159 | 35.740.828 | (-40%) | 92.557.943 | (56%) |
| SHA | 38.147.258 | 22.874.547 | (-40%) | 36.407.317 | (-5%) |
| **Programa** | **Carga grande gcc** | **LLVM** | | **acllvmbe** | |
| Quick Sort | 140.151.850 | - | | 254.618.324 | (82%) |
| Bit Count | 1.344.426.276 | - | | 1.661.105.260 | (24%) |
| CRC32 | 2.076.635.460 | 1.943.580.015 | (-6%) | 2.289.525.652 | (10%) |
| ADPCM Coder | 1.530.880.948 | 1.308.116.148 | (-15%) | 1.971.254.083 | (29%) |
| ADPCM Decoder | 1.159.530.080 | 963.225.004 | (-17%) | 1.698.784.337 | (47%) |
| Dijkstra | 434.039.081 | - | | 833.975.356 | (92%) |
| Patricia | 552.377.323 | 538.779.359 | (-2%) | 547.756.096 | (-1%) |
| Rijndael Encode | 634.773.070 | 381.867.895 | (-40%) | 973.537.772 | (53%) |
| Rijndael Decode | 617.903.740 | 372.101.496 | (-40%) | 963.771.556 | (56%) |
| SHA | 397.169.072 | 238.130.203 | (-40%) | 379.054.616 | (-5%) |

Tabela 6.3: Número de instruções executadas na simulação de programas Mibench para ARM compilados com três compiladores diferentes, sem aplicar otimizações. A comparação percentual destaca a diferença entre o caso do compilador LLVM com *backend* manual e gerado automaticamente contra o caso do compilador gcc.

ARM versão 3.4.5, 99.983.461 instruções quando compilado com a infraestrutura LLVM que utiliza seu *backend* ARM original criado manualmente (redução de 6% no número de instruções quando comparado com o gcc) e 117.778.730 instruções quando compilado com a infraestrutura que utiliza o *backend* ARM gerado por `acllvmbe` (aumento de 10% no número de instruções quando comparado com o gcc). No caso de entradas grandes, os números de instruções executadas são 2.076.635.460, 1.943.580.015 (-6%) e 2.289.525.652 (+10%), respectivamente. Note que, se adotarmos o inverso do número de instruções executadas como um parâmetro para inferir o desempenho da execução, os programas produzidos com o compilador gcc estão piores do que aqueles produzidos pela infraestrutura LLVM 2.5 original. Isso acontece porque nesta tabela, todos os programas foram compilados sem otimizações.

A Tabela 6.4 apresenta os mesmos resultados, mas com todos os programas compilados com todas as otimizações oferecidas por cada suíte de compilação. No caso da infraestrutura LLVM, o programa `opt -O3` foi utilizado para aplicar otimizações independentes de máquina no *bitcode* intermediário. No caso do gcc, o parâmetro `-O3` foi passado para que as otimizações fossem aplicadas. Nestes casos, percebe-se que o compilador gcc possui

| Programa | Carga pequena gcc | LLVM | | acllvmbe | |
|---|---|---|---|---|---|
| Quick Sort | 38.014.320 | 38.758.376 | (2%) | 39.210.747 | (3%) |
| Bit Count | 43.665.614 | - | | 68.382.676 | (57%) |
| CRC32 | 93.138.546 | 94.507.415 | (1%) | 109.564.985 | (18%) |
| ADPCM Coder | 28.496.952 | 45.222.516 | (59%) | 87.347.510 | (207%) |
| ADPCM Decoder | 21.603.580 | 30.582.799 | (42%) | 73.032.781 | (238%) |
| Dijkstra | 54.296.017 | 60.553.077 | (12%) | 114.246.568 | (110%) |
| Patricia | 83.439.224 | 83.857.959 | (1%) | 85.614.471 | (3%) |
| Rijndael Encode | 34.492.042 | 27.124.918 | (-21%) | 54.881.327 | (59%) |
| Rijndael Decode | 34.450.875 | 26.362.402 | (-23%) | 54.860.209 | (59%) |
| SHA | 14.224.166 | 13.132.912 | (-8%) | 23.796.172 | (67%) |
| **Programa** | **Carga grande gcc** | **LLVM** | | **acllvmbe** | |
| Quick Sort | 132.120.218 | 129.299.489 | (-2%) | - | |
| Bit Count | 653.669.370 | - | | 1.023.227.556 | (57%) |
| CRC32 | 1.810.523.420 | 1.837.134.625 | (1%) | 2.129.857.891 | (18%) |
| ADPCM Coder | 565.799.031 | 870.397.231 | (54%) | 1.698.338.642 | (200%) |
| ADPCM Decoder | 424.193.067 | 596.896.174 | (41%) | 1.426.725.799 | (236%) |
| Dijkstra | 247.953.214 | 279.600.744 | (13%) | 550.825.976 | (122%) |
| Patricia | 513.660.736 | 516.418.835 | (1%) | 527.024.270 | (3%) |
| Rijndael Encode | 359.131.278 | 282.407.580 | (-21%) | 571.443.781 | (59%) |
| Rijndael Decode | 358.701.665 | 274.467.747 | (-23%) | 571.217.700 | (59%) |
| SHA | 148.052.353 | 136.688.143 | (-8%) | 247.726.101 | (67%) |

Tabela 6.4: Número de instruções executadas na simulação de programas Mibench para ARM compilados com três compiladores diferentes, aplicando otimizações. A comparação percentual destaca a diferença entre o caso do compilador LLVM com *backend* manual e gerado automaticamente contra o caso do compilador gcc.

maior sucesso em reduzir o número de instruções executadas. A Figura 6.2 apresenta dois gráficos, um para entradas pequenas e outro para entradas grandes, refletindo os valores da Tabela 6.3, sem otimizações. Já a Figura 6.3 apresenta os mesmos gráficos, mas baseada nos valores da Tabela 6.4, com as otimizações descritas.

Todavia, a comparação mais importante para este trabalho é do código produzido pelo *backend* gerado por `acllvmbe` com o código produzido pelo *backend* LLVM ARM original. Perceba que em todos os casos, `acllvmbe` gera um *backend* que produz um código que executa mais instruções para os mesmos programas. Esta diferença no número de instruções é também aparente na Tabela 6.5, que apresenta o tamanho da seção de código para todos os binários produzidos nestes testes (sem otimizações e com otimizações), demonstrando a capacidade de cada compilador em gerar código eficiente em espaço. O fato do *backend* gerado automaticamente produzir código sempre maior é um efeito direto da dificuldade de fazer com que um mecanismo de geração automática de código reconheça instruções de uso específico da arquitetura. O conhecimento de um conjunto de instruções mínimo para que seja possível gerar código para qualquer programa é crucial para garantir o funcionamento do compilador, mas não garante eficiência do código quanto ao número de

| Programa | Sem otimizações | | | | | Com otimizações | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | gcc | LLVM | | acllvmbe | | gcc | LLVM | | acllvmbe | |
| Quick Sort | 1.232 | 1.020 | (-17%) | 2.476 | (101%) | 780 | 848 | (9%) | 1.452 | (86%) |
| Bit Count | 5.876 | 5.588 | (-5%) | 6.708 | (14%) | 4.548 | 5.016 | (10%) | 6.888 | (51%) |
| CRC32 | 1.176 | 1.120 | (-5%) | 1.432 | (22%) | 1.068 | 948 | (-11%) | 1.444 | (35%) |
| ADPCM Coder | 2.248 | 1.912 | (-15%) | 2.728 | (21%) | 1.256 | 1.456 | (16%) | 2.492 | (98%) |
| ADPCM Decoder | 2.224 | 1.900 | (-15%) | 2.704 | (22%) | 1.244 | 1.444 | (16%) | 2.464 | (98%) |
| Dijkstra | 2.428 | 1.992 | (-18%) | 3.036 | (25%) | 2.100 | 2.128 | (1%) | 3.708 | (77%) |
| Patricia | 5.440 | 4.964 | (-9%) | 6.328 | (16%) | 3.804 | 3.564 | (-6%) | 5.132 | (35%) |
| Rijndael | 25.928 | 16.892 | (-35%) | 43.956 | (70%) | 14.564 | 11.020 | (-24%) | 25.476 | (75%) |
| SHA | 3.152 | 2.556 | (-19%) | 4.036 | (28%) | 2.348 | 2.072 | (-12%) | 3.812 | (62%) |

Tabela 6.5: Tamanho, em bytes, da seção de código dos binários Mibench para ARM compilados com três compiladores diferentes, para os casos sem otimizações e com otimizações. A comparação percentual destaca a diferença do tamanho de código produzido pelo compilador LLVM com *backend* manual e gerado automaticamente contra o tamanho de código produzido pelo compilador gcc.

instruções a serem executadas pelo programa. Quanto mais informação sobre as instruções disponíveis em conjunto de instruções é utilizada, melhor o *backend* gerado.

O *backend* gerado possui padrões da linguagem intermediária da infraestrutura LLVM fixos para qualquer arquitetura. Por exemplo, o padrão trivial que aborda a soma de dois registradores faz parte de todos os *backends* gerados por `acllvmbe`, e a responsabilidade do sistema é encontrar a implementação de cada padrão utilizando instruções da arquitetura alvo. Por esse motivo, uma instrução específica que realiza o cálculo representado por um padrão que não se encontra nesse conjunto fixo de padrões nunca será utilizada. Já a abordagem de escrita manual de um *backend* LLVM, utilizando a linguagem TableGen (Capítulo 3), é realizada de forma que cada instrução atenda um padrão diferente. Se este conjunto de padrões não for suficiente para cobrir todos os casos da linguagem intermediária, o desenvolvedor deve adicionar manualmente padrões extras, ou configurar com código C++ as etapas de *lowering* para transformar os casos não cobertos em casos cobertos. Esta diferença fundamental no projeto entre o *backend* gerado automaticamente e o *backend* escrito manualmente com TableGen faz com que o primeiro utilize menos instruções específicas, aumentando o tamanho do código, como pode ser observado pelos resultados dos experimentos apresentados nas Figuras 6.2 e 6.3.

Contudo, estes resultados são uma consequência do compromisso flexibilidade do *backend* contra automatização da geração de suas partes. Por exemplo, programando um novo *backend* LLVM em C++ com a *Application Programming Interface*(API) do gerador de código independente do LLVM, é possível especificar exatamente quais instruções da máquina serão emitidas no prólogo e epílogo de cada função ou mesmo especificar qual instrução é melhor para ser utilizada quando um objeto na pilha é endereçado. Estas

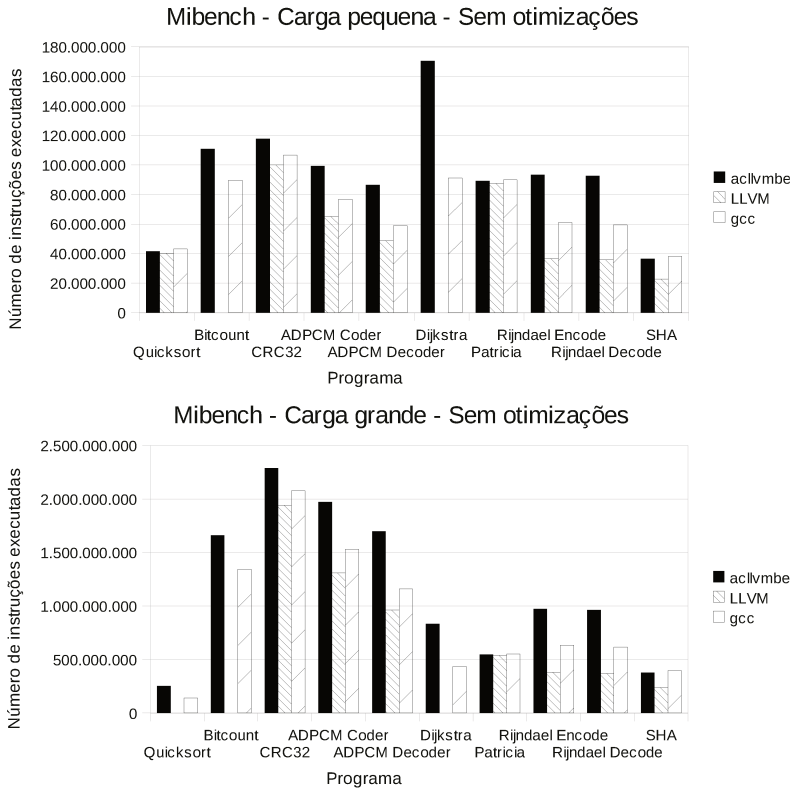



Figura 6.2: Número de instruções executadas na simulação de programas Mibench para ARM compilados com três compiladores diferentes, sem aplicar otimizações.

são decisões importantes que são tomadas automaticamente pelo sistema `acllvmbe`. Para isso, um padrão de acesso a memória é gerado e o sistema é consultado para concluir quais instruções podem ser utilizadas para endereçar um objeto na pilha. No caso do prólogo e epílogo, um modelo padrão de pilha é usado, e o usuário deve especificar no modelo do processador se a pilha é crescente ou decrescente.

O resultado é que criar um *backend* LLVM em C++ manualmente com um único programador é uma tarefa de aproximadamente 3 meses, se levado como exemplo a construção do *backend* MIPS pelo aluno de doutorado Bruno Cardoso Lopes do Laboratório de Sistemas de Computação do Instituto de Computação da UNICAMP. Este projeto foi incorporado ao LLVM e atualmente é o *backend* MIPS oficial. Já criar um *backend* utilizando `acllvmbe` é uma tarefa de 2 dias, que foi o tempo levado para criar os *backends* ARM, MIPS, SPARC e PowerPC deste projeto. Isto acontece porque toda a tarefa se resume em consultar um manual da arquitetura e descrever o comportamento de cada instrução, sem a necessidade de tomar decisões específicas de domínio de especialistas

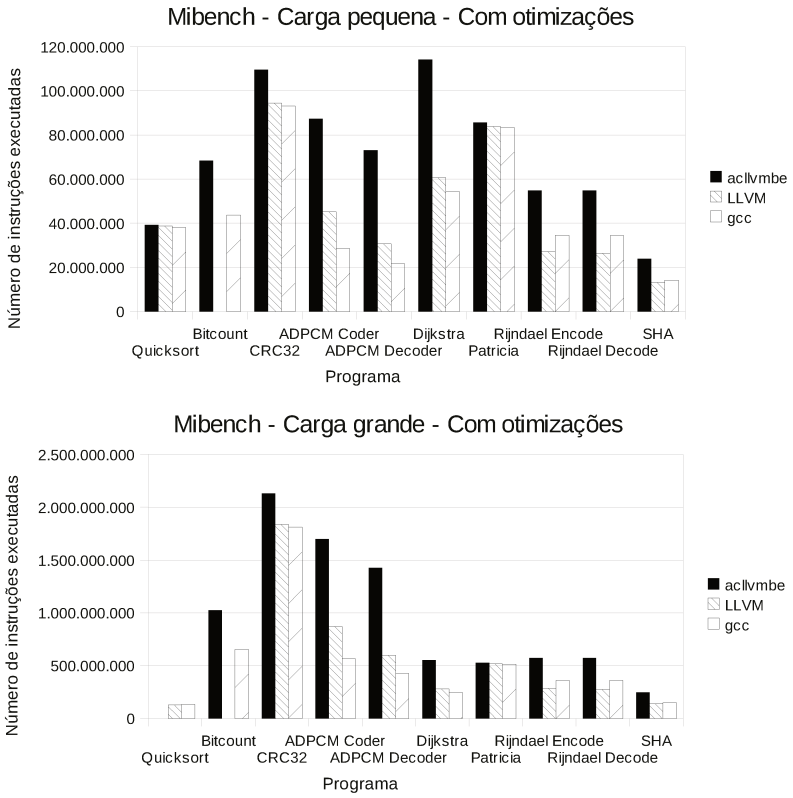



Figura 6.3: Número de instruções executadas na simulação de programas Mibench para ARM compilados com três compiladores diferentes, aplicando otimizações.

da área de compiladores. Esta automatização, portanto, apresenta sucesso para fornecer rapidamente um compilador para uma arquitetura alvo, ainda que o código não seja tão eficiente quanto o de um compilador específico. Esta característica é muito útil para uso em projetos incrementais onde uma plataforma precisa de um compilador inicial para que o *software* seja desenvolvido tão cedo quanto o *hardware*, diminuindo os tempos necessários para o projeto total. Em uma etapa em que a plataforma *hardware* esteja amadurecida, um compilador construído manualmente, com maior qualidade de código, pode ser desenvolvido para substituir o compilador inicial gerado automaticamente. Ainda, esta etapa de construção do compilador manual não é completamente desassistida, uma vez que os desenvolvedores podem utilizar o *backend* gerado automaticamente como ponto de partida, economizando muito tempo.

É importante observar que a qualidade do código gerado é dependente da quantidade que cada padrão, ou fragmento de linguagem intermediária LLVM, é utilizado. Cada padrão possui uma implementação fixa com instruções da máquina alvo e não possuem

a mesma diferença de qualidade entre eles. Existem padrões que, em relação ao *backend* ARM oficial, possuem um aumento do número de instruções maior do que outros. Por esse motivo, ocorre a perda de desempenho desigual observada nos gráficos das Figuras 6.2 e 6.3. Para alguns programas, como Patricia, há pouca diferença entre a quantidade de instruções executadas pelo código gerado pelo *backend* sintetizado por `acllvmbe` e o *backend* ARM oficial LLVM. Para outros programas, como Dijkstra, o código criado com `acllvmbe` executa quase o dobro do número de instruções. O caso Dijkstra exercita especificamente os padrões que foram implementados menos eficientemente pelo *backend* ARM, enquanto Patricia não sofre deste problema.

No caso específico da arquitetura ARM, esta diferença de quantidade de instruções acontece principalmente devido à incapacidade do sistema `acllvmbe` de incorporar as instruções de carregamento e armazenamento múltiplo de memória, utilizadas no prólogo de cada função e também na estratégia de carregamento de imediatos longos. No sistema `acllvmbe`, para abordar o problema de uma forma geral, um imediato de 32 bits pode ser quebrado em diversos imediatos menores, para serem unidos com a ajuda de instruções de deslocamento e adição. Isto pode levar à sequências de código de até 8 instruções, enquanto o *backend* ARM original aborda casos mais específicos com maior eficiência. Por exemplo, se o imediato de 32 bits é o número `-1` em complemento de 2, este pode ser carregado na memória com uma instrução de negação de um registrador que esteja com seu conteúdo igual a zero. Neste caso específico, o problema poderia ser amenizado com o uso de um otimizador *peephole* atuando antes da emissão do código, que é um projeto futuro factível para o sistema `acllvmbe`. O otimizador substituiria sequências de código que aparecem com frequência por sequências menores, que realizam o mesmo trabalho, mas de forma mais eficiente.

No caso das instruções de carregamento múltiplo de memória, seu uso é restrito ao prólogo e epílogo das funções, em que geralmente um grande número de registradores é salvo e carregado da memória. No caso do ARM, uma única instrução pode resolver o problema. Mas o sistema `acllvmbe` gera uma sequência de instruções com uma instrução de carga ou armazenamento para cada registrador carregado ou salvo. Apesar do desempenho das duas sequências ser teoricamente o mesmo, uma vez que os dois trechos vão sofrer o mesmo atraso para acessar a mesma quantidade de posições de memória, o uso de uma única instrução para carregar múltiplas posições, como possível no ARM e utilizado pelo *backend* ARM oficial, diminui o tamanho de código e consequentemente diminui a pressão que o programa tem sobre a *cache* de instruções do sistema. Para ilustrar esses casos, o código ARM gerado para o primeiro bloco básico da função `dijkstra` pelo *backend* sintetizado por `acllvmbe` e pelo *backend* LLVM ARM oficial é mostrado na Figura 6.4.

Repare na figura que a versão `acllvmbe` possui mais do que o dobro do número de instruções do que a versão LLVM original. As instruções `store` para salvar registradores

acllvmbe

```
        .align  2
        .globl  dijkstra
dijkstra:
        sub r13, r13, #32
        str r14, [r13, #4]
        str r4, [r13, #28]
        str r5, [r13, #24]
        str r6, [r13, #20]
        str r7, [r13, #16]
        mov r2, #(1620 >> 8)
        mov r2, r2, LSL #8
        mov r3, #(1608 >> 8)
        add r2, r2, #(1620 & 0xFF)
        mov r3, r3, LSL #8
        add r2, PC, r2
        add r3, r3, #(1608 & 0xFF)
        add r3, PC, r3
        mov r4, #0
        ldr r2, [r2]
        ldr r3, [r3]
        str r4, [r2]
        str r1, [r13, #12]
        mov r5, r0
```

LLVM original

```
        .globl  dijkstra
        .align  2
dijkstra:
        stmfd sp!, {r4, r5, r6, r7, lr}
        ldr r3, .LCPI5_0
        mov r2, #0
        str r2, [r3]
        ldr r3, .LCPI5_1
        mov r4, r1
        mov r5, r0
```

Figura 6.4: Comparação entre código ARM gerado pelo *backend* LLVM sintetizado por `acllvmbe`, à esquerda, e pelo *backend* LLVM ARM original, à direita, para o primeiro bloco básico da função `dijkstra`.

*callee saved* são todas substituídas por uma única instrução ARM `stmfd`. Esta instrução armazena os registradores da lista em uma pilha cheia decrescente. As instruções para cálculo de endereço relativo ao *program counter* para acessar os dados nas posições informadas pelas etiquetas `.LCPI5_0` e `.LCPI5_1` são todas substituídas por uma única instrução de carga para cada etiqueta. A forma comprimida `ldr r3, .LCPI5_0` de escrita instrui o montador ARM a utilizar um endereçamento base mais deslocamento, em que a base é o registrador *program counter* do ARM e o deslocamento é o número imediato necessário para alcançar, a partir desta instrução, o dado. No caso de `acllvmbe`, é feita a suposição de que os imediatos 1620 e 1608, que correspondem ao deslocamento em relação ao *program counter* necessário para endereçar estes dados, devem ser quebrados em dois imediatos distintos de forma a serem codificados em 8 bits. Contudo, a arquitetura ARM dispõe, geralmente, de 13 bits para armazenar um imediato, não necessitando de instruções extras, neste caso. O sistema `acllvmbe` não oferece restrições para tratar ar-

quiteturas com imediatos de 13 bits (diferente de 16 ou 8 bits). Entretanto, para diminuir a o tempo de geração dos *backends*, foram criadas regras para quebrar imediatos apenas de 16 ou 8 bits. O caso ARM foi modelado como 8 bits, pois é um valor menor do que 13 e que seguramente caberia nas instruções. Portanto, o sistema não tem conhecimento desta vantagem da arquitetura em tratar imediatos maiores.

O exemplo acima ilustra bem o compromisso tempo de geração contra qualidade do código, que será explorado mais adiante. Para diminuir a quantidade de regras no algoritmo de busca e tornar os tempos de geração mais rápidos, menos casos de imediatos são previstos, ao custo de potencialmente gerar código menos eficiente. Contudo, este caso específico pode ser resolvido facilmente por um otimizador *peephole*.

Os dados mostrados nos gráficos das Figuras 6.2 e 6.3, portanto, mostram um desempenho pior para `acllvmbe` especificamente quando há muitas chamadas de funções (em que o caso do prólogo subótimo é executado diversas vezes) e uso de imediatos grandes. Todavia, essa diferença na qualidade do código pode ser reduzida por um simples otimizador *peephole* guloso, que substitui as janelas de sequências de desempenho inferior por sequências mais curtas e eficientes. O problema de geração automática de otimizadores *peephole* foi tema de diversos trabalhos de pesquisa [12, 17, 18, 26, 39, 40, 42, 50] e pode ser facilmente acoplado ao projeto `acllvmbe` atual, mostrando-se, portanto, como um trabalho futuro importante.

### Cobertura do Conjunto de Instruções

Como a implementação dos padrões LLVM é inferida automaticamente pelo algoritmo de busca, para investigar a capacidade do *backend* gerado em utilizar diferentes instruções do conjunto de instruções ARM, a Tabela 6.6 apresenta os resultados de um experimento que visa medir a cobertura do conjunto de instruções ARM. A cobertura é avaliada pelo cálculo da razão entre o número de diferentes tipos de instruções utilizadas na execução de um programa pelo número total de tipos de instruções na arquitetura ARM. Ainda, como o experimento contempla a execução de programas Mibench que utilizam a biblioteca padrão C glibc 2.3.6 compartilhada compilada pelo mesmo compilador (gcc 3.4.5), para evitar que o código desta biblioteca interferisse na contagem, apenas o código do aplicativo foi levado em consideração, isto é, código que foi efetivamente gerado pelo compilador em teste.

A cobertura é calculada medindo-se a frequência de uso de cada instrução descrita no modelo ARM para ArchC versão 0.7.0. O quociente é o total de tipos de instruções descritas neste modelo. Este modelo foi construído representando todas as instruções da arquitetura ARMv5e. É importante notar que algumas instruções foram representadas com apenas um opcode, ao passo que outras, como a maioria das instruções de processamento de dados, são representadas com três tipos distintos de modelo de instrução ArchC,

uma para cada modo de endereçamento. Existem ainda 12 tipos de instruções do tipo *load* e 12 do tipo *store*, cuja diferença não está somente no modo de endereçamento mas também na largura da requisição (*byte*, *half* ou *word*). O modelo ARM para ArchC versão 0.7.0 é código livre disponível no sítio do projeto ArchC. Ao todo, somam-se 100 tipos diferentes de *opcodes* considerados. Portanto, caso algum tipo de instrução associado a um modo de endereçamento não seja utilizado por um programa, sua cobertura do conjunto de instruções ARM decresce em uma unidade percentual. A tabela apresenta a cobertura separadamente para cada programa em duas versões distintas: executado com entradas pequenas e grandes. No final, a média e seu desvio padrão, bem como valores de máximo e mínimo são apresentados. O total, na última linha da tabela, representa a cobertura calculada com o histograma de frequência de uso das instruções que contempla a soma de todas as execuções, realizadas para cada compilador.

Pela tabela, é possível concluir que o *backend* gerado por `acllvmbe` possui cobertura semelhante ao dos dois outros compiladores programados manualmente, LLVM e gcc. Ainda que, na média, o *backend* LLVM tenda a gerar código que utilize menos tipos diferentes de instruções, na execução do programa `Quicksort` com carga grande, apresentou a maior diversidade de instruções utilizadas, isto é, maior cobertura do conjunto de instruções ARM.

| **Programa** | **Carga pequena** | | | **Carga grande** | | |
|---|---|---|---|---|---|---|
| | **acllvmbe** | **LLVM** | **gcc** | **acllvmbe** | **LLVM** | **gcc** |
| Quick Sort | 19% | 20% | 18% | 37% | 36% | 36% |
| Bit Count | 34% | - | 35% | 33% | - | 35% |
| CRC32 | 20% | 22% | 22% | 20% | 22% | 22% |
| ADPCM Coder | 24% | 27% | 22% | 24% | 28% | 22% |
| ADPCM Decoder | 23% | 26% | 21% | 23% | 27% | 21% |
| Dijkstra | 21% | 23% | 18% | 21% | 23% | 19% |
| Patricia | 26% | 27% | 25% | 26% | 27% | 25% |
| Rijndael Encode | 24% | 29% | 25% | 24% | 29% | 25% |
| Rijndael Decode | 23% | 29% | 24% | 23% | 29% | 24% |
| SHA | 23% | 26% | 27% | 23% | 26% | 27% |
| **Média** | **24,55±4,77%** | **26,44±3,65%** | **24,65±5,28%** | | | |
| Máximo | 37% | 36% | 36% | | | |
| Mínimo | 19% | 20% | 18% | | | |
| **Total** | **43%** | **48%** | **45%** | | | |

Tabela 6.6: Percentagem total do conjunto de instruções ARM coberta em cada execução dos programas Mibench compilador com 3 tipos de compiladores diferentes: com *backend* LLVM gerado automaticamente, com *backend* LLVM escrito manualmente e para o gcc versão 3.4.5.

Para entender melhor os problemas de cobertura do conjunto de instruções da arquitetura ARM, a Tabela 6.7 detalha todas as instruções que não foram usadas, durante todos os testes, por algum *backend* de compilador ARM. Ainda, para cada instrução, sua sin-

taxe é apresentada, para apresentar especificamente qual modo de endereçamento desta instrução não foi utilizado. Caso este tipo de instrução não seja utilizado em nenhum de seus modos de endereçamento, a sintaxe não especifica operandos para que fique evidente que todo este *opcode* foi inutilizado, independentemente da forma de endereçar seus operandos. Essas sintaxes são marcadas com uma nota de rodapé. Por exemplo, a primeira linha da tabela diz que o segundo modo de endereçamento (dos três disponíveis) da instrução `mvn`, *Move and negate*, que desloca um de seus operandos por uma quantidade arbitrária, especificada em registrador, de bits, nunca é utilizada por nenhum *backend*. Vale observar que para a maioria dos casos em que uma instrução de processamento de dados aparece na lista, apenas um dos três modos de endereçamento não foi utilizado e, este é sempre o mesmo modo de endereçamento, que oferece a possibilidade de se deslocar um dos operandos por um número arbitrário de casas, para a direita ou para a esquerda, utilizando deslocamento aritmético, lógico ou rotação, em que o número de casas a se deslocar está armazenado em um registrador. Este modo de endereçamento dificilmente é utilizado em conjunto com operações de processamento de dados comuns. Entretanto, ele foi utilizado na instrução `mov`, em que o resultado final é o de apenas mover um resultado deslocado para o registrador destino, como uma instrução de deslocamento de uma arquitetura RISC padrão. Através dos resultados mostrados nesta tabela, fica claro que, para a carga de trabalho oferecida pelo *Mibench* com os compiladores utilizados, este modo de endereçamento poderia ser excluído e transformado em uma instrução de propósito específico para deslocamento.

Como o modelo ARM para ArchC especifica uma instrução separada para cada combinação de *opcode* de processamento de dados com um dos três modos de endereçamento possíveis, aliado ao fato de que um desses três modos é largamente inutilizado, observa-se na Tabela 6.6 uma cobertura reduzida (menos de 50%) por parte de todos os compiladores.

| Instrução | Descrição | acllvmbe | LLVM | gcc |
|---|---|---|---|---|
| mvn{<cond>}{s} <Rd>, <Rn>, <shift> <Rs> | Realiza a negação do registrador Rn deslocado pelo conteúdo do registrador Rs de acordo com o tipo de deslocamento especificado em shift. Armazena o resultado em Rd. | | | |
| adc{<cond>}{s} <Rd>, <Rn>, <Rm>, <shift> <Rs> | Soma com *carry* de Rn com Rm deslocado pelo conteúdo de Rs com o tipo de deslocamento especificado em shift. Armazena o resultado em Rd. | | | |
| clz[1] | *Count leading zeros* retorna o número de bits zero que precedem o primeiro bit um no valor do registrador fonte. | | | |
| cmp{<cond>}{s} <Rn>, <Rm>, <shift> <Rs> | Comparação do valor de Rm com Rs deslocado. | | | |
| mla{<cond>}{s} <Rd>, <Rm>, <Rs>, <Rn> | *Multiply and accumulate* | | • | |
| bic{<cond>}{s} <Rd>, <Rn>, <shift> <Rs> | *Bit clear* irá realizar a operação "E" lógico com o complemento do valor Rn deslocado de Rs casas. Resultado armazenado em Rd. | | | |
| eor{<cond>}{s} <Rd>, <Rn>, <Rm> <shift> <Rs> | "OU" exclusivo do valor de Rn com o complemento do valor Rm deslocado de Rs casas. Resultado armazenado em Rd. | | | |
| teq{<cond>} <Rd>, <Rn>, <shift> <Rs> | *Test equivalence* atualiza o registrador de estado baseado em uma comparação por "OU" exclusivo entre os dois operandos. | | | |
| ldrt, ldrbt, strt e strbt[1] | *Load/Store register with translation* são instruções que, caso estejam em modo privilegiado, tratam o acesso à memória como se fosse modo usuário. | | | |
| sbc[1] | *Subtract with carry* foi originalmente concebida para realizar subtrações de múltiplas palavras. É utilizada apenas por gcc em um de seus três modos de endereçamento. | | | • |
| rsc[1] | *Reverse subtract with carry* cobre o mesmo caso da instrução `sbc`, mas invertendo seus operandos. Porém, nesta versão, não é utilizada nem por gcc. | | | |
| ldrh[1] | *Load register halfword* carrega apenas meia palavra (16 bits) da memória. Sua contrapartida, *store halfword*, contudo, é utilizado por todos os *backends* exceto gcc. | | | |
| tst[1] | Instrução de comparação que atualiza o registrado estado através da comparação de um "E" lógico entre seus dois operandos. | | • | |
| add{<cond>}{s} <Rd>, <Rn>, <Rm>, <shift> <Rs> | Soma de Rn com Rm deslocado pelo conteúdo de Rs com o tipo de deslocamento especificado em shift. Armazena o resultado em Rd. | | | |
| and{<cond>}{s} <Rd>, <Rn>, <Rm>, <shift> <Rs> | "E" lógico entre Rn e Rm deslocado pelo conteúdo de Rs com o tipo de deslocamento especificado em shift. Armazena o resultado em Rd. | | | • |
| cmn{<cond>}{s} <Rm>, <Rn>, <shift> <Rs> | Realiza a comparação entre Rm e o complemento do registrador Rn deslocado pelo conteúdo do registrador Rs de acordo com o tipo de deslocamento especificado em shift. | | | |
| sub{<cond>}{s} <Rd>, <Rn>, <Rm>, <shift> <Rs> | Subtração de Rn com Rm deslocado pelo conteúdo de Rs com o tipo de deslocamento especificado em shift. Armazena o resultado em Rd. | | | |
| swp e swpb[1] | *Swap* e *Swap byte* realizam a troca de um dado (palavra ou byte, dependendo da instrução utilizada) entre registrador e memória. Pode ser utilizada para implementar semáforos. | | | |
| bkpt <imediato> | *Breakpoint* possui o propósito específico de depurar programas. | | | |
| smull[1] | *Signed multiply long* produz uma multiplicação sinalizada com resultado de 64 bits. | • | • | |
| umull[1] | *Unsigned multiply long* produz uma multiplicação não sinalizada com resultado de 64 bits. | | | |
| smlal[1] | *Signed multiply accumulate long* produz uma multiplicação sinalizada com resultado de 64 bits e soma a este resultado um valor de 64 bits obtido com a concatenação de dois registradores. | | | |

[1]Todas as formas de se endereçar os operandos não são utilizadas

Tabela 6.7: Lista de instruções que não foram cobertas por algum dos compiladores durante o experimento de cobertura do conjunto de instruções ARM da Tabela 6.6, apresentando sua sintaxe, descrição e se a instrução foi coberta pelo *backend* ARM gerado por `acllvmbe`, LLVM original ou por gcc.

A Figura 6.5 traz mais informações sobre o experimento, apresentando não apenas a cobertura, mas três histogramas com a frequência de ocorrência das 30 instruções da arquitetura ARM mais frequentes no código gerado por cada compilador para os testes expostos na Tabela 6.6. Repare que o histograma das instruções geradas por `acllvmbe` revela que o *backend* tende a utilizar menos tipos variados de instruções do que os demais, revelando oportunidade para otimizações.

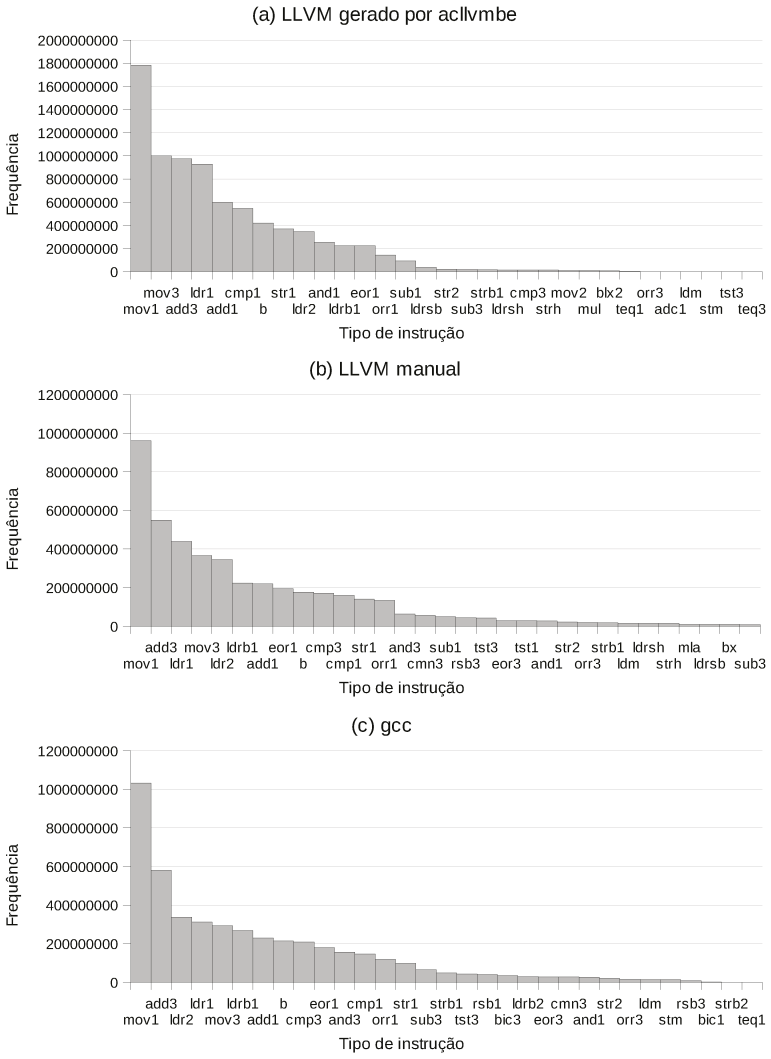



Figura 6.5: Histogramas com a frequência de ocorrência de instruções ARM para o *benchmark* Mibench compilado com três compiladores diferentes.

### 6.1.2 SPARC, MIPS e PowerPC

#### Metodologia de Validação

Por motivos de disponibilidade da plataforma de simulação ARM e prévia experiência, a arquitetura ARM foi escolhida para os primeiros testes e o compilador produzido foi testado com programas maiores do *benchmark* Mibench. No intuito de mostrar que o sistema `acllvmbe` não está limitado à arquitetura ARM e realmente possui flexibilidade suficiente para gerar compiladores para outras arquiteturas, foi construído um modelo também para as arquiteturas SPARC, MIPS e PowerPC e, a partir desses modelos, um *backend* LLVM para cada uma dessas arquiteturas foi gerado.

Há dificuldades para testar essas plataformas com programas como aqueles da Tabela 6.2, contudo, pois são programas que utilizam a biblioteca C. O suporte dos simuladores ArchC está limitado ao uso da biblioteca C `newlib` [59], com exceção da arquitetura ARM, para a qual foram feitas as adequações necessárias durante os trabalhos relativos aos ligadores dinâmicos desenvolvidos previamente [7]. Entretanto, o *frontend* utilizado `llvm-gcc` é dependente da biblioteca C `glibc`. Como a proposta deste trabalho é o de gerar um compilador *bare metal*, isto é, que não é usado para produzir aplicativos de usuário dependentes da biblioteca padrão C, mas sim que utilizam diretamente o *hardware*, foi desenvolvido um programa pequeno em C que utiliza diretamente as chamadas de sistema da plataforma ArchC e é independente de biblioteca C. Por exemplo, ao invés de utilizar a função `printf()` para imprimir texto na saída padrão, a chamada de sistema `write()` provida pela plataforma ArchC é diretamente usada.

Dessa maneira, cria-se um método desvinculado ao uso da biblioteca padrão C para mostrar que é possível gerar automaticamente um compilador para as arquiteturas SPARC, MIPS e PowerPC. É importante ressaltar que a restrição de que o compilador gerado para essas arquiteturas seja *bare metal* não é tão forte quanto parece. Os compiladores *bare metal* não podem ser usados para compilar aplicativos de usuário, mas são essenciais para compilar sistemas operacionais, como o *kernel* do Linux, que não podem depender de biblioteca de modo usuário como a `glibc` ou `newlib`. Na criação de plataformas embarcadas com arquiteturas específicas, para as quais ainda não existe um compilador, este é justamente o caso, em que também não há um sistema operacional para a arquitetura e o *software* a ser desenvolvido precisa ser de baixo nível o suficiente para manipular o sistema diretamente, sem a biblioteca padrão C.

#### Resultados

Após gerados os respectivos *backends* para as arquiteturas ARM, MIPS, SPARC e PowerPC, o código apresentado na Figura 6.6 foi compilado e uma versão do programa para cada máquina foi criada. Os simuladores gerados com a ferramenta `acsim` [8] da plataforma

**teste.c**

```
/* teste.c
   Este código utiliza diretamente as chamadas de sistema
   oferecidas pela plataforma de simulação ArchC.
*/

/* Declarações de tipo */
typedef void (*functype)();

/* Declarações de funções locais */
static void init();
static void wr(int, const char *, int);
static void ex();
static void print(const char *);

/* Definição das funções */

/* Ponto de entrada.
   Esta é a primeira função executada e irá saltar para a
   função main(), executar o programa e então utilizar a
   chamada de sistema exit().
*/
void init() {
  main();
  ex();
}

/* Executa a chamada de sistema write() */
void wr(int fd, const char *buf, int count) {
  functype func = (functype) 0x50;
  func();
}

/* Executa a chamada de sistema exit() */
void ex() {
  functype func = (functype) 0x64;
  func();
}
```

```
/* Função simples para contar o tamanho da string e chamar
   a função write() com os parâmetros corretos. */
void
print(const char * str) {
  char c;
  int sz = 0;
  do {
    c = str[sz++];
  } while (c != '\0');
  --sz;
  wr(1, str, sz);
}

/* Função main() */
int
main() {
  print("Ola, mundo!");

  return 0;
}
```

Figura 6.6: Código C independente de biblioteca padrão que é compilado para as arquiteturas SPARC, MIPS e PowerPC para testar a versatilidade do sistema `acllvmbe` em gerar compiladores para diferentes arquiteturas.

ArchC foram utilizados para executar os binários gerados e todos produziram os resultados corretos. A Tabela 6.8 apresenta uma comparação do tamanho da seção ELF `.text`, que contém as instruções do programa, dos binários compilados com cada *backend* gerado. Os programas compilados para MIPS e SPARC ficaram maiores, uma vez que estas arquiteturas possuem *delay slot*. O *delay slot* ocorre em arquiteturas com *pipeline* simples em que a instrução imediatamente após uma instrução de salto é sempre executada. Ainda, no caso de uma instrução de carga de valor da memória para um registrador, o valor do registrador pode não estar pronto para uso na próxima instrução.

Esses são casos típicos em que um algoritmo escalonador de instruções deveria alterar a ordem das instruções do programa para evitar *hazards*, ou seja, as condições que exercitam casos em que a presença de um *pipeline* no processador cria efeitos não esperados pelo programador. Contudo, neste trabalho, o foco foi desenvolver um seletor de instruções. O alocador de registradores foi fornecido pela infraestrutura LLVM, ao passo que o escalonador de instruções utilizado é trivial. A especialização desses dois outros

| **Backend** | **Tamanho da seção .text** |
|---|---|
| ARM | 308 bytes |
| MIPS | 584 bytes |
| SPARC | 468 bytes |
| PowerPC | 372 bytes |

Tabela 6.8: Comparação do tamanho do código produzido por cada *backend* para o programa de teste cujo código foi apresentado na Figura 6.6.

componentes do *backend* de forma a levar em consideração características da arquitetura alvo é um trabalho futuro. Desse modo, a política para resolver *hazards* consiste em conservadoramente inserir instruções nulas (`nops`) após instruções que possuem o potencial para *hazard*.

A Figura 6.7 mostra o código em linguagem de montagem gerado por cada *backend* para a função `print` do programa C da Figura 6.6. Este exemplo é importante para mostrar como a estrutura de blocos básicos permanece a mesma entre todos os programas. A única diferença são as instruções emitidas, que abordam os casos específicos para cada arquitetura segundo a implementação de um fragmento da linguagem intermediária da infraestrutura LLVM encontrada pelo algoritmo de busca. A figura também torna evidente a estrutura comum entre todos os *backends* que foram gerados automaticamente pelo sistema `acllvmbe`, na qual a diferença consiste basicamente no seletor de instruções especializado para cada arquitetura.

**teste.c - Função print()**

**ARM**

```
        .align  2
        .globl  print
        .type   print, %function
print:

        sub r13, r13, #24
        str r14, [r13, #4]
        str r0, [r13, #20]
        mov r0, #0
        str r0, [r13, #16]

.BB4_1: @ bb
        ldr r0, [r13, #16]
        ldr r1, [r13, #20]
        add r0, r1, r0
        ldrb r0, [r0]
        add r1, r13, #15
        strb r0, [r1]
        ldr r0, [r13, #16]
        add r0, r0, #1
        str r0, [r13, #16]
        ldrb r0, [r1]
        mov r1, #0
        cmp r0, r1
        bne .BB4_1      @ bb

.BB4_2: @ bb1
        mov r0, #255
        mov r0, r0, LSL #8
        add r0, r0, #255
        mov r1, r0, LSL #16
        add r0, r1, r0
        ldr r1, [r13, #16]
        add r2, r1, r0
        str r2, [r13, #16]
        ldr r1, [r13, #20]
        mov r0, #1
        bl wr

.BB4_3: @ return
        add r13, r13, #24
        ldr r14, [r13, #-20]
        mov r15, r14
```

**MIPS**

```
        .align  2
        .globl  print
        .type   print, @function
print:

        add $14, $0, 24
        sub $29, $29, $14
        addi $14, $29, 4
        sw $31, ($14)
        addi $1, $29, 20
        sw $4, ($1)
        addi $1, $29, 16
        add $2, $0, 0
        sw $2, ($1)

.BB4_1: # bb
        addi $1, $29, 20
        addi $2, $29, 16
        lw $2, ($2)
        nop
        lw $1, ($1)
        nop
        addu $1, $1, $2
        lbu $1, ($1)
        nop
        addi $2, $29, 15
        sb $1, ($2)
        addi $1, $29, 16
        lw $1, ($1)
        nop
        addi $3, $29, 16
        addi $1, $1, 1
        sw $1, ($3)
        lbu $1, ($2)
        nop
        add $2, $0, 0
        bne  $1, $2, .BB4_1
        nop

.BB4_2: # bb1
        add $1, $0, 255
        sll $1, $1, 8
        addi $1, $1, 255
        sll $2, $1, 16
        addu $1, $2, $1
        addi $2, $29, 16
        lw $2, ($2)
        nop
        addi $3, $29, 16
        addu $6, $2, $1
        sw $6, ($3)
        addi $1, $29, 20
        lw $5, ($1)
        nop
        add $4, $0, 1
        jal wr
        nop

.BB4_3: # return
        addi $29, $29, 24
        addi $14, $29, -20
        lw $31, ($14)
        nop
        jr $31
        nop
```

**SPARC**

```
        .align  2
        .globl  print
        .type   print, @function
print:

        sub %r14, 24, %r14
        add %r14, 4, %r7
        st %r15, [%r7]
        add %r14, 20, %r1
        st %r8, [%r1]
        add %r14, 16, %r1
        mov 0, %r2
        st %r2, [%r1]

.BB4_1: # bb
        add %r14, 20, %r1
        add %r14, 16, %r2
        ld [%r2], %r2
        ld [%r1], %r1
        add %r1, %r2, %r1
        ldub [%r1], %r1
        add %r14, 15, %r2
        stb %r1, [%r2]
        add %r14, 16, %r1
        ld [%r1], %r1
        add %r14, 16, %r3
        add %r1, 1, %r1
        st %r1, [%r3]
        ldub [%r2], %r1
        mov 0, %r2
        cmp %r1, %r2
        bne .BB4_1
        nop

.BB4_2: # bb1
        mov 255, %r1
        sll %r1, 8, %r1
        add %r1, 255, %r1
        sll %r1, 16, %r2
        add %r2, %r1, %r1
        add %r14, 16, %r2
        ld [%r2], %r2
        add %r14, 16, %r3
        add %r2, %r1, %r10
        st %r10, [%r3]
        add %r14, 20, %r1
        ld [%r1], %r9
        mov 1, %r8
        call wr
        nop

.BB4_3: # return
        add %r14, 24, %r14
        add %r14, -20, %r7
        ld [%r7], %r15
        retl
        nop
```

**PowerPC**

```
        .align  2
        .globl  print
        .type   print, @function
print:

        li 11, 24
        subf 13, 11, 13
        mflr 24
        stw 24, 4 (13)
        stw 3, 20 (13)
        li 0, 0
        stw 0, 16 (13)

.BB4_1: # bb
        lwz 0, 16 (13)
        lwz 3, 20 (13)
        add 0, 3, 0
        lbz 0, 0 (0)
        addi 3, 13, 15
        stb 0, 0 (3)
        lwz 0, 16 (13)
        addi 0, 0, 1
        stw 0, 16 (13)
        lbz 0, 0 (3)
        li 3, 0
        cmpw 0, 0, 3
        bc 4, 2, .BB4_1 # bb

.BB4_2: # bb1
        lwz 0, 16 (13)
        mflr 3
        add 5, 0, 3
        stw 5, 16 (13)
        lwz 4, 20 (13)
        li 3, 1
        bla wr
        ori 11, 11, 0

.BB4_3: # return
        addi 13, 13, 24
        lwz 24, -20 (13)
        mtlr 24
        blr
```

Figura 6.7: Comparação entre os códigos emitidos por cada *backend* para a função `print()` do código C apresentado na Figura 6.6.

### Seleção de Instruções Gerada para Cada Arquitetura

Um exemplo do que ocorre na fase de seleção de instruções do *backend* LLVM (o leitor pode consultar o Capítulo 3 para mais informações sobre as diferentes fases do *backend* LLVM) para cada arquitetura é ilustrado nas Figuras 6.8 e 6.9. Na primeira, um *Directed Acyclic Graph* (DAG), ou grafo direcionado acíclico, é utilizado para representar a entrada sobre a qual o algoritmo de seleção de instruções irá atuar. Neste exemplo, o DAG correspondente ao primeiro bloco básico da função `print` do código da Figura 6.6 é apresentado. Este DAG contém nós que correspondem a operações da linguagem intermediária LLVM.

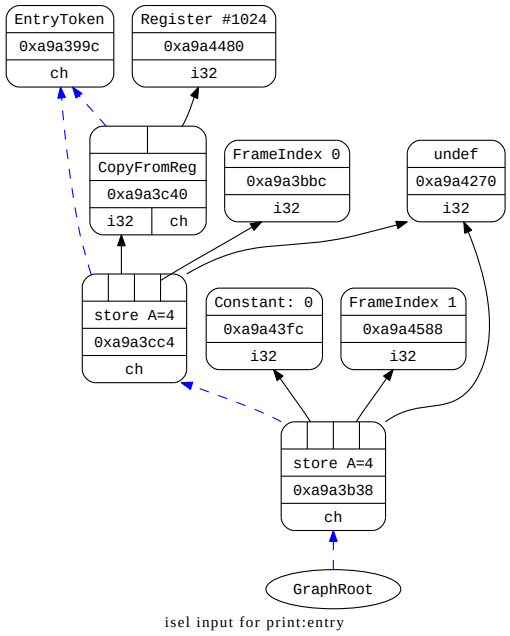


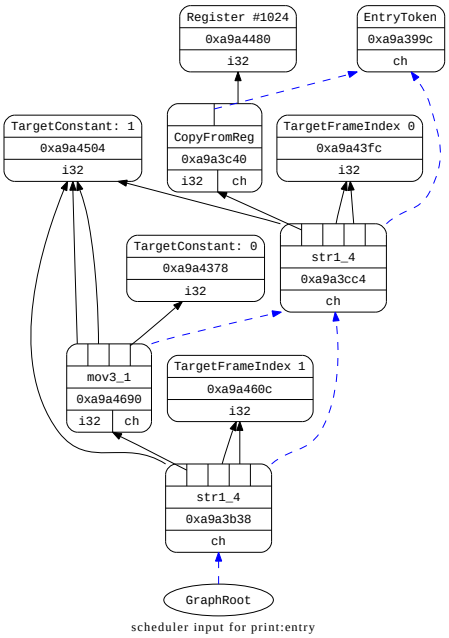

Figura 6.8: DAG utilizado como entrada para algoritmo de seleção de instruções dos *backends* LLVM gerados. Este DAG representa o primeiro bloco básico da função `print` do código da Figura 6.6.

Nesta etapa do processo de compilação, a linguagem intermediária da infraestrutura LLVM, que é representada como uma lista de instruções, já foi transformada em um DAG e todas as etapas de legalização já foram aplicadas. Ou seja, nesta etapa, os nós do

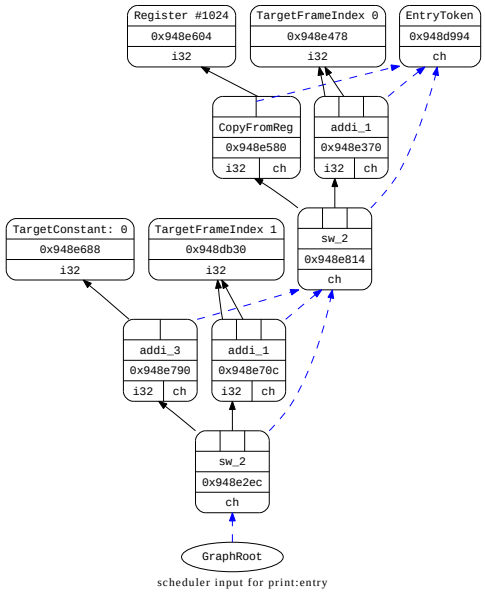
DAG contêm apenas operações que são suportadas pelo *backend*. As arestas expressam o fluxo de dados (relação de definição e uso de valores), de modo que os operandos de uma determinada operação precisam ser impressos antes do que a operação propriamente dita. Para facilitar a visualização de como um DAG é traduzido em instruções em linguagem de montagem, a raiz do grafo, ou seja, o nó fonte, que possui grau de entrada igual a zero, aparece por último. Os nós que possuem grau de saída igual a zero, os operandos terminais, aparecem primeiro na figura, pois devem ser os primeiros a serem calculados antes de serem utilizados por operações subsequentes. A ordem exata com que cada operando é calculado, nesta etapa, ainda é indefinida. O componente escalonador de instruções é quem recebe um DAG como entrada, precisamente o resultado produzido pela etapa de seleção de instruções, e irá transformá-lo em uma lista ordenada de operações, tomando decisões importantes como, em uma dada operação, em que ordem com que os subgrafos que representam seus operandos são avaliados não altera o resultado final, qual subgrafo de expressão deve ser avaliado primeiro. Apesar dessa ordem não alterar o resultado final, ela pode aumentar ou diminuir a pressão sobre o banco de registradores do processador, pois determina o número de registradores necessários para avaliar a expressão completa. Finalmente, após o escalonamento de instruções, o alocador de registrador atua.

Alguns nós precisam ser ordenados não só quanto à dependência do fluxo de dados, mas também quanto à execução de operações sensíveis a ordem, como operações de acesso à memória. Para isso, as arestas tracejadas indicam dependência de ordem de execução, na qual o vértice de saída da aresta contém uma operação que deve ser executada antes daquela do vértice entrada. A Figura 6.9 apresenta o grafo que é resultado do algoritmo de seleção de instruções para cada arquitetura. Perceba que cada *backend* irá transformar os nós do grafo de forma a inserir operações que correspondem a instruções de sua arquitetura.

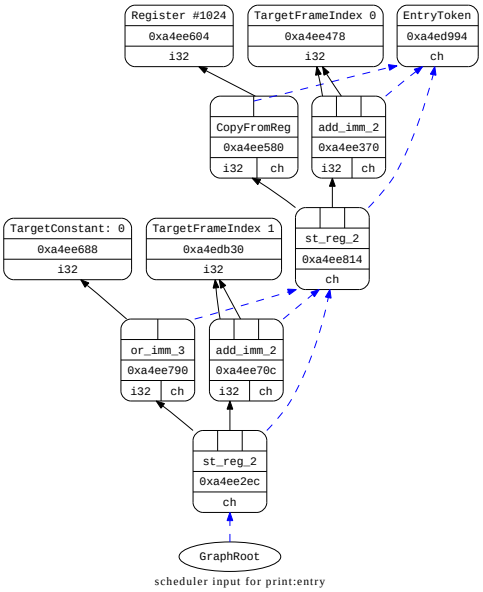
Os resultados finais do processo de compilação para os grafos apresentados na Figura 6.9 aparecem no primeiro bloco básico da função `print`, em linguagem de montagem, exibida na Figura 6.7. Repare que, por exemplo, o primeiro bloco básico da função em linguagem de montagem ARM possui 3 operações `store`, mas no DAG da Figura 6.9 aparecem apenas 2 nós que representam operações `str1_4` (armazenamento em memória com endereçamento base mais deslocamento na arquitetura ARM). Isso acontece porque, após o escalonador de instruções determinar a ordem a partir do DAG de expressão, o prólogo e epílogo da função é emitido. Como este é o primeiro bloco básico, ele possui a tarefa de hospedar o código do prólogo em seu início. O código do prólogo, por si só, pode conter operações de `store` adicionais. No caso ARM, as duas instruções iniciais correspondem ao prólogo, ao passo que as três instruções subsequentes correspondem a operações apresentadas em seu DAG resultado da seleção de instruções ARM. Os demais nós do DAG não aparecem como instruções de máquina, mas sim como operandos.



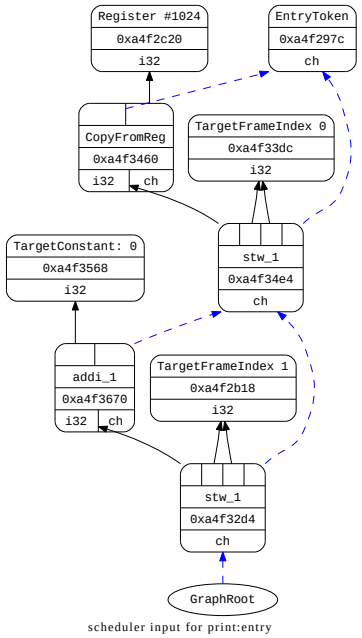

(a) ARM




(b) MIPS




(c) SPARC




(d) PowerPC

Figura 6.9: Os respectivos DAGs de resultado após a fase de seleção de instruções no DAG da Figura 6.8 para cada *backend*, ARM, MIPS, SPARC e PowerPC.

A regra exata de mapeamento de um nó, ou conjunto de nós, do DAG de instruções LLVM da Figura 6.8 em nós que representam instruções de uma arquitetura alvo, como na Figura 6.9, é determinada pelo algoritmo de busca. Para cada padrão LLVM, a busca determina uma lista de instruções da máquina alvo que irá implementar o determinado padrão e, então, traduz essa lista para um fragmento de DAG, que será utilizado no *backend* LLVM para transformar o DAG de expressão em instruções da máquina alvo, como aqueles apresentados na Figura 6.9.

Apesar da validação para MIPS, SPARC e PowerPC não ter contemplado programas do Mibench, devido a restrições de uso da biblioteca C, isto não significa que o *backend* gerado seja incompleto. O programa prova de conceito `acllvmbe` para geração de compiladores não produz *backends* incompletos, mas só conclui com sucesso quando o algoritmo de busca encontrou uma implementação válida em linguagem de montagem alvo para cada um dos 57 padrões que cobrem a linguagem intermediária LLVM. Como exemplo, considere a implementação do padrão ROR, definido no arquivo cabeçalho de regras pela seguinte construção:

```
define pattern ROR as (
 "(rotr i32:$a1, i32:$a2)";
 (transfer a3:regs (ror a1:regs a2:regs));
);
```

Nesta construção, é estabelecido que o fragmento de linguagem intermediária LLVM definido pelo nó `rotr` com operandos `a1` e `a2` corresponde ao fragmento de árvore semântica `(transfer a3 (ror a1 a2))`. Neste caso, a operação descrita aqui rotaciona à direita um valor qualquer definido pela subexpressão `a1` em `a2` casas. A Tabela 6.9 contém o resultado do algoritmo de busca deste padrão para todas as arquiteturas testadas.

Neste exemplo, fica evidente a versatilidade do algoritmo de busca por transformações em árvore. Para a arquitetura ARM, que possui uma única instrução que é capaz de rotacionar, não foi necessário aplicar nenhuma regra de transformação para concluir que uma variante da instrução `mov` deveria ser empregada. Já na arquitetura MIPS foi necessário construir uma sequência de 5 instruções que implementam a rotação à direita utilizando apenas as instruções MIPS existentes, que oferecem deslocamento sem rotação. Nesta lista, as letras de `A` a `D` representam registradores virtuais que devem ser substituídos por registradores reais. Os operandos `a1`, `a2` e `a3` fazem correspondência direta àqueles de mesmo nome descritos no padrão `ROR`, ou seja, `a2` é o número de casas a se deslocar, `a1` é o valor que sofrerá o deslocamento e `a3` é um registrador que irá guardar o resultado.

Note que foram necessárias 5 regras de transformação para provar que a árvore semântica do padrão `ROR` é equivalente à lista de instruções MIPS. A primeira regra utilizada, e mais importante, instrui o sistema a expandir um nó do tipo `ror` na subárvore completa que

implementa a rotação com deslocamentos. As 4 demais regras apenas decompõem a árvore para que seja possível implementar o padrão com 5 instruções distintas. Por exemplo, no caso da arquitetura PowerPC, apenas 3 decomposições são realizadas, e a lista contém 4 instruções. Isso aconteceu porque, no caminho de busca, foi encontrado uma única instrução PowerPC capaz de realizar a tarefa de duas instruções MIPS, a saber, carga da constante 32 e subtração dessa constante por um registrador. Finalmente, o caso SPARC é exatamente igual ao caso MIPS, porém utilizando instruções SPARC.

| **Máquina** | **Implementação** | **Número de regras aplicadas** |
|---|---|---|
| ARM | `mov a3, a1, ROR a2` | 0 |
| MIPS | `addi A, $0, 32`<br>`sub B, A, a2`<br>`sllv C, a1, B`<br>`srlv D, a1, a2`<br>`instr_or a3, C, D` | 5 |
| SPARC | `or %g0, 32, %A`<br>`sub %A, %a2, %B`<br>`sll %a1, %B, %C`<br>`srl %a1, %a2, %D`<br>`or %C, %D, %a3` | 5 |
| PowerPC | `subfic A, a2, 32`<br>`slw B, a1, A`<br>`srw C, a1, a2`<br>`ore a3, B, C` | 4 |

Tabela 6.9: Comparação entre a implementação encontrada para o padrão `ROR`, que cobre o nó `Rotr` da linguagem intermediária LLVM, para diferentes arquiteturas.

É importante atentar ao fato de que esta lista de instruções encontrada pelo algo-

ritmo de busca utiliza, muitas vezes, registradores virtuais para armazenar valores intermediários, que devem ser utilizados por alguma instrução subsequente da lista. Isso não representa um problema para a seleção de instruções. Essa lista será transformada em um DAG, e os valores intermediários serão codificados como arestas de dependência de dados para que, futuramente, o alocador de registradores determine qual registrador da máquina irá armazenar este valor intermediário. Todavia, pode ser necessário gerar código em etapas tardias da emissão de código. Por exemplo, no momento em que o alocador de registradores determinar ser necessário realizar o *spill* de um registrador, isto é, armazená-lo na pilha, faz-se necessário emitir código não na forma de DAG, mas na forma de lista, que realiza o armazenamento e carga deste valor na pilha. Este código de manipulação da pilha também é inferido pelo algoritmo de busca e, caso haja uso de registradores virtuais no próprio código de *spill*, não é mais possível contar com o alocador de registradores do *backend*. Para resolver este problema, `acllvmbe` utiliza um conjunto de registradores reservados para uso próprio, chamado de *registradores auxiliares* e definidos pelo usuário. Este conjunto é utilizado junto a um algoritmo simples de alocação de registradores para atribuir registradores reais ao código descoberto pelo algoritmo de busca. O número de registradores auxiliares necessário para gerar o *backend* com sucesso depende da arquitetura e das sequências de instruções geradas pelos padrões críticos, que são necessários em etapas tardias da emissão de código do *backend*.

## 6.2 Análises de Desempenho do Algoritmo de Busca

Esta seção é dedicada a analisar o desempenho do algoritmo de busca utilizado pelo sistema `acllvmbe`. Todos os testes dessa seção foram executados em uma máquina Intel Core 2 Quad Q6600 2.4GHz com 32KB de *cache* L1 de dados, 4MB de *cache* L2 de dados e 4GB de memória RAM. Apesar de que uma parte razoável do esforço de engenharia do projeto de `acllvmbe` tenha sido dedicada nas atividades de manipulação dos resultados da busca para transformá-lo em código C++ que utiliza a API LLVM, o algoritmo de busca ainda é o mecanismo central que viabiliza todo o projeto. O algoritmo de busca pode, ainda, ser utilizado no contexto de outros projetos, como por exemplo, em etapas de tradução binária estática entre diferentes conjuntos de instrução, ou em projetos de programação automática em geral, como aconteceu com a geração automática de testes para modelos de potência de processadores baseado em `acllvmbe`.

É essencial, portanto, o estudo do desempenho do algoritmo de busca. Na verdade, uma implementação ineficiente dos conceitos apresentados aqui pode ser a diferença entre o fracasso, em que o tempo para atingir a solução esperada ultrapassa o limite do factível para um ser humano esperar, e o sucesso, em que uma solução do problema de programação automática com a implementação de 57 padrões LLVM é rapidamente

alcançada em apenas 14 segundos.

O algoritmo de busca é discutido na Seção 5.2.3 e sua implementação apresentada na Seção 5.4.2. Esta metodologia é caracterizada na literatura como um **sistema de reescrita de árvore**. Para alcançar o objetivo de provar que uma árvore semântica (a expressão do padrão LLVM a se implementar na arquitetura alvo) equivale a outra (uma instrução alvo que se propõe a implementar este padrão), um conjunto de regras de transformações é aplicado para efetivamente reescrever a expressão original até que esta seja estruturalmente idêntica ao objetivo.

Quando em execução, os principais fatores que determinam o tempo de execução da busca são o número de regras disponíveis para se aplicar (quanto mais regras, maior o tempo de execução) e o número de instruções descritas no modelo do processador alvo. Sendo assim, o projetista deve, se possível, ajustar o número de regras e de instruções a um conjunto estritamente necessário. Note que a descrição de instruções complexas, que realizam, muitas vezes, mais do que uma operação simples por vez, pode produzir o efeito contrário do esperado e acelerar o mecanismo de busca para um fragmento em particular. Isso acontece porque, para provar que um fragmento grande equivale a uma lista de pequenas instruções, é necessária a aplicação de muito mais regras de transformação do que para provar a equivalência com uma única instrução que já possui a implementação de todo o fragmento em sua descrição. A decomposição da expressão, utilizada para encaixar 2 ou mais instruções em um único fragmento, irá gerar uma chamada recursiva do algoritmo de busca para os dois (ou mais) subproblemas criados e, portanto, pode multiplicar os tempos de busca pelo número de subproblemas criados.

### 6.2.1 Tamanho da *Cache* de Transformações

Independentemente do conjunto de regras e instruções que o usuário escolha, a implementação do algoritmo pode afetar severamente os tempos de busca. No intuito de reduzir os tempos de busca, o principal mecanismo implementado foi a *cache* de transformações. Dadas duas árvores de expressão `A` e `B`, a primeira representando o padrão a se implementar e a segunda representando a semântica de uma instrução da máquina alvo candidata a implementar este padrão, deseja-se provar, reescrevendo `A`, que `A` equivale a `B`. Se isto for verdade, encontra-se uma implementação para o padrão. A *cache* de transformações armazena pares `A` e `B` para os quais já foi descoberto não ser possível provar que `A` equivale a `B`, de forma a desistir prematuramente dessa busca em particular e assim podando a árvore de busca. Dessa maneira, economiza-se um grande tempo que seria gasto em um caminho infrutífero da busca. Desligar a *cache* faz com que o tempo total de busca para um único padrão salte da ordem de segundos para horas.

O motivo do grande ganho de desempenho com o uso da *cache* explica-se pelo fato de

que a natureza recursiva do algoritmo de busca tente resolver inúmeras vezes o mesmo subproblema. Ocorre, então, que um subproblema que já foi determinado como impossível de se provar a equivalência em uma determinada chamada recursiva seja *esquecido* após o término do escopo do resultado da função, de modo que uma nova chamada recursiva, em outro contexto, recalcule o mesmo subproblema. Para evitar isso, a *cache* armazena resultados infrutíferos anteriormente avaliados.

O algoritmo com o uso da *cache*, então, é muito sensível ao desempenho da implementação desta, pois, conforme o tempo passa e diversas instâncias do subproblema são marcadas como infrutíferas, o programa passa a mais tempo consultando a *cache* do que efetuando a busca propriamente dita. Para reduzir esse tempo, a *cache* é implementada como uma *hash table* resolvendo conflitos por encadeamento em lista. Cada entrada na tabela armazena uma dupla `A`, `B` cuja transformação de `A` em `B` é reconhecidamente impossível.

Determinou-se empiricamente que o tamanho da *cache* de transformações deve ser relativamente reduzido, se comparado com o número de transformações armazenadas. A Tabela 6.10 apresenta o tempo total de execução do algoritmo de busca para encontrar a implementação completa de todos os padrões LLVM para o modelo MIPS, o comportamento do tempo total conforme o tamanho da *cache* de transformações é alterado. O número total de transformações armazenadas na *cache*, no término da busca por todos os padrões, é de 164513. Ou seja, para uma *cache* de tamanho 256, na média, o número de itens na lista encadeada para uma entrada da *cache* é 642.

| **# entradas** | **128** | **256** | **512** | **1024** |
|---|---|---|---|---|
| Tempo | 29,56±0,14s | 24,71±0,13s | 54,73±0,05 | 67,35±0,26s |
| Requisições L2 | 109.292 | 89.165 | 201.613 | 275.783 |

Tabela 6.10: Comparação do desempenho do algoritmo de busca para a implementação dos padrões LLVM com instruções do modelo MIPS, utilizando diferentes tamanhos (número de entradas) para a *cache* de transformações.

Se a *cache* for esvaziada a cada nova busca de padrão, o número de itens pode variar de 100 a 88000 no caso MIPS. Entretanto, neste caso, o conhecimento gerado por buscas anteriores não está disponível para novas buscas. A tabela também mostra o número de requisições para a *cache* de dados L2 do processador para explicar o motivo da perda de desempenho com o aumento da capacidade da *hash table*. Repare que com 128 entradas a busca fica mais devagar do que com 256, devido ao grande número de colisões ocasionado pelo uso de uma *hash table* menor. Contudo, a tabela deixa claro que aumentar a capacidade da *hash table* não significa melhorar o desempenho do algoritmo de busca.

Isto acontece porque com o aumento da *hash table*, a estrutura de dados passa a não caber mais na *cache* L1 do processador, aumentando muito o número de requisições para a *cache* L2. Note que o número de requisições L2 para dados mais do que dobra quando o tamanho da *hash table* passa de 256 para 512 entradas.

Dessa maneira, é importante manter a *cache* de transformações pequena, pois como o laço de busca a utiliza com bastante frequência, cria-se uma dependência forte com o tamanho da *cache* L1 de dados do processador que está executando o algoritmo. Por esse motivo, é, na verdade, desvantajoso popular a *cache* de transformações com informações de buscas passadas. Apesar de que possa ter conhecimento útil sobre buscas infrutíferas, o custo de manter essa informação armazenada nas listas encadeadas supera a vantagem obtida por utilizar essas informações. O ato de reiniciar a *cache* a cada nova busca, no caso MIPS com *hash table* de capacidade 256 faz com que o tempo total da busca para todos os padrões LLVM reduza de 24,71±0,13s para 14,64±0,14s.

O uso de *hash tables* com capacidades que não são potências de 2 foi desconsiderado, uma vez que pioram consideravelmente o desempenho da busca pela introdução de instruções de divisão custosas no lugar de instruções de deslocamento, no momento de transformar o valor da função de *hash* em uma posição da tabela.

### 6.2.2 Paralelização do Algoritmo de Busca

O modo de operação da geração de *backend* consiste em sucessivamente buscar implementações para cada um dos 57 padrões LLVM pré-determinados. Para tirar proveito de estações de trabalho que possuam processadores CMP (*chip multicore*), isto é, arquitetura de memória compartilhada com múltiplos núcleos de execução, foi criada uma versão de *acllvmbe* capaz de despachar as tarefas de busca de cada padrão entre várias *threads* de execução. Este mecanismo foi implementado utilizando o padrão OpenMP. Assim, é possível especificar exatamente quantas *threads* serão utilizadas para realizar as tarefas de busca. O escalonamento de tarefas é dinâmico. Por exemplo, se houver 2 *threads*, não necessariamente haverá a divisão de tarefas em 29 e 28 padrões para cada. As *threads* de execução irão realizando a busca sob demanda. Caso haja um padrão cuja busca é suficientemente longa, pode ser que uma *thread* realize apenas a busca deste único padrão.

Um ponto importante é que não pode haver compartilhamento das informações da *cache* de transformações entre buscas anteriores, uma vez que cada *thread* possui sua própria versão da *cache*. É excessivamente custoso criar uma *cache* global para todas as *threads* e administrar a contenção. Contudo, como apresentado na subseção anterior, trabalhar com *caches* independentes e que sejam reiniciadas a cada nova busca é, na verdade, vantajoso. Desse modo, a paralelização da busca por padrões torna-se atraente.

A Tabela 6.11 mostra o tempo de execução para a geração de *backend* para as arquite-

turas ARM, MIPS, SPARC e PowerPC em diferentes casos. Os casos testados são com 4 *threads*, 2 *threads* e 1 *thread*, isto é, sem paralelização do trabalho. Para os casos em que há mais de uma *thread* de execução, dois tempos diferentes são mostrados. O primeiro é o tempo de relógio real, ou seja, a diferença entre o que um relógio comum marca no final da execução e no início. O segundo tempo é o tempo de usuário, que aponta quanto tempo foi alocado para que o processo em execução realizasse a tarefa, com exceção do tempo gasto no sistema operacional. Como há mais de um núcleo de processamento executando este código ao mesmo tempo, este tempo normalmente supera o tempo real, pois o tempo é contabilizado separadamente para cada núcleo e depois somado.

| **Modelo** | **4 threads** | **2 threads** | **1 thread** |
|---|---|---|---|
| ARM | 2'57"±16" (real)<br>5'26"±16" (user) | 3'22"±7" (real)<br>5'16"±8" (user) | 4'13"±2" |
| MIPS | 53"±2" (real)<br>1'32"±2" (user) | 57"±4" (real)<br>1'20"±4" (user) | 1'15"±0,2" |
| SPARC | 2'01"±8" (real)<br>3'04"±8" (user) | 2'06"±4" (real)<br>2'38"±4" (user) | 2'31"±1" |
| PowerPC | 14'55"±44" (real)<br>17'42"±41" (user) | 11'24"±2'43" (real)<br>13'04"±2'42" (user) | 11'41"±1" |

Tabela 6.11: Comparação do desempenho do algoritmo de busca com 4 *threads*, 2 *threads* e sequencial, utilizando tamanho de *cache* de transformações de 1024 entradas.

O gráfico da Figura 6.10 mostra as informações da tabela de modo a visualizar facilmente o modo como o algoritmo escala conforme o número de *threads* aumenta. Idealmente, supondo que a busca por cada padrão precise do mesmo tempo para concluir, o tempo real de uma execução com múltiplas *threads* deveria ser idêntico ao tempo real da execução sequencial dividida pelo número de *threads* para realizar o trabalho. Entretanto, há dois efeitos que impedem que o algoritmo escale perfeitamente com o número de *threads*. Em primeiro lugar, os tempos de busca para cada padrão variam muito. Muitos padrões são triviais, ou seja, possuem uma correspondência direta com uma instrução da arquitetura alvo e, portanto, sua busca é concluída muito rapidamente. Outros padrões possuem busca que consome muito tempo, pois serão implementados com uma longa lista de instruções da arquitetura alvo. Portanto, pode ocorrer um grande desbalanceamento de carga de trabalho entre as diferentes *threads*. O modelo que melhor ilustra isso é o PowerPC, em que o padrão `CONST32`, que realiza a carga de um imediato de 32 bits em um registrador, domina quase 50% do tempo total de busca.

Entretanto, mesmo com a carga de trabalho entre *threads* sendo desbalanceada, o

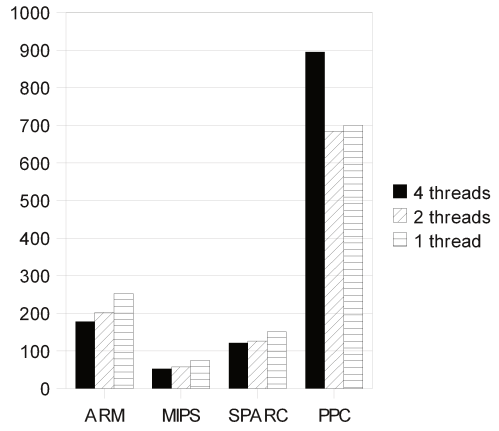



Figura 6.10: Comparação gráfica dos tempos de execução da busca por padrões com 4 *threads*, 2 *threads* e sequencial, utilizando tamanho de *cache* de transformações de 1024 entradas. O eixo Y apresenta o tempo total em segundos.

tempo de usuário das versões com 4 e 2 *threads* deveria se igualar com o tempo da versão sequencial, pois estamos apenas redistribuindo o trabalho sequencial em diferentes *threads*. O segundo efeito que faz com que o algoritmo paralelo desvie-se do ideal explica o aumento no tempo de usuário quando o número de *threads* aumenta. Esse efeito ocorre com a concorrência pela *cache* L2 que é compartilhada a cada par de núcleos na arquitetura Core 2 Quad. Para destacar esse fenômeno, a Tabela 6.11 apresenta valores para uma versão do algoritmo que implementa a *cache* de transformações com uma *hash table* com capacidade de 1024 entradas. Conforme analisado na subseção anterior, essa versão causa mais requisições à *cache* L2 de dados e é mais demorada. Por esse motivo, quando ocorre a paralelização, há maior contenção pelo acesso à *cache* de dados do processador. Em contraste, a Tabela 6.12 e seu respectivo gráfico na Figura 6.11 apresentam os mesmos dados, porém com a versão mais veloz em que a *hash table* possui capacidade de 256 entradas e realiza menos requisições à *cache* L2 de dados do processador. Além de ser mais veloz, essa versão escala melhor conforme o número de *threads* aumenta, devido à contenção pelos recursos compartilhados (*cache* e memória principal) ser reduzida, com exceção da arquitetura PowerPC, que possui problemas para escalar devido ao grande desbalanceamento. O desvio padrão para os tempos de execução costumam ser altos, pois o caráter indeterminístico com que as contenções ocorrem pode causar grandes variações no tempo de execução.

| **Modelo** | **4 threads** | **2 threads** | **1 thread** |
|---|---|---|---|
| ARM | 16±0,5” (real)<br>49±1,32” (user) | 18±0,30” (real)<br>36±0,6” (user) | 24±0,29” |
| MIPS | 7±0,48” (real)<br>15±0,57” (user) | 9±0,35” (real)<br>13±0,37” (user) | 15±0,2” |
| SPARC | 8±1,45” (real)<br>22±2,99” (user) | 9±1,04” (real)<br>19±2,09” (user) | 16±0,01” |
| PowerPC | 120±37” (real)<br>138±36” (user) | 151±19” (real)<br>161±19” (user) | 140±0,02” |

Tabela 6.12: Comparação do desempenho do algoritmo de busca com 4 *threads*, 2 *threads* e sequencial, utilizando tamanho de *cache* de transformações de 256 entradas.

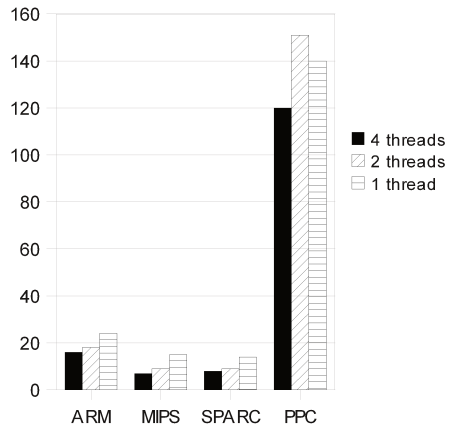



Figura 6.11: Comparação gráfica dos tempos de execução da busca por padrões com 4 *threads*, 2 *threads* e sequencial, utilizando tamanho de *cache* de transformações de 256 entradas. O eixo Y apresenta o tempo total em segundos.

# Capítulo 7

# Conclusão e Trabalhos Futuros

Este trabalho concentrou-se no estudo da melhor forma para a síntese automática de compiladores para a linguagem C que produzam código para plataformas descritas na linguagem de descrição de arquiteturas ArchC. A construção de compiladores é uma tarefa tediosa e difícil. Mesmo adotando como ponto de partida um projeto preexistente de compilador redirecionável, isto é, construído para que seja facilmente estendível para gerar código para novas arquiteturas, é necessário ter profundo conhecimento específico do domínio de compiladores, da linguagem intermediária do compilador utilizado e da arquitetura alvo para a qual o código será gerado. Para evitar que o modelo ArchC dependa da inclusão de informações específicas de um compilador para que seja possível redirecionar um compilador para a arquitetura em questão, adotou-se um mecanismo com maior grau de automação, em que o usuário descreve apenas informações declarativas sobre o conjunto de instruções. É de responsabilidade do sistema, então, buscar a melhor forma para criar o compilador. O elevado grau de automatização cria diferentes compromissos. A tarefa do usuário é bastante facilitada, ao preço da flexibilidade reduzida, uma vez que muitas decisões sobre o projeto do compilador são tomadas automaticamente.

Para viabilizar a síntese de um compilador para um modelo de arquitetura descrito em ArchC, foi utilizado o projeto do compilador LLVM, que é didático, escrito em C++ utilizando conceitos de orientação a objetos e criado para ser facilmente redirecionado. Por esse motivo, a síntese de compilador reduziu-se ao problema de geração do *backend* do compilador LLVM pois, utilizando a infraestrutura LLVM, o usuário dispõe de um compilador C completo para todas as arquiteturas que possuem um *backend* LLVM. Como a API LLVM dispõe de algoritmos básicos que todo *backend* utiliza, como alocador de registradores e um escalonador de instruções padrão, este trabalho focou-se na tarefa inicial, mas desafiadora, de geração de um seletor de instruções a partir de descrições das instruções da arquitetura alvo. Essas descrições não possuem informação sobre a linguagem intermediária LLVM e, portanto, esta conexão é realizada pela ferramenta desenvolvida de

geração automática. Neste contexto, foi estudado o problema da programação automática para criação de pequenos trechos de código em linguagem de montagem que realizam a computação equivalente a fragmentos da linguagem intermediária LLVM.

O problema da programação automática pode ser resolvido com um provador de teoremas simplificado, na qual a semântica do fragmento a se programar é a instância do problema. Axiomas ou regras de transformação que mantêm a equivalência entre duas árvores de expressão são então utilizados para manipular a instância original a fim de provar sua equivalência semântica com as instruções da máquina alvo. Do ponto de vista de implementação, esta solução foi implementada com um algoritmo de busca com uma abordagem da área de inteligência artificial. Nesse contexto, como o problema de programação automática e, mais amplamente, da geração de código, são problemas intratáveis, é importante que simplificações razoáveis e heurísticas eficientes sejam utilizadas para que um computador consiga, de fato, encontrar soluções aproximadas em tempo hábil.

Por ser crucial para a solução do problema da programação automática e, por conseguinte, da geração automática de *backends*, técnicas heurísticas foram propostas para acelerar o algoritmo de busca. A aplicação de apenas uma regra por nível da árvore de expressão, a limitação na profundidade da recursão, o uso de uma tabela de espalhamento para armazenar resultados anteriores da busca pelo qual se descobriu um caminho infrutífero da busca bem como a paralelização focada na execução de processadores de múltiplos núcleos foram técnicas utilizadas para este fim. Combinadas, conseguem fazer com que a programação automática de 57 padrões da linguagem intermediária da infraestrutura LLVM não somente seja possível, mas que seja alcançada em menos de 20 segundos para a maioria das arquiteturas modeladas com a descrição de instruções proposta.

Ainda que o problema da programação automática seja resolvido e uma implementação em instruções da máquina alvo seja encontrada para todos os padrões escolhidos para cobrir a linguagem intermediária da infraestrutura LLVM, é necessário converter tais resultados em código C++ que será utilizado para construir automaticamente todo o *backend* para a nova arquitetura na infraestrutura LLVM. Após a criação do *backend*, testes de compilação de programas do *benchmark* Mibench atestaram seu funcionamento correto e revelaram que a qualidade do código gerado pelo *backend* pode se comparar com a de código de compiladores consagrados, caso seja utilizado um otimizador *peephole* para realizar substituições simples de algumas sequências ineficientes.

Portanto, a geração automática do otimizador *peephole* é um trabalho futuro importante para o projeto e ainda pode contar com a infraestrutura de resolução do problema da programação automática já criada. Com esta infraestrutura, é possível utilizar um sistema de regras mais abrangente que, apesar de diminuir a velocidade de busca, irá se concentrar em achar trechos mais eficientes e que contribuam para a construção do otimizador

*peephole*. Não só a otimização de código é um ramo de pesquisa futuro que se abre com o uso desta infraestrutura, mas também a geração automática de tradutores de binário a partir de modelos descritivos das instruções dos conjuntos de instrução envolvidos. Dessa forma, tanto a geração automática de otimizadores para arquiteturas descritas em ArchC como a geração automática de tradutores binários são trabalhos futuros viabilizados com este projeto.

Outro trabalho futuro possível é a realização da extração de semântica através da tradução do código C++ SystemC que descreve o comportamento de uma instrução do modelo comportamental ArchC para a representação de semântica adotada. Apesar de ser tecnicamente desafiante realizar esta tradução, uma vez que diversas simplificações seriam necessárias para extrair uma árvore de expressão simples o suficiente para ser útil na geração de código, ainda seria possível gerar a tradução parcialmente e oferecê-la ao projetista como um ponto de partida para diminuir a quantidade de trabalho necessária para descrever o comportamento das instruções com árvores de expressão. Esta tarefa diminuiria ainda mais a quantidade de informações extras necessárias no modelo ArchC para a geração de *backend* de compilador.

# Referências Bibliográficas


[1] Boost C++ Libraries. http://www.boost.org/doc/.

[2] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23(3):488–501, 1976.

[3] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, 1989.

[4] Alfred V. Aho, Ravi Sethi, and J. D. Ullman. A formal approach to code optimization. In *Proceedings of a symposium on Compiler optimization*, pages 86–100, New York, NY, USA, 1970. ACM.

[5] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, December 1997.

[6] Guido Araujo and Sharad Malik. Optimal code generation for embedded memory non-homogeneous register architectures. In *ISSS '95: Proceedings of the 8th international symposium on System synthesis*, pages 36–41, New York, NY, USA, 1995. ACM.

[7] Rafael Auler, Alexandro Baldassin, and Paulo Centoducatte. Automatic Architecture Description Language (ADL)-based toolchain generation: the dynamic linking framework. In *SBLP '10: The 14th Brazilian Symposium on Programming Languages*, 2010.

[8] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33(5):453–484, October 2005.

[9] A. Baldassin, P. C. Centoducatte, and S. Rigo. Extending the ArchC language for automatic generation of assemblers. In *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing*, pages 60–68, October 2005.

[10] Alexandro Baldassin. Geração automática de montadores em ArchC. Master's thesis, Instituto de Computação, UNICAMP, Campinas, Abril 2005.

[11] Alexandro Baldassin, Paulo Centoducatte, Sandro Rigo, Daniel Casarotto, Luiz C. V. Santos, Max Schultz, and Olinto Furtado. An open-source binary utility generator. *ACM Trans. Des. Autom. Electron. Syst.*, 13(2):1–17, 2008.

[12] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. *SIGPLAN Not.*, 41:394–403, October 2006.

[13] Florian Brandner, Dietmar Ebner, and Andreas Krall. Compiler generation from structural architecture descriptions. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 13–22, New York, NY, USA, 2007. ACM.

[14] R. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Trans. Program. Lang. Syst.*, 2(2):173–190, 1980.

[15] Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. C compiler retargeting based on instruction semantics models. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1150–1155, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT press, 2007.

[17] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst.*, 6(4):505–526, 1984.

[18] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. *SIGPLAN Not.*, 39(4):104–111, 2004.

[19] Joao Dias and Norman Ramsey. Automatically generating instruction selectors using declarative machine descriptions. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 403–416, New York, NY, USA, 2010. ACM.

[20] Michael K. Donegan, Robert E. Noonan, and Stefan Feyock. A code generator generator language. In *SIGPLAN '79: Proceedings of the 1979 SIGPLAN symposium on Compiler construction*, pages 58–64, New York, NY, USA, 1979. ACM.

[21] Charles Donnelly and Richard Stallman. *Bison, the YACC-compatible parser generator*. Free Software Foundation, Inc., 1.35 edition, February 2002.

[22] H. Emmelmann, F.-W. Schröer, and Rudolf Landwehr. BEG: a generator for efficient back ends. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 227–237, New York, NY, USA, 1989. ACM.

[23] Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. Effective compiler generation by architecture description. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 145–152, New York, NY, USA, 2006. ACM.

[24] A. Fauth and A. Knoll. Automated generation of dsp program development tools using a machine description formalism. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 457–460 vol.1, Apr 1993.

[25] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings of the EDTC: The European Design and Test Conference*, pages 503–507. IEEE Computer Society, March 1995.

[26] C. W. Fraser and A. L. Wendt. Automatic generation of fast optimizing code generators. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 79–84, New York, NY, USA, 1988. ACM.

[27] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, 1992.

[28] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992.

[29] Steve Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Professional, 2000.

[30] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 231–254, New York, NY, USA, 1978. ACM.

[31] Samuel Shoji Fukujima Goto. Síntese de linguagens de descrição de arquitetura. Master's thesis, Instituto de Computação, UNICAMP, Campinas, Junho 2011.

[32] SPAM Research Group. *SPAM Compiler User's Manual*, 1997.

[33] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual IEEEWorkshop on Workload Characterization*, pages 3–14, December 2001.

[34] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: an instruction set description language for retargetability. In *Proceedings of the 34th Annual Conference on Design Automation*, pages 299–302. ACM Press, June 1997.

[35] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. *Design, Automation and Test in Europe Conference and Exhibition*, 0:485, 1999.

[36] Silvina Hanono and Srinivas Devadas. Instruction selection, resource allocation, and scheduling in the aviv retargetable code generator. In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 510–515, New York, NY, USA, 1998. ACM.

[37] Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture exploration for embedded processors with LISA*. Kluwer Academic Publishers, 2002.

[38] Manuel Hohenauer, Hanno Scharwaechter, Kingshuk Karuri, Oliver Wahlen, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Gunnar Braun, and Hans van Someren. 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*, page 21276, Washington, DC, USA, 2004. IEEE Computer Society.

[39] Roger Hoover and Kenneth Zadeck. Generating machine specific optimizing compilers. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 219–229, New York, NY, USA, 1996. ACM.

[40] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28:967–989, November 2006.

[41] Gerry Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[42] Peter B. Kessler. Discovering machine-specific code improvements. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 249–254, New York, NY, USA, 1986. ACM.

[43] David Ryan Koes and Seth Copen Goldstein. Near-optimal instruction selection on DAGs. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 45–54, New York, NY, USA, 2008. ACM.

[44] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G.Goossens. CHESS: Retargetable code generation for embedded DSP processors. In *Code generation for embedded processors*, chapter 5, pages 85–102. Kluwer Academic Publishers, 1995.

[45] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, pages 75+, Washington, DC, USA, 2004. IEEE Computer Society.

[46] Rainer Leupers and Peter Marwedel. Retargetable generation of code selectors from HDL processor models. In *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, page 140, Washington, DC, USA, 1997. IEEE Computer Society.

[47] C. Liem, T. May, and P. Paulin. Instruction-set matching and selection for DSP and ASIP code generation. In *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings.*, pages 31–37, Feb-3 Mar 1994.

[48] Clifford Liem, Pierre Paulin, Marco Cornero, and Ahmed Jerraya. Industrial experience using rule-driven retargetable code generation for multimedia applications. In *ISSS '95: Proceedings of the 8th international symposium on System synthesis*, pages 60–68, New York, NY, USA, 1995. ACM.

[49] Peter Marwedel. The MIMOLA design system: Tools for the design of digital processors. In *Proceedings of the 21st Design Automation Conference*, DAC '84, pages 587–593, Piscataway, NJ, USA, 1984. IEEE Press.

[50] Henry Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[51] Cathy May. *The PowerPC Architecture: A Specification for A New Family of RISC Processors*. Morgan Kaufmann Publishers, 1994.

[52] A. Nymeyer and J.P. Katoen. Code generation based on formal BURS theory and heuristic search. *Acta Informatica*, 34(8):597–635, August 1997.

[53] Vern Paxson. *Flex, a fast scanner generator*. Free Software Foundation, Inc., 2.5 edition, March 1995.

[54] E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: an application BURS theory. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 294–308, New York, NY, USA, 1988. ACM.

[55] Roland H. Pesch and Jeffrey M. Osier. *The GNU binary utilities*. Free Software Foundation, Inc., May 1993. version 2.15.

[56] Todd A. Proebsting. BURS automata generation. *ACM Trans. Program. Lang. Syst.*, 17(3):461–486, 1995.

[57] Wei Qin, Subramanian Rajagopalan, and Sharad Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 47–56, New York, NY, USA, 2004. ACM.

[58] Norman Ramsey and Jack W. Davidson. Machine descriptions to built tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, pages 172–188, 1998.

[59] Red Hat, Inc. *Red Hat newlib C library documentation*, 2008. version 1.17.

[60] René Seindal. *GNU M4 - GNU Macro Processor*. Free Software Foundation, Inc., 1.4.16 edition, March 2011.

[61] Ravi Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, 1970.

[62] SPARC International, Inc. *The SPARC architecture manual - Version 8*, 1992. Revision SAV080SI9308.

[63] Y. N. Srikant and Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2007.

[64] Richard Stallman. *Using the GNU compiler collection.* Free Software Foundation, Inc., May 2004. For GCC version 3.4.3.

[65] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM.

[66] S. Tijang. An Olive Twig. Technical report, Synopsis, Inc., 1993.

[67] B. Wess. Automatic instruction code generation based on trellis diagrams. In *Circuits and Systems, 1992. ISCAS '92. Proceedings., 1992 IEEE International Symposium on*, volume 2, pages 645–648 vol.2, May 1992.

# Apêndice A

# Árvore de Semântica

A árvore de semântica discutida na Seção 4.4.1 não possui um conjunto fixo de operadores. A descrição de um modelo pode criar seus próprios operadores com as diretivas apresentadas no Capítulo 4 desde que descreva também as regras de equivalência ensinando o sistema como utilizar este operador. Entretanto, existe um conjunto básico de operadores pré-definidos sobre os quais foram construídos os modelos do ARM, MIPS, SPARC e PowerPC e também constituiu a linguagem básica com a qual os padrões que cobrem subárvores da representação intermediária do LLVM foram especificados.

É importante ter em mente que quanto menos transformações forem usadas para partir de uma árvore que descreve um padrão da representação intermediária do LLVM e alcançar as árvores que descrevem as instruções de uma máquina, mais rápido o *backend* para essa máquina será gerado. Contudo, as instruções de uma máquina podem realizar tarefas de maneiras bem diferentes daquelas esperadas pelos padrões definidos para cobrir a representação intermediária do LLVM. Por esse motivo, o sistema de inferências construído é necessário e uma linguagem abrangente para descrever todos esses casos foi disponibilizada.

A Seção A.1 apresenta uma listagem completa de todos os tipos de operadores pré-definidos para os modelos testados neste trabalho. A Seção A.2 mostra as regras de transformação utilizadas pelo algoritmo de busca que manipula as expressões até que sejam encontradas as instruções da máquina alvo que implementam um determinado comportamento. Finalmente, a Seção A.3 lista todos os padrões da linguagem intermediária do LLVM que foram utilizados.

## A.1 Tipos de Operadores

| Nome | Tipo | Descrição |
|---|---|---|
| `+` | Binário | Produz a soma de seus dois operandos. |
| `-` | Binário | Subtrai o primeiro operando do segundo. |
| `*` | Binário | Multiplica um operando pelo outro. |
| `sdiv` | Binário | Obtém o quociente da divisão de inteiros sinalizados. O primeiro operando é o dividendo e o segundo é o divisor. |
| `udiv` | Binário | Obtém o quociente da divisão de inteiros não sinalizados. O primeiro operando é o dividendo e o segundo é o divisor. |
| `srem` | Binário | Obtém o resto da divisão de inteiros sinalizados. O primeiro operando é o dividendo e o segundo é o divisor. |
| `urem` | Binário | Obtém o resto da divisão de inteiros não sinalizados. O primeiro operando é o dividendo e o segundo é o divisor. |
| `~` | Unário | Negação bit a bit de seu operando. |
| `mulhs` | Binário | Multiplicação sinalizada de dois operandos de tamanho $N$ bits que produz um resultado de tamanho $2N$ bits, retornando a parte alta. |
| `mulhs` | Binário | Multiplicação não sinalizada de dois operandos de tamanho $N$ bits que produz um resultado de tamanho $2N$ bits, retornando a parte alta. |
| `xor` | Binário | OU exclusivo entre os seus dois operandos. |
| `transfer` | Binário | Copia o valor do filho direito para o filho esquerdo. |
| `zext` | Binário | Estende com zeros o tamanho do primeiro operando até que ele possua o tamanho, em bits, indicado pelo segundo operando. |
| `sext` | Binário | Estende com zeros ou uns (dependendo do sinal do operando) o tamanho do primeiro operando até que ele possua o tamanho, em bits, indicado pelo segundo operando. |
| `trunc` | Binário | Trunca o valor do primeiro operando, reduzindo seu tamanho, até que ele possua o tamanho, em bits, indicado pelo segundo operando. |
| `dec` | Binário | Metaoperador especial utilizado para representar uma regra de decomposição, indicando que a árvore casada pela regra pode ser igualmente representada pelo conjunto de árvores correspondentes aos operandos deste operador. |

| **Nome** | **Tipo** | **Descrição** |
|---|---|---|
| `shl` | Binário | Deslocamento à esquerda do primeiro operando pelo número de bits indicado pelo segundo operando. |
| `shr` | Binário | Deslocamento à direita do primeiro operando pelo número de bits indicado pelo segundo operando. |
| `asr` | Binário | Deslocamento aritmético à direita, isto é, preservando o sinal do primeiro operando. O deslocamento é realizado pelo número de bits representado pelo segundo operando. |
| `rol` | Binário | Rotaciona à esquerda o primeiro operando pelo número de bits indicado pelo segundo operando. |
| `ror` | Binário | Rotaciona à direita o primeiro operando pelo número de bits indicado pelo segundo operando. |
| `and` | Binário | E lógico bit a bit entre os dois operandos. |
| `or` | Binário | OU lógico bit a bit entre os dois operandos. |
| `memref` | Unário | Referência a uma posição de memória. O valor da posição de memória a ser acessada é indicada pelo primeiro operando. |
| `call` | Unário | Salta para um endereço, salvando o endereço da próxima instrução para retorno. |
| `ret` | Folha | Salta para o endereço de retorno. |
| `jump` | Unário | Salta para o endereço indicado pelo primeiro operando. |
| `cjump` | Ternário | Salto condicional para o endereço indicado pelo terceiro operando. O primeiro operando contém o tipo de comparação a se efetuar com o segundo operando, que normalmente contém um operador de comparação. |
| `comp` | Binário | Compara o primeiro operando com o segundo e produz um resultado que deve ser julgado com a ajuda de `cjump`. Em conjunto com ele, este operador modela o salto condicional de arquiteturas que possuem registrador de estado. O resultado deste operador representa, em geral, mudança no registrador de estado que é interpretada por outra instrução. A arquitetura ARM é um exemplo. |
| `jumpnz` | Binário | Este operador representa saltos condicionais para o segundo operando se, e apenas se, o primeiro operador é diferente de zero. Este operador modela saltos condicionais em arquiteturas sem registrador de estado, como por exemplo a arquitetura MIPS. |

| Nome | Tipo | Descrição |
|---|---|---|
| `setif` | Ternário | Compara o segundo com o terceiro operando e retorna 1 se a condição indicada pelo primeiro operando é obedecida. Usado em conjunto com `jumpnz`. |

## A.2 Regras de Transformação

```
1. AnyOperator:any => (dec (transfer AReg:regs^ AnyOperator:any)
     AReg:regs^);
```

A primeira regra, já apresentada no Capítulo 4, é a regra geral de inferência de decomposição de árvore. Ela expressa a informação de que qualquer nó da árvore semântica (uso do nó folha `any` para descrever este caso) pode ser transformado em um registrador (segundo operando do operador `dec`), desde que em outra árvore semântica (primeiro operando do operador `dec`) ocorra a movimentação do valor deste nó para o registrador em questão. Essa regra simplifica árvores de expressão grandes partindo-a em duas árvores diferentes que são ligadas pelo uso de um registrador em comum. O operador `dec` é especial e, quando é encontrado pelo sistema de aplicação de regras, é automaticamente excluído e a árvore é decomposta em duas.

```
2. (transfer (memref AnyOperator1:any) (memref AnyOperator2:any))
     => (dec (transfer AReg:regs (memref AnyOperator2:any))
             (transfer (memref AnyOperator1:any) AReg:regs));
```

A regra 2 é similar à regra geral de inferência de decomposição de árvore. Essa regra expressa como a movimentação de dados de memória para memória pode ser decomposta em `load` seguido por `store`.

```
3. (transfer DestReg:regs AnyOperator:any) =>
     (dec (transfer AReg:regs^ AnyOperator:any)
          (transfer DestReg:regs AReg:regs^) );
```

A regra 3 também é similar à regra geral de inferência de decomposição de árvore, de número 1. Essa regra expressa como decompor uma movimentação de um valor qualquer para um registrador. Ela é usada quando a movimentação simples de um valor para um registrador não casa porque o registrador de destino da instrução é um registrador especial, e não de propósito geral. A regra, portanto, introduz um novo registrador temporário para receber o valor.

```
4. (cjump CondOp:any (comp lhs:any rhs:any) tgt:any) =>
     (jumpnz (setif CondOp:any lhs:any rhs:any) tgt:any);
```

A regra 4 indica como um operador `cjump` (salto condicional para arquiteturas com registrador de estado) pode ser transformado em `jumpnz` (salto condicional para arquiteturas sem registrador de estado).

```
5. (cjump const:cond:gt (comp lhs:regs rhs:regs) imm:tgt:int) =>
     (jumpnz (setif const:cond:lt rhs:regs (+ lhs:regs
       const:byte:1)) imm:tgt:int);

6. (cjump const:cond:ge (comp lhs:regs rhs:regs) imm:tgt:int) =>
     (jumpnz (setif const:cond:lt rhs:regs lhs:regs) imm:tgt:int);

7. (cjump const:cond:le (comp lhs:regs rhs:regs) imm:tgt:int) =>
     (jumpnz (setif const:cond:lt lhs:regs (+ rhs:regs
       const:byte:1)) imm:tgt:int);

8. (cjump const:cond:ugt (comp lhs:regs rhs:regs) imm:tgt:int) =>
     (jumpnz (setif const:cond:ult rhs:regs (+ lhs:regs
       const:byte:1)) imm:tgt:int);

9. (cjump const:cond:ule (comp lhs:regs rhs:regs) imm:tgt:int) =>
     (jumpnz (setif const:cond:ult lhs:regs (+ rhs:regs
       const:byte:1)) imm:tgt:int);

10. (cjump const:cond:uge (comp lhs:regs rhs:regs) imm:tgt:int) =>
      (jumpnz (setif const:cond:ult rhs:regs lhs:regs) imm:tgt:int);

11. (cjump const:cond:uge (comp lhs:regs rhs:regs) imm:tgt:int) =>
      (cjump const:cond:ule (comp rhs:regs (- lhs:regs const:byte:1))
         imm:tgt:int);

12. (cjump const:cond:ult (comp lhs:regs rhs:regs) imm:tgt:int) =>
      (cjump const:cond:ule (comp lhs:regs (- rhs:regs const:byte:1))
        imm:tgt:int);
```

A regra 5 indica como um salto condicional que testa se um registrador é maior do que outro pode ser transformado em um salto condicional que testa se um registrador é menor do que outro, invertendo os operandos da comparação e somando 1. As demais regras postulam transformações sobre como saltos condicionais podem ser manipulados. No geral, o conhecimento dessas regras é importante para que todos os testes condicionais

sejam implementáveis até em arquiteturas que não dispõem de todas as comparações possíveis no LLVM, como a arquitetura MIPS.

```
13. (rol child1:any child2:any) =>
  (or (shl child1:any child2:any) (shr child1:any (- const:byte:32
    child2:any)));

14. (ror child1:any child2:any) =>
  (or (shr child1:any child2:any) (shl child1:any (- const:byte:32
     child2:any)));
```

As regras 13 e 14 expandem nós de rotação para a esquerda e para a direita em subárvores que realizem esta mesma tarefa utilizando operadores de deslocamento e OU lógico. Essas regras são necessárias para que a rotação (requisitada pela linguagem intermediária do compilador LLVM) seja implementável em qualquer arquitetura, mesmo que não exista instruções que realizem rotação. Note que, contudo, em uma arquitetura que não possua nem instruções de deslocamento, a geração do *backend* é impossível.

```
15. imm:oper:int => (dec (transfer AReg:regs (+ (shl imm:op1:short
      const:byte:16) imm:op2:short)) AReg:regs)
      ( oper binds to op1 using "oper >> 16";
        oper binds to op2 using "oper & 0xFFFF";
      );

16. imm:oper:short => (dec (transfer AReg:regs (+ (shl imm:op1:byte
      const:byte:8) imm:op2:byte)) AReg:regs)
      ( oper binds to op1 using "oper >> 8";
        oper binds to op2 using "oper & 0xFF";
      );
```

A regra 15 diz como um imediato de 32 bits pode ser quebrado em dois imediatos de 16 bits, ao passo que a regra 16 diz como um imediato de 16 bits pode ser quebrado em dois imediatos de 8 bits. Essas regras são importantes para que imediatos grandes possam caber em arquiteturas cujas instruções não dedicam muitos bits para carregar o imediato.

```
17. (pcrelative aa:any) => (+ (getpc) aa:any);
```

A última regra, de número 17, postula que o operador `pcrelative` e seu operando é equivalente à soma do registrador *program counter* (obtido com `getpc`) a este operando.

## A.3 Padrões da Linguagem Intermediária do LLVM

Esta seção contém uma listagem do conjunto de padrões pré-definidos que cobrem a linguagem intermediária do LLVM. Todos os *backends* gerados com o projeto `acllvmbe` utilizam o mesmo conjunto de padrões, sendo que a diferença entre eles é como esses padrões são implementados. A implementação dos padrões utilizando instruções de máquina é inferida através da árvore semântica correspondente ao padrão, aplicando as regras de transformação apresentadas na seção anterior, se necessárias, até que uma instrução, ou uma lista de instruções, tenha árvores semânticas que casem com o comportamento solicitado pela implementação do padrão. Já á arvore descrita em *RI LLVM* (Representação Intermediária do LLVM) utiliza a notação das operações LLVM e descreve qual trecho da linguagem intermediária do LLVM é coberto por este padrão. Os padrões são apresentados na ordem em que são casados contra a linguagem intermediária do LLVM no momento de geração de código.

| **Nome:** | STORETRUNC8 |
|---|---|
| **Descrição:** | Trunca um valor para 8 bits e o armazena em memória. |
| **RI LLVM:** | `(truncstorei8 i32:$src, i32:$addr)` |
| **Árvore:** | `(transfer (memref addr:regs) (trunc src:reg const:num:8))` |

| **Nome:** | STORETRUNC16 |
|---|---|
| **Descrição:** | Trunca um valor para 16 bits e o armazena em memória. |
| **RI LLVM:** | `(truncstorei16 i32:$src, i32:$addr)` |
| **Árvore:** | `(transfer (memref addr:regs) (trunc src:reg const:num:16))` |

| **Nome:** | STOREFICONST |
|---|---|
| **Descrição:** | Grava em uma posição do *frame* da função o valor de uma constante. |
| **RI LLVM:** | `(store (tgtimm i32:$src), (frameindex i32:$val))` |
| **Árvore:** | `(transfer (memref (+ val:regs imm:val:tgtimm)) imm:src:tgtimm)` |

| **Nome:** | STOREFI |
|---|---|
| **Descrição:** | Armazena um valor qualquer em uma posição do *frame* da função. Este padrão é importante por indicar como os registradores são *spilled* para a pilha (o *frame* da função atual). |
| **RI LLVM:** | `(store i32:$src, (frameindex i32:$val))` |
| **Árvore:** | `(transfer (memref (+ val:regs imm:val:tgtimm))`<br>`src:regs)` |

| **Nome:** | STOREADDCONST |
|---|---|
| **Descrição:** | Armazena um valor para uma posição de memória calculada com a soma de um registrador a uma constante. |
| **RI LLVM:** | `(store i32:$src, (add i32:$a1, (tgtimm i32:$a2)))` |
| **Árvore:** | `(transfer (memref (+ a1:regs`<br>`imm:a2:tgtimm)) src:regs)` |

| **Nome:** | STOREADD |
|---|---|
| **Descrição:** | Armazena um valor em uma posição de memória calculada com a soma de dois registradores. |
| **RI LLVM:** | `(store i32:$src, (add i32:$reg1, i32:$reg2))` |
| **Árvore:** | `(transfer (memref (+ reg1:regs reg2:regs))` |

| **Nome:** | STORE |
|---|---|
| **Descrição:** | Armazena um valor em uma posição de memória indicada por um registrador. |
| **RI LLVM:** | `(store i32:$src, i32:$addr)` |
| **Árvore:** | `(transfer (memref addr:regs) src:regs)` |

| **Nome:** | LOADSEXT8 |
|---|---|
| **Descrição:** | Carrega da memória um valor de 8 bits e faz extensão sinalizada para 32 bits. |
| **RI LLVM:** | `(loadsexti8 i32:$addr)` |
| **Árvore:** | `(transfer a3:regs (sext (memref addr:regs)`<br>`const:num:8))` |

| **Nome:** | LOADSEXT16 |
|---|---|
| **Descrição:** | Carrega da memória um valor de 16 bits e faz extensão sinalizada para 32 bits. |
| **RI LLVM:** | `(loadsexti16 i32:$addr)` |
| **Árvore:** | `(transfer a3:regs (sext (memref addr:regs)`<br>`const:num:16))` |

| **Nome:** | LOADZEXT1 |
|---|---|
| **Descrição:** | Carrega um valor de 1 bit da memória e o estende com zeros para 32 bits. |
| **RI LLVM:** | `(loadzexti1 i32:$addr)` |
| **Árvore:** | `(transfer a3:regs (zext (memref addr:regs)`<br>`const:num:8))` |

| **Nome:** | LOADZEXT8 |
|---|---|
| **Descrição:** | Carrega um valor de 8 bits da memória e o estende com zeros para 32 bits. |
| **RI LLVM:** | `(loadzexti8 i32:$addr)` |
| **Árvore:** | `(transfer a3:regs (zext (memref addr:regs)`<br>`const:num:8))` |

| **Nome:** | LOADZEXT16 |
|---|---|
| **Descrição:** | Carrega um valor de 16 bits da memória e o estende com zeros para 32 bits. |
| **RI LLVM:** | `(loadzexti16 i32:$addr)` |
| **Árvore:** | `(transfer a3:regs (zext (memref addr:regs)`<br>`const:num:16))` |

| **Nome:** | LOADFI |
|---|---|
| **Descrição:** | Carrega um valor do *stack frame* da função corrente. Este padrão determina como o *backend* carrega de volta valores que foram *spilled* para a pilha. |
| **RI LLVM:** | `(load (frameindex i32:$addr))` |
| **Árvore:** | `(transfer dest:regs (memref (+ addr:regs`<br>`imm:addr:tgtimm)))` |

| **Nome:** | LOADADDCONST |
|---|---|
| **Descrição:** | Carrega o valor da posição de memória calculada com a soma de um registrador a uma constante. |
| **RI LLVM:** | `(load (add i32:$a1, (tgtimm i32:$a2)))` |
| **Árvore:** | `(transfer dest:regs (memref (+ a1:regs imm:a2:tgtimm)))` |

| **Nome:** | LOADADD |
|---|---|
| **Descrição:** | Carrega o valor da posição de memória calculada com a soma de dois registradores. |
| **RI LLVM:** | `(load (add i32:$reg1, i32:$reg2))` |
| **Árvore:** | `(transfer dest:regs (memref (+ reg1:regs reg2:regs)))` |

| **Nome:** | LOAD |
|---|---|
| **Descrição:** | Carrega o valor da posição de memória indicada por um registrador. |
| **RI LLVM:** | `(load i32:$addr)` |
| **Árvore:** | `(transfer a3:regs (memref addr:regs))` |

| **Nome:** | GLOBALADDRESS |
|---|---|
| **Descrição:** | Obtém o endereço de um símbolo global. O *backend* gerado emite uma tabela com todos o símbolos globais referenciados na função ao final dela e, por isso, o comportamento é implementado com uma carga relativa ao PC. |
| **RI LLVM:** | `(globaladdr i32:$addr)` |
| **Árvore:** | `(transfer a3:regs (memref (pcrelative imm:addr:short)))` |

| **Nome:** | FRAMEINDEX |
|---|---|
| **Descrição:** | Casa com nós `frameindex` que não foram casados nos padrões anteriores. Traduz em uma carga de valor da pilha. |
| **RI LLVM:** | `(frameindex i32:$addr)` |
| **Árvore:** | `(transfer a3:regs (+ addr:regs imm:addr:tgtimm))` |

| **Nome:** | CALL |
|---|---|
| **Descrição:** | Chamada para uma função cujo endereço é representado por um símbolo global. |
| **RI LLVM:** | `(call (tglobaladdr i32:$tgt))` |
| **Árvore:** | `(call imm:tgt:int)` |

| **Nome:** | CALL2 |
|---|---|
| **Descrição:** | Chamada para uma função cujo endereço é representado por um símbolo externo. |
| **RI LLVM:** | `(call (texternalsymbol i32:$tgt))` |
| **Árvore:** | `(call imm:tgt:int)` |

| **Nome:** | CALL3 |
|---|---|
| **Descrição:** | Chamada para uma função cujo endereço está contido em um registrador (chamada indireta). |
| **RI LLVM:** | `(call i32:$tgt)` |
| **Árvore:** | `(call tgt:regs)` |

| **Nome:** | ADDCONST |
|---|---|
| **Descrição:** | Soma de um registrador e uma constante. |
| **RI LLVM:** | `(add i32:$a1, (tgtimm i32:$a2))` |
| **Árvore:** | `(transfer a3:regs (+ a1:regs imm:a2:tgtimm))` |

| **Nome:** | ADD |
|---|---|
| **Descrição:** | Soma de dois registradores. |
| **RI LLVM:** | `(add i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (+ a1:regs a2:regs))` |

| **Nome:** | SUBCONST |
|---|---|
| **Descrição:** | Subtração de um registrador por uma constante. |
| **RI LLVM:** | `(sub i32:$a1, (tgtimm i32:$a2))` |
| **Árvore:** | `(transfer a3:regs (- a1:regs imm:a2:tgtimm))` |

| **Nome:** | SUB |
|---|---|
| **Descrição:** | Subtração entre dois registradores. |
| **RI LLVM:** | `(sub i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (- a1:regs a2:regs))` |

| **Nome:** | MUL |
|---|---|
| **Descrição:** | Multiplicação entre dois registradores. |
| **RI LLVM:** | `(mul i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (* a1:regs a2:regs))` |

| **Nome:** | MULHS |
|---|---|
| **Descrição:** | Multiplicação sinalizada entre dois registradores obtendo a parte alta do resultado de 64 bits. |
| **RI LLVM:** | `(mulhs i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (mulhs a1:regs a2:regs))` |

| **Nome:** | MULHU |
|---|---|
| **Descrição:** | Multiplicação não sinalizada entre dois registradores obtendo a parte alta do resultado de 64 bits. |
| **RI LLVM:** | `(mulhu i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (mulhu a1:regs a2:regs))` |

| **Nome:** | SHLCONST |
|---|---|
| **Descrição:** | Desloca para a esquerda um valor em um registrador por um número constante de casas. |
| **RI LLVM:** | `(shl i32:$a1, (tgtimm i32:$a2))` |
| **Árvore:** | `(transfer a3:regs (shl a1:regs imm:a2:tgtimm))` |

| **Nome:** | SHRCONST |
|---|---|
| **Descrição:** | Desloca para a direita um valor em um registrador por um número constante de casas. |
| **RI LLVM:** | `(srl i32:$a1, (tgtimm i32:$a2))` |
| **Árvore:** | `(transfer a3:regs (shr a1:regs imm:a2:tgtimm))` |

| **Nome:** | SRACONST |
|---|---|
| **Descrição:** | Desloca para a direita um valor em um registrador por um número constante de casas, utilizando deslocamento aritmético (se o número for negativo, preserva o sinal). |
| **RI LLVM:** | `(sra i32:$a1, (tgtimm i32:$a2))` |
| **Árvore:** | `(transfer a3:regs (asr a1:regs imm:a2:tgtimm))` |

| **Nome:** | ROLCONST |
|---|---|
| **Descrição:** | Rotaciona para a esquerda o valor de um registrador por um número constante de casas. |
| **RI LLVM:** | `(rotl i32:$a1, (tgtimm i32:$a2))` |
| **Árvore:** | `(transfer a3:regs (rol a1:regs imm:a2:tgtimm))` |

| **Nome:** | RORCONST |
|---|---|
| **Descrição:** | Rotaciona para a direita o valor de um registrador por um número constante de casas. |
| **RI LLVM:** | `(rotr i32:$a1, (tgtimm i32:$a2))` |
| **Árvore:** | `(transfer a3:regs (ror a1:regs imm:a2:tgtimm))` |

| **Nome:** | SHL |
|---|---|
| **Descrição:** | Desloca para a esquerda um valor em um registrador por um número de casas especificado em outro registrador. |
| **RI LLVM:** | `(shl i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (shl a1:regs a2:regs))` |

| **Nome:** | SHR |
|---|---|
| **Descrição:** | Desloca para a direita um valor em um registrador por um número de casas especificado em outro registrador. |
| **RI LLVM:** | `(srl i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (shr a1:regs a2:regs))` |

| **Nome:** | SRA |
|---|---|
| **Descrição:** | Desloca para a direita um valor em um registrador por um número de casas especificado em outro registrador. Efetua deslocamento aritmético. |
| **RI LLVM:** | `(sra i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (asr a1:regs a2:regs))` |

| **Nome:** | ROL |
|---|---|
| **Descrição:** | Rotaciona para a esquerda um valor em um registrador por um número de casas especificado em outro registrador. |
| **RI LLVM:** | `(rotl i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (rol a1:regs a2:regs))` |

| **Nome:** | ROR |
|---|---|
| **Descrição:** | Rotaciona para a direita um valor em um registrador por um número de casas especificado em outro registrador. |
| **RI LLVM:** | `(rotr i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (ror a1:regs a2:regs))` |

| **Nome:** | OR |
|---|---|
| **Descrição:** | Realiza o OU lógico entre dois registradores. |
| **RI LLVM:** | `(or i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (or a1:regs a2:regs))` |

| **Nome:** | AND |
|---|---|
| **Descrição:** | Realiza o E lógico entre dois registradores.. |
| **RI LLVM:** | `(and i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (and a1:regs a2:regs))` |

| **Nome:** | XOR |
|---|---|
| **Descrição:** | Realiza o OU exclusivo entre dois registradores. |
| **RI LLVM:** | `(xor i32:$a1, i32:$a2)` |
| **Árvore:** | `(transfer a3:regs (xor a1:regs a2:regs))` |

| **Nome:** | BRCOND |
|---|---|
| **Descrição:** | Compara o valor entre dois registradores e realiza o salto para o endereço de um bloco básico se o valor do primeiro registrador for menor que o valor do segundo registrador. |
| **RI LLVM:** | `(br_cc (condcode SETLT) i32:$lhs i32:$rhs (basicblock i32:$tgt))` |
| **Árvore:** | `(cjump const:cond:lt (comp lhs:regs rhs:regs) imm:tgt:int)` |

| **Nome:** | BRCOND2 |
|---|---|
| **Descrição:** | Compara o valor entre dois registradores e realiza o salto para o endereço de um bloco básico se o valor do primeiro registrador for maior que o valor do segundo registrador. |
| **RI LLVM:** | `(br_cc (condcode SETGT) i32:$lhs i32:$rhs (basicblock i32:$tgt))` |
| **Árvore:** | `(cjump const:cond:gt (comp lhs:regs rhs:regs) imm:tgt:int)` |

| **Nome:** | BRCOND3 |
|---|---|
| **Descrição:** | Compara o valor entre dois registradores e realiza o salto para o endereço de um bloco básico se o valor do primeiro registrador for menor ou igual ao valor do segundo registrador. |
| **RI LLVM:** | `(br_cc (condcode SETLE) i32:$lhs i32:$rhs (basicblock i32:$tgt))` |
| **Árvore:** | `(cjump const:cond:le (comp lhs:regs rhs:regs) imm:tgt:int)` |

| **Nome:** | BRCOND4 |
|---|---|
| **Descrição:** | Compara o valor entre dois registradores e realiza o salto para o endereço de um bloco básico se o valor do primeiro registrador for maior ou igual ao valor do segundo registrador. |
| **RI LLVM:** | `(br_cc (condcode SETGE) i32:$lhs i32:$rhs (basicblock i32:$tgt))` |
| **Árvore:** | `(cjump const:cond:ge (comp lhs:regs rhs:regs) imm:tgt:int)` |

| **Nome:** | BRCOND5 |
|---|---|
| **Descrição:** | Compara o valor entre dois registradores não sinalizados e realiza o salto para o endereço de um bloco básico se o valor do primeiro registrador for menor que o valor do segundo registrador. |
| **RI LLVM:** | `(br_cc (condcode SETULT) i32:$lhs i32:$rhs (basicblock i32:$tgt))` |
| **Árvore:** | `(cjump const:cond:ult (comp lhs:regs rhs:regs) imm:tgt:int)` |

| **Nome:** | BRCOND6 |
|---|---|
| **Descrição:** | Compara o valor entre dois registradores não sinalizados e realiza o salto para o endereço de um bloco básico se o valor do primeiro registrador for maior que o valor do segundo registrador. |
| **RI LLVM:** | `(br_cc (condcode SETUGT) i32:$lhs i32:$rhs (basicblock i32:$tgt))` |
| **Árvore:** | `(cjump const:cond:ugt (comp lhs:regs rhs:regs) imm:tgt:int)` |

| **Nome:** | BRCOND7 |
|---|---|
| **Descrição:** | Compara o valor entre dois registradores não sinalizados e realiza o salto para o endereço de um bloco básico se o valor do primeiro registrador for maior ou igual ao valor do segundo registrador. |
| **RI LLVM:** | `(br_cc (condcode SETUGE) i32:$lhs i32:$rhs (basicblock i32:$tgt))` |
| **Árvore:** | `(cjump const:cond:uge (comp lhs:regs rhs:regs) imm:tgt:int)` |

| **Nome:** | BRCOND8 |
|---|---|
| **Descrição:** | Compara o valor entre dois registradores não sinalizados e realiza o salto para o endereço de um bloco básico se o valor do primeiro registrador for menor ou igual ao valor do segundo registrador. |
| **RI LLVM:** | `(br_cc (condcode SETULE) i32:$lhs i32:$rhs (basicblock i32:$tgt))` |
| **Árvore:** | `(cjump const:cond:ule (comp lhs:regs rhs:regs) imm:tgt:int)` |

| **Nome:** | BRCOND9 |
|---|---|
| **Descrição:** | Compara o valor entre dois registradores e realiza o salto para o endereço de um bloco básico se o valor do primeiro registrador não for igual ao valor do segundo registrador. |
| **RI LLVM:** | `(br_cc (condcode SETNE) i32:$lhs i32:$rhs (basicblock i32:$tgt))` |
| **Árvore:** | `(cjump const:cond:ne (comp lhs:regs rhs:regs) imm:tgt:int)` |

| **Nome:** | BRCOND10 |
|---|---|
| **Descrição:** | Compara o valor entre dois registradores e realiza o salto para o endereço de um bloco básico se o valor do primeiro registrador for igual ao valor do segundo registrador. |
| **RI LLVM:** | `(br_cc (condcode SETEQ) i32:$lhs i32:$rhs (basicblock i32:$tgt))` |
| **Árvore:** | `(cjump const:cond:eq (comp lhs:regs rhs:regs) imm:tgt:int)` |

| **Nome:** | BRCOND11 |
|---|---|
| **Descrição:** | Compara o valor de um registrador e salta para o endereço de um bloco básico caso o valor seja igual a 1. |
| **RI LLVM:** | `"(brcond i32:$res (basicblock i32:$tgt))";` |
| **Árvore:** | `(cjump const:cond:eq (comp res:regs const:byte:1) imm:tgt:int)` |

| **Nome:** | BR |
|---|---|
| **Descrição:** | Salto para o endereço de um bloco básico. |
| **RI LLVM:** | `(br (basicblock i32:$tgt))` |
| **Árvore:** | `(jump imm:tgt:int)` |

| **Nome:** | BRIND |
|---|---|
| **Descrição:** | Salto para o endereço contido em um registrador (salto indireto utilizado para implementar tabelas de salto). |
| **RI LLVM:** | `(brind i32:$tgt)` |
| **Árvore:** | `(jump tgt:regs)` |

| **Nome:** | CONST |
|---|---|
| **Descrição:** | Carrega em um registrador o valor de uma constante do tamanho do imediato da arquitetura alvo. |
| **RI LLVM:** | `(tgtimm i32:$a1)` |
| **Árvore:** | `(transfer a2:regs imm:a1:tgtimm)` |

| **Nome:** | CONST16 |
|---|---|
| **Descrição:** | Carrega em um registrador o valor de uma constante de 16 bits. |
| **RI LLVM:** | `(shortimm i32:$a1)` |
| **Árvore:** | `(transfer a2:regs imm:a1:short)` |

| **Nome:** | CONST32 |
|---|---|
| **Descrição:** | Carrega em um registrador o valor de uma constante de 32 bits. |
| **RI LLVM:** | `(imm i32:$a1)` |
| **Árvore:** | `(transfer a2:regs imm:a1:int)` |