Ferramentas para Seleção de Padrões de Instruções para Arquiteturas Reconfiguráveis

Rogério de Rangel Moreira

Dissertação de Mestrado

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

M813f
L813f

Moreira, Rogério de Rangel

Ferramentas para seleção de padrões de instruções para arquiteturas reconfiguráveis / Rogério de Rangel Moreira -- Campinas, [S.P. :s.n.], 2004.

Orientador : Rodolfo Jardim de Azevedo.

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

1. Sistemas embutidos de computador. 2. Sistemas especialistas (Ciência da computação). 3. Hardware. I. Azevedo, Rodolfo Jardim de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Ferramentas para Seleção de Padrões de Instruções para Arquiteturas Reconfiguráveis

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rogério de Rangel Moreira e aprovada pela Banca Examinadora.

Campinas, 12 de outubro de 2005.

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.

Instituto de Computação
Universidade Estadual de Campinas

Ferramentas para Seleção de Padrões de Instruções para Arquiteturas Reconfiguráveis

Rogério de Rangel Moreira

12 de outubro de 2005

Banca Examinadora:

- Rodolfo Jardim de Azevedo (Orientador)
- Ricardo Pannain
  Instituto de Computação - UNICAMP
- Ricardo Luís de Freitas
  PUC - Campinas
- Guido Araújo (suplente)
  Instituto de Computação - UNICAMP

Resumo

Devido ao aumento da quantidade de sistemas embarcados no dia a dia das pessoas, faz-se necessário que tais sistemas tenham desempenho ótimo aliado a um baixo custo de produção. O Projeto *Chameleon* é um projeto direcionado para a área de arquiteturas embarcadas reconfiguráveis, voltado para a determinação de padrões de programas que devem ser implementados em *hardware* de forma a otimizar o desempenho de tais sistemas embarcados.

Este trabalho apresenta a biblioteca para seleção de padrões de programas - *Pattern Matcher*, que corresponde a um dos módulos do Projeto *Chameleon*. A sua principal característica é a habilidade de filtrar padrões de programas originados de uma massa de dados de grande proporções, onde fica praticamente impossível uma análise manual de quais padrões devem ser selecionados para implementação no *hardware*.

As principais contribuições deste projeto são: um conjunto de filtros que podem ser utilizados de forma individual ou conjugada de modo a determinar os padrões que atendem às necessidades dos projetistas, além de um conjunto de funções estatísticas que permitem analisar um conjunto de padrões de programas, juntamente com uma arquitetura de *software* modular capaz de suportar novas extensões de filtros e análises estatísticas.

Abstract

Due to the ever increasing usage of embedded systems in our day lives, these systems are required to accomplish an optimal performance along with a low cost os production. The *Chameleon* Project focuses on embedded reconfigurable architectures, where it struggles to pin point which code patterns are electable to be implemented in hardware in order to optimize the overall system performance.

This theses presents the Pattern Matcher, a library that fits the *Chameleon* Project architecture. Its mains capabilities comprises the automatic code pattern filtering. Since these patterns belong to a huge data base, it would be virtually impossible to manually select the code patterns that may fit a developer's need among thousands of them.

The main contributions of this theses are: a set of filters that can be used individually or chained in order to pick those code patterns that fit someone needs, along with a set of statistical functionalities that allow one to have an overview of an entire library of code patterns and an extensible software architecture that can be extended to support new filters and statistical functionalities.

Agradecimentos

A Deus, por tudo em minha vida. Para minha esposa Carol, pelas noites perdidas, pela compreensão e por acreditar que esse trabalho seria possível. Aos meus pais, pela crença na educação como base para o futuro das pessoas e pelo apoio dado a mim durante toda a vida, apesar da distância que nos separa fisicamente. Ao pessoal do LSC, principalmente ao Paulo Eduardo, Edson Borin e Felipe Klein, com as intermináveis discussões sobre algoritmos, pela paciência e tempo dedicado na validação da ferramenta.

À minha família.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

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 [14]. Com isso pode-se observar que, cada vez mais, tais sistemas fazem parte de nosso dia a dia, de forma a não percebermos mais a existência deles, tão comuns que eles se tornaram.

Esses sistemas embarcados (ou dedicados) normalmente executam sempre uma mesma e determinada tarefa, sofrendo atualização em seu *software* raramente, chegando a executar o mesmo conjunto de rotinas por anos a fio. Considerando essa popularidade, é necessário que esses sistemas embarcados desempenhem suas funções com a melhor performance e menor custo possível.

O projeto *Chameleon* se propõe a aumentar a performance de execução de programas aplicativos, por meio da substituição de trechos de programas por instruções implementadas em *hardware* na arquitetura RISC.

Com base nas necessidades do projeto, surgiu a oportunidade do desenvolvimento da biblioteca para seleções de padrões - *Pattern Matcher* a qual objetiva tornar viável aos usuários do sistema a seleção de padrões os quais serão posteriormente portados para o *hardware*.

1.1 Contribuição

Com o aumento a cada dia dos programas aplicativos - tanto em tamanho quanto em complexidade, têm se tornado cada vez mais difícil uma análise manual dos programas na tentativa de determinar que trechos de programas devem ser portados ao *hardware*.

Esse trabalho se propõe ao desenvolvimento da biblioteca para seleção de padrões de código de forma a se integrar ao projeto *Chameleon* possibilitando seu uso tanto de maneira programática e embutida no sistema na forma de chamadas de função a sua API (Application Programming Interface), quanto de forma iterativa e operada pelo usuário.

Para isso, serão desenvolvidos filtros que correspondem aos critérios que os padrões[1] devem atender para que os requisitos dos usuários sejam satisfeitos. Esses filtros podem ser aplicados um a um ou agrupados de forma a montar expressões lógicas, obtendo-se assim critérios mais elaborados de filtragem.

Com base nesses filtros de restrição, será possível estender a biblioteca de forma a adicionar funcionalidades de coleta de dados estatísticos. Onde, ao invés de se usar tais filtros para eliminação de padrões, esses são usados para contabilizar dados pertinentes gerando uma base de dados com as características dos mesmos. Essas funcionalidades estatísticas permitem ao projetista não eliminar padrões, mas sim facilitar seu trabalho de forma a ter uma visão consolidada de seus padrões possibilitando um melhor direcionamento de seus estudos e posterior criação de filtros mais otimizados.

Em resumo, as contribuições desta dissertação são:

- Um conjunto de filtros que podem ser utilizados para eliminar aqueles padrões de código que não são interessantes de serem implementados em *hardware*;
- Um conjunto de funções estatísticas de padrões para auxiliar o projetista na tomada de decisões;
- Um *framework* de *software* onde é possível a adição de novos filtros ou funções estatísticas de forma modular e que não afeta em outras partes do sistema;
- Uma análise dos *benchmarks MediaBench* e *MiBench* tomando como base as funções estatísticas serão apresentadas.

1.2 Organização

Este trabalho está organizado da seguinte forma: no Capítulo 2, os trabalhos relacionados são mostrados; em seguida no Capítulo 3, a biblioteca de seleção de padrões é apresentada levando em conta suas principais características e métodos de utilização. No Capítulo 4, é feita uma a apresentação dos *benchmarks MediaBench e MiBench* juntamente com uma aplicação prática da biblioteca junto a esses *benchmarks*, determinando algumas características relevantes dos mesmos. Finalmente, o Capítulo 5 contém a conclusão desse trabalho além de possíveis trabalhos futuros com o intuito de adicionar funcionalidades a biblioteca.

Capítulo 2

Trabalhos Relacionados, Visão Geral e Caracterização do Problema

Esse trabalho descreve o desenvolvimento de uma das diversas partes do projeto *Chameleon*, o qual tem por objetivo disponibilizar ferramentas para a especialização de processadores dedicados, onde trechos de um programa aplicativo são transformados em novas instruções implementadas em *hardware*.

2.1 O conceito de *Pattern*

Antes de descrever as etapas do processo de geração de instruções em *hardware* no projeto *Chameleon*, é necessário elucidar o que vem a ser um *pattern*, também referenciado ao longo do texto como padrão.

Um *pattern* pode ser considerado como sendo um conjunto de operandos e operadores, os quais representam um determinado trecho de um programa. Esses operandos e operadores que compõem um padrão podem ser representados na forma de grafos, como pode ser visto na Figura 2.1, representando a expressão "r[32]=(((r32]+ (-1)) == 0)".

Os operadores, representados por elipses nos grafos, são instruções em linguagem de máquina (*Assembly*) ou parte delas que servem para representar as operações efetuadas em operandos, por exemplo: soma, multiplicação, módulo, etc [12]. Um operador utiliza os valores dos operandos para efetuar sua computação (entradas) e caso exista resultado ou produto da execução da operação, este será armazenado em um outro operando (saída).

Já os operandos, representados por quadrados nos grafos, contém ou armazenam os

valores envolvidos nas computações dos operadores. Os operandos podem ser constantes reais ou inteiras, registradores, etc. Caso o operando seja um registrador, este pode ter seu valor definido por um operador [12].

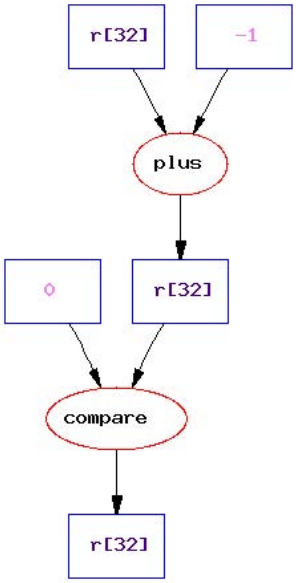

Figura 2.1: Exemplo de *pattern* extraído do MiBench [15]

2.2 O Projeto *Chameleon*

Diversas são as etapas necessárias para que um trecho de programa venha a ser implementado como novas instruções em *hardware*, como pode ser observado no diagrama da Figura 2.2. As etapas são descritas a seguir:

1. Compilação Para Leon (SPARC): O código fonte é compilado para a arquitetura SPARC, implementado pelo processador Leon, utilizando o compilador GCC. O código é compilado utilizando diretivas para instrumentar o código de forma a que se possa determinar a quantidade de vezes que cada conjunto de instruções foi executado.

2. Execução do programa: Uma vez instrumentado, o programa é executado e as informações de *profiling* são geradas no formato do compilador GCC.

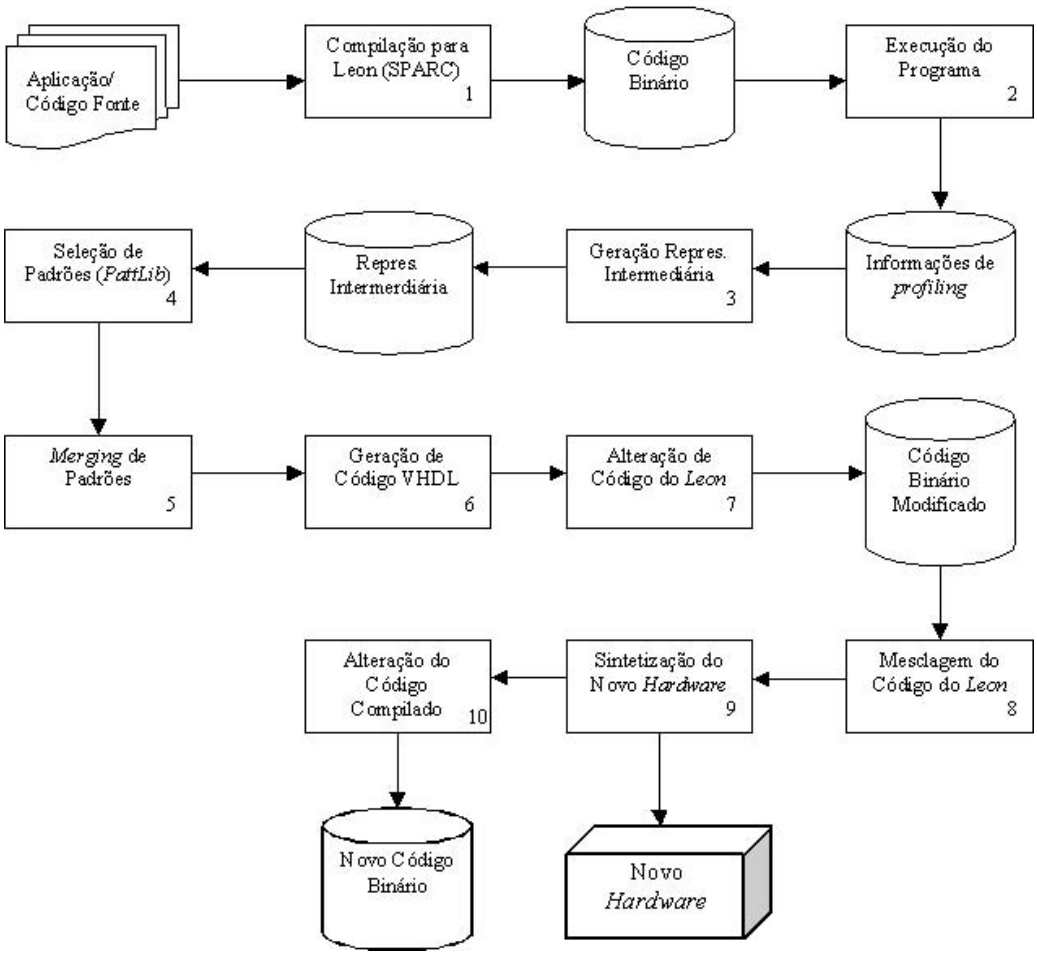

Figura 2.2: As fases do processo do Projeto *Chameleon*

3. Geração de Representação Intermediária: De posse das informações de *profiling*, estas são traduzidas para uma representação intermediária na forma de grafos, manipuláveis através da biblioteca de programas *PattLib* (*Pattern Library*).

   A *Pattern Library* é uma biblioteca de gerenciamento de padrões que fornece uma API (*Application Programming Interface*) escrita em linguagem C [11] para gerenciamento e manipulação de grafos (*patterns*) que representam instruções de um determinado programa. Entre suas funcionalidades estão ler/escrever grafos no sistema de arquivos, comparação de grafos além da facilidade de navegação nos diversos conjuntos de instruções de código dos programas.

   Dentre as vantagens de se possuir uma representação intermediária, está o fato do projeto como um todo ser facilmente customizável e estendido, uma vez que novas ferramentas podem ser desenvolvidas tomando como entrada e saída de dados essa mesma representação, ao invés de necessitar que o usuário tenha conhecimento de todo o processo, desde a etapa de tradução dos programas para uma representação particular, passando pela manipulação e obtenção de resultados específicos e terminando com a geração de novas instruções ou programadas adaptados implementados em hardware da plataforma destino, como em [2, 3, 18, 20].

4. Seleção dos padrões: Uma vez gerados os *patterns* (grafos com as instruções) na representação da *PattLib*, se faz necessário decidir aqueles que serão implementados em hardware. Esse é o escopo dessa dissertação e das ferramentas aqui desenvolvidas; de posse dessas ferramentas, o usuário do sistema pode aplicar diversos filtros de forma a determinar os melhores candidatos a implementação em hardware. Entre os filtros, destaca-se a possibilidade de determinar padrões aritméticos (ou seja, cujos operadores sejam soma, subtração, multiplicação ou divisão - apesar desses dois últimos operadores terem um custo mais alto), determinação da quantidade de operandos de entrada e de saída ou ainda a seleção do padrão baseado no seu custo (quantidade de operações efetuadas entre os operandos de entrada e saída do padrão).

   Os padrões elegíveis para seleção são idealmente aqueles que tem um alto custo computacional e que sejam executados em grande parte do tempo de vida dos programas, pois não há valia em se gerar código em hardware para um conjunto de instruções

que são executadas poucas vezes ou então fazem parte de uma determinada rotina de tratamento de erros.

Fica a critério do usuário a administração da biblioteca de padrões, podendo ele adicionar manualmente outros padrões ao conjunto daqueles que irão ser traduzidos para hardware.

5. *Merging* de Padrões: Uma vez selecionados os padrões, é feita uma tentativa de *merging* (unificação) de forma a reaproveitar as suas reconexões e reutilizar suas unidades funcionais.

6. Geração de Código VHDL: Tendo como base o código resultado do *merging* dos padrões, é gerado módulo em VHDL que implementa a instrução o qual será integrado ao processador Leon.

7. Alteração do código do processador Leon: O código do processador é alterado de forma a suportar novas instruções criadas pelo usuário.

8. Mesclagem do código: O código VHDL das instruções selecionadas é mesclado ao código do processador.

9. Síntese em Hardware: O código mesclado é sintetizado em hardware, utilizando FPGA's, porém na indústria poderia ser feito o uso de máscaras de confecção de ASICS.

10. Alteração do código compilado: Uma vez que um determinado conjunto de instruções é transportado para a CPU, novas instruções de máquina são geradas. Os programas que fazem uso destes padrões precisam ser alterados para que invoquem essas recém-criadas instruções possibilitando assim o ganho de desempenho desejado.

2.3 Trabalhos Relacionados

A seguir serão apresentados trabalhos relacionados relevantes ao desenvolvimento desse projeto. Os trabalhos estão agrupados por área de estudo, a começar pelos trabalhos

gerais relacionados a arquiteturas reconfiguráveis como Kumar [17] que descreve as características necessárias aos sistemas de arquiteturas reconfiguráveis, onde vários desses baseiam-se em simulações sem atingir implementações concretas [8, 13], passando pelos trabalhos voltados para a manipulação de bits e terminando com alguns trabalhos que enfatizam a criação de *benchmarks* para caracterizar programas em plataformas embarcadas.

No relatório técnico [6] é observado que a Computação Reconfigurável é proposta para preencher o vazio entre o hardware e o software, alcançando assim um desempenho maior que o do software enquanto mantém seu alto nível de flexibilidade.

Sua pesquisa caracteriza Sistemas Reconfiguráveis como sendo formados por uma combinação de lógica reconfigurável e um microprocessador de propósito geral, onde o processador executa as operações que não podem ser efetuadas eficientemente na lógica reconfigurável como laços e acessos a memória enquanto os núcleos computacionais são mapeados em hardware reconfigurável (FPGA's [9] comerciais ou ainda hardware customizados dedicados).

A autora faz ainda uma descrição dos ambientes de compilação para arquiteturas reconfiguráveis, onde estes ambientes podem ser compostos desde ferramentas para auxiliar um programador a mapear um circuito no hardware até ferramentas completamente automatizadas.

Ela ainda indica que o processo de projeto de Sistemas Reconfiguráveis envolve a seleção das partes de um programa a serem implementadas em hardware reconfigurável e em software, tradução da parte destinada ao hardware em linguagem específica, compilação e por fim, configuração no equipamento em tempo de execução. Essas tarefas, quando executadas automaticamente acarretam em mínimo esforço embora quando feitas manualmente resultam em circuitos mais otimizados.

Em se tratando de artigos relacionados à manipulação de bits, o *Bitwise* é um compilador que se propõe a minimizar a quantidade de bits (*bitwidth*) necessária para representar operandos - variáveis inteiras ou então apontadores de endereços de memória em aplicações relacionadas a multimídia e transmissão de imagens e som (*streaming*), uma vez que grande parte das operações efetuadas nesses operandos não fazem uso de todos 32 bits.

Para isso, **Stepheson[18]** sugere um algoritmo que executa o programa do começo

ao fim e de forma reversa para determinar bits invariantes em instruções estáticas de maneira a determinar a menor quantidade de bits necessários para manter a corretude da aplicação e obter o maior estreitamento possível na quantidade de bits nestas instruções.

Outro aspecto importante é que o compilador incorpora técnicas sofisticadas de análise de laços, pois parte significante das instruções de um programa que são executadas dinamicamente estão dentro de tais blocos; essas técnicas envolvem a determinação e classificação de seqüências, um grupo de instruções mutuamente dependentes no grafo de dependências de um programa. Uma vez determinada as seqüências, parte-se para encontrar seus invariantes e em seguida proceder com a remoção dos bits desnecessários, de forma que a resultante da otimização das seqüências de um laço representa a otimização deste como um todo.

Uma vez feita a análise e reduzida a quantidade de bits necessários, o programa é compilado de forma a obter ganhos como a redução da área de lógica nos circuitos (de 15% a 18%), aumento da velocidade de execução (de 3% a 249%) e redução no consumo de energia do sistema do processador(de 46% a 73%).

Já **Wagner [20]** aborda o projeto de um compilador industrial para processadores ASIP (*Application Specific Instruction Set Processors*) o qual é dotado de instruções especiais para acesso a registradores de dados em nível de bits, os quais são necessários para processamento de protocolos de comunicação orientados a pacotes (*NP - Network Processors*).

A motivação para a criação de tais processadores (no caso, o *NP*) urge da crescente demanda por equipamentos que suportem altas velocidades de transmissão de dados em rede (por exemplo, roteadores internet ou adaptadores *Ethernet*), onde o fluxo de dados consiste em pacotes de diferentes tamanhos - cabeçalho do pacote e *payload* com dimensões variáveis.

O conjunto de instruções do *hardware NP* permite computação ALU [9] de pacotes de bits que não estão alinhados com o tamanho de palavra do processador, onde um pacote pode estar armazenado, por exemplo, em uma parte de dois diferentes registradores. Para isso, o endereçamento de um pacote é representado pelo número identificador do registrador, seu deslocamento nesse registrador além do tamanho do pacote. Caso a soma do deslocamento com o tamanho do pacote exceda o tamanho da palavra do registrador, então o pacote se alastra por dois registradores, o que torna o projeto do compilador mais

desafiante, segundo o autor.

Tal mecanismo de endereçamento bem como funções específicas para manipulação de bits (otimização das operações de *shift* e *mask*) são implementadas pelo compilador na forma de funções conhecidas específicas (*CFK - Compiler Known Functions*). No caso de programas já existentes, é necessário que o programador traduza as rotinas de manipulação de bits existentes (por exemplo, escritas em linguagem C padrão [11]) para fazerem uso das CFK's - o que pode-se tornar um processo penoso.

Apesar desta dificuldade, tal compilador se apresenta como uma alternativa a montagem manual de programas para a arquitetura de *hardware NP*, o que seria um processo que consumiria uma quantidade ainda maior de tempo e esforço, além de ser sujeito a erros.

Em **[2, 3], Budiu** apresenta técnicas de análise e otimização de compilador para que seja possível deduzir de forma automática a quantidade mínima de bits necessária para que uma determinada computação possa ser processada, mesmo que essa computação seja especificada com uma quantidade maior de bits. Com o uso dessas técnicas aplicadas a compiladores para *hardware* reconfiguráveis é possível atingir um ganho na ordem de 20 vezes em relação ao tamanho do circuito sintetizado.

A motivação para tais otimizações vem do fato de que importantes aplicações executam computações com largura de bits (*bitwidth*) diferentes das encontradas em linguagens de alto nível que disponibilizam ao programador uma quantidade limitada de tamanhos (8, 16, 32 e 64 bits) o que acarreta num desperdício de recursos e conseqüentemente uma performance não ótima da aplicação.

O algoritmo proposto infere informações a respeito de cada bit de todos os valores de um programa. Cada análise se propõe a computar um diferente atributo de cada operando: uma para computar se este possui sinal ou não (por exemplo, no caso de variáveis do tipo inteiro), duas para computar o tamanho máximo e uma análise para computar o tipo do operando.

Tais análise são efetuadas seguindo o fluxo normal de execução do programa *(forward analysis)* de forma a obter uma determinada informação a respeito de suas saídas levando em conta os valores de entrada, bem como na ordem reversa *(backward analysis)* para inferência de valores de entrada tendo em conta os valores de saída. Assim como alguma dessas análises são iterativas, recalculando valores até que um trecho do programa seja

atingindo, outras efetuam uma única passagem pelo programa.

Uma vez efetuadas tais análises, é possível concluir os bits significativos em cada operando e aqueles que são desnecessários. Com base nessas informações, as aplicações são recompiladas obtendo a redução de tamanho desejada e a maximização da performance de execução.

Conforme pode ser observado nos artigos descritos acima, os mesmos objetivam o desenvolvimento de ferramentas para a análise de bits e padrões de bits num determinado contexto ou conjunto de programas. Para isso, os autores desenvolveram toda uma infraestrutura de apoio, quase sempre reinventando a roda, ao invés de concentrar seus esforços na análise propriamente dita. Já com o *Chameleon* o mesmo não ocorreria, uma vez que de posse da *PattLib*, o projeto trataria de traduzir os programas desejados para a representação intermediária dessa biblioteca viabilizando assim um estudo detalhado objeto do trabalho, como por exemplo, determinação de padrões de bits, ao invés de despender esforço desnecessário nos processos de geração da representação intermediária bem como da tradução dessa representação para o *hardware* da plataforma desejada.

2.4 Motivação

A biblioteca de padrões (*PattLib - Pattern Library*) que faz parte do projeto Chameleon, tem como responsabilidade a gerência de padrões de instruções de código. Cabe a ela a interpretação da representação intermediária do compilador GCC e a geração de arquivos textos facilmente manipuláveis na forma de Grafos.

Não cabe a *PattLib* a filtragem e seleção dos padrões que serão elegíveis para geração em hardware - inicialmente o processador Leon. Dessa forma, fez-se necessário a criação de uma biblioteca que fizesse uso das funcionalidades da *PattLib* mas que provesse um conjunto maior de funcionalidades orientadas ao usuário do projeto.

Essas funcionalidades compunham inicialmente um conjunto de filtros que pudessem ser concatenados na forma de expressões lógicas (AND, OR e NOT) de modo a obter um conjunto de padrões o mais próximo da realidade do usuário. Entre os filtros estão a busca por um determinado tipo de operador ou ainda a quantidade de entradas de um grafo. O usuário pode aplicar individualmente os filtros via linha de comando ou ainda utilizar da API (*Application Programmin Interface*) de forma a criar expressões lógicas

do tipo: selecione todos os padrões que contenham o operador ADD e que tenham três operadores de entrada.

Observou-se ainda que em determinados momentos, o usuário não tinha a noção de quais filtros ou critérios de seleção este desejaria aplicar para a escolha dos padrões - sendo assim, ao trabalho proposto foi adicionada a possibilidade de percorrer um conjunto de padrões (grafos) e coletar diversos dados estatísticos sobre os mesmos. Esses dados estatísticos são de grande valia para a geração de histogramas (inicialmente fazendo uso do utilitário de domínio público GnuPlot) onde o usuário tem a possibilidade de ter uma visão geral do comportamento dos padrões que formam a sua biblioteca para que em seguida possa decidir os quais serão selecionados para posterior geração em VHDL.

2.5 Principais Contribuições

Nesse capítulo foi descrito o projeto *Chameleon* como um todo, percorrendo, em alto nível, cada uma das etapas de seu processo. Mais adiante, foram apresentados primeiramente trabalhos gerais a área, trabalhos relacionados a manipulação e detecção de padrões de bits em programas aplicativos onde pode-se observar que toda uma infra-estrutura necessária era sempre recriada para que um estudo pudesse ser feito.

De posse das informações obtidas nesses trabalhos relacionados, os objetivos principais do trabalho proposto são dois:

A **utilização de filtros de características de trechos de programas** de forma que o usuário possa concentrar seus esforços em criar filtros e/ou efetuar filtragem de blocos de código em busca de determinada característica no seu domínio de aplicação. Para isso, o trabalho proposto faz amplo uso da infra estrutura e representação intermediária disponibilizada pelo *Chameleon* na tentativa de reusar a maior quantidade de código possível de forma a criar componentes de *software* facilmente encaixáveis em qualquer contexto.

Já o segundo objetivo desse trabalho é **contabilizar dados estatísticos** aproveitando toda a lógica e critérios aplicados na criação dos filtros, só que dessa vez, ao invés de apontar quais trechos de código satisfazem determinada condição, é possível inferir quantitativamente quais trechos possuem, por exemplo, uma quantidade maior de instruções de manipulação de bits. Especificamente, serão utilizados o MiBench e o MediaBench

para coleta e caracterização de dados estatísticos relacionados a seus *benchmarks*.

Sendo assim, o próximo capítulo trata de detalhar tanto as funções de filtragem de padrões quanto as facilidades estatísticas disponibilizadas pela biblioteca *Pattern Matcher*, a qual serve como base para viabilizar tais funcionalidades e será descrita em seguida.

Capítulo 3

A biblioteca para Seleção de Padrões: Pattern Matcher

No capítulo anterior observou-se a existência da *PattLib*, a qual disponibiliza a funcionalidade de uma representação intermediária para que essa possa ser manipulada por outras ferramentas de análise de padrões de código.

O presente capítulo relata o desenvolvimento da biblioteca de filtragem de padrões (*Pattern Matcher*, ou simplesmente PM) a qual tem como característica principal a viabilização de buscas de padrões que satisfaçam um determinado conjunto de critérios (ou filtros) de pesquisa por parte do usuário. Esses filtros podem ser combinados de forma a obter expressões lógicas baseadas nos operadores *AND, OR e NOT* viabilizando um mecanismo mais sofisticado de filtragem. Baseando-se também nesses filtros, é possível a coleta de dados estatísticos para posterior análise, quando for necessário por exemplo, determinar a quantidade de padrões que atendam a um determinado critério. O escopo deste trabalho compreende o desenvolvimento da biblioteca *Pattern Matcher* bem como todas as características e funcionalidades intrínsecas a mesma.

A ferramenta aplica o conceito de *pattern-to-pattern*, ou seja, tanto os dados de entrada quanto os dados de saída (resultado dos filtros) são *patterns* (padrões). Com isso, é possível fazer com que o resultado da execução de um filtro sirva como entrada para a execução de outros filtros e assim por diante até que se obtenha os padrões que satisfaçam todos os critérios estabelecidos como pode ser visto na Figura 3.1

Viu-se posteriormente que o PM, em virtude da sua habilidade de percorrer os grafos contendo os padrões de código, poderia ser estendido para disponibilizar funções de coleta

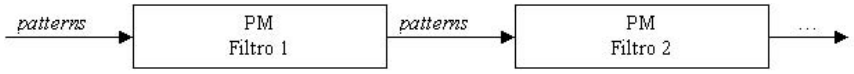

Figura 3.1: Descrição do princípio das ferramentas *pattern to pattern.*

estatística sobre um determinado conjunto de padrões - essas estatísticas são de valia no momento em que o usuário necessita inferir sobre a sua biblioteca de padrões ou então, por exemplo, quando este ainda não possui nenhum critério estabelecido para filtragem de padrões e deseja coletar dados iniciais. De posse dos dados estatísticos, é possível a geração de gráficos (histogramas) inicialmente utilizando a ferramenta GnuPlot [16].

3.1 Funcionalidades da Biblioteca

As funcionalidades da biblioteca estão divididas em três grandes grupos: filtros de padrões, funções estatísticas e funções utilitárias, a saber:

Filtros de Padrões: Um filtro pode ser caracterizado como sendo um critério estabelecido por um usuário no que se refere as características que precisam ser observadas em um trecho de código para que este o satisfaça. Filtros podem ser integrados de forma conjunta para gerar expressões mais complexas de acordo com a necessidade.

Abaixo serão levantados os filtros que o PM disponibiliza em sua biblioteca:

Filtragem por tipo de operador: Retorna padrões que possuam entre seus operadores algum de um tipo especificado, como por exemplo, operador de deslocamento de bits, operador aritmético de soma e assim por diante. Os operadores são identificados por números constantes os quais encontram-se no arquivo de *header* da *PattLib*.

Filtragem por conjunto de operadores: Diferentemente do filtro por tipo de operador que busca padrões que contenham operadores de um determinado tipo, esse filtro busca padrões que contenham única e exclusivamente um determinado conjunto de operadores. Fica a cargo do usuário decidir se os operadores necessitam estar todos presentes nos padrões (opção `ALL`) ou se a existência de um subconjunto desses operadores no padrão torna o critério válido

(`ANY`). Deve-se observar que em ambos os casos a existência de um operador que não pertença ao conjunto de operadores especificados no padrão descaracteriza o filtro.

Filtragem por quantidade de operandos de entrada ou saída: Seleciona os padrões de código que contenham uma determinada quantidade de operandos ou de entrada ou de saída. A quantidade desejada é especificada na forma de operadores de comparação menor ou igual, menor que, igual, maior que, e por fim maior ou igual. Por exemplo, pode-se querer saber quais os padrões que contenham pelo menos cinco operadores de entrada.

Filtro por peso de operadores: Escolhe os padrões que possuem um determinado peso. Para isso, é estabelecido que um ou mais operadores têm um peso especificado pelo usuário, ou seja, pode se desejar que os operadores de multiplicação e divisão assumam um peso arbitrário cinco em virtude de seus processos computacionais serem mais pesados. Os outros operadores assumirão o peso padrão: uma unidade. Levando esses aspectos em conta, pode-se selecionar por exemplo todos os padrões que possuam peso total de 15 unidades onde os operadores de soma tenham peso dois e os operadores de multiplicação tenham peso quatro. Ao peso do operador pode ser atribuído uma característica de grandeza escalar como o tempo de execução de uma dada operação, o custo de implementação em *hardware*, etc.

Estatísticas de padrões: Estatísticas de padrões aproveitam parte da lógica computacional dos filtros, mas diferentemente destes, onde pode-se avaliar simplesmente se um trecho de código atende às características de um determinado filtro ou não, com as funções estatísticas é possível computar quantitativamente quais trechos de programas atendem a um determinado critério.

Por exemplo, é possível computar a quantidade de instruções aritméticas por trecho de código, onde caso um determinado trecho de programa não contenha funções com tal característica, será contabilizado zero para o mesmo. Do contrário, no caso de um filtro de instruções aritméticas, esse apontaria apenas os padrões que possuem as mesmas.

Abaixo serão levantadas as funções de coleta estatística suportadas pela biblioteca PM:

Estatística de tamanho de padrões: Computa num determinado conjunto de grafos o tamanho dos padrões. Por tamanho entenda-se a quantidade de operadores e a quantidade de operandos de um determinado padrão.

Estatística de quantidade de operandos de entrada e saída: Levanta a quantidade de operandos de entrada e de saída de uma lista de padrões. Com essa informação, é possível posteriormente que o usuário selecione apenas um determinado subconjunto dos padrões, aqueles que satisfizerem uma determinada quantidade desejada.

Estatística de peso de padrões: Semelhante ao filtro por peso de padrões, este critério de estatística faz o levantamento do peso total de um determinado conjunto de grafos, onde os pesos podem ser diferenciados a critério do usuário.

Estatística de operadores lógicos e aritméticos: Retorna os padrões que contenham apenas operadores que sejam de lógica (como AND, OR e assim por diante) bem como os operadores que sejam aritméticos com exceção dos operadores de multiplicação, divisão e resto, em virtude do seu alto custo.

Outras Funcionalidades: Tendo em vista que a ferramenta é do tipo *pattern-to-pattern*, ou seja, a entrada de um filtro é um pattern (padrão) e o resultado também é um padrão, duas funções utilitárias foram desenvolvidas:

União: Processa a união de dois arquivos de blocos de instruções em um terceiro. Esta operação de união leva em consideração a identidade dos padrões de forma que se dois padrões contém um conjunto de instruções em comum, este só estará presente uma única vez no arquivo resultado.

Intersecção: Contrariamente a união, faz a fusão de dois arquivos que contém padrões, onde o arquivo resultante conterá apenas os padrões comuns aos dois arquivos de entrada.

3.2 Formas de Acesso

Existem duas formas de utilização da ferramenta: linha de comando ou chamada de função via APIs do sistema. A primeira permite que um filtro ou função estatística seja invocada por vez, seguindo o conceito *pattern-to-pattern* onde o resultado obtido da execução de um filtro pode servir como dado de entrada para uma próxima execução da ferramenta; já a execução por linha de comando viabiliza o uso da *Pattern Library* por parte de outros programas, via chamada de função em linguagem C, permitindo que filtros mais sofisticados sejam estabelecidos, conforme será descrito em seguida.

3.3 Invocação por Linha de Comando

A ferramenta possibilita que um filtro seja aplicado por vez. Caso surja a necessidade de aplicação de mais de um filtro, esta pode ser feita tomando como entrada o resultado da execução do filtro anterior.

Para sua utilização, o programa principal deverá ser invocado sempre passando como dois primeiros parâmetros o nome do arquivo que contém os padrões de entrada a serem filtrados juntamente com o nome do arquivo que conterá os padrões filtrados (saída). De posse destas informações, restará informar a operação desejada bem como seus parâmetros pertinentes.

`pm <arquivo entrada> <arquivo de saída> <parâmetros restantes>`

Os parâmetros restantes podem ser:

`OPT_TYPE <operator#>`: Filtragem por tipo de operador, seguido do número que represente o operador desejado. Por exemplo, todos os padrões que contenham o operador MOV (código 54 de acordo com a biblioteca *PattLib*).

```
pm adpcm_decoder.pat out.pat OPT_TYPE 54
```

`OPERATOR_SET <ANY/ALL> <operator#1> <operator#2> ... <operator#n>`: Filtragem por conjunto de operadores seguido da indicação de se deve considerar todos os operadores ou apenas alguns além da lista desses operadores. Por exemplo, todos os padrões que contenham apenas o operador MOV.

```
pm adpcm_decoder.pat out.pat BASIC_BLOCK ANY 54
```

Obs: Como existe apenas um operador a ser utilizado com critério de busca o parâmetro `ANY/ALL` não altera o resultado da operação - o contrário ocorreria apenas quando do uso de dois ou mais operadores.

`OPD_# <type> <I/O> <function> <count>:` Filtragem por quantidade de operandos (de entrada ou de saída), função lógica e quantidade. As funções lógicas podem ser:

`LT:` Menor que

`LE:` Menor ou igual

`EQ:` Igual

`GT:` Maior que

`GE:` Maior ou igual

Por exemplo, todos os padrões que contenham cinco ou mais operandos como entrada do padrão:

```
pm adpcm_decoder.pat out.pat OPD_# I GT 5
```

`UNION <result file>:` Função utilitária para união de padrões. Neste caso, os dois primeiros parâmetros tornam-se os arquivos a serem unidos e o quarto parâmetro especifica o arquivo a conter o resultado da união.

Por exemplo, união de todos os padrões dos arquivos output1.pat e output2.pat:

```
pm output1.pat output2.pat UNION union_file.pat
```

`INTERSECT <result_file>:` Função utilitária para intersecção de padrões. Funciona de forma análoga a função de união, onde os dois primeiros parâmetros são os arquivos a terem a intersecção feita e o quarto parâmetro indica o arquivo a armazenar o resultado da operação.

Por exemplo, intersecção de todos os padrões dos arquivos output1.pat e output2.pat:

```
pm output1.pat output2.pat INTERSECT insersect_file.pat
```

`PATT_WEIGHT <weight to match> <OP1> <WGT1> <OP2> <WGT2>... :` Filtro para buscar padrões que atinjam um determinado peso. Os parâmetros indicam o peso a ser

levado em conta seguido dos operadores e seus pesos a serem utilizados. Caso não sejam especificados operadores e pesos, será assumido o peso um como regra.

Por exemplo, todos os padrões que tenham peso 10, adotando o peso quatro para o operador de multiplicação (constante de valor numérico 13).

```
pm adpcm_decoder.pat out.pat PATT_WEIGHT 10 13 4
```

`STATISTICS LOGIC_ARITH`: Função estatística que aponta todos os padrões que contém apenas operadores aritméticos ou lógicos os quais compreendem `NOT, AND, PLUS, SS_PLUS, US_PLUS, LO_SUM, MINUS, SS_MINUS, US_MINUS, COMPARE` e `NEG`.

Por exemplo, computar os padrões que contenham apenas operadores lógicos:

```
pm adpcm_decoder.pat out.pat STATISTICS LOGIC_ARITH
```

`STATISTICS PATT_DEPTH`: Função de estatística que computa a profundidade dos padrões de um determinado arquivo. Por profundidade entenda-se a quantidade de operadores entre o operando de entrada e o operando de saída.

Por exemplo, calcular o tamanho dos padrões contidos no arquivo adpcm_decoder.pat:

```
pm adpcm_decoder.pat out.pat STATISTICS PATT_SIZE
```

`STATISTICS PATT_IO`: Função de estatística que computa a quantidade de operandos e operadores dos padrões num arquivo.

Por exemplo, calcular a quantidade de operadores de entrada e de saída dos padrões contidos no arquivo adpcm_decoder.pat:

```
pm adpcm_decoder.pat out.pat STATISTICS PATT_IO
```

`STATISTICS PATT_WEIGHT <OP1> <WGT1> <OP2> <WGT2>...` : Computa o peso dos padrões de um arquivo de forma análoga ao filtro pelo peso dos padrões. A diferença nesse caso é que no modo de estatísticas não é estabelecido um peso como critério. Neste caso, o peso de todos os padrões são calculados e disponibilizados.

Por exemplo, computar o peso de todos os padrões considerando que o operador de multiplicação (código 13) tem peso quatro:

```
pm adpcm_decoder.pat out.pat STATISTICS PATT_WEIGHT 13 4
```

3.4 Invocação por Chamada de Função da API

O PM disponibiliza uma API escrita em linguagem de programação C que pode ser utilizada por um usuário programador que deseje embutir em sua aplicação funcionalidades do projeto, da mesma forma que o PM faz uso da biblioteca PattLib.

De posse das funções da API é possível montar uma árvore com os filtros (critérios) a serem aplicados onde para que um padrão seja considerado válido é necessário que este atenta às operações lógicas encontradas na árvore de filtros. Esta árvore é uma árvore binária onde o processo de avaliação se dá partindo do lado esquerdo para o lado direito, sendo que se por exemplo as folhas do lado esquerdo de uma árvore retornarem falso e o operador for um AND não há necessidade de avaliar o lado direito da árvore (*lazy evaluation*). O oposto também é aplicado quando do operador OR.

Para que a árvore possa ser montada é necessário que seja criados nós, onde um nó poder conter uma operação lógica ou filtro a ser aplicado. A estrutura do nó é a seguinte:

```
typedef struct tree_node_t {
  node_type n_type;
  union u_node {
    node_data_logic *logic_data;
    node_data_func  *func_data;
  };
  int result;
  void aux*;
} tree_node;
```

onde:

- `n_type`: É o tipo do nó, lógico ou que contenha uma operação a ser efetuada;
- `logic_data`: Caso o nó seja lógico, contém a operação lógica a ser efetuada, bem como nós a esquerda e a direita;
- `func_data`: Caso o nó seja de função contém o filtro a ser executado;

- `result`: Campo auxiliar que armazena o resultado lógico da avaliação do nó;
- `aux`: Utilizado internamente quando computando funções de estatística.

A estrutura do nó lógico é a seguinte:

```
typedef struct node_data_logic_t  {
  logical_operation lo;
  struct tree_node_t *node_left;  /* left node  */
  struct tree_node_t *node_right; /* right node */
} node_data_logic;
```

onde:

- `lo`: É a operação lógica a ser efetuada nos nós da esquerda e da direita;
- `node_left`: Representa o nó abaixo e a esquerda do nó atual;
- `node_right`: Representa o nó abaixo e a direita do nó atual.

Já as operações lógicas que podem ser aplicadas num nó são definidas na estrutura a seguir:

```
typedef enum logical_operation_t  {
  AND,
  OR,
  NOT
} logical_operation;
```

É importante ressaltar que o operador `NOT` só pode ser aplicado a nós com apenas um único filho.

No caso dos nós que contenham operações (filtros) estes são representados da seguinte forma:

```
typedef struct node_data_func_t  {
  int (*function)(void *, void *);
  void *criteria;
} node_data_func;
```

onde:

- `function`: Representa o filtro ou função estatística a ser computada nos padrões;
- `criteria:` Contém os parâmetros necessários a execução do filtro.

De posse das estruturas necessárias para montagem da árvore de execução, duas funções da API são disponibilizadas para criação de nós lógicos e de função:

```
tree_node* mk_l(logical_operation, tree_node* left,
                tree_node* right);
```

Onde `mk_l` pode ser entendido por *make logical node*. Os parâmetros da função são os seguintes:

- `operation`: A operação lógica;
- `left`: O nó inferior a esquerda do nó a ser criado;
- `right`: O nó inferior a direita do nó a ser criado.

O retorno da função é um objeto do tipo `tree_node` que contém os nós da esquerda e da direta juntamente com a operação lógica a ser computada.

Para criação de nós que contenham filtros utiliza-se a função abaixo:

```
tree_node* mk_f(void *function, void *criteria);
```

Onde `mk_f` pode ser entendido por *make function node*, especificando os seguintes parâmetros:

- `operation`: representa o filtro a ser aplicado;
- `criteria`: representa os parâmetros do filtro a ser aplicado.

Exemplo:

Deseja-se criar uma árvore para a seguinte expressão:

```
(((PLUS && AND) && (NOT && COMPARE)) && (!LD)) && (IN>=3 && (OUT>1 &&
                                                    OUT<3))
```

Onde:

`PLUS` representa um filtro que busca padrões que contenham o operador `PLUS`. A mesma analogia pode ser aplicada para as expressões `AND`, `NOT`, `COMPARE` e `LD`. Já `IN>= 3` representa um filtro que busca padrões que contenham no mínimo três operadores de entrada. A mesma analogia pode ser aplicada para `OUT>1` e `OUT<3`.

A representação gráfica da árvore é descrita na Figura 3.4:

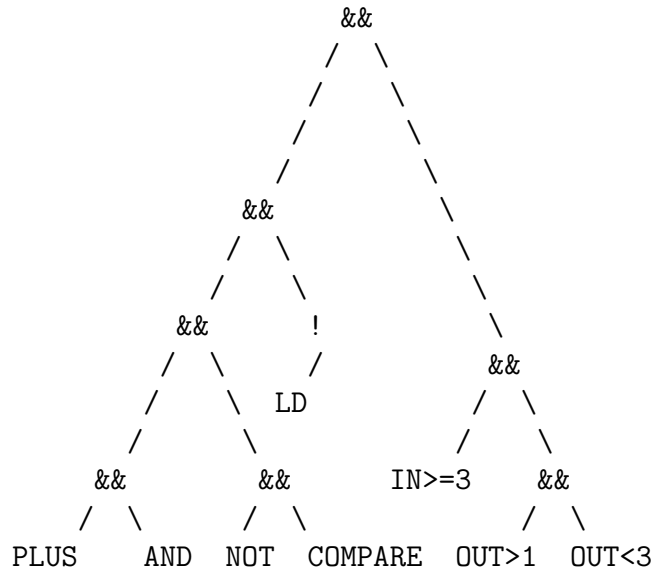

Figura 3.2: Representação gráfica da árvore a ser implementada pelo conjunto de funções `MK_F` e `MK_L`.

Para tanto, o seguinte conjunto de funções deve ser invocado:

```
tree_nde *root =
  mk_l(AND,
    mk_l(AND,
      mk_l(AND,
        mk_l(AND,
          mk_f(PM_MATCH_OPERATOR_MATCHES, plus),
```

```
        mk_f(PM_MATCH_OPERATOR_MATCHES, and)
      ),
        mk_l(AND,
        mk_f(PM_MATCH_OPERATOR_MATCHES, not),
        mk_f(PM_MATCH_OPERATOR_MATCHES, compare)
      )
    ),
    mk_l(NOT,
      mk_f(PM_MATCH_OPERATOR_MATCHES, ld), NULL)
  ),
  mk_l(AND,
    mk_l(AND,
      mk_f(PM_MATCH_IO_OPERAND_COUNT, criteria1),
      mk_f(PM_MATCH_IO_OPERAND_COUNT, criteria2)
    ),
    mk_f(PM_MATCH_IO_OPERAND_COUNT, criteria3)
  )
);
```

Capítulo 4

Análise dos *Benchmarks MiBench e MediaBench*

O capítulo anterior apresentou a biblioteca para seleção de padrões, onde foram descritas suas formas de utilização, funcionamento e suas funções de captura estatística.

Este capítulo se destina a descrever os *benchmarks MiBench e MediaBench* e a apresentar os resultados obtidos da execução da ferramenta junto a estes *benchmarks* de forma a caracterizar tanto os aspectos comuns quanto as características particulares a cada um deles - essas informações são de grande utilidade para os projetistas que pretendem implementar padrões em *hardware*.

4.1 *Os Benchmarks MiBench e MediaBench*

Na última década, diversas arquiteturas de microprocessadores emergiram voltados para as áreas de telecomunicações e multimídia. Com isso, percebeu-se que os *benchmarks* existentes e que eram utilizados em larga escala para arquiteturas de uso geral, não se adequavam para esses novos sistemas em surgimento;

Esses *benchmarks* tem como objetivo servir de parâmetro para aferir se uma arquitetura ou versão de microprocessador possui desempenho superior a uma outra, viabilizando assim um estudo comparativo entre dois ou mais sistemas. Dessa forma, diante da necessidade da existência de *benchmarks* voltados para a área de telecomunicações e multimídia, foi criado o **MediaBench [4]** que diferentemente do MiBench [15], tem aplicação mais específica e é de domínio mais restrito.

Os objetivos do MediaBench são:

1. Caracterizar de forma precisa o poder de processamento de sistemas multimídia e de telecomunicações;

2. Concentrar-se em aplicações escritas em linguagens de alto nível e que sejam portáveis entre arquiteturas de processador, uma vez que essa é uma tendência do desenvolvimento de *software*;

3. Estabelecer os benefícios do MediaBench comparados com o SPEC.

4. Desenvolver ferramentas que sirvam tanto para avaliação quanto para a síntese de sistemas.

As aplicações (ou *benchmarks*) que compõem o MediaBench são todas de domínio público, fato esse importante, pois viabiliza sua disseminação na comunidade. Outro fator importante a ser ressaltado é que essas aplicações são todas escritas em linguagens de alto nível - facilitando assim a portabilidade e conseqüentemente viabilizando o uso na maior quantidade de plataformas possível, ao invés de escrever código em linguagem de máquina (*Assembly*), voltadas exclusivamente a determinadas plataformas e de alto grau de dificuldade no tocante a portabilidade para outras arquiteturas.

O MediaBench é composto pelos seguintes *benchmarks*:

JPEG: Um dos métodos padronizados para compressão e descompressão de imagens; seus testes envolvem tanto imagens coloridas quanto preto e branco. São duas as aplicações envolvendo o JPEG: *cjpeg* para a compressão de imagens e *djpeg* para a descompressão.

MPEG: Considerado um dos padrões *de facto* em compressão de vídeo digital de alta qualidade, as aplicações que fazem uso do formato são o *mpeg2enc* e *mpeg2dec* para codificação e decodificação de arquivos de vídeo, respectivamente.

GSM: É o padrão europeu para transcodificação de voz, com taxa de transferência de 13 kbit/s. As aplicações envolvem tanto codificação quando decodificação de trechos de voz, simulando assim um diálogo em redes de telefonia celular.

Compressão de Voz G.721: Faz parte da implementação de referência dos padrões G.711, G.712 e G.713 do CCITT (*Iternational Telegraph and Telephone Consultative Committee*) para compressão de voz.

PGP: Utiliza o conceito de assinaturas digitais, onde uma assinatura é composta por uma função de *hash* de uma mensagem com tamanho de 128 bits criptografados fortemente, com base no algoritmo MD5.

PEGWIT: Programa utilizado para autenticação e criptografia pública.

Ghostscript : Interpretador da linguagem *Postscript*, a aplicação envolvida no *benchmark* é o *gs*, a qual não possui interface gráfica e efetua apenas processamento de entrada e saída em arquivos.

Mesa: Biblioteca para manipulação de gráficos em três dimensões, baseada na biblioteca proprietária *OpenGL*. São três as aplicações utilizadas: *mipmap*, para executar o mapeamento rápido de texturas, *osdemo* para renderização e *texgen* para geração de texturização.

RASTA: É um programa para reconhecimento de voz, o qual faz uso de diversas técnicas e que se propõe a tratar tanto ruído quanto distorções da voz simultaneamente na tentativa de estabelecer filtros para obtenção de uma melhor qualidade na voz e conseqüentemente um reconhecimento mais aprimorado dos discursos.

EPIC: Um algoritmo experimental para compressão de imagens, tem como característica principal atingir velocidades extremamente altas na compressão de imagens sem necessitar de *hardware* dedicado para operações de ponto flutuante.

ADPCM: Considerado um dos mais simples e antigas formas de codificação e decodificação de audio, é utilizado no MediaBench devido sua aceitação e popularidade.

Uma vez selecionados os *benchmarks* foi chegado o momento de comparar o MediaBench com os dados do SPEC, levando em conta arquiteturas embarcadas e voltadas para multimídia e telecomunicações.

Observou-se que o SPEC não tem como ponto forte a execução intensiva de *cache* de instruções, com isso, foi possível perceber que o MediaBench faz melhor uso desses de

caches de instruções nas arquiteturas embarcadas. Mais adiante, em relação a *cache* de dados, foi constatado que o MediaBench é mais efetivo para leituras enquanto o SPEC tem melhor desempenho para escritas. Isso se deve em parte à particularidade dos testes de gravação do MediaBench, uma vez que diversos deles envolvem *streaming* de dados o que por si só não viabiliza acertos de *cache*.

Por fim, em relação a sintetização de sistemas, o MediaBench foi utilizado para realizar um experimento em um sistema em um único *chip* (*system-on-a-chip*), de forma a avaliar sua utilidade ao invés de propor uma nova abordagem para o problema. O experimento consiste em otimizar a arquitetura de *cache* de um sistema RISC com o objetivo de maximizar os acertos do *cache* levando em conta o custo do hardware. Com isso, foi possível observar que uma vez que o SPEC não faz uso efetivo de *cache* em sistemas embarcados, no estudo, esse *benchmark* sugeriria um *chip* de *cache* com tamanho 18% maior que o proposto pelo MediaBench.

Com o apresentado acima, observou-se que o MediaBench é uma alternativa viável a *benchmaks* para plataformas de uso geral (i.e. computadores pessoais de mesa) principalmente pois foca em arquiteturas anteriormente consideradas marginais.

Mais recentemente, **Guthaus [15]** relata que programas para aferição de performance, *benchmarks*, são utilizados para caracterizar o desempenho de um computador de uso geral por meio de um conjunto de aplicativos e dados divididos em categorias, como por exemplo, números inteiros e de ponto flutuante. Dentre estes, um dos mais adotados pela indústria é o CPU *benchmark* da *Standard Performance Evaluation*, embora não seja o foco deste trabalho, pois entre outros aspectos, se destina apenas à plataformas de computadores pessoais de mesa e servidores - uma pequena fração dos microprocessadores existentes mundo afora, onde a grande maioria é voltada para aplicações embarcadas (*embedded applications*), os quais respondem por mais da metade de todo o lucro da fabricação de microprocessadores [19].

Voltando-se para esse mercado, foi desenvolvido um *benchmark* pelo EDN Embedded Microprocessor Benchmark Consortium (EEMBC) [7] na tentativa de caracterizar o difícil domínio de aplicações para os microprocessadores embarcados, o qual pode variar desde simples sistemas de sensores até telefones celular com poder de processamento equivalente a computadores de mesa [10]. O próprio consórcio reconheceu a dificuldade em criar um único conjunto de programas que caracterizasse todo o domínio de aplicações embarcadas

Industrial/ Automotivo	**Disp. Consumidores**	**Escritório**	**Redes**	**Segurança**	**Telecom.**
basicmath	jpeg	ghostscript	dijkstra	blowfish enc.	CRC32
bitcount	lame	ispell	patricia	blowfish dec.	FFT
qsort	mad	rsynth	(CRC32)	pgp sign	IFFT
susan (edges)	tiff2bw	sphinx	(sha)	pgp verify	ADPCM enc.
susan (corners)	tiff2rgba	stringsearch	(blowfish)	rijndael enc.	ADPCM dec.
susan (smoothing)	tiffdither			rijndael dec.	GSM enc.
	tiffmedian			sha	GSM dec.
	typeset				

Tabela 4.1: *Benchmarks* do *MiBench*

e tendo em vista tal obstáculo, separou em cinco diferentes grupos de aplicações, cada um voltado para um mercado consumidor diferente.

Devido ao alto custo de associação junto ao consórcio EEMBC, seus *benchmark* são de difícil acesso ao público em geral, foi criado o MiBench [15] que se assemelha ao EEMBC, sendo que ao invés de cinco categorias de aplicações, são propostas seis, cada uma focando uma determinada área de adoção de sistemas embarcados: controle industrial e automotivo, dispositivos para consumidores, automação de escritório, redes de computadores, segurança e telecomunicações.

A seguir, serão brevemente descritos os *benchmarks* do MiBench, conforme apresentado na Tabela 4.1:

1. Controle Industrial e Automotivo: Tem como objetivo demonstrar o uso de processadores em sistemas de controle embarcados. Tais processadores requerem performance em computações de matemática básica, manipulação de bits, entrada e saída de dados e simples organizações de dados. Suas aplicações principais são em sistemas controladores de *air bags* para automóveis e monitoramento de performance e sensores para motores.

   Os testes para caracterizar tais ambientes envolvem:

   (a) Testes de matemática básica (*basicmath*) como o cálculo de raiz quadrada de um número inteiro, pois normalmente os processadores não possuem hardware aritmético dedicado;

(b) Testes para contagem da quantidade de bits (*bitcount*) em uma matriz de números inteiros;

(c) Execução de um algoritmo de classificação (*qsort*) para ordenação de um vetor de *strings*;

(d) Execução de um programa (*susan*) para reconhecimento de formas geométricas em imagens.

2. Dispositivos para Consumidores: Representam a grande maioria dos dispositivos embarcados que se tornaram populares nos últimos anos como *scanners* de imagem, câmeras digitais e computadores de bolso (PDA - *Personal Digital Assistant* ou Assistentes Digitais Pessoais). As aplicações que são executadas nesses dispositivos são focadas principalmente em multimídia, como manipulação de imagens JPEG, codificação/decodificação de músicas MP3, formatação de arquivos HTML e jogos.

   Os testes de tais dispositivos incluem:

   (a) Codificação/Decodificação de pequenos e grandes arquivos JPEG (*jpeg enc./dec.*), pois tais arquivos são freqüentemente encontrados em documentos texto;

   (b) Conversão de cores e manipulação de imagens de tamanho variável no formato TIFF (*tiff2bw, tiff2rgba, tiffdither e tiffmedian*);

   (c) Codificação/decodificação de pequenos e grandes arquivos MP1, MP2 e MP3 (*lame e mad*);

   (d) Formatação de documentos HTML (*typeset*) os quais são principalmente aplicados a navegadores web de PDA's.

3. Automação de Escritório: São constituídos principalmente por algoritmos de manipulação de texto encontrados em equipamentos para escritórios como máquinas de fax, impressoras e processadores de texto. Os PDA's citados na seção anterior também fazem grande uso de ferramentas de texto, portanto também podem ser caracterizados nessa categoria.

   Entre os testes de *benchmark* encontram-se:

   (a) Execução de algoritmo (*ghostscript*), um interpretador da linguagem postscript cada vez mais encontrado em impressoras;

(b) Busca de palavras (*stringsearch*) em documentos texto;

(c) Dicionário ortográfico com sugestão de correção e suporte a línguas diferentes do Inglês (*ispell*);

(d) Programa para sintetização de voz (*rsynth e sphinx*) a partir de um dado texto.

4. Redes de Computadores: São representados por equipamentos como *switches* e roteadores, onde o seu uso envolve o cálculo de menor caminho entre dois *hosts*, localização de informações em estruturas de dados com árvores e tabelas além de processamento e entrada e saída de dados.

   Alguns *benchmarks*, dada a sua relevância para o ambiente de redes são reutilizados de outras áreas, os quais são o *CRC32*, o *sha* e o *blowfish*. Especificamente, os algoritmos utilizados são:

   (a) Determinação de menor caminho entre dois vértices de um grafo que representa uma grande matriz de adjacências (algoritmo *diskstra*) ;

   (b) Criação/consulta em uma estrutura de dados (*patricia*) para armazenamento de tabelas de roteamento em aplicações de rede, alimentada com o tráfego de endereços IP de um servidor *web* de alto tráfego;

5. Segurança: Com o advento da internet, a segurança de dados têm ganho destaque e se tornado de grande importância para aplicações que envolvam o comércio on-line, portanto uma categoria foi dedicada a tais periféricos. Entre as principais aplicações encontram-se criptografia e decriptografia de dados e *hashing*.

   Os *benchmarks* compreendem:

   (a) Criptografia e decriptografia de dados usando chaves de 32 a 448 bits aplicados em criptografias domésticas e para exportação do governo americano (*blowfish*);

   (b) Execução de funções de *hashing* MD4 e MD5 pelo *sha*;

   (c) Testes envolvendo criação de assinatura e verificação de chaves PGP (*Pretty Good Privacy*) que permite comunicação segura entre pessoas de posse de assinaturas digitais e chaves públicas RSA;

   (d) Cifragem de dados com chaves de 128, 192 ou 256 bits (*rijndael*).

6. Telecomunicações: Dada a explosão no uso de dispositivos para comunicação sem fio encontrados no mercado consumidor, e que se conectem em rede, faz-se necessário a criação de uma categoria a parte. Essa categoria engloba principalmente equipamentos como que façam uso de algoritmos que envolvam codificação de voz, análise de freqüências e validação de *checksums*.

   Entre os testes de performance encontram-se:

   (a) A transformação e transformação inversa Fournier de matrizes de dados, normalmente utilizadas em processamento digital de sinais *(FFT/IFFT)*;

   (b) Codificação/decodificação de voz no padrão de transmissão celular GSM *(Global Standard for Mobile Communication)* tendo como entrada trechos de conversão de tamanho variado *(gsm enc./dec.)*;

   (c) Codificação/decodificação de voz no padrão ADPCM *(Adaptative Pulse Code Modulation)* o qual é uma variação do conhecido PCM *(Pulse Code Modulation)* que usa como entrada um trecho de conversação de 16 bits e o comprime num padrão de 4 bits *(adpcm enc./dec.)*;

   (d) Checagem em 32 bits de CRC *(Cyclic Redudancy Check)* em um arquivo, uma vez que tal validação é freqüentemente utilizada para detectar erros em transmissão de dados. Como entrada para tal checagem é utilizado o arquivo de voz do ADPCM *benchmark*.

Caracterização: De posse dos *benchmarks* a serem executados, é possível caracterizá-los de duas formas.

Quanto aos grupos de instruções:

(a) de controle (ramificações condicionais ou não),

(b) inteiras,

(c) ponto flutuante,

(d) de acesso a memória (operações de recuperação e armazenamento).

Quanto aos tipos de aplicações:

(a) controle intensivo: relativo a existência de uma grande quantidade de laços e ramificações encontradas nos programas;

(b) computação intensiva: relacionado a uma grande quantidade de operações aritméticas e de ALU;

(c) I/O intensivo: relativo a operações de transferências de dados.

Os *Benchmarks* e a *PattLib*: É possível criar uma distinção entre as características do MiBench [15] e os dados que são armazenados na *PattLib*, onde o primeiro se preocupa em definir grandes grupos de aplicações e suas características, como as relacionadas ao tipo de instruções encontradas em seus programas ou ainda relacionada a quantidade de vezes que determinado grupo de instruções é executado (alta quantidade de operações de entrada e saída). Já a biblioteca *PattLib*, preocupa-se em permitir ao usuário que esse infira sobre os *benchmarks* e possa extrair destes informações referentes, por exemplo, à existência de determinado grupo de instruções encontradas nas aplicações. Dessa forma, é possível determinar se um programa (ou *benchmark*) possui instruções de aritmética básica simples (como adição ou subtração), sem considerar operações de multiplicação ou módulo (as quais são de custo mais alto de execução) enquanto no tocante do MiBench [15] é possível apenas classificar os programas quanto a suas características de execução.

Nesse trabalho pretendemos obter informações mais profundas sobre grupos de instruções (padrões) de forma a facilitar a análise e seleção dos padrões a serem implementados em *hardware*. Toda essa análise será baseada nos padrões armazenados na *PattLib* extraídos de programas do MiBench e MediaBench.

4.2 Metodologia de Análise

Para viabilizar a geração dos gráficos que possibilitam a análise dos *benchmarks*, foi necessário que uma série de passos fossem seguidos, ao ver:

1. Escolha dos arquivos de padrões: O projeto *Chameleon* gera quatro tipos de arquivos de padrões para cada programa analisado:

   raw: Com os dados originais de *profiling* do GCC [5];

fixed: Com as referências às instruções corrigidas - não é interessante para análise por possuir apenas uma instrução;

grow: Com as referências às instruções corrigidas e que representam padrões expandidos - através das relações de dependências, podem ultrapassar os limites do bloco básico;

grow-loops: Representam o *loop* principal da aplicação - não é interessante pois pode descartar outros trechos de programas que também seja executados bastante.

Foi decidido pela utilização dos arquivos gerados na forma *grow* pois apresentam uma boa granularidade para análise, execução e coleta dos dados;

2. Execução da ferramenta por meio de um *script bash* para iterar por todos os arquivos de padrões e armazenamento de suas saídas num determinado diretório em disco;

3. Importação para base de dados auxiliar: Com o intuito de facilitar a geração dos dados consolidados nos quais os gráficos são baseados, as informações geradas pela biblioteca foram importadas para um banco de dados *MS Access* onde é possível a execução comandos em linguagem *SQL (Structured Query Language)* utilizando as facilidades de agrupamento e ordenação providas pela linguagem.

4. Criação de consultas *SQL*: Com os dados importados, consultas *SQL* foram geradas para cada um dos gráficos desejados;

5. Geração dos gráficos: O resultado dos comandos SQL foram importados para planilhas em formato *Excel* para construção dos gráficos.

Os passos acima descritos constituem uma das abordagens possíveis para geração dos dados estatísticos a partir das informações da *PattLib*. Entre os próximos trabalhos previstos para a ferramenta, está a automatização de tais passos tornando o processo menos sujeito a erros e mais integrado.

4.3 *Análise do MediaBench*

Nessa seção serão apresentados os histogramas de entradas e saídas por padrão, quantidade de operandos e operadores por padrão além dos operadores utilizados com maior freqüência levando-se como fonte de dados de entrada o *benchmark MediaBench*

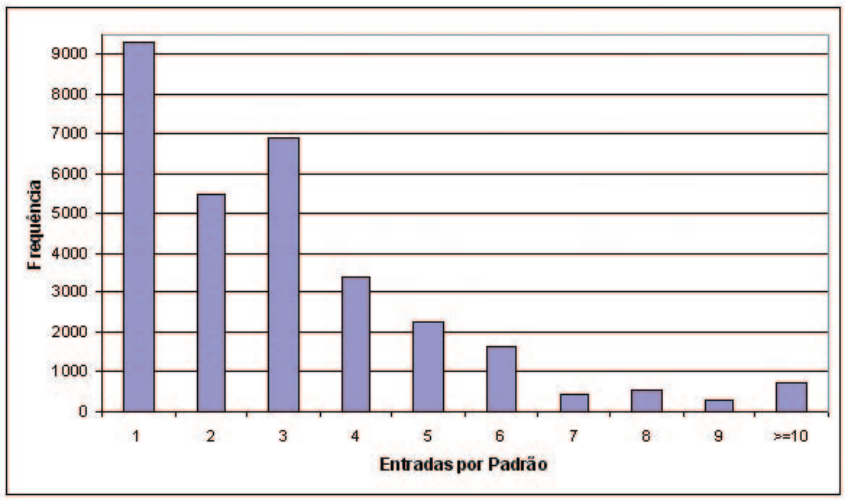

Figura 4.1: Entradas por Padrão no *MediaBench.*

Figura 4.2: Saídas por Padrão no *MediaBench.*

A Figura 4.1 apresenta a quantidade de padrões com um determinado número de entradas do *MediaBench.* Pode-se notar que há uma maior concentração de padrões com

uma, duas ou três entradas - uma característica das da arquitetura RISC [9] onde as instruções possuem normalmente duas entradas e uma saída. Especificamente, observa-se que existem pouco mais de nove mil padrões com uma entrada, acima de cinco mil padrões com duas entradas e aproximadamente sete mil padrões com três entradas, conforme apresentado nas três primeiras colunas do gráfico.

Pode-se valer da mesma interpretação para a Figura 4.2, onde a quase totalidade dos padrões possuem apenas uma saída: aproximadamente dezoito mil padrões contém uma saída, enquanto padrões com duas saídas não chegam a totalizar quatro mil unidades e padrões com três saídas têm representatividade quase zero.

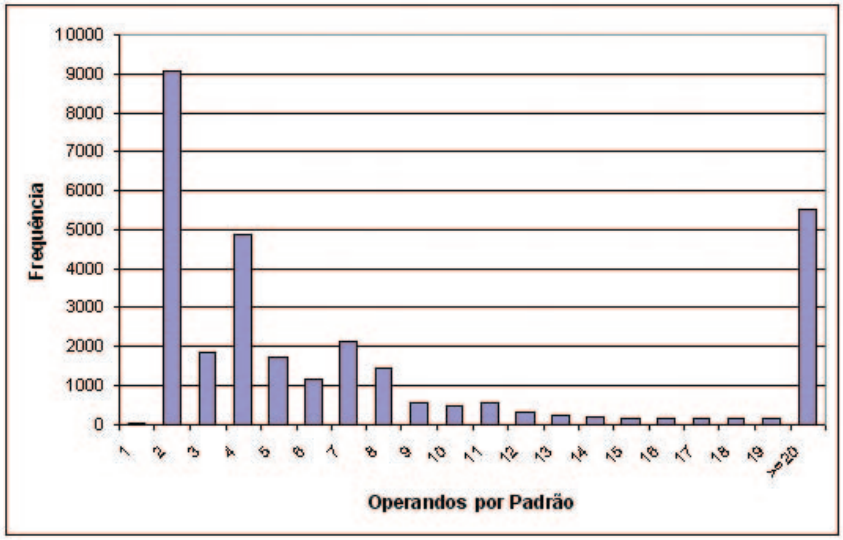

Figura 4.3: Quantidade de Operandos por Padrão no *MediaBench.*

Em relação a quantidade de operandos por padrão no *MediaBench*, a Figura 4.3 apresenta uma distribuição bimodal onde a maioria dos padrões se concentra ora com uma baixa quantidade de operandos, ora com uma acentuada quantidade de operandos - na segunda coluna da figura: acima de nove mil padrões com duas entradas enquanto na última coluna observa-se que aproximadamente cinco mil padrões possuem vinte ou mais entradas.

Do mesmo modo, a Figura 4.4 a qual representa a quantidade de operadores por padrão do mesmo *benchmark*, também apresenta uma distribuição modal: doze mil padrões com apenas um operador (primeira coluna) e acima de quatro mil padrões com vinte ou mais operadores (última coluna).

Em ambos os casos anteriores, é importante ressaltar que a relação bimodal existe não a simples existência de um alto número de operadores e operandos, mas sim ao fato de que o compilador GCC [5] cria operadores que "sujam"os operandos - de forma a indicar que um determinado registrador será usado posteriormente no padrão.

Nesse momento, podemos caracterizar que os padrões do *MediaBench* se comportam tipicamente como instruções da arquitetura RISC (o que de fato são!), onde a maioria dos padrões possui uma baixa quantidade de entradas (mais de nove mil padrões com uma entrada), uma pequena quantidade de saídas (a maioria dos padrões possui apenas uma saída: perto de dezoito mil) e um reduzido número de operadores por padrão (dezoito mil padrões possuem apenas um operador).

Finalmente, a Figura 4.5 exibe os operadores que ocorrem com maior freqüência no *MediaBench*: primeiramente o operador de soma com sinal (SS_PLUS) seguido pela instrução de armazenamento em registrador (ST) e pela instrução de extração de sinal (SIGN_EXTRACT). As instruções com baixa representatividade foram agrupadas na categoria *Outros* pois se apresentadas individualmente não seria possível distinguí-las no histograma.

É importante ressaltar que os operadores USE e CONST não fizeram parte de nenhuma estatística relacionada a quantidade de operadores (tanto no *MediaBench* quanto no *MiBench*), pois o primeiro representa apenas que um registrador será usado, enquanto o segundo caracteriza uma constante literal de um programa.

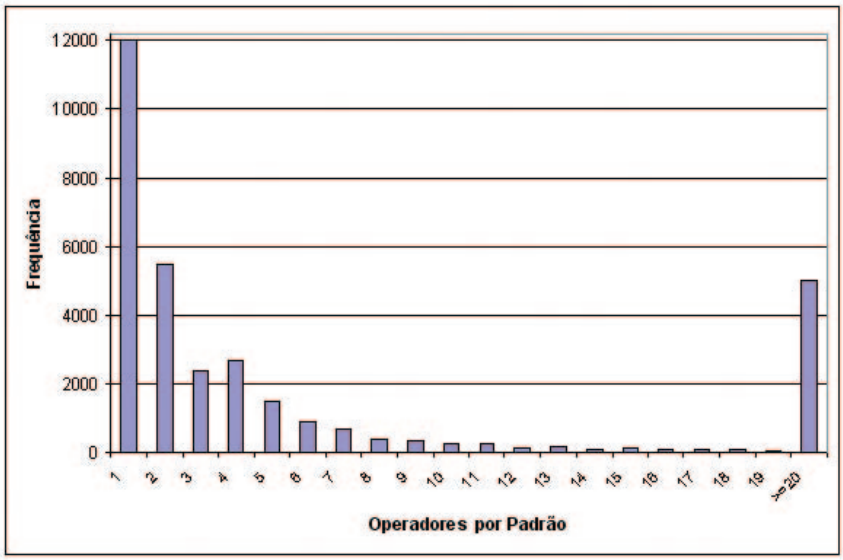

Figura 4.4: Quantidade de Operadores por Padrão no *MediaBench.*

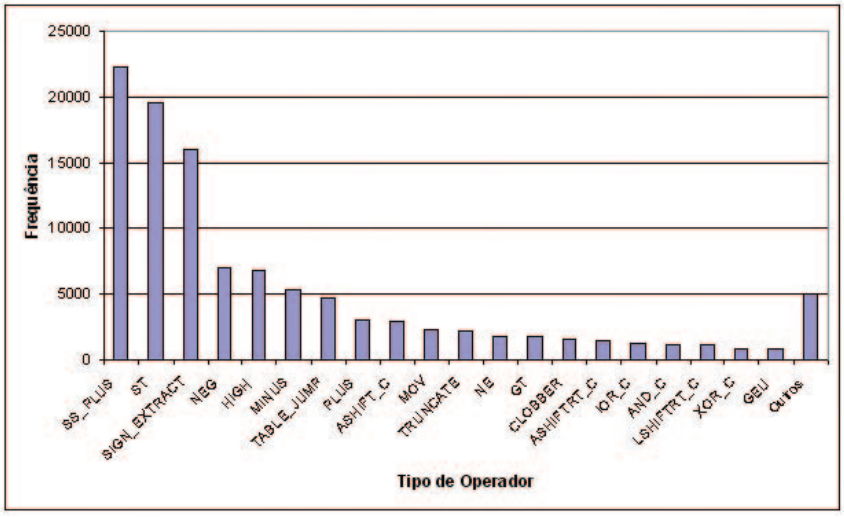

Figura 4.5: Tipos de Operadores por Padrão no *MediaBench.*

4.4 Análise do *MiBench*

Nessa seção serão apresentados os histogramas de entradas e saídas por padrão, quantidade de operandos e operadores por padrão além dos operadores utilizados com maior freqüência levando-se como fonte de dados de entrada o *benchmark MiBench*. Os dados estão agrupados de acordo com as diversas categorias do *benchmark* (*automotive, consumer, network, office, security e telecomm*), de forma a tornar possível uma análise mais detalhada dos mesmos.

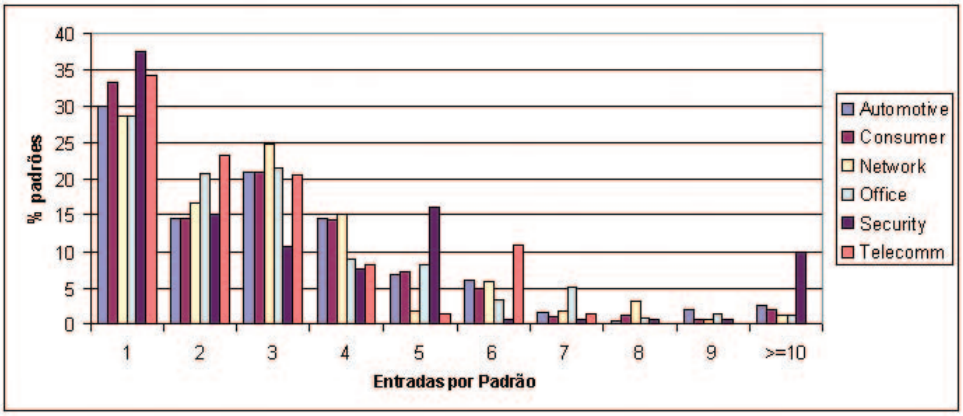

Figura 4.6: Entradas por Padrão no *MiBench*.

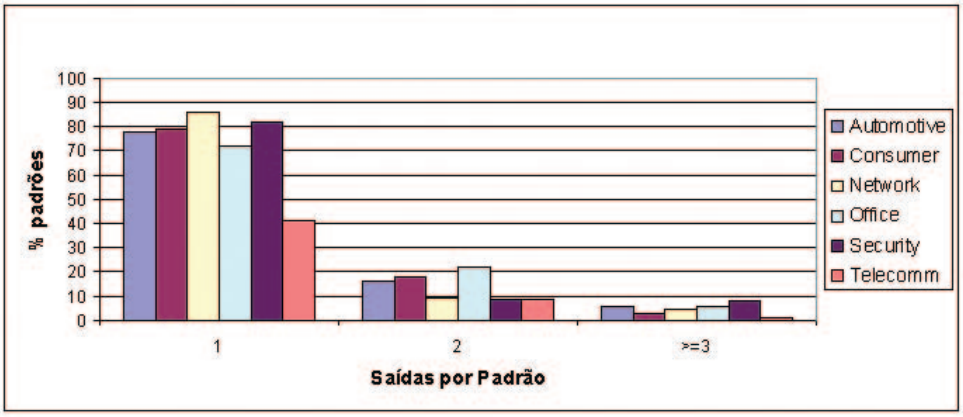

Figura 4.7: Saídas por Padrão no *MiBench*.

Tanto na Figura 4.6 quanto na Figura 4.7 observa-se uma tendência dos padrões em todas as categorias do *benchmark* em concentrar uma maior quantidade de padrões com uma ou duas entradas e apenas uma saída, da mesma forma que os padrões do

MediaBench apresentados anteriormente: padrões pequenos, arquitetura RISC. Como exemplo, a categoria *Security* apresenta acima de trinta e cinco por cento de todos seus padrões com apenas uma entrada (quinta coluna da Figura 4.6) e mais de oitenta por cento de seus padrões possuem uma única saída (quinta coluna, Figura 4.7).

Ambos os gráficos estão dispostos em escala percentual pois devido a grande diferença de quantidade de padrões entre as diversas categorias do *benchmark*, ficaria impossível agrupá-los linearmente em um único histograma.

Nas figuras seguintes serão apresentadas a quantidade de operandos por padrão; observa-se que todos têm característica semelhante aos padrões do *MediaBench*: uma distribuição bimodal.

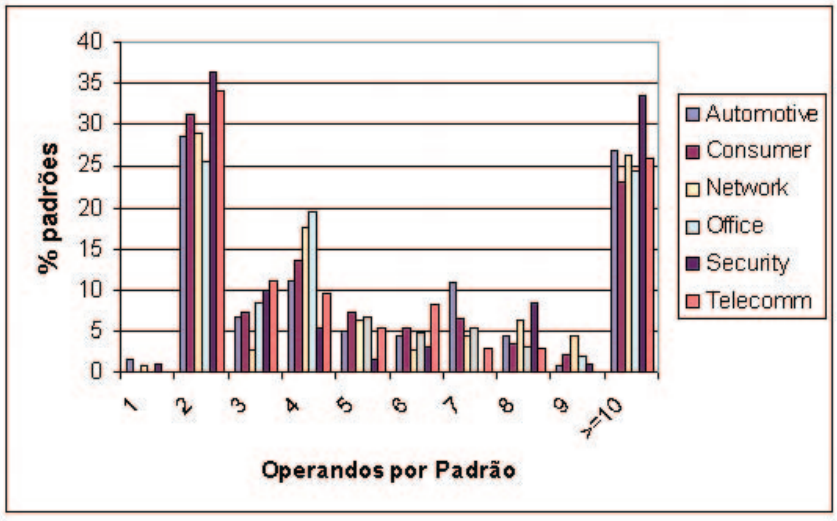

Figura 4.8: Operandos por Padrão no *MiBench.*

Especificamente, a Figura 4.8 apresenta uma concentração nos padrões com dois operadores (todas categorias com pelo menos vinte e cinco por cento dos padrões contendo dois operadores) além de uma concentração nos padrões com mais de dez operadores por padrão (também próximo a vinte e cinco por cento do total dos padrões). O truncamento nessa última faixa fez-se necessário em virtude da grande quantidade de padrões com quantidade nula de operandos.

Da mesma forma, a Figura 4.9 apresenta uma distribuição bimodal, com a maioria dos padrões concentrando-se com apenas um operador por padrão (i.e. quase cinqüenta por cento dos padrões da categoria *Security* possuem apenas um operador) e apresen-

tando outra representativa concentração nos padrões com mais de dez operadores (i.e. a categoria *Network* concentra quase vinte por cento de seus padrões com dez ou mais operadores).

Finalmente, analisando os tipos de operadores mais encontrados na Figura 4.10, Figura 4.11, Figura 4.12, Figura 4.13, Figura 4.14, Figura 4.15, observa-se que os operadores de soma com sinal (SS_PLUS) e armazenamento em registrador (ST - *store*) aparecem como primeiro e segundas operações mais executadas (alternando sua ordem em algumas das figuras), exceto pela Figura 4.13 onde como segundo operador mais utilizado têm-se a operação de HIGH.

Além disso, percebe-se uma predominância dos operadores aritméticos em todas as classes do *MiBench* com concentração nas operações de soma e subtração (SS_PLUS e PLUS e MINUS) bem como de operadores lógicos como negação (NEG) e de comparação entre valores (EQ, GT).

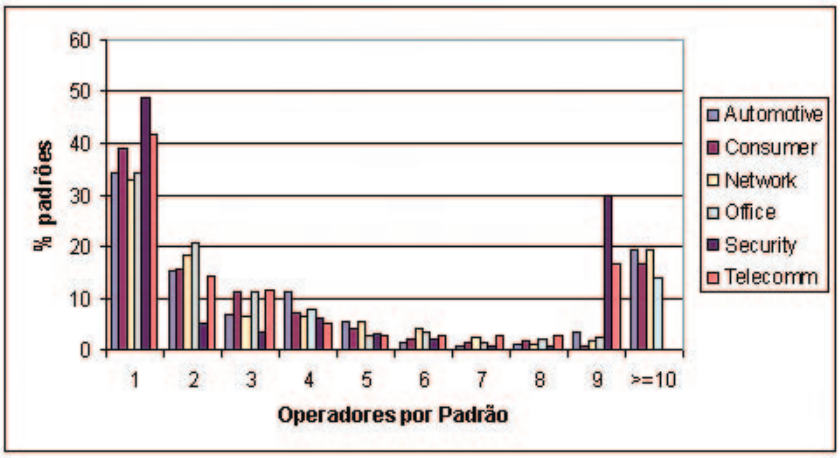

Figura 4.9: Operandos por Padrão no *MiBench.*

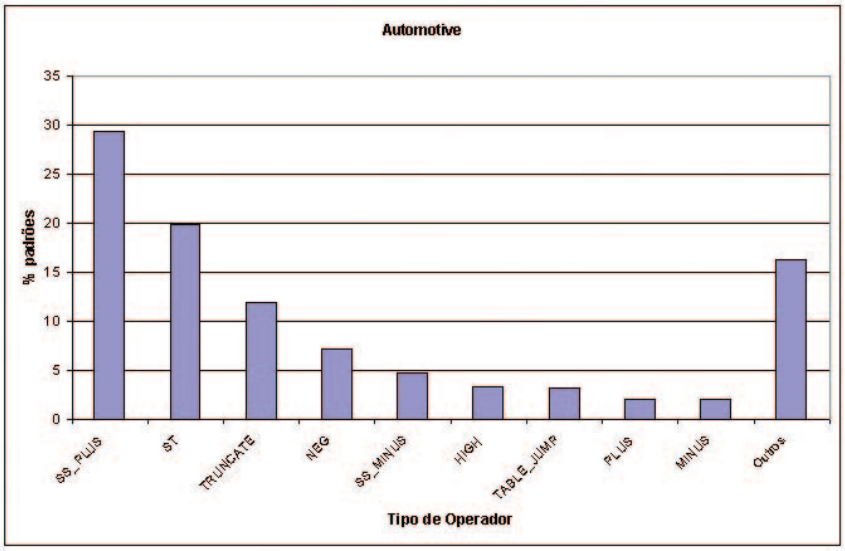

Figura 4.10: Tipos de Operadores por Padrão no *MiBench.*

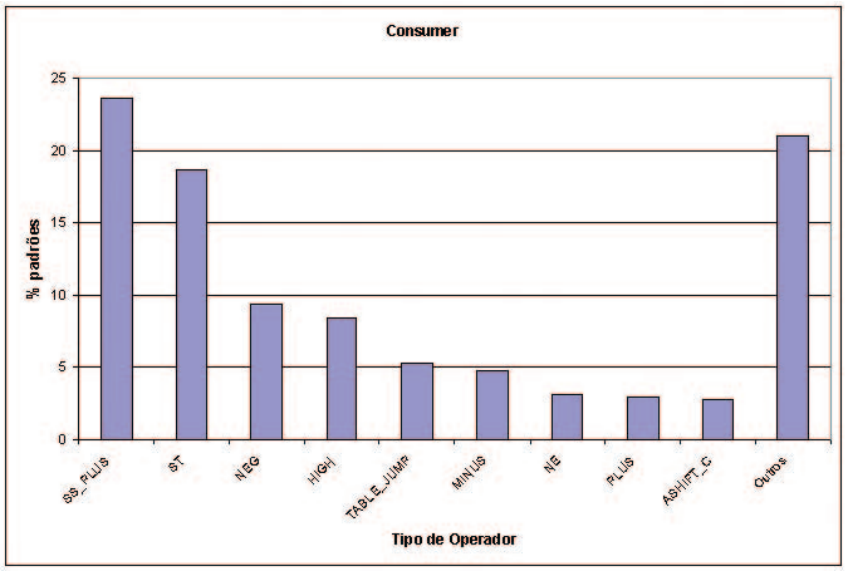

Figura 4.11: Tipos de Operadores por Padrão no *MiBench.*

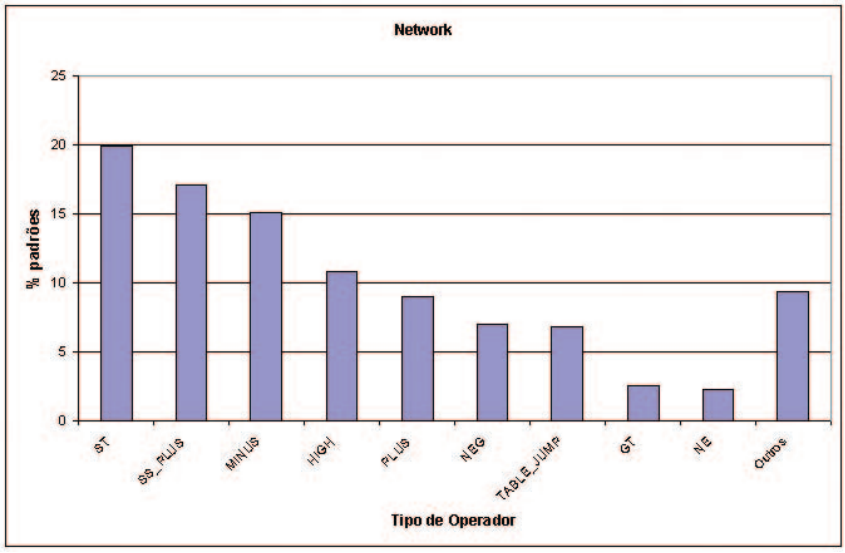

Figura 4.12: Tipos de Operadores por Padrão no *MiBench - Network.*

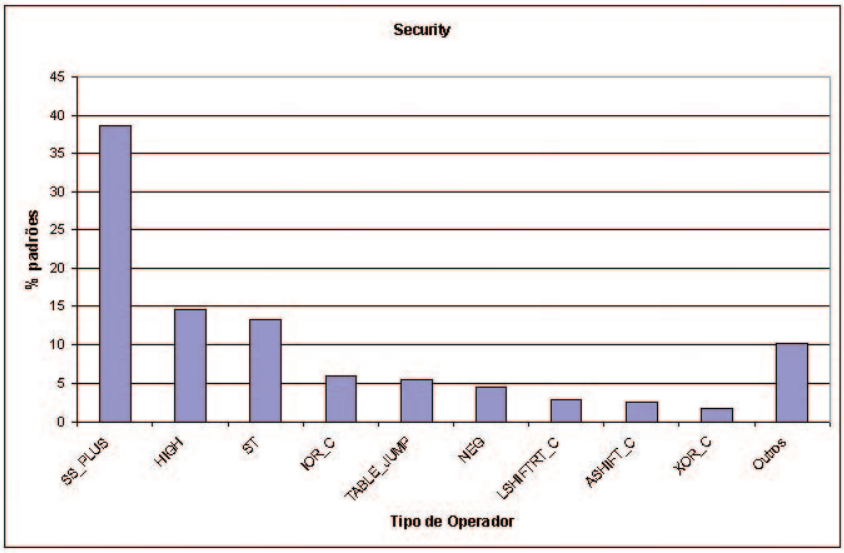

Figura 4.13: Tipos de Operadores por Padrão no *MiBench - Security.*

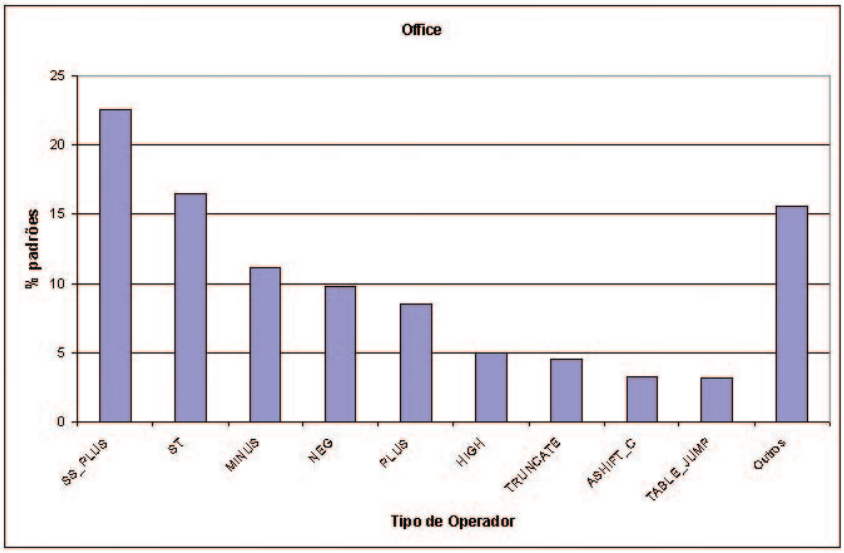

Figura 4.14: Tipos de Operadores por Padrão no *MiBench - Office.*

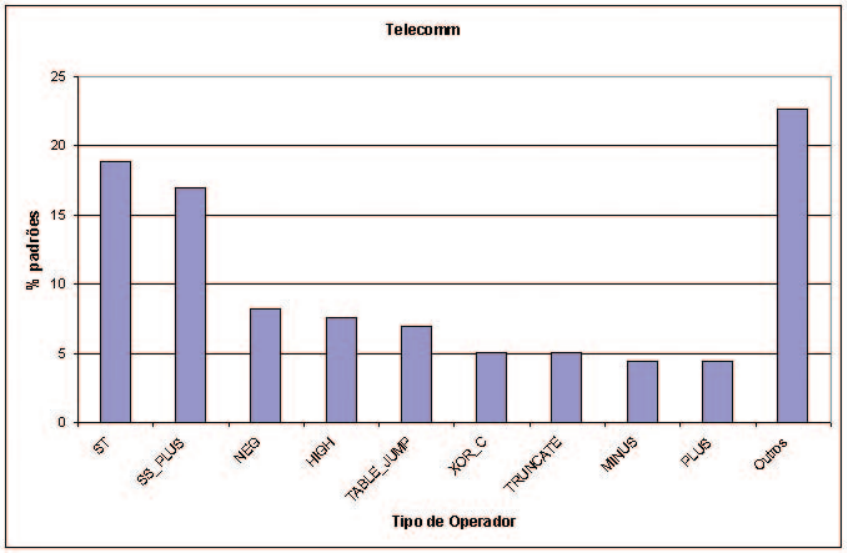

Figura 4.15: Tipos de Operadores por Padrão no *MiBench - Telecomm.*

4.5 Aspectos comuns ao *MediaBench* e *MiBench*

Com base nos gráficos apresentados nesse capítulo, podemos observar que os *benchmarks* apresentam diversas semelhanças entre si, a citar:

- Baixa quantidade de entradas e saídas por padrão;
- Baixo número de entradas e de saídas;
- Maior freqüência de operadores aritméticos.

Esses aspectos são de grande valia para os projetistas, pois os ajudam na tomada de decisões, onde poderiam optar por implementar em *hardware* primeiramente as instruções pequenas (conforme as características apresentadas acima) e que possuam operadores aritméticos; tal informação, sem o auxilio da biblioteca, requereria um estudo manual e trabalhoso dos padrões, um a um, para que um resultado semelhante pudesse ser observado.

Capítulo 5

Conclusão e Trabalhos Futuros

O resultado central desse trabalho foi o desenvolvimento da biblioteca para análise e filtragem de padrões (*Pattern Matcher* e trechos de programas voltada a se encaixar como um módulo do projeto *Chameleon*.

Foi introduzido o conceito de ferramentas *pattern-to-pattern* onde bibliotecas ou programas recebem como entrada um padrão, executam uma determinada tarefa e geram como saída um outro padrão pós-processado o que viabiliza uma interface padrão entre os diversos módulos da plataforma *Chameleon*.

Com a biblioteca é possível filtrar uma base de dados com milhares de padrões de forma a obter aqueles úteis e candidatos a se tornarem instruções a serem implementadas em *hardware*.

Viu-se que existem diversos filtros que podem ser aplicados de forma programática - onde o próprio módulo do *Chameleon* responsável pela geração dos filtros pode invocar a biblioteca possibilitando uma filtragem inicial dos padrões que não possuem nenhuma representatividade, para descartar padrões automaticamente - fato esse que reduz o tempo de processamento e economiza espaço em disco rígido. Esses filtros podem ser aplicados individualmente ou agrupados em forma de uma árvore lógica construída anteriormente a execução da biblioteca. Os filtros também podem ser executados por linha de comando no caso dos padrões já existirem em disco. Nessa opção é possível apenas a execução de um filtro por vez; caso deseje-se a execução de mais de um filtro, basta apenas executar o próximo filtro sobre a saída do filtro anterior conforme sugere as ferramentas *patt-to-patt*.

Foram desenvolvidas ainda ferramentas estatísticas que possibilitam um melhor entendimento dos padrões existentes. Com os dados estatísticos gerados foi possível fazer

uma análise dos benchmarks `MediaBench e Mibench` sob um ponto de vista nunca anteriormente feito, onde aspectos como quantidade de entradas e saídas, quantidade de operandos e operadores assim como os operadores mais utilizados nos benchmarks foram levantados e gráficos elaborados com bases nos dados pela biblioteca.

5.1 Trabalhos Futuros

Embora a biblioteca já esteja sendo utilizada operacionalmente na plataforma *Chameleon*, uma série de funcionalidades ainda podem ser implementadas com o objetivo de tornar seu uso mais amigável e incrementar suas funcionalidades, a ver:

- Uma função para marcação de operadores onde além de se calcular a quantidade de entradas e saídas de um padrão, seus operandos seriam marcados como sendo de entrada ou de saída. Tal informação seria armazenada diretamente na estruturada de dados do padrão e em seguida persistida em disco para uso posterior por outras ferramentas da biblioteca;

- Suporte a múltiplos filtros quando da execução da biblioteca por linha de comando. Um arquivo seria lido com informações dos filtros o qual conteria os critérios a serem aplicados nesses mesmos filtros; para isso uma espécie de gramática seria desenvolvida onde o usuário usaria essa linguagem para descrever suas necessidades. Tais arquivos de critérios poderiam ser armazenados criando assim uma biblioteca de filtros pré-definidos que poderiam ser executados por qualquer pessoa;

- Geração da saída dos dados estatítisticos no formato *GnuPlot* o que tornaria mais automatizada a geração dos gráficos, pois atualmente os dados são gerados em formato CSV (*Comma Separated Values* - ou Valores Separados por Vírgulas);

- Funcionalidade para iterar sobre mais de um padrão pois atualmente é possível apenas aplicar um filtro ou gerar dados estatíticos para um único padrão por vez; a forma de contornar essa situação é a utilização de *scripts* para navegação por diretórios e invocação da ferramenta para cada um dos arquivos de padrões existentes.

Apêndice A

Dados coletados do *MediaBench*

Entradas por Padrão	**Frequência**
1	9325
2	5473
3	6922
4	3402
5	2260
6	1639
7	450
8	547
9	278
>=10	718

Tabela A.1: Entradas por Padrão no *MediaBench*

Saídas por Padrão	**Frequência**
1	17566
2	3765
3	482
4	198
>=5	159

Tabela A.2: Saídas por Padrão no *MediaBench*

Operandos por Padrão	**Frequência**
1	37
2	9079
3	1870
4	4896
5	1725
6	1130
7	2141
8	1434
9	580
10	474
11	577
12	319
13	255
14	225
15	144
16	144
17	137
18	155
19	133
>=20	5528

Tabela A.3: Operandos por Padrão no *MediaBench*

Operadores por Padrão	**Frequência**
1	12061
2	5468
3	2380
4	2671
5	1483
6	920
7	671
8	372
9	340
10	247
11	232
12	126
13	135
14	92
15	114
16	83
17	78
18	58
19	45
>=20	5013

Tabela A.4: Operadores por Padrão no *MediaBench*

Operador	**Quantidade**
SS_PLUS	22273
ST	19565
SIGN_EXTRACT	16039
NEG	7068
HIGH	6760
MINUS	5277
TABLE_JUMP	4682
PLUS	3051
ASHIFT_C	2975
MOV	2344
TRUNCATE	2268
NE	1800
GT	1796
CLOBBER	1575
ASHIFTRT_C	1456
IOR_C	1168
AND_C	1103
LSHIFTRT_C	1067
XOR_C	870
GEU	847
Outros	5032

Tabela A.5: Tipos de Operador por Padrão no *MediaBench*

Apêndice B

Dados coletados do *MiBench*

Entradas	Automotive	Consumer	Network	Office	Security	Telecomm
1	152	5014	90	101	49	25
2	74	2196	52	73	20	17
3	106	3152	78	76	14	15
4	74	2175	48	32	10	6
5	35	1098	6	29	21	1
6	31	755	18	12	1	8
7	9	168	6	18	1	1
8	3	170	10	3	1	0
9	10	93	2	5	1	0
>=10	13	311	4	4	13	0

Tabela B.1: Entradas por Padrão no *MiBench*

Saídas	Automotive	Consumer	Network	Office	Security	Telecomm
1	269	8548	184	195	77	41
2	57	1936	20	60	8	9
>=3	19	331	10	16	9	1

Tabela B.2: Saídas por Padrão no *MiBench*

Operandos	Automotive	Consumer	Network	Office	Security	Telecomm
1	8	8	2	0	1	0
2	149	4728	92	90	48	25
3	35	1077	8	30	13	8
4	58	2051	56	69	7	7
5	26	1073	20	24	2	4
6	23	834	8	17	4	6
7	56	994	14	19	0	2
8	23	525	20	11	11	2
9	3	329	14	7	1	0
>=10	140	3513	84	86	44	19

Tabela B.3: Operandos por Padrão no *MiBench*

Operadores	**Automotive**	**Consumer**	**Network**	**Office**	**Security**	**Telecomm**
1	182	6235	108	126	66	32
2	82	2505	60	77	7	11
3	37	1780	22	42	5	9
4	61	1170	22	29	8	4
5	29	682	18	10	4	2
6	8	331	14	12	3	2
7	3	231	8	6	1	2
8	6	277	4	8	1	2
9	19	132	6	9	0	0
>=10	104	2622	64	51	40	13

Tabela B.4: Operadores por Padrão no *MiBench*

Operador	**Quantidade**
SS_PLUS	657
ST	443
TRUNCATE	266
NEG	159
SS_MINUS	106
HIGH	73
TABLE_JUMP	72
PLUS	45
MINUS	45
Outros	363

Tabela B.5: Tipos de Operador por Padrão no *MiBench - Automotive*

Operador	Quantidade
SS_PLUS	9044
ST	7177
NEG	3612
HIGH	3229
TABLE_JUMP	2044
MINUS	1830
NE	1187
PLUS	1122
ASHIFT_C	1051
Outros	8058

Tabela B.6: Tipos de Operador por Padrão no *MiBench - Consumer*

Operador	Quantidade
ST	158
SS_PLUS	136
MINUS	120
HIGH	86
PLUS	72
NEG	56
TABLE_JUMP	54
GT	20
NE	18
Outros	74

Tabela B.7: Tipos de Operador por Padrão no *MiBench - Network*

Operador	**Quantidade**
SS_PLUS	226
ST	165
MINUS	111
NEG	98
PLUS	85
HIGH	50
TRUNCATE	45
ASHIFT_C	32
TABLE_JUMP	31
Outros	156

Tabela B.8: Tipos de Operador por Padrão no *MiBench - Office*

Operador	**Quantidade**
SS_PLUS	162
HIGH	61
ST	56
IOR_C	25
TABLE_JUMP	23
NEG	19
LSHIFTRT_C	12
ASHIFT_C	11
XOR_C	7
Outros	43

Tabela B.9: Tipos de Operador por Padrão no *MiBench - Security*

Operador	**Quantidade**
ST	30
SS_PLUS	27
NEG	13
HIGH	12
TABLE_JUMP	11
XOR_C	8
TRUNCATE	8
MINUS	7
PLUS	7
Outros	36

Tabela B.10: Tipos de Operador por Padrão no *MiBench - Telecomm*

Referências Bibliográficas

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, 1988.

[2] Mihai Budiu and Seth Copen Goldstein. Detecting and exploiting narrow bitwidth computations. Technical report, Carnegie Mellon University, 1999.

[3] Mihai Budiu and Seth Copen Goldstein. Bitvalue inference: Detecting and exploiting narrow bitwidth computations. Technical report, Carnegie Mellon University, October 2000.

[4] Miodrag Potkonjak Chunho Lee and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communication systems. Technical report, The Departments of Computer Science and Electrical Engineering - The University of California at Los Angeles, 1999.

[5] GCC Gnu Code Compiler. `http://gcc.gnu.org`.

[6] Katherine Compton and Scott Hauck. Configurable computing: A survey of systems and software. Technical report, Nortwestern University, 1999.

[7] EDN Embedded Microprocessor Benchmark Consortium. `http://www.eembc.org`.

[8] Scott Hauck and Anant Agarwal. Software technologies for reconfigurable systems. Technical report, Northwestern University, Dept of ECE, 1996.

[9] John L. Hennessy and David A. Petterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 1996.

[10] Motorola Inc. `http://www.motorola.com`.

[11] Brian W. Kernighan and Dennis Ritchie. *C Programming Language*. Prentice Hall PTR, 1988.

[12] LSC The Computer Systems Laboratory. The pattern library user documentation. `http://ns.lsc.ic.unicamp.br/ chameleon/pattlib/doc/html/index.html`.

[13] W. H. Mangione-Smith and B. L. Hutchings. Configurable computing: The road ahead. in reconfigurable architecture workshop. Technical report, Northwestern University, Dept of ECE, 1997.

[14] Peter Marwedel. Embedded system design. Kluwer Academic Publishers. Novembro de 2003.

[15] ET ALL Matthews R. Guthaus, Jeffrey S. Ringenberg. Mibench, a free, commercially representative embedded benchmark suite. Technical report, Electrical Enginering and Computer Science - The University of Michigan, 1999.

[16] Gnu Plot. `http://www.gnuplot.org`.

[17] L. Pires D. Pandalai S. Kumar and H. Spaanenburg. *Technology for configurable computing system.* In IEEE Symposium on Field-Programmable Custom Computing Machines, 1998.

[18] Mark Stepheson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. Technical report, Massachusetts Institute of Technology, Laboratory of Computer Science, 2000.

[19] J. Turley. Embedded processors by the numbers, embedded systems programming. `http://www.embed-ded.com/1999/9905/9905turley.htm`, May 1999.

[20] Jens Wagner and Rainer Leupers. C compiler for an industrial network processor. Technical report, University of Dortmund, Dept. of Computer Science, 2001.