Seleção de Padrões de Código para Síntese de *Datapaths* Especializados

Paulo Eduardo Ferreira de Castro

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Seleção de Padrões de Código para Síntese de Datapaths Especializados

Paulo Eduardo Ferreira de Castro[1]

29 de junho de 2004

Banca Examinadora:

- Rodolfo Jardim de Azevedo (Orientador)
  Instituto de Computação - UNICAMP
- Carlos Augusto Paiva da Silva Martins
  PUC-MG
- Célio Cardoso Guimarães
  Instituto de Computação - UNICAMP

Co-orientador:

- Guido Costa Souza de Araújo
  Instituto de Computação - UNICAMP

1 Financiado pela FAPESP, processo 01/12763-1.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Castro, Paulo Eduardo Ferreira de

C279s Seleção de padrões de código para síntese de datapaths especializados / Paulo Eduardo Ferreira de Castro -- Campinas, [S.P. :s.n.], 2004.

Orientadores: Rodolfo Jardim de Azevedo, Guido Costa Souza de Araújo

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

1. Sistemas embutidos de computador. 2. Arquitetura de computador. 3. Dispositivos lógico programáveis. I. Azevedo, Rodolfo Jardim de. II. Araújo, Guido Costa Souza de. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Seleção de Padrões de Código para Síntese de Datapaths Especializados

Este exemplar corresponde à redação final, devidamente corrigida, da Dissertação defendida por Paulo Eduardo Ferreira de Castro e aprovada pela Banca Examinadora.

Campinas, 21 de julho de 2004.

Prof. Dr. Rodolfo Jardim de Azevedo
(Orientador)

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

T E R M O D E A P R O V A Ç Ã O

Tese defendida e aprovada em 29 de junho de 2004, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Carlos Augusto Paiva da Silva Martins
PUC - MG

Prof. Dr. Célio Cardoso Guimarães
IC - UNICAMP

Prof. Dr. Rodolfo Jardim de Azevedo
IC - UNICAMP

Resumo

Sistemas computacionais embutidos, caracterizados pelo emprego de microprocessadores inseridos em contextos eletromecânicos maiores, são atualmente encontrados por toda parte: automóveis, satélites, roteadores de redes, telefones celulares, câmeras digitais, equipamentos hospitalares, sensores biométricos, caixas bancários, equipamentos industriais e muitos outros. Estes sistemas estão freqüentemente sujeitos a severas restrições de desempenho (aplicações de tempo real ou banda larga), consumo energético (dispositivos portáteis) e, invariavelmente, custo de produção. Esta dissertação discute técnicas de adaptação da arquitetura *(hardware)* para aplicações específicas *(software)* de sistemas embutidos, capazes de levar a grandes ganhos de desempenho aliados a outras vantagens. Contribui especificamente com métodos e ferramentas para a extração, análise e seleção automática de padrões (trechos) de código de aplicações que resultem nos maiores ganhos de desempenho (com possível redução de consumo energético) se implementados em *hardware*, na forma de ASIPs *(Application Specific Instruction-Set Processors)*. Faz parte do *Projeto Chameleon* em desenvolvimento no LSC/UNICAMP, que objetiva a especialização de processadores RISC para sistemas embutidos.

Abstract

Embedded computing systems, characterized by the use of microprocessors in larger electromechanical systems, can be currently found everywhere: vehicles, satellites, network routers, cell phones, digital cameras, medical equipment, biometric id sensors, ATMs, industrial equipment, military and scientific applications and many more. Those systems are often subjected to severe constraints in performance (real time or broadband applications), energy consumption (portable devices) and, invariably, production cost. This dissertation discusses techniques of hardware architecture adaptation to best suit specific embedded applications (software), which have demonstrated to be capable of large performance improvements along with other advantages. It specifically contributes with methods and tools for automatic extraction, analysis and selection of patterns (regions) in application code that result in the largest performance gains (with possible power savings) if implemented in hardware, in the form of ASIPs (Application Specific Instruction Set Processors). It is part of the *Chameleon Project* under development at LSC/UNICAMP, which focuses in the specialization of RISC processors for embedded systems.

Agradecimentos

Agradeço à minha família, pelo apoio pessoal e material, irrestrito e permanente.

Ao meu orientador, Dr. Rodolfo Jardim de Azevedo, e ao meu co-orientador, Dr. Guido Costa Souza de Araújo, pela dedicação, atenção, direcionamento, críticas e sugestões ao longo do mestrado.

À equipe do *Projeto Chameleon*, pelas idéias em reuniões, contribuições ao trabalho e espírito de equipe: Bruno Santos, Edson Borin, Eduardo Billo, Felipe Klein, Luciana Shiroma, Nahri Moreano e Paulo Centoducatte (em ordem alfabética). Em especial a Edson Borin e a Nahri Moreano, pelas sugestões de algoritmos, esclarecimentos e contribuições de código.

A todos os membros do LSC, pela amizade e convivência harmoniosa por mais de 2 anos. Aos professores e funcionários do IC, por construírem um ambiente propício à aprendizagem e à pesquisa. À comunidade acadêmica da UNICAMP, pela valorização da pós-graduação e da pesquisa científica.

Ao Governo do Estado de São Paulo que, através de sua Fundação de Amparo à Pesquisa (FAPESP), possibilitou minha dedicação exclusiva ao mestrado com uma bolsa de estudos (processo 01/12763-1). Ao CNPq (processo PDI&TI 55.2117/2002-1) e novamente à FAPESP (processo 2000/15083-9), pelo financiamento de infraestrutura e bolsas ligadas aos projetos do LSC.

Aos meus pais.

Sumário

Índice de Figuras

Índice de Tabelas

Lista de Acrônimos

ADPCM	Adaptive Differential Pulse Code Modulation
ALU	Arithmetic and Logic Unit
ASIC	Application Specific Integrated Circuit
ATM	Automated Teller Machine
CDFG	Control and Data Flow Graph
CDG	Control Dependence Graph
CFG	Control Flow Graph
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection [GCC04]
GPL	GNU General Public License [GPL04]
GPS	Global Positioning System
GSM	Global System for Mobile Communications (tecnologia celular)
IC	Instituto de Computação (UNICAMP)
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
JPEG	Joint Photographic Experts Group
LSC	Laboratório de Sistemas de Computação (IC-UNICAMP)
LUT	Lookup Table
MD4	Message Digest 4 – one way hash function
MD5	Message Digest 5 – one way hash function
MMX	Multi Media eXtension (Instruction Set Extension)
MP3	Moving Pictures Experts Group Layer-3 Audio
MPEG	Moving Picture Experts Group (ISO/IEC)
NFS	Network File System
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Language
RTX	RTL Expression
SHA	Secure Hash Algorithm
SIMD	Single Instruction, Multiple Data
SRAM	Static Random Access Memory
SSE	Streaming SIMD Extension (former KNI - Katmai New Instructions)
UF	Unidade Funcional (lógica agregada à *datapath* do processador)
UNICAMP	Universidade Estadual de Campinas
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Large Instruction Word
VLSI	Very Large Scale Integration

Capítulo 1

Introdução

O trabalho apresentado nesta dissertação se enquadra na área de projeto de arquitetura de computadores, tendo ligações com a área de arquiteturas reconfiguráveis. O principal objetivo é a melhora no desempenho de sistemas embutidos, caracterizados como dispositivos eletrônicos que agregam capacidades de processamento de informação através de microprocessadores e/ou microcontroladores. Este capítulo introduz o tema, estando dividido em três seções. A seção 1.1 apresenta as motivações para o trabalho, citando aplicações práticas. A seção 1.2 oferece uma introdução a técnicas gerais de reconfiguração de arquiteturas para aplicações específicas através da reconsideração da fronteira entre o *hardware* e *software*. As contribuições específicas desta dissertação e a organização dos capítulos são fornecidas na seção 1.3.

1.1. Motivações

Sistemas embutidos já são largamente disseminados na atualidade, num cenário que projeta expansões ainda maiores para os próximos anos na medida em que os equipamentos eletrônicos portáteis se tornem mais baratos e expandam suas capacidades de comunicação de dados digitais por redes sem fio ubíquas. Exemplos dos sistemas embutidos que podem se beneficiar das técnicas deste trabalho permeiam diversos domínios de aplicação:

- Sistemas embutidos em veículos de transporte, tais como: computadores de bordo em automóveis que apresentam ao motorista mapas da região em tempo real; inúmeros dispositivos em aviões para monitorar e atuar sobre elementos mecânicos que aumentem a segurança e estabilidade do vôo; controladores de linhas de trens rápidos que atuem sobre trilhos para evitar acidentes;

- Sistemas embutidos em equipamentos de telecomunicações, tais como em satélites, roteadores, *switches*, antenas, regeneradores de sinal em linhas de cabos metálicos ou fibras ópticas, etc.;

- Sistemas embutidos em eletrodomésticos e equipamentos eletrônicos individuais, tais como em telefones celulares multimídia, câmeras digitais, televisores digitais, tocadores digitais *(MP3 players)*, organizadores e agendas eletrônicos, dispositivos de orientação com acesso ao sistema de posicionamento global (GPS), automação de ambientes domésticos (controle inteligente de iluminação e aquecimento em casas e edifícios), etc.;

- Sistemas embutidos em equipamentos hospitalares de Unidades de Tratamento Intensivo (UTI) e também em dispositivos inseridos no paciente (órgãos artificiais, marca-passo), equipamentos laboratoriais de análises clínicas, equipamentos de raios X, ressonância magnética e ultrasom, etc.;

- Sistemas embutidos em equipamentos industriais, tais como robôs para montagem de partes complexas e ações repetitivas, atuadores e monitores em linhas de produção, controle remoto de ambientes insalubres, etc.;

- Sistemas computacionais embutidos em dispositivos de segurança e acesso, tais como em leitores de impressões digitais e íris, detectores de armas e explosivos, acionadores de portas e alarmes que se comunicam com centrais de segurança, etc.;

- Sistemas computacionais embutidos para segurança da informação, tais como validação e criptografia de informações bancárias e senhas em pagamentos eletrônicos por cartões de débito/crédito, transações bancárias em ATMs (caixas em quiosques), autenticação de assinaturas digitais por dispositivos portáteis, etc.;

- Aplicações militares, aplicações em perícia civil, inúmeras aplicações científicas (física, química, engenharia), exploração e viagens espaciais, etc.

Pesquisas mostram que o norte-americano médio tem contato com 60 microprocessadores por dia (*circa* 1996); estima-se que 79% dos microprocessadores produzidos têm como destino os sistemas embutidos e que programadores escrevem 5 vezes mais código para sistemas embutidos do que para computadores convencionais [Mar03].

Os ganhos de desempenho que este trabalho tem por objetivo se traduzem primariamente na redução do tempo de execução das aplicações que executam em sistemas embutidos e, secundariamente, na redução do tamanho do código[2]. A redução do tamanho do código, por sua vez, vai na direção de uma menor necessidade de memória e, com isso, de um menor custo econômico e consumo energético [BSL+01, SZW+01]. O aumento na velocidade de execução é um objetivo muito importante para sistemas embutidos cujas aplicações sejam classificadas como *tempo real*, nas quais uma resposta do sistema precisa ocorrer dentro de um intervalo de tempo máximo, e também para sistemas *reativos*, que atuam permanentemente em resposta a condições apresentadas pelo ambiente. Se um sistema embutido que requer um baixo tempo de execução está inserido num ambiente estático e controlado, os projetistas podem se dar ao luxo de elaborar soluções de alto consumo energético e sem restrições de tamanho, como o aumento da freqüência do *clock* ou adição de processadores. Porém, no caso de dispositivos portáteis ou que operem com a energia solar ou de baterias, ganhos de tempo de execução não podem ser trocados por aumentos significativos de energia consumida e espaço.

Este trabalho compreende contribuições para um projeto maior em desenvolvimento no Laboratório de Sistemas de Computação (LSC) do IC/UNICAMP, denominado *Projeto Chameleon*, no qual trabalham atualmente 3 docentes, 2 alunos de doutorado, 4 alunos de mestrado e 2 alunos de iniciação científica (graduação). No Projeto Chameleon, obtemos ganhos de desempenho em aplicações embutidas explorando uma reconfiguração da arquitetura do microprocessador, em troca de um muito pequeno e, na maioria dos casos, insignificante aumento na área de silício do processador. Exploramos o fato de que os sistemas embutidos tipicamente executam um pequeno conjunto de programas que não se alteram (em oposição a estações de trabalho de uso geral, projetadas para executar programas arbitrários). Propomos que os programas a serem executados no

2 Na verdade, a redução do tamanho código se dá na forma de um efeito colateral positivo de nossas técnicas.

sistema embutido sejam (automaticamente) analisados para levar a uma reconfiguração de uma arquitetura de um processador pré-existente que o torne especializado para a execução eficiente daqueles programas.

1.2. Especialização de Arquiteturas: *Hardware versus Software*

Existem duas formas básicas de se executar um procedimento utilizando *chips* de silício. Uma delas é a implementação das operações utilizando transistores (ou, em maior nível de abstração, portas lógicas e unidades funcionais como registradores, flip-flops e ALUs), uma abordagem que pode ser chamada *hard-wired*, ou simplesmente *implementação em hardware*. *Chips* projetados para aplicações específicas são denominados *Application Specific Integrated Circuits* (ASICs). Do ponto de vista de eficiência computacional, é a forma pela qual se consegue a execução mais rápida das operações. No entanto, essa não é uma abordagem flexível: se qualquer alteração for necessária nas funções computadas, é necessário fabricar um novo *chip*. A outra forma de se executar um procedimento é através de processadores de propósito geral, capazes de executar instruções de um programa, uma abordagem que pode ser chamada *implementação em software*. É uma abordagem bastante flexível no sentido de que, sendo necessário alterar as operações computadas, basta alterar o programa sem a necessidade de modificar o *chip*.

Entre estas duas abordagens existem as *arquiteturas reconfiguráveis,* usualmente definidas como arquiteturas dotadas de um *hardware* que pode ser reprogramado para otimizar a computação de funções específicas. Exemplos deste tipo de *hardware* são LUTs *(Look-Up Tables)* e FPGAs *(Field Programmable Gate Arrays)*, que podem ser eletricamente reconfiguradas *in loco* no circuito. FPGAs provêem reconfigurações permanentes, que se mantêm quando o circuito é desligado, enquanto LUTs são usualmente implementadas na forma de SRAMs voláteis, como ilustra a Figura 1. Alguns dispositivos reconfiguráveis são dotados de múltiplos contextos, permitindo que um contexto seja reprogramado enquanto outro contexto está sendo reconfigurado, e que a transição do contexto ativo ocorra mais bem rapidamente que o tempo de reconfiguração, como mostra a Figura 2. A

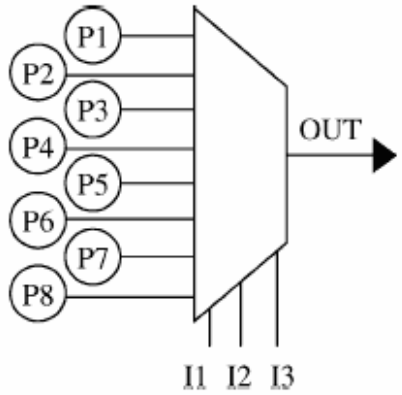


Figura 1 - Uma LUT. Funciona como uma SRAM volátil, em que os bits de endereço I1 a I3 selecionam a função lógica e os bits de configuração P1 a P8 determinam o resultado. Figura obtida de [CH02].

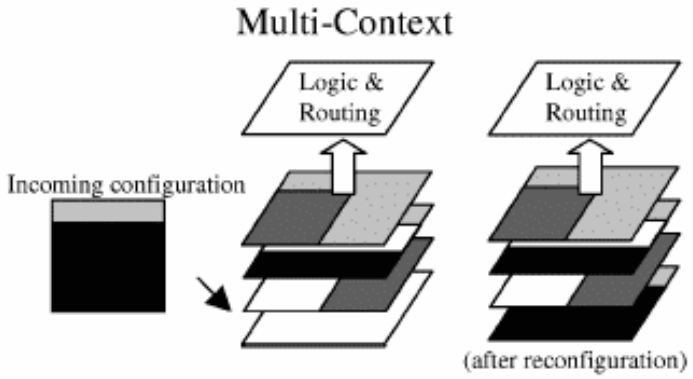


Figura 2 - Ilustração de uma FPGA dotada de múltiplos contextos. Cada contexto implementa um circuito diferente e é representado por um plano. Vários contextos podem ser pré-programados; apenas um está ativo num dado instante, mas a transição entre eles é relativamente rápida. Um contexto pode programado enquanto outro está ativo. Figura obtida de [CH02].

granularidade da programação de unidades reconfiguráveis permite ainda classificá-las como *fine-grained*, quando a programação ocorre no nível de portas lógicas e *bits*, ou *coarse-grained*, quando a reprogramação se traduz em mudar a interconexão de circuitos pré-definidos como ALUs e blocos de memória. Unidades *fine-grained* permitem a implementação de circuitos arbitrários e altamente otimizados, mas possuem maior tempo de reconfiguração. Unidades *coarse-grained* são usualmente mais velozes, mas oferecem um leque menor de possibilidades para os circuitos implementados.

Projetos arbitrariamente complexos, tais como processadores inteiros, podem atualmente ser programados em FPGAs e tecnologias assemelhadas. Apesar dos avanços tecnológicos dos últimos anos, que aumentou consideravelmente a capacidade de FPGAs e LUTs (tamanho dos circuitos que é capaz de implementar) e suas velocidades de execução e reconfiguração, elas ainda continuam sendo significativamente mais lentas que os processadores dedicados convencionais, e o tempo de reconfiguração prejudica o seu emprego extensivo em arquiteturas que exploram a reconfiguração em tempo de execução. No entanto, uma abordagem que tem sido alvo de trabalho de um grande número de pesquisadores é a combinação de *hardware* programável com processadores convencionais, atribuindo ao primeiro a execução de algumas funções específicas e, ao segundo, a execução de funções arbitrárias. Nesta direção, algumas formas de acoplamento entre unidades programáveis e unidades não programáveis podem ser classificadas como ilustra a Figura 3. As quatro possibilidades ilustradas (em cinza) para as unidades reconfiguráveis são:

- Uma ou mais unidades funcionais (UF) agregadas a estágios do *pipeline* do processador, especialmente os estágios de decodificação de instruções e de execução. Neste caso, a UF pode ser adicionada ao circuito original do processador existente ou mesmo substituí-lo completamente. Caso seja apenas adicionada, multiplexadores são utilizados para selecionar se o resultado da computação virá do circuito original ou da UF adicionada. Assim é possível, por exemplo, adicionar novas instruções no processador que ativem a UF, ou então modificar a computação executada por instruções que já existiam no processador original. Dois exemplos bastante citados destas arquiteturas são PRISC [RS94] e Chimaera [HFHK97].

- Um coprocessador reconfigurável, que utilize um barramento especial (usualmente síncrono) com o processador principal para acesso aos *opcodes* das instruções trazidas da memória e ao banco de registradores. Freqüentemente, esta solução exige dois ou três ciclos apenas para troca de dados entre o processador e o coprocessador, que é feita através de instruções especiais (como *load coprocessor* ou *store coprocessor*). Exemplos destas arquiteturas são [WC96], [HW97], [MO98], [RLG+98] e [CS00].

- Uma unidade computacional anexa ao barramento de dados e endereço – usualmente o mesmo barramento da memória. Este também é caso de *memórias ativas* (memórias semelhantes a *caches* programáveis que não apenas armazenam mas executam computações entre a escrita e a leitura de dados) e o caso de placas conectadas a barramentos como ISA ou PCI. Esta solução usualmente exige vários ciclos para troca de dados com o processador, especialmente quando o barramento processador/*cache* executa com *clock* mais alto que o barramento processador/RAM. Referências para estas arquiteturas são [VBR+96], [LTS99] e [AM98].

- Uma unidade reconfigurável externa, utilizando interfaces de entrada/saída tais como Ethernet ou USB. O tempo de comunicação pode ser bastante grande, ordens de grandeza maior que o período do *clock* do processador. Esta solução só é aplicável a casos em que a computação especial é

realizada esporadicamente e consome bastante tempo para terminar.

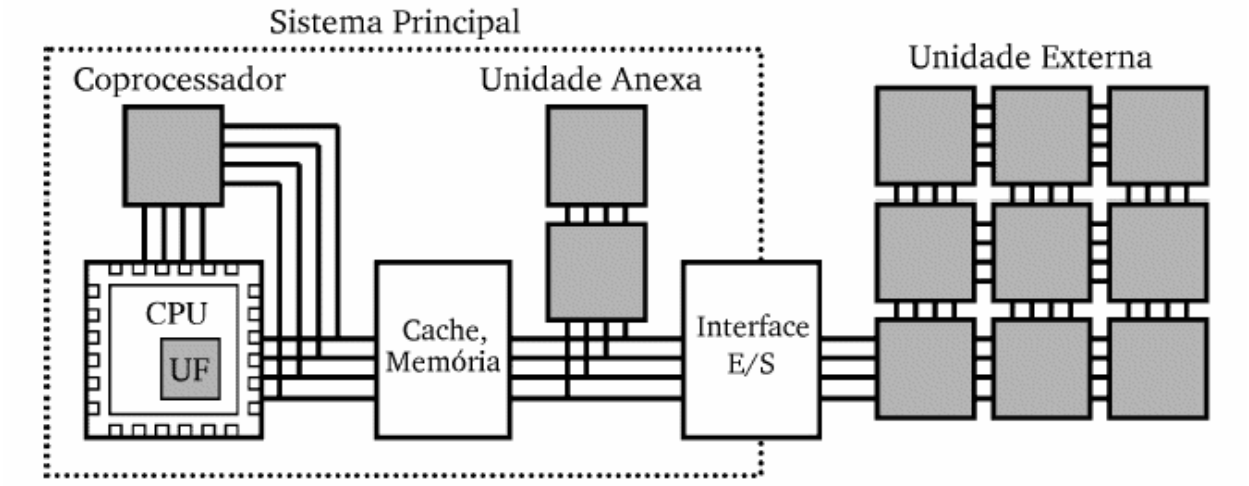


Figura 3 - Formas usuais de acoplamento entre unidades reconfiguráveis (em cinza) e não reconfiguráveis (em branco). Figura obtida de [CH02].

De um modo geral, quanto mais distante do processador estiver a unidade funcional reconfigurável, maior o tempo de comunicação porém mais flexível é a solução – no sentido de que, quanto mais próximo ao processador, mais dedicado ou específico precisa ser o projeto.

A abordagem do Projeto Chameleon é a adição de unidades funcionais ao *datapath* de processadores RISC. Como será visto no Capítulo 2, estas unidades funcionais são associadas a novas instruções adicionadas ao conjunto de instruções original, e implementam computações específicas dos programas que executarão no sistema embutido. Como a nova unidade funcional é interligada diretamente aos estágios do *pipeline* do processador, não há *overhead* de comunicação, e ganhos de poucos ciclos em relação à computação originalmente realizada em *software* (com as instruções originais do processador) podem se traduzir em ganhos expressivos no tempo global de execução se as operações especiais computadas pertencerem a laços da aplicação que são repetidos muitas vezes. Aplicações multimídia, criptográficas, científicas e de telecomunicações (típicas de sistemas embutidos) sempre possuem laços altamente executados para computação de algoritmos de compactação, codificação, ordenação e buscas, cálculo numérico e discreto, operações em grafos, etc.

Há, no entanto, uma diferença importante entre a abordagem do Projeto Chameleon e a abordagem previamente apresentada do uso de uma UF reconfigurável. O Projeto Chameleon propõe que o desenvolvimento e implementação de UFs se dê uma única vez, em tempo de compilação/simulação – diferentemente de reconfigurações em tempo de execução. Assim, não há de fato a necessidade de se utilizar uma unidade reconfigurável como uma FPGA. Um novo projeto do processador é criado de forma consideravelmente automatizada utilizando linguagens de descrição de *hardware* como VHDL e Verilog, ou linguagens de descrição de arquitetura como ArchC e SystemC (mais detalhes no próximo capítulo). Este novo projeto pode ser usado para a implementação de um ASIC, de modo que o circuito da UF execute tão rapidamente quanto o processador original e não haja qualquer atraso devido a reconfigurações. A especialização do processador original se dá num sentido mais amplo, semelhante à reconfiguração *coarse-grained*, através de multiplexadores que são ativados pelas novas instruções e que selecionam o resultado da computação proveniente do *datapath* original

ou das UFs adicionadas, como ilustra a Figura 4. Obviamente, FPGAs podem ser usadas para prototipagem e experimentação, e isto é exatamente o que fazemos no LSC.

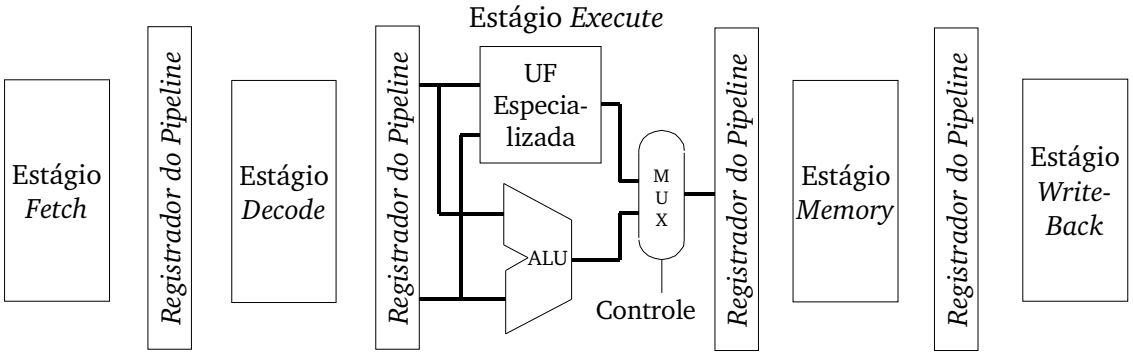


Figura 4 - Ilustração do uso de um multiplexador para selecionar o resultado da computação no estágio de execução de um processador RISC pipelined. Esta é a abordagem empregada nas atuais implementações do Projeto Chameleon. Uma instrução especial ativa o sinal de controle do MUX que seleciona o resultado da computação da unidade funcional especializada; do contrário, é selecionado o resultado da ALU original do processador.

1.3. Contribuições desta dissertação

Como já mencionado, o trabalho aqui apresentado é parte do Projeto Chameleon em desenvolvimento no LSC. As contribuições específicas desta dissertação são:

- A implementação da Biblioteca de Padrões, que abrange: a especificação de um formato de arquivo para armazenamento e troca de padrões de código entre diferentes aplicações do Projeto Chameleon; a especificação de estruturas de dados para representação de padrões em memória, especialmente na forma de grafos de fluxo de controle (CFG), fluxo dados (DFG) e dependências de controle e dados (CDFG); a implementação de funções para leitura/escrita de arquivos, operações sobre grafos (isomorfismo, poda, crescimento, conversão, etc.) e várias outras;

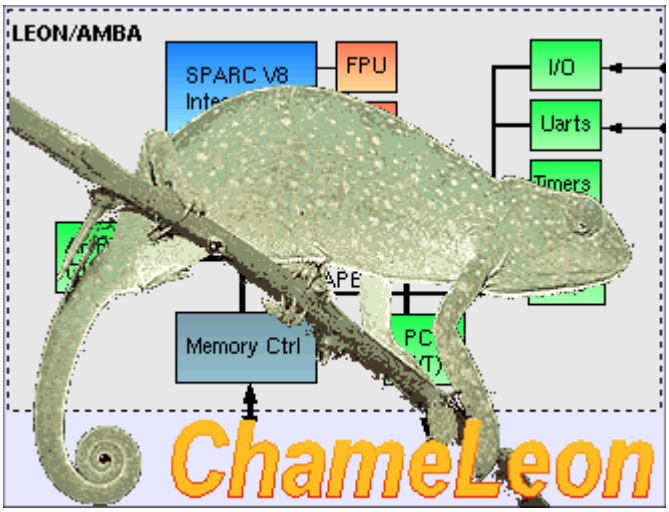
- Projeto e implementação de algoritmos e ferramentas (linha de comando) para extração de padrões de código de aplicações, crescimento de padrões na forma de subgrafos do CFG de funções, filtragem de padrões, casamento (comparação) de padrões entre aplicações distintas de diferentes pacotes de *benchmarks* de forma automática, atribuição de pesos a padrões proporcionais ao ganho de desempenho de suas implementações em *hardware*, conversão de grafos de padrões em formatos gráficos para visualização e geração de relatórios (em formatos para leitura por humanos ou por outras ferramentas de *software*);

- Proposição de experimentos para a extração de padrões candidatos a uma implementação em *hardware*; produção de gráficos com dados estatísticos dos padrões; classificação dos padrões selecionados em categorias e análise dos resultados obtidos.

O Capítulo 2 detalha o Projeto Chameleon, situando estas contribuições no fluxograma global do projeto, além de apresentar outros trabalhos relacionados. O Capítulo 3 apresenta a Biblioteca de Padrões, aprofundando sobre o conceito de padrão de código, dos grafos utilizados na representação de padrões, dos algoritmos que operam sobre padrões e do formato de arquivo projetado para armazenamento de padrões. O Capítulo 4 descreve os experimentos realizados e apresenta os resultados experimentais obtidos, terminando com uma classificação dos padrões selecionados em categorias funcionais. Finalmente, o Capítulo 5 apresenta as conclusões finais e as sugestões para trabalhos futuros.

Capítulo 2

Projeto Chameleon e Trabalhos Relacionados



2.1. Projeto Chameleon

O *Projeto Chameleon* teve início no LSC (Laboratório de Sistemas de Computação, IC/UNICAMP) no ano de 2002. O objetivo do projeto é a especialização de processadores para sistemas embutidos. Exploramos o fato de que estes sistemas estão sempre executando um mesmo pequeno conjunto de aplicações, de forma a ser possível identificar os trechos de código que demandam maior capacidade de processamento antes mesmo da construção do sistema. Uma vez identificados, estes trechos de código são transformados em *hardware* na forma de novas unidades funcionais inseridas na *datapath* do processador, e ativadas através de novas instruções adicionadas ao conjunto de instruções original.

Em mais detalhes, dado o código de uma aplicação, identificamos *padrões de código* – trechos do código representados na forma de grafos de fluxo de dados e de controle – que são executados com maior freqüência através da realização de *profiling*[3]. Com base em diversos critérios, alguns dos padrões são selecionados e transformados, e é criada uma descrição HDL[4] que os implementa em novas unidades funcionais inseridas no processador, associadas a novas instruções. Modificamos o *back-end* de um compilador para que reconheça e possa gerar código binário que faça uso das novas instruções, levando a um conjunto otimizado de *software* e *hardware*. No Projeto Chameleon, construímos ferramentas que automatizam grande parte deste processo. A Figura 5 ilustra as entradas e saídas das transformações realizadas no projeto.

O processador que adotamos é o *Leon*, um modelo VHDL aberto da arquitetura SPARC V8 [SPARC04], com 5 estágios de *pipeline*, sem unidade de ponto flutuante, criado e mantido pela empresa suíça *Gaisler Research* [Gaisler04]. Trabalhamos com a linguagem C e utilizamos um dos compiladores GCC *(GNU Compiler Collection)* [GCC04], junto às ferramentas de código aberto do pacote *binutils* [binutils03] que incluem a ferramenta de *profiling gprof* e o *assembler as*, entre outras.

3 *Profiling* - Instrumentação (adição de código) de um programa para determinação dos trechos (instruções, funções ou blocos básicos) mais executados.

4 *HDL - Hardware Description Language* - linguagens que permitem a especificação, documentação e síntese de *hardware*. Duas bem conhecidas HDLs são VHDL e Verilog.

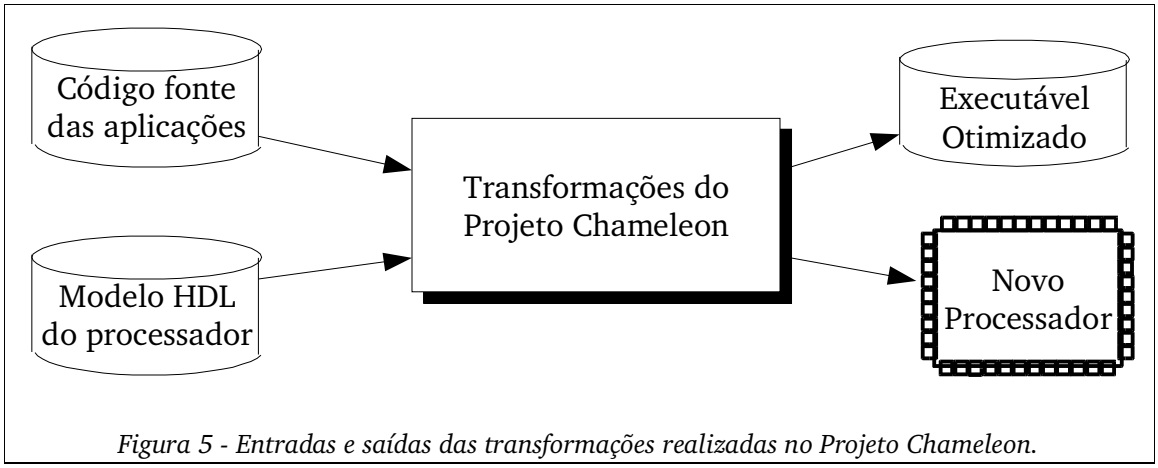


Figura 5 - Entradas e saídas das transformações realizadas no Projeto Chameleon.

A Figura 6 mostra um diagrama das etapas do Projeto Chameleon. Cada etapa numerada no diagrama será resumidamente explicada a seguir. Alguns dos termos e conceitos citados serão abordados em mais detalhes no próximo capítulo, em especial o conceito de padrões de código e os processos de crescimento e seleção dos mesmos.

1. Iniciamos com o código fonte de uma aplicação arbitrária, escrito em linguagem de alto nível (atualmente, apenas na linguagem C). Para efeitos de obtenção de resultados experimentais, utilizamos aplicações de pacotes de *benchmarks*, incluindo MediaBench I e II [LPS97], MiBench [GRE+01], DSP Stone [ZVSM94] e NetBench [MSH01]. O Capítulo 4 descreve algumas aplicações destes pacotes.

2. Utilizamos o GCC para compilar a aplicação, com opções de linha de comando para instrumentar o código para a realização de *profiling*. Com estas opções, a subseqüente execução da aplicação produzirá um arquivo com dados que permitem concluir o número de vezes que cada bloco básico foi executado. Utilizamos o GCC como *cross compiler*, executado num ambiente Linux/Intel mas produzindo código para a arquitetura SPARC. Utilizamos a biblioteca C de código aberto NewLib [NewLib04], especialmente voltada para sistemas embutidos, que é pequena e provê apenas as funções essenciais para este tipo de aplicação.

3. Geramos um binário para a arquitetura SPARC V8, cuja especificação é seguida pelo processador *Leon*.

4. Obtemos os dados de *profiling* executando a aplicação sobre uma versão original (não modificada) do processador Leon. Utilizamos para isso *kits* de desenvolvimento com FPGAs (mais detalhes no capítulo Resultados Experimentais). Uma vez obtidos os dados de *profiling*, uma segunda compilação do código fonte é feita (item 2 novamente). Com isso, o número de vezes que cada bloco básico foi executado fica disponível na representação intermediária do GCC, dando suporte a otimizações avançadas de código e podendo ser transferido para os arquivos de padrões (item 5).

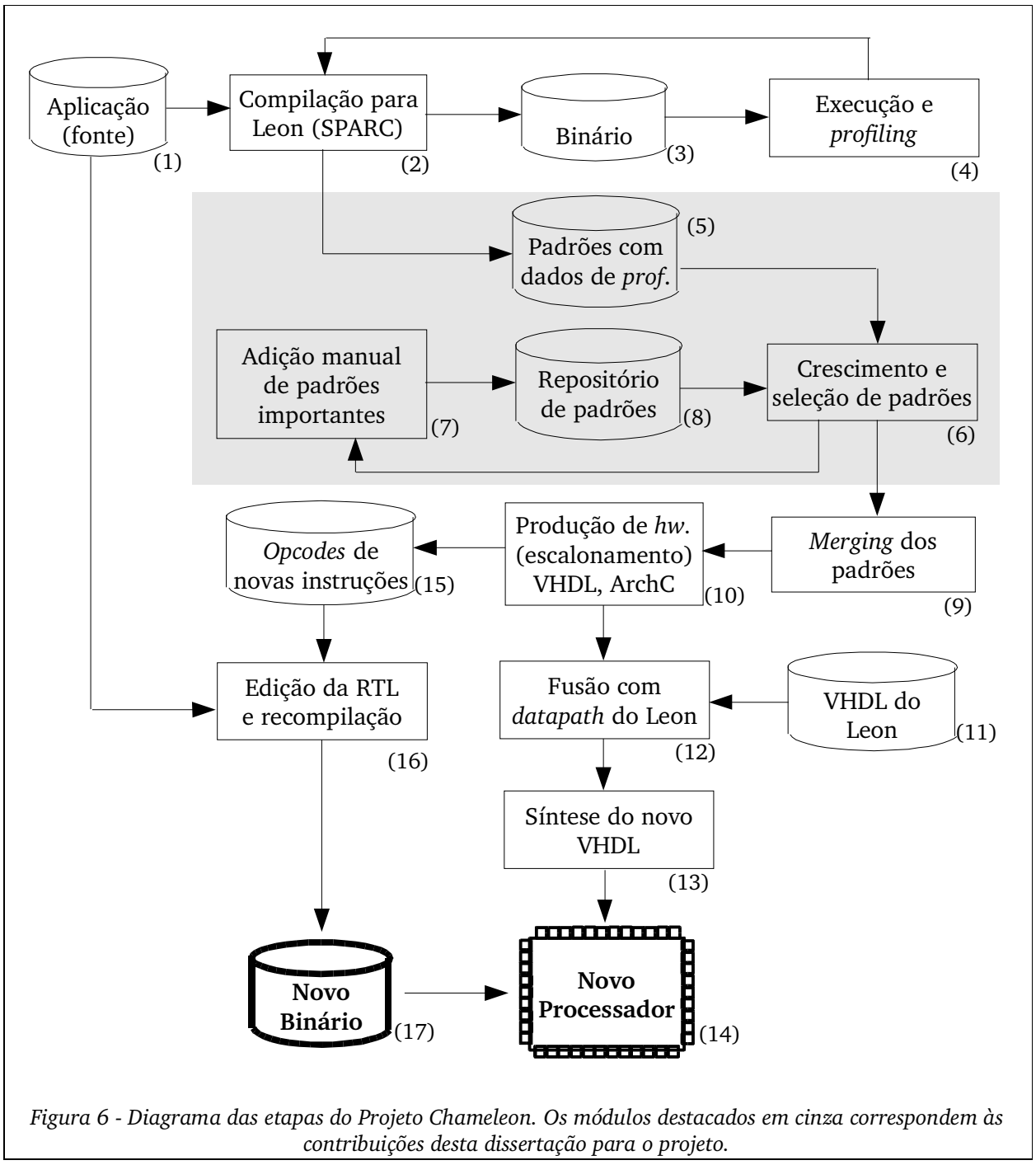


Figura 6 - Diagrama das etapas do Projeto Chameleon. Os módulos destacados em cinza correspondem às contribuições desta dissertação para o projeto.

5. Extraímos padrões de código para arquivos de extensão ".pat", sendo um arquivo para cada função da aplicação. Na extração inicial, geramos um padrão para cada RTX[5] da representação intermediária do GCC. Os padrões guardam diversas informações extraídas da representação intermediária do GCC, incluindo o CFG (*Control Flow Graph*), blocos básicos dominadores e pós-dominadores, freqüências de execução, *definition-use chains* para representar dependências de dados, etc. No próximo capítulo, o propósito e significado destas informações serão explicados em

5 A representação intermediária do GCC é denominada RTL (*Register Transfer Language*). Esta representação é constituída de instruções elementares denominadas RTX.

detalhes, assim como o formato dos arquivos de padrões ".pat".

6. Os padrões extraídos no item 5 passam por um processo de *crescimento* que produz padrões maiores, e em seguida por um processo de seleção que visa determinar os padrões que trariam maior ganho de desempenho para a aplicação se implementados em *hardware*. Estes procedimentos serão explicados em detalhes no próximo capítulo.

7. Com a experiência e as informações adquiridas com o Projeto Chameleon, identificamos manualmente alguns padrões de código típicos que se repetem nas aplicações que compõem *benchmarks* para sistemas dedicados, como os citados no item 1. Buscamos especialmente padrões que, se implementados em *hardware*, darão grande ganho de desempenho (redução do tempo de execução) a um custo muito baixo de *hardware*. Por exemplo, padrões que contenham instruções de deslocamento (*shift* lógico ou aritmético) por uma quantidade fixa de *bits*, seguidas de instruções aritméticas (soma e subtração) e de comparação. Em *hardware*, o deslocamento de *bits* por constante pode ser feito com fios, a um custo quase nulo. Uma descrição otimizada em VHDL é então escrita (também manualmente) para a implementação daqueles padrões em *hardware*, e armazenada em seções especiais dos arquivos ".pat".

8. Os padrões produzidos no item 7 são armazenados em um repositório permanente de padrões. Este repositório contém padrões com descrição VHDL otimizada, e as ferramentas de seleção de padrões buscam por ocorrências dos mesmos nas aplicações sendo processadas. No entanto, note-se que o processo de extração e seleção de padrões não se limita a encontrar ocorrências dos padrões do repositório permanente; padrões extraídos das próprias aplicações são armazenados em repositórios temporários, que são utilizados em procedimentos automáticos de casamento de padrões, conforme será detalhado no próximo capítulo.

9. Os padrões selecionados para a implementação em *hardware* são unidos num procedimento chamado *merging*. Trata-se basicamente de tentar reutilizar unidades funcionais e especialmente suas interconexões, como ilustrado na Figura 7. Por exemplo, se dois diferentes padrões utilizam um somador de 32 bits, é desejável reutilizá-lo para ambos os padrões, visto que eles não executam ao mesmo tempo. No entanto, a saída do somador pode ser ligada a diferentes unidades funcionais nos diferentes padrões, dando origem a um problema de casamento de interconexões. Uma das alunas de doutorado do LSC, Nahri Moreano, em conjunto com o Prof. Guido Araújo e pesquisadores da universidade de *Princeton*, EUA, publicaram um artigo a respeito do problema [MAHM02].

10. O resultado do *merging* é convertido em HDL. Neste processo, modificamos os grafos dos padrões de modo a inserir multiplexadores e registradores que definam um *pipeline*, no qual as unidades funcionais são escalonadas de forma a explorar as possibilidades de paralelismo de operações. Uma vez inseridos os multiplexadores e registradores, escrevemos a descrição do *hardware* em linguagens como VHDL ou ArchC, em modelos comportamentais ou estruturais[6]. ArchC [RAA+03, VBR+04] é uma linguagem de descrição de arquiteturas de processadores baseada em C++ e projetada por docentes e alunos do LSC. Possuímos uma descrição em ArchC do processador SPARC V8 (compatível com o Leon), a partir da qual é gerado um simulador da arquitetura, entre

6 Diz-se que o código é estrutural quando é escrito no nível de sinais elétricos e circuitos lógicos, sendo por outro lado chamado de comportamental quando se vale de abstrações de linguagens de programação de alto nível como o uso de operadores aritméticos.

outras ferramentas.

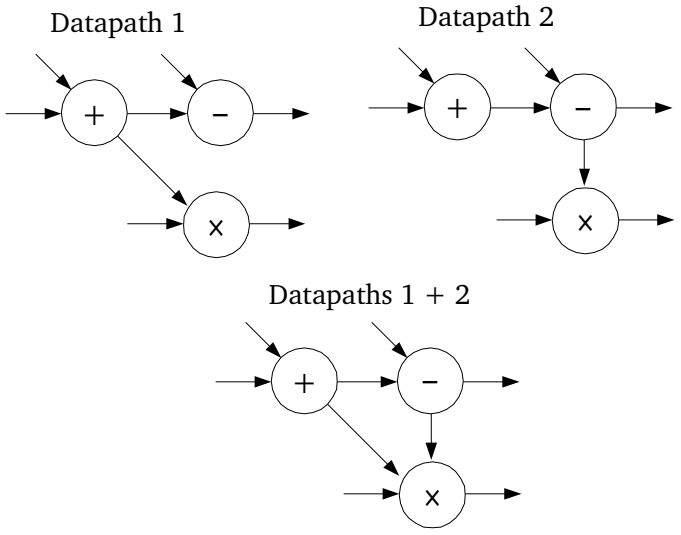


Figura 7 - Datapaths Merging. Os grafos de fluxo de dados e de controle de dois padrões a serem implementados em hardware passam pelo processo de merging, de forma a sobrepor unidades funcionais e suas interconexões, minimizando a área e o custo do hardware.

11. O código VHDL original do processador Leon é alterado (apenas 1 vez) para incluir interfaces configuráveis, porém pré-definidas, com a unidade de inteiros do processador, e também alterado para a inclusão de alguns sinais de controle novos. Estas interfaces facilitam a anexação de unidades funcionais ao *datapath* e permitem a decodificação de novas instruções para o processador.

12. O código VHDL que tenha sido escrito para implementar os padrões é automaticamente mesclado com o código do Leon, utilizando as interfaces citadas no item 11.

13. O código VHDL é sintetizado em *hardware*. Utilizamos ferramentas de síntese da Mentor Graphics [Mentor04], que produzem arquivos de configuração de FPGAs que são carregados na placa de desenvolvimento, conforme descrito no item 4.

14. O novo processador resulta da síntese discutida no item 13. Nos protótipos de laboratório, utilizamos FPGAs para fins de verificação, mas na indústria a síntese poderia ser feita em máscaras VLSI para confecção de ASICs.

15. A cada padrão de código implementado em *hardware* é atribuída uma nova instrução do processador, que é armazenada no campo *tmp_opcode* do arquivo de padrões, conforme descrição do formato de arquivo dada no próximo capítulo. Atualmente, implementamos 4 diferentes tipos de instruções SPARC, como mostra a Figura . Na instrução `cham` do tipo 1, temos dois registradores de origem (rs), um de destino (rd) e ainda uma constante de 8 *bits*; na instrução `cham2` do tipo 2, a constante é omitida. Se utilizarmos a constante omitida na decodificação, obtemos 256 novas instruções do tipo 2. Nas duas formas da instrução `cham3` do tipo 3, um dos registradores de origem é retirado, dando lugar a uma constante de 13 *bits*. A instrução `zol` do tipo 4 é um *zero-overhead loop*: instrui o processador a executar *times* vezes as *n* instruções

imediatamente seguintes à instrução *zol*, onde *n* = *offset*.

16. Nosso compilador GCC modificado é reexecutado, desta vez com opções de linha de comando que o instruem a ler um arquivo ".pat" que contém padrões com campos *tmp_opcode* preenchidos com as instruções *assembly* a serem inseridas no lugar das RTXs originais do padrão. Por exemplo, a notação "tmp_opcode = cham 3, \s[1:2], \s[1:3], \d[1:1]" indica que a nova instrução lê o conteúdo de dois registradores (indicados por "*s*") e escreve num terceiro registrador (indicado por "\d"). Os valores entre colchetes indicam o número do DFG e do operando correspondente ao registrador (conforme descrição do formato do arquivo dada no próximo capítulo). Como ilustrado na Figura , trata-se de uma instrução do tipo 1 em que o campo imediato, que no exemplo tem o valor 3, especifica a quarta instrução do tipo 2 implementada no processador (supondo que o campo imediato esteja sendo empregado na decodificação da instrução).
Previamente, e uma única vez, é necessário modificar arquivos de descrição do conjunto de instruções utilizados pelo *assembler as* do pacote *binutils*, para que as novas instruções emitidas pelo GCC nos arquivos *assembly* "*.s*" sejam reconhecidas e o código binário seja emitido corretamente para elas. Além disso, o GCC também possui uma linguagem de especificação de parâmetros de instruções do processador-alvo denominada *machine description*. Modificando as descrições, podemos especificar se nossas novas instruções podem ser executadas em *delay slots* de *branches*, o que otimiza ainda mais o código.

Instrução do tipo 1:
`cham imm, rs1, rs2, rd`

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Uso:	1	0	rd					0	0	1	0	0	1	rs1					0	*8 bits immediate*								rs2				

Instrução do tipo 2:
`cham2 rs1, rs2, rd`

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Uso:	1	1	rd					0	0	1	0	0	0	rs1					0	*unused*								rs2				

Instruções do tipo 3:
`cham3 rs1, imm, rd`
`cham3 imm, rs1, rd`

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Uso:	1	1	rd					0	0	1	0	0	1	rs1					1	*13 bits immediate*												

Instrução do tipo 4:
`zol times, offset`

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Uso:	0	0	a1					0	0	1	a2			*times (20 bits immediate)*																		

offset: particionado nos campos *a1* e *a2*

Figura 8 - Formatos das instruções atualmente utilizadas no Projeto Chameleon, para a arquitetura SPARC (processador Leon).

17. Obtém-se o novo arquivo binário, adaptado para execução no novo processador. Com o uso das ferramentas de ArchC (comentada no item 10), é possível ainda simular as novas instruções para obter dados dos ganhos de desempenho sem a necessidade da implementação física do processador modificado.

Versões iniciais de todas as ferramentas que permitem a execução do processo acima descrito já foram implementadas, tornando possível testar o ciclo completo de desenvolvimento. No entanto, a obtenção de resultados significativos de *speedup* ainda depende de mais algum trabalho, especialmente na produção do *hardware* — por exemplo, ainda temos limitações no número de ciclos e no acesso à memória feitos pelos padrões implementados no *pipeline* do processador.

2.2. Outros trabalhos em reconfiguração/especialização de arquiteturas

Muitos estudos têm sido desenvolvidos na área de reconfiguração ou especialização de arquiteturas [CH02, MH97, HA96]. Os modelos de reconfiguração do *hardware* variam bastante, indo desde arquiteturas simples que permitem apenas modificar o estágio de execução do *pipeline*, como o processador PRISC [RS94] e o processador comercial Nios [Nios04], até arquiteturas virtuais que permitem a reconfiguração completa de todos os estágios, como PipeRench [SWT+02]. A seguir são descritos diversos projetos baseados em arquiteturas reconfiguráveis que estão em algum aspecto relacionados ao Projeto Chameleon. A Tabela 1 apresenta um quadro comparativo dos projetos citados, analisando em cada coluna:

- Tipo da reconfiguração. Se *estática*, significa que, a princípio, propõe-se programar as unidades reconfiguráveis uma única vez antes da execução de uma aplicação, ou mesmo utilizar unidades funcionais especializadas que não sejam reconfiguráveis. Se *dinâmica*, significa que a unidade pode ser reconfigurada durante a execução da aplicação. Esta coluna também relata se o processador base utilizado nas implementações é RISC, CISC ou um microcontrolador;

- Abrangência da reconfiguração: se a reconfiguração ou especialização da arquitetura envolve apenas um ou alguns estágios do pipeline, ou se consiste no uso de um coprocessador ou unidade anexa ao barramento de dados/endereços do processador principal;

- Forma de modificação do conjunto de instruções do processdor base. Se *incremental*, significa que uma ou mais instruções são adicionadas ao conjunto de instruções original. Se *completa*, significa uma reformulação de todo o conjunto de instruções. Se *pré-definidas*, significa que um ou mais *opcodes* são pré-alocados com significados específicos, como por exemplo "incie a reconfiguração" (neste caso, as computações distintas são identificadas apenas por campos do *opcode*);

- Forma de modificação dos compiladores e *assemblers* para que possam utilizar as novas instruções, se *automatizada* ou *manual*. Se manual, possivelmente significa que o usuário deve inserir código *assembly in line* em meio ao código C. Esta coluna também contém referências ao compilador utilizado, quando esta informação estava disponível nas publicações;

- Forma de identificação dos padrões de código a serem implementados em *hardware*: *automatizada* ou *manual*. Também referencia a extensão da identificação: se dentro de blocos básicos, ou a partir de um CFG ou DFG global, ou a partir do código *assembly* executável;

- Geração de código HDL para os padrões: *manual* ou *automatizada*.

Ainda na Tabela 1, a abreviação *N.A.* significa *não se aplica* (ou está fora do escopo da publicação), *b.b.* significa *bloco básico*, *c.* significa *compilador*, *d.d.* significa *dependências de dados* e um ponto de interrogação (?) significa que a publicação do trabalho deixava a informação implícita mas

Proposta	***Reconfiguração Est./Din.***	***Abrangência da Reconfiguração***	***Modificação do Instruction Set***	***Modificação de compiladores, assemblers***	***Identificação de padrões para implementação em hw***	***Geração de código HDL para os padrões***
Chameleon (LSC)	Estática, proc. RISC	Estágio de Execução	Incremental, automatizada	Automatizada, c. GCC	Automatizada, além de b.b.	Manual
[CFHZ04]	Estática, proc. RISC	Estágio de Execução	Incremental, automatizada	Automatizada, c. *SUIF*	Automatizada. Padrões pequenos.	Manual
[SRRJ02]	Estática, proc. RISC	Estágio de Execução	Incremental *(Tensilica)*	Manual *(assembly in line)*	Automatizada, apenas d.d.	Automatizada *(Tensilica)*
[AC01]	Estática, proc. VLIW	N.A.	Incremental	Automatizada, c. *Move*	Automatizada, dentro de b.b.	N.A.
[API03]	Estática, proc. VLIW	N.A.	Incremental, automatizada	Automatizada, c. *Machine SUIF*	Automatizada, dentro de b.b.	N.A.
[CKY+99]	Estática, DSP	N.A.	Incremental, automatizada	N.A.	Automatizada, *μ-operations*	N.A.
[CZTM03]	Estática, proc. VLIW	Estágio de Execução	Incremental, automatizada	Automatizada, c. *Trimaran*	Automatizada, apenas d.d.	Manual
[KKMB02]	Estática	N.A.	Incremental, automatizada	Automatizada, c. *SUIF*	Automatizada, apenas d.d.	N.A.
[HD95]	Estática, proc. RISC	N.A.	Todas as instruções	Automatizada, c. *Aquarius Prolog*	Automatizada, dentro de b.b.	Manual
[PGLM94]	Estática, DSP	N.A.	Todas as instruções	Automatizada	Automatizada, *μ-operations*	N.A.
[LLS+00] MorphoSys	Dinâmica, proc. RISC	Execução e Memória	Instruções pré-definidas	Manual *(assembly in line)*	Manual	N.A.
[VNK+03]	Dinâmica, proc. RISC	Execução e Memória	Instruções pré-definidas	Automatizada	Automatizada, linguagem SA-C	N.A.
[LCD03]	Dinâmica, proc. RISC	Execução e Memória	Instruções pré-def. (?)	N.A.	Automatizada, *tree covering*	N.A.
[SWT+02] PipeRench	Dinâmica	Todo o *datapath*	N.A.	N.A.	N.A.	N.A.
[AS93] PRISM	Dinâmica, proc. CISC	Unidade anexa	Instruções pré-definidas	N.A.	Manual	Automatizada (?)
[RS94] PRISC	Dinâmica, proc. RISC	Estágio de Execução	Instrução pré-definida	N.A. (Partem do código *assembly*)	Automatizada	Automatizada
[HW97] Garp	Dinâmica, proc. RISC	Execução e Memória	Instruções pré-definidas	Manual *(assembly in line)*	Manual	Manual
[HFHK97] Chimaera	Dinâmica, proc. RISC	Estágio de Execução	Incremental, manual	Manual *(assembly)*	Manual	Manual
[BA97] Wormhole	Dinâmica	Todo o *datapath*	N.A.	N.A.	N.A.	N.A.

Tabela 1 - Quadro comparativo dos trabalhos relacionados comentados.

sem referências para confirmação.

[CFHZ04] Application-Specific Instruction Generation for Configurable Processor Architectures

Em [CFHZ04], Cong *et alii* utilizam a plataforma comercial Nios [Nios04] da empresa Altera, que consiste em um processador RISC com um estágio de execução preparado para a inserção de uma unidade funcional especializada (em FPGA) que pode implementar até 5 novas instruções – cada instrução com dois operandos de entrada e um operando de saída. Eles trabalham com o compilador SUIF [SUIF04], com o qual produzem um DAG que representa o código da aplicação e do qual extraem padrões a partir de dependências de dados. Não fica claro se dependências de controle são consideradas, mas estruturas como *if-then-else* cujas computações sejam independentes quanto a dados parecem não ser abrangidas. (De qualquer modo, tais construções certamente não poderiam ser implementadas com a restrição de duas entradas e uma saída do Nios.) A seleção dos melhores padrões para implementação em *hardware* é feita com a atribuição de um ganho (equivalente ao peso no Projeto Chameleon), que é proporcial ao número economizado de ciclos e ao número de ocorrências do padrão, determinado por comparações de grafos feitas com o pacote de funções de isomorfismo Nauty [Nauty04]. Aparentemente, o cômputo do ganho não leva em conta a freqüência de execução do padrão que poderia ser obtida por *profiling* das aplicações, o que pode levar à escolha de padrões que sejam raramente ou nunca executados na prática. Para o mapeamento dos padrões escolhidos sobre o código da aplicação, formulam o problema como uma instância de *binate covering*, um problema NP-difícil que argumentam poder ser resolvido de forma exata na prática para o tamanho das entradas (padrões). Afirmam terem obtido *speedup* médio de 2.75 para o conjunto de programas do *benchmark* DSP-Stone [ZVSM94], que no entanto é um pacote com programas bastante pequenos que se concentram no núcleo de computações para DSP.

[SRRJ02] Synthesis of Custom Processors based on Extensible Platforms

A plataforma comercial Xtensa da empresa Tensilica [Tensilica04] é utilizada por Sun *et alii* em [SRRJ02]. A plataforma Xtensa contém um modelo HDL em Verilog de um processador RISC e utiliza a linguagem proprietária TIE para descrever o código a ser implementado por uma nova instrução adicionada ao processador. Outra ferramenta da plataforma converte a descrição TIE para código Verilog que corresponderá a uma nova unidade funcional adicionada ao estágio de execução do processador. Os autores concentram sua contribuição na seleção de padrões código (que eles chamam de *templates*) que serão transformados nas novas instruções do processador. O método que propõem parte do *program dependence graph* no nível de código C (obtido pela ferramenta [Aristotle]) para identificar dependências de dados entre operações. Assim, seus *templates* são crescidos por dependências de dados, e não incluem todas as possíveis construções de controle (nem todos os *ifs* ou *switches* são cobertos). São atribuídos pesos aos *templates* proporcionais à fração do tempo de execução que consomem (obtida por *profiling*), e inversamente proporcionais ao número de ciclos que a nova instrução requer – o que, aparentemente, acaba priorizando os menores *templates*. O artigo afirma que o *templates* mais promissores são simulados para identificar o número de ciclos requeridos pela nova instrução, mas não deixa claro como os padrões mais promissores são selecionados se, para a aplicação da fórmula de peso que propõem, é necessário já ter executado a simulação. Para a seleção final das instruções candidatas, propõem uma formulação em programação linear inteira, resolvida por algoritmo *branch-and-bound*, que considera o número de ciclos da nova instrução, o aumento no

período do clock e o aumento em área do processador. Como resultados, apresentam medidas de *speedup* e redução do produto *energia·delay* que chegaram, respectivamente, a 5.4 e 24.2 para 6 programas escolhidos de forma aparentemente arbitrária.

[AC01] Designing Domain-Specific Processors

Em [AC01], Arnold e Corporaal focalizam a extração e casamento de padrões realizada num compilador denominado *Move* antes da etapa de seleção de instruções. Não realizam uma implementação prática em *hardware*, mas assumem um modelo de uma arquitetura VLIW composta de unidades funcionais interconectadas por um barramento (modelo *coarse-grained)*. Os padrões são extraídos a partir do *execution trace* de aplicações obtido por simulação, e aplicam um algoritmo de casamento de DAGs *(directed acyclic graphs)* para identificar padrões repetidos ao mesmo tempo em que eles são extraídos. Os padrões considerados se limitam ao interior de blocos básicos e são conectados por dependências de dados, restringindo sensivelmente o paralelismo que podem encontrar. Uma vez determinado um universo de padrões, é aplicado um algoritmo do tipo programação dinâmica para realizar uma cobertura de custo mínimo sobre o código da aplicação, semelhante aos algoritmos de seleção de instruções de compiladores. A decisão de quais padrões transformar em unidades funcionais no *hardware* foi feita manualmente, a partir de uma categorização dos padrões que identificou 5 novas UFs que realizam operações lógicas, aritméticas (incluindo ponto flutuante) e acesso à memória. No entanto, esta identificação foi feita de forma teórica e as UFs não foram realmente implementadas.

[API03] Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints

Atuso *et alia* focalizam seu trabalho em [API03] no método de extração e seleção de padrões de código. Tratam os padrões como cortes *(cuts)* de grafos de fluxo de dados que podem ser desconexos, mas que se limitam ao interior de blocos básicos (e portanto não abrangem estruturas como *if-then-else*). Observam que o número de possíveis cortes num grafo de $|V|$ vértices de um bloco básico é $2^{|V|}$, e propõem um método de poda do universo de possibilidades que exclui os cortes ditos desconexos (que não podem ser substituídos por uma nova instrução no código da aplicação) e aqueles que violam restrições no número máximo de entradas e saídas admitidas pelas instruções do processador. Para um corte *S*, assumem que exista uma função *M(S)* que atribua um peso ao corte proporcional ao *speedup* que ele ofereça, mas não apresentam uma implementação para *M*. Um algoritmo iterativo para a seleção dos padrões é apresentado, que oferece resultados melhores que os de outros dois algoritmos previamente propostos *(Clubbing* e *MaxMISO)*. Utilizaram o *back-end Machine SUIF* [SH00] para a extração dos padrões, e não realizaram implementações em processadores reais.

[CKY+99] Synthesis of Application Specific Instructions for Embedded DSP Software

Em [CKY+99], Choi *et alii* propõem um método para a geração de novas instruções para arquiteturas DSP microprogramadas *(μ-program)* nas quais as próprias instruções originais (denominadas MOPs – *μ*-operations) já são capazes de executar até duas operações especificadas num mesmo *opcode*, com operandos provenientes de duas memórias (RAM) separadas. O método proposto

consiste em, primeiramente, converter o código fonte em uma lista de MOPs (não citam o compilador que utilizam para isto) e realizar um *profiling* para determinar as MOPs mais executadas. Depois, realizam o casamento das MOPs com três classes de instruções: *P-class*, de instruções primitivas de um ciclo como lógicas, aritméticas e de controle, *C-class* de instruções que consistem na combinação de algumas instruções *P-class* mas utilizando apenas os recursos originais de microprogramação da arquitetura DSP, e finalmente *S-class* de novas instruções implementadas por unidades funcionais especiais agregadas ao processador. A seleção de instruções parte de uma restrição de tempo T_c requerida pelo usuário, e de uma estimativa T_p do tempo necessário para executar a aplicação utilizando instruções *P-class*. Aplica-se então um algoritmo que tentará gerar instruções *C-class* até que o ganho supere a diferença $Td = Tp - Tc$. Se as instruções *C-class* não forem suficiente para atingir o ganho necessário, parte-se para a produção de instruções *S-class*. O algoritmo é baseado no problema NP-hard *subset-sum*, e tenta maximizar a cobertura das novas instruções sobre o código e minimizar o número de novas instruções geradas.

[CZTM03] Automatic Design of Application Specific Instruction Set Extensions through Dataflow Graph Exploration

Os autores de [CZTM03] partem de um código *assembly* otimizado, não escalonado e não alocado, gerado pelo compilador Trimaran [Trimaran04], e constroem grafos de fluxo de dados (DFGs) do qual extraem padrões que denominam apenas de *subgrafos*. A construção dos subgrafos parte de um nó do DFG que é crescido percorrendo arestas numa direção decidida por uma *função guia*. Esta função leva em conta itens como caminho crítico (prefere-se crescer na direção do caminho crítico), latência da nova instrução (procura-se obter padrões que mais reduzam o número de ciclos), área da nova unidade funcional (padrões que requeiram menor área são preferidos) e número de entradas e saídas (prefere-se crescer numa direção que não aumente o número de entradas e saídas do padrão). Os subgrafos assim obtidos são então casados para encontrar sobreposições de operações, marcando padrões que estão contidos em outros padrões (e que portanto teriam custo zero de aproveitamento se o padrão maior for implementado em *hardware*), e ainda marcando padrões coringa que diferem entre si por apenas uma operação (de modo que o custo de implementar um dos padrões seja muito baixo se o outro tiver sido escolhido para implementação). A seleção final dos padrões é feita por um algoritmo guloso que leva em conta uma relação valor/custo atribuída a cada padrão. Esta relação é atualizada para cada padrão que é afetado pela escolha de um outro padrão – o custo de se implementar um padrão pode ser reduzido se determinados outros padrões forem escolhidos. Resultados experimentais foram obtidos aplicando este método a uma arquitetura VLIW que executa quatro operações por ciclo, e foram obtidos *speedups* de até 1.87.

[KKMB02] Instruction Generation for Hybrid Reconfigurable Systems

O artigo [KKMB02] apresenta um algoritmo relativamente genérico para a geração e o casamento de padrões (que denominam *templates*) simultaneamente, a partir do grafo de fluxo de dados e controle da aplicação (CDFG). Definem como objetivos a minimização do número de *templates* gerados e do número de instâncias de cada *template*, ao mesmo tempo em que a cobertura do grafo pelos *templates* deve ser maximizada. Tratando-se de um problema NP-completo, propõem um algoritmo heurístico baseado na contração de arestas do CDFG, gerando um novo nó a partir de dois antigos nós. Os *supernós* assim gerados são os *templates* candidatos à implementação em

hardware. O algoritmo é tal que leva à produção de *templates* bem pequenos (por exemplo, de duas instruções) que rapidamente são capazes de cobrir todo o grafo de fluxo de dados, de forma que com uma média de 22 templates cobriram 82% do CDFG de vários programas de *benchmarks* (em média, 17% do CDFG era composto de nós isolados que não poderiam ser cobertos pela técnica usada). Embora os autores coloquem estes dados como um mérito, questionamos a eficácia da abordagem na medida em que tais pequenos *templates* resultarão em pequenos ganhos de desempenho numa implementação real. (O artigo não apresenta dados de implementações reais, com valores de *speedup*.) Outro fato que o artigo apresenta como uma vantagem da qual discordamos é a inexistência de uma restrição quanto ao número de entradas e saídas de padrões – enquanto essa restrição é irrelevante ao algoritmo de identificação de padrões, ela se torna crucial se uma implementação na forma de novas instruções do processador for desejada. O algoritmo também não leva em conta a freqüência de execução dos *templates*, levando potencialmente à seleção de *templates* pouco executados. Por outro lado, duas interessantes observações feitas no artigo, com as quais concordamos, dão conta de que:

- A geração de código para sistemas reconfiguráveis requer compiladores capazes não apenas de selecionar instruções mas também de gerar instruções para a implementação de um código de alto nível (como C) – o que corresponde, respectivamente, à extração e casamento de padrões;

- A geração de código para sistemas reconfiguráveis esbarra numa dificuldade também encontrada na geração de código para arquiteturas VLIW superescalares: o baixo paralelismo exposto no código de aplicações, sendo proveitoso recorrer a técnicas desenvolvidas para aquelas arquiteturas como execução predicada e construção de hiperblocos.

[HD95] Synthesis of Application Specific Instruction Sets

Em [HD95], Huang e Despain propõem um método para a geração automatizada de um conjunto de instruções *completo* que represente a melhor integração entre um processador RISC *pipelined* e uma aplicação específica. O processador RISC assumido possui 6 estágios no pipeline e suas capacidades são descritas em termos de MOPs (micro-operações) que especificam as conexões entre módulos do *datapath* e funcionalidades como cópia de valores entre registradores, atribuição de constantes, leitura e escrita da memória, incremento do *program counter*, etc. A partir da representação intermediária de um compilador para a aplicação específica, geram grafos de fluxo de dados e controle (CDFGs) para cada bloco básico. As operações computadas em cada CDFG são traduzidas em termos das micro-operações do processador, que são então escalonadas. Um algoritmo baseado em *simulated annealing* é utilizado para reduzir o custo de uma função objetivo, partindo de um estado inicial representado por um escalonamento das micro-operações da aplicação. Um compilador para Prolog (Aquarius Prolog Compiler [Roy90]) utilizado para a produção dos CDFGs e para a geração de código *assembly* referente ao novo conjunto de instruções. Resultados experimentais foram obtidos para quatro aplicações Prolog e comparados contra os resultados da implementação manual de uma extensão de um conjunto de instruções RISC para execução eficiente de aplicações Prolog feita no projeto Aquarius [Hol90]. A síntese automática de instruções resultou em código *assembly* até 17% mais compacto com utilização menor ou igual de recursos de *hardware* (número de portas de leitura e escrita no banco de registradores e na memória RAM), em comparação à implementação manual.

[PGLM94] Instruction Set Definition and Instruction Selection for ASIPs

O trabalho descrito em [PGLM94] também trata da seleção de instruções para ASIPs e da geração automatizada de um conjunto de instruções *completo* para aplicações específicas, abordando especialmente arquiteturas DSP que não são regulares como processadores RISC. Partem de uma descrição da micro-arquitetura do processador (por exemplo, a existência de conexões diretas entre ALUs e *shifters* ou entre multiplicadores e somadores em DSPs) que mapeiam sobre as operações executadas pela aplicação específica representada na forma de um CDFG (grafo fluxo de dados e controle). Uma vez mapeadas as micro-operações sobre o CDFG da aplicação, executam um método que denominaram *bundling* e que consiste em identificar seqüências maximais de micro-operações associadas a unidades funcionais que possuem ligação direta entre si (através de barramentos ou *latches*). Estas seqüências, isoladas ou em conjunto a outras seqüências maximais, farão posteriormente parte de instruções projetadas para a arquitetura. Uma ferramenta executada sobre o CDFG obtém estatísticas da freqüência de ocorrência de operações da aplicação e de *bundles* (seqüências maximais de micro-operações), que orientam o projetista na determinação do conjunto de instruções. Não fornecem detalhes sobre o compilador utilizado.

[LLS+00] MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications

O projeto MorphoSys [LLS+00] focaliza a extensão de um processador RISC (chamado *TinyRISC*) para conter um *array* de 8x8 unidades reconfiguráveis com vistas a acelerar a execução de algoritmos para aplicações gráficas, como *motion estimation*. As unidades reconfiguráveis são *coarse grained*, sendo todas iguais e dotadas de uma ALU de 16 *bits* (capaz de multiplicação), dois multiplexadores que selecionam as entradas da ALU, uma unidade de deslocamento *(shift),* um banco local de quatro registradores e um registrador de contexto que define a operação a ser executada na unidade e armazena uma constante de 12 bits. O *array* 8x8 é fisicamente construído de modo a permitir o "*broadcast"* de dados que são carregados nos registradores de todas as unidades de uma linha ou coluna, uma abordagem SIMD. Resultados experimentais foram obtidos através da simulação de um modelo VHDL do processador, que executou um trecho de código C gerado por uma versão modificada do compilador *SUIF* [SUIF04]. No entanto, a implementação requer que o usuário insira instruções *assembly in line* no código C, e não há qualquer suporte à identificação automática de padrões. Para três aplicações gráficas, a arquitetura resultou em desempenho comparável ao de implementações ASIC, utilizando até dez vezes menos ciclos dos que implementações baseadas apenas em *software*. Os autores realizaram também um projeto físico VLSI da arquitetura.

[VNK+03] Automatic Compilation to a Coarse-Grained Reconfigurable System-on-Chip

O trabalho [VNK+03] apresenta uma infraestrutura de compilação focada na linguagem de programação SA-C [HB01, HRB+99] tendo como alvo a arquitetura *MorphoSys* [LLS+00] previamente discutida. A linguagem SA-C foi projetada para explicitar o paralelismo das operações do código de aplicações, e tem características como *single-assignment* (toda sentença é uma atribuição, funções podem retornar vários valores e variáveis só podem ser atribuídas uma vez), suporte a vetores multidimensionais (acesso direto a colunas, linhas, planos de *arrays*), operadores pré-definidos para

processamento de imagem com operações sobre vetores, laços com construções mapeadas no *hardware* e outras características. Diversos algoritmos e técnicas foram desenvolvidos para mapear construções de SA-C sobre as unidades funcionais oferecidas pela arquitetura MorphoSys, levando a um novo compilador para SA-C. Ganhos de até 8 vezes em tempo de execução foram obtidos para aplicações gráficas executadas em um Pentium III a 800MHz em comparação com o processador de MorphoSys executando a 200 MHz.

[LCD03] An Algorithm For Mapping Loops Onto Coarse-Grained Reconfigurable Architectures

RAA – *Reconfigurable Array Architectures* são arquiteturas reconfiguráveis *coarse-grained* caracterizadas pela disposição das unidades funcionais em linha e colunas interconectadas por barramentos verticais e horizontais compartilhados, como a arquitetura MorphSys previamente citada. Em [LCD03], Lee *et alii* afirmam que arquiteturas RAA oferecem grande ganho de desempenho mas não dispõem de ferramentas de compilação que permitam o mapeamento do código da aplicação com qualidade suficiente para aproveitar ao máximo o paralelismo e o acesso a memória feito diretamente pelas linhas e colunas dos elementos do *array*. Neste sentido, propõem um algoritmo de mapeamento de laços em RAAs, valendo-se das técnicas de *loop pipelining* (várias iterações executadas simultaneamente num *pipeline*) e *data context switching* que elimina algumas dependências *loop-carried* se o laço estiver inserido num outro laço externo que não possua tais dependências. O mapeamento é realizado em três fases: na primeira fase, realizam uma cobertura do código do laço com as operações disponíveis nas unidades funcionais (que chamam de PEs – *Processing Elements*) de cada elemento do *array* (computações que não possam ser cobertas por PEs são executadas pelo processador principal de uso geral.) Na segunda fase, denominada *clustering*, identificam grupos *(clusters)* de PEs que possam ser mapeados numa única linha do *array*. Estes *clusters* possuem a restrição de um único valor computado de saída, para reduzir a complexidade e facilitar a transferência de dados entre *clusters*. Na terceira fase, as transferências de dados entre *clusters* são mapeadas para os recursos de roteamento (barramentos) da arquitetura. Argumentos são apresentados para justificar que a complexidade de tempo do procedimento é tratável. Realizaram experimentos para obter dados sobre as interconexões necessárias entre PEs e a taxa de utilização da banda de memória disponível. Comparações com otimizações manuais sugeriram que o método proposto produz mapeamentos próximos do ótimo para os laços testados.

[SWT+02] PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology

PipeRench é um *pipeline* em que todos os estágios são reconfiguráveis. Os autores distinguem um *pipeline* virtual, com vários estágios e funções arbitrárias, do *pipeline* físico que pode ter um menor número de estágios – que são chamados de *stripes* (tiras). Cada *stripe* físico é formado por 16 PEs *(Processing Elements)*, e cada PE contém 8 registradores de 8 *bits* e um bloco lógico programável baseado em SRAM. A configuração de um *stripe* utiliza 672 bits. Antes do início da execução de uma aplicação específica, os *bits* de configurações para todos os estágios virtuais que ela requeira são armazenados numa memória de configuração no *chip*. Quando a execução inicia, os *bits* de configuração trafegam pelos estágios junto aos *bits* de dados. Quando um *stripe* físico precisa ser reconfigurado para implementar um *stripe* virtual, seu estado (valor dos registradores) é salvo numa memória de estado também localizada no *chip*. A entrada e saída de dados entre o *pipeline* PipeRench e o exterior é feita através de memórias FIFO. Uma implementação real do *pipeline* foi realizada com

tecnologia 0.18 μm e o desempenho para as aplicações reportadas foi comparável ao de DSPs comerciais (ASICs), e muito superior ao de processadores de uso geral. O artigo não trata da adaptação de compiladores para geração automática de código para a aquitetura PipeRench, nem aborda o procedimento utilizado para a execução do código.

[AS93] PRISM – Processor Reconfiguration Through Instruction-Set Metamorphosis

O projeto PRISM foi um dos pioneiros a propor uma plataforma de *hardware* e *software* integrada na qual cabe ao compilador extrair padrões de códigos a serem executados em novas unidades funcionais adicionadas ao processador central. A implementação apresentada como *proof-of-concept* consistia de um microcontrolador CISC interconectado a quatro FPGAs por um barramento de 16 bits (as FPGAs estavam numa placa distinta). A transferência de dados entre o processador e as FPGAs consumia de 48 a 72 ciclos, mas mesmo assim foram reportados *speedups* de até 54 vezes para funções inteiramente implementadas nas FPGAs (desconsiderado o tempo de configuração das mesmas). Aparentemente, a geração de código HDL para síntese das funções nas FPGAs foi realizada de forma manual, embora o artigo advogue na direção de técnicas automáticas para a esta tarefa.

[RS94] PRISC – PRogrammable Instruction Set Computers

O trabalho PRISC foi um dos primeiros a apresentar um fluxograma completamente automatizado para a especialização de um processador RISC para uma aplicação específica, embora não tenham realizado uma implementação física. O *hardware* proposto consiste de um processador MIPS R2000 em cujo estágio de execução é acoplada uma unidade funcional (UF) reconfigurável do tipo *look-up table* (baseada em SRAM), capaz de implementar operações lógicas. Alocaram um *opcode* do conjunto de instruções MIPS para disparar a execução da UF. Este *opcode* possui um campo de 11 bits que seleciona uma operação lógica pré-definida a ser executada; se a operação solicitada não for a programada no momento na UF, uma exceção *(trap)* é gerada para disparar a reconfiguração da unidade. As imagens *(bitstreams)* para reconfiguração da UF são armazenadas no próprio segmento de dados do arquivo binário executável da aplicação. A identificação dos padrões a serem implementados na UF parte de uma análise do *assembly* final da aplicação anotado com dados de *profiling* para identificar o código mais executado. Apenas conjuntos seqüenciais de instruções são considerados, embora sejam feitas análises e otimizações sobre estas seqüências para identificar, por exemplo, estruturas *if-then-else* e *switch* auto-contidas. Os padrões escolhidos não podem ter mais do que duas entradas e uma saída. As simulações realizadas mostraram *speedups* de até 1.91 considerado o tempo global de execução de aplicações do pacote de *benchmark* SPECint92 [SPEC92].

[HW97] Garp – A MIPS Processor with a Reconfigurable Coprocessor

Em Garp, Hauser e Wawrzynek tratam do projeto de um processador MIPS-II juntamente a um *array* de unidades reconfiguráveis que seriam implementados num mesmo *die*. O *array* é reconfigurado dinamicamente pelo programa em execução no processador principal. Não há provisão para a identificação automática de padrões, cabendo ao usuário identificar trechos de código da aplicação que sejam executados eficientemente no *array* de unidades reconfiguráveis. Uma ferramenta foi projetada que lê um arquivo texto com a descrição da configuração do array feita pelo usuário e

produz um declaração em código C de um array com o *bitstream* da configuração. O usuário deve ainda inserir código *assembly in line* no programa C para disparar a configuração e a execução das unidades funcionais. *Speedups* de até 24 vezes foram obtidos num conjunto de aplicações simulado.

[HFHK97] The Chimaera Reconfigurable Functional Unit

A proposta de arquitetura reconfigurável Chimaera foi uma das primeiras a sugerir uma unidade reconfigurável acoplada ao processador principal com acesso direto ao banco de registradores (provavelmente implementada na mesma pastilha de silício). Em [HFHK97], Hauch *et alii* descrevem uma arquitetura dinamicamente reconfigurável em que várias instruções podem estar simultaneamente implementadas numa unidade do tipo *fine-grained*. Propõem que, se uma instrução especial for executada e a computação correspondente não estiver implementada na unidade funcional, o processador seja paralizado até que a informação de reconfiguração seja carregada da memória e a unidade seja reconfigurada, e sugerem a utilização de um *hardware* que permita a reconfiguração parcial apenas da lógica necessária à execução daquela instrução. Como o tempo de reconfiguração pode ser grande, propõem que a unidade reconfigurável seja utilizada como um *cache* de implementações de instruções especiais, que tais instruções sejam executadas de forma especulativa utilizando *shadow register files* e que técnicas sejam empregadas para reduzir ao máximo o *cache miss*, atentando para a freqüência e a ordem com que instruções especiais são inseridas em laços. No entanto, não apresentam algoritmos para automatizar estas tarefas, e os dados experimentais apresentados foram derivados de implementações manuais – modificação do código *assembly*. O processador base utilizado nos experimentos foi um processador MIPS R4000, e foram obtidos speedups de até 2.06.

[BA97] Wormhole Run-Time Reconfiguration

O trabalho apresentado em [BA97] difere substancialmente do modelo de unidades reconfiguráveis associadas a um processador principal. Bittner e Athanas sugerem o uso de uma arquitetura completamente reconfigurável, baseada no conceito de *streams*. Uma *stream* contém um cabeçalho com informações de reconfiguração do *datapath*, incluindo a reconfiguração do caminho dos dados *(crossbar routing)* e das unidades funcionais (dispostas num *array* no estilo *coarse-grained*). Em seguida ao cabeçalho aparecem os dados a serem processados (operandos), que trilham o caminho previamente configurado. Os autores sugerem ainda que o cabeçalho de reconfiguração possa conter uma identificação da computação sem os dados que reconfiguram o *hardware*, caso em que um gerenciador de reconfiguração, apoiado pelo sistema operacional, decidiria que *hardware* implementar com base nas demandas por recursos de múltiplos processos ou múltiplas *threads* de um mesmo processo. O artigo apresenta um exemplo de uma implementação de um multiplicador de ponto flutuante que atingiu o desempenho de 50 MFLOPS, mas não traz informações sobre a integração da arquitetura com ferramentas de compilação nem sobre a forma como o código executável seria mapeado para construir as *streams*.

Neste capítulo, o Projeto Chameleon e trabalhos relacionados foram apresentados. Numa comparação entre o Projeto Chameleon e os demais trabalhos, observamos que diversos dos trabalhos tratam apenas do lado do *hardware,* focando as formas de conexão e implementação de unidades especializadas reconfiguráveis e até mesmo projetos físicos VLSI, deixando a cargo do usuário a implementação de padrões em linguagens HDL e a inserção de código *assembly in line* em meio ao código de alto nível da aplicação. Já outros trabalhos cuidam apenas do lado do *software* (por exemplo, algoritmos para extração de padrões) e não realizam implementações reais. No próximo capítulo, é apresentada a Biblioteca de Padrões, a principal contribuição desta dissertação ao Projeto Chameleon, e os métodos que empregamos para a extração e a seleção de padrões de código para a implementação em *hardware*. Dito de modo bastante informal, a Biblioteca de Padrões é o núcleo da "parte *software*" do Projeto Chameleon.

Capítulo 3

Extração e Seleção de Padrões

O capítulo anterior apresentou uma introdução ao Projeto Chameleon. Neste capítulo, serão abordadas as etapas de extração, crescimento e seleção de padrões para a implementação em *hardware*, que correspondem às etapas 5 a 8 na Figura 6. Também será apresentada a *Biblioteca de Padrões*, um conjunto de funções e estruturas de dados para manipulação de padrões. Estes itens constituem as contribuições mais relevantes desta dissertação ao Projeto Chameleon.

O capítulo está dividido da seguinte forma: primeiramente, a seção 3.1 define mais precisamente o conceito de padrão de código e justifica sua representação na forma de grafos de fluxo de dados e de controle. Em seguida, a seção 3.2 descreve a forma implementada para a extração de padrões do código de aplicações e seu crescimento por dependências de dados e de controle. A seção 3.3 aborda o casamento (isomorfismo) de padrões. A seção 3.4 discute critérios adotados para a seleção dos melhores padrões dentre os candidatos à implementação em *hardware*. Finalmente, as seções 3.5 a 3.7 apresentam a implementação da Biblioteca de Padrões.

3.1. Padrões de Código

Definimos *padrões* de código como sendo trechos de código de programas, representados na forma de grafos de fluxo de dados e de controle (CDFGs – *Control and Data Flow Graphs* [GDWL93]), direcionados e acíclicos (DAGs – *Directed Acyclic Graphs* [ASU86]). A Figura 9 ilustra um grafo de fluxo de dados extraído do interior de um bloco básico. Detalhes da construção e semântica dos grafos serão abordados posteriormente.

A implementação em *hardware* de padrões representados como CDFGs, ao invés da simples consideração de trechos seqüenciais de instruções, tem o potencial de obtenção de maiores ganhos de desempenho porque existe maior flexibilidade na escolha de padrões que se repitam em várias partes do programa. Por exemplo, considere uma seqüência de instruções (código *assembly* ou a representação intermediária de um compilador) dentro de um laço que é executado 10^6 vezes. Poderíamos hipoteticamente implementar esta seqüência de instruções em *hardware*, estratégia que denominamos *gulosa*, e associar uma nova instrução do processador a ela. No entanto, dificilmente esta nova instrução seria utilizada em outra parte do programa, se para tanto fosse necessário encontrar uma outra seqüência idêntica de instruções (quanto maior a seqüência, mais difícil). Se, alternativamente, representarmos a seqüência de instruções como uma ou mais componentes conexas de um CDFG, será mais fácil encontrar outros trechos do programa que possuam o mesmo fluxo de dados e de controle. A razão para isto é que, em seqüências de instruções, freqüentemente existe

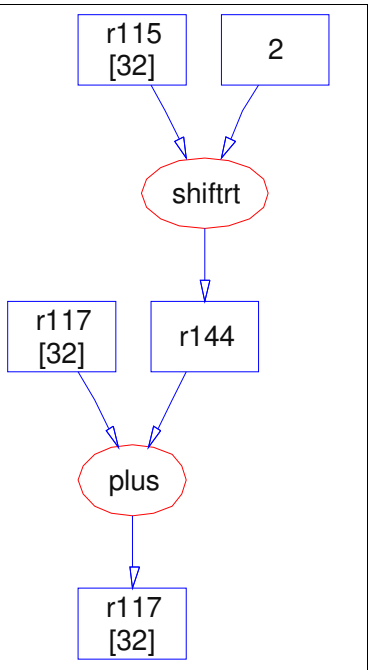


Figura 9 - Um padrão extraído automaticamente da representação intermediária do compilador GCC, exibindo um DFG do interior de um bloco básico da aplicação ADPCM (compressão e descompressão de voz). Os nós quadrados representam operandos, e os nós em forma de elipse representam operadores.

flexibilidade quanto à troca da ordem em que algumas instruções aparecem (independência quanto ao fluxo de dados), e também flexibilidade quanto ao bloco básico em que elas aparecem (independência quanto ao fluxo de controle). Utilizando um CDFG, podemos representar apenas as dependências estritamente necessárias de dados e controle. Além disso, instruções seqüenciais no código podem representar computações não necessariamente relacionadas (por exemplo, o cômputo de resultados independentes, que possuem dados de entrada diferentes), o que aumentaria indesejavelmente o número de registradores de entrada e saída necessários numa implementação em *hardware*. Com o uso de CDFGs, computações independentes acabam representadas como componentes desconexas do grafo, que podem ser escolhidas separadamente para a implementação no *hardware*, com uso de um número mínimo de registradores. Encontrar métodos para a divisão do código de um programa em padrões que resultem no melhor desempenho para a aplicação quando implementados em *hardware* é um dos desafios do Projeto *Chameleon*, e as abordagens implementadas são consideradas neste capítulo.

Além das dependências de dados e de controle expressas pelos grafos, cada padrão carrega ainda consigo informações sobre a localização de origem (nome do arquivo fonte, nome da função, números de blocos básicos e de RTXs conforme representação intermediária do GCC), freqüência de execução proveniente de *profiling*, representação do formato e mnemônico de uma nova instrução do processador associada ao padrão e códigos HDL de possíveis implementações na *datapath* do processador.

3.2. Obtenção e Crescimento de Padrões

No capítulo anterior, foi explicado o procedimento que utilizamos para obter as informações de *profiling* a partir da execução das aplicações no processador Leon. Uma vez que a freqüência de execução dos blocos básicos esteja disponível na representação intermediária do GCC (na segunda compilação da aplicação), procedemos à extração dos padrões, que são armazenados em arquivos ".pat" – cujo formato é discutido em detalhes em uma seção posterior. Inicialmente, extraímos um padrão para cada RTX da representação intermediária RTL do GCC, para cada função da aplicação que está sendo compilada[7]. Este padrões carregam consigo informações de dependência de dados armazenadas como *definition-use chains*[8], e informações de dependência de controle derivadas do grafo de fluxo de controle (CFG) da função. As razões para extrairmos padrões tão simples e pequenos neste ponto são duas: em primeiro lugar, extraindo todos estes padrões garantimos que nenhuma restrição é colocada sobre as possibilidades de crescimento e casamento de padrões das etapas seguintes e, em

7 O compilador GCC, assim como a maioria dos compiladores, realiza a otimização e geração de código por funções. O CFG, por exemplo, é gerado para cada função da aplicação. Por isso, na extração inicial de padrões, eles são agrupados por funções.

8 *definition-use chains* – informação de *reaching definitions* obtida da análise de fluxo de dados (DFA) [ASU86]

segundo lugar, reduzimos ao máximo a quantidade de código que inserimos no compilador GCC, de modo a minimizarmos o esforço de adaptação requerido quando novas versões do compilador forem produzidas pela comunidade de desenvolvedores. Uma vez que os arquivos de padrões tenham sido gerados, podemos aplicar ferramentas de transformação que denominamos *pat-to-pat*, em referência ao fato de as entradas e saídas serem arquivos ".pat", independentes do compilador. A Figura 10 ilustra esta idéia.

Apesar das razões existentes para produzirmos inicialmente padrões tão pequenos quanto uma única instrução, estes padrões não são interessantes para a implementação em *hardware*, visto que, por hipótese, o processador já é capaz de executá-los com alguma instrução do seu conjunto de instruções. Assim, realizamos um crescimento dos padrões, visando torná-los maiores. Intuitivamente, supondo que cada padrão será substituído por uma única instrução executada em um único ciclo, o ganho em número de ciclos executados será tanto maior quanto maior for o padrão. Obviamente, porém, esta seria uma visão bastante simplificada, pois não considera várias questões que influenciam no ganho de desempenho: o tamanho do *hardware* que pode ser implementado é limitado pela área de silício disponível e pode ter um custo econômico significativo, a implementação de um grande número de operações numa única instrução pode ter o efeito colateral de aumentar o período do ciclo do *clock*, e sempre existe a possibilidade de se criar novas instruções que requeiram vários ciclos para executar.

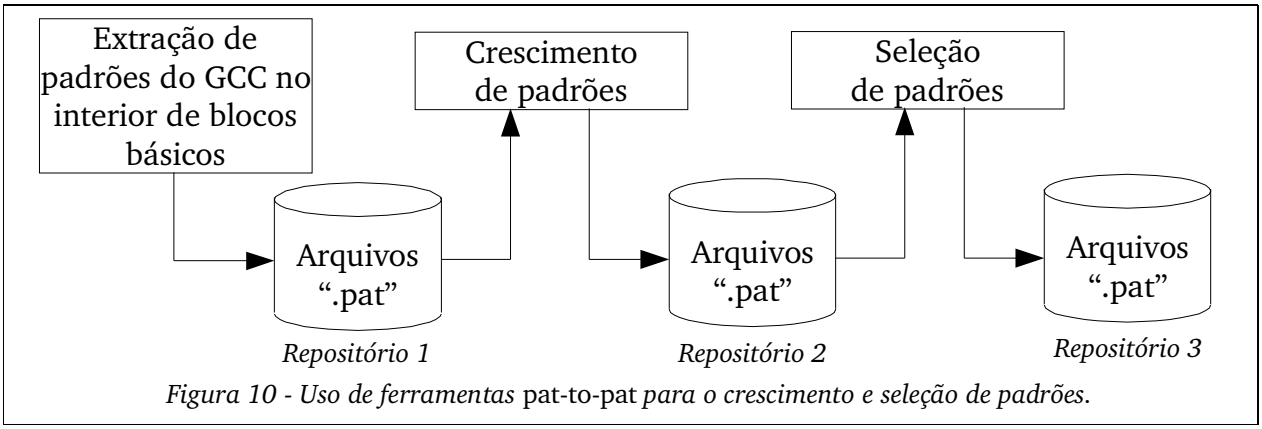


Figura 10 - Uso de ferramentas pat-to-pat *para o crescimento e seleção de padrões.*

Existem muitas formas de se efetuar o crescimento de padrões, levando a diferentes resultados. O número de padrões que se pode obter é exponencial no número de instruções do programa. Por exemplo, é possível imaginar um padrão para cada combinação das instruções do programa, duas a duas, três a três e assim sucessivamente. No entanto, além de computacionalmente infactível devido à explosão combinatorial, esta abordagem levaria à consideração de padrões que, de fato, não poderiam ser implementados em *hardware*, simplesmente porque não seria possível substituí-los por uma única instrução no código da aplicação devido a violações em dependências de dados de outras instruções. Numa seção adiante, este ponto é abordado no contexto de interferência entre padrões. A seguir, são apresentados e justificados os métodos que encontramos para obter padrões relevantes para a implementação em *hardware*.

Realizamos o crescimento de padrões em duas etapas. Na primeira etapa, o crescimento ocorre no interior de blocos básicos, seguindo dependências de dados na construção de um DFG. Buscamos obter DFGs maximais no interior dos blocos básicos, o que significa obter os maiores grafos de fluxo de dados conexos possíveis. A Figura 11 ilustra este procedimento. Considerando que, em geral, os blocos básicos são compostos de poucas instruções (uma média de 5 instruções por bloco para a arquitetura

SPARC [HL98]), os DFGs maximais também não costumam ser grandes. Apesar disso, observamos que, com certa freqüência, obtemos mais de um DFG maximal por bloco básico, evidenciando independência quanto ao fluxo de dados. Acreditamos que a consideração de padrões a partir de DFGs maximais no interior de blocos básicos seja interessante porque produz padrões pequenos que possuem implementação barata em *hardware* (requerem em geral poucos registradores de entrada e saída, e potencialmente poderiam ser executados em um único ciclo de máquina), e porque possuem maior potencial de ocorrerem repetidamente em diversas partes da aplicação. Esta estratégia também limita o número de diferentes padrões a quantidades computacionalmente tratáveis. No entanto, alguns experimentos também nos mostraram que estes padrões pequenos acabam levando a ganhos bastante modestos em tempo de execução. Por esta razão, implementamos uma segunda etapa de crescimento de padrões, além da fronteira de blocos básicos. Os padrões obtidos nesta segunda etapa de crescimento não substituem os obtidos na primeira etapa, mas sim somam-se a eles para uma fase posterior de casamento e seleção de padrões.

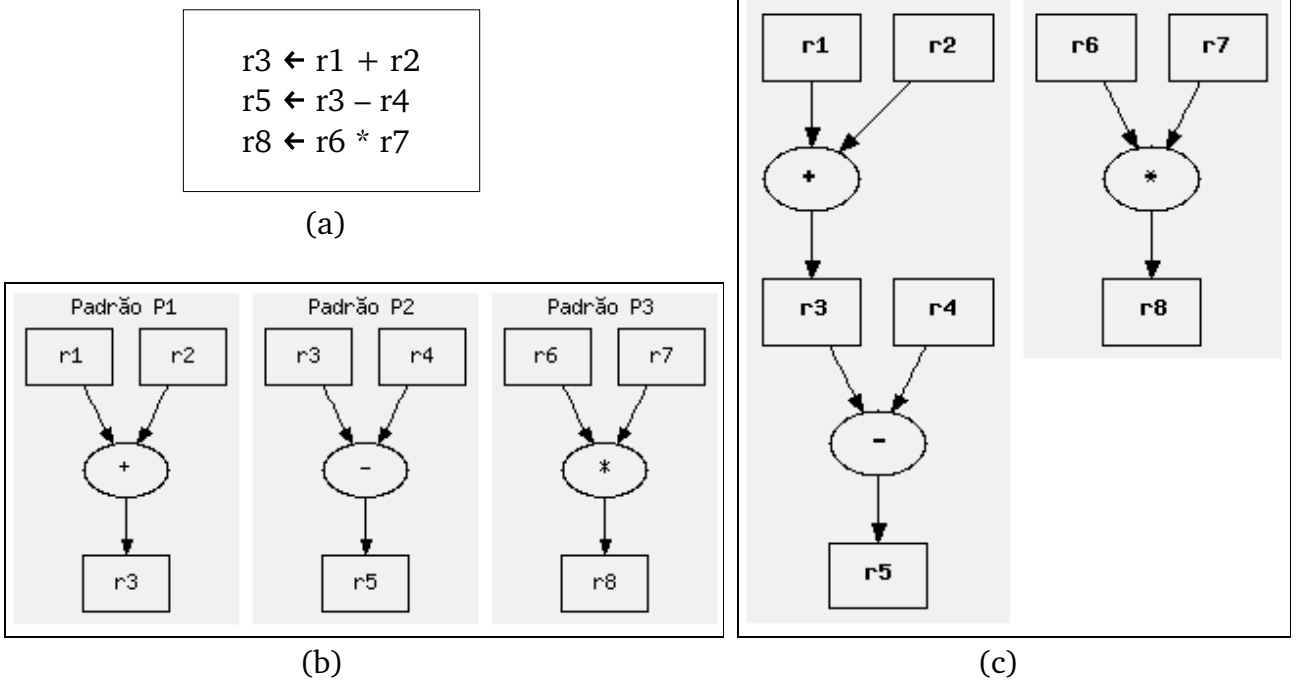


Figura 11 - Ilustração do crescimento de padrões no interior de blocos básicos. Em (a), um código no interior de um bloco básico; em (b), os padrões para cada instrução do código, da forma como são inicialmente extraídos da representação intermediária do compilador GCC; em (c), os padrões maximais correspondentes ao mesmo código, seguindo as dependências de dados.

Na segunda etapa do crescimento de padrões, o objetivo é obter subgrafos do CFG de uma função. A Figura 12 ilustra um típico padrão indicado para a fase posterior de seleção e casamento de padrões. Note-se que, neste caso, assumimos que a implementação em *hardware* será capaz de executar operações condicionalmente, com base no resultado de comparações (sentenças do tipo *if-then-else)*. Com estas implementações, os ganhos de desempenho que se pode obter podem ser bem maiores que no caso de padrões inteiramente contidos em um bloco básico. Na seleção de subgrafos, optamos por considerar apenas aqueles que possuam um único bloco básico de entrada e um único

bloco básico de saída. Neste caso, o bloco de entrada domina[9] os demais blocos do subgrafo, e o bloco de saída pós-domina[10] os mesmos. A principal razão para essa consideração é que, desta forma, todo o subgrafo pode ser implementado em *hardware* sem a preocupação de que um *branch* em outra parte do programa tenha como destino um bloco básico implementado em *hardware*, ou analogamente um *branch* de um bloco implementado em *hardware* tenha como destino um bloco não implementado em *hardware* – estes casos seriam como saltos do *software* para o *hardware* ou vice-versa, o que seria difícil de implementar e prejudicaria o desempenho. Além disso, com um subgrafo delimitado por dominadores e pós-dominadores, a nova instrução criada para executar o padrão em *hardware* poderá ser inserida no código da aplicação na posição originalmente ocupada por um destes blocos básicos, através de uma análise relativamente simples de dependências de dados. Nas ferramentas de extração de subgrafos do CFG implementadas, é possível indicar manualmente (por parâmetros de linha de comando) os blocos dominadores e pós-dominadores que delimitem um subgrafo, ou executar uma extração automática de todos os pares de dominadores e pós-dominadores imediatos existentes no CFG. A Figura 13 ilustra um trecho de um CFG com os pares dominadores e pós-dominadores imediatos coloridos. Estes grafos são automaticamente gerados para qualquer programa de entrada. Quando um padrão é crescido além de blocos básicos, sua freqüência de execução é computada como sendo o valor máximo das freqüências de cada um dos seus blocos básicos constituintes.

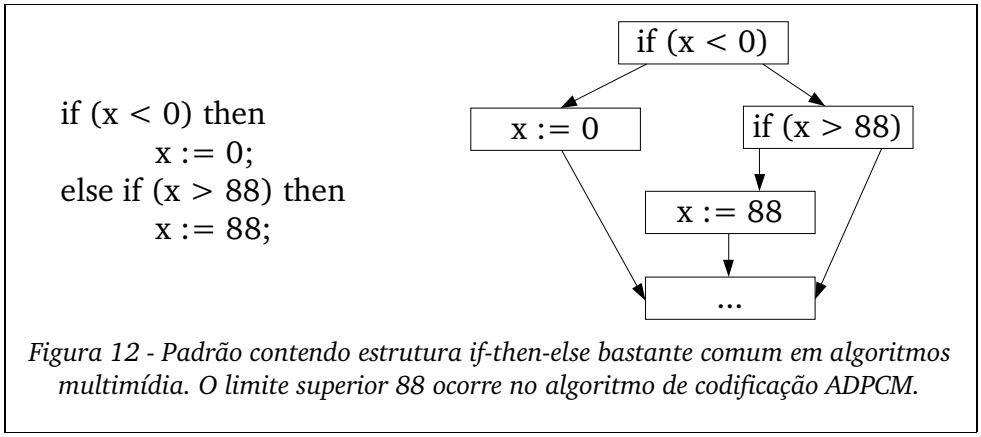


Figura 12 - Padrão contendo estrutura if-then-else bastante comum em algoritmos multimídia. O limite superior 88 ocorre no algoritmo de codificação ADPCM.

3.3. Casamento de Padrões – Isomorfismo de Grafos

No processo de seleção dos padrões candidatos à implementação em *hardware* (item 6 da Figura 6), é necessário saber se um padrão ocorre em outras partes do código de uma mesma ou de outras aplicações, para que se possa somar as freqüências de execução das diferentes ocorrências do padrão e assim atribuir-lhe um peso. Além disso, uma vez que se tenha decidido que um certo conjunto de padrões será implementado em *hardware*, é preciso conhecer a posição de todas as ocorrências do padrão no código das aplicações para que se possa editar o código, substituindo os padrões pelas novas instruções especializadas criadas no processador. Dado um padrão *P*, a

9 Dominador – Um bloco básico *A* domina um bloco básico *B* se todos os caminhos de execução *(traces)* do nó inicial do grafo até o nó *B* passam por *A*. O bloco *A* é dominador imediato de *B* quando é também o último dominador de *B* num caminho do nó inicial até *B*.

10 Pós-dominador – Um bloco básico *C* pós-domina um bloco básico *B* se todos os caminhos de execução *(traces)* de *B* até o bloco básico final do CFG passam por *C*. *C* é pós-dominador imediato de *B* se é também o primeiro pós-dominador de *B* num caminho entre *B* e o nó final.

determinação de sua ocorrência em diversas partes de um código é feita através da comparação dos grafos de fluxo de dados e controle (CDFGs) de *P* e dos demais padrões extraídos do código. Se os CDFGs de dois padrões são isomorfos ("iguais"), dizemos que ocorre o casamento dos padrões. Supondo que *P* tenha sido extraído através dos mesmos métodos que os demais padrões, então todas as outras ocorrências de *P* serão detectadas por esta abordagem. Por outro lado, claramente este procedimento não cobre todas as possíveis ocorrências de *qualquer* padrão; como já discutido, a consideração de todos os possíveis padrões resultaria numa explosão combinatorial. Acreditamos que os padrões considerados, quais sejam os DFGs maximais extraídos do interior de blocos básicos e os subgrafos do CFG limitados por nós dominadores e pós-dominadores, cobrem um conjunto significativo de padrões com características de potencial ganho de desempenho para a aplicação. Além disso, mesmo reduzindo o conjunto de possíveis padrões aos citados, a quantidade de padrões pode ser ainda suficientemente grande a ponto de ser necessário filtrá-los. A seguir, serão abordados os tipos de grafos usados para a comparação de padrões.

Como já discutido, a vantagem de se representar padrões na forma de grafos é expor explicitamente as dependências de dados e controle, o que não seria conseguido com a simples comparação de seqüências de instruções. O DFG expõe as dependências de dados entre as instruções. O CFG é o principal grafo no qual os compiladores baseiam as otimizações, e expõe o fluxo de controle entre as instruções da forma como o programador da aplicação a escreveu. Para representar as dependências de dados e controle num único grafo, foi implementado o CDFG – grafo de fluxo de dados e de controle.

Fluxo de controle não é o mesmo que *dependência de controle*, e a simples transposição das arestas de fluxo de controle de um CFG para o CDFG poderia resultar em diferentes CDFGs que, na verdade, representariam um mesmo padrão numa implementação em *hardware*. Como nosso objetivo é utilizar o CDFG para comparar padrões, desejamos que, tanto quanto possível, diferentes CDFGs impliquem em diferentes padrões, enquanto CDFGs isomorfos impliquem nas mesmas implementações em *hardware*. Neste sentido, o principal cuidado foi optar pelo uso de um CDG (*Control Dependence Graph*) [App98, FOW87] para fornecer as dependências de controle representadas no CDFG.

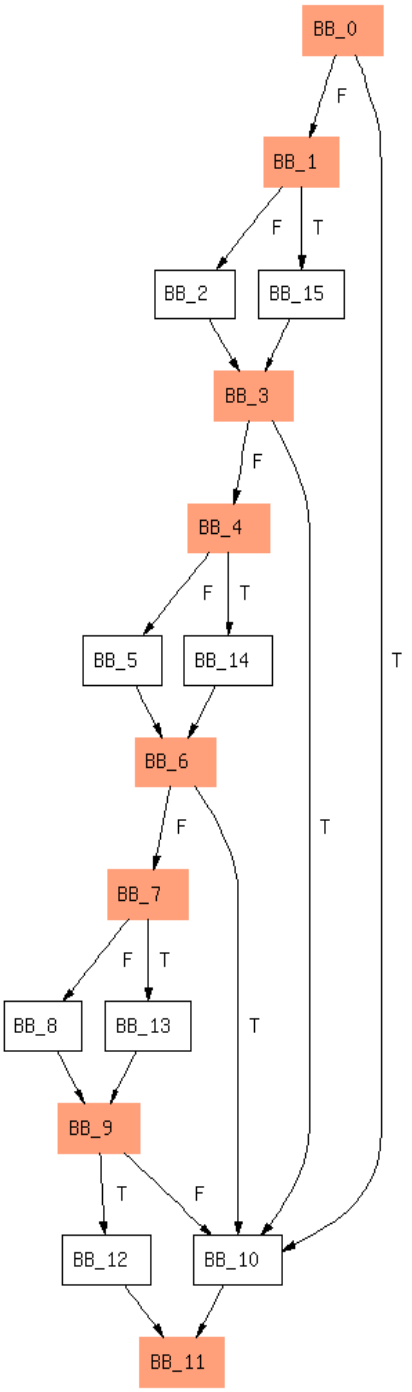


Figura 13 - Um trecho do CFG da função "audio_init_input" da aplicação GSM do pacote de benchmark MediaBench. Os nós coloridos identificam os pares dominador e pós-dominador imediatos: blocos 1-3, 4-6, 7-9 e 0-11.

A Figura 14 exibe dois CFGs com diferentes blocos básicos, e um CDG que é comum a ambos. Como pode ser observado, os dois CFGs representam, na verdade, a mesma computação, e os blocos básicos A e B diferem pelo simples fato de o programador ter trocado a posição da sentença que define r3 no código da aplicação. Se os CDFGs fossem criados com as arestas de fluxo de controle do CFG, a instrução "r3 := r2 + 7" poderia ser representada ligada a diferentes arestas de controle, resultando em grafos diferentes para uma mesma implementação em *hardware*. O CDG, porém, evidencia que os blocos A e B são independentes em termos de controle, levando à construção de um único CDFG para os padrões extraídos dos CFGs em questão.

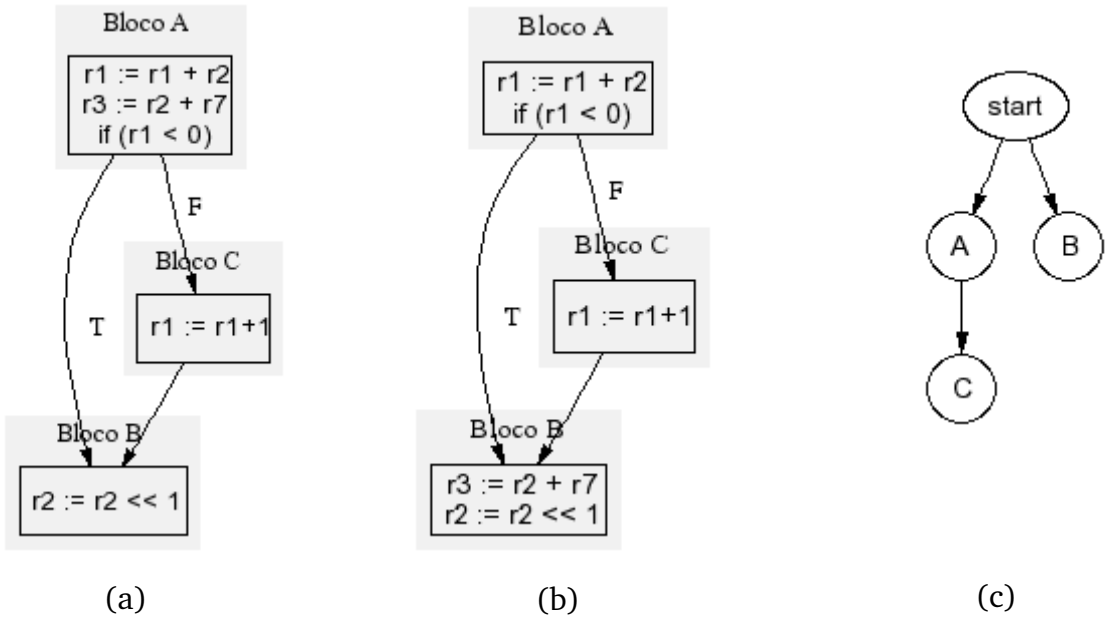
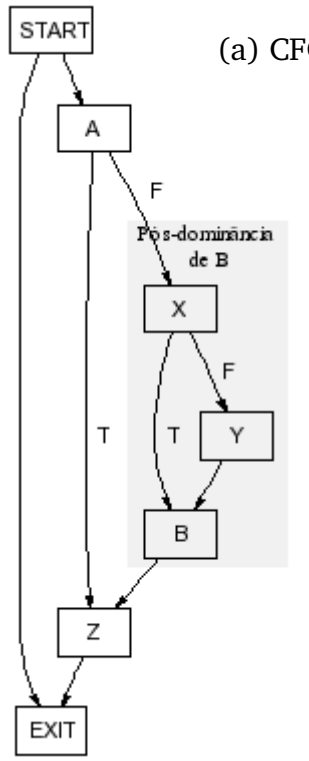


Figura 14 - Em (a) e (b), dois CFGs com diferentes blocos básicos (a instrução que define r3 foi movida do bloco A para o bloco B), representando porém a mesma computação. Em (c), o CDG correspondente a ambos os CFGs evidencia que os blocos A e B não possuem dependência de controle entre si.

Para a construção do CDG, foram adaptados alguns algoritmos dados em [App98]. Na definição do livro, as arestas do CDG representam as dependências de controle, mas não possuem rótulos *true* ou *false* como num CFG. Para a comparação de CDFGs de padrões, é importante que as dependências de controle sejam anotadas com estes atributos, pois uma implementação em *hardware* só pode ser reutilizada por mais de um padrão se as mesmas decisões forem tomadas nos blocos básicos dos padrões casados. Como ilustra a Figura 15, decidimos que uma aresta entre dois blocos básicos A e B de um CDG herda o mesmo rótulo que a aresta entre os blocos básicos A e X do CFG que cruza a região de pós-dominância[11] de B, sendo X um nó pós-dominado por B (incluindo ele próprio). A Figura 15 mostra os nós pós-dominados por B em sombreado, e a aresta A→X cruzando a região sombreada. Note que, no CDG, a aresta A→B herdou o rótulo F da aresta A→X do CFG. A Figura 16 mostra dois algoritmos derivados de um algoritmo de [App98] para cômputo da fronteira de dominância[12] de um bloco básico; a primeira adaptação leva ao cômputo do conjunto de vértices que compõem a fronteira de pós-dominância, e a segunda adaptação leva à obtenção da lista de arestas do CFG que cruzam a região de pós-dominância. Os rótulos destas arestas são então copiados para as

11 A região de pós-dominância de B é composta pelos blocos básicos pós-dominados por B.



(a) CFG

(b) CDG

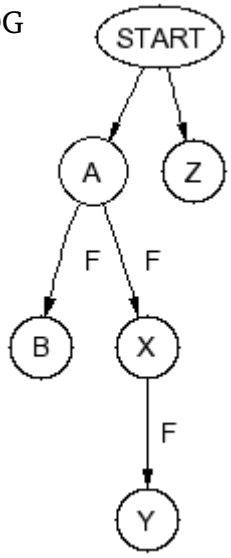


Figura 15 - Ilustração dos rótulos dados às arestas do CDG. Em (a), um CFG com a região de pós-dominância do bloco básico B sombreada; em (b), o CDG com rótulos nas arestas. Note que a aresta A→B no CDG herdou o rótulo da aresta A→X do CFG, que cruza a fronteira de pós-dominância de B.

arestas do CDG. A complexidade dos algoritmos é a mesma do algoritmo original de [App98], linear no número de arestas do CFG mais o tamanho das fronteiras de pós-dominância computadas que, na maioria dos casos, é linear no tamanho do CFG.

De posse do CFG, do CDG adaptado com rótulos nas arestas e dos DFGs maximais extraídos do interior de blocos básicos, é realizada a construção do CDFG para subgrafos do CFG. A Figura 17 ilustra um subgrafo de um CFG com os respectivos CDG e CDFG. No CDFG, as arestas tracejadas, com rótulos *true/false*, representam dependências de controle retiradas do CDG, e as arestas pontilhadas representam dependências de dados entre diferentes blocos básicos. As arestas sólidas representam as arestas dos DFGs maximais extraídos do interior de blocos básicos.

Outro uso que feito do CDFG ocorre na separação de componentes conexas para a geração de padrões. Após um padrão ter sido crescido para compor um subgrafo de um CFG, seu CDFG é gerado e, através de buscas lineares no grafo, são separadas as componentes conexas (por arestas que representem dependências de dados ou de controle). Cada componente conexa é extraída como um novo padrão. Grafos de padrões conexos são mais fáceis de comparar, e acreditamos que devam resultar em uma melhor relação entre tempo de execução e a área necessária para a implementação no *hardware* (grafos desconexos intuitivamente requerem mais registradores de entrada e saída).

12 Simplificando a definição formal dada em [App98], a fronteira de dominância de um bloco básico *x* é o conjunto de blocos básicos *w* tais que *x* domina um predecessor de *w*, mas não domina *w*. Analogamente, a fronteira de pós-dominância de *x* é o conjunto de blocos básicos *y* tais que *x* pós-domina um sucessor de *y*, mas não pós-domina *y*.

(a)

```
computePDF[n] =
        S ← {}
        for each node y in pred[n]
                if ipdom(y) ≠ n     /* pós-dominador imediato de y */
                        S ← S ∪ {y}
        for each child c of n in the postdominator tree
                computePDF[c]
        for each element w of PDF[c]
                if n does not postdominate w
                        S ← S ∪ {w}
        PDF[n] ← S
```

(b)

```
computePDF[n] =
        S ← {}
        for each node y in pred[n]
                if ipdom(y) ≠ n              /* pós-dominador imediato de y */
                        S ← S ∪ {(y,n)}   /* a aresta mantém o rótulo T/F */
        for each child c of n in the postdominator tree
                computePDF[c]
        for each edge (w,c) of PDF[c]
                if n does not postdominate w
                        S ← S ∪ {(w,c)}  /* a aresta mantém o rótulo T/F */
        PDF[n] ← S
```

Figura 16 - Adaptações feitas sobre o algoritmo para cômputo de fronteira de dominância dado na página 440 de [App98]. Em ambos algoritmos, computePDF[n] deve ser inicialmente chamado para a raiz n da árvore de pós-dominadores. Em (a), o algoritmo computa a fronteira de pós-dominância PDF[B] para todo bloco básico B, e S é um conjunto de blocos básicos. Em (b), o algoritmo obtém o conjunto das arestas que cruzam a região de pós-dominância de B para todo bloco básico B, guardando este conjunto em PDF[B]; S é um conjunto de arestas. Com isso, o CDG possuirá uma aresta (x,y) sempre que (x,k) ∈ PDF[y], e o rótulo T/F de (x,y) é o mesmo rótulo que (x,k) possui no CFG.

A comparação de CDFGs é um problema conhecido como *isomorfismo de dígrafos*[13], dado que os grafos são orientados. Trata-se de um problema NP-difícil, de modo que não esperávamos encontrar algoritmos polinomiais para resolvê-lo. Assim, foi implementado um algoritmo de complexidade exponencial moderada $O(n^{\log(n)})$, sendo n o número de vértices, com base no algoritmo apresentado no artigo [DDL77]. Este algoritmo compara dois grafos realizando uma busca em profundidade com *backtracking* utilizando critérios de poda *(prunning)* eficientes. Contudo, antes de aplicar o algoritmo, são feitas algumas verificações elementares nos grafos a serem comparados. Conclui-se rapidamente que dois grafos são diferentes se:

13 Um grafo $G_1(V_1, E_1)$ é dito isomorfo a um grafo $G_2(V_2, E_2)$ quando existe uma função unívoca f: $V_1 \rightarrow V_2$ que preserva adjacência. Isto é, $(v, w) \in E_1$ se e somente se $(f(v), f(w)) \in E_2$.

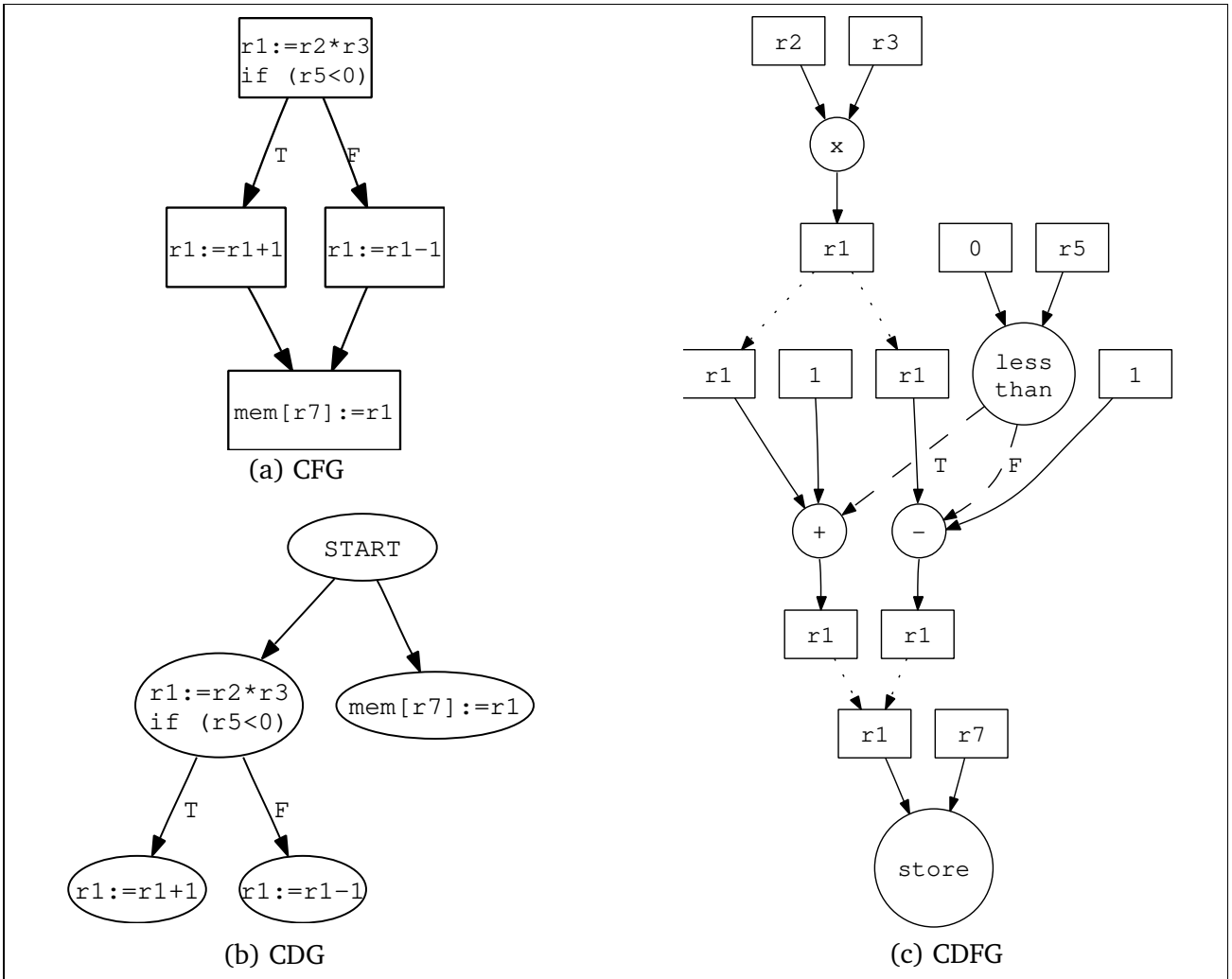


Figura 17 - Construção de um CDFG a partir de um subgrafo do CFG. Em (a), temos um subgrafo if-then-else de um CFG, em (b) temos o CDG correspondente e em (c), o CDFG. No CDFG, as arestas tracejadas (com rótulos T/F) representam dependências de controle retiradas do CDG; as arestas pontilhadas representam dependências de dados entre diferentes blocos básicos.

- Os grafos possuem diferentes números de vértices ou arestas;
- O número de vértices de um determinado tipo é diferente nos dois grafos (por exemplo, se os grafos possuem diferentes números de operadores de soma, ou de operandos que são constantes inteiras, etc.);
- O número de níveis, determinado por um escalonamento *as late as possible*[14], é diferente nos dois grafos. A comparação de níveis só se aplica a grafos acíclicos, que é o caso dos DFGs extraídos do interior de blocos básicos. Os CDFGs, por outro lado, possuem dependências circulares *loop carried*[15];

14 O escalonamento consiste em dividir os vértices em níveis, de tal forma que os vértices de um mesmo nível não possuam dependências entre si. O escalonamento é dito *as late as possible* se, quando um vértice pode ser colocado em diversos níveis sem violar dependências, ele é escalonado o mais tarde possível, ou seja, no nível mais baixo.

15 Dependências *loop carried* são dependências de dados entre diferentes iterações de um laço, por exemplo quando um valor atribuído numa iteração é utilizado na iteração seguinte (i := i+1).

- Em cada nível, o número de vértices de cada tipo é diferente.

Felizmente, a grande maioria dos grafos a serem comparados são descartados como diferentes antes da aplicação do algoritmo exponencial. Um aspecto importante sobre o casamento de padrões é a existência de operações comutativas, cuja ordem dos operandos é indiferente, como a adição e a multiplicação. Por exemplo, o algoritmo de casamento deve ser capaz de reconhecer que os grafos da Figura 18 podem ser casados. Para isso, a implementação do algoritmo aceita apontadores para funções que determinam se um dado operador é comutativo ou não. Outro ponto é a desconsideração dos nomes de registradores. Os grafos da Figura 19 podem ser casados, pois na síntese do *datapath* importa apenas o fluxo de dados e de controle que determina as interconexões entre unidades funcionais.

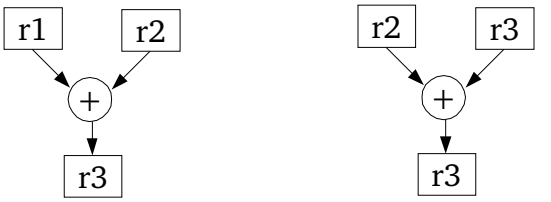


Figura 18 - A operação de soma é comutativa, portanto os grafos acima podem ser casados.

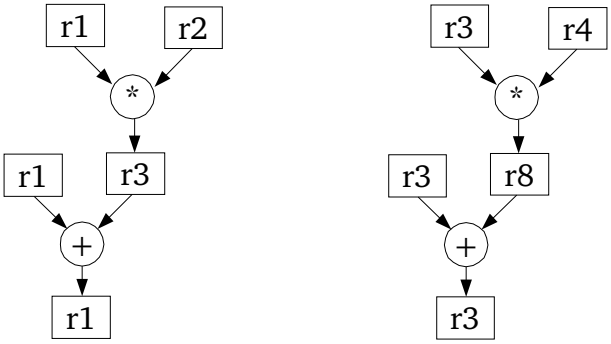


Figura 19 - Os grafos acima podem ser casadas, pois os nomes dos registradores apenas determinam o fluxo de dados.

3.4. Seleção de Padrões

A escolha de quais padrões implementar em *hardware* possui duas etapas. Na primeira etapa, extrai-se um conjunto de padrões candidatos à implementação que satisfaçam a diversas restrições. Estes padrões, somados aos existentes no repositório de padrões, são comparados entre si para verificar a existência de repetições – ocorrendo repetições, informações de freqüência de execução são somadas. Atribui-se então um peso (um número real) a cada padrão. Na segunda etapa, realiza-se a seleção de um conjunto de padrões de peso máximo que não interfiram entre si. Trata-se de um problema de otimização, para o qual foi feita uma redução ao problema de *conjunto independente de vértices* (NP-difícil) e foi implementada uma heurística gulosa.

O universo de padrões candidatos à implementação em *hardware* é composto pelos padrões maximais extraídos do interior de blocos básicos, pelos padrões crescidos na forma de subgrafos do CFG e pelos padrões previamente existentes no repositório de padrões do Chameleon, conforme ilustrado na Figura 20. Diversas restrições são aplicadas sobre este universo, antes e após sua construção, conforme discutido a seguir.

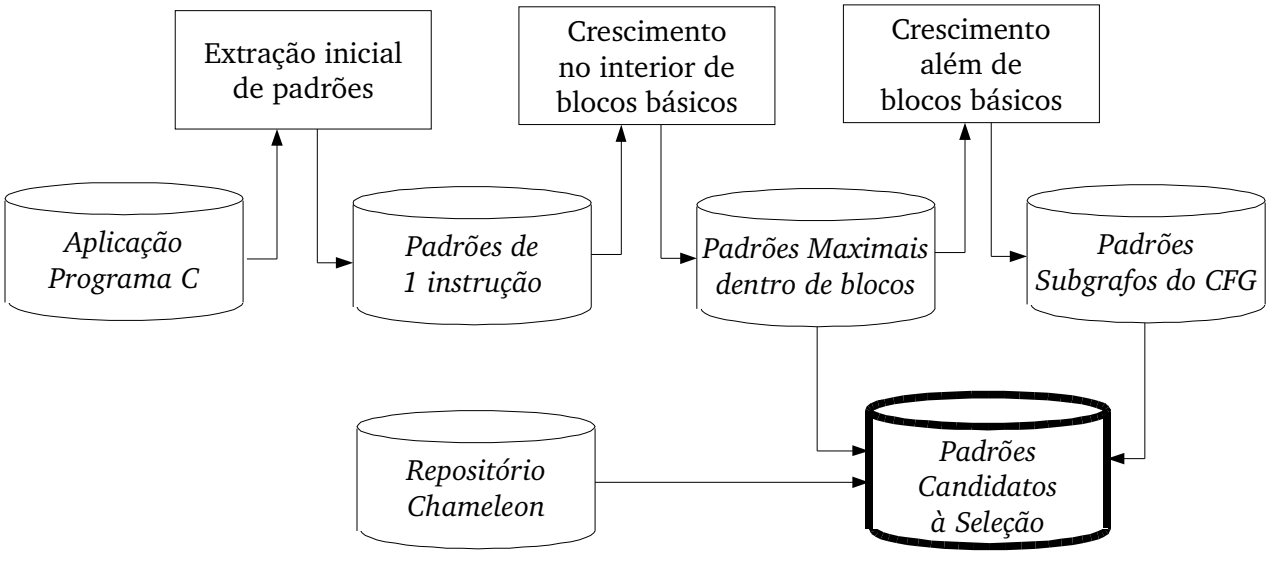


Figura 20 - Origem dos padrões candidatos à seleção para implementação em hardware. Entre os padrões "subgrafos do CFG", inclui-se o corpo de laços internos.

Nas implementações em *hardware* atuais, para reduzir a complexidade do trabalho, não é feito acesso direto à memória RAM — se isso fosse permitido no estágio de execução, seria necessário gerenciar os conflitos com os acessos simultâneos feitos no estágio de memória. Assim, os dados de entrada e saída do padrão provêm e são armazenados no banco de registradores do processador. Neste caso, os padrões não podem conter instruções *load* e *store*. Também não são permitidas chamadas a funções. Padrões que contenham estas instruções são filtrados antes mesmo de serem crescidos.

O número de operandos de entrada e saída de um padrão também é uma restrição, condicionada às características da implementação em *hardware*. Os filtros implementados aceitam como parâmetros os números máximos de entradas e saídas de um padrão. Nas implementações em *hardware* mais simples, uma única nova instrução especifica dois registradores de entrada e um registrador de saída, o que caracteriza um banco de registradores *dual-port read* e *single-port write* típico (utilizado na arquitetura SPARC). Outra possibilidade é uso de duas ou mais instruções para executar a computação de um único padrão. Por exemplo, sendo utilizadas duas instruções, seria possível implementar um padrão que lesse de até quatro registradores e escrevesse em até dois registradores – neste caso, possivelmente a implementação utilizaria mais de dois ciclos para realizar a computação. Para identificar os operandos de entrada e de saída de um padrão que seja um subgrafo de um CFG, primeiramente são verificadas as *use-definition chains (ud-chains)* e *definition-use chains (du-chains)*[16], criadas pelo compilador GCC. Se as *ud-chains* indicarem que um uso de um registrador

16 *ud-chains* – uma lista das definições que atingem um uso de um registrador [ASU86]
du-chains – uma lista dos usos que são atingidos por uma definição de um registrador [ASU86]

no padrão é atingido por uma definição externa ao padrão, conclui-se que se trata de um registrador de entrada. Analogamente, se as *du-chains* indicarem que um registrador definido no padrão é usado em um ponto externo, conclui-se que se trata de um registrador de saída. Porém, as *ud-chains* e *du-chains* não são suficientes para verificar se um registrador é de entrada ou de saída caso todas as suas definições e usos sejam internos aos próprio padrão, devido às dependências de dados *loop-carried* existentes quando o padrão pertence a um laço (o que ocorre na maioria das vezes, pois os laços possuem alta freqüência de execução). Neste caso, para determinar se um registrador é uma entrada do padrão, elaboramos o algoritmo iterativo da Figura 21. O algoritmo é semelhante ao usado no cômputo de *reaching definitions* na análise de fluxo de dados para algoritmos de otimização de código em compiladores [ASU86]. Dizemos que um operando *a* é *necessariamente definido* num ponto *p* do CFG se, em todos os caminhos do bloco básico inicial do padrão (subgrafo do CFG) até o ponto *p*, existe uma definição de *a*. O algoritmo constrói três conjuntos de operandos para cada bloco básico *B*:

- gen[*B*] = conjunto dos operandos que são definidos no interior do bloco básico *B*
- in[*B*] = conjunto dos operandos *necessariamente definidos* na entrada do bloco básico *B*
- out[*B*] = conjunto dos operandos *necessariamente definidos* na saída do bloco básico *B*

```
função computa_operandos_de_entrada(CFG do padrão)
{
    Adicione B0 vazio como bloco básico inicial do CFG;
    in[B0] := out[B0] := ∅; /* in e out nunca mudam para o bloco inicial B0 */
    for B ≠ B0 do out[B] := U; /* conjunto Universo: vetor de bits igual a 11111... */
    change := true;
    while change do begin
        change := false;
        for B ≠ B0 in depth-first order do begin
            in[B] := ⋂ (P predecessor de B) out[P]; /* para todo P predecessor de B */
            oldout := out[B];
            out[B] := gen[B] ∪ in[B];
            if out[B] ≠ oldout then change := true
        end
    end
}
```

Figura 21 - Algoritmo elaborado para determinar os operandos (registradores) que são entradas de um padrão.

Um operando *q* em um bloco básico *B* será um operando de entrada se o conjunto *in[B]* não contiver *q* e não existir uma definição de *q* entre o início de *B* e a primeira instrução (operador) que usa *q*. Uma vez determinados os operandos de entrada por este algoritmo, conclui-se que um operando é de saída se a sua *du-chain* aponta para um operando de entrada. O algoritmo iterativo elaborado tem a mesma complexidade dos algoritmos iterativos para *reaching definitions* dados em [ASU86]. No livro, os autores apresentam argumentos para sustentar que, quando os blocos básicos são visitados na ordem de uma busca em profundidade, o número de iterações do algoritmo será no pior caso igual a 2 mais o número de laços do CFG, com um limite superior empírico de 5 iterações e média de 2.75 iterações para CFGs de programas típicos. Os conjuntos de operandos computados pelo algoritmo são

implementados na forma de vetores de bits, de modo que as operações união e interseção são executadas muito rapidamente com as operações lógicas *and* e *or* sobre números inteiros. Cada bit *i* destes conjuntos se refere a um operando *i* tal que o bit *i* é igual a 1 se o operando (registrador) *i* pertence ao conjunto, e 0 caso contrário.

Além das restrições relacionadas ao conteúdo dos padrões, existem restrições relacionadas ao seu tamanho, limitado à área disponível para a implementação do *hardware* (área numa pastilha de silício ou número de células numa FPGA). Numa simplificação, esta área pode ser traduzida como um número máximo de unidades lógicas e aritméticas disponíveis para uso *em paralelo*. Por exemplo, pode-se restringir o *hardware* sintetizado a dois somadores, um comparador e um multiplicador. Neste caso, padrões que computassem mais operações precisariam fazê-las seqüencialmente, devendo ser avaliado se tal implementação realmente resultaria em ganho de tempo de execução – pois o processador original já é capaz de realizar as operações seqüencialmente. Como trabalho futuro, vislumbramos a aplicação de algoritmos de alocação de recursos que façam uso de *reservation tables*, semelhantes aos utilizados para máquinas VLIW ou super escalares [JJ91], para a avaliação automática da capacidade de paralelismo dos padrões no processo de seleção dos mesmos.

Uma vez obtido o conjunto de padrões que passaram pelos filtros, é realizada uma comparação entre os CDFGs dos padrões, todos contra todos, conforme procedimentos descritos em seção anterior. Cada grupo de padrões isomorfos é *substituído* por um novo padrão, que agrupa as informações de localização dos padrões originais (arquivos de origem, nomes das funções, números de RTX[17] e blocos básicos) e contém a soma de suas freqüências de execução.

Os pesos atribuídos aos padrões devem ser proporcionais ao ganho de desempenho que se espera obter com a implementação do padrão em *hardware*. No momento, adotamos uma fórmula bastante simples para o cálculo dos pesos:

peso $= f(c, d, e) = c*(d + e)$, onde:
c = estimativa do número de ciclos economizados com a implementação em *hardware*
d = freqüência dinâmica (número de vezes que o padrão foi executado)
e = freqüência estática (número de ocorrências do padrão)

O número de ciclos economizados c é uma estimativa da diferença entre o número de ciclos de máquina necessários para executar um padrão com as instruções originais do processador e o número de ciclos de máquina necessários para executar o mesmo padrão com a implementação de uma instrução especializada. Através de uma análise do conjunto de padrões extraídos dos benchmarks MediaBench e MiBench, apresentada e justificada no próximo capítulo, concluímos ser uma aproximação razoável assumir que o número de ciclos necessários para executar um padrão numa nova instrução seja igual a 1, ainda que, na verdade, a nova instrução possa requerer mais ciclos (em nossos experimentos, até 10 ciclos). Assim, é assumido simplesmente que a nova instrução especializada requer um ciclo para executar, enquanto a execução do padrão com as instruções originais do processador requer um ciclo para cada operador do padrão (operações de soma, subtração, *and, or, shift*, etc.). Em padrões do tipo subgrafos do CFG, consideramos os operadores no caminho crítico – o caminho entre o bloco básico inicial e o bloco básico final que possui o maior número de operadores. Deste modo, a estimativa *c* do número de ciclos economizados acaba sendo

17 RTX – expressões da linguagem de representação intermediária RTL do GCC, identificadas por números inteiros únicos.

proporcional ao tamanho do padrão, e acaba também indo na direção de privilegiar os padrões que mais reduzirão o tamanho do código dos programas. É claro que também reconhecemos que a implementação das instruções especializadas pode aumentar o período do ciclo, de modo que uma redução do número de ciclos para a execução de um código, nestas condições, pode não representar uma redução do tempo de execução.

A freqüência dinâmica $\underline{d}$ corresponde à freqüência de execução do padrão, de acordo com os dados extraídos de *profiling*. Conforme comentado anteriormente, quando um padrão consiste em um subgrafo de um CFG, sua freqüência de execução é computada como sendo o valor máximo dentre as freqüências de execução de cada bloco básico que o compõe (que corresponde ao número de vezes em que o bloco básico foi executado).

A freqüência estática $\underline{e}$ corresponde ao número de ocorrências do padrão no código da aplicação – número de padrões num grupo isomorfo. A freqüência estática é tipicamente muito menor do que a freqüência dinâmica, tendo assim influência pequena na fórmula para cálculo do peso. A relevância teórica da freqüência estática está relacionada à representatividade dos dados de *profiling*, que são obtidos para uma instância particular dos dados de entrada das aplicações. Se os dados de entrada não forem representativos das condições reais de operação, é possível que outros caminhos do CFG sejam tomados com freqüência, sendo então preferível implementar em *hardware* aqueles padrões que apareçam em vários trechos do código. Assim como o fator $\underline{c}$, a freqüência estática $\underline{e}$ tem o efeito colateral (benéfico) de privilegiar os padrões que mais reduzirão o tamanho do código dos programas.

Uma vez atribuídos os pesos, inicia-se a segunda etapa da seleção de padrões, que consiste na verificação de *interferência* entre padrões e aplicação dos algoritmos finais de seleção. Dois padrões interferem entre si quando suas ocorrências no código da aplicação se sobrepõem, não sendo possível ou vantajoso implementá-los simultaneamente em *hardware*. Por exemplo, a Figura 22 mostra um caso

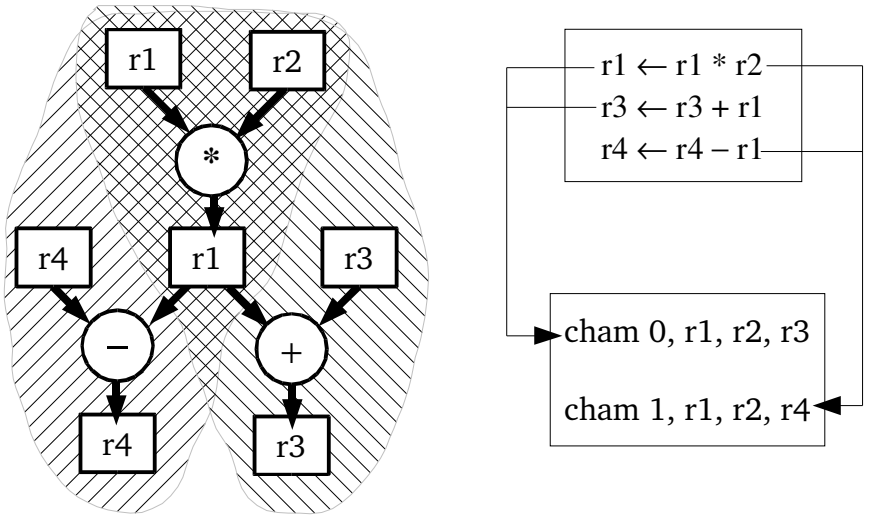


Figura 22 - Interferência de padrões. A figura mostra dois padrões hachurados que se sobrepõem. Se ambos forem implementados em hardware, produzindo as instruções "cham 0" e "cham 1" para substituir o código original exibido na caixa ao alto, o ganho em número de ciclos de execução não será a soma dos ganhos de cada padrão isoladamente.

em que, se fossem implementados os dois padrões (realçados pelas diferentes hachuras de fundo) simultaneamente, o ganho final não seria a soma dos ganhos de cada um. Isso porque cada padrão isoladamente substituiria duas instruções por uma única especializada, mas ambos em conjunto substituiriam apenas três instruções por duas especializadas (ao invés de substituir quatro por duas). Já a figura 23 mostra um caso ainda mais crítico, em que não é possível implementar os dois padrões simultaneamente, uma vez que a entrada de um dos padrões requer um valor intermediário computado na *datapath* do outro padrão. Se o valor intermediário fosse armazenado no banco de registradores, a implementação do *hardware* se tornaria mais custosa e provavelmente mais lenta.

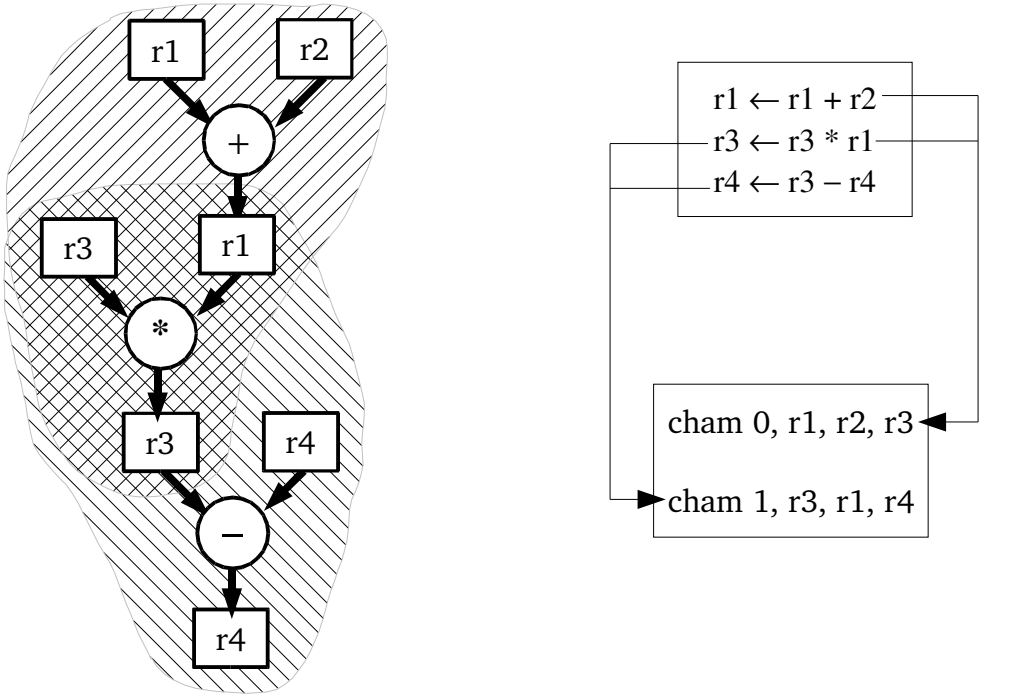


Figura 23 - Interferência de padrões. A figura mostra dois padrões hachurados que se sobrepõem. Se ambos forem implementados em hardware, produzindo as instruções "cham 0" e "cham 1" para substituir o código original exibido na caixa ao alto, o valor de r3 usado por "cham 1" estará incorreto (a multiplicação seria executada duas vezes).

Para verificar se dois padrões se sobrepõem – se eles interferem entre si –, são utilizadas as informações de localização do padrão que incluem o nome do arquivo do código fonte da aplicação, o nome da função da qual o padrão foi extraído e os números RTX da representação intermediária do GCC.

Assim, o problema de seleção dos padrões a serem implementados em *hardware* se reduz ao problema clássico do *conjunto independente de vértices de peso máximo*[18]. Cria-se um grafo $G=(V,E)$ em que cada vértice $v \in V$ está associado a um padrão candidato à implementação, possuindo o mesmo

18 O problema do conjunto independente de vértices de peso máximo é um problema de otimização NP-completo [CLRS01], enunciado como segue. Dado um grafo $G=(V,E)$, um conjunto independente de vértices é um subconjunto $V' \subseteq V$ tal que todo par de vértices de V' não é adjacente. Isto é, se $v, w \in V'$ então $(v, w) \notin E$. Um conjunto independente de peso máximo é um conjunto em que a soma dos pesos dos vértices pertencentes ao conjunto é máxima.

peso que o padrão. Insere-se uma aresta $(u, v) \in E$ se, e somente se, os padrões associados aos vértices u e v interferem entre si. Neste grafo, um conjunto independente de vértices de peso máximo representa um conjunto de padrões de peso máximo que não interferem entre si, o que define uma cobertura de peso máximo dos padrões sobre o código da aplicação. Como o problema do conjunto independente de vértices é NP-completo, propusemos um algoritmo guloso simples como primeira abordagem para a seleção de padrões, exibido na Figura 24. O laço do algoritmo executa $2|E|$ operações (continência de v em C), e a ordenação dos vértices usando *quicksort* tem complexidade média $O(|V|log|V|)$. Assim, a complexidade média do algoritmo é $O(máx(|V|log|V|, 2|E|))$. Como proposta para melhoria futura, pode-se pensar em algoritmos de aproximação mais elaborados para o conjunto independente de vértices de peso máximo, tais como os apresentados em [BH90, BBPP99].

```
função determina_conj_indep_peso_máx_guloso(G=(V,E)) {
        Seja S o conjunto dos vértices V ordenados pelo peso.
        Seja C o conjunto que conterá a solução gulosa do problema.
        C ← ∅;
        Para cada vértice v de S em ordem decrescente faça {
                Se nenhum vértice adjacente a v está contido em C, então
                        C ← C ∪ {v};
        }
        retorne C;
}
```

Figura 24 - Algoritmo guloso simples para obter uma solução heurística do problema de interferência entre padrões formulado como um problema de conjunto independente de vértices de peso máximo.

3.5. Biblioteca de Padrões

Desde cedo, identificamos no Projeto *Chameleon* a necessidade de se especificar estruturas de dados e um formato de arquivo para a representação e armazenamento dos padrões. Com estas especificações, torna-se possível o transporte de padrões entre as diferentes aplicações que implementam as diferentes etapas do *Chameleon*. As especificações permitiram ainda que fosse criada uma biblioteca de funções de manipulação de padrões, denominada Biblioteca de Padrões (referida nas implementações e documentação como *PatLib – Pattern Library)*, que permite o reuso de código nas ferramentas do *Chameleon* com a conseqüente redução do tempo de desenvolvimento e de correção de erros. As funcionalidades providas por esta biblioteca de funções incluem:

- A especificação de estruturas de dados (em linguagem C) para a representação dos padrões em memória, incluindo os diferentes tipos de grafos (DFG, CFG, CDG e CDFG), vetores e listas ligadas com seus respectivos iteradores (implementados como macros de C);
- Funções de criação, destruição , replicação e iteração sobre listas de padrões em memória;

- A especificação do formato de arquivo para armazenamento ou transporte de padrões entre aplicações, realizada através de notação BNF[19] e documentação em língua natural;

- Um *parser* constituído de funções para leitura e escrita de arquivos e diretórios de padrões (um diretório de padrões é um conjunto de arquivos de padrões). A implementação do *parser* foi parcialmente automatizada a partir da definição de expressões regulares no formato da ferramenta *Flex* [Flex03] e de uma gramática LALR[20] livre de contexto escrita no formato da ferramenta *Bison* [Bison03] que, respectivamente, produzem código C para analisadores léxico e sintático [App98]. Além da simples leitura e escrita de padrões em arquivos, o parser aponta com mensagens úteis as linhas e caracteres que contenham erros de sintaxe e alguns erros semânticos (por exemplo, inconsistência na numeração de vértices de grafos) nos arquivos escritos manualmente, facilitando a depuração de erros (assim como um compilador faz com códigos fonte incorretos);

- Algoritmos de comparação de grafos e filtragem de padrões[21];

- Documentação do código fonte criada utilizando a ferramenta *doxygen* [Hee04].

Várias outras funcionalidades importantes discutidas neste capítulo foram implementadas por ferramentas independentes (citadas no próximo capítulo), que tecnicamente não fazem parte do código da Biblioteca de Padrões, como por exemplo o código de crescimento de padrões por dependências de dados e controle, código de verificação de interferência entre padrões, código de atribuição de pesos e seleção de padrões e outros. As próximas seções deste capítulo apresentam o formato de arquivo de padrões e algumas das funções da API da Biblioteca de Padrões.

3.6. Formato de arquivo de padrões

Os arquivos de padrões possuem um formato textual (codificação ISO 8859-1 de 8 bits por caractere) que contém seções e campos para armazenar todas as informações coletadas acerca de padrões no Projeto Chameleon. O formato do arquivo é apresentado a seguir numa notação informal, para tornar a compreensão mais imediata. O Apêndice A contém as descrições de expressões regulares e da gramática utilizada para gerar o *parser* do arquivo, nos formatos das ferramentas *Flex* [Flex03] e *Bison* [Bison03] previamente citadas.

Por convenção, os arquivos possuem a extensão ".pat" (*patterns*). Cada arquivo pode conter vários padrões (delimitados pelos *tokens* `pattern <nome> = { ... }`). Padrões são colocados num mesmo arquivo quando estão de alguma forma relacionados. A extração inicial de padrões, indicada como *Repositório 1* na Figura 10, é realizada por nossa versão modificada do compilador GCC que cria um arquivo ".pat" para cada função da aplicação, com um padrão por RTX. Em seguida, novos arquivos são criados à medida em que os padrões sofrem transformações. Por exemplo, considere os arquivos abaixo:

```
adpcm.c/gcc/adpcm_coder.pat
adpcm.c/gcc/adpcm_decoder.pat

adpcm.c/grow/adpcm_coder.grow.pat
```

19 BNF – *Backus-Naur Form* [ASU86]

20 LALR – *Look-Ahead Left-to-right parse, Rightmost-derivation* [App98]

21 Parte dos algoritmos de filtragem provém do trabalho de outro aluno de mestrado do LSC, Rogério Moreira.

```
adpcm.c/grow/adpcm_decoder.grow.pat

adpcm.c/ctlgrow/adpcm_coder.ctlgrow.pat
adpcm.c/ctlgrow/adpcm_decoder.ctlgrow.pat

adpcm.c/ctlgrow-loops/adpcm_coder-2-23/adpcm_coder-2-23.grow_loop.pat
adpcm.c/ctlgrow-loops/adpcm_decoder-2-19/adpcm_decoder-2-19.grow_loop.pat
```

Estes arquivos ".pat" contêm padrões extraídos das funções *adpcm_coder* e *adpcm_decoder* do arquivo *adpcm.c* da aplicação ADPCM do pacote de *benchmark* MediaBench. O subdiretório *gcc* contém padrões com apenas uma RTX cada um. Em seguida, são crescidos no interior de blocos básicos gerando padrões de DFGs maximais, contidos no subdiretório *grow*. Quando os padrões são crescidos além das fronteiras de blocos básicos, são armazenados no subdiretório *ctlgrow* (em referência ao crescimento por fluxo de controle). O subdiretório *ctlgrow-loops* contém padrões que correspondem ao código de um laço inteiro; por exemplo, o arquivo *adpcm_coder-2-23.grow_loop.pat* contém um único padrão que corresponde ao subgrafo do CFG da função *adpcm_coder* delimitado pelos blocos básicos 2 e 23 (dominador e pós-dominador, respectivamente), que consiste no corpo de um laço.

Cada padrão, numa seção "*pattern name = { ... }*" de um arquivo de padrões, é representado como um conjunto de DFGs (grafos de fluxo de dados) conexos, sendo cada DFG inteiramente contido em um bloco básico. Cada DFG é identificado por um número inteiro, que o relaciona a uma seção *match_dfg* que contém a informação do bloco básico e função aos quais o DFG pertence. O conjunto de seções *match_dfg* de diferentes padrões permite recriar em memória o CFG da função da qual foram extraídos. Cada operando de cada DFG possui também listas *definition-use (du)* e *use-definition (ud),* explicadas anteriormente, que permitem estabelecer as dependências de dados entre diferentes DFGs do mesmo ou de diferentes blocos básicos (que podem ter sido referenciados em diferentes padrões de diferentes arquivos ".pat"). Através do CFG e das listas *ud/du*, é possível montar em memória os CDFGs (grafos de fluxo de dados e controle) explicados numa seção anterior.

A notação informal do formato do arquivo é dada na Figura 25. Naquela notação, INT corresponde a um número inteiro, FLOAT corresponde a um número real (ponto flutuante), STRING corresponde a uma seqüência de caracteres alfanuméricos, TEXT corresponde a uma *string* de múltiplas linhas, "..." (reticências) indicam uma lista de argumentos e "|" indica uma opção (*ou*) entre sentenças mutuamente exclusivas. Todas as demais palavras e caracteres são reservados – pertencem à gramática do *parser*. O significado de cada palavra e seção do arquivo é dado a seguir. Observe-se que algumas das seções e campos do arquivo são aplicáveis ou relevantes apenas em certos pontos do fluxograma de processamento do Projeto Chameleon, descrito no capítulo anterior. Conforme o padrão é transferido entre diferentes ferramentas, estes campos e seções podem ser alterados, adicionados ou removidos.

```
# 1-line comments start with '#'
/* comments of several lines are like in C */

version = INT.INT.INT ;
tags = STRING, STRING, ... ;

%file "STRING"

pattern STRING = {
       attributes = {
              speed-up = FLOAT ;
              tmp_opcode = STRING ;
              tmp_static_prof = INT ;
              tmp_dynamic_prof = INT ;
              match = {
                     file = "STRING" ;
                     function = "STRING" ;
                     match_dfg INT = {
                            frequency = INT ;
                            rtx = [INT INT] [INT INT] ... ;
                            operand_values = INT, INT, ... ;
                            bbinfo = {
                               bb = INT ;
                               succ = INT, INT, ... ;
                               pred = INT, INT, ... ;
                               idom = INT ;
                               ipdom = INT ;
                               branch = STRING INT INT INT INT ;
                            }
                     }
                     match_dfg INT = { } ...
              }
              match = { } ...
       }
       code = { TEXT }
       dfg INT = {
              aux = %{ TEXT %}
              operand INT = {
                     aux = %{ TEXT %}
                     type = INT ;
                     const_int = INT ; | const_real = FLOAT ; |
                        | const_string = "STRING" ; | reg_width = INT ;
                     def[INT] du = [STRING INT INT INT INT] ... ;
                     uses[INT, INT ...] ud = [STRING INT INT INT INT] ... ;
              }
              operator INT = {
                     aux = %{ TEXT %}
                     operands = INT, INT, ... ;
                     result = INT ;
                     type = INT ;
                     rtx = INT INT;
              }
       }
       vhdl | verilog | archc STRING = {
              cost = {
                     area = FLOAT ;
                     delay = FLOAT ;
              }
              source = %{ TEXT %}
       }
}
```

Figura 25 - Notação informal para as seções do formato de arquivo de padrões ".pat".

- *version* – a versão deste formato de arquivo, tal como “0.5.3”;

- *tags* – uma lista opcional de palavras que traz informações sobre a origem ou estado deste arquivo, e que é modificada conforme o arquivo é passado pelas diferentes etapas do Chameleon. Por exemplo, a tag `grow_dompdom` indica que o arquivo foi gerado automaticamente pela ferramenta *ctlgrow* de crescimento de padrões além de blocos básicos, consistindo em padrões delimitados por pares de blocos básicos dominador e pós-dominador imediatos. A data e a hora em que o padrão foi criado também são inseridas nesta lista;

- *%file “filename”* – indica que o arquivo de nome *“filename”* deve ser incluído no lugar da diretiva *%file* antes que a análise sintática seja feita. Funciona como a diretiva *“#include”* interpretada pelos pré-processadores da linguagem C;

- *pattern name* – indica o começo da descrição de um padrão, sendo *name* um nome dado ao padrão. Cada arquivo pode conter muitos padrões;

- *attributes* – inicia a seção de atributos de um padrão:

    - *speed-up* – estimativa do número de ciclos de máquina economizados com a substituição do código deste padrão por uma única instrução, usada na fase de seleção de padrões;

    - *tmp_opcode* – atributo temporário associado a este padrão quando ele é passado entre algumas aplicações do *Chameleon* (de 10 para 15 na Figura 6), indicando o *opcode* atribuído à instrução que implementa o padrão. Este atributo é temporário porque depende de outras instruções já criadas na mesma rodada do *Chameleon*, não fazendo sentido armazená-lo no repositório de padrões;

    - *tmp_dynamic_prof, tmp_static_prof* – atributos temporários associados ao padrão na fase de seleção, com dados de freqüência de execução e número de ocorrências do padrão, respectivamente. *tmp_dynamic_prof* é temporário porque está associado ao *profiling* realizado com determinada instância dos dados de entrada de uma aplicação, e *tmp_static_prof* é temporário porque depende do conjunto de padrões fornecidos para a etapa de seleção (os padrões podem vir de várias aplicações);

    - *match* – inicia uma seção com atributos referentes ao local em que este padrão foi encontrado nos programas analisados (nomes de arquivo e função). Pode haver várias seções *match* correspondendo a diferentes ocorrências de padrões isomorfos na mesma ou em diferentes aplicações. Conforme discutido anteriormente, padrões isomorfos são substituídos por um novo padrão com as seções *match* agrupadas;

        - *file* – nome do arquivo do código fonte C no qual este padrão foi encontrado;

        - *function* – nome da função em que o padrão foi encontrado;

        - *match_dfg X* – seção que agrupa as informações de localização de um DFG (grafo de

fluxo de dados) do padrão, especificado pelo número inteiro X. Conforme explicado anteriormente, o crescimento de padrões faz com que cada padrão possa ser composto por vários DFGs agrupados;

- *frequency* – O número de vezes que o bloco básico que contém o DFG a que esta seção match_dfg se refere foi executado durante a realização de *profiling*;

- *rtx [X Y]* – A lista de itens *[X Y]* identifica as RTXs – instruções da representação intermediária do GCC – que compõem o DFG. *X* é o número de uma RTX (que a identifica unicamente numa função), e *Y* é a posição em que ela aparece dentro do bloco básico (para determinar se ela aparece antes ou depois de uma outra RTX do mesmo bloco básico);

- *operand_values* – lista de números de registradores temporários *(pseudo registers)* atribuídos aos operandos de instruções pelo GCC na representação intermediária. Cada elemento da lista corresponde, na ordem, a um operando do DFG. Estes valores são usados para editar a representação intermediária do GCC após uma nova instrução do Chameleon ter sido produzida;

- *bbinfo* – inicia uma seção com informações sobre o bloco básico do qual o DFG foi extraído, de acordo com a numeração de blocos dada pelo GCC:

    - *bb* – número do bloco básico que contém o DFG;

    - *succ* – *(successors)* número do único ou dos dois blocos básicos sucessores deste padrão. Haverá dois sucessores se o bloco básico terminar em desvio condicional *(branch)*, ou apenas um sucessor se ele terminar em desvio incondicional *(jump)* ou a se execução prosseguir com a instrução seguinte ao bloco básico *(fall through)*;

    - *idom* – *(immediate dominator)* número do bloco básico dominador imediato deste DFG;

    - *ipdom* – *(immediate pos-dominator)* número do bloco básico pós-dominador imediato deste DFG;

    - *branch* – informação sobre a instrução condicional que termina o bloco básico que contém o DFG, caso ela exista. A *string* e os quatro inteiros se referem a: nome do padrão, número do DFG, número do operador que corresponde à instrução de *branch* no DFG, número do bloco básico sucessor se a condição for verdadeira e número do bloco básico sucessor se a condição for falsa;

- *code* - inicia uma seção que contém uma descrição legível do código do padrão, numa notação que é uma simplificação da *Register Transfer Language (RTL)* do GCC. O principal propósito desta seção é facilitar o entendimento do padrão por projetistas (visualização), não sendo no momento utilizada por ferramentas do *Chameleon*;

- *dfg Q* - inicia uma seção de descrição de um dos DFGs que compõem o padrão. Para *n* DFGs, o inteiro *Q* varia de *1* a *n*. O grafo é bipartido entre duas categorias de vértices, operandos e operadores, não existindo arestas que ligam vértices da mesma categoria;

    - *aux* – espaço reservado para conter qualquer informação que o usuário deseje associar a este DFG, na forma de uma cadeia de caracteres arbitrária delimitada pela seqüência de escape "%{" e "%}";

    - *operand x* – representa o vértice operando de número *x* do DFG. Cada operando pode ser usado (lido) por diversos operadores, mas é definido (escrito) por apenas um operador. Assim, cada vértice operando possui apenas uma aresta de entrada (o operador que o define) mas várias arestas de saída (os operadores que usam o valor do operando);

        - *aux* – espaço reservado para conter qualquer informação que o usuário deseje associar a este operando, na forma de uma cadeia de caracteres arbitrária delimitada pela seqüência de escape "%{" e "%}";

        - *type* – tipo do operando: um registrador ou uma constante (inteira, real ou alfanumérica);

        - *const_int, const_real, const_string* – valor da constante inteira, real ou alfanumérica, caso este operando seja do tipo constante;

        - *reg_width* – número de bits do registrador, caso este operando represente um registrador;

        - *def[k] du = [s i j m n] ...* - informação de *du-chain* deste operando, extraída da representação intermediária do GCC. Para a definição de número *k* (conforme numeração da representação intermediária do GCC), a lista de itens *[s i j m n]* representa os usos atingidos pela definição: *s* o nome do padrão, *i* o número do DFG naquele padrão, *j* o número do operador naquele DFG, *m* o número do uso dado pelo GCC e *n* o número do bloco básico também dado pelo GCC;

        - *uses[a, b, ...] ud = [s i j m n]* - informação de *ud-chain* deste operando, extraída da representação intermediária do GCC. Os usos *a*, *b*, ... deste operando possuem a numeração dada pela representação intermediária do GCC. A lista de itens *[s i j m n]* representa as definições que atingem os usos, com o significado dado no item anterior;

    - *operator y* – representa o vértice operador de número *y* do DFG. Cada operador pode usar

(ler) o valor de vários operandos, mas define (escreve) o valor de um único operando. Assim, possui várias arestas de entrada (os operandos usados), mas apenas uma aresta de saída (o operando definido);

    - *aux* – espaço reservado para conter qualquer informação que o usuário deseje associar a este operador, na forma de uma cadeia de caracteres arbitrária delimitada pela seqüência de escape "`%{`" e "`%}`";

    - *operands* – lista de operandos usados por este operador (lista de adjacências, com os vértices de entrada);

    - *result* – número do operando definido por este operador (lista de adjacências com um único vértice de saída). Cada operador define exatamente um operando;

    - *type* – o tipo do operador. A Tabela 2 apresenta os tipos atualmente reconhecidos pela Biblioteca de Padrões;

    - *rtx X Y* – identificação da RTX na representação intermediária do GCC correspondente a este operador. *X* é o número da RTX, e *Y* a posição em que ela aparece dentro do bloco básico que a contém;

- *vhdl* ou *verilog* ou *archc* – inicia a seção de descrição do *hardware* que implementa o padrão, nas linguagens VHDL, Verilog ou ArchC. Cada arquivo de padrões pode conter várias dessas seções, cada uma identificada por um nome diferente, possivelmente concatenado a número de versão. Por exemplo, um mesmo padrão pode ter diferentes implementações em *hardware*, como um somador de 32 bits com implementações *ripple carry* ou *carry lookahead*;

    - *cost* – inicia uma seção que descreve custos associados à implementação em *hardware*;

        - *area* – estimativa da área de silício requerida pela implementação do *hardware*;

        - *delay* – valor proporcional ao tempo de execução estimado da implementação do *hardware*;

- *source* - inicia a seção que contém o código HDL. Para evitar ambigüidades com os caracteres "`{`" e "`}`" eventualmente existentes nestas descrições, a seção é delimitada por "`%{`" e "`%}`". As descrições de *hardware* do padrão podem possuir uma tecnologia de implementação implícita, tal como uma certa linha de FPGAs de um certo fabricante. Para diferenciar tais implementações, os arquivos dos padrões podem ser colocados em diferentes diretórios do sistema de arquivos.

Operador	***Descrição***
LD	*Load* (lê da memória)
ST	*Store* (escreve na memória)
HIGH	Acessa metade alta da palavra
PLUS	Soma
SS_PLUS	Soma, *signed saturation in overflow*
US_PLUS	Soma, *unsigned saturation in overfl.*
LO_SUM	Soma a metade baixa da palavra
MINUS	Subtração
SS_MINUS	Subtr., *signed saturation in overflow*
US_MINUS	Subtr., *unsigned saturation in overfl.*
COMPARE	Subtração para comparação
NEG	Negação de valor lógico *true/false*
MULT	Multiplicação com sinal
MULT_C	Multiplicação c/ sinal por constante
DIV	Quociente em divisão com sinal
UDIV	Quociente em divisão sem sinal
MOD	Resto de divisão com sinal
UMOD	Resto de divisão sem sinal
SMIN	Menor inteiro com sinal
SMAX	Maior inteiro com sinal
UMIN	Menor inteiro sem sinal
UMAX	Maior inteiro sem sinal
NOT	Complemento *bit a bit*
AND	*E bit a bit*
AND_C	*E bit a bit* com constante
IOR	*OU bit a bit*
IOR_C	*OU bit a bit* com constante
XOR	*OU* exclusivo *bit a bit*
XOR_C	*OU* exclusivo *bit a bit* com constante
ASHIFT	Desloc. aritmético p/ esquerda
ASHIFT_C	ASHIFT por constante
ASHIFTRT	Desloc. aritmético p/ direita
ASHIFTRT_C	ASHIFTRT por constante
LSHIFTRT	Deslocamento lógico para a direita
LSHIFTRT_C	LSHIFTRT por constante

Operador	***Descrição***
ROTATE	Rotação de *bits* para a esquerda
ROTATE_C	Rotação p/ a esquerda por constante
ROTATERT	Rotação de *bits* para a direita
ROTATERT_C	Rotação para a direita por constante
ABS	Valor absoluto
SQRT	Raiz quadrada
FFS	1 + índice do *bit* 1 menos significativo
EQ	Comparação de igualdade
NE	Comparação de desigualdade
GT	Comparação *maior que*
GTU	Comparação *maior que* sem sinal
LT	Comparação *menor que*
LTU	Comparação *menor que* sem sinal
GE	Comparação *maior ou igual*
GEU	Comparação *maior ou igual* sem sinal
LE	Comparação *menor ou igual*
LEU	Comparação *menor ou igual* sem sinal
SIGN_EXTEND	Extensão de sinal p/ palavra maior
TRUNCATE	Trunca uma palavra grande
CALL	Chamada de função
TABLE_JUMP	Tabela de *jump* (construção *switch)*
JUMP	Salto incondicional
CLOBBER	Indica que o reg. pode ser sobrescrito
USE	Indica que o registrador é usado
SIGN_EXTRACT	Extrai uma seq. de bits e estende sinal
ZERO_EXTRACT	SIGN_EXTRACT mas não estende sinal
SUBREG	Acessa a palavra c/ largura diferente
MOV	Copia o valor de um reg. para outro
CONST	Indica valor constante em *runtime*
MUX	Multiplexador (reprent. do *hardware)*
MUX1	MUX com 1 *bit* de seleção
MUX2	MUX com 2 *bits* de seleção
MUX3	MUX com 3 *bits* de seleção
MUX4	MUX com 4 *bits* de seleção
MUXN	MUX com N *bits* de seleção

Tabela 2 - Operadores reconhecidos pela Biblioteca de Padrões.

3.7. Algumas funções da API da Biblioteca de Padrões

O código fonte da Biblioteca de Padrões possui atualmente mais de 14 mil linhas de código (incluindo comentários, mas sem contar o código fonte gerado automaticamente pelas ferramentas *flex* e *bison* e sem contar o código fonte de ferramentas que fazem uso da biblioteca, como as ferramentas citadas no próximo capítulo[22]), e implementa mais de 100 funções. Muitas destas funções cuidam da simples criação, duplicação e destruição de estruturas de dados, e não serão discutidas aqui. Nesta seção, são citadas algumas das funções de mais alto nível implementadas pela biblioteca.

```
int read_pattern_file(ptl_pattfile_t *pattfile);
int write_pattern_file(const ptl_pattfile_t *pattfile);
```

Estas funções lêem e escrevem um arquivo de padrões. A leitura cria uma lista ligada com os padrões contidos no arquivo, e a escrita recebe uma lista ligada para produzir um arquivo (as listas estão em campos da estrutura `ptl_pattfile_t`).

```
int read_pattern_filelist (const char *filelist_file,
                           ptl_pattfile_list_t **filelist);
```

Esta função lê os padrões especificados num arquivo texto (`filelist_file`) em que cada linha contém o nome de um arquivo de padrões e o nome do padrão desejado. É utilizada para troca de nomes de padrões entre ferramentas (por exemplo, num certo experimento descrito no próximo capítulo, a ferramenta *patselect* produz uma lista de nomes de padrões para a ferramenta *patmatch*).

```
int read_pattern_dir(ptl_pattdir_t *pattdir);
int write_pattern_dir(ptl_pattdir_t* pattdir);
```

Estas funções lêem e escrevem todos os padrões de todos os arquivos com extensão "`.pat`" contidos num diretório (também chamado de repositório). Trabalham com listas ligadas de estruturas `ptl_pattfile_t` que representam os arquivos e que, por sua vez, contêm listas ligadas dos padrões contidos no arquivo.

```
ptl_gcmp_graph_t *ptl_gcmp_build_cfg (ptl_pattern_list_t *pattl);
```

Se a lista de padrões passada como argumento a esta função contiver todos os padrões extraídos de uma função da aplicação, então é construído um CFG dos blocos básicos daquela função. Se a lista de padrões contiver um único padrão, e este padrão se estender além de um bloco básico, então é construído um CFG para aquele padrão – que será um subgrafo do CFG da função que contém o padrão.

22 A soma total do código implementado por mim, considerando a Biblioteca de Padrões e as ferramentas *pat-to-pat*, resulta em aproximadamente 20 mil linhas (incluindo comentários e documentação *doxygen* [Hee04]).

```
int ptl_gcmp_create_cdg_lists(ptl_gcmp_graph_t *cfg);
```

Esta função cria um CDG de uma função a partir de seu CFG. O CDG não é representado como um grafo distinto, mas sim como listas de dependências em cada vértice (bloco básico) do CFG.

```
ptl_gcmp_graph_t *
      ptl_gcmp_build_cdfg(ptl_pattern_t *patt, ptl_gcmp_graph_t *cfg,
                          char intra_loop, char inter_data,
                          char inter_control, char contract_inter_data,
                          char remove_duplicated_edges,
                          char precedence_edges, char clean_call_clobber);
```

Esta função cria um CDFG para um padrão `patt` crescido além de um bloco básico, utilizando seu CFG previamente construído. Recebe vários argumentos que controlam a criação do CDFG:

- `intra_loop` – *flag* que indica se arestas de dependências de dados *loop-carried* devem ser inseridas entre nós no interior dos DFGs que compõem o CDFG;
- `inter_data` – *flag* que indica se arestas de dependências de dados entre DFGs de diferentes blocos básicos devem ser inseridas a partir das informações de *def-use chains* dos padrões;
- `inter_control` – *flag* que indica se arestas de dependências de controle entre DFGs de diferentes blocos básicos devem ser inseridas a partir do CDG da função da aplicação;
- `contract_inter_data` – *flag* que indica se arestas de dependências de dados entre DFGs de diferentes blocos básicos, que interligam uma teia de dois ou mais registradores, devem ser contraídas de modo a restar um único registrador, como ilustra a Figura 26. Na figura, 3 vértices correspondentes ao registrador *r3*, interligados por arestas tracejadas, são substituídos por um único vértice, o que facilita a comparação de grafos por isomorfismo. Arestas de dependências de controle, não mostradas, revelam que a execução de *BB 1* ou *BB 2* depende de uma decisão *if-then-else,* que determina se o valor utilizado em *BB 3* foi computado em *BB 1* ou em *BB 2;*
- `remove_duplicated_edges` – em alguns casos, as *def-use chains* contêm dependências de dados replicadas geradas pelo GCC, que acabam levando a arestas duplicadas no CDFG. Esta *flag* ativa um código de remoção de arestas duplicadas;
- `precedence_edges` – *flag* que indica se arestas de precedência entre operadores *load* e *store* de DFGs desconexos devem ser inseridas;
- `clean_call_clobber` – *flag* que indica se operadores *clobber*, utilizados pelo GCC para indicar que registradores são sujos por chamadas de função, devem ser removidos (útil para visualização, dado que em geral aproximadamente 90 operadores *clobber* são gerados para cada operador call, levando a grafos enormes que não cabem na tela do computador).

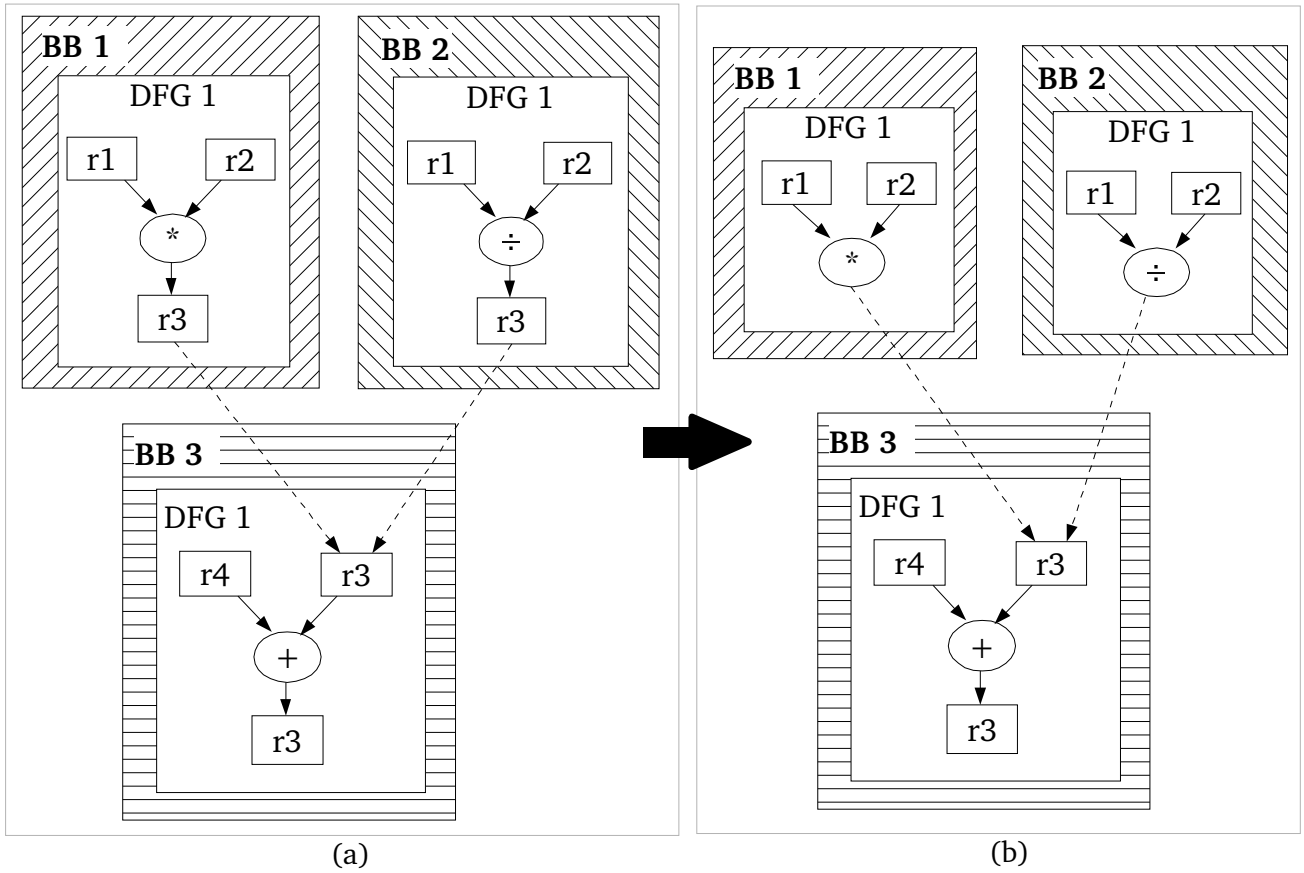


Figura 26 - Exemplo da contração de arestas de dependências de dados entre diferentes blocos básicos (tracejadas). Os vértices operandos correspondentes ao registrador r3 em (a), nos blocos básicos BB 1 e BB 2, foram removidos junto a suas arestas de entrada, resultando no grafo mostrado em (b).

```
int ptl_gcmp_mark_cdfg_components(ptl_gcmp_graph_t *cdfg);

ptl_gcmp_components_t *
      ptl_gcmp_split_cdfg_components (ptl_gcmp_graph_t *cdfg,
                                      int n_components);
```

Esta funções identificam e separam as componentes conexas (por dependências de dados ou controle) de um CDFG, permitindo gerar padrões distintos para cada componente conexa.

```
typedef struct {
      char (*is_commutative)(ptl_gcmp_graph_t *g,
                             int vertex);

      char (*matchable_edges)(ptl_gcmp_graph_t *graph1,
                              ptl_gcmp_graph_t *graph2,
                              int edge1, int edge2,
                              ptl_gcmp_commut_func_t cf);
```

```
      char (*matchable_vertices)(ptl_gcmp_graph_t *graph1,
                                 ptl_gcmp_graph_t *graph2,
                                 int vertex1, int vertex2);
} ptl_gcmp_fpointers_t;
char ptl_compare_cdfgs_single_component(ptl_gcmp_graph_t *cdfgA,
                                        ptl_gcmp_graph_t *cdfgB,
                                        ptl_gcmp_fpointers_t *funcp,
                                        ptl_gcmp_vmapping_t *vmap);
```

A função `ptl_compare_cdfgs_single_component` compara duas componentes conexas de dois dados CDFGs (A e B). O argumento `funcp` é uma estrutura com apontadores para três funções que configuram o algoritmo de isomorfismo:

- `is_commutative` – esta função retorna verdadeiro (1) se o vértice dado como argumento é um operador comutativo (por exemplo, um operador de soma ou multiplicação, mas não um operador de subtração ou divisão). Isto permite que o algoritmo de isomorfismo saiba se a ordem dos operandos de entrada do operador é significativa na comparação dos grafos;
- `matchable_edges` – esta função determina se duas arestas podem ser casadas. A implementação *default* casa duas arestas se elas forem do mesmo tipo (por exemplo, dados ou controle) e, se forem de controle, são casadas apenas se tiverem o mesmo atributo *true/false*;
- `matchable_vertices` – esta função determina se dois vértices podem ser casados. A implementação *default* casa dois vértices se eles forem do mesmo tipo (por exemplo, *add* casa com *add*). Uma variação interessante desta função permite especificar se operandos constantes (por exemplo, os inteiros 5 e 7) são casados independentemente do valor da constante. Por exemplo, seria possível codificar em *hardware* (HDL) um *pool* de 32 diferentes conjuntos de constantes utilizadas por 32 diferentes ocorrências de um padrão nas aplicações, e utilizar um campo de 5 *bits* no *opcode* da instrução que implementa o padrão para selecionar qual dos 32 conjuntos de constantes deve ser usado naquela computação do padrão. Para tal implementação em *hardware*, seria interessante que diferentes constantes fossem casadas no isomorfismo entre padrões.

```
void ptl_gcmp_write_dfg_to_dot_file(ptl_gcmp_graph_t *g,
                                    FILE *f, char by_level);
void ptl_gcmp_write_cfg_to_dot_file(ptl_gcmp_graph_t *g, FILE* file);
void ptl_gcmp_write_cdg_to_dot_file(ptl_gcmp_graph_t *g, FILE* f);
void ptl_gcmp_write_cdfg_to_dot_file_flat(ptl_gcmp_graph_t *cdfg, FILE* f);
void ptl_gcmp_write_cdfg_to_dot_file_hierarchical(ptl_gcmp_graph_t *cdfg,
                                                  FILE* f);
```

Estas funções escrevem os grafos DFG, CFG, CDG e CDFG no formato da ferramenta *Graphviz Dot* [GEK+04] de renderização de grafos orientados, permitindo a visualização e impressão. No caso do CDFG, o grafo pode ser desenhado no estilo *flat* ou *hierarchical*; o último desenha os operandos e operadores separados por DFGs e blocos básicos (semelhante ao desenho da Figura 26), enquanto o primeiro não faz estas separações.

Como já citado, várias outras funcionalidades importantes discutidas neste capítulo foram implementadas por ferramentas independentes e por isso não pertencem à API da Biblioteca de Padrões. O próximo capítulo apresenta estas ferramentas e sua utilização em experimentos realizados para extração, crescimento e seleção de padrões de código.

Capítulo 4

Resultados Experimentais

No capítulo anterior, foram discutidos os métodos empregados na extração, crescimento e seleção de padrões, e foi introduzida a Biblioteca de Padrões. Neste capítulo, são apresentados resultados experimentais obtidos através da aplicação daqueles métodos. A seção 4.1 descreve o ambiente de trabalho – infraestrutura para realização dos experimentos. A seção 4.2 apresenta as aplicações dos pacotes de *benchmarks* utilizados para a extração de padrões. As ferramentas de *software* desenvolvidas para a realização dos experimentos (e para uso no projeto Chameleon) são comentadas na seção 4.3. A seção 4.4 discute um experimento realizado para averiguar a influência do número de ciclos das novas instruções na fórmula para cômputo dos pesos dos padrões. Finalmente, a seção 4.5 apresenta um experimento que nos permitiu identificar padrões que são comuns a diversas aplicações dos *benchmarks*. Estes padrões foram classificados em categorias funcionais, diversas das quais foram consideradas interessantes para a implementação em *hardware* na forma de novas instruções.

4.1. Ambiente de trabalho

Os resultados ora apresentados foram obtidos com a aplicação das ferramentas de *software* desenvolvidas no Projeto Chameleon sobre um conjunto de programas de pacotes de *benchmarks*. Os programas foram executados na versão 1.0.13 do modelo VHDL do processador Leon 2 [Gaisler04], sintetizado numa FPGA Virtex Xilinx [Xilinx04] de 800 mil *gates*. A FPGA estava inserida numa placa de desenvolvimento modelo XSV-800 fornecida pela empresa Xess [Xess04], dotada de 4MB de RAM estática e 2 MB de memória *flash*, com *clock* programado para 10 MHz. A placa é conectada a um computador PC através de interfaces paralela (porta padrão de impressora) e serial (RS 232). A primeira é utilizada para programar a FPGA, e a segunda é utilizada para carregar o código da aplicação a ser executada e transferir arquivos. Não utilizamos um sistema operacional[23], e sim um conjunto de funções que implementamos que provêem alguns serviços de sistema, em especial o acesso transparente para a aplicação ao sistema de arquivos do PC conectado à placa através da interface serial – pois a placa de desenvolvimento não possui um disco[24]. Assim, a aplicação pode ler e escrever arquivos armazenados no PC, incluindo os arquivos de *profiling* (extensão *.da*) gerados durante a execução das aplicações compiladas pelo GCC com a opção de instrumentação de código para *profiling*.

23 Tivemos sucesso em executar uma versão reduzida do *kernel* do sistema Linux [*μClinux*] sobre o Leon, permitindo o uso de NFS *(Network File System)* através de uma interface de rede *Ethernet* da placa de desenvolvimento. No entanto, a placa que utilizávamos tinha toda a sua memória consumida pelo sistema, não sobrando muito para a aplicação. Estamos atualmente testando novas placas, uma delas com 16 MB de RAM, uma FPGA com 3 milhões de gates e capaz de ser conectada ao barramento PCI de um PC.

24 Este "sistema de arquivos virtual" foi implementado por Edson Borin, um aluno de doutorado do LSC.

4.2. Benchmarks

Foram utilizados 6 programas do pacote MediaBench I [LPS97] e 9 programas do pacote MiBench [GRE+01], todos na linguagem C. Os pacotes possuem um total de 10 e 20 aplicações respectivamente, mas algumas delas não puderam ser executadas na placa de desenvolvimento com o Leon por falta de memória ou por falta de implementações completas de algumas chamadas de sistema. Além disso, algumas aplicações do MiBench são iguais às do MediaBench (JPEG, ADPCM, GSM). Ainda assim, as 15 aplicações executadas compreendem uma larga faixa de aplicações multimídia para sistemas embutidos, como detalha a Tabela 3.

Os pacotes de *benchmarks* acompanham arquivos de entrada e de saída dos programas – por exemplo, arquivos com áudio não compactado/codificado como entrada e arquivos de saída com o resultado da compressão/codificação. Logicamente, todos os programas executados na placa de desenvolvimento, dos quais extraímos padrões, produziram saídas corretas como esperado.

4.3. Ferramentas utilizadas

Na obtenção dos dados experimentais, algumas ferramentas de *software* desenvolvidas no Projeto Chameleon, do tipo *linha de comando*[25], foram aplicadas sobre os padrões de código:

- `grow-patt` – realiza o crescimento de padrões no interior de blocos básicos, na forma de grafos maximais de fluxo de dados (DFGs);
- `ctlgrow` – realiza o crescimento de padrões além da fronteira de um bloco básico, na forma de subgrafos do CFG (grafo de fluxo de controle) de uma função do código fonte. Permite a especificação de filtros de padrões com base em números máximo e mínimo de: blocos básicos, operadores (instruções), laços no interior do padrão, instruções condicionais *(branches)* e freqüência de execução;
- `find-loops` – identifica conjuntos de blocos básicos que compõem um laço, fornecendo os parâmetros apropriados para a ferramenta `ctlgrow` para que seja gerado um padrão correspondente ao corpo do laço;
- `patmatch` – efetua o casamento de padrões (isomorfismo dos grafos de fluxo de dados e de controle – CDFGs), com a opção de comparar os padrões "todos contra todos" ou comparar apenas os padrões de diferentes aplicações dos *benchmarks*. Os resultados da comparação são gerados na forma de relatórios em arquivos texto e na forma de gráficos (utilizando a ferramenta de geração de gráficos *gnuplot* [Gnuplot04]);

25 Uma interface gráfica baseada em HTML já existe atualmente no LSC, permitindo a aplicação de algumas das ferramentas sem a necessidade da digitação de opções de linha de comando. A interface foi desenvolvida para que usuários de outros projetos possam utilizar os padrões extraídos no projeto Chameleon.

	Aplicação	***Categoria***[26]	***Descrição***
MediaBench	ADPCM	Telecom	Adaptive Differential Pulse Code Modulation – uma das mais simples e antigas codificações de áudio.
	GSM	Telecom	Padrão europeu GSM 06.10 para transcodificação de voz, utilizando "*residual pulse excitation/long term prediction coding at 13 kbit/s*".
	G721	Telecom	Implementações de referência dos padrões de compressão de voz CCITT (International Telegraph and Telephone Consultative Committee) G.711, G.721 e G.723.
	JPEG	Consumidor	Método de compressão de imagens coloridas e monocromáticas com perda (a imagem comprimida não é idêntica à imagem original).
	MPEG2	Consumidor	Padrão para compressão/transmissão de vídeo digital de alta qualidade. O núcleo da computação é uma transformada discreta de cosseno para codificação e a transformada inversa para decodificação.
	PEGWIT	Segurança	Um programa para criptografia e autenticação por chaves públicas. *"It uses an elliptic curve over GF(2255), SHA1 for hashing, and the symmetric block cipher square."*
MiBench	BitCount	Automação	Programa que conta o número de bits 1 e 0 de um array de inteiros, utilizando cinco diferentes algoritmos.
	QSort	Automação	Ordena um grande vetor de strings usando o conhecido algoritmo *quicksort*.
	Susan	Automação	Um pacote para reconhecimento de imagem, desenvolvido para identificar regiões em imagens de ressonância magnética do cérebro. Também empregado em controle de qualidade baseado em visão.
	Lame	Consumidor	Codificador MP3 GPL que suporta taxa de compressão constante, média e variável.
	Mad	Consumidor	Decodificador de áudio MPEG de alta qualidade. Suporta extensões MPEG-1 e MPEG-2 para menores freqüências de amostragem, e o formato MPEG-2.5.
	Dijkstra	Redes	Constrói um grande grafo em representação de matriz de ajacências e calcula o caminho mais curto entre todos os pares de vértices usando repetidas aplicações do algoritmo de Dijkstra.
	String Search	Escritório	Busca por palavras em frases usando um algoritmo de comparação insensível a maiúsculas/minúsculas.
	SHA	Segurança	Secure Hash Algorithm – produz um resumo *(message digest)* de 160 bits para uma dada entrada. Usado na troca segura de chaves criptográficas e para geração de assinaturas digitais. Também usado nas funções de hash MD4 e MD5.
	CRC32	Telecom	Efetua o Cyclic Redundancy Check de 32 bits em um arquivo. Usado com freqüência para detectar erros em transmissões de dados.

Tabela 3 - Aplicações dos pacotes MediaBench e Mibench utilizadas na obtenção dos resultados experimentais.

26 Categoria conforme a classificação proposta no pacote MiBench [GRE+01].

- `patselect` – aplica algoritmos para a seleção dos padrões que apresentariam maior ganho se implementados em *hardware*, e também implementa alguns filtros de padrões. Produz relatórios em formato texto e na forma de gráficos (assim como a ferramenta `patmatch`);
- `pat2dot, print_cfg` – ferramentas que geram imagens dos grafos dos padrões, para visualização e impressão, no formato da ferramenta de desenho de grafos *Graphviz* [GEK+04].

As ferramentas desenvolvidas por mim foram *ctlgrow, patmatch, patselect* e *print_cfg*. Algumas das ferramentas (*grow-patt, ctlgrow* e *find-loops*) são do tipo *pat-to-pat*: lêem arquivos de padrões, efetuam modificações ou adições e geram novos arquivos de padrões. As demais ferramentas (*patmatch* e *patselect*) lêem arquivos de padrões e produzem relatórios e gráficos como saída. As ferramentas são aplicadas seguindo o fluxograma da Figura 27. Primeiramente, a versão modificada do GCC extrai os padrões de uma única instrução (RTX). Os padrões são então crescidos no interior de blocos básicos pela ferramenta *grow-patt*, levando a padrões maximais com informações do bloco básico a que pertencem. Estes padrões são utilizados pela ferramenta *ctlgrow* para uma extração dos subgrafos do CFG delimitados por blocos básicos dominador e pós-dominador imediatos. Além disso, a ferramenta *ctlgrow* recebe parâmetros da ferramenta *find-loops* (números de blocos básicos) para produzir padrões que correspondam ao corpo inteiro de laços internos. O conjunto formado pelos padrões maximais no interior de blocos básicos e pelos padrões produzidos pela ferramenta *ctlgrow* é então fornecido como entrada para as ferramentas *patmatch e patselect*. A aplicação de filtros (opcional) sobre os padrões ocorre juntamente com a aplicação das ferramentas *ctlgrow* e *patselect*.

A ordem em que as ferramentas de isomorfismo *(patmatch)* e atribuição de pesos e ordenação *(patselect)* são aplicadas leva a diferentes resultados. Quando o casamento de padrões (isomorfismo de grafos) é realizado primeiro, comparando os padrões "todos contra todos", a ferramenta *patmatch* produz um relatório do número de ocorrências de cada padrão, que é lido em seguida pela ferramenta *patselect* para o cômputo dos pesos dos padrões levando em conta as suas ocorrências no código. Isso permite gerar uma lista ordenada dos melhores padrões para a implementação em *hardware* segundo os critérios e algoritmos adotados. Como a ferramenta *patselect* também é capaz de filtrar e ordenar padrões pelo peso, obtemos resultados também úteis invertendo a ordem de aplicação das ferramentas – neste caso, a ferramenta *patmatch* lê os relatórios produzidos pela ferramenta *patselect*. Aplicando-se primeiramente a ferramenta *patselect* (considerando cada padrão com uma única ocorrência para cômputo do peso), e depois realizando o casamento apenas dos padrões selecionados entre as diferentes aplicações dos *benchmarks*, obtém-se uma lista de padrões ordenada pelo número de ocorrências em diferentes aplicações, com o intuito de encontrar padrões bastante executados (selecionados pelo peso) e que se repetem em vários programas – sendo potencialmente genéricos a ponto de valer a pena escrever implementações VHDL eficientes para eles e armazená-los no repositório de padrões do Projeto Chameleon. As seções a seguir detalham estes procedimentos e apresentam os gráficos e tabelas obtidos.

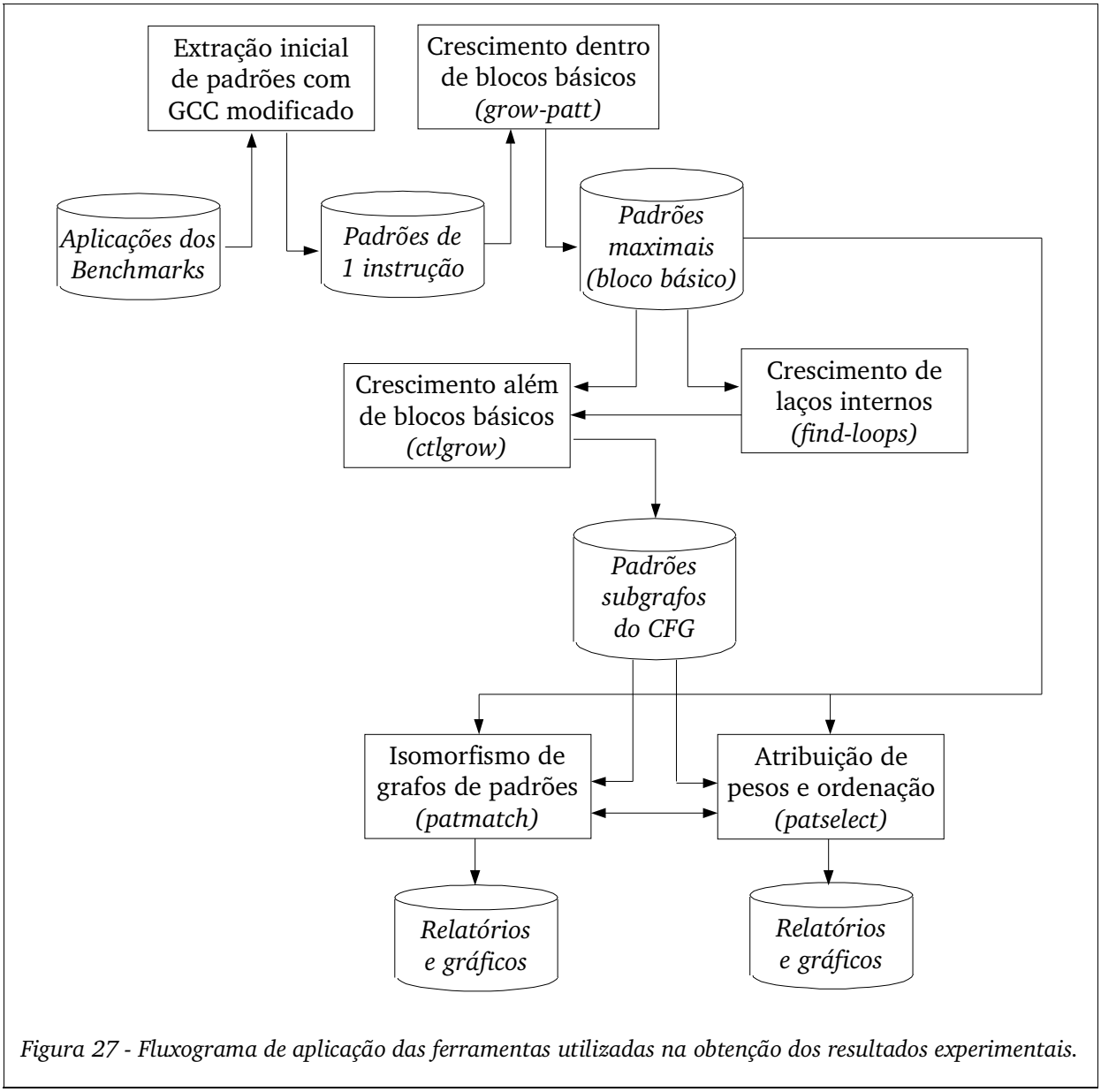


Figura 27 - Fluxograma de aplicação das ferramentas utilizadas na obtenção dos resultados experimentais.

4.4. Averiguação da influência do número de ciclos na fórmula do peso

No capítulo anterior, foi apresentada uma fórmula simples proposta para a atribuição de pesos aos padrões, tal que quanto maior peso, maior o ganho de desempenho esperado com a implementação do padrão em *hardware*, associado a novas instruções no processador. A fórmula é:

peso = $f\ (c, d, e) = c*(d + e)$, onde:

c = estimativa do número de ciclos economizados com a implementação em *hardware*
d = freqüência dinâmica (número de vezes que o padrão foi executado)
e = freqüência estática (número de ocorrências do padrão)

A variável c na fórmula do peso corresponde à diferença entre o número de ciclos necessários

para executar o padrão com as instruções originais do processador e o número de ciclos necessários para executar a nova instrução especializada que implementa o padrão[27]:

$$c = ops - n$$

O número *ops* de ciclos necessários para executar o padrão com as instruções originais de um processador RISC (como o Leon/SPARC) pode ser satisfatoriamente aproximado pelo número de operações executadas no padrão de acordo com a representação intermediária do compilador (operações tais como *add/sub, shift, load/store, mov, and/or,* etc.). Por outro lado, só é possível determinar o número n de ciclos requeridos por uma implementação especializada do padrão em *hardware* após tal implementação ter sido realizada e simulada. No entanto, o propósito da atribuição de pesos aos padrões é permitir a seleção dos melhores padrões a serem implementados, o que claramente deve ser feito antes da implementação. Assim, recorremos a uma estimativa: assumimos que as novas instruções executarão em apenas um ciclo.

Assumir que as novas instruções executam todas em um ciclo poderia prejudicar o nosso modelo. Por exemplo, considere um padrão A que originalmente era implementado com 14 instruções, 1 ciclo por instrução. A estimativa c para o padrão A seria calculada como $c = 14 - 1 = 13$. Numa lista de padrões ordenada pelo peso, isto potencialmente colocaria o padrão A à frente de um padrão B que possuísse uma estimativa $c = 6 - 1 = 5$ (esse com certeza seria o caso se as outras variáveis na fórmula do peso fossem iguais para ambos os padrões). No entanto, suponha que, sendo realizadas implementações de ambos os padrões, fosse possível verificar que uma nova instrução criada para o padrão A necessitaria de 10 ciclos para ser executada, enquanto uma nova instrução criada para o padrão B necessitaria de fato de apenas 1 ciclo. Neste caso, o peso computado para o padrão A deveria, na verdade, tê-lo colocado atrás do padrão B na lista ordenada, pois para o padrão A teríamos $c = 14 - 10 = 4$ e, para o padrão B, teríamos $c = 5$.

Buscamos então uma forma de averiguar o quão freqüentes seriam os casos de erro na ordenação dos padrões pelo peso se assumíssemos que qualquer nova instrução especializada executaria em apenas um ciclo para a finalidade do cômputo da estimativa c na fórmula do peso. Para isso, utilizamos nosso universo de padrões (extraídos das aplicações de *benchmarks*, conforme já citado) para montar dez listas ordenadas de todos os padrões. Cada uma destas listas foi ordenada pelo peso assumindo que a nova instrução precisaria de n ciclos para ser executada; a primeira lista assumia $n = 1$, a segunda lista assumia $n = 2$, a terceira $n = 3$ e assim sucessivamente, conforme ilustra a Figura 28. Em seguida, realizamos a interseção de cada uma das listas com a primeira lista (que assumia $n = 1$), para verificar quantos dentre os padrões de maior peso continuavam sendo os mesmos independentemente do número de ciclos que se assumia no cômputo de seu peso.

27 Como comentado no capítulo anterior, assumimos que a implementação de novas instruções não irá afetar o período do *clock*. Caso contrário, este efeito precisaria ser levado em conta.

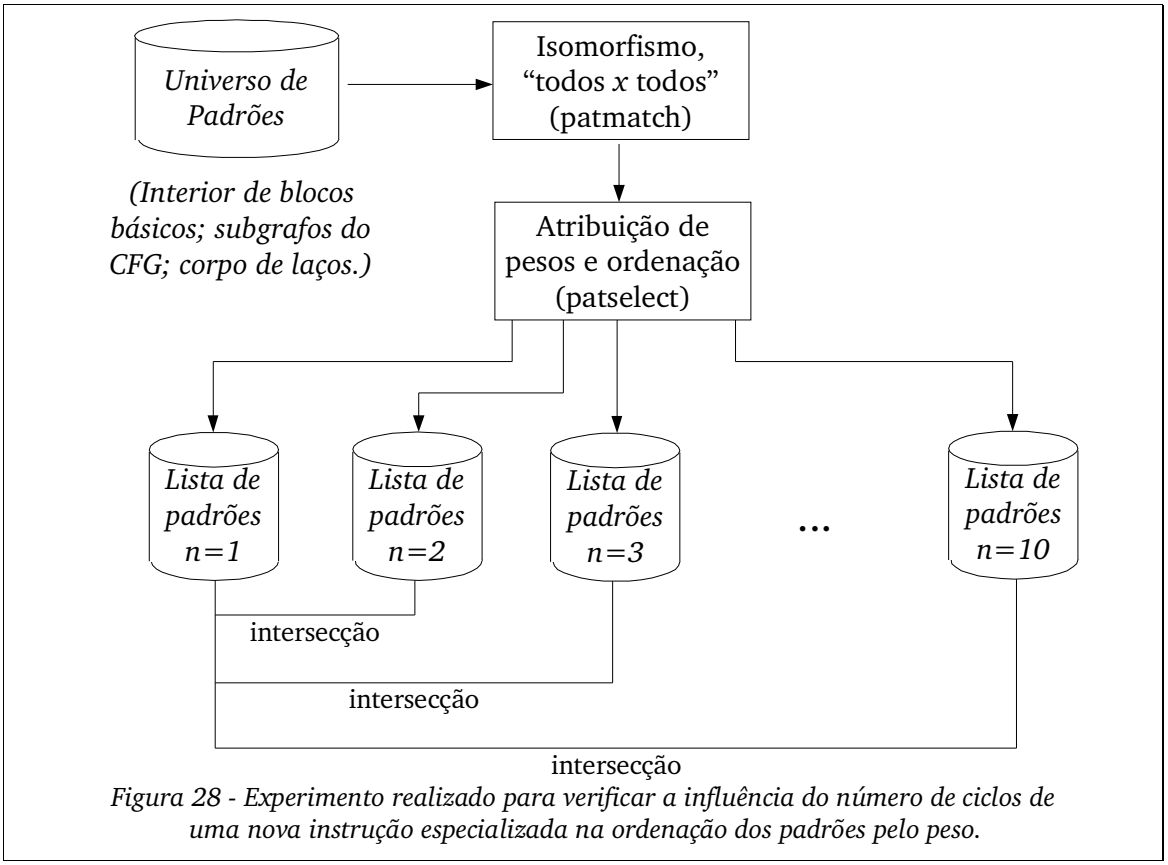


Figura 28 - Experimento realizado para verificar a influência do número de ciclos de uma nova instrução especializada na ordenação dos padrões pelo peso.

As Figuras 29 e 30 mostram dois gráficos com os resultados da intersecção das listas ordenadas para os 1000 padrões de maior peso de um universo de 54105 padrões (composto pelos padrões crescidos no interior de blocos básicos, padrões do tipo *subgrafo do CFG* delimitados por blocos básicos dominadores e pós-dominadores imediatos, e padrões correspondentes ao corpo de laços internos). O segundo gráfico revela que, quer a implementação de padrões em uma nova instrução requisesse 1 ciclo, ou quer a nova instrução requisesse 10 ciclos, mais de 85% dos padrões nas listas ordenadas seriam os mesmos. Isto vale para listas ordenadas de qualquer tamanho (número de padrões) até 1000 padrões; em geral, o uso prático destas listas não requereria que elas contivessem mais do que algumas dezenas de padrões. Assim, somos levados a concluir que a aproximação pela qual assumimos que todas as novas instruções implementadas no processador executarão em apenas 1 ciclo excluirá, no pior caso, 15% dos melhores padrões (de maior peso) para instruções que, após implementadas, requeiram até 10 ciclos para serem executadas. Caso as novas instruções requeiram até 4 ciclos, apenas 7% dos padrões de maior peso serão excluídos da lista ordenada em função de nossa aproximação.

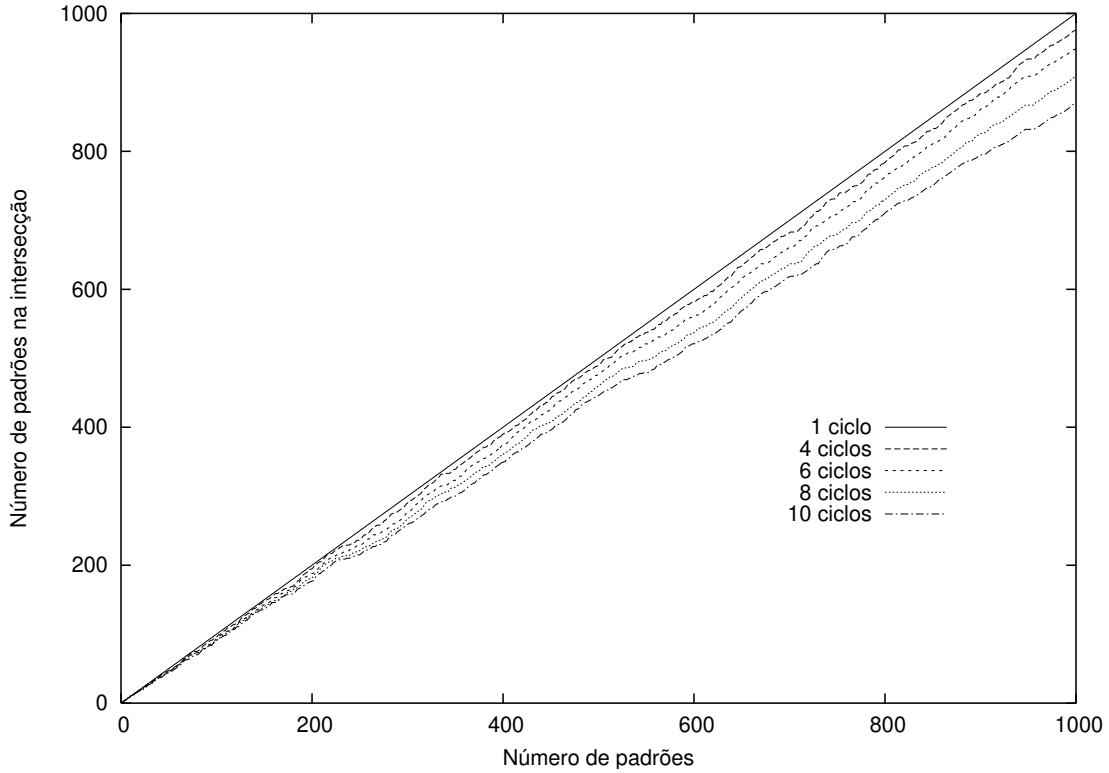


Figura 29 - Gráfico do número de padrões na interseção entre a lista de padrões ordenada pelo peso com n=1 ciclo e as outras listas com n>1. Veja também a Figura 28.

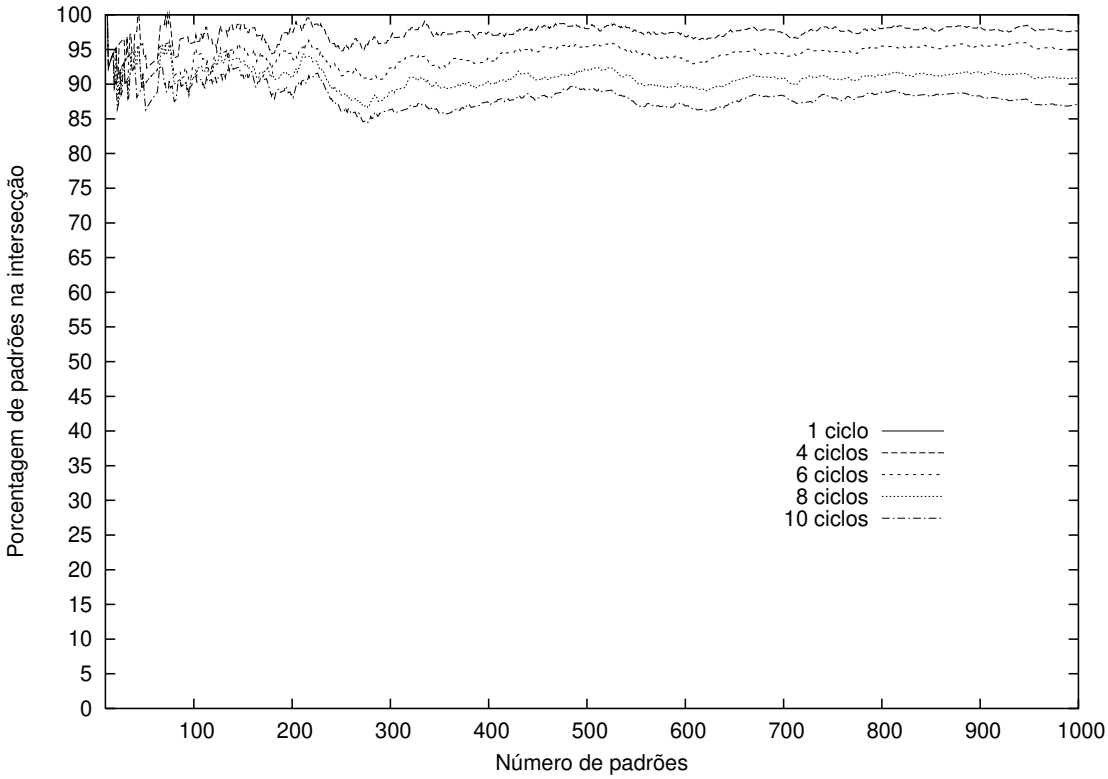


Figura 30 - Gráfico da porcentagem de padrões na interseção entre a lista de padrões ordenada pelo peso com n=1 ciclo e as outras listas com n>1. Veja também a Figura 28.

4.5. Identificação de padrões comuns a diversas aplicações

Na forma de um estudo de caso, elaboramos um procedimento para identificar os padrões de maior peso que se repetiam em diferentes aplicações dos *benchmarks*, com o intuito de permitir a criação de novas instruções no processador que pudessem ser úteis para conjuntos de aplicações e não apenas para uma única. Para isso, invertemos a ordem de aplicação das ferramentas *patmatch* e *patselect* (em comparação com a Figura 28), levando à nova configuração exibida na Figura 31. Para cada aplicação dos pacotes de *benchmarks*, selecionamos os padrões de maior peso que satisfaziam alguns filtros, e em seguida realizamos o casamento de padrões (isomorfismo de grafos CDFGs) para montar uma lista de padrões que se repetiam em diferentes aplicações, ordenada pelo número de repetições. Para que os dados gerados carregassem informação mais expressiva, as listas e gráficos produzidos para o resultado do casamento de padrões apresentam os padrões agrupados por *grupos isomorfos*, que são conjuntos de padrões isomorfos entre si mas com diferentes ocorrências no código das aplicações. A Figura 32 ilustra o conceito.

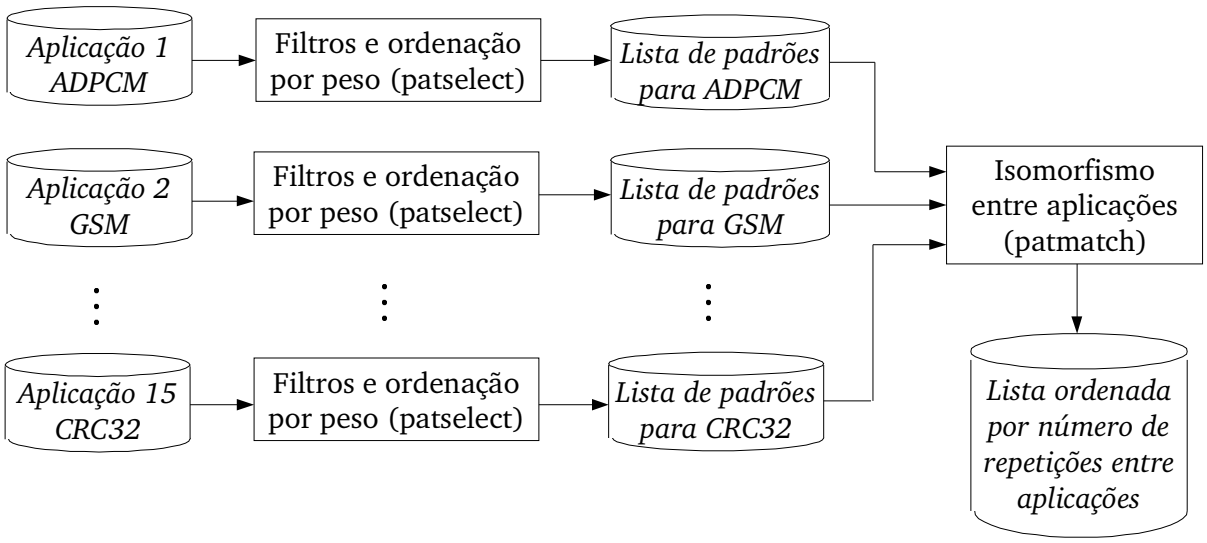


Figura 31 - Experimento para obtenção de uma lista de padrões bastante executados e que se repetem em várias aplicações dos benchmarks.

Numa versão do experimento, foram selecionados, para cada aplicação dos pacotes de *benchmarks*, os 100 padrões de maior peso que satisfaziam alguns filtros. Em outra versão, para cada aplicação foram selecionados todos os padrões que satisfaziam os filtros (e não apenas os padrões de maior peso), para posterior casamento. Esta última versão nos levou ao gráfico mostrado na Figura 33, que exibe o número de grupos de padrões isomorfos (conforme Figura 32) que possuem ocorrências em certa quantidade de aplicações. Um ponto numa a coordenada *(x,y)* indica que *y* grupos de padrões isomorfos possuem ocorrências em exatamente *x* aplicações, considerados apenas os padrões que satisfazem os filtros indicados na legenda. O gráfico contém 4 curvas, cada uma para um conjunto de padrões obtido pela aplicação de um determinado filtro (para uma das curvas, não foi aplicado qualquer filtro). Os filtros excluíam padrões de acordo com o número de operadores em cada padrão. Com os filtros, selecionamos padrões que possuíssem entre 2 e 20 operadores. A justificativa para o limite inferior de 2 operadores está relacionada com a inutilidade de se criar uma nova instrução que implemente um padrão de um único operador, pois o processador original certamente já seria capaz de executá-lo com alguma instrução. Os limites superiores 10 e 20 foram escolhidos para selecionar

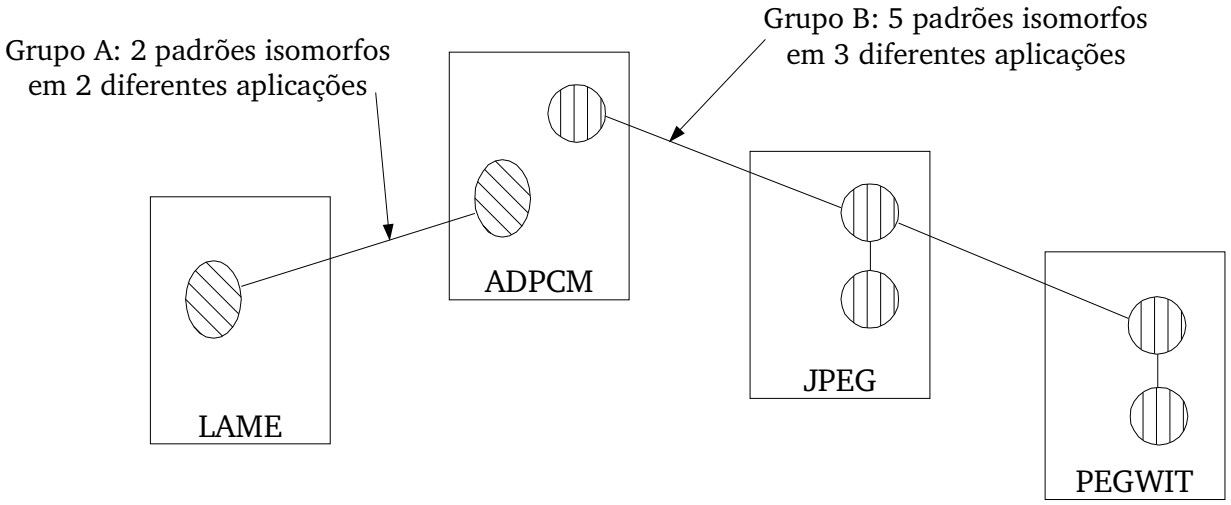


Figura 32 - Ilustração do conceito de grupos de padrões isomorfos. São mostrados dois grupos isomorfos, A e B. O grupo A possui duas ocorrências de padrões, em duas distintas aplicações. O grupo B possui cinco ocorrências de padrões, em três distintas aplicações.

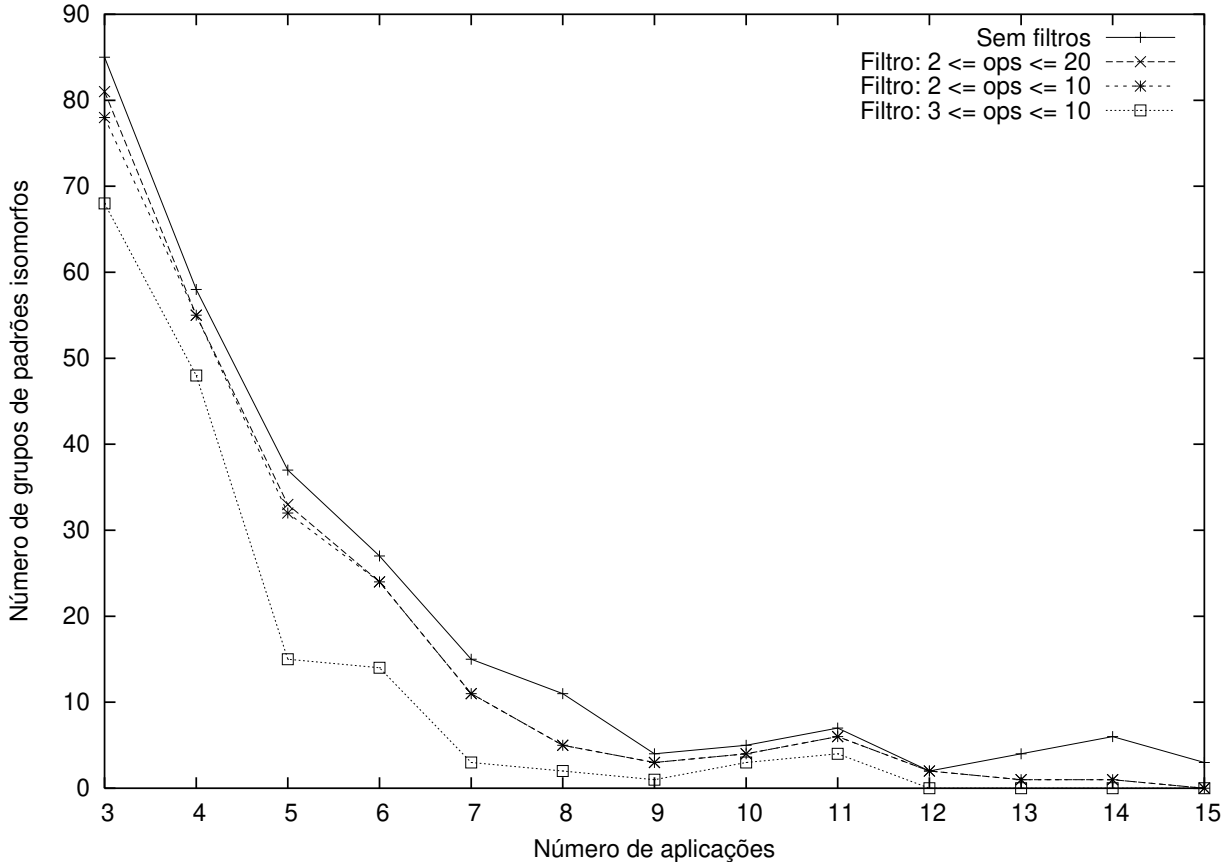


Figura 33 - Gráfico do resultado da comparação (isomorfismo) de padrões pelo experimento da Figura 31. A abreviação ops *na legenda se refere ao número de operadores em cada padrão.*

padrões relativamente pequenos, que pudessem ser implementados em um ou poucos ciclos com uma única nova instrução. As escolhas dos limites nos números de operadores dos filtros foi também apoiada pelo gráfico da Figura 34, que mostra a distribuição do universo de padrões pelo número de operadores de cada padrão. Como pode ser observado, mais de um terço dos padrões possuem apenas um operador, 77% dos padrões possuem 10 ou menos operadores, e 81% dos padrões possuem 20 ou menos operadores. O gráfico também mostra que 13% dos padrões se concentram numa faixa entre 89 e 95 operadores; todos estes são padrões que possuem um operador *call* que, na representação intermediária do GCC, vem sempre associado a aproximadamente 90 operadores do tipo *clobber*, utilizados para indicar que os registradores do processador podem ser modificados pela chamada à função. Como não estamos interessados em considerar padrões que possuam um operador *call*, visto que tais padrões não seriam implementados em *hardware*, filtramos também estes padrões. Apenas 6% dos padrões possuem mais do que 20 operadores e estão fora da faixa dos operadores com *call*.

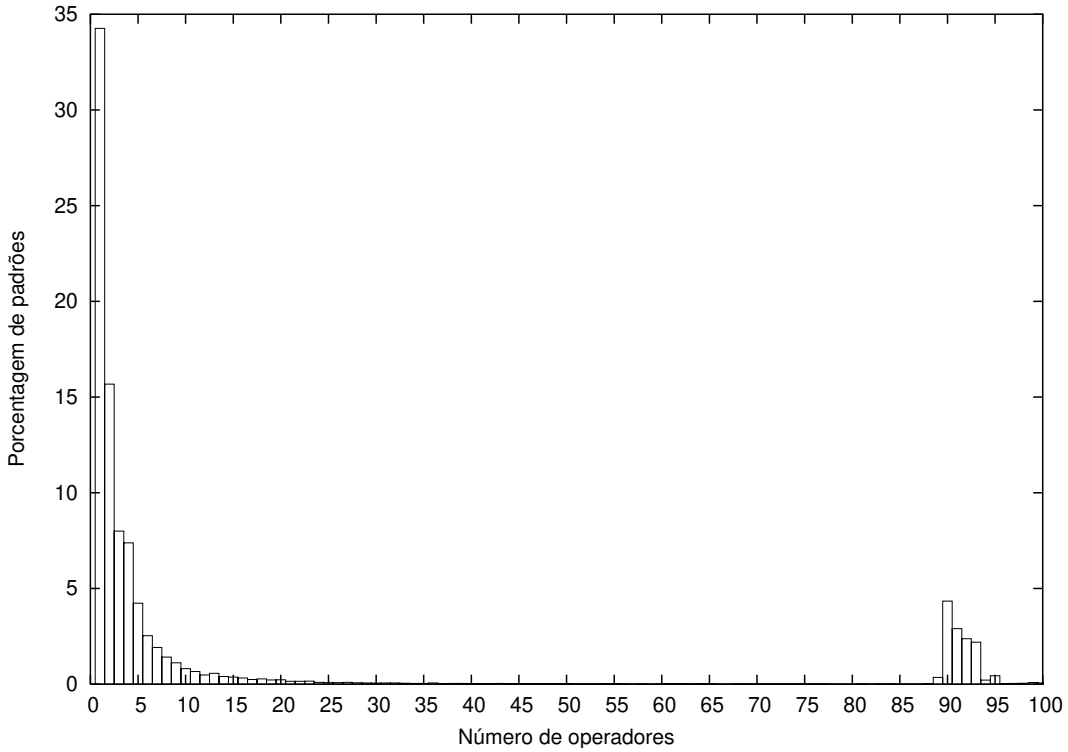


Figura 34 - Gráfico da distribuição do universo de padrões pelo número de operadores em cada padrão. Os padrões na faixa entre 89 e 95 operadores possuem todos um operador call *que, na representação intermediária do GCC, vem sempre acompanhado de aproximadamente 90 operadores do tipo* clobber *que indicam os registradores do processador que podem ser modificados pela chamada à função.*

A seguir, são examinadas as características dos grupos de padrões isomorfos que levaram ao gráfico da Figura 33.

A Figura 35 mostra os principais padrões que se repetem em 12 ou mais aplicações (do total de 15 aplicações consideradas), sem a aplicação de qualquer filtro. Além dos padrões de 1 ou 2 operadores mostrados na figura, alguns padrões contendo o operador *"call"* (chamada de função) foram encontrados ocorrendo em até 14 aplicações (por exemplo, uma chamada para a função *fprintf ()*). Notamos que, infelizmente, nenhum destes padrões é interessante para implementação em *hardware* numa nova instrução, pois os processadores RISC em geral já possuem instruções que

implementam as computações neles expressas – e em todos os demais padrões identificados que contêm apenas 1 ou 2 operadores. O padrão do item *(h)* da Figura 35, se implementado juntamente ao teste da condição de término dos laços, pode levar à implementação de uma instrução ZOL *(Zero Overhead Loop)*, que é conhecida por trazer bons ganhos de desempenho [UWW+99].

Partimos então para uma investigação dos padrões que contêm 3 ou mais operadores. Para restringir a quantidade de padrões, foram analisados apenas os padrões que se repetem em 3 ou mais aplicações e que possuem entre 3 e 10 operadores – uma das curvas do gráfico da Figura 33. Do universo original de 54105 padrões, 7219 padrões (13%) satisfazem aquelas restrições, divididos em 158 grupos isomorfos (conforme Figura 32). Olhando para os grafos (CDFGs) gerados para cada grupo isomorfo e inspecionando o código fonte de onde alguns de seus padrões foram extraídos, foi possível classificar os grupos segundo 41 categorias funcionais. Dentre estas, 12 categorias foram consideradas interessantes para uma implementação em *hardware,* pelos critérios:

- Um padrão que execute um *load* ou *store* e mais uma ou duas instruções lógicas/aritméticas não é um bom candidato, pois o acesso à memória requer vários ciclos (exceto nos casos especiais citados nas categorias e *MemCpy* e *MemSet*);
- Em oposição, um padrão composto de algumas instruções lógicas e aritméticas que possam ser implementadas em menos ciclos numa nova instrução é um bom candidato (especialmente se contiver *shift* por constante, *Subreg* e *ZEx/SEx*, que são facilmente implementados em *hardware*);
- Padrões cuja computação envolva poucos operandos de entrada e tenham poucos operandos de saídas são preferíveis, pois os bancos de registradores dos processadores RISC usuais permitem, por ciclo, a leitura de dois registradores e a escrita de um registrador.

Estas 12 categorias são apresentadas na Tabela 4. As Tabelas 5 e 6 mostram as demais categorias, que não foram consideradas interessantes para a implementação em *hardware*. Nestas tabelas, a coluna *NA* (número de aplicações) contém o maior número de aplicações distintas em que ocorre um mesmo grupo isomorfo pertencente à categoria. A coluna *NG* (número de grupos) contém a quantidade de grupos isomorfos distintos pertencentes à categoria. O valor *NG* também pode ser interpretado como o número de variações encontradas nos padrões da categoria. Por exemplo, a categoria *Ld-Cmp-Brn* (Tabela 5) possui *NA*=6 e *NG*=23. Isto significa que ela possui 23 variações em operandos e operadores da seqüência *load-compare-branch*, tais como: *branch if equal to; branch if greater than; branch if greater than or equal to; branch if greater unsigned; compare to constant; compare to register;* mova (ou não) um valor entre registradores antes de executar um *load*; monte (ou não) o valor de um endereço utilizando os operadores *high* e *lo_sum*; etc. Cada uma destas variações ocorre em, no máximo 6 aplicações. A coluna *descrição das computações* exibe as computações dos principais operadores encontrados nos grupos da categoria. A semântica dos operadores é explicada no próprio texto da tabela, podendo ser ainda consultada a Tabela 2 para mais informações. A coluna *F* contém o número da figura que ilustra um grafo (CDFG) de um padrão representativo da categoria (as figuras aparecem após as tabelas). Nos grafos, os vértices em formato retangular representam operandos (constantes ou registradores), e os vértices em formato elíptico representam operadores (instruções da representação intermediária do GCC). As arestas são anotadas com rótulos como p*n*, onde *n* é um número. Esta notação indica a *porta* de entrada da aresta num operador. Por exemplo, no operador *st* (*store)*, a porta de entrada 0 (p0) carrega o endereço de memória a ser escrito, e a porta de entrada 1 (p1) carrega o valor a ser escrito.

Categoria	*NA*	*NG*	*F*	*Descrição das principais computações*
Ld Ptr ++	6	1	36	Carrega o conteúdo de uma posição de memória através de um apontador, que é incrementado em seguida. Aparece em todas as chamadas às funções *getc(), getchar()* com código otimizado com a opção "-O3". Pode ser interessante implementar este padrão em *hardware* se o processador for capaz de efetuar aritmética em RAM, como é o caso de alguns processadores CISC e microcontroladores.
Ld-ZEx-Cmp-Brn	4	1	41	Lê de uma posição de memória, efetua *sign extension* e *zero extract* (operador do GCC que indica a extração de um conjunto de *bits* de uma palavra), e em seguida *compare* e *branch*. Devido ao fato de as operações *sign extension* e *zero extract* serem muito facilmente implementadas em *hardware*, pode ser um padrão bom candidato a uma nova instrução.
MemCpy	4	1	42	Copia uma região de memória para outra, incrementando dois apontadores independentes que apontam para as regiões. Alguns processadores CISC, como o Intel x86, provêem algum tipo de suporte para esta operação.
MemCpy-ZOL	3	1	42	Semelhante a MemCpy, acrescido de uma variável de iteração de laço com operadores *plus, compare* e *branch* que podem ser implementados numa instrução ZOL (*Zero Overhead Loop*) (veja categoria ZOL).
MemSet	4	1	43	Escreve uma constante numa região de memória *(array)* através de um apontador que é incrementado a cada iteração do laço. O controle do laço é feito por uma variável separada, com operadores *plus, compare* e *branch*, podendo ser implementado com uma instrução ZOL (*Zero Overhead Loop*).
MemSet-ZOL	3	1	43	Combinação das categorias MemSet e ZOL, utilizando a variável de iteração como apontador para escrever o conteúdo de um registrador numa região de memória *(array)*.
Mov-Or-Plus	4	3	44	Um operador *move* seguido de uma computação *or* e *plus* em seqüência. A implementação de uma única nova instrução que execute um *or* seguido de um *plus*, com dois registradores de entrada e um de saída e uma pequena constante num campo imediato, pode ser interessante.
Shf-Plus-Shf	4	1	45	Operador *shift* seguido de um *plus* e outro *shift*. Possui duas pequenas constantes de entrada para os *shifts* (5 bits), um registrador de entrada e outro de saída. Interessante para implementar em *hardware*.
Shf-Shf-Mov	3	1	46	Semelhante a Shf-Plus-Shf, sem o *plus*.
Subreg-Plus-Subreg	4	1	47	Soma apenas uma parte de uma palavra (por exemplo, o *byte* menos significativo). Instruções com essa capacidade são comuns em extensões multimídia como MMX e SSE. Se o processador original não oferecer uma forma eficiente de fazer isso, a implementação de uma nova instrução pode ser vantajosa.
ZEx-Cmp-Brn	5	2	47	Operação *zero extract* seguida de um *compare* e *branch*. Como esta operação pode ser facilmente implementada em *hardware*, sua implementação numa nova instrução pode ser interessante.
ZOL	11	11	47	Operação de controle de laço que consiste em incrementar (ou decrementar) uma variável de iteração, compará-la com uma constante e efetuar um *branch*. Uma instrução que implemente esta operação é conhecida como ZOL – *Zero Overhead Loop*. O processador SPARC (Leon) não possui uma instrução que faça isso.

Tabela 4 - Categorias definidas para as computações dos grupos de padrões isomorfos encontrados pelo experimento da Figura 31. Esta tabela mostra as categorias que foram consideradas interessantes para serem implementadas em hardware. *As Tabelas 5 e 6 mostram as demais categorias.*
NA = Número de Aplicações, NG = Número de Grupos, F = Figura.

Categoria	***NA***	***NG***	***F***	***Descrição das principais computações***
And-Cmp-Brn	3	3	36	Operação *AND*, seguida de *compare* e *branch*. Computação já implementada pela instrução SPARC *and_cc*, que realiza o AND e modifica as flags do registrador CC (*Condition Codes*).
Expressão Constante	4	2	-	Somas seguidas de um operador *const*, gerado pelo GCC para indicar que o valor não mudará após computado. Não interessante para *hardware* porque são padrões executados poucas vezes.
Ld Ptr	9	12	36	Carrega o conteúdo de uma posição de memória através de um apontador, possivelmente montando a constante do endereço com os operadores *high* e *lo_sum*.

Ld Struct	10	2	36	Efetua uma indireção por apontador e carrega o conteúdo de um endereço de memória deslocado de uma constante, como no acesso de um campo de uma estrutura a partir de um apontador para ela. Por exemplo:
				`mystruct_t *mystruct;`
				`mystruct->size;`

Ld Struct, Ld-Ld-Cmp-Brn	4	1	37	Semelhante a Ld Struct, efetuando em seguida mais dois *loads*, *compare* e *branch*.
Ld Struct, Ld-Plus-St	5	1	37	Semelhante a Ld Struct, efetuando em seguida *load, plus, store*.
Ld Struct, St	3	1	38	Semelhante a Ld Struct, efetuando um *store* em seguida.
Ld-Cmp-Brn	6	23	37	Lê uma posição de memória através de um apontador, e com o resultado efetua *compare* e *branch*.
Ld-Ld	4	2	36	Dois *loads* com um endereço base ou algum outro operando em comum.
Ld-Ld-Cmp-Brn	3	2	38	Efetua dois *loads* em paralelo, e em seguida *compare* e *branch*.
Ld-Ld-Ld	3	2	38	Três *loads* com um endereço base ou algum outro operando em comum.
Ld-Plus-St	4	6	39	Carrega uma posição de memória, efetua uma soma e escreve novamente na memória.
Ld-Plus-St-Cmp-Brn	5	3	39	Semelhante a Ld-Plus-St, e em seguida efetua um *compare* e *branch*.
Ld-SEx	4	3	40	Lê uma posição de memória e efetua *sign extension*, como se convertendo de 32 para 8 *bits* ou vice-versa. Padrões freqüentemente acompanhados de operadores *move*.
Ld-Ld-SEx	3	1	40	Semelhante a Ld-SEx, mas com dois *loads*.
Ld-Shf-Cmp-Brn	3	1	40	Executa um *load* seguido de *shift, compare* e *branch*.
Ld-St	4	7	39	Lê de uma posição de memória e escreve em outra.
Ld-St-Cmp-Brn	3	1	41	Lê de uma posição de memória e escreve em outra, e em seguida executa *compare* e *branch*.

Tabela 5 - Categorias definidas para as computações dos grupos de padrões isomorfos encontrados pelo experimento da Figura 31. Esta tabela mostra as categorias que não foram consideradas interessantes para a implementação em hardware. *NA = Número de Aplicações, NG = Número de Grupos, F = Figura. Continua na Tabela 6...*

Categoria	***NA***	***NG***	***F***	***Descrição das principais computações***
Mov-Cmp-Brn	11	20	44	Operador *move* seguido de *compare* e *branch*.
Mov-Mov-Use	4	1	44	Dois operadores *move* em seqüência, e um operador *use* usado pelo GCC para indicar que um registrador está reservado e não deve ser escolhido pelo alocador de registradores.
Mov-Or-Mov	4	1	44	Um operador *move* seguido de um operador *or* e outro operador *move*.
Mov-Or-Or	3	1	44	Um operador *move* seguido de dois operadores *or* executados em paralelo. Inapropriado para implementação de nova instrução porque requer três entradas e dois registradores de saída.
Mov-Plus-Mov	4	2	44	Uma computação seqüencial com *move*, *plus* e *move* novamente.
Or-Cmp-Brn	4	4	45	Operador *or* seguido de *compare* e *branch*. Computação já implementada pela instrução *or_cc* do SPARC, que executa a operação *or* e modifica o registrador CC (*Condition Codes*).
Plus-Cmp-Brn	3	4	45	Operador *plus* seguido de *compare* e *branch*. Computação já implementada pela instrução *add_cc* do SPARC, que executa uma soma e modifica o registrador CC (*Condition Codes*).
Shf-Cmp-Brn	3	3	45	Operador *shift* seguido de *compare* e *branch*. Computação já implementada pela instrução *shift_cc* do SPARC, que executa o deslocamento e modifica o registrador CC (*Condition Codes*).
Shf-Plus-Plus	4	1	45	Operador *shift* seguido de duas somas. Não é interessante para implementar em *hardware* porque possui 4 entradas e 2 saídas (a menos que um banco de registradores de várias portas estivesse disponível).
St Ptr	6	21	45	Escreve em uma posição de memória a partir de um apontador, possivelmente utilizando os operadores *high* e *lo_sum* para montar uma constante de 32 bits.
St Ptr, Cmp-Brn	4	2	46	Escreve um valor em uma posição de memória, compara o valor com zero e faz um *branch*.

Tabela 6 - Continuação da Tabela 5. NA = Número de Aplicações, NG = Número de Grupos, F = Figura.

Neste capítulo, foram apresentados os experimentos realizados para a verificação dos métodos propostos e discutidos no capítulo anterior. Pudemos identificar e classificar categorias de padrões que são comuns a diversas aplicações multimídia, diversas delas interessantes para a implementação na forma de novas instruções no processador. Concluímos que as aproximações consideradas na fórmula para cômputo do peso dos padrões produzem resultados razoáveis que são úteis como estimativas do ganho a ser conseguido com a implementação dos padrões em *hardware*. Até o momento, apenas pequenas experiências, semi-automatizadas, foram realizadas na forma de implementações de novas unidades funcionais e instruções no processador Leon. Outros pesquisadores do LSC deverão utilizar os resultados obtidos nesta dissertação para as próximas implementações a serem realizadas. O próximo capítulo apresenta as conclusões gerais da dissertação, identifica algumas melhorias que podem ser feitas no trabalho realizado e aponta trabalhos futuros na linha de pesquisa do Projeto Chameleon.

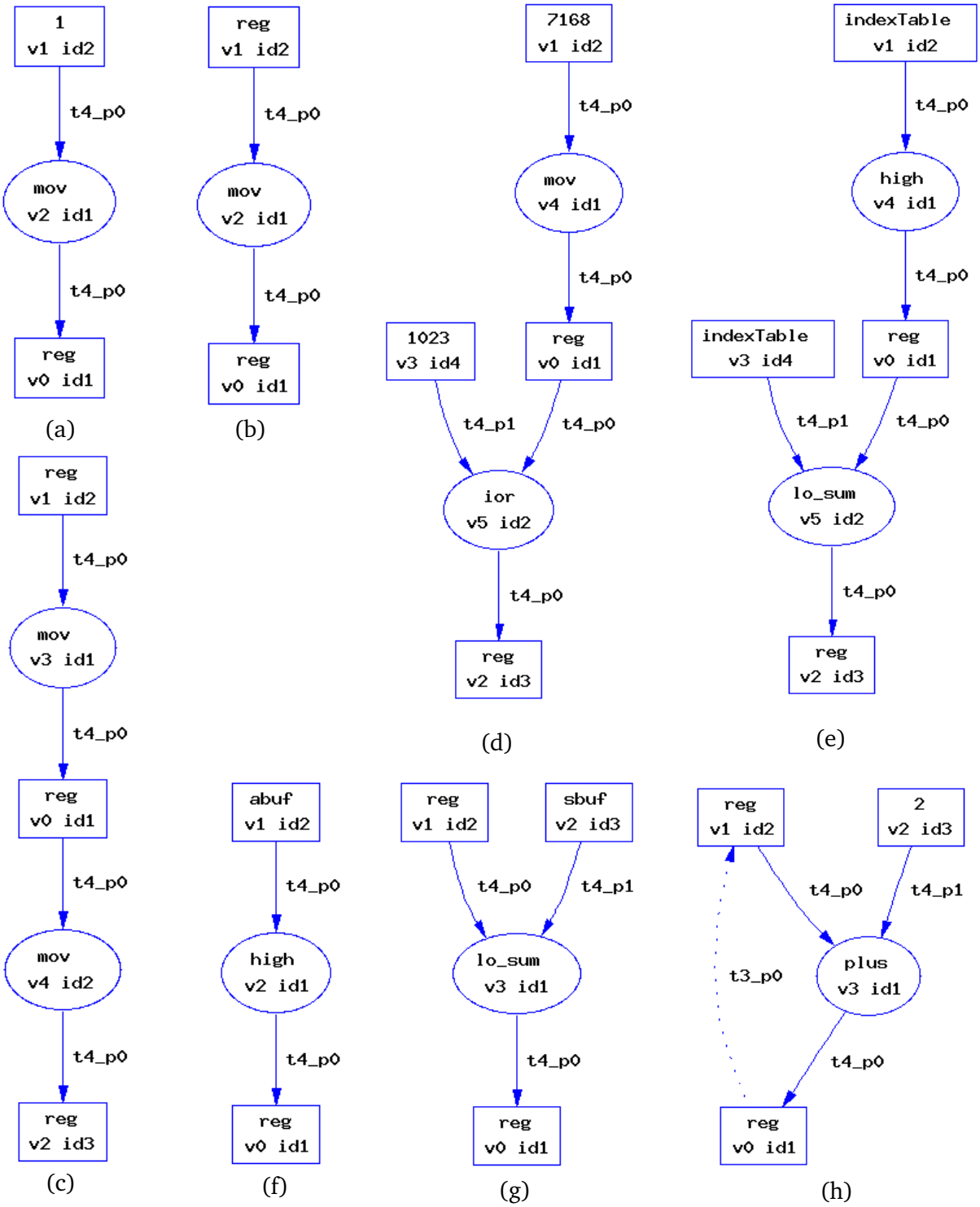


Figura 35 - Padrões que se repetem em 12 ou mais aplicações. Em (a), (b) e (c), padrões contendo apenas operações "mov"; em (d), a operação "mov" é usada para permitir um "or" de constantes inteiras (ior); em (e), (f) e (g), constantes de 32 bits são montadas em registradores somando os 16 bits altos (high) com os 16 bits baixos (lo_sum); em (h), o incremento de de uma variável de iteração em laços (em pontilhado, dependência loop-caried). Obs.: constantes como "1", "2", "1023" e "7168" não são as mesmas nas diferentes aplicações; o algoritmo de isomorfismo foi configurado para casar quaisquer constantes de mesma largura de bits.

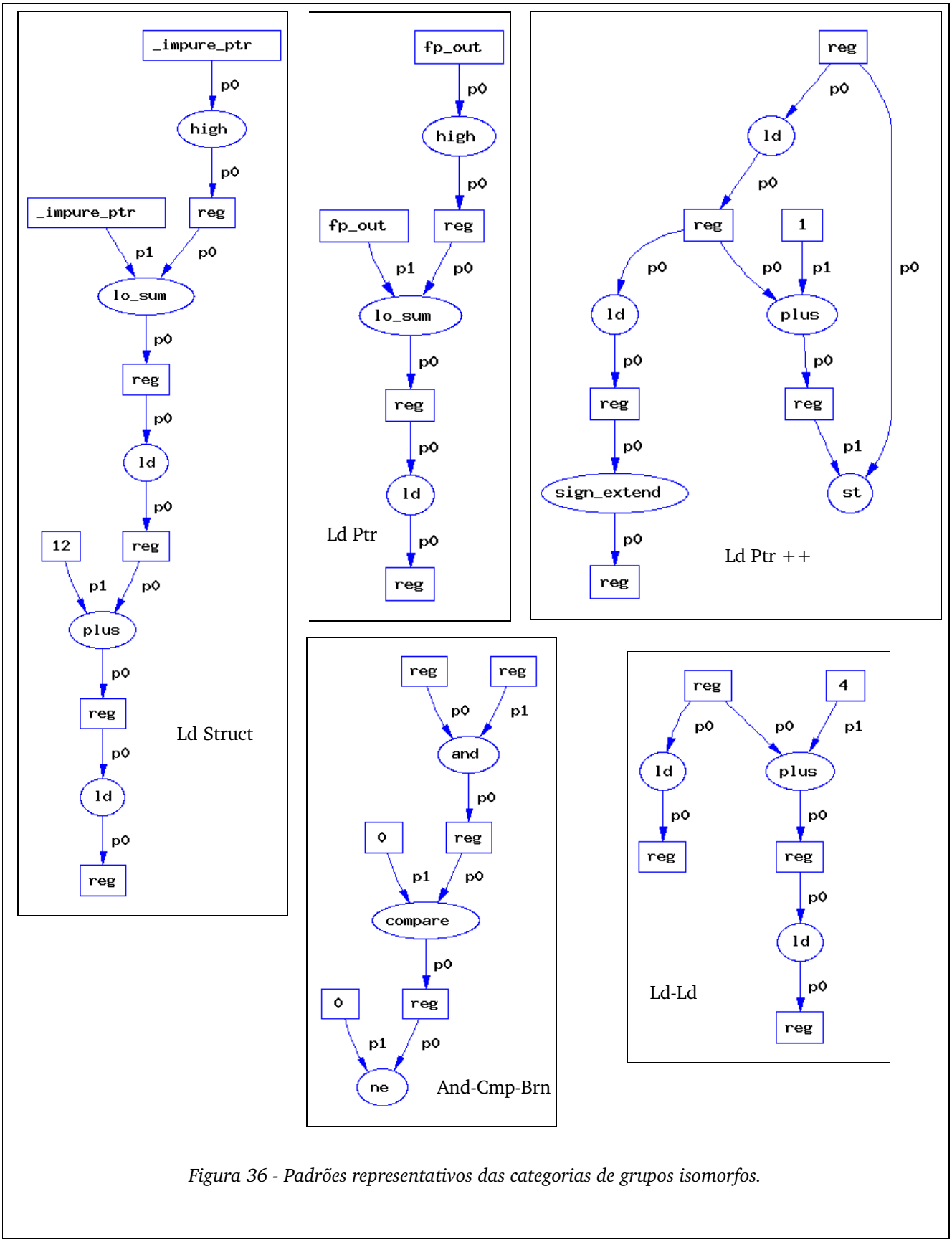


Figura 36 - Padrões representativos das categorias de grupos isomorfos.

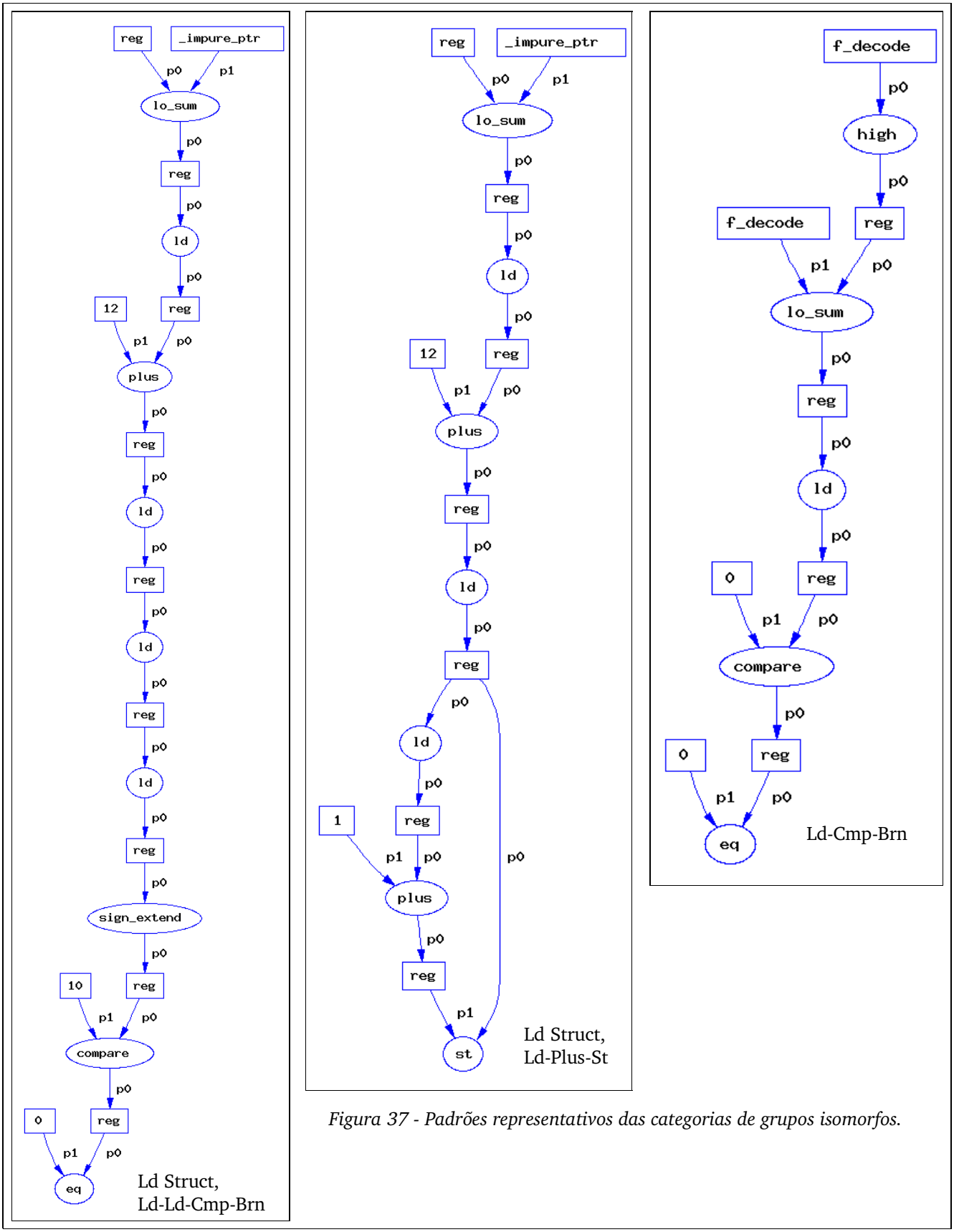


Figura 37 - Padrões representativos das categorias de grupos isomorfos.

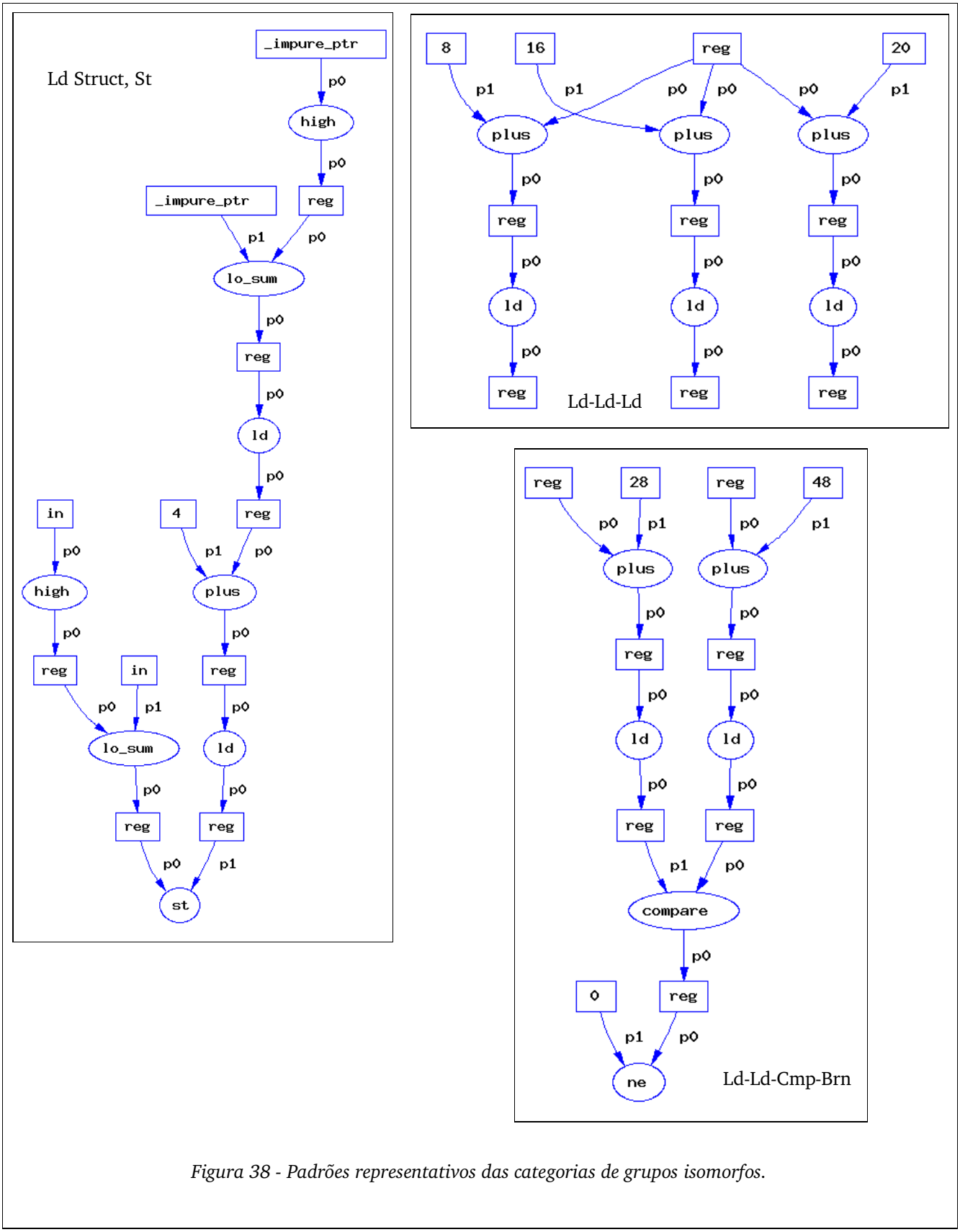
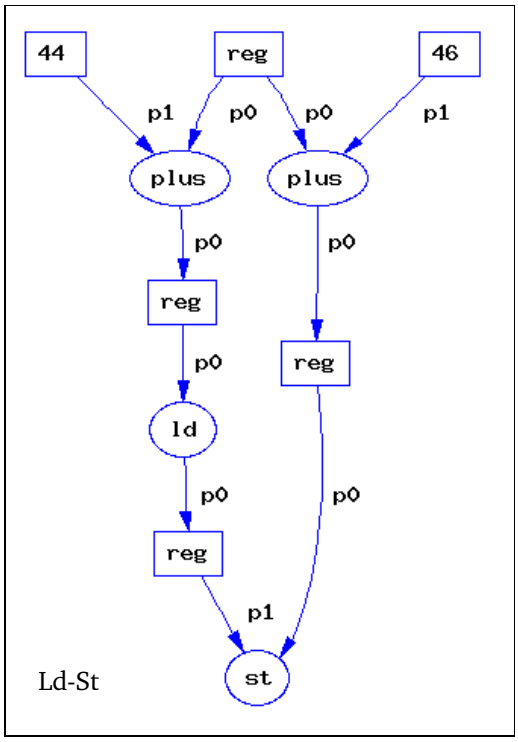
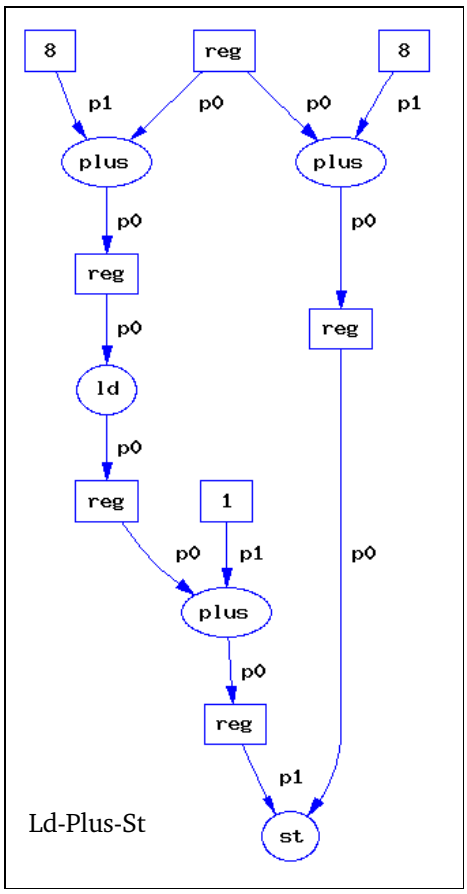


Figura 38 - Padrões representativos das categorias de grupos isomorfos.





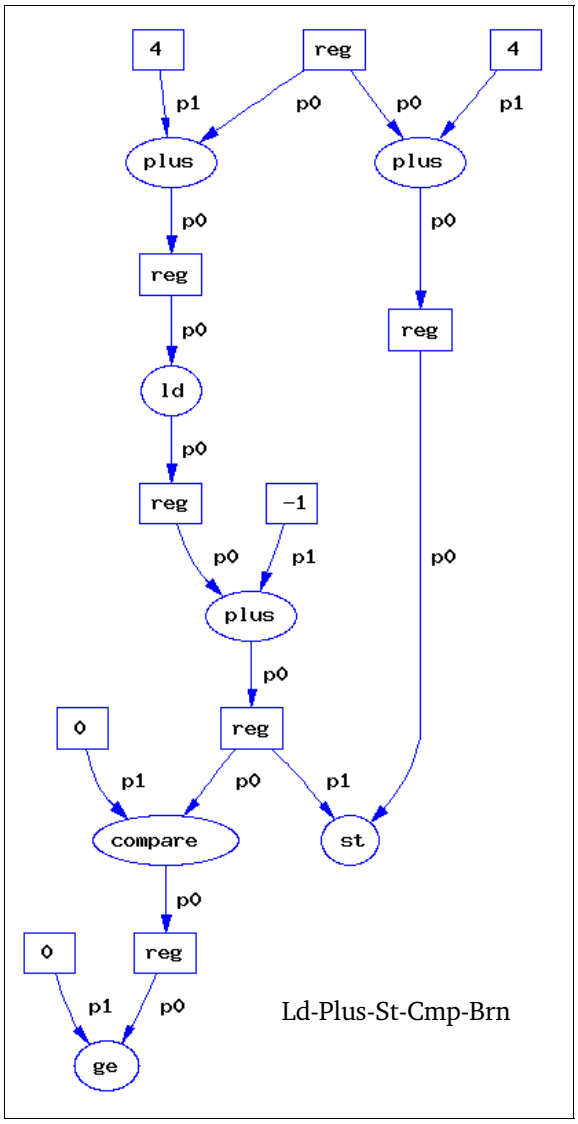


Figura 39 - Padrões representativos das categorias de grupos isomorfos.

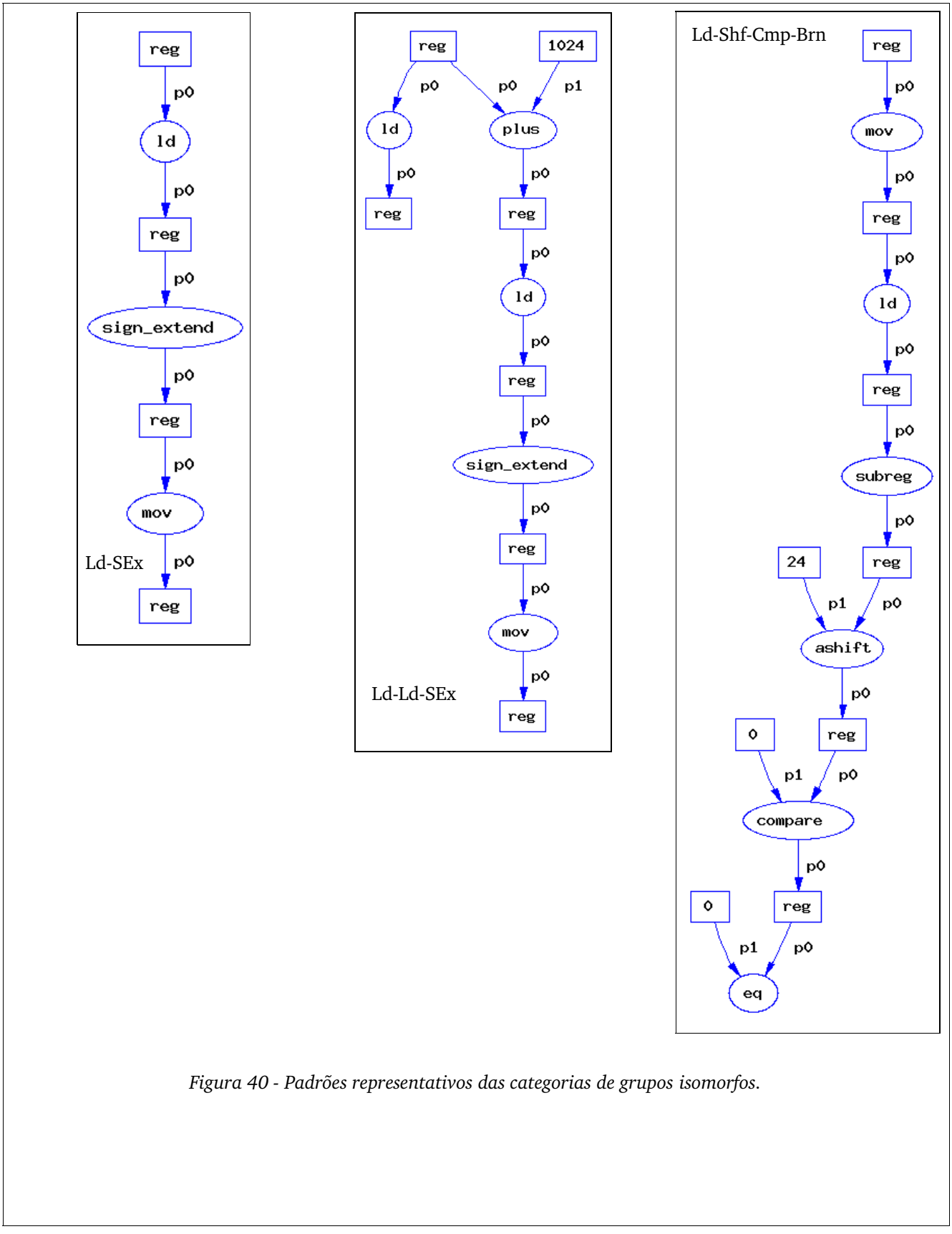


Figura 40 - Padrões representativos das categorias de grupos isomorfos.

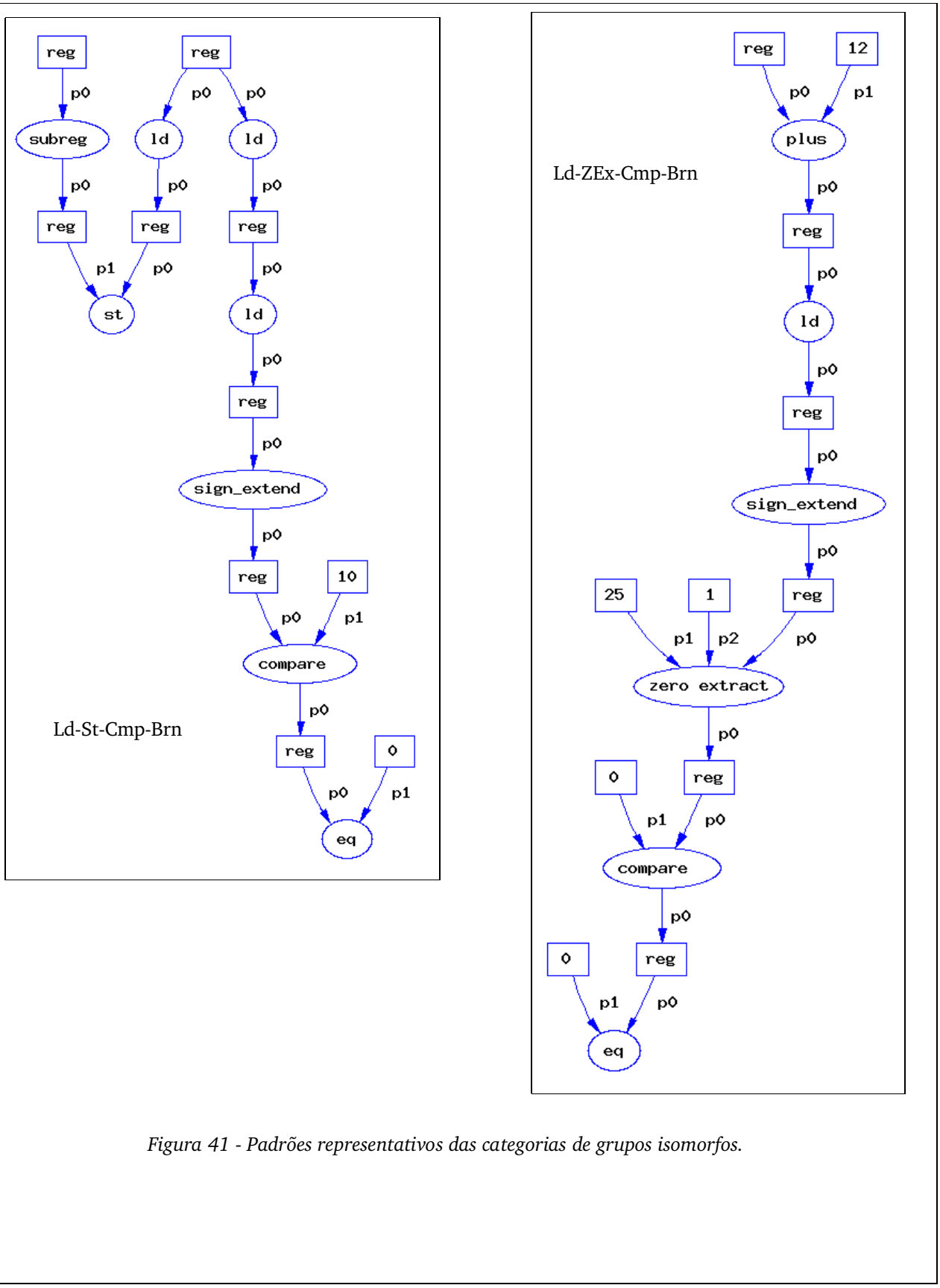
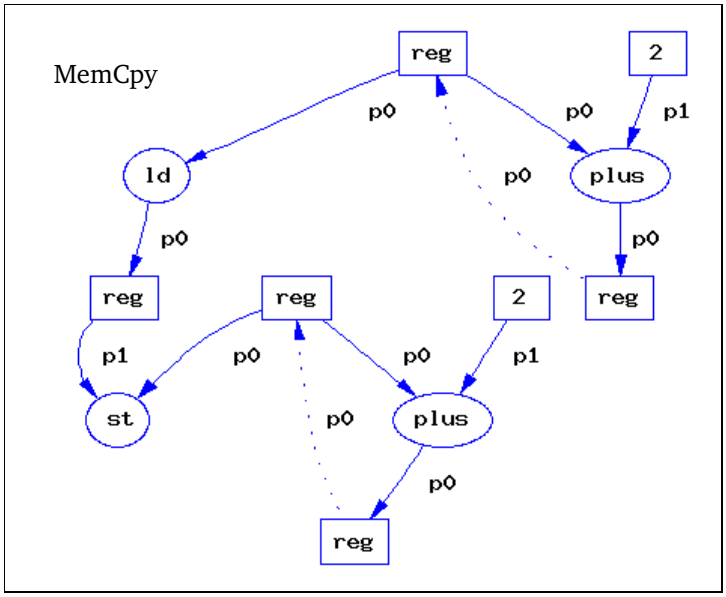


Figura 41 - Padrões representativos das categorias de grupos isomorfos.



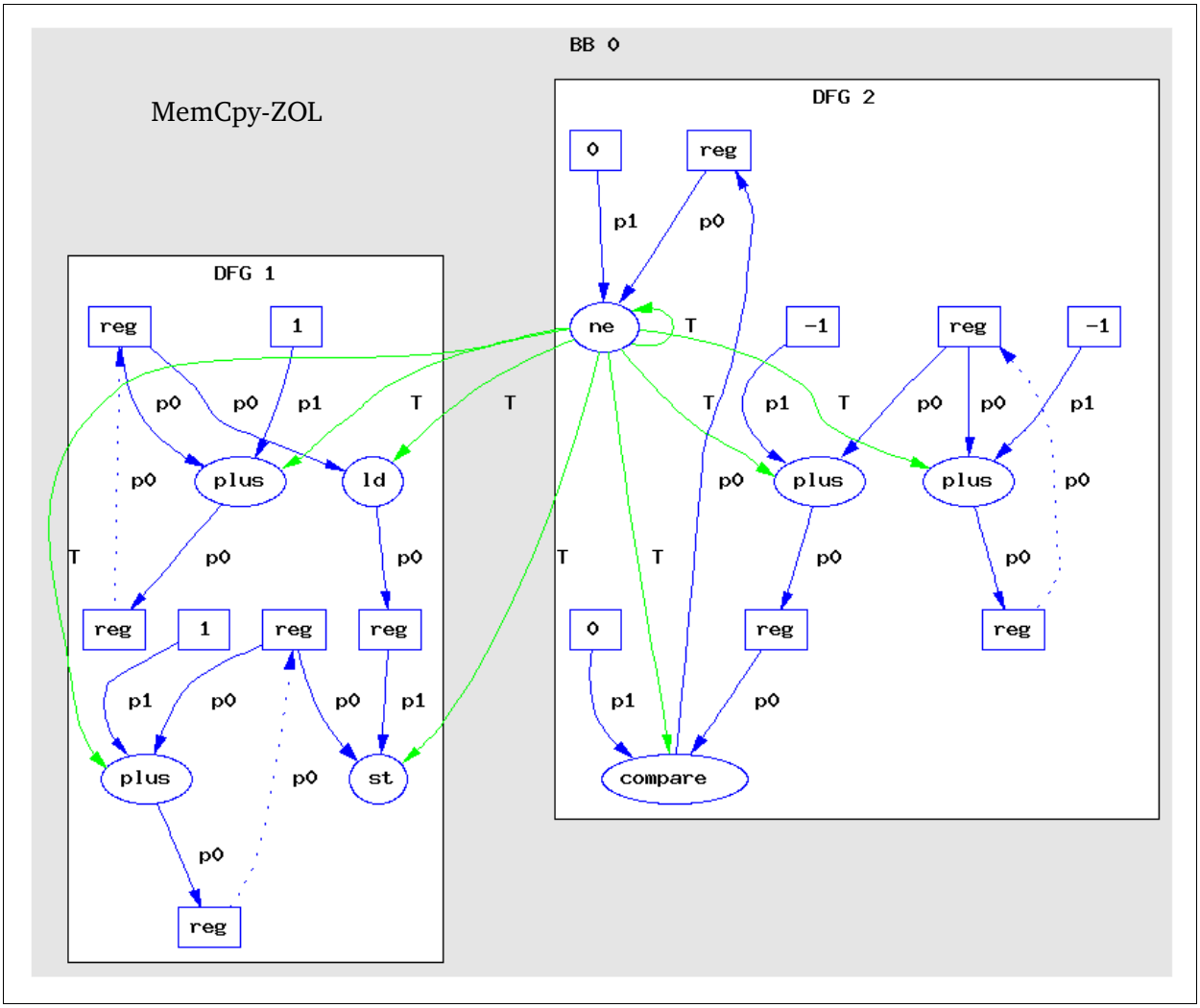


Figura 42 - Padrões representativos das categorias de grupos isomorfos.

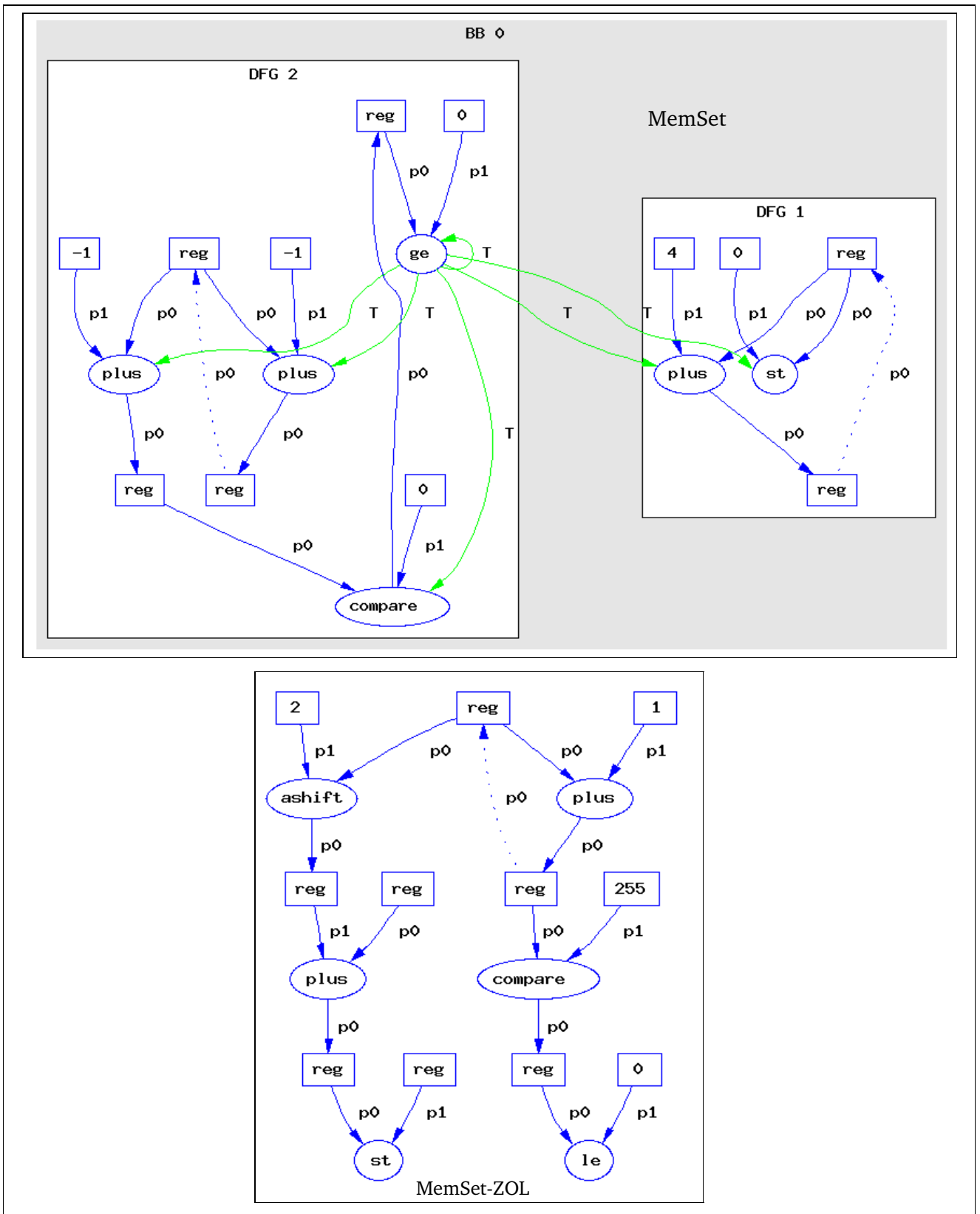


Figura 43 - Padrões representativos das categorias de grupos isomorfos.

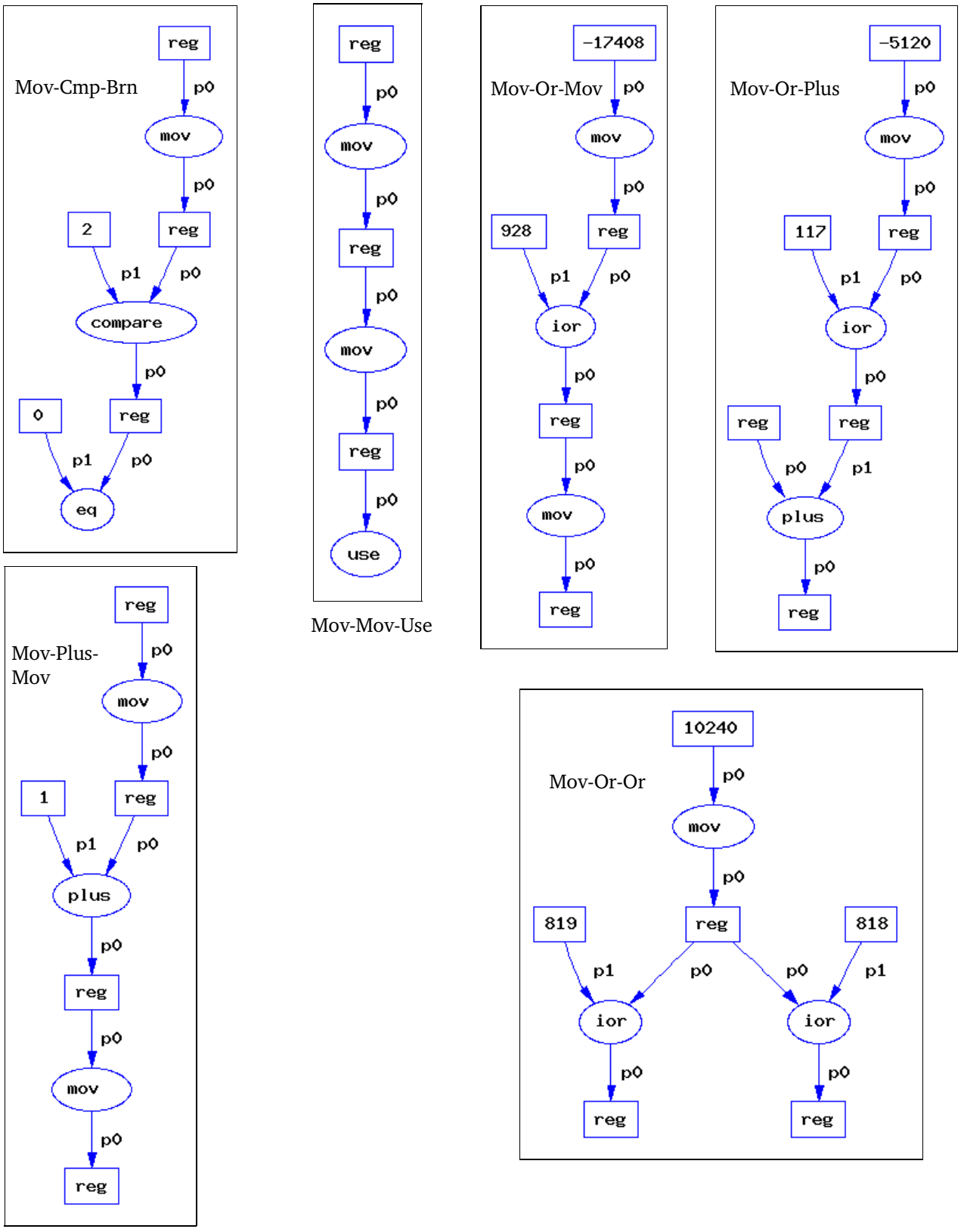


Figura 44 - Padrões representativos das categorias de grupos isomorfos.

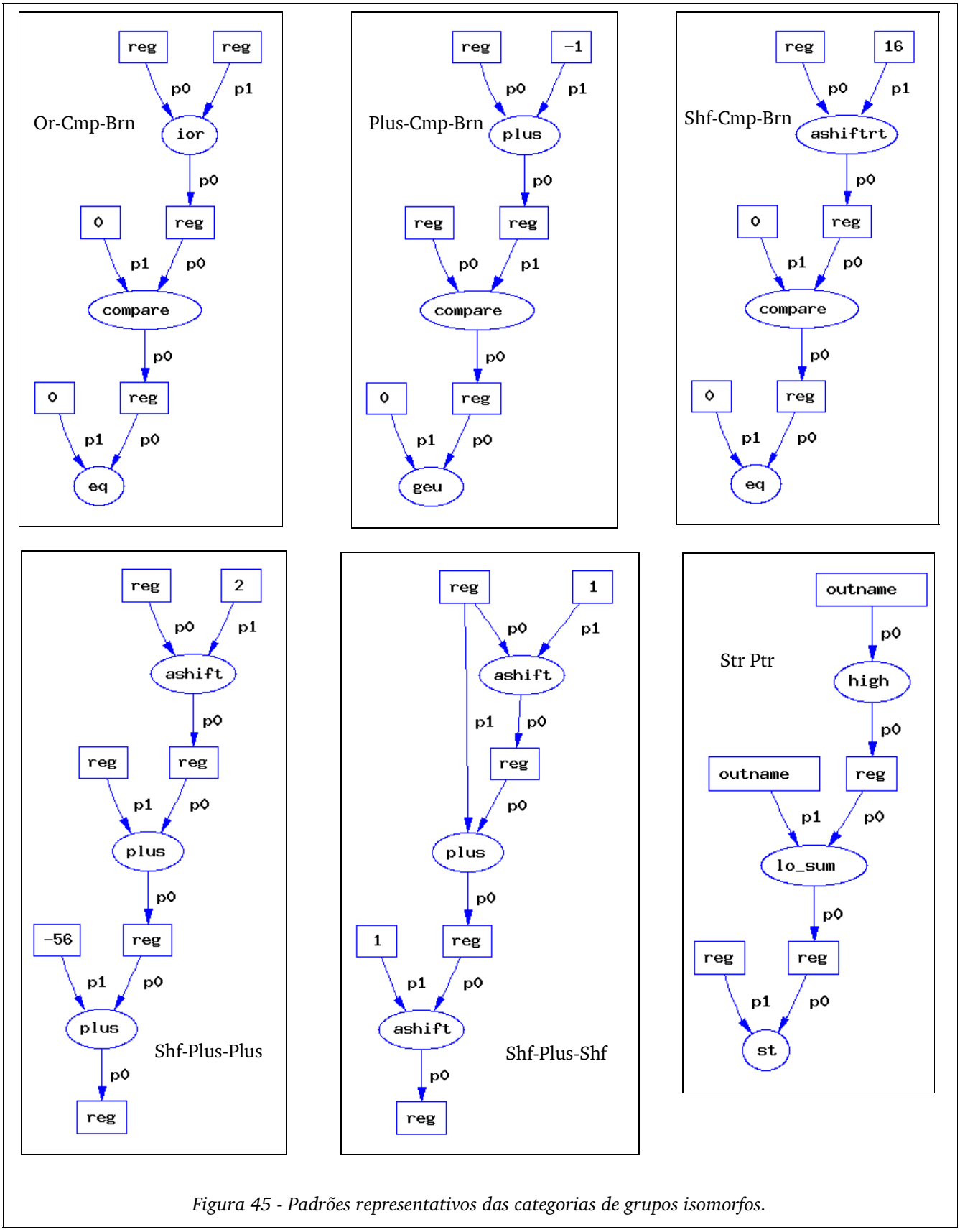


Figura 45 - Padrões representativos das categorias de grupos isomorfos.

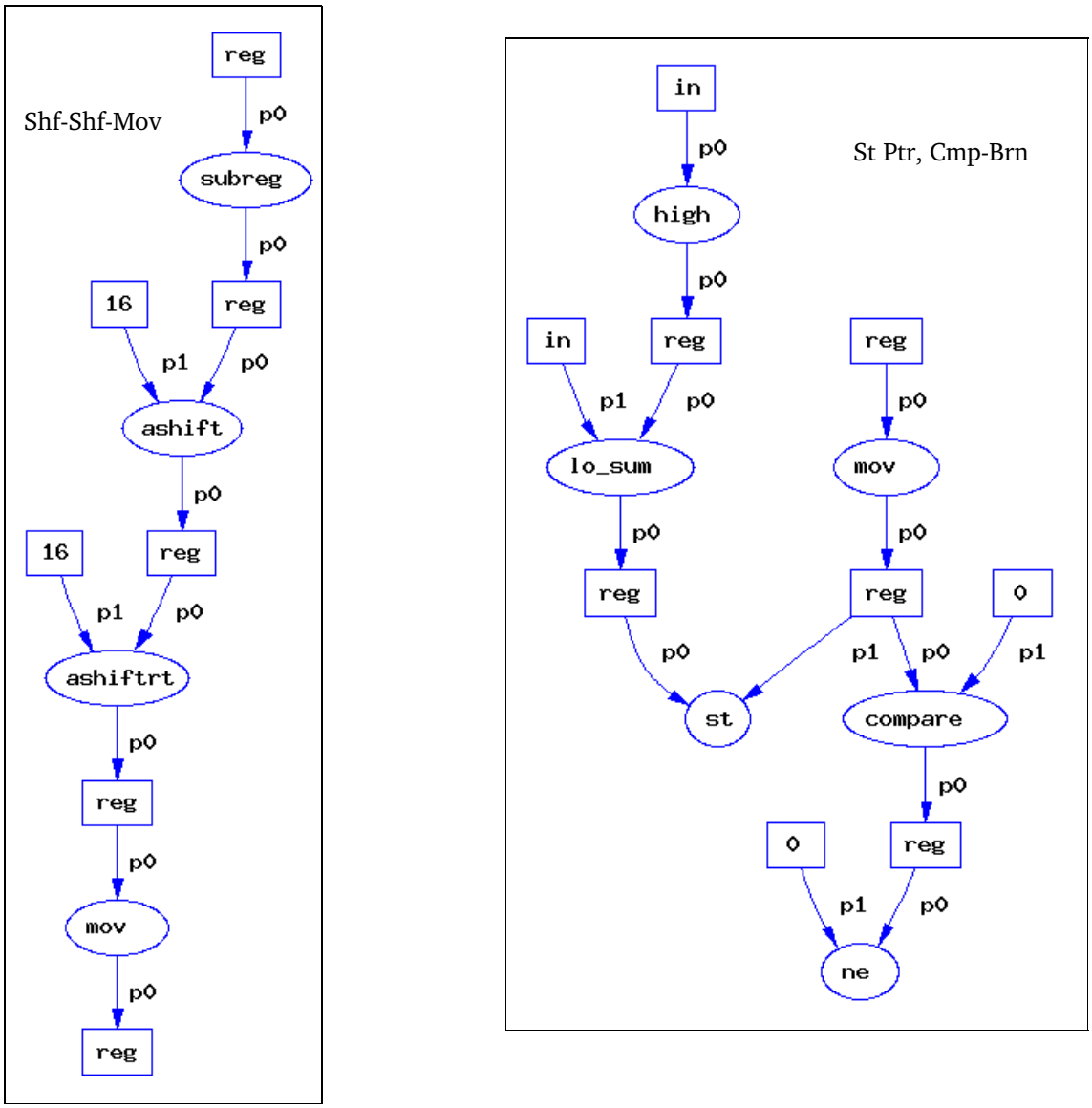
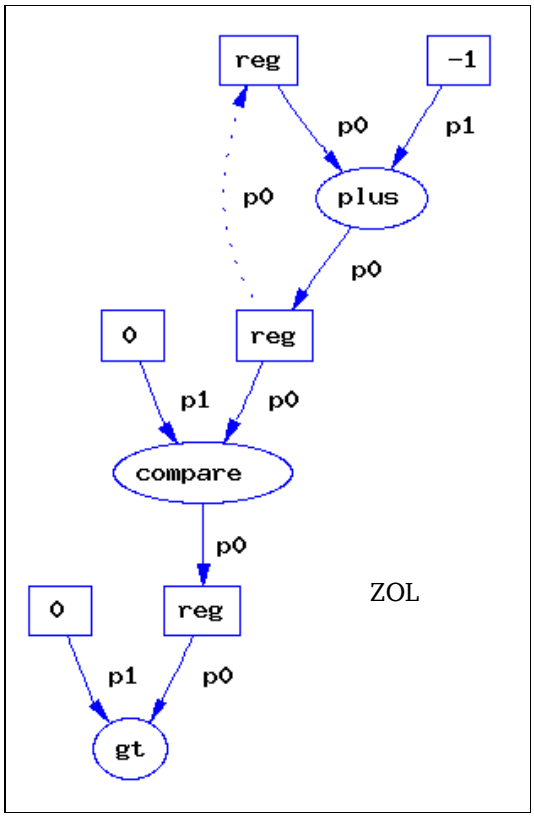
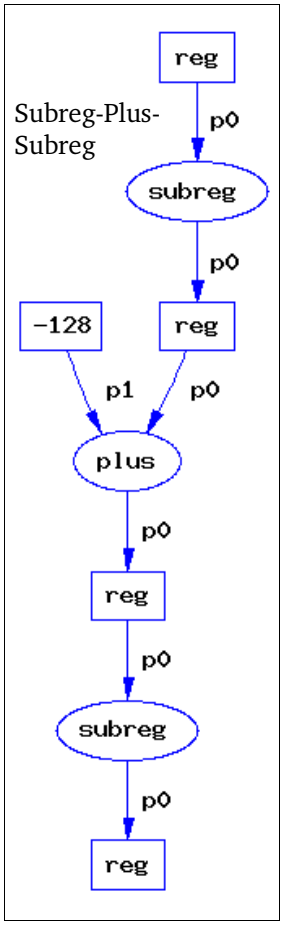


Figura 46 - Padrões representativos das categorias de grupos isomorfos.





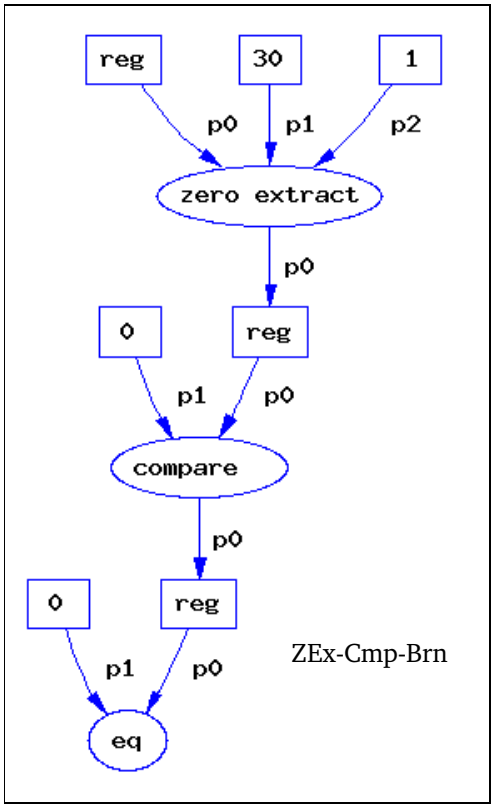


Figura 47 - Padrões representativos das categorias de grupos isomorfos.

Capítulo 5

Conclusões e Trabalhos Futuros

Nesta dissertação, foram abordadas técnicas de adaptação de arquiteturas *(hardware)* de sistemas embutidos para especializá-los para as aplicações *(software)* que executam. No Capítulo 2, foi apresentado o Projeto Chameleon do LSC, e foram citados diversos trabalhos relacionados. O Capítulo 3 apresentou a Biblioteca de Padrões, a principal contribuição deste trabalho enquanto parte do Projeto Chameleon, tendo sido descritos algoritmos e técnicas para a extração, crescimento, filtragem, comparação e seleção de padrões, além da apresentação da API e do formato de arquivo para armazenamento de padrões. Alguns experimentos foram discutidos no Capítulo 4, envolvendo resultados de isomorfismo e seleção de padrões e uma classificação de padrões freqüentes em categorias funcionais.

A contribuição da Biblioteca de Padrões foi bastante relevante ao Projeto Chameleon, ao especificar estruturas de dados e de arquivos e implementar uma biblioteca de funções para leitura, modificação e escrita de padrões. O uso de um formato para representação de padrões substancialmente independente do compilador permite a construção de ferramentas modulares ao longo de todo o fluxo de desenvolvimento, desde a extração dos padrões até a produção de código HDL que os implementa. Permite ainda a construção de um repositório de padrões, para guardar implementações bem sucedidas de padrões que possam ser reutilizadas em aplicações diferentes. Nenhum dos trabalhos relacionados estudados menciona o uso de um formato integrado para representação de padrões na trajetória do *software* ao *hardware*. A implementação da Biblioteca de Padrões e de outras ferramentas citadas no Capítulo 4, desenvolvidas ao longo do mestrado, totalizam aproximadamente 20 mil linhas de código.

O desempenho das ferramentas de *software* desenvolvidas no projeto superou nossas expectativas. Por exemplo, havia a preocupação de que o algoritmo exponencial implementado para a comparação de padrões pudesse tomar muito tempo. No entanto, os experimentos realizados com um universo de aproximadamente 50 mil padrões exigiram menos de um minuto para realizar comparações dos padrões “todos contra todos”, evidenciando que a característica exponencial do algoritmo não se manifestou. Concluímos que a grande maioria dos padrões possui diferenças significativas que são detectadas pelas heurísticas empregadas (por exemplo, comparação do número de vértices e arestas de cada tipo), evitando assim o algoritmo exponencial. Além disso, os padrões isomorfos existentes são usualmente pequenos, para os quais mesmo algoritmos exponenciais executam rapidamente. Portanto, verificamos que ainda existe capacidade computacional para considerar um número maior de padrões nos trabalhos futuros.

Os gráficos e tabelas obtidos nos experimentos permitiram testar os conceitos e trouxeram conhecimento adicional do domínio do problema. O ganho de experiência obtido com as implementações realizadas nos traz diversas idéias de melhorias e alternativas a serem exploradas em trabalhos futuros dentro do próprio Projeto Chameleon. Algumas destas idéias são discutidas a seguir.

5.1. Trabalhos Futuros

Observamos que a forma pela qual os padrões são crescidos além de blocos básicos (como subgrafos delimitados por blocos dominadores e pós-dominadores imediatos) e posteriormente comparados não permite o reconhecimento de padrões que estão contidos uns nos outros, como ilustra a Figura 48. Se fosse identificado que o padrão A está contido no padrão B, então a freqüência de execução do padrão A, no cômputo de seu peso, poderia incluir também a freqüência de execução do padrão B, e uma nova instrução poderia ser gerada para implementar apenas a computação de A (sem a operação excedente de B). A modificação do algoritmo de isomorfismo para reconhecer subgrafos no caso geral certamente levaria a um procedimento de complexidade computacional intratável (visto que a simples comparação de igualdade de dois DAGs é exponencial no pior caso). Porém, uma heurística poderia ser experimentada: de cada subgrafo, extrair estruturas *if-then* e *if-then-else* que excluam as operações executadas após o *if* (bloco pós-dominador) ou antes do *if* (bloco dominador). Neste caso, seria possível reconhecer que os três blocos básicos superiores nos padrões A e B da Figura 48 são isomorfos.

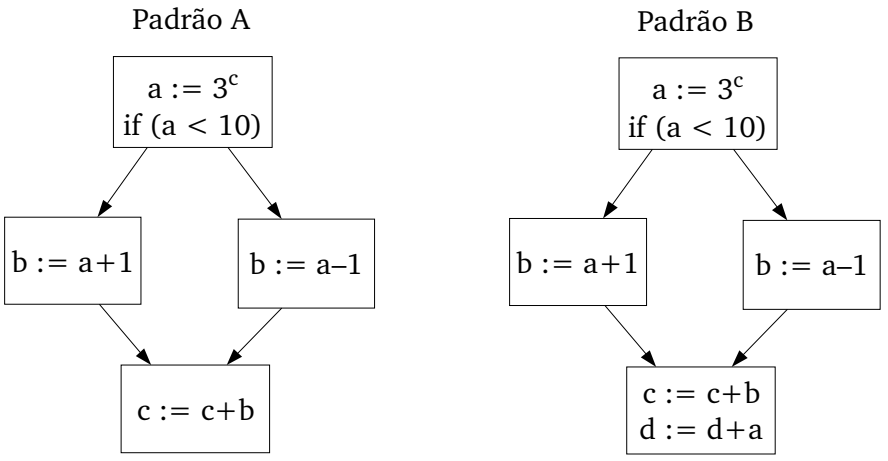


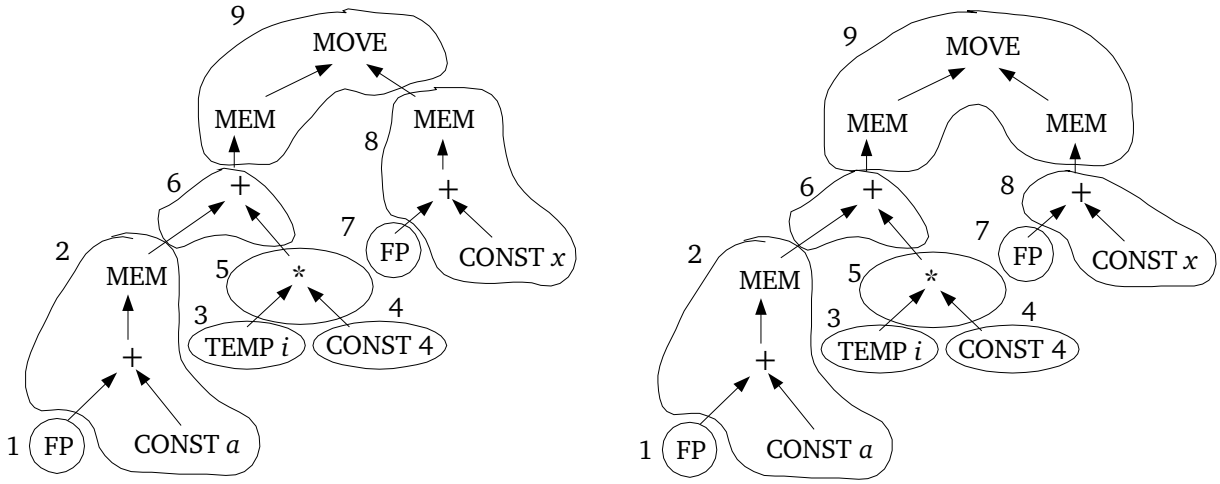
Figura 48 - Caso em que um padrão (A) está contido em outro padrão (B). O reconhecimento destes e outros casos é um tema para trabalho futuro.

Ainda em relação ao crescimento de padrões, seria interessante a implementação de opções que ofereçam mais controle ao projetista do sistema embutido que deseje analisar padrões manualmente. Por exemplo, as ferramentas deveriam permitir a poda de padrões com base na remoção de instruções indicadas pelo usuário (removendo junto as instruções dependentes), ou o crescimento com base em uma lista de blocos básicos e números de DFGs fornecida pelo usuário.

Também no processo de seleção de padrões são consideradas algumas melhorias. Uma delas consiste em aprimorar o cálculo dos pesos atribuídos aos padrões, utilizando uma técnica automatizada de escalonamento das operações para se obter uma estimativa mais precisa do número

de ciclos que sua implementação em *hardware* iria consumir. Ainda que esta técnica não permitisse prever o número de ciclos que uma implementação HDL manualmente otimizada poderia requerer, permitiria obter uma estimativa que explorasse as possibilidades de execução paralela de DFGs desconexos num subgrafo do CFG. Outra possibilidade que já está sendo experimentada no LSC é a simulação *cycle-accurate* de padrões descritos em ArchC.

Ainda na seleção de padrões, poderia ser estudada uma abordagem utilizando cobertura de peso máximo dos padrões sobre o código da aplicação, como na seleção de instruções feita no *back-end* de um compilador. No caso do compilador, existe uma representação intermediária do código, na forma de árvores ou DAGs, que deve ser *coberta (tiling)* por padrões que são diretamente mapeados a seqüências de instruções do processador alvo [ASU86, App98], como ilustra a Figura 49. As duas árvores de expressão da figura são implementadas por diferentes instruções do processador, com custos (número de ciclos) diferentes. Na árvore da esquerda, os padrões 8 e 9 correspondem às instruções *load* e *store*; na árvore da direita, correspondem às instruções *addi* e *movem*. No caso do projeto *Chameleon*, os padrões a serem casados viriam do conjunto de padrões candidatos à implementação em *hardware*. Por exemplo, para cada função da aplicação (após expansões *inline*) poderia ser gerado um grande CDFG que seria coberto por padrões menores previamente selecionados. As partes do CDFG que ficassem descobertas seriam implementadas com as instruções básicas do processador original. Quando o grafo a ser coberto é uma árvore (como no caso da Figura 49), a cobertura de peso máximo pode ser realizada com complexidade de tempo linear através de técnicas



2	LOAD	$r_1 \leftarrow M[\text{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_1 * r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\text{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$

2	LOAD	$r_1 \leftarrow M[\text{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_1 * r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \text{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$

Figura 49 - Duas possíveis coberturas de código (seleção de instruções de um compilador) com pesos diferentes. Os padrões que cobrem as árvores de expressão estão numerados e associados às instruções das tabelas de acordo com a numeração na primeira coluna. Retirado de Appel [App98].

de programação dinâmica [AGT89, Dij99, HO82]. No caso dos DFGs e CDFGs, porém, os grafos são DAGs, e o problema de cobertura de DAGs com peso máximo é NP completo [Ertl99, Pro98]. No entanto, o problema de cobertura pode ser formulado com um problema de programação linear inteira (PLI) e resolvido de forma aproximada por métodos como *branch and cut* e programação linear racional. O uso de PLI é atraente também por permitir de forma relativamente simples a inclusão de restrições de arquitetura como área de silício ou o número de *opcodes* disponíveis para novas instruções, como mostra o exemplo a seguir.

Exemplo de Modelagem da Cobertura de Padrões Através de PLI

Considere um conjunto (pequeno) de padrões candidatos à implementação em *hardware* que tenham sido escolhidos por satisfazerem uma série de restrições. Uma *ocorrência* de um padrão é definida como um trecho de código da aplicação (não necessariamente seqüencial) que pode ser coberto por um padrão. (Pela nomenclatura utilizada no Capítulo 4, distintas ocorrências de um mesmo padrão constituem um *grupo isomorfo*.) As ocorrências dos padrões devem ter sido previamente identificadas, dado que os padrões foram extraídos da própria aplicação. A cada ocorrência j de cada padrão i, associa-se uma variável binária $y_{ij} \in \{0, 1\}$ tal que:

- $y_{ij} = 1$ → a ocorrência j será coberta pelo padrão i
- $y_{ij} = 0$ → a ocorrência j não será coberta pelo padrão i

A função objetivo seria então:

$$max \sum_{i=1}^{n} \sum_{j=1}^{m_i} g_{ij} y_{ij}$$, onde:

n = número de padrões
m_i = número de ocorrências do padrão i
g_{ij} = peso da ocorrência j do padrão $i = f(c_i, d_{ij}) = c_i * d_{ij}$
c_i = nº de ciclos economizados pelo padrão i
d_{ij} = freqüência de execução da ocorrência j do padrão i

Sujeita às seguintes restrições:

- Interferência de padrões: $\sum_{y_{ij} \in T_{ij}} y_{ij} \leq 1 \quad , \quad \forall i, \forall j$ onde:
  T_{ij} = conjunto das ocorrências que interferem com a ocorrência j do padrão i (inclusive ela própria)

- *Opcodes* livres para novas instruções: $\sum_{i=1}^{n} x_i \leq P$ onde:

  P = número de *opcodes* livres para novas instruções

$$x_i = \begin{cases} 1 & \text{se o padrão i será implementado em } \textit{hardware} \\ 0 & \text{caso contrário} \end{cases} \qquad x_i = \begin{cases} 1 & \text{se } \sum_{j=1}^{m_i} y_{ij} > 0 \\ 0 & \text{caso contrário} \end{cases}$$

- Outras restrições de recursos: $\sum_{i=1}^{n} r_i x_i \leqslant R$ onde:

  r_i = recursos requeridos pelo padrão i
  R = total de recursos disponíveis (área, potência...)

Outro trabalho futuro seria a definição de um formato de arquivo para a especificação de filtros de padrões, que permitisse expressar combinações de filtros como por exemplo "os padrões que possuam 5 ou mais operadores, que tenham no máximo uma saída e cujos operadores sejam lógicos ou aritméticos *(*ex.: *shift, add)*". Uma sugestão de sintaxe para o arquivo é dada na Figura 50. Cada arquivo poderia conter especificações de vários filtros; na figura, o filtro 1 especifica a consulta citada neste parágrafo. O nome do arquivo de filtros seria então passado como parâmetro de linha de comando para as ferramentas que extraem, geram ou selecionam padrões. Por exemplo, a ferramenta *ctlgrow,* que realiza o crescimento de padrões, poderia ser executada com a linha de comando:

```
ctlgrow  [...]  -filterfile  arquivo.fil  -filternum 1
```

Neste caso, a ferramenta produziria apenas os padrões que satisfizessem o filtro 1 do arquivo "*arquivo.fil*". Nas implementações atuais, os filtros são especificados no próprio código C das ferramentas.

```
filter 1 = {
   apply { f1 and not f2 and (f3 or f4)) }
   where {
      f1 = min_operators(5);
      f2 = max_outputs(1);
      f3 = operators_class(logic);
      f4 = operators_calss(arithmetic);
   }
}
filter 2 = {
   apply { ... }
   where { ... }
}
...
```

Figura 50 - Proposta de formato de arquivo para especificação de filtros de padrões (trabalho futuro).

Ainda outra melhoria que poderia ser feita seria a migração do atual formato de representação de grafos em arquivos, que foi especificamente projetado para o Projeto Chameleon, para um formato padronizado e extensível como por exemplo XGMML [PK01, Him97] ou GraphXML [HM00], ambos formatos baseados em XML [BPS+04]. A atual implementação da Biblioteca de Padrões, embora eficiente, exige um esforço de programação em C e modificações na gramática do utilitário *bison* [Bison03] se qualquer alteração for proposta nos formatos de grafos (por exemplo, a representação explícita de arestas com atributos, que atualmente não existe). A adoção de um formato baseado em XML tornaria os arquivos mais portáveis e as alterações mais fáceis.

Apêndice A

Expressões Regulares e Gramática para o *Parser* dos Arquivos de Padrões

A.1. Arquivo de Expressões Regulares Fornecido para Ferramenta *Flex* [Flex03]

```
%{
#include "pattlib.h"
#include "parser.h"
#include "pattlib_tab.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <errno.h>

#ifndef YYLLOC_SINGLE_LINE
#define YYLLOC_SINGLE_LINE \
    ptl_yylloc.first_line = ptl_yylloc.last_line; \
    ptl_yylloc.first_column = ptl_yylloc.last_column + 1; \
    ptl_yylloc.last_column += yyleng;
#endif

YY_BUFFER_STATE prev_buffer_state = NULL;
YYLTYPE prev_ptl_yylloc;


void lexer_init(void);
static void count_lines(void);
extern char *parsing_filename;
extern char *included_filename;

%}

%option noyywrap
%option stack
/* %option prefix="ptl_yy"*/

INF  [iI][nN][fF][iI][nN][iI][tT][yY]

%x sc_string1
%x sc_string2
%x incl

%%

    static char last, *endptr, code_mode=0;
    static int s;


[ \t]+  YYLLOC_SINGLE_LINE;
\n      ptl_yylloc.last_line++; ptl_yylloc.last_column = -1;

#.*\n    ptl_yylloc.last_line++; ptl_yylloc.last_column = -1;

^%file     {
    YYLLOC_SINGLE_LINE;
    if (prev_buffer_state == NULL) yy_push_state(incl);
    else {
        print_error_header(stdout);
        printf("Lexer error: only one level of %%file is allowed\n");
    }
```

```
}

version           YYLLOC_SINGLE_LINE; return PTLVERSION;
tags              YYLLOC_SINGLE_LINE; return TAGS;
pattern           YYLLOC_SINGLE_LINE; return PATTERN;
attributes        YYLLOC_SINGLE_LINE; return ATTRIBUTES;
speed-up          YYLLOC_SINGLE_LINE; return SPEEDUP;
match             YYLLOC_SINGLE_LINE; return MATCH;
file              YYLLOC_SINGLE_LINE; return MATCH_FILE;
function          YYLLOC_SINGLE_LINE; return MATCH_FUNCTION;
frequency         YYLLOC_SINGLE_LINE; return MATCH_FREQUENCY;
match_dfg         YYLLOC_SINGLE_LINE; return MATCH_DFG;
rtx               YYLLOC_SINGLE_LINE; return RTX;
alias_id          YYLLOC_SINGLE_LINE; return ALIAS_ID;
selectors         YYLLOC_SINGLE_LINE; return SELECTORS;
operand_values    YYLLOC_SINGLE_LINE; return MATCH_OPERAND_VALUES;
bbinfo            YYLLOC_SINGLE_LINE; return BBINFO;
bb                YYLLOC_SINGLE_LINE; return BB;
pred              YYLLOC_SINGLE_LINE; return PRED;
succ              YYLLOC_SINGLE_LINE; return SUCC;
idom              YYLLOC_SINGLE_LINE; return IDOM;
ipdom             YYLLOC_SINGLE_LINE; return IPDOM;
branch            YYLLOC_SINGLE_LINE; return BRANCH;
tmp_opcode        YYLLOC_SINGLE_LINE; return TMP_OPCODE;
tmp_static_prof   YYLLOC_SINGLE_LINE; return TMP_STATIC_PROF;
tmp_dynamic_prof  YYLLOC_SINGLE_LINE; return TMP_DYNAMIC_PROF;
dfg               YYLLOC_SINGLE_LINE; return GRAPH;
use_predicate     YYLLOC_SINGLE_LINE; return USE_PREDICATE;
gen_predicate     YYLLOC_SINGLE_LINE; return GEN_PREDICATE;
true              YYLLOC_SINGLE_LINE; return TRUE;
false             YYLLOC_SINGLE_LINE; return FALSE;
def               YYLLOC_SINGLE_LINE; return DEF;
uses              YYLLOC_SINGLE_LINE; return USES;
du                YYLLOC_SINGLE_LINE; return DU;
ud                YYLLOC_SINGLE_LINE; return UD;
operand           YYLLOC_SINGLE_LINE; return OPERAND;
operands          YYLLOC_SINGLE_LINE; return OPERANDS;
operator          YYLLOC_SINGLE_LINE; return OPERATOR;
type              YYLLOC_SINGLE_LINE; return TYPE;
result            YYLLOC_SINGLE_LINE; return RESULT;
aux               YYLLOC_SINGLE_LINE; return AUX;
const_real        YYLLOC_SINGLE_LINE; return CONST_REAL;
const_int         YYLLOC_SINGLE_LINE; return CONST_INT;
const_string      YYLLOC_SINGLE_LINE; return CONST_STRING;
reg_width         YYLLOC_SINGLE_LINE; return REG_WIDTH;
code              YYLLOC_SINGLE_LINE; code_mode=1; return CODE;
vhdl              YYLLOC_SINGLE_LINE; return VHDL;
verilog           YYLLOC_SINGLE_LINE; return VERILOG;
archc             YYLLOC_SINGLE_LINE; return ARCHC;
source            YYLLOC_SINGLE_LINE; return SOURCE;
cost              YYLLOC_SINGLE_LINE; return COST;
area              YYLLOC_SINGLE_LINE; return AREA;
delay             YYLLOC_SINGLE_LINE; return DELAY;
\%\{              YYLLOC_SINGLE_LINE; s=last=0; BEGIN(sc_string2); return BEGINCODE;
\%\}              YYLLOC_SINGLE_LINE; return ENDCODE;
=                 YYLLOC_SINGLE_LINE; return '=';
\{              %{ YYLLOC_SINGLE_LINE;
           if (code_mode!=0) {
                 code_mode = 0;
                 BEGIN(sc_string1);
           }
           return '{';
        %}
\}                YYLLOC_SINGLE_LINE; return '}';
\[                YYLLOC_SINGLE_LINE; return '[';
\]                YYLLOC_SINGLE_LINE; return ']';
,                 YYLLOC_SINGLE_LINE; return ',';
;                 YYLLOC_SINGLE_LINE; return ';';


"/*"  {   /* code for eating C-like comments */
        register int c;
        YYLLOC_SINGLE_LINE;
        for ( ; ; ) {
            /* eat up text of comment */
            while ( (c = input()) != '*' && c != EOF && c != '\n' ) ptl_yylloc.last_column++;
```

```
            if ( c == '*' ) {
               ptl_yylloc.last_column++;
               while ( (c = input()) == '*' ) ptl_yylloc.last_column++;
               ptl_yylloc.last_column++;
               if ( c == '/' ) break;    /* found the end */
            }
            if ( c == '\n' ) {
               ptl_yylloc.last_line++;
               ptl_yylloc.last_column = -1;
            }
            if ( c == EOF ) {
               print_error_header(stdout);
               printf("Lexer error: EOF in comment\n");
               break;
            }
         }
      }


[0-9]+\.[0-9]+\.[0-9]+      YYLLOC_SINGLE_LINE;   ptl_yylval.bison_string   =   strdup(yytext);   return
PTLVERSION_STR;

[+\-]?[0-9]+   YYLLOC_SINGLE_LINE; ptl_yylval.bison_int = atoll(yytext); return INT;

([+\-]?[0-9]+(\.[0-9]+)?)|{INF}          |
[+\-]?[0-9]+(\.[0-9]+)?[eE][+\-]?[0-9]+  {
        YYLLOC_SINGLE_LINE;
       ptl_yylval.bison_double = strtod(yytext, &endptr);
       if (ptl_yylval.bison_double == 0.0) {
              print_error_header(stdout);
              if (errno == ERANGE)
                     printf("Lexer error: underflow converting float number \"%s\"\n", yytext);
              else if (endptr == yytext)
                     printf("Lexer error: impossible to convert float number \"%s\"\n", yytext);
       }
       else if ( ((ptl_yylval.bison_double == HUGE_VAL) || (ptl_yylval.bison_double == -HUGE_VAL))
                 && (errno == ERANGE)) {
              print_error_header(stdout);
              printf("Lexer error: overflow converting float number \"%s\"\n", yytext);
       }
       return FLOAT;
}

\"[^\n\"]*\"  {
      YYLLOC_SINGLE_LINE;
      ptl_yylval.bison_string = yytext+1;
      ptl_yylval.bison_string[strlen(ptl_yylval.bison_string)-1]=0;
      ptl_yylval.bison_string = strdup(ptl_yylval.bison_string);
      return ID2;
}

[[:alpha:]_]+[[:alnum:]_./:\-]*   YYLLOC_SINGLE_LINE; ptl_yylval.bison_string = strdup(yytext); return
ID;

.       {
       YYLLOC_SINGLE_LINE;
       print_error_header(stdout);
       printf("Lexer error: unexpected char \'%c\' in INITIAL state\n",(int)(*yytext));
}


<sc_string1>[ \t\n]+\}  count_lines(); BEGIN(INITIAL); return '}';
<sc_string1>[^}]+  { ptl_yylval.bison_string = strdup(yytext);
                     count_lines(); BEGIN(INITIAL); return STRING; }
<sc_string1>\}     YYLLOC_SINGLE_LINE; BEGIN(INITIAL); return '}';

<sc_string2>.|\n {
    endptr = yytext + s++;
    if ((*endptr == '}') && (last == '%')) {
        *--endptr = 0;
        ptl_yylval.bison_string = strdup(yytext);
        unput('}');
        unput('%');
       count_lines();
        BEGIN(INITIAL);
        return STRING;
    }
```

```
    else {
        last = *endptr;
        yymore();
    }
}

<incl>[ \t]*   YYLLOC_SINGLE_LINE;  /* eats the whitespace */
<incl>\".+\"   { /* gets the include file name */

       char *cp;
       FILE *ff;

       YYLLOC_SINGLE_LINE;

       cp = yytext + yyleng - 1; /* position of closing '"' */
       *cp = '\0';               /* clear it */
       cp = yytext + 1;          /* skip opening '"' */

       ff = fopen( cp, "r" );

       if ( ! ff ) {
          print_error_header(stdout);
          printf("Lexer error: unable to include file \"%s\", skipping it\n",cp);
       }
       else {
          included_filename = strdup(cp);
          prev_buffer_state = YY_CURRENT_BUFFER;
          prev_ptl_yylloc = ptl_yylloc;
          lexer_init();
          yyin = ff;
          yy_switch_to_buffer( yy_create_buffer( yyin, YY_BUF_SIZE ) );
       }
       yy_pop_state();
}
<incl>.|\n    unput(*yytext); yy_pop_state();

<<EOF>> {
       if (prev_buffer_state == NULL) {
          yyterminate();
       }
       else {
          free(included_filename);
          included_filename=NULL;
          yy_delete_buffer( YY_CURRENT_BUFFER );
          yy_switch_to_buffer( prev_buffer_state );
          prev_buffer_state = NULL;
          ptl_yylloc = prev_ptl_yylloc;
       }
}


%%

void my_yyrestart(FILE *f) {

    yyrestart(f);
    BEGIN(INITIAL);
}

static void count_lines(void) {

    int i, last_newline;

    ptl_yylloc.first_line = ptl_yylloc.last_line;
    ptl_yylloc.first_column = ptl_yylloc.last_column + 1;

    for (i=0, last_newline=-1; i < yyleng; i++) {
      if (yytext[i] == '\n') {
        ++ptl_yylloc.last_line; last_newline = i;
      }
    }

    if (last_newline>=0) ptl_yylloc.last_column = yyleng - last_newline - 2;
    else ptl_yylloc.last_column += yyleng;
}
```

A.2. Gramática do Parser no Formato da Ferramenta *Bison* [Bison03]

Observação: na gramática abaixo, foi removido todo o código fonte em C referente às ações associadas às reduções de regras, visto que o propósito é apenas documentar a sintaxe.

```
%{
%}

%locations

%union {
  char                    *bison_string;
  char                     bison_char;
  double                   bison_double;
  long long                bison_int;
};


%token <bison_string> PTLVERSION_STR
%token <bison_string> ID
%token <bison_string> ID2
%token <bison_string> STRING
%token <bison_int>    INT
%token <bison_double> FLOAT
%token <bison_char>   UNEXPECTED_CHAR

%token TAGS
%token PTLVERSION
%token PATTERN
%token ATTRIBUTES
%token MATCH
%token MATCH_FILE
%token MATCH_FUNCTION
%token MATCH_DFG
%token MATCH_FREQUENCY
%token MATCH_OPERAND_VALUES
%token BBINFO
%token BB
%token PRED
%token SUCC
%token IDOM
%token IPDOM
%token BRANCH
%token SPEEDUP
%token TMP_OPCODE
%token TMP_STATIC_PROF
%token TMP_DYNAMIC_PROF
%token CODE
%token SOURCE

%token GRAPH
%token USE_PREDICATE
%token GEN_PREDICATE
%token TRUE
%token FALSE
%token OPERAND
%token DEF
%token USES
%token DU
%token UD
%token OPERATOR
%token RTX
%token ALIAS_ID
%token SELECTORS
%token OPERANDS
%token CONST_REAL
%token CONST_INT
%token CONST_STRING
%token REG_WIDTH
%token TYPE
%token RESULT
%token AUX
```

```
%token VHDL
%token VERILOG
%token ARCHC
%token AREA
%token DELAY
%token COST
%token BEGINCODE  /* %{ */
%token ENDCODE    /* %} */

%{
%}

%%

pattfile: /* empty */ {}
| pattfile PTLVERSION '=' PTLVERSION_STR ';' {}
| pattfile TAGS '=' idlist ';' {}
| pattfile pattern {}
;

idlist: ID      {}
| idlist ',' ID {}
;

pattern: PATTERN ID '=' '{' pattern_body '}' {}
;

pattern_body:  /*empty*/ {}
| pattern_body pb_component {}
;

pb_component: attribs {}
| code    {}
| graph   {}
| vhdl    {}
| verilog {}
| archc   {}
| TAGS '=' idlist ';' {};

attribs: ATTRIBUTES '=' '{' attr_body '}' {};

attr_body: /* empty */ { }
| attr_body attr_stmt ';' { }
| attr_body MATCH '=' '{' match_body '}' {}
| attr_body error ';' {};

attr_stmt:  SPEEDUP '=' FLOAT  {}
        | SPEEDUP '=' INT {}
        | TMP_STATIC_PROF '=' FLOAT {}
        | TMP_STATIC_PROF '=' INT {}
        | TMP_DYNAMIC_PROF '=' FLOAT {}
        | TMP_DYNAMIC_PROF '=' INT {}
        | TMP_OPCODE '=' ID2 {};

code: CODE '=' '{' STRING '}' {};

graph:  GRAPH INT '=' '{' '}' {}
| GRAPH INT '=' '{' graph_body '}' {};

graph_body: graph_section {}
| graph_body graph_section {};

graph_section: AUX '=' BEGINCODE STRING ENDCODE {}
| OPERAND INT '=' '{' operand_body '}'  {}
| OPERATOR INT '=' '{' operator_body '}'  {}
| USE_PREDICATE '=' pred_list ';' {}
| GEN_PREDICATE '=' pred_list ';' {};

pred_list: pred_item {}
| pred_list pred_item {};

pred_item: '[' ID INT INT ']' {}
| '[' ID INT INT TRUE ']' {}
| '[' ID INT INT FALSE ']' {};

operand_body: {}
| operand_body AUX '=' BEGINCODE STRING ENDCODE {}
```

```
| operand_body operand_stmt ';' {}
| operand_body error ';' {};

operand_stmt:  TYPE '=' INT  {}
| CONST_INT '=' INT      {}
| CONST_REAL '=' FLOAT   {}
| CONST_REAL '=' INT     {}
| CONST_STRING '=' ID2   {}
| REG_WIDTH '=' INT      {}
| DEF '[' INT ']' DU '=' reaching_list {}
| DEF '[' INT ']' {}
| USES '[' intlist ']' UD '=' reaching_list {}
| USES '[' intlist ']' {};

reaching_list: reach_item {}
| reaching_list reach_item {};

reach_item: '[' ID INT INT INT INT ']' {};

operator_body: {}
| operator_body AUX '=' BEGINCODE STRING ENDCODE {}
| operator_body operator_stmt ';' {}
| operator_body error ';' {};

operator_stmt: OPERANDS '=' intlist {}
| RESULT '=' INT {}
| TYPE '=' INT {}
| RTX '=' INT INT {}
| ALIAS_ID '=' INT {}
| SELECTORS '=' INT {};

match_body: /* empty */ { }
| match_body match_stmt ';' { }
| match_body MATCH_DFG INT '=' '{' match_dfg '}' {}
| match_body error ';' {};

match_stmt: MATCH_FILE '=' ID2 {}
| MATCH_FUNCTION '=' ID2 {};

match_dfg: /* empty */ {}
| match_dfg match_dfg_stmt ';' { }
| match_dfg BBINFO '=' '{' bbinfo '}' {}
| match_dfg error ';' {};

match_dfg_stmt: MATCH_FREQUENCY '=' INT {}
| RTX '=' rtx_list {}
| MATCH_OPERAND_VALUES '=' intlist {};

rtx_list: rtx_item {}
| rtx_list rtx_item {};

rtx_item: '[' INT INT ']' {};

bbinfo: /* empty */ {}
| bbinfo bb_stmt ';' {}
| bbinfo error ';' {};

bb_stmt: BB '=' INT {}
| SUCC '=' intlist {}
| PRED '=' intlist {}
| IDOM '=' INT {}
| IPDOM '=' INT {}
| BRANCH '=' ID INT INT INT INT {};

intlist: INT {}
| intlist ',' INT {};

vhdl: VHDL ID '=' '{' hdl_body '}' {};

verilog: VERILOG ID '=' '{' hdl_body '}' {};

archc: ARCHC ID '=' '{' hdl_body '}' {};

hdl_body: /*empty*/ {}
| hdl_body hb_component {};

hb_component: cost {}
```

```
| SOURCE '=' BEGINCODE STRING ENDCODE {};

cost: COST '=' '{' cost_body '}' {};

cost_body: /* empty */ {}
| cost_body cb_stmt ';' {}
| cost_body error ';' {};

cb_stmt: /* empty */ {}
| AREA '=' FLOAT  {}
| AREA '=' INT  {}
| DELAY '=' FLOAT {}
| DELAY '=' INT {};

%%
```

Referências Bibliográficas

[μClinux] Embedded Linux/Microcontroller Project. http://www.uclinux.org

[AC01] Marnix Arnold and Henk Corporaal. *Designing Domain-Specific Processors.* Proceedings of the 9th International Symposium on Hardware/Software Codesign, pp. 61-66, 2001.

[AGT89] Alfred V. Aho, Mahadevan Ganapathi and Steven W. K. Tjiang. *Code Generation Using Tree Matching and Dynamic Programming.* ACM Transactions on Programming Languages and Systems, 11(4):491-516, 10/1989.

[AM98] Annapolis Microsystems, Inc. *Wildfire Reference Manual*. Annapolis, MD, USA, 1998.

[API03] K. Atasu, L. Pozzi, and P. Ienne. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. In Proceedings of the 40th Design Automation Conference (DAC), pages 256-261, 2003.

[App98] Appel, Andrew W. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[Aristotle] Aristotle Research Group, http://www.cc.gatech.edu/aristotle/

[AS93] P. Athanas and H. Silverman. *Processor Reconfiguration through Instruction-Set Metamorphosis*. IEEE Computer, vol. 26, pp. 11-18, March 1993.

[ASU86] Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Pub Co, 1/1986.

[BA97] Ray Bittner and Peter Athanas. *Wormhole Run-Time Reconfiguration.* FPGA'97, ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 1997.

[BBPP99] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos and Marcello Pelillo. *Handbook of Combinatorial Optimization*, 1999.

[BH90] Ravi Boppana, Magnús M. Halldórsson. *Approximating Maximum Independent Sets by Excluding Subgraphs.* SWAT 90 2nd Scandinavian Workshop on Algorithm Theory, 1990.

[binutils03] Uma coleção de ferramentas de código fonte aberto para manipulação de arquivos executáveis binários, incluindo *assembler* e *profiler*, entre outras. http://sources.redhat.com/binutils/ , http://ftp.gnu.org/gnu/binutils/

[Bison03] *GNU Parser Generator*. GNU, http://www.gnu.org/software/bison/, 2003.

[BPS+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler and François Yergeau. *Extensible Markup Language (XML) 1.0 (Third Edition)*. W3C Recommendation, 04 February 2004. http://www.w3.org

[BSL+01] Banakar, Rajeshwari, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan and Peter Marwedel. *Comparsion of Cache- and Scratch-Pad based Memory Systems with respect to Performance, Area and Energy Consumption.* Technical Report, 762, University of Dortmund, Dept. of CS XII, 2001.

[CFHZ04] Jason Cong, Yiping Fan, Guoling Han, and Zhiru Zhang. *Application-Specific Instruction Generation for Configurable Processor Architectures.* Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, pp. 183-189.

[CH02] Katherine Compton and Scott Hauck. *Reconfigurable Computing: A Survey of Systems and Software*. ACM Computing Surveys, Vol. 34, No. 2, June 2002, pp. 171-210.

[CKY+99] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang, and C.-M. Kyung. *Synthesis of Application Specific Instructions for Embedded DSP Software*. IEEE Transactions on Computers, 48(6):603-614, 1999.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*, second edition. The MIT Press, 2001.

[CS00] Chameleon Systems, Inc. 2000. *CS2000 Advance Product Specification*. San Jose, CA, USA, 1998. (Nota: não há qualquer relação entre esta empresa e o "Projeto Chameleon" desenvolvido no IC/UNICAMP.)

[CZTM03] N. Clark, H. Zhong, W. Tang, and S. Mahlke. *Automatic Design of Application Specific Instruction Set Extensions through Dataflow Graph Exploration*. International Journal of Parallel Programming, 31(6):429-449, 2003.

[DDL77] Deo, Narsingh, J. M. Davis and R. E. Lord. *A new algorithm for digraph isomorphism.* BIT, 17, 16-30, 1977.

[Dij99] E. J. Dijkstra. *An Efficient Implementation of Tree Matching*. 1999. http://citeseer.ist.psu.edu/dijkstra99efficient.html

[Ertl99] M. Anton Ertl. *Optimal Code Selection in DAGs*. ACM POPL 99, San Antonio, Texas, USA, 1999.

[Flex03] *GNU Fast Lexical Analyser Generator*. GNU, http://www.gnu.org/software/flex/, 2003.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. *The Program Dependence Graph and Its Use in Optimization*. ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3, July 1987, Pages 319-349.

[Gaisler04] *Gaisler Research*. 2004. Empresa suíça responsável pela criação e manutenção do código VHDL aberto do processador Leon. http://www.gaisler.com

[GCC04] *GNU Compiler Collection.* 2004. http://gcc.gnu.org

[GDWL93] Daniel Gajski, Nikil Dutt, Allen Wu and Steve Lin. *High-Level Synthesis – Introduction to Chip and System Design. Chapter 5: Design Representation and Transformations.* Kluwer Academic Publishers, 1993.

[GEK+04] Emden Gansner, John Ellson, Eleftherios Koutsofios et al. *Graphviz – Open Source Graph Drawing Software*. AT&T Labs Research, 2004. http://www.research.att.com/sw/tools/graphviz/

[Gnuplot04] Ferramenta de desenho de gráficos de funções com código aberto e gratuito, para vários sistemas operacionais. http://www.gnuplot.info

[GPL04] *GNU General Public License*. 2004. http://www.gnu.org

[GRE+01] 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.* IEEE 4th Annual Workshop on Workload Characterization, Austin, TX, 12/2001.

[HB01] J. P. Hammes and A. P. W. Böhm. *The SA-C language.* Colorado State University, 2001. http://www.cs.colostate.edu/cameron

[HD95] Ing-Jer Huang and Alvin M. Despain. *Synthesis of Application Specific Instruction Sets.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 14(6):663-675, 1995.

[Hee04] Dimitri van Heesch. Doxygen – ferramenta para documentação de código fonte sob licença GPL. http://www.doxygen.org, 2004.

[HFHK97] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao. *The Chimaera Reconfigurable Functional Unit*. IEEE Symposium on FPGAs for Custom Computing Machines, 1997.

[Him97] Michael Himsolt. *GML: A portable Graph File Format*. Technical Report, Universität Passau, Passau, Germany, 1997. http://infosun.fmi.uni-passau.de/Graphlet/GML/

[HL98] Huang, Jian and David J. Lilja. *Exploiting Basic Block Value Locality with Block Reuse.* Technical Report, HPPC-98-09, University of Minnesota, Depts. of Electrical Engineering and Computer Science, Minneapolis, Minnesota, USA, 8/1998.

[HM00] I. Herman and M. S. Marshall. *GraphXML – An XML Based Graph Interchange Format.* Technical Report INS-R0009, CWI, 2000. http://www.cwi.nl/InfoVisu/GraphXML.

[HO82] Christoph W. Hoffman and Michael J. O'Donnell. *Pattern Matching in Trees.* Journal of the ACM, 29(1):68-95, 1/1982.

[Hol90] Bruce Holmer et al. *Fast Prolog with an Extended General Purpose Architecture.* Proc. of 27th International Symposium on Computer Architecture, pp. 282-291, 1990.

[HRB+99] J. Hammes, R. Rinker, W. Böhm, W. Najjar, B. Draper, and R. Beveridge. *Cameron: High Level Language Compilation for Reconfigurable Systems*. In Conference on Parallel Architectures and Compilation Techniques, Newport Beach, CA, 1999.

[HW97] J. R. Hauser, and J. Wawrzynek. *Garp: A MIPS Processor with a Reconfigurable Coprocessor*. IEEE Symposium on Field-Programmable Custom Computing Machines, 12 21, 1997.

[KKMB02] R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh. *Instruction Generation for Hybrid Reconfigurable Systems.* ACM Transactions on Design Automation of Electronic Systems (TODAES), 7(4):605-627, 2002.

[LCD03] Jong-eun Lee, Kiyoung Choi, Nikil D. Dutt. *An Algorithm For Mapping Loops Onto Coarse-Grained Reconfigurable Architectures*. Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems, pages 183-188.

[LLS+00] Guangming Lu, Ming-hau Lee, Hartej Singh, Nader Bagherzadeh, Fadi J. Kurdahi, Eliseu M. Filho. *MorphoSys: a Reconfigurable Processor Targeted to High Performance Image Application*. IEEE Transactions on Computers 49, 5, 465-481, 2000.

[LPM98] Chunho Lee, Miodrag Potkonjak and William H. Magione-Smith. *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems*. Micro-30, 1998.

[LPS97] Chunho Lee, Miodrag Potkonjak and William H. Mangione-Smith. *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems.* Micro-30, 30th Annual International Symposium on Microarchitecture, 1997.

[LTS99] R. Laufer, R. R. Taylor, and H. Schmit. *PCI-PipeRench and the SwordAPI: A System for Stream-Based Reconfigurable Computing*. IEEE Symposium on Field-Programmable Custom Computing Machines, 200-208, 1999.

[MAHM02] N. B. Moreano, G. Araujo, Z. Huang and S. Malik. *Datapath Merging and Interconnection Sharing for Reconfigurable Architectures*. Proc. of the 15th. ACM/IEEE International Symposium on System Synthesis, 38-43, October 2002.

[Mar03] Peter Marwedel. *Embedded System Design.* Kluwer Academic Publishers. Novembro de 2003, ISBN 1-4020-7690-8. *Slides* disponíveis na Internet em: http://ls12-www.cs.uni-dortmund.de/~marwedel/kluwer-es-book/

[Mentor04] Mentor Graphics Inc. Electronic Design Automation Technologies. 2004. http://www.mentor.com

[MO98] T. Miyamori, and K. Olukotun. *A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications*. IEEE Symposium on Field-Programmable Custom Computing Machines, 2-11, 1998.

[MSH01] Gokhan Memik, B. Mangione-Smith and W. Hu. *NetBench: A Benchmarking Suite for Network Processors*. Compiler and Architecture Research Group (CARES) at University of California, Los Angeles, Technical Report No. 2001_2_01.

[Nauty04] Nauty Automorphism Package. 2004. http://cs.anu.edu.au/people/bdm/nauty

[NewLib04] Biblioteca C de código aberto para sistemas embutidos. http://sources.redhat.com/newlib/

[Nios04] *Nios Processor*. Altera Inc., 2004. http://www.altera.com

[PGLM94] J. Van Praet, G. Goossens, D. Lanneer and H. De Man. *Instruction Set Definition and Instruction Selection for ASIPs*. In Proceedings of the 7th International Symposium on High-Level Synthesis, pages 11-16, 1994.

[PK01] John Punin, Mukkai Krishnamoorthy. *XGMML (eXtensible Graph Markup and Modeling Language) 1.0 Draft Specification*. 2001. http://www.cs.rpi.edu/~puninj/XGMML/

[Pro98] Todd A. Proebsting. *Least-Cost Instruction Selection in DAGs is NP-complete.* http://research.microsoft.com/~toddpro/papers/proof.htm, 1998.

[RAA+03] Sandro Rigo, Rodolfo J. Azevedo, and Guido Araújo. *The ArchC Architecture Description Language.* IC- Technical Report 15 - June 2003. http://www.archc.org

[RLG+98] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. ARNOLD, and M. Gokhale. *The NAPA Adaptive Processing Architecture*. IEEE Symposium on Field-Programmable Custom Computing Machines, 28-37, 1998.

[Roy90] Peter L. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* Ph.D. thesis, Computer Science Department, Technical Report UCB/CSD 90/600, Univ. of California, Berkeley, 1990.

[RS94] Rahul Razdan and Michael D. Smith. *A High-Performance Microarchitecture with Hardware-Programmable Functional Units*. Proceedings of the 27th Annual International Symposium on Microarchitecture, pages 172-180, 1994.

[SH00] M. D. Smith and G. Holloway. *An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization*. Harvard University, Cambridge, Mass., 2000. http://www.eecs.harvard.edu/hube/research/machsuif.html

[SPARC04] *The SPARC Architecture Manual, Version 8*. SPARC International, Inc. 2004. http://www.sparc.org/standards.html

[SPEC92] *Standard Performance Evaluation Corporation (SPEC) Newsletter*, Volume 4, Issue 1, Mar. 1992.

[SRRJ02] Fei Sun, Srivaths Ravi, Anand Raghunathan and Niraj K. Jha. *Synthesis of Custom Processors based on Extensible Platforms.* Proceedings of the International Conference on Computer-Aided Design (ICCAD), pages 641-648, 2002.

[SUIF04] *Stanford SUIF Compiler*, 2004. http://suif.stanford.edu

[SWT+02] Herman Schmit, David Whelihan, Andrew Tsai, Matthew Moe, Benjamin Levine and R. Reed Taylor. *PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology.* IEEE Custom Integrated Circuit Conference (CICC), 2002.

[SZW+01] Stefan Steinke, Christoph Zobiegala, Lars Wehmeyer and Peter Marwedel. *Moving Program Objects to Scratch-Pad Memory for Energy Reduction.* Technical Report, 756, University of Dortmund, Dept. of CS XII, 2001.

[Trimaran04] *Trimaran, An Infrastructure for Research in ILP.* 2004. http://www.trimaran.org

[UWW+99] Gang-Ryung Uh, Yuhong Wang, David Whalley, Sanjay Jinturkar, Chris Bums, and Vincent Cao. *Effective Exploitation of a Zero Overhead Loop Buffer*. Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers and tools for embedded systems.

[VBR+04] Pablo Viana, Edna Barros, Sandro Rigo, Rodolfo J. Azevedo and Guido Araújo. *Modeling and Simulating Memory Hierarchies in a Platform-based Design Methodology.* Accepted for publication at Design, Automation and Test in Europe (DATE'04) - Paris - France, February 2004.

[VBR+96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. *Programmable Active Memories: Reconfigurable Systems Come of Age*. IEEE Trans. VLSI Syst. 4, 1, 56-69, 1996.

[VNK+03] Girish Venkataramani, Walid Najjar, Fadi Kurdahi, Nader Bagherzadeh, Wim Bohm, Jeff Hammes. *Automatic Compilation to a Coarse-Grained Reconfigurable System-on-Chip.* ACM Transactions on Embedded Computing Systems, Vol. 2, No. 4, November 2003, Pages 560-589.

[WC96] R. D. Wittig and P. Chow. *OneChip: An FPGA Processor with Reconfigurable Logic*. IEEE Symposium on FPGAs for Custom Computing, 1996.

[Xess04] Xess Corporation. 2004. http://www.xess.com

[Xilinx04] Xilinx Inc. Programmable Logic Devices, FPGA & CPLD. 2004. http://www.xilinx.com

[Tensilica04] Tensilica Inc. 2004. http://www.tensilica.com

[ZVSM94] V. Zivojnovic, J. M. Velarde, C. Schläger and H. Meyr. *DSPstone: A DSP-Oriented Benchmarking Methodology.* Proc. of ICSPAT'94 - Dallas, Oct. 1994.