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

Danillo Moura Santos

API Multiplataforma para Aplicações Multimídia Embarcadas

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

Prof. Dr. Antônio Augusto Medeiros Fröhlich
Orientador

Florianópolis, Abril de 2010

API Multiplataforma para
Aplicações Multimídia Embarcadas

Danillo Moura Santos

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

Prof. Dr. Mário Antônio Ribeiro Dantas
Coordenador do PPGCC

Banca Examinadora

Prof. Dr. Antônio Augusto Medeiros Fröhlich
Orientador

Prof. Dr. Eduardo Augusto Bezerra
FACIN/PUCRS

Prof. Dr. Carlos Aurélio Faria da Rocha
EEL/UFSC

Prof. Dr. Roberto Willrich
INE/UFSC

Aos meus pais.

Agradecimentos

Agradeço ao Professor Guto pela orientação, dedicação e auxílio durante esta jornada. Obrigado por permitir que eu fosse parte do LISHA. Sinto-me honrado por ter feito parte de um grupo tão pequeno em número de pessoas e tão expressivo no reconhecimento.

Agradeço aos amigos do LISHA pelo companheirismo, sugestões, trocas de idéias e pelos momentos de diversão. Sem o apoio de vocês, este trabalho seria impossível. Agradeço por me ajudarem a aprender a aprender, a saber fazer perguntas e a descobrir como e onde buscar as respostas. Obrigado sobretudo ao Hugo Marcondes, ao Roberto de Matos e à Carla Minello, pela ajuda durante o período distante.

Agradeço à minha família por todo o apoio e palavras de conforto nos momentos difíceis. Obrigado por sempre me fazerem acreditar que sou melhor do que penso.

Por último, agradeço à Bárbara Rita Cardoso pelo companheirismo, incentivo e apoio durante os últimos anos.

“Sem você, meu amor,
eu não sou ninguém”
Vinicius de Moraes

Resumo

Diferentes plataformas são utilizadas para o desenvolvimento de aplicações multimídia embarcadas. É comum que compiladores estejam disponíveis para estas plataformas porém, o código gerado a partir de linguagens de alto nível não é capaz de explorar todo o potencial do hardware da plataforma alvo. Para otimizar partes críticas da aplicação, geralmente são implementadas rotinas em linguagem de máquina (*Assembly*). Entretanto, o uso de linguagem *Assembly* na aplicação dificulta sua portabilidade para outras plataformas pois seu código necessita ser reescrito. A migração de uma aplicação para uma nova plataforma, com arquitetura e características de hardware diferentes, requer a reescrita do código da aplicação para a Interface para Programação de Aplicação (API) da arquitetura fornecida pelo fabricante. Este processo requer tempo, atrasando a criação de novos produtos, aumentando assim os custos de desenvolvimento e possivelmente resulta em aplicações ineficientes, que não exploram toda a potencialidade do hardware utilizado.

Este trabalho apresenta a *Embedded Multimedia Cross-Platform API* (EMCA) que tem como objetivo fornecer ao desenvolvedor de aplicações multimídia uma interface independente de hardware para algoritmos de processamento de sinais digitais, facilitando a migração da aplicação para diferentes plataformas. Através do uso de mediadores de hardware a EMCA permite a implementação de algoritmos DSP independentes de plataforma. São expostos os mediadores de hardware de MAC e *Barrel Shifter* e a interface de Transformada Rápida de Fourier (FFT) da EMCA. Foi avaliada a utilização da EMCA em um decodificador de áudio Codificação de Áudio Avançada (AAC), mostrando que a sua especialização para arquiteturas embarcadas permite a otimização da aplicação sem comprometer sua portabilidade para outras plataformas.

Abstract

Embedded multimedia applications are deployed in different hardware platforms. Compilers are usually available for these platforms however, rarely the code generated from high-level languages compilation is fully optimized for the target hardware. In order to optimize critical parts of the application, it is common to develop Assembly language routines. The use of Assembly language in the application code makes the application dependent of the target platform, becoming difficult to be used in other platforms since the code needs to be rewritten and optimized once again. Migrating an application optimized for a platform, to a new platform, with a new architecture and new hardware features, requires rewriting the application code to the API, developed by the platform vendor. This process takes time, delaying new products introduction to the market and increasing development costs. The application migration may also results in inefficient usage of the new platform hardware, not exploiting its full processing capabilities.

Embedded Multimedia Cross-Platform API (EMCA) is presented in this work. EMCA provides the application developer an hardware independent interface for digital signal processing algorithms, present in embedded multimedia applications, making possible the efficient migration of the application for different hardware platforms. The usage of hardware mediators allows DSP algorithms implementation to be platform independent. EMCA MAC and Barrel Shifter mediators are introduced together with EMCA FFT interface. The usage of EMCA was analyzed in an AAC audio decoder and the results show that the specialization to specific architectures favour the application optimization without compromising its portability to new platforms.

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

AAC Codificação de Áudio Avançada

AIF *Actor-Interchange Format*

ADIF *Audio Data Interchange Format*

ADL Linguagem de Descrição de Arquitetura

ADTS *Audio Data Transport Stream*

AMD *Advanced Micro Devices*

ADESD Desenvolvimento de Sistemas Embarcados Direcionado pela Aplicação

API Interface para Programação de Aplicação

ARM Máquina RISC Avançada

CCD *Charge-coupled Device*

CD *Compact Disc*

CODECs *Codificadores Decodificadores*

CODEC *Codificador Decodificador*

CSDF *Cyclo-Static Dataflow*

DARPA *Defense Advanced Research Projects Agency*

DCT Transformada Discreta de Cosseno

DFT Transformada Discreta de Fourier

DIF *Dataflow Interchange Format*

DSP Processamento de Sinais Digitais

EMCA *Embedded Multimedia Cross-Platform API*

FAAD2 *Freeware Advanced Audio Decoder 2*

FFT Transformada Rápida de Fourier

GCC *Gnu Compilers Collection*

GPLv2 *GNU General Public License version 2.0*

GPP Processador de uso Geral

GPU Unidade de Processamento Gráfico

IFFT Transformada Rápida Inversa de Fourier

FPGA *Field-Programmable Gate Array*

FPU Unidade Ponto Flutuante

IMDCT Transformada Inversa de Cosseno Modificada

ISA Arquitetura de Conjunto de Instruções

LATM *Low-overhead MPEG-4 Audio Transport Multiplex*

LC *Low Complexity*

LD *Low Delay*

LISA *Language for Instruction set architectures (ADL)*

LOAS *Low Overhead Audio Stream*

LTP *Long-Term Prediction*

MDCT Transformada de Cosseno Modificada

MAC Multiplicador Acumulador

MDA *Missile Defense Agency*

MMM *Multimedia Macros*

MMU Unidade de Gerenciamento de Memória

MMX *Multimedia Extension*

MP3 *MPEG-1 Audio Layer 3*

MPEG *Moving Pictures Experts Group*

MSA *Micro Signal Architecture*

PCM Modulação por Código de Pulso

PNS *Perceptual Noise Shaping*

PS *Parametric Stereo*

RISC Computador com Conjunto de Instruções Reduzido

RTG *Register Transfer Graph*

RTL *Register Transfer Level*

SBR *Spectral Band Replication*

SDF *Synchronous Data Flow*

SFML *Simple and Fast Multimedia Library*

SIMD *Simple Instruction Multiple Data*

SO Sistema Operacional

SSDF *Scalable Synchronout Data Flow*

SSE *Streaming SIMD Extensions*

SSR *Scalable Sampling Rate*

TNS *Temporal Noise Shaping*

UML *Unified Modeling Language*

VLIW *Very Long Instruction Word*

VSIPL *Vector Signal Image Processing Library*

WAV *Waveform Audio File Format*

Sumário

Capítulo 1

Introdução

Interfaces para Programação de Aplicações (API, do inglês *Application Programming Interface*) para multimídia são utilizadas há mais de duas décadas, com diferentes finalidades [BAR 98] [RAN 05]. Em sistemas de propósito geral como PCs, estas APIs foram criadas com o objetivo de fornecer ao desenvolvedor um nível de abstração maior do hardware que o exportado pelo sistema operacional, independente da arquitetura. O uso de bibliotecas, através de APIs bem definidas, reduz o tempo de desenvolvimento de novas aplicações.

APIs são criadas por fabricantes de processadores utilizados em sistemas embarcados para que desenvolvedores de aplicações façam uso de bibliotecas de software otimizadas, implementadas em linguagem C, C++ ou *Assembly* da arquitetura alvo, para um fim específico [TEX 08] [FRE 07]. As APIs criadas por fabricantes permitem exportar para o desenvolvedor uma interface única para diferentes processadores de uma mesma família. Fabricantes definem APIs diferentes para suportar funções similares, comuns em aplicações multimídia.

Recentemente, devido à diversidade de arquiteturas utilizadas em aplicações embarcadas, alguns grupos de áreas específicas como computação gráfica, processamento de imagens [JAN 01], processamento de áudio e vídeo [OPE 08b], videogames e equipamentos de comunicação [OPE 08a], criaram APIs comuns para diferentes arquiteturas, com o intuito de padronizar uma camada de software multiplataforma que suporte uma determinada gama de aplicações. O emprego destas APIs no desenvolvimento evita a reescrita da aplicação, ou ao menos de parte desta, quando surge a necessidade do uso de diferentes plataformas. A seção 2.5 lista algumas destas APIs, ressaltando suas características principais.

Neste sentido, o esforço necessário para migração de uma aplicação multimídia embarcada da plataforma na qual esta foi desenvolvida para uma outra pode ser amenizado caso os sistemas de baixo nível, como

bibliotecas de processamento de sinais, e a camada de acesso ao hardware suportem a nova plataforma. Desta maneira, a existência de uma versão do mesmo sistema operacional para a nova plataforma alvo evita a reescrita de boa parte da aplicação, pois as chamadas de sistema serão as mesmas.

Aplicações multimídia embarcadas fazem uso exaustivo de construções de hardware específicas, como Multiplicador Acumulador (MAC), e de instruções especiais de acesso à memória para otimizar algoritmos Processamento de Sinais Digitais (DSP) como as transformadas e os filtros. A criação de APIs voltadas para classes de aplicações específicas permite que um conjunto de funções comuns seja evidenciado, favorecendo o reuso e sugerindo partes do código que, se otimizadas, possivelmente melhorariam significativamente o desempenho da aplicação. Além disso, a padronização de uma interface única para diferentes arquiteturas permite a migração de aplicações para diferentes plataformas de modo suave, sem a necessidade de reescrever código que use construções de hardware específicas.

1.1 Objetivos

O objetivo principal deste trabalho é a proposta de uma API multiplataforma para aplicações multimídia embarcadas, assim como a metodologia utilizada na sua implementação. Esta API tem por objetivo abstrair construções de hardware existentes em arquiteturas embarcadas para otimizar algoritmos DSP empregados na implementação de aplicações multimídia. A interface para estes algoritmos deve ser mantida independentemente da plataforma para qual a API é especializada, fazendo uso das abstrações de hardware para possibilitar implementações eficientes.

A seguir são listados os objetivos específicos definidos em relação ao objetivo principal:

- Pesquisar APIs multimídia para sistemas embarcados; contextualizar o trabalho entre os demais, dissertando sobre as características dos trabalhos listados;

- Estudar os algoritmos DSP que compõem a implementação de *Codificadores Decodificadores* (CODECs) utilizados em aplicações multimídia;

- Propor uma API, independente de plataforma, para suportar algoritmos DSP;

- Validar a API proposta na implementação de um decodificador de áudio, analisando o impacto no seu desempenho;
- Avaliar a eficiência da API proposta em diferentes plataformas, com características distintas a respeito de componentes de hardware empregados na otimização dos algoritmos DSP supracitados.

1.2 Organização do texto

O capítulo 2 expõe trabalhos relacionados ao tema, citando algumas APIs já criadas com objetivos similares.

O capítulo 3 apresenta a EMCA, a API proposta neste trabalho. Neste capítulo são apresentadas as características desta API e a metodologia utilizada na sua implementação.

O capítulo 4 apresenta um estudo de caso no qual a implementação de um decodificador de áudio AAC teve sua implementação alterada para utilizar a EMCA e foi implementada em diferentes plataformas. Uma comparação de desempenho com e sem o uso da EMCA é exposta neste capítulo.

Por fim, o capítulo 5 apresenta as conclusões sobre este trabalho e cita possíveis trabalhos futuros.

Capítulo 2

Portabilidade de Aplicações Multimídia

Aplicações multimídia empregam CODECs em sua composição para realização de codificação, decodificação e processamento dos sinais. CODECs são compostos por algoritmos DSP, dentre os quais destacam-se as transformadas e os filtros, considerados os mais comuns neste domínio. Estes algoritmos demandam muito tempo de processamento em aplicações multimídia, como pode ser visto na seção 4.4. A portabilidade eficiente de algoritmos DSP para diferentes plataformas já foi abordada por diferentes trabalhos de pesquisa, em diferentes níveis de desenvolvimento. Uma implementação eficiente deve fazer uso dos componentes específicos de hardware presentes na arquitetura alvo para otimizar uma determinada tarefa. Deve ainda explorar sua organização de memória para possibilitar o acesso paralelo, quando possível, a dados processados em algoritmos DSP.

Este capítulo apresenta diferentes abordagens existentes para obter a portabilidade de algoritmos DSP, empregados em aplicações multimídia, para prover implementações independentes de plataforma. A seção 2.1 apresenta técnicas e ferramentas utilizadas para prover portabilidade de algoritmos DSP representados em modelos de fluxo de dados (*data flow*). A seção 2.2, cita alguns trabalhos que objetivam melhorar a compilação de código de linguagens de alto nível para processadores DSP. Já a seção 2.3 apresenta o trabalho que descreve o compilador FREEDOM, utilizado para gerar código *Register Transfer Level* (RTL) a partir do código Assembly de uma arquitetura específica. A seção 2.4 relata uma técnica utilizada para prover portabilidade de código através do uso de macros para aplicações multimídia que fazem uso de conjuntos de instruções específicos. E por fim, a seção 2.5 exibe quatro APIs independentes de plataforma, amplamente difundidas, que suportam aplicações multimídia.

2.1 Portabilidade de algoritmos DSP em modelos de fluxo de dados

Algumas ferramentas, como Matlab, permitem a modelagem de algoritmos DSP em Diagramas de Fluxo de Dados. Esta representação é interessante para modelagem de dados em fluxos contínuos (*streams*), como áudio e vídeo, pois permite a representação de forma natural, fazendo uso de grafos. Cada nodo do grafo (chamado de *ator* neste contexto) representa, ou uma computação indivisível (bloco básico), ou outro subgrafo. A complexidade das operações executadas em cada nodo varia de acordo com o sistema modelado, representando desde operações elementares como adição e multiplicação, até algoritmos DSP como FFT e filtros.

Uma aresta *a = (n1,n2)* representa uma espécie de memória FIFO (*First-In First-Out*) que sincroniza a fonte *n1*, nodo de onde o dado sai, e o destino *n2*, nodo para o qual o dado foi enviado, como mostrado na Figura 2.1. Na modelagem de sistemas DSP, uma aresta sempre terá um atraso associado, representado por *del(a)*. Uma aresta possui também um número que determina o conjunto de dados processados na execução do nodo, ou *ator*, destino. No modelo de fluxo de dados, um *ator* pode executar sempre que ele possui o número de dados necessário em suas entradas. Caso o *ator* execute sem os dados necessários, um *buffer underflow* acontece nas arestas que não possuem o número de dados esperado [BHA 00]. É importante que os dados em cada FIFO, aresta, mantenham-se associados durante toda a simulação do modelo *(bounding)*, evitando que algumas FIFOs sejam sobrecarregadas ocasionando estouros de memórias (*buffer overflows*).

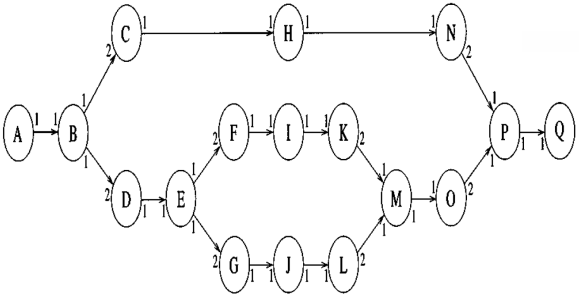

Figura 2.1: Modelo de algoritmo utilizando SDF [BHA 00].

O uso de modelos de fluxo de dados é comum em ferramentas comerciais de desenvolvimento de DSP, como o Simulink da Math Works, *Signal Processing Worksystem* da Cadence e a *Advanced Design System* da HP. Estas ferramentas utilizam *Synchronous Data Flow* (SDF) [LEE 87], que é o modelo de fluxo de dados mais comum no desenvolvimento de DSP. SDF permite a simulação e verificação de algoritmos, além de possibilitar a detecção de impasses (*deadlocks*). A validação da consistência dos modelos de fluxo de dados é bem sucedida desde que o modelo não gere *buffer underflows ou overflows*. Além da checagem de consistência é possível também realizar, durante a síntese do software a partir do modelo SDF, o escalonamento da execução dos nodos como descrito por Marwedel [BHA 00]. No caso de SDF, o escalonamento da execução dos nodos é estático, criando uma tabela com a ordem de execução de cada nodo. Existem modelos de fluxo de dados que são variantes do SDF, permitindo contornar algumas de suas limitações. O *Cyclo-Static Dataflow* (CSDF) [BIL 95], por exemplo, é uma extensão do SDF que permite que o número de dados consumidos e produzidos em um *ator*, seja variável, permitindo a modelagem de algoritmos que operam em conjuntos de dados com tamanho variável. Uma outra extensão do SDF é o *Scalable Synchronout Data Flow* (SSDF) [RIT 93], criado na universidade de Aachen e empregado na ferramenta COSSAP, mantida pela Synopsys. Esta representação permite a vetorização dos *atores*, favorecendo a síntese destes para código eficiente em arquiteturas com suporte a processamento de vetores.

Tendo em vista a portabilidade de algoritmos desenvolvidos em representações de fluxo de dados para diferentes ferramentas Bhattacharyya propõe o *Dataflow Interchange Format* (DIF) [HSU 04]. DIF é uma representação textual adaptada para capturar modelos de representações de fluxo de dados, independente do modelo e da ferramenta utilizada na modelagem. DIF foi desenvolvida para ser importada e exportada por diferentes ferramentas, permitindo também que desenvolvedores façam alterações em sua representação textual. O *Actor-Interchange Format* (AIF) permite uma descrição DIF ser importada ou exportada para ferramentas de síntese de software DSP [HSU 05]. Neste contexto, exportar um modelo em uma determinada ferramenta significa traduzir aquele modelo para uma representação DIF, e isto é feito utilizando AIF integrado à ferramenta que está sendo utilizada. Por outro lado, importar um modelo significa tornar um modelo DIF compatível com a ferramenta utilizada. AIF foi validada no projeto Ptolemy II [LEE 03], desenvolvido em Berkeley, utilizando síntese de código Java a partir de modelos de fluxo de dados representados em DIF, e também integrada à ferramenta *Autocod-*

ing ToolSet, desenvolvida pela *Missile Defense Agency* (MDA), que gera código C a partir de modelos DIF.

Após a verificação do algoritmo e seus parâmetros em modelos de fluxo de dados, é realizada a síntese de software, que é o processo de tradução de representações de fluxo de dados, como SDF, para linguagens de programação de aplicações DSP, como C.

A estratégia de permitir a portabilidade de algoritmos DSP em representações de fluxo de dados apresenta vantagens, pois neste nível são possíveis otimizações independentes de implementação e a validação e simulação de algoritmos. Porém, a qualidade do código de algoritmos modelados em fluxo de dados é dependente das ferramentas de síntese de software para linguagens de programação. Até a presente data, estas ferramentas geram código genérico, que necessita de otimizações manuais para arquiteturas específicas, dificultando que o modelo de fluxo de dados seja automaticamente transformado em código *Assembly* eficiente da plataforma alvo, que faça uso de toda sua potencialidade.

A linguagem de programação mais utilizada no desenvolvimento de aplicações DSP é C. Compiladores C estão disponíveis para todas as grandes famílias de processadores DSP, como a família TMS320xx da Texas Instruments, a família 21xx da Analog Devices e a família 56xx da Motorola. A seção a seguir apresenta trabalhos que buscam melhorar a compilação de código C para processadores DSP.

2.2 Compilação redirecionável para processadores DSP

Assim como acontece em aplicações embarcadas que não fazem processamento de sinais, seria interessante que o código de aplicações DSP fosse escrito em linguagens de alto nível, como C/C++, e pudesse ser compilado para diferentes arquiteturas. Como relatado em [ZIV 95], a compilação para processadores DSP não gera código eficiente devido à heterogeneidade das suas unidades funcionais e das diferentes organizações de memória existentes. Neste trabalho foi medido o sobrecusto de desempenho e de tamanho de código de diferentes algoritmos DSP implementados em C comparados com algoritmos otimizados em *Assembly*. Implementações em C mostraram-se até 700% mais lentas que as implementações otimizadas manualmente em linguagem *Assembly*. Em um trabalho similar, realizado por Levy, fabricantes de arquiteturas DSP foram solicitados para compilarem um conjunto de *benchmarks* escritos em C, disponíveis em duas versões, uma otimizada para a arquitetura, através do uso de bibliotecas dos fabricantes, e outra independente de plataforma. Os resultados mostraram que o código C genérico, sem otimiza-

ções, resulta em sobrecusto inaceitável em tamanho de código e desempenho [LEV 97].

A compilação pode ser dividida em cinco fases: análise léxica, sintática, semântica, otimização independente de máquina e geração de código. A geração de código pode ser dividida em três fases: seleção de código; alocação de registradores e escalonamento de instruções. A seleção de código mapeia a representação intermediária do compilador de acordo com as métricas utilizadas, visando reduzir o tamanho do código, número de instruções, melhorar o desempenho ou, reduzir número de ciclos de execução. A alocação de registradores mapeia variáveis e valores intermediários em registradores da máquina. O escalonamento de instruções ordena as instruções selecionadas para permitir uso eficiente do *pipeline* do processador e evitar a movimentação desnecessária de registradores para a memória. Devido a diferença existente entre a semântica de código de uma aplicação C e as instruções *Assembly* presentes no conjunto de instruções de um processador DSP, algumas técnicas são utilizadas para melhorar a geração de código para processadores DSP [BHA 00].

Guido Araújo identificou quais estruturas são usualmente empregadas em arquiteturas DSP ponto fixo, estabelecendo uma classe de arquiteturas para as quais seria possível gerar código eficiente [ARA 98]. O trabalho mencionado propõe um algoritmo ótimo que em duas fases realiza a seleção de instruções, alocação de registradores e escalonamento de instruções para processadores DSP ponto fixo em tempo polinomial. Este algoritmo é baseado na descrição do conjunto de instruções (Arquitetura de Conjunto de Instruções (ISA)) do processador DSP. Na primeira fase, o algoritmo realiza a seleção de instruções e a alocação de registradores, levando em consideração quais registradores são exclusivos para uso em uma determinada unidade computacional da arquitetura alvo. Em arquiteturas de processadores DSP é comum que apenas os registradores de um determinado banco sejam utilizados como operandos para instruções específicas, e, portanto, este aspecto é considerado na alocação de registradores. A seleção de instruções e alocação de registradores não é suficiente para geração de código otimizado, é necessário que as instruções sejam ordenadas de maneira conveniente para preencher o *pipeline* da arquitetura alvo e tirar o máximo proveito das unidades computacionais disponíveis. A segunda fase do algoritmo proposto por Araújo realiza o ordenamento das instruções utilizando o modelo *Register Transfer Graph* (RTG), onde os nodos dos grafos representam qualquer componente do *Datapath* onde um dado pode ser armazenado, como bancos de registradores ou acumuladores. Para *Datapaths* acíclicos, o algoritmo proposto permite a

geração de código ótimo.

O trabalho de doutorado de Ashok Sudarsanam [SUD 98], buscou tornar a compilação para DSPs mais eficiente utilizando bibliotecas de otimização de código dependentes de arquitetura para compilação redirecionável (*Retargetable Compiling*). Partindo do pressuposto que um compilador redirecionável pode gerar código para diferentes arquiteturas por meio de *flags*, Sudarsanam propõe o uso de bibliotecas para suportar cada característica (*feature*) presente na arquitetura, sendo estas invocadas com o uso de *flags*. De acordo com a abordagem proposta, um compilador redirecionável poderia gerar código para arquiteturas *Very Long Instruction Word* (VLIW), com múltiplos multiplicadores em hardware, ativando as *flags* respectivas a esta arquitetura durante a compilação, o código gerado faria uso de bibliotecas em linguagem *Assembly* que utilizariam estes multiplicadores paralelamente. Este trabalho obteve significativos resultados em compilação redirecionável para três DSPs ponto fixo diferentes.

A área de compilação redirecionável, incluindo compilação para processadores DSP, existe há muito tempo e muitos autores criaram diferentes técnicas e abordagens para o problema. Rainer Leupers realizou um estudo das técnicas até então utilizadas em compilação redirecionável. Este trabalho propõe uma taxonomia de arquiteturas interessante, separando processadores DSP e processadores multimídia [LEU 00]. Processadores multimídia, de acordo com o autor, seriam uma mistura de processadores Computador com Conjunto de Instruções Reduzido (RISC) e processadores DSP, com múltiplas unidades funcionais que funcionam em paralelo (VLIW). Neste mesmo artigo, o autor cita que em aplicações multimídia o uso de classes C++ ou bibliotecas em *Assembly*, com interfaces em arquivos de cabeçalhos (*headers*) C, são as técnicas mais utilizadas para a otimização de algoritmos críticos.

Além dos trabalhos já citados, outros também realizaram avanços em compilação redirecionável como em [JUN 01], onde é apresentado um otimizador de código para arquiteturas DSP portável. Um trabalho mais recente, do qual Leupers também fez parte [HOH 09], propõe a criação de um *Framework* de otimizações redirecionáveis para arquiteturas multimídia com suporte a *Simple Instruction Multiple Data* (SIMD). Este trabalho relata uma abordagem onde as otimizações realizadas na geração de código do compilador são baseadas na descrição da arquitetura alvo. A descrição do comportamento, da estrutura e das interfaces de entrada e saída da arquitetura é feita utilizando *Language for Instruction set architectures (ADL)* (LISA). Este trabalho mostrou excelentes resultados para a arquitetura TriMedia e ARM11 em comparação ao código C gerado pelo compilador sem as otimizações.

A complexidade e diversidade das arquiteturas utilizadas em aplicações DSP embarcadas tornam a abordagem de geração de código utilizando compiladores redirecionável difícil de ser empregada, pois a geração de código do compilador teria que ser totalmente adaptada para cada nova característica existente em uma nova arquitetura. Esta técnica pode ser melhor empregada no caso da existência de um modelo detalhado da arquitetura, em uma Linguagem de Descrição de Arquitetura (ADL), como proposto por Leupers em [HOH 09], para que diferentes partes da geração de código possam tomar decisões de otimização de código para a arquitetura alvo. O compilador *RECORD* [LEU 97b] emprega modelos de processadores na linguagem MIMOLA, que contém a unidade de geração de endereços, os registradores, unidade de controle e o *Datapath* [LEU 97a]. O módulo de geração de código do compilador RECORD pode ser totalmente gerado a partir da descrição do processador, e também é possível a extração do seu conjunto de instruções.

Processadores DSP possuem registradores com funções específicas e organizações de memória não convencionais. Um exemplo é a utilização de determinados registradores para a indexação de *arrays* que são manipulados na unidade de geração de endereços, em paralelo com a realização de operações em dados na unidade aritmética que pode ser visto na Figura 3.7, no capítulo seguinte. Sem que o compilador conheça as especificidades da arquitetura alvo da compilação, é pouco provável que a geração de código faça uso de todo o seu potencial.

2.3 Tradução de código binário de processadores DSP para dispositivos lógico-programáveis

A criação de dispositivos *Field-Programmable Gate Array* (FPGA) com blocos em hardware para suportar operações comuns em DSP apresenta uma nova alternativa de plataforma para o desenvolvimento de aplicações multimídia. É esperado que implementações de aplicações multimídia explorem todo o paralelismo do hardware de FPGAs e tenham desempenho melhor que implementações realizadas em processadores DSP. Entretanto, a implementação de de aplicações de processadores DSP para FPGAs não é uma tarefa trivial.

Zaretsky apresenta um processo de tradução automática de código DSP *Assembly*, da família C6000 da *Texas Instruments* para código RTL que pode ser sintetizado em FPGAs [ZAR 04]. A proposta dos autores é manter o fluxo de desenvolvimento tradicional de aplicações DSP iniciado na modelagem em fluxo de dados, seguido da síntese de software, e por último a compilação para processadores DSP. O código gerado para

uma arquitetura DSP específica seria então "traduzido" para código RTL, utilizando o compilador FREEDOM apresentado naquele trabalho. Este trabalho apresenta uma técnica de portabilidade que pode resultar em melhor desempenho da aplicação, com fatores de aceleração de até 20 vezes em relação ao número de ciclos de instrução, nos experimentos realizados.

O compilador FREEDOM foi desenvolvido para ser independente de *Assembly*, para tanto, necessita da descrição do conjunto de instruções da arquitetura como entrada. A partir do código binário do DSP e do conjunto de instruções, o analisador sintático (*parser*) gera uma representação virtual interna chamada *Machine Language Abstract Syntax Tree (MST)* que é genérica o suficiente para suportar a maioria dos conjuntos de instruções. O MST é então transformado em uma representação interna de fluxo de dados (*Control and Data Flow Graph* - CDFG), na qual são feitas otimizações e separações de *arrays*. Graças a representação na forma de fluxo de dados é possível identificar blocos que podem ser paralelizados no hardware, o que resulta no ganho de desempenho desta proposta. Diversas otimizações, como desenrolamento de loops, remoção de subexpressões comuns, remoção de áreas de código não alcançadas, alocação de registradores, etc. são executadas na representação CDFG. Algumas limitações de processadores DSP são contornadas na geração do código RTL pelo compilador. Processadores DSP normalmente possuem dois barramentos de acesso as memórias de dados, o que pode ser expandido na geração do sistema em RTL, assim como o número de registradores utilizados pela aplicação. A síntese de blocos específicos de memória existentes para FPGA alvo permite maior eficiência do hardware gerado, ocupando menos área e reduzindo os tempos de acesso.

O trabalho de Zaretsky obteve significativos resultados com o compilador FREEDOM [ZAR 04]. A tradução de blocos de código DSP, *kernels* DSP como filtros e FFTs, do processador C6000 da *Texas Instruments* para RTL sintetizado na FPGA VirtexII da Xilinx utilizando a ferramenta Synplify Pro 7.2 apresentou redução de até 20 vezes no número de instruções necessárias para realizar a implementação de alguns algoritmos DSP específicos. Nos experimentos realizados, a plataforma *Virtex2* da Xilinx, com *clock* máximo de 150MHz, possibilitou uma redução de aproximadamente 6 vezes do tempo de execução da aplicação comparado ao processador C6000 com *clock* de 300MHz.

2.4 Portabilidade de aplicações multimídia em nível de aplicação

Desenvolvedores de aplicações multimídia fazem uso de extensões de hardware existentes em DSPs para otimizar algoritmos que deman-

dam muito processamento. Algumas destas extensões provaram-se indispensáveis e foram empregadas também em Processadores de Uso Geral (GPP), como múltiplas unidades de execução para suporte a instruções SIMD empregadas no processamento de vetores. O uso destas extensões não é trivial e dificilmente é realizado de forma automática (compilação). Por isso, aplicações que buscam melhor desempenho através do uso destas extensões de hardware, necessitam de otimizações em código *Assembly*.

Para permitir a otimização de código C para processadores DSP, fabricantes e desenvolvedores criaram técnicas que permitem exportar rotinas otimizadas para determinadas plataformas alvo. Uma destas técnicas é a criação de *macros* internas do compilador, a serem substituídas por instruções *Assembly* da arquitetura alvo. Esta solução exige que o desenvolvedor utilize no código da aplicação funções internas do compilador que são extensões ao padrão ANSI C, criadas para indicar ao compilador rotinas de código a serem geradas. Esta técnica impede que o código da aplicação seja compilado por outro compilador, que provavelmente, não possui as rotinas internas. Uma outra técnica é a criação de bibliotecas *Assembly* para a arquitetura alvo, juntamente com a disponibilização de arquivos de cabeçalhos de funções C para seu acesso, porém esta abordagem também torna o código da aplicação dependente da API das bibliotecas da plataforma alvo.

Rojas [ROJ 03] apresenta uma maneira de tornar aplicações multimídia portáveis, independente de compilador e da API do fabricante, com o uso de macros do pré-processador C (CPP). MMM (*Multimedia Macros*) permite a definição de macros que resolvem problemas como a largura de registradores e diferentes conjuntos de instruções. A "biblioteca" de macros proposta neste trabalho foi testada em quatro arquiteturas e mostrou-se eficiente, pois o desempenho compara-se ao da própria implementação em código *Assembly* otimizada. A Figura 2.2 mostra um exemplo de uso da MMM onde são criados três vetores de dados de 128 bits, os vetores **`A`** e **`B`** são inicializados a partir de ponteiros e então são somados. O resultado da soma é atribuído ao vetor **`C`**.

O mapeamento de vetores de 128 bits, 16 *bytes*, para arquiteturas com conjuntos de instruções SSE2 (Conjunto de instruções multimídia Intel) ou AltiVec (Conjunto de instruções multimídia para PowerPC) utiliza registradores de 128 bits, disponíveis nestes conjuntos de instruções multimídia. Desta maneira, as operações sobre estes dados são realizadas em um ciclo de *clock*. Em arquiteturas sem instruções multimídia e sem registradores de 128 bits, é utilizada uma versão de MMM que emula estas operações, utilizando tipos de dados menores, como nas arquiteturas MMX e TriMedia.

```
DECLARE_I16x8(A);
DECLARE_I16x8(B);
DECLARE_I16x8(C);

LOAD_A_I16x8(A, pSrcA);
LOAD_A_I16x8(B, pSrcB);

ADD_I16x8(C, A, B);
```

Figura 2.2: Uso da MMM para declaração, inicialização e adição de dados de 128 bits [ROJ 03]

A abordagem empregada na construção da MMM é interessante, porém o uso de macros pode dificultar o desenvolvimento de aplicações complexas. No entanto, esta proposta é realmente válida para o porte de aplicações para arquiteturas onde a MMM já tenha sido validada, permitindo que o desenvolvedor que a utiliza concentre-se na aplicação e não em depurar o código gerado a partir das macros. A seção a seguir apresenta APIs que também permitem a portabilidade de aplicações multimídia no nível de aplicação.

2.5 APIs multimídia multiplataforma

O uso de APIs para diferentes plataformas, criadas por fóruns compostos por fabricantes de arquiteturas, ou por grupos de criação e manutenção de padrões abertos, é a maneira mais aceita para prover portabilidade de aplicações multimídia em sistemas embarcados. Diversas iniciativas existem com o objetivo de padronizar uma interface de software para acesso a algoritmos DSP utilizados em plataformas embarcadas. Propostas de APIs frequentemente utilizadas em sistemas embarcados são apresentadas a seguir.

2.5.1 OpenMAX e OpenSL ES

A *OpenMAX* [OPE 08b], proposta pelo grupo Khronos, criador da API de computação gráfica OpenGL [KHR 09], especifica uma API padrão livre em três camadas: aplicação (*OpenMAX Application Layer - AL*), integração (*OpenMAX Integration Layer - IL*) e desenvolvimento (*OpenMAX Development Layer - DL*). As três camadas da *OpenMAX* podem ser vistas na Figura 2.3. A camada de aplicação, AL, é voltada para o suporte de dados de mídia, como áudio e vídeo, fornecendo componentes

de abstração para sua gravação e reprodução. A camada de integração, IL, objetiva realizar a integração dos componentes de mídia fornecidos na AL em diferentes hardwares, de maneira uniforme. A IL é uma espécie de *middleware* que permite a conexão de diferentes implementações, de diferentes fabricantes, com requisitos de inicialização distintos através de uma interface única.

A camada de desenvolvimento, DL, define a interface de acesso ao hardware. Na DL são definidas funções comuns em multimídia, como algoritmos DSP utilizados em *Codificador Decodificador*s (CODECs) de áudio e vídeo, que são exportadas para a camada de integração. O objetivo da DL é prover portabilidade de algoritmos acelerados em hardware em diferentes plataformas, identificando seus pontos comuns e sua aplicação em CODECs frequentemente utilizados. A DL deve ser reimplementada para cada plataforma que almeja ser compatível com a API *OpenMAX*.

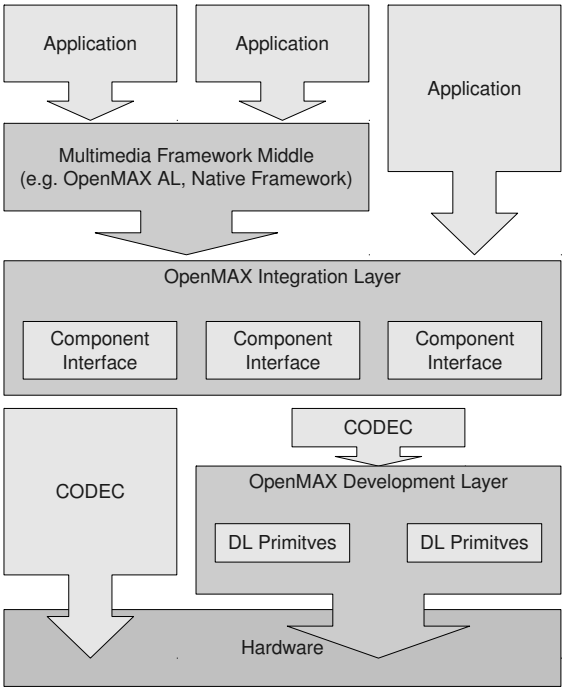

Figura 2.3: Disposição das camadas de aplicação, integração e desenvolvimento da API *OpenMAX* [OPE 08b]

Apesar de oferecer em sua interface acesso à blocos de alto nível como decodificadores de áudio e vídeo, a DL é a API da *OpenMAX* que mais se assemelha com a API proposta neste trabalho. Na DL, são definidas interfaces de filtros, FFTs, DCTs, e outros algoritmos de DSP. Diferentes domínios foram criados para particionar a DL: processamento de sinais; processamento de imagens; codificação de imagens; codificação de áudio; e codificação de vídeo. Cada um destes domínios possui interface para blocos (*kernels*) empregados nas respectivas tarefas. Todas as interfaces de métodos e tipos de dados são classificadas em quatro categorias no que diz respeito a conformidade da implementação com o padrão. Interfaces obrigatórias (*shall*) devem estar presentes em implementações da DL, caso contrário, funções internas da API podem não funcionar corretamente. Interfaces desejáveis (*will*) não são usadas pela implementação da API, porém são frequentemente usadas no domínio de aplicação ou em implementações de terceiros. Interfaces que podem (*should*) ser implementadas também não são mandatórias para conformidade com o padrão, porém sua implementação está de acordo com boas práticas de compatibilidade. Interfaces opcionais oferecem características extras que nem sempre são utilizadas no domínio de aplicação.

Assim como a VSIPL, apresentada na seção a seguir, a DL também define tipos de dados. A DL define diferentes precisões para tipos de dados escalares e de ponto flutuante, além de estruturas de dados como retângulos e pontos. As operações internas da API podem utilizar tipos de dados com precisão maior que os operandos e consequentemente as saídas devem ser sempre truncadas. A DL oferece instruções para ajuste *(scale)* de resultados saturados. Uma implementação de referência da DL, em código C, é disponibilizada pelo grupo Khronos, que pode ser utilizada como base para novas implementações.

A DL não prevê funções de inicialização da biblioteca e nem gerenciamento de memória, o que resulta em menor impacto no desempenho da aplicação. Um estudo feito pela *On2 Technology* [RIN 09] utilizando a implementação da DL para a plataforma ARM Cortex-8, que possui o conjunto de instruções SIMD NEON [ARM 97], mostrou que é possível realizar otimizações no código C, utilizando uma implementação específica da DL. A DL foi utilizada na implementação de um CODEC de vídeo MPEG-4 resultando em baixo impacto no seu desempenho.

O padrão *OpenMAX* permite que fabricantes criem implementações dos diferentes níveis da API para suas arquiteturas. A implementação da API de desenvolvimento, que é a mais difundida, pode ser modificada sem que o código da aplicação, que utiliza as APIs IL e AL, necessite de modificações. A Máquina RISC Avançada (ARM)® possui uma imple-

mentação em C de distribuição livre para a camada de desenvolvimento da *OpenMAX* com funções para implementação de CODECs de áudio e vídeo [ARM 08]. A *Mentor Graphics*, possui uma implementação da camada de integração da *OpenMAX*, disponível como parte do seu sistema operacional Nucleus, voltado para aplicações multimídia [NUC 09]. O fato de ser um padrão aberto favorece a disseminação da *OpenMAX*, que disponibiliza um conjunto de *Headers* em código C para guiar os desenvolvedores de cada nível da API e também desenvolvedores de aplicações.

A *OpenSL ES*, também criada pelo grupo Khronos, define uma interface para aplicações de áudio embarcadas, separada em três perfis: músicas, telefones e jogos. Claramente, o foco desta especificação são aparelhos celulares, PDAs, *smartphones*, dispositivos de jogos e reprodutores de música móveis. A especificação da *OpenSL ES* define um conjunto de funções, com *headers* C, usadas na gravação e reprodução de áudio, como CODECs e chamadas para *drivers* de interfaces proprietárias presentes em diferentes plataformas. A *OpenSL ES* é base de APIs de áudio alto nível, como a *Mobile Media API* (JSR-135) [MMA 09] e *Advanced Multimedia Supplements* (JSR-234) [AMM 09], ambas partes da interface Java para sistemas embarcados.

A *OpenSL ES* emprega orientação a objetos (OO) na sua implementação. Dois conceitos fundamentais de OO definem a API, objetos e interfaces. Um objeto é uma abstração de um conjunto de recursos, empregados em um conjunto de tarefas bem definido. O tipo de um objeto determina o conjunto de tarefas para o qual ele é usado, e é definido em sua criação. Uma interface é um conjunto de características e métodos que um objeto possui. Um identificador de interface (ID) determina um tipo de interface, e é utilizado no código para se referir a esta interface. A API da *OpenSL ES* define métodos para acesso e alocação de recursos, que preveem compartilhamento destes. A implementação da API deve prever o uso de *Threads* para permitir o compartilhamento de recursos *(thread-safe)* quando necessário.

Seis tipos de objetos estão presentes na API da *OpenSL ES*:

- *Engine*: Permite a criação e consulta dos demais objetos;
- Mídia: Implementam casos de uso multimídia, permitindo gravação e reprodução de áudio;
- Extratores de Metadata: Permitem a extração das propriedades de objetos de mídia;
- Saída de áudio: Permite a combinação de mais de uma fonte de áudio em um destino;

- LED *arrays*: Permite o controle dos LEDs de dispositivos;
- Controle de vibração: Controle de vibração do dispositivo;

Como mostrado acima, os objetos definidos na *OpenSL ES* são voltados para o controle e manipulação de áudio em sistemas embarcados, como celulares e *handhelds*, provendo acesso a dispositivos presentes nestas plataformas.

2.5.2 VSIPL

A *Vector Signal Image Processing Library* (VSIPL) [JAN 01] foi a precursora das APIs para processamento de sinais na forma de padrão aberto. VSIPL é fruto de uma iniciativa conjunta de diversas empresas e institutos de pesquisa (*HRL Laboratories, Space and Naval Warfare Systems Center-San Diego, MIT Lincoln Laboratory, Georgia Tech Research Institute*, entre outras), financiadas pelo *Defense Advanced Research Projects Agency* (DARPA). O *VSIPL Forum* foi criado para definição de uma API para processamento de sinais independente de plataforma, a especificação 1.0 da VSIPL foi divulgada em Março de 2000 e a última versão, VSIPL 1.3, data de Janeiro de 2008.

O propósito da VSIPL é suportar aplicações de alto desempenho portáveis. O padrão, baseado em bibliotecas estáveis de computação científica, exporta uma camada de abstração independente de arquitetura e organização de memória para aplicações de DSP embarcadas. A especificação VSIPL API 1.3 [SCH 08] exporta uma interface única, independente de plataforma, divida em diferentes funcionalidades, listadas abaixo.

- Suporte: Inicialização e finalização da biblioteca; Criação e destruição de objetos; Gerenciamento de Memória;
- Operações sobre números escalares;
- Operações sobre vetores;
- Geração de números aleatórios;
- DSP: FFT, filtros, correlação e convolução;
- Álgebra Linear: Operações sobre matrizes; Sistemas lineares; Método dos mínimos quadrados;

Não é necessário que todas as funções especificadas na VSIPL sejam implementadas para prover uma biblioteca compatível. O VSIPL *Forum* criou perfis para diferentes subconjuntos da API. O perfil *Core* prevê a implementação das APIs de DSP e de operações sobre matrizes. O perfil *Core Lite* prevê um subconjunto das funções nas APIs de operações sobre vetores e DSP frequentemente empregadas em aplicações multimídia embarcadas.

A especificação VSIPL provê funções e tipos de dados que devem ser implementadas para que uma biblioteca seja compatível com um perfil da API. Os dados são armazenados em um *array* unidimensional, e aparecem como um bloco de memória contíguo para o desenvolvedor da aplicação. *Arrays* de dados podem ser vistos como tipo real ou complexo. Todas as operações sobre tipos de dados VSIPL acontecem de maneira indireta, através de *view objects*.

Todos os objetos presentes na VSIPL são um tipo de dado abstrato, que define os atributos e os dados acessado pelo objeto. Todos os objetos são criados e alocados pela VSIPL. O desenvolvedor da aplicação não possui permissão para alocar objetos fora da VSIPL. Os *arrays* de dados, local da memória onde dados são armazenados, podem existir em dois espaços lógicos: espaço de dados do usuário e espaço de dados da VSIPL. As funções criadas pelo usuário só podem operar dados no espaço de usuário e as funções definidas na VSIPL só operam sobre dados no espaço da VSIPL. O gerenciamento de memória é realizado pela implementação da VSIPL. Os dados podem migrar entre o espaço de usuário e o espaço da VSIPL.

O desenvolvedor da aplicação pode acessar dados no espaço da VSIPL através de blocos (*blocks*). Um *array* de dados associado a um bloco é um conjunto de elementos contíguos do mesmo tipo, definidos pela VSIPL. Assim como os dados, existem duas categorias de blocos, de usuário e da VSIPL. Um bloco de usuário está associado a um *array* de dados de usuário e um bloco VSIPL está associado a um *array* de dados VSIPL. Blocos podem ser criados sem *arrays* de dados e depois serem associados. Um bloco associado a um *array* de dados pode estar em dois estados: admitido (*admitted*) ou liberado (*released*). Quando admitido, um bloco estará no espaço lógico da VSIPL, e quando liberado estará no espaço de dados do usuário. Blocos VSIPL estão sempre no estado admitido. Blocos de usuário são criados em estado liberado e devem ser admitidos para permitir sua manipulação por objetos VSIPL.

Blocos são sempre armazenados em espaços de memória contíguos porém, é comum que desenvolvedores necessitem operar sobre subconjuntos de dados não contíguos destes blocos. Para tanto, a VSIPL requer

que o desenvolvedor defina uma *view* de um bloco. *Views* permitem a definição do comprimento dos dados, o passo em que estes dados devem ser acessados e o deslocamento a partir do qual os dados devem ser acessados. A Figura 2.4 mostra a estrutura de uma aplicação que utiliza a VSIPL.

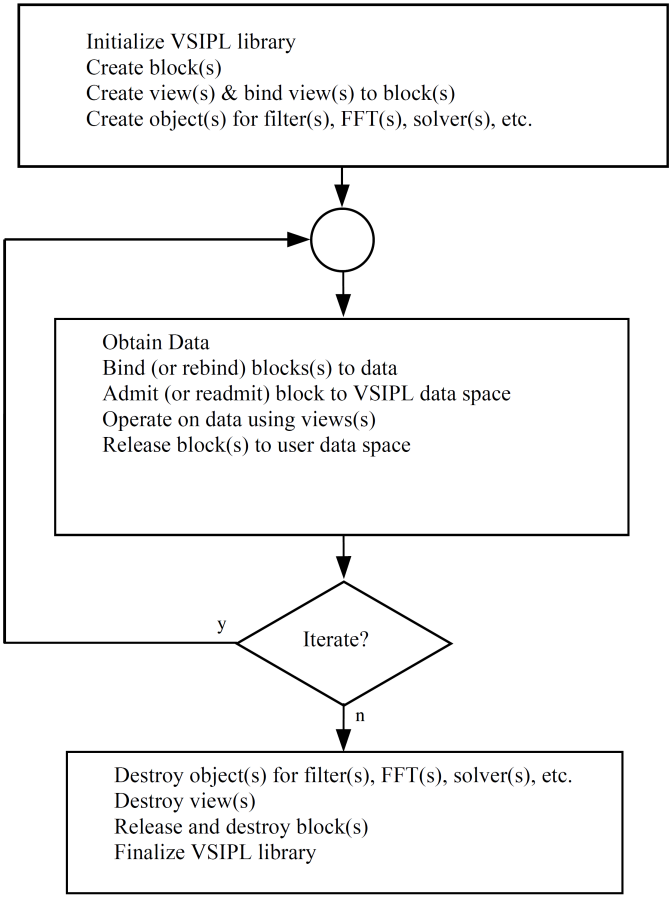

Figura 2.4: Estrutura de uma aplicação utilizando VSIPL [SCH 08].

Uma aplicação VSIPL deve inicializar a biblioteca com uma chamada ao método **`vsip_init`** antes de chamar métodos de objetos da biblioteca. Depois da execução da aplicação, o método **`vsip_finalize`**

deve ser executado antes da finalização. Toda implementação VSIPL suporta pelo menos um tipo ponto flutuante e um inteiro. Todas as funções especificadas na VSIPL possuem "**vsip_**" no início do nome, já os tipos de dado reais possuem "**r_**" no início do nome e os tipos complexos iniciam com "**c_**".

A VSIPL foi bastante empregada na área de computação paralela e implementações atuais, como a VSIPL++ 2.2 da CodeSourcery [COD 09], são otimizadas para determinadas plataformas embarcadas, como o Cell/B.E., o que a torna interessante para aplicações de alto desempenho. O instituto de pesquisas Georgia Tech em parceria com a nVidia, disponibiliza uma implementação da VSIPL (apenas código binário) para Unidade de Processamento Gráficos (GPUs) das plataformas CUDA da nVidia. O uso desta biblioteca permitiu fatores de aceleração de até 75 vezes para uma aplicação de mapa de *doppler* em uma GeForce GTX 280 utilizada em um processador Core2Duo da Intel de 2.83 GHz [KER 08].

2.5.3 SFML

Simple and Fast Multimedia Library (SFML) [SFM 09] é uma API C++ livre, que proporciona acesso ao hardware gráfico, entradas e saídas, áudio, etc. para diferentes plataformas. A licença desta API é livre para qualquer uso, inclusive comercial. Existem versões da SFML para Unix, para *Mac* e para *Windows*, com implementações em diferentes linguagens: C, .NET, *Python*, *Ruby*, além de C++. Esta API foi construída para oferecer portabilidade em sistemas de propósito geral para aplicações gráficas e multimídia. Um aspecto interessante da SFML é a simplicidade, que já está presente no nome. A Figura 2.5 mostra as chamadas necessárias para abrir e executar um arquivo de áudio Ogg, formato livre e aberto que acomoda arquivos de áudio, vídeo e texto de diferentes CODECs, utilizando a SFML.

O exemplo presente na Figura 2.5 esconde completamente da aplicação o hardware em que esta executa. Desta maneira, a migração desta aplicação para outra plataforma, que seja suportada pela SFML, não requer adaptação. Ao invés de definir a interface de chamadas a transformadas e filtros, que são parte da implementação de CODECs de áudio, esta API define a chamada direta ao decodificador. Das APIs já mencionadas, apenas a camada de aplicação da *OpenMAX* se assemelha à interface da SFML.

O ponto comum das APIs OpenMAX, OpenSL ES e VSIPL, apresentadas nas seções 2.5.1 e 2.5.2 respectivamente, é que estas são padrões abertos, deixando a implementação a cargo de diferentes fabricantes ou

```
#include <SFML/Audio.hpp>
#include <iostream>

int main() {
    // Carrega o arquivo de audio Ogg
    sf::Music Music;
    if (!Music.OpenFromFile("musica.ogg")) return EXIT_FAILURE;

    // Executa o arquivo
    Music.Play();
    std::cout << "Playing ... " << std::endl;

    // Loop enquanto executa
    while (Music.GetStatus() == sf::Music::Playing)
        // Reserva tempo para que outras threads usem a CPU
        sf::Sleep(0.1f);

    return EXIT_SUCCESS;
}
```

Figura 2.5: Execução de arquivo de áudio Ogg com chamada a API SFML [SFM 09]

grupos de desenvolvimento. Em alguns casos, como a implementação da VSIPL realizada pela *CodeSourcery*, a implementação é um produto específico para uma plataforma alvo, comercializado pelo desenvolvedor. Uma outra abordagem, como a SFML apresentada nesta seção, fornece a implementação da API proposta como parte do padrão, sendo esta uma camada de software provida pela equipe que especifica a interface.

Uma API para aplicações multimídia em sistemas embarcados deve permitir ao desenvolvedor optar por utilizar implementações consolidadas de algoritmos disponíveis na sua interface, ou fornecer a sua própria implementação. No entanto, o desenvolvedor deve ter acesso ao hardware disponível, encapsulado pela API, para permitir portabilidade. Portanto, ao exportar uma transformada como a FFT, a API não deve restringir o desenvolvedor a utilizar uma implementação específica, e sim permitir que este crie sua própria implementação de FFT, utilizando, ou não, as estruturas de hardware encapsuladas. É importante que uma API permita ao desenvolvedor utilizar implementações de algoritmos de terceiros, buscando uma solução que melhor se adapte a sua aplicação. Este é o princípio da EMCA, como mostrado no capítulo a seguir.

Capítulo 3

Embedded Multimedia Cross-Platform API

A *Embedded Multimedia Cross-Platform API* (EMCA) foi criada para prover portabilidade eficientemente para algoritmos de DSP utilizados em aplicações multimídia. Para satisfazer este objetivo, a EMCA foi desenvolvida a partir da engenharia de domínio de aplicações multimídia seguindo a metodologia Desenvolvimento de Sistemas Embarcados Direcionado pela Aplicação (ADESD), apresentada na seção 3.2, proposta por Fröhlich em [FRÖ 01]. Foi realizada uma análise das extensões de hardware criadas para otimização de aplicações multimídia em plataformas embarcadas a serem suportadas pela EMCA.

A engenharia de domínio realizada analisou diferentes aplicações multimídia identificando componentes de software significativos que poderiam ser acessados através de interfaces bem definidas. Esta organização auxilia a decomposição do problema em partes menores, possibilitando a criação de interfaces para algoritmos DSP que podem ser especializadas para diferentes arquiteturas.

3.1 Fundamentação

Aplicações multimídia fazem uso de CODECs em sua composição. A função de um CODEC é manipular, traduzir, compactar ou descompactar sinais de áudio, vídeo, ou imagens. Sinais de áudio e vídeo são captados por algum tipo de sensor, como sensores fotoelétricos para imagens, e microfones para áudio, e utilizam um conversor específico, como *Charge-coupled Device* (CCD) para imagens, e conversores A/D para áudio, para então serem digitalizados. O dado digital que representa o sinal sem compressão, pode ser muito grande para ser armazenado ou transmitido. Desta maneira, a função de um codificador é compactar um sinal digital levando em consideração suas características e a percepção do ser humano; um decodificador, por sua vez, descompacta este sinal para ser

reproduzido.

CODECs baseados em modelos psicoacústicos, como *MPEG-1 Audio Layer 3* (MP3) e AAC, consideram, durante a compressão, a capacidade do ouvido humano e do cérebro de perceber um sinal. Um CODEC AAC, por exemplo, elimina de um sinal de áudio, durante a compressão, freqüências que são mascaradas por outras. O tratamento do sinal realizado por um CODEC emprega diferentes algoritmos DSP para permitir a sua conversão do domínio do tempo, como este foi amostrado e digitalizado, para o domínio da frequência, no qual será compactado. A compactação de um sinal digital é realizada com este representado no domínio da frequência, pois desta forma fica evidente para o CODEC quais partes do sinal são similares e podem ser compactadas e quais podem ser eliminadas sem que nossos ouvidos, ou nosso cérebro, percebam.

Os algoritmos de DSP utilizados para transformar um sinal do domínio do tempo para o domínio da frequência variam de acordo com o sinal a ser processado. Sinais de vídeo normalmente utilizam a Transformada Discreta de Cosseno (DCT) para transformação, já os sinais de áudio utilizam uma versão modificada desta, a Transformada de Cosseno Modificada (MDCT), que pode ser implementada com um algoritmo de FFT. Estes algoritmos de DSP exigem muito processamento da plataforma em que executam e, em razão disto, algumas construções em hardware são utilizadas para uma implementação eficiente.

Estas construções de hardware específicas, como o Multiplicador Acumulador (MAC) e o *Barrel Shifter*, foram introduzidas inicialmente em processadores DSPs utilizados em aplicações embarcadas. Posteriormente, algumas construções provaram-se indispensáveis para aplicações gráficas que demandam processamento vetorial, como múltiplas unidades lógico-aritméticas para suportar instruções SIMD, e foram adicionadas também em GPPs. Como exemplo, podemos citar conjuntos de instruções multimídia, como o *Multimedia Extension* (MMX) [INT 99] da Intel, 3DNow! [AMD 00] da AMD e o *Streaming SIMD Extensions* (SSE) [INT 09a] e seus variantes (SSE2, SSE3, e SSE4) também da Intel.

É comum que aplicações multimídia sejam desenvolvidas em linguagens de alto nível, assim como CODECs utilizados na sua composição. Entretanto, é frequente que o código gerado pelo compilador não faça uso de toda capacidade presente na arquitetura alvo. Por esta razão, o projeto de aplicações multimídia para sistemas embarcados requer que a otimização de partes críticas do código seja feita em linguagem *Assembly* da arquitetura alvo. Desta maneira, o desenvolvedor tem acesso às construções de hardware disponíveis nesta arquitetura para acelerar loops que são críticos ao desempenho da aplicação.

Como mostra a seção 4.4, o algoritmo Transformada Inversa de Cosseno Modificada (IMDCT), utilizado no decodificador AAC, é responsável por aproximadamente um terço do tempo de execução do decodificador, sendo portanto, candidato a otimizações em linguagem de baixo nível. Contudo, como já citado na seção 2, quando as otimizações para uma determinada arquitetura são realizadas no código fonte da aplicação, sua portabilidade para outras arquiteturas é dificultado, pois esta torna-se dependente da plataforma alvo.

A EMCA permite ao desenvolvedor acesso direto às construções de hardware encontradas em arquiteturas embarcadas, através do uso de mediadores de hardware, como apresentado na seção 3.3. A interface da EMCA provê algoritmos de DSP para a aplicação, os quais fazem uso dos mediadores de hardware na sua implementação. Estes algoritmos podem ser especializados para diferentes arquiteturas, sem que a aplicação seja modificada, utilizando as extensões de hardware existentes em cada uma delas. A interface da EMCA para algoritmos DSP é apresentada na seção 3.4.

3.2 Desenvolvimento de Sistemas Embarcados Direcionado pela Aplicação

A metodologia ADESD provê meios para identificação e modelagem de componentes de software pertencentes ao sistema através de um processo de engenharia de domínio. Entidades identificadas em um domínio são organizadas em famílias de abstrações, seguindo a análise de variabilidade presente no *Projeto baseado em famílias*, proposto por Parnas em [PAR 76].

Abstrações que incorporam detalhes do ambiente em que se encontram tem pequenas chances de serem reusadas em cenários diferentes. A dependência de hardware é reduzida em ADESD utilizando o conceito de separação de aspectos proposto por Gregor Kiczales em [KIC 97], no processo de decomposição do domínio. Este conceito possibilita a identificação de variações de cenário, que ao invés de serem modeladas como novos membros de uma família, define um aspecto do cenário. ADESD propõe um processo de engenharia de domínio que modela componentes de software utilizando principalmente três construções: famílias de abstrações independentes de cenários, adaptadores de cenários e interfaces infladas.

Famílias de abstrações independentes de cenário são identificadas durante a fase de decomposição do domínio. Abstrações são identificadas a partir de entidades significativas do domínio e agrupadas em famílias,

de acordo com suas características comuns.

Adaptadores de cenário são utilizados para resolver as dependências de cenário [FRÖ 00]. Estas devem ser identificadas como aspectos durante a decomposição do domínio, mantendo as abstrações independentes de cenários. Os adaptadores de cenários são utilizados para aplicar os aspectos de cenários nas abstrações de maneira transparente.

Interfaces Infladas possuem as funções de todos os membros de uma família, resultando em uma visão única da família como se esta fosse um "super-componente". Isto possibilita que o desenvolvedor da aplicação escreva a aplicação com base em uma interface concisa e bem conhecida, adiando a decisão de qual membro da família deve ser usado até o momento em que o sistema é gerado. Esse membro será então agregado ao sistema em tempo de compilação.

Para possibilitar que os componentes e o sistema operacional sejam portáveis para a maioria das arquiteturas, um sistema projetado de acordo com ADESD utiliza mediadores de hardware [POL 04]. A idéia principal deste artefato de portabilidade é manter um *contrato de interface* entre o sistema operacional e o hardware. Cada componente de hardware é acessado através de seu próprio mediador, o que possibilita a portabilidade das abstrações que o usam, sem criar dependências desnecessárias. Mediadores são metaprogramados estaticamente e se "dissolvem" nas abstrações do sistema assim que o contrato de interface é firmado. Em outras palavras, um mediador de hardware provê as funcionalidades do componente de hardware correspondente através de uma interface orientada ao sistema.

3.3 Interface da EMCA para acesso ao hardware

Mediadores de hardware [POL 04] são utilizados na EMCA para prover uma interface de acesso aos recursos de hardware presentes na plataforma alvo. Os mediadores de hardware foram criados para fornecer uma interface de acesso ao hardware independente de plataforma em sistemas desenvolvidos utilizando a metodologia ADESD [FRÖ 01]. Porém, a funcionalidade dos mediadores de hardware pode ser estendida para permitir uma uniformização de comportamento a componentes de hardware distintos, como apresentado em [MAR 09].

A EMCA emprega mediadores de hardware para fornecer uma interface de acesso a componentes em hardware como MAC, independente deste próprio hardware, e para prover uma implementação em software para arquiteturas que não possuem o respectivo componente. A partir da análise de alguns processadores DSP, dois principais mediadores, que

proveem funções frequentemente utilizadas no núcleo de algoritmos DSP, foram identificados: o MAC e o *Barrel Shifter*.

O MAC é uma operação básica frequente em algoritmos de DSP, como filtros digitais e transformadas [HAY 04]. Uma implementação eficiente desta operação é fundamental para o bom desempenho de aplicações multimídia e, para tanto, processadores DSP possuem características arquiteturais que aceleram operações de MAC [KUU 00]. A semântica da operação MAC é mostrada na equação 3.1.

$$acc = acc + X * Y \tag{3.1}$$

A operação MAC consiste da multiplicação de dois números seguida da soma destes com o valor de um acumulador. Para otimizar esta operação, processadores DSP, e também alguns GPPs, possuem implementações de MAC em hardware, ou seja, um circuito lógico combinacional capaz de realizar uma multiplicação seguida de adição em um ciclo de *clock*. Para permitir que a cada ciclo uma operação MAC seja realizada, processadores modernos possuem dois barramentos de dados separados, além de um barramento de instruções. Esta disposição de memória é conhecida como Arquitetura de Harvard Modificada [HAY 04], conforme mostrado na Figura 3.1. No caso da equação 3.1 é possível realizar a cada ciclo uma operação de MAC, desde que os dados X e Y estejam armazenados em diferentes barramentos de dados, podendo ser acessados ao mesmo tempo.

Na arquitetura de Harvard convencional a memória de programa é separada da memória de dados, inclusive os barramentos de acesso são diferentes, permitindo que no mesmo estágio de execução do processador (*pipeline*), as duas possam ser acessadas. Esta organização de memória pode ser vista na Figura 3.1 (b). Na arquitetura de Harvard modificada, exibida na Figura 3.1 (c), são utilizadas duas memória de dados, permitindo que uma instrução possa acessar dois operandos em memória no mesmo estágio de execução além de acessar o barramento de instruções. A Figura 3.1 (a) mostra a arquitetura de memória von Neumann, tradicionalmente utilizada em microcontroladores e alguns processadores de uso geral.

Além da arquitetura de memória Harvard modificada, duas outras características arquiteturais estão presentes em processadores DSP que permitem implementações eficientes do MACs. Loops em hardware permitem que uma ou mais instruções sejam repetidas um número determinado de vezes sem que instruções de teste sejam executadas. A separação da unidade aritmética de geração de endereços da unidade arimética de dados permite que em um mesmo ciclo de *clock* ponteiros para dados em

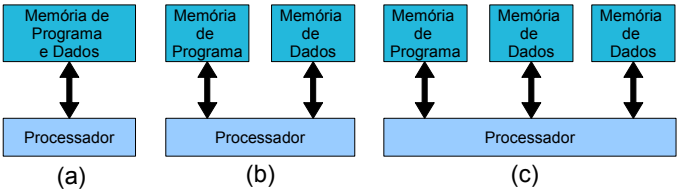

Figura 3.1: Arquiteturas quanto a disposição da memória. (a) von Neumann (b) Harvard (c) Harvard Modificada

memória sejam atualizados e operações aritméticas sobre os dados sejam realizadas. Devido a estas características, é possível que uma instrução MAC, no interior de um loop, seja otimizada para executar a cada ciclo, operando com dados diferentes até que todos os dados sejam processados. A operação de MAC está presente no núcleo de diversos algoritmos, como multiplicação de matrizes, cálculo de produto escalar, filtros de resposta finita, transformada discreta de cosseno, transformada de Fourier, etc.

A EMCA possui uma interface de acesso a MAC, **`E_MAC`**, mostrada na Figura 3.2 [1].

Em arquiteturas que possuem o componente de hardware MAC, a implementação do mediador **`E_MAC`** utiliza instruções *Assembly* da arquitetura alvo. A implementação dos mediadores de hardware utiliza meta-programação estática, ou seja, *templates* de C++. As chamadas para o método **`Perform`** da classe derivada de **`E_MAC`** são traduzidas para o código *Assembly* que realiza a operação MAC com o tipo de dado passado como argumento. Caso não exista uma implementação para um tipo de dado específico, uma implementação genérica, de alto nível, é utilizada. Para arquiteturas sem MAC em hardware uma implementação em software, disponibilizada pela classe **`MAC`** presente na Figura 3.2, deste mediador está disponível.

O método **`Perform`**, presente na interface dos mediadores de hardware, pode ser sobrecarregado, possibilitando diferentes implementações para diferentes tipos de dados. Desta maneira, uma operação utilizando dados inteiros, pode tirar máximo proveito da existência de um componente MAC otimizado para inteiros na arquitetura alvo. A Figura 3.3 mostra uma implementação da interface **`E_MAC`**, **`BF_MAC`**, que possui uma implementação especializada do método **`Perform`** para alguns tipos

[1] Todas as interfaces da EMCA, mediadores ou algoritmos DSP, possuem o prefixo "**`E_`**".

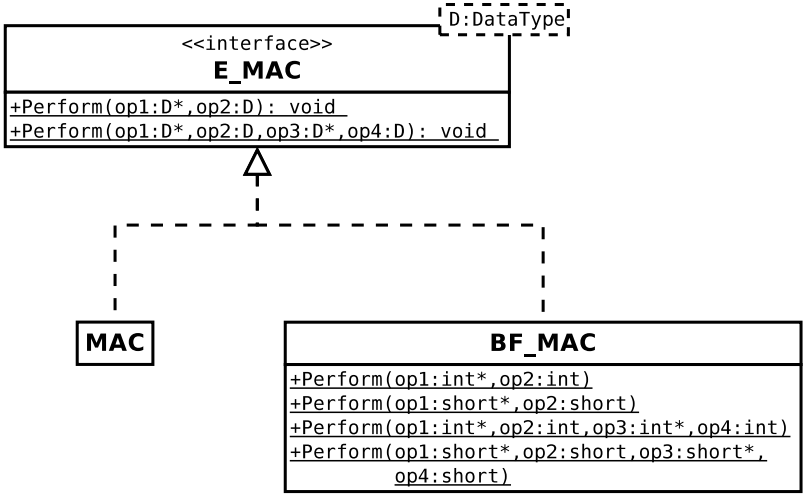

Figura 3.2: Interface do mediador de hardware **`E_MAC`** e suas implementações

de dados otimizados para arquitetura Blackfin. O método **`Perform`** é declarado *inline*, evitando a geração de código de chamada de função, como pode ser visto na Figura 3.4.

A parte superior da Figura 3.3 mostra a implementação do método **`Perform`** do mediador de hardware **`BF_MAC`**. A implementação deste método, em linguagem *Assembly* da arquitetura Blackfin, faz uso do componente MAC em hardware presente nesta arquitetura. A utilização do MAC acontece devido ao uso do acumulador **`A0`** na soma com o resultado da multiplicação. A parte inferior da Figura 3.3 mostra uma parte do código de uma aplicação que realiza a multiplicação seguida de soma dos dados de um vetor pelo seu elemento seguinte. A Figura 3.4 mostra parte do código da aplicação que executa o método **`Perform`**.

Como pode ser visto na Figura 3.4 não são realizadas chamadas de função para o método **`Perform`** pois este foi declarado como *inline*. Esta abordagem resulta em uma implementação eficiente, utilizando o componente de hardware MAC presente na plataforma e sem o sobrecusto de chamadas de função.

Assim como a arquitetura Blackfin que possui duas unidades MAC de 16 bits paralelas, outros processadores também podem prover meios para realização de mais de uma operação MAC simultaneamente. Para tanto, o método **`Perform`** possui uma definição com quatro operandos

```
// file : BF_MAC.h
inline void BF_MAC::Perform (short* op1, short op2) {

    __asm__(
    "I0=%1;\n\t"
    "R0=[I0];\n\t"
    "R1=%2;\n\t"
    "A0=R0;\n\t"
    "A0+=R0.L*R1.L;\n\t"
    "R2=A0;\n\t"
    "%0=R2>>1;\n\t"
    : "=&d"(*op1)
    : "a"(op1), "a"(op2)
    : "I0", "A0", "R0", "R1", "R2");
}

// file : mac_test.cc
print_array(data, N);

for (i = 0; i < N; i++)
    E_MAC::Perform(&data[i], data[(i+1)%N]);

print_array(data, N);
```

Figura 3.3: Implementação inline do método **`Perform`** do mediador **`BF_MAC`** e sua chamada na aplicação

como argumentos, permitindo a implementação de duas operações MAC paralelamente. Para manter a portabilidade para arquiteturas sem múltiplos MACs uma implementação genérica é disponibilizada, onde as operações são realizadas em seqüência. Mais detalhes da arquitetura Blackfin estão disponíveis na seção 3.5 e na seção 4.3.

O conceito utilizado no mediador de hardware **`E_MAC`** é estendido para o *Barrel Shifter*. O *Barrel Shifter* é um circuito digital presente em algumas arquiteturas de processadores, que permite fazer o deslocamento (*shift*) de um dado, por um número especificado de bits, em um ciclo de *clock*. Assim como operações MAC, operações de *shift* são frequentes em algoritmos DSP [HAY 04]. A Figura 3.5 mostra a interface **`E_Shifter`** e duas implementações desta: para arquiteturas sem *Barrel Shifter* em hardware, e para arquitetura Blackfin.

Assim como a interface do mediador de hardware **`E_MAC`**, a interface **`E_Shifter`** também foi especializada para a arquitetura Blackfin com a implementação **`BF_Shifter`**. Esta implementação provê métodos otimizados para realização de *shifts* lógicos e aritméticos em dados **`int`** e **`short`** no Blackfin.

```
// file : mac_test.asm
ca0: CALL 0xb98 <__init+0x3cc>; // print
ca4: P5 = R0;
ca6: P1 = R6;
ca8: R0 = 0x0 (X);
caa: P0 = 0x8 (X);
cac: R3 = R0;
cae: R3 += 0x1;
cb0: R0 = R3 & R7;
cb2: P2 = R0;
cb4: P3 = R6;
cb6: P2 = P3 + (P2 << 0x1);
cb8: R2 = W[P2] (X);
cba: P2 = R2;
cbc:    I0  = P1;
cbe:    R0 = [I0 ];
cc0:    R1 = P2;
cc2:    A0 = R0;
cc6:    A0 += R0.L * R1.L;
cca:    R2 = A0;
cce:    R5 = R2 >> 0x1;
cd2: W[P1] = R5;
cd4: P0 += -0x1;
cd6: CC = P0 == 0x0;
cd8: IF  !CC JUMP 0xe58 <_main+0x258>;
cda: P3 = P4;
cdc: CALL 0xb98 <__init+0x3cc>; // print
 ...
e58: R0 = R3;
e5a: P1 += 0x2;
e5c: JUMP.S 0xcac <_main+0xac>;
.
```

Figura 3.4: Parte do código *Assembly* gerado a partir da aplicação da Figura 3.3

O conceito de mediador de hardware, como utilizado na EMCA, pode ser estendido para diferentes construções de hardware, desde que estas não alterem o fluxo de execução do programa, como estruturas de controle de loop em hardware (*zero-overhead* loop). O suporte a este tipo de estrutura é realizado pelas interfaces de algoritmos DSP de forma natural. A especialização das interfaces DSP da EMCA pode utilizar bibliotecas específicas para implementação de rotinas utilizando loops em hardware como apresentado na seção 3.5.

A interface da EMCA para algoritmos DSP, apresentada na seção 3.4, provê acesso para o algoritmo de FFT que faz uso da interface **`E_MAC`** na sua implementação. Esta abordagem em dois níveis, separando o algoritmo do hardware utilizado, permite o uso de uma mesma implementação de algoritmo em diferentes arquiteturas, sem que seja necessário

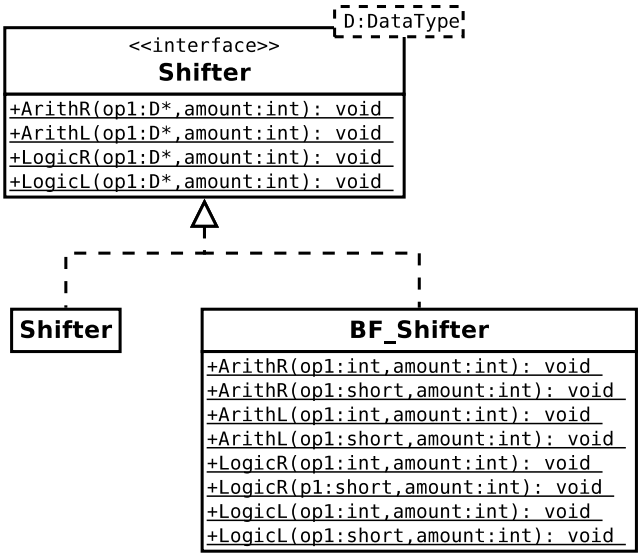

Figura 3.5: Interface do mediador de hardware **`E_Shifter`** e suas implementações

reescrever o código. Por outro lado, esta abordagem não impede que o desenvolvedor reescreva o código de um algoritmo quando o seu desempenho pode ser melhorado para uma arquitetura específica. Mesmo com otimizações para arquiteturas específicas, o "contrato" da aplicação com a interface da EMCA permite sua especialização para diferentes plataformas sem alterar o código da aplicação.

3.4 Interface da EMCA para algoritmos DSP

O objetivo da EMCA é prover ao desenvolvedor uma interface para algoritmos DSP, utilizados em aplicações multimídia, portável para diferentes plataformas. Para permitir uma implementação eficiente destes algoritmos para diferentes arquiteturas, o acesso às construções de hardware, dependente de cada arquitetura, foi encapsulado em mediadores. Assim, para especializar a EMCA para uma nova arquitetura, somente é necessário que o desenvolvedor forneça novas implementações para os mediadores. Porém, o isolamento das interfaces de algoritmos também permite que implementações especializadas sejam possíveis.

Como explicado na seção 3.1, transformadas são algoritmos DSP

empregados na implementação de CODECs, utilizados em aplicações multimídia. Os algoritmos de transformadas são utilizados em dados digitalizados, transformando amostras de um sinal do domínio do tempo para o domínio da frequência e vice-versa.

Um dos principais algoritmos de DSP é a Transformada Discreta de Fourier (DFT), que é um procedimento matemático utilizado para determinar o conteúdo harmônico, ou as frequências, presentes em uma seqüência discreta de um sinal. Uma seqüência discreta é um conjunto de dados obtidos através de amostragem de um sinal contínuo no tempo [LYO 04]. Por exemplo, uma seqüência discreta pode ser obtida pela digitalização de um sinal de áudio. Em 1965, Cooley e Turkey [COO 65] publicaram um algoritmo para o cálculo da DFT que é amplamente difundido e conhecido como Transformada Rápida de Fourier (FFT). Este algoritmo diminui consideravelmente o número de operações matemáticas quando comparado à DFT.

A Figura 3.6 mostra a interface **`E_FFT`** da EMCA para as operações de FFT e Transformada Rápida Inversa de Fourier (IFFT). Como veremos na seção 4.2, a FFT pode ser utilizada na implementação da MDCT, que por sua vez é empregada na decodificação de áudio MP3 e AAC. Esta interface é similar a proposta por Sergey Chernenko em [CHE 09].

A propriedade de inversão é intrínseca às transformadas matemáticas, e por isso a EMCA possui uma interface única para transformadas com os métodos **`Forward`** e **`Inverse`**, como mostra a Figura 3.6. Duas versões destes dois métodos são disponibilizadas, uma prevê o uso de dois conjuntos de dados separados, sendo um o conjunto de dados de entrada **`Input`** e o outro a saída **`Output`**. E a outra versão possui apenas um conjunto de dados, **`Data`**, como argumento destes métodos que seria utilizado como entrada e saída, ou seja, os dados de entrada seriam perdidos.

A classe **`BF_FFT`** mostrada na Figura 3.6 é a implementação da interface **`E_FFT`** para a arquitetura Blackfin. A implementação **`BF_FFT`** define o método **`InitTwiddleTable`**, não presente na implementação **`CFFT`**, é necessário para a implementação **`BF_FFT`**, pois o algoritmo de FFT utilizado prevê o uso de tabelas de ajuste (*Twiddle Tables*), como explicado na seção 3.5.

A Figura 3.6 também mostra a interface **`E_Complex`**, que é o tipo de dado manipulado na **`E_FFT`**. A interface **`E_Complex`** também foi especializada, provendo uma implementação mais eficiente para a arquitetura Blackfin utilizando ponto flutuante, classe **`BF_Complex`**. Também é disponibilizada uma implementação que utiliza o tipo de dado **`complex_fract16`**, introduzido na seção 3.5, onde são mostradas as otimizações realizadas com o objetivo de melhorar o desempenho da para

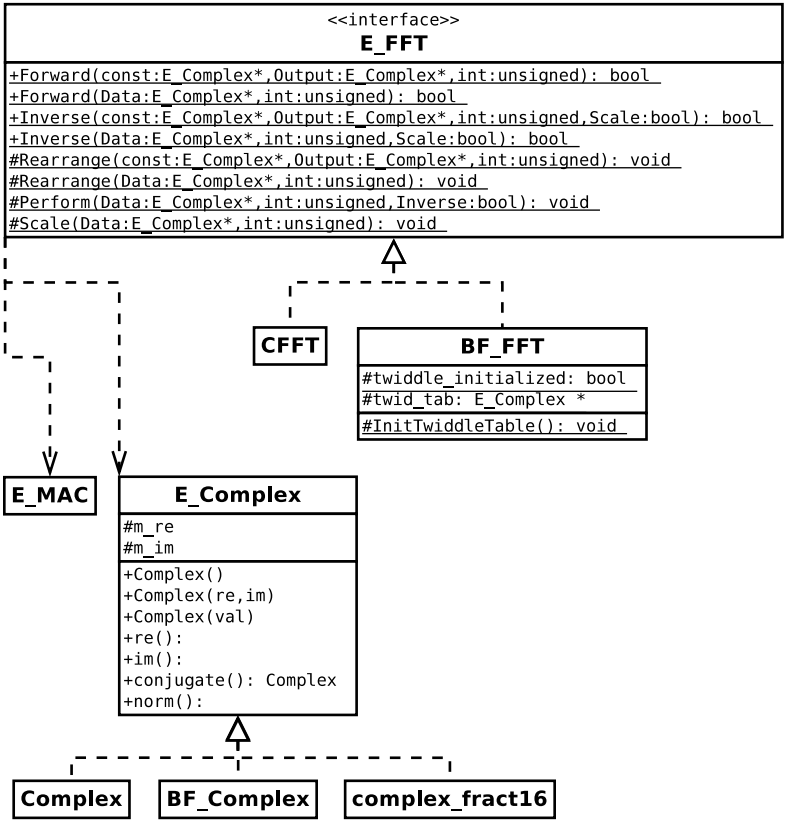

Figura 3.6: Interface **`E_FFT`** e as implementações **`CFFT`** em C++, e **`BF_FFT`** especializada para arquitetura Blackfin

a arquitetura Blackfin.

3.5 EMCA para Blackfin

Blackfin [DEV 08] é uma família de microprocessadores RISC de 32 bits desenvolvida em conjunto pela Intel e Analog Devices e comercializado pela Analog Devices. O microprocessador utilizado neste trabalho para estudo de caso da EMCA, ver seção 4, foi o BF-537, presente na plataforma BF537-STAMP. O Blackfin utiliza a *Micro Signal Architecture* (MSA), que emprega aspectos do DSP SHARC, desenvolvido e

comercializado pela Analog Devices, e do processador Xscale, implementação de microcontroladores compostos por *cores* ARM, desenvolvidos pela Intel e atualmente comercializados pela Marvell.

O Blackfin possui arquitetura de memória Harvard modificada, com dois barramentos para caches de dados e um barramento para cache de instruções, todos com acesso simultâneo. Este microprocessador possui ainda uma memória *scratchpad* de 4KB que, assim como as memórias cache, funcionam no mesmo *clock* do processador, que é 500MHz na plataforma BF537-STAMP. O BF537 possui duas unidades MAC de 16 bits, com respectivos acumuladores de 40 bits, um *Barrel Shifter* de 40 bits e unidade aritmética de geração de endereços separada da unidade lógico aritmética de dados. O diagrama com as unidades computacionais do Blackfin, e dos registradores disponíveis, pode ser visto na Figura 3.7.

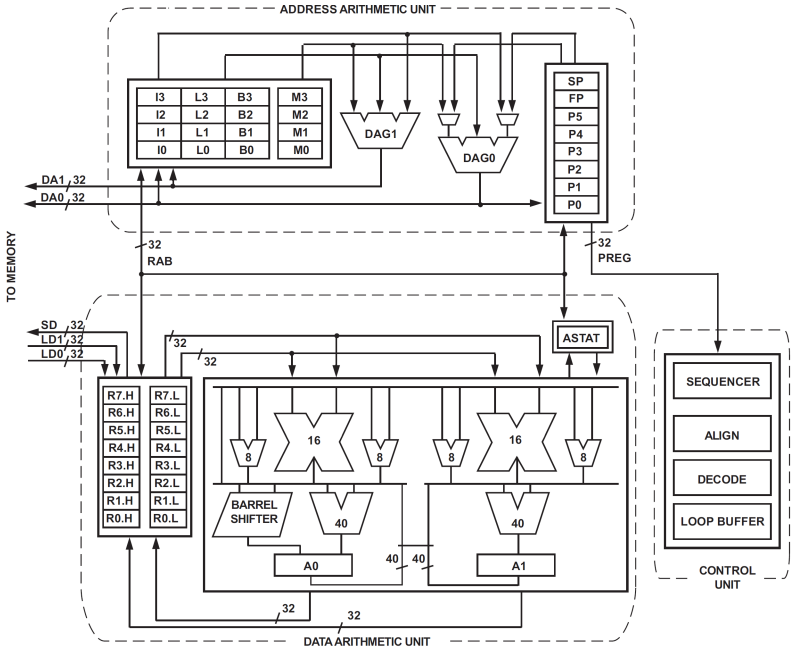

Figura 3.7: Arquitetura do núcleo do processador Blackfin [DEV 08]

Os registradores **`In`** presentes na unidade aritmética de endereços são otimizados para uso como índice de *arrays*, e os registradores **`Ln`** são otimizados para controle de loops em hardware. O controle de loop em

hardware, característico de processadores DSP [HAY 04], evita o uso de uma instrução de comparação no fim da execução de um loop. Ao invés disso, um registrador específico é setado com o número de vezes que um determinado loop deve ser executado, e o controle é realizado pelo hardware. O uso dos registradores **In** como índices permite a incrementação destes em paralelo com a execução de até duas instruções MAC, executadas na unidade aritmética de dados.

A arquitetura Blackfin não possui Unidade Ponto Flutuante (FPU), portanto, as operações com números ponto-flutuante são emuladas em software. Como mostrado na seção 4.4, o uso de operações ponto-flutuante em arquiteturas ponto-fixo, como o Blackfin, resulta em queda de desempenho considerável. A implementação genérica da interface **E_FFT**, classe **CFFT** na Figura 3.6, utiliza operações em ponto-flutuante e, tendo em vista uma implementação mais eficiente desta interface para a arquitetura Blackfin, foi criada a implementação **BF_FFT**.

O algoritmo de FFT utilizado na implementação **BF_FFT** é diferente do utilizado na implementação **CFFT**, porém os resultados são similares. Os métodos **Forward** e **Inverse** da **BF_FFT** invocam, respectivamente, os métodos **cfft_fr16** e **ifft_fr16** disponibilizados na **libbfdsp** [ANA 09].

A **libbfdsp** é uma biblioteca de funções DSP otimizadas para arquitetura Blackfin, distribuída como parte do conjunto de ferramentas *Gnu Compilers Collection* (GCC) para esta arquitetura. Os métodos **cfft_fr16** e **ifft_fr16**, presentes na **libbfdsp**, requerem o uso de uma *twiddle table*, ou seja, uma tabela trigonométrica consultada diversas vezes durante a execução do algoritmo de FFT. A *twiddle table* é gerada somente uma vez durante a execução da aplicação, e utilizada diversas vezes para diferentes tamanhos de blocos de coeficientes. Esta tabela é criada na primeira execução de um dos métodos **Forward** ou **Inverse** e armazenada em um atributo estático da classe **BF_FFT**.

A implementação da função **ifft_fr16** foi criada pela própria Analog Devices para a arquitetura Blackfin. Esta implementação, realizada em linguagem Assembly, utiliza os dois MACs disponíveis na arquitetura paralelamente para otimizar o algoritmo, sendo o controle do loop interno do mesmo feito com os registradores especiais de controle de loop em hardware. É possível alocar os dados de entrada e saída do algoritmo em memórias de dados diferentes, permitindo aceleração máxima deste algoritmo nesta arquitetura.

A implementação de FFT provida na **libbfdsp** utiliza o tipo de dado **complex_fract16**, também otimizado para a arquitetura Blackfin. Este tipo de dado é uma representação de números complexos no

formato fracional 1.15, onde um bit representa a parte inteira do número e 15 bits a parte fracional. Os números representados em **`fract16`** variam de -1 a 0.999969. Esta representação é suficiente para grande parte das aplicações que utilizam a FFT, e funções de conversão para **`float`** são disponibilizadas. A interface **`E_Complex`** foi implementada utilizando **`complex_fract16`** para a arquitetura Blackfin.

A implementação da interface **`E_FFT`** para a arquitetura Blackfin não faz uso de mediadores de hardware. O algoritmo utilizado, já otimizado para arquitetura, foi implementado levando em consideração as unidades computacionais disponíveis nesta. Como citado anteriormente, a organização em camadas, utilizada na EMCA, permite este tipo de implementação.

O uso de rotinas criadas pelo fabricante otimizadas para uma determinada plataforma, como as presente na **`libbfdsp`**, frequentemente superam implementações de terceiros no que diz respeito ao desempenho. Um exemplo disto é mostrado na seção 4.4. Com o uso da EMCA, é possível empregar estas implementações otimizadas para arquiteturas específicas na aplicação sem que esta deixe de ser portável para outras arquiteturas.

3.6 Utilização e aplicabilidade da EMCA

A EMCA estabelece uma interface única para algoritmos DSP utilizados na implementação de CODECs utilizados em aplicações multimídia. Além disto, são estabelecidas interfaces para uso de componentes de hardware frequentemente utilizados em algoritmos DSP existentes em arquiteturas utilizadas neste tipo de aplicação. A interface proposta independe da plataforma alvo, permitindo desta maneira a portabilidade eficiente de algoritmos utilizados na implementação de CODECs e consequentemente de aplicações multimídia.

Para utilização da EMCA em uma aplicação, é necessária a inclusão do *header* **`EMCA.h`**, onde são definidas quais implementações são utilizadas para a arquitetura alvo da compilação. Além disso, é imprescindível a inclusão dos arquivos objetos (**.o*) referentes a implementação da EMCA para a plataforma alvo na lista de objetos a serem ligados para geração do executável. Caso sejam realizadas chamadas a rotinas presentes em bibliotecas disponibilizadas pelo fabricante da arquitetura alvo, tais bibliotecas também devem ser incluídas na ligação e geração do executável. Como feito na especialização da EMCA para a arquitetura Blackfin onde foi utilizada a **`libbfdsp`**.

Os algoritmos de DSP são comuns a outras áreas e portanto esta

API pode ser utilizada no suporte de outras aplicações embarcadas. A FFT, por exemplo, pode ser empregada em diversas aplicações, como por exemplo, na área de comunicação, processamento de imagens, radares, etc.

Aplicações que realizam multiplicação de matrizes podem ser implementadas utilizando o mediador MAC, melhorando o seu desempenho em arquiteturas com suporte em hardware. Operações com números ponto flutuante, que requerem ajuste da mantissa, podem utilizar o mediador *Barrel Shifter* para favorecer o desempenho quando este estiver presente no hardware da plataforma utilizada. Nestes dois casos, a implementação em software nas plataformas sem suporte em hardware terá desempenho similar a uma implementação que não utiliza a EMCA. Porém, os algoritmos presentes na EMCA terão seu desempenho favorecido em plataformas com componentes de hardware relativos aos mediadores.

A Figura 3.8 mostra as camadas que a EMCA exporta para a aplicação. Para permitir a portabilidade de algoritmos DSP, CODECs devem fazer uso da EMCA na sua implementação. Porém, é possível que a aplicação acesse direto a interface de algoritmo DSP, ou até mesmo os mediadores.

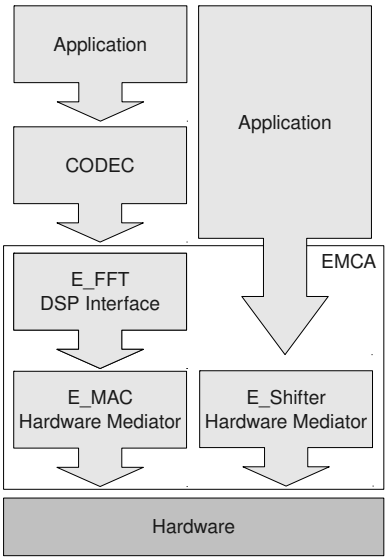

Figura 3.8: Disposição das interfaces exportadas pela EMCA para a aplicação.

A interface dos mediadores de hardware exportada pela EMCA é a mesma, independentemente de plataforma, assim como a interface dos algoritmos DSP suportados.

Capítulo 4

Estudo de caso: EMCA utilizada no FAAD2

O objetivo deste estudo de caso é avaliar o esforço necessário para tornar uma aplicação já existente compatível com a EMCA e analisar as alterações decorrentes do uso da API no seu desempenho. É analisado o impacto do uso da interface **`E_FFT`**, proposta no capítulo 3, no decodificador de áudio *Freeware Advanced Audio Decoder 2* (FAAD2) em diferentes plataformas.

A escolha de um decodificador de áudio para estudo de caso da EMCA pretende legitimar sua funcionalidade em plataformas com diferentes características de hardware. Um decodificador de áudio AAC faz uso da IMDCT na implementação de bancos de filtros (*filterbanks*).

Uma implementação eficiente de IMDCT pode ser obtida utilizando o algoritmo de IFFT, como descrito em [DUH 91]. A implementação citada, considerada ótima para a aplicação de decodificação de áudio, é utilizada no FAAD2. A interface de IFFT do FAAD2 foi substituída por chamadas à interface **`E_FFT`** definida pela EMCA. A utilização da interface **`E_FFT`** da EMCA em um decodificador de áudio justifica-se pela sua aplicabilidade em diferentes plataformas, fazendo uso de características de hardware existentes.

A seguir, na seção 4.1, é mostrada a estrutura geral e uma breve explicação de cada bloco que compõe um decodificador AAC. Na seção 4.2, o decodificador FAAD2 é apresentado, e são listadas as alterações necessárias para torná-lo compatível com a EMCA. Em seguida, a seção 4.3 descreve o cenário de testes utilizado e a seção 4.4, mostra uma análise do impacto da EMCA no desempenho da decodificação de arquivos de áudio AAC *Low Complexity* (LC) utilizando o FAAD2.

4.1 Decodificador de áudio AAC

Codificação de Áudio Avançada (AAC) [MPE 97] é um padrão de áudio especificado no MPEG-2 [MPE 96] Parte 7, que foi posteriormente complementado no MPEG-4 [MPE 99] Parte 3, Subparte 4. O padrão definido pelo *Moving Pictures Experts Group* (MPEG) não especifica como o codificador ou o decodificador de áudio devem ser implementados, e sim o formato do *bitstream*, que é a saída do codificador e a entrada do decodificador. Diferentes perfis de áudio AAC, foram criados para diferentes aplicações. Os três perfis mais utilizados, especificados no MPEG-2 Parte 7, são *Main Profile*, *Low Complexity* (LC) e *Scalable Sampling Rate* (SSR).

O *Main Profile* é o perfil mais completo especificado pela MPEG-2 parte 7. Neste perfil, são obtidas as melhores taxas de compressão em diferentes frequências de amostragem e diferentes taxas de transmissão ou compressão (*bitrates*). Todos os blocos, características (*features*), e ferramentas são utilizadas neste perfil. O perfil LC não utiliza controle de ganho e predição, com o objetivo de reduzir a demanda de processamento, tornando-o adequado para plataformas com recursos computacionais escassos. O perfil SSR foi criado para ser simples, de menor complexidade computacional, adequado para quando o sinal a ser codificado possui frequências limitadas, como sinais de voz. O MPEG-4 Parte 3 define 10 perfis de áudio, alguns dos quais são similares aos três já citados, e outros estendem os perfis existentes com novas características. Os perfis AAC definidos no MPEG-4 não são relatados pois fogem do escopo deste trabalho. Devido à complexidade do padrão e das operações envolvidas, as implementações existentes utilizam os mesmos blocos de alto nível que formam o modelo de referência criado pelo grupo MPEG [MPE 01], como base do projeto do software do CODEC. Normalmente, um decodificador AAC para o perfil LC é composto pelos blocos mostrados na Figura 4.1.

O bloco formatador de *bitstream*, mostrado na Figura 4.1, desencapsula os dados da *stream* contidos em contêineres, e cria uma seqüência de blocos de dados de áudio AAC (*frames*). Áudio AAC pode ser distribuído em diferentes tipos de arquivo, contêineres, com diferentes finalidades.

Os dois tipos de arquivo mais comuns são *Audio Data Interchange Format* (ADIF), que possui apenas um cabeçalho seguido pelos blocos de áudio AAC, e *Audio Data Transport Stream* (ADTS), que consiste em uma série de *frames*, cada um com um cabeçalho e um bloco de áudio AAC, ambos definidos no padrão MPEG-2 parte 7. O padrão MPEG-4 parte 3 especifica mais dois formatos: *Low-overhead MPEG-4 Audio*

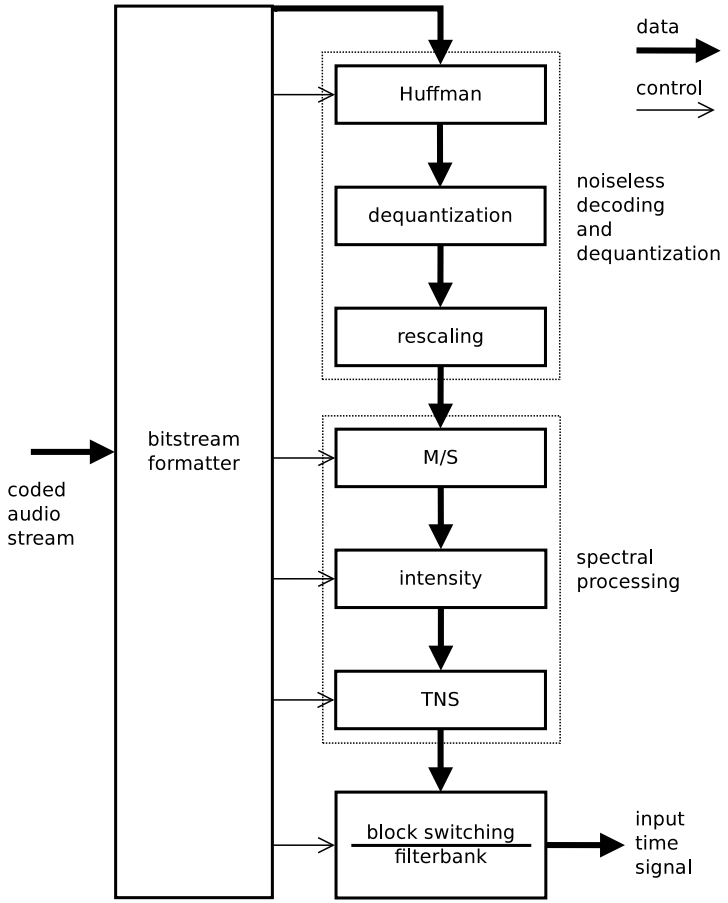

Figura 4.1: Blocos que formam um decodificador de áudio AAC perfil LC. Adaptado de [SOL 00].

Transport Multiplex (LATM), que permite combinar mais de um fluxo de áudio em um mesmo *bitstream*, e *Low Overhead Audio Stream* (LOAS), formato autossincronizante.

Depois de desempacotados, os dados de áudio passam pela descompactação sem perda (*noiseless decoding*) iniciada pelo algoritmo de Huffman Adaptativo [SOL 00]. Este é um tipo de algoritmo de compressão de entropia com tamanho de símbolo variável (*Variable length*

Code). Após a decodificação de entropia, é feita a quantização inversa (*dequantization*), onde os coeficientes utilizados na codificação AAC são recuperados. Codificadores AAC realizam um ajuste de escala nos coeficientes antes da quantização para evitar erros grosseiros. Portanto, na seqüência da quantização inversa, o decodificador realiza o ajuste de escala inverso (*rescaling*).

Durante a codificação AAC, os coeficientes são manipulados para buscar maior compressão do áudio. Esta fase, chamada de *spectral processing*, possui módulos capazes de remover dos dados de áudio, na forma de coeficientes, informações que não podem ser percebidas pelo ouvido humano. O módulo M/S (*Mid/Side Stereo Coding*) e o módulo de intensidade (*Intensity Stereo*) identificam e decodificam as similaridades presentes em sinais com canais estéreos.

O módulo *Temporal Noise Shaping* (TNS) existe para remover *preechoes* do sinal. *Preechoes* são comuns em sinais de áudio transitórios, com variações rápidas. Nestas situações, erros de quantização são comuns e são reduzidos com o uso de TNS.

O padrão AAC MPEG-2 Parte 7 especifica o uso de dois tamanhos para os blocos de coeficientes, 1024 e 128. Estes blocos são resultados da transformação de conjuntos de 2048 e 256 amostras de áudio (janelas). As janelas de 2048 amostras geram blocos com 1024 coeficientes e as janelas de 256 amostras geram blocos com 128 coeficientes. Esta característica é importante pois permite que sinais estáticos (ou sinais que variam pouco, chamados *tons*), sejam codificados em janelas maiores, permitindo uma maior correlação entre os coeficientes e uso de menos bits na sua representação codificada (*stream*). Já as janelas menores podem ser utilizadas em sinais com variações de frequência em curtos espaços de tempo.

Como mostrado na Figura 4.1, existe um módulo que faz a transição entre os diferentes tipos de blocos de dados (*block switching*). Isto é necessário visto que a transformação das amostras em coeficientes é feita com uma sobreposição de metade das amostras, ou seja, para janelas com 2048 amostras, somente metade do buffer de entrada é atualizado com amostras novas, as demais são as 1024 amostras mais recentes do bloco anterior copiadas (*overlapping*).

No módulo banco de filtros (*filterbank*), presente na Figura 4.1, é realizada a IMDCT no decodificador. Neste módulo, os coeficientes são transformados em amostras de áudio em formato bruto, áudio sem compressão, como por exemplo Modulação por Código de Pulso (PCM) e *Waveform Audio File Format* (WAV), parecidas com as captadas e codificadas. As amostras são parecidas, pois o processo de codificação e decodificação de áudio AAC pode introduzir ruídos no sinal original (*Lossy*

Encoding) e, portanto, o sinal decodificado não é idêntico ao sinal codificado.

Estudos realizados pelo grupo MPEG [QUA 99], utilizando o modelo de referência de implementação de um codificador AAC, mostram que aproximadamente 47% das instruções que formam um codificador AAC perfil LC fazem parte do bloco IMDCT, como mostra a tabela 4.1.

	1 Audio Chan	**5 Audio Chan**	
	# Instr	**# Instr**	%
Huffman	13657	68285	32.5
Inverse Quant. and Scale	1708	8540	4.1
M/S synthesis		1708	0.8
Coupling Channel		11546	5.5
TNS (average)	4065	20325	9.7
IMDCT	19968	99840	47.5
Total	**39398**	**210244**	**100.0**

Tabela 4.1: Resumo da complexidade das instruções que formam um codificador AAC LC [QUA 99]

Os dados na tabela 4.1 apontam que o bloco de IMDCT é o maior bloco do codificador AAC analisado. Para arquiteturas com restrições de memória, como sistemas embarcados, este bloco é o principal alvo de redução de tamanho de código. O estudo mencionado [QUA 99] avaliou a implementação de referência MPEG do codificador utilizando tipo de dado ponto flutuante em máquinas com arquiteturas 32 bits. O decodificador AAC não foi avaliado neste estudo, porém os dados mostram que uma otimização do algoritmo de IMDCT resulta em ganhos significativos para o CODEC em geral. Portanto, uma vez que o CODEC utilize a EMCA, que provê interface para este tipo de algoritmo (IMDCT utilizando a IFFT), a otimização desta para uma determinada arquitetura trará ganhos imediatos ao decodificador. O emprego da EMCA no decodificador FAAD2 é apresentado nas seções a seguir.

4.2 O decodificador FAAD2

O FAAD2 [BAK 00] é um decodificador de áudio AAC relativamente rápido, portável, escrito em linguagem C e de código fonte livre (licenciado sob a *GNU General Public License version 2.0* (GPLv2)). O

FAAD2 suporta o *Main Profile* e o perfil LC, especificados no MPEG-2 Parte 7, e os perfis definidos no MPEG-4 Parte3: *Long-Term Prediction* (LTP), *Spectral Band Replication* (SBR), *Low Delay* (LD), e possui também suporte a *Parametric Stereo* (PS). O FAAD2 inclui a ***libfaad***, utilizada em diversos softwares multimídia [FAA 09], como MPEG4ip, VideoLan Client (VLC), Winamp, MPLayer e outros.

A Figura 4.2 mostra algumas funções, e os seus respectivos módulos, executadas pelo decodificador FAAD2 na decodificação de um arquivo de áudio AAC LC. Nessa Figura, foi utilizado um diagrama de colaboração da *Unified Modeling Language* (UML) para representar o fluxo de execução interno do FAAD2. Os módulos, representados pelos nomes dos quadrados, são arquivos de código C que agrupam funções com finalidades similares. As mensagens, texto sobre as linhas, são as chamadas de funções entre os diferentes módulos. As setas mostram qual módulo executa a função (origem da seta) e em qual módulo esta função é definida e implementada (destino da seta).

O módulo **`syntax`**, mostrado na Figura 4.2, define funções para leitura do *bitstream* e separação do mesmo em *frames*. Já o módulo **`specrec`** define funções para agrupamento e reposicionamento de janelas (*Window Grouping*), para quantização inversa e aplicação de fatores de escala (*Scale Factors*) quando necessário.

O módulo **`output`** define funções para criação do arquivo de saída da decodificação, o FAAD2 suporta a geração dos formatos WAV e PCM. Enquanto o módulo **`pns`** define funções para realizar a substituição de ruído perceptivo (*Perceptual Noise Shaping* (PNS)). Esta característica foi adicionada na especificação AAC MPEG-4 parte 7 visando aumentar a eficiência de *bitrate* em aplicações que exigem baixo *bitrate* através da substituição de ruídos constantes nos sinais. Os demais módulos fazem parte da implementação de referência criada pelo grupo MPEG [MPE 01] e já foram introduzidos na seção 4.1.

A Figura 4.2 destaca dois módulos, **`mdct`** e **`cfft`**. Como mostra a seção a seguir, o módulo **`mdct`** foi alterado para a utilização da EMCA e o módulo **`cfft`** foi substituído.

4.2.1 Adaptação do FAAD2 para EMCA

Para tornar o FAAD2 compatível com a EMCA, este teve de ser adaptado. O módulo **`mdct`**, destacado na Figura 4.2, foi alterado para que, ao invés da chamada da função **`cfftb`** que executa a IFFT definida pelo FAAD2, seja chamada a função **`E_FFT::Inverse`** definida pela EMCA. A interface da EMCA foi introduzida na seção 3. O módulo

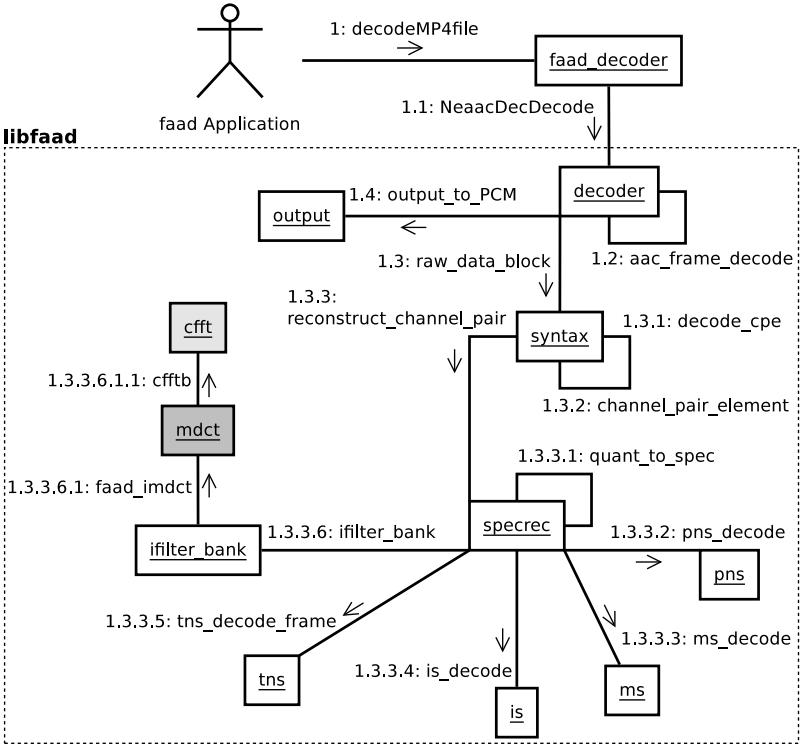

Figura 4.2: Visão geral dos módulos e funções do FAAD2 executadas na decodificação de arquivos AAC LC

`cfft` foi completamente removido da compilação quando utilizada a EMCA.

Para tornar possível a utilização da **`E_FFT`**, foi necessário converter os dados de entrada do módulo **`cfft`** para **`E_Complex`**, que é o tipo de dado processado pela **`E_FFT`**. A conversão de tipos foi implementada dentro do módulo **`mdct`** na função **`faad_imdct`** antes da chamada da **`E_FFT`**. Para tal, foi necessária a criação do *array* **`fft_inout_data`**, que posteriormente é inicializado com os dados a serem processados na chamada da função **`E_FFT::Inverse`**. Após realizada a operação de IFFT, os dados do *array* **`fft_inout_data`** são convertidos para o formato utilizado internamente no FAAD2.

Além da mudança do código da função **`faad_imdct`**, foi neces-

sária a adaptação dos **Makefiles** do FAAD2. O uso da EMCA requer a inclusão de arquivo cabeçalho com definições de tipos, onde são definidas as classes e tipos para a arquitetura-alvo. Porém, é necessário que os arquivos de código sejam compilados e ligados e, por isso, os **Makefiles** foram modificados.

4.3 Cenário de testes

Para validar o funcionamento da EMCA, esta foi empregada na implementação do FAAD2. A interface **E_FFT** substituiu a chamada de função **cfftb** existente, permitindo que o decodificador utilize o algoritmo IFFT presente na EMCA.

Com a finalidade de avaliar o impacto da EMCA no desempenho do decodificador, foram feitos testes de decodificação de arquivos de áudio AAC no perfil LC para arquivos WAV. O arquivo de áudio AAC usado foi uma música com variação de ritmo, com 5,011s de duração, dois canais, frequência de amostragem padrão de CDs de áudio, 44,1KHz. O decodificador FAAD2 foi utilizado *standalone*, e não na forma de *plugin*, com execução por linha de comando. Os testes foram realizados nas plataformas x86, e Blackfin.

O termo x86 designa uma família de conjuntos de instruções derivadas do processador 8086 da Intel [INT 09b]. Esta arquitetura está presente na maioria dos computadores desktops existentes e sempre que possível é a primeira plataforma de testes para novas aplicações. Para os testes realizados, foi utilizado um computador com processador Intel Pentium Dual-Core E5300 de 2,6GHz, rodando o Sistema Operacional (SO) Ubuntu Linux *kernel* 2.6.31-14. A versão do FAAD2 utilizada foi a 2.7, disponível em [BAK 00]. O compilador utilizado foi o g++ 4.4.1 [FSF 09b] com GNU Binutils 2.20 [FSF 09a].

Para os testes na arquitetura Blackfin, apresentada na seção 3.5, foi utilizado o SO μClinux [DEV 09] com *kernel* 2.6.31-6. O μClinux é uma versão de Linux para Microcontroladores sem Unidade de Gerenciamento de Memória (MMU). A versão do FAAD2 utilizada foi a 2.7, configurada com **-host=bfin-linux-uclibc**. O *cross-compiler* utilizado foi o bfin-linux-uclibc-g++ 4.1.2 com o GNU Binutils 2.17. Importante ressaltar que, para compilação nesta arquitetura, foram adicionadas as seguintes *flags* à compilação: **-lbffastfp** e **-lbfdsp**. Estas duas flags são passadas ao ligador para que este faça uso de bibliotecas específicas do compilador para o Blackfin. A flag **-lbffastfp** direciona o compilador a usar as rotinas de ponto flutuante otimizadas para esta arquitetura, uma vez que não existindo FPU, estas são emuladas em soft-

ware. A flag **`-lbfdsp`** permite a utilização da biblioteca de DSP existente para o Blackfin. Esta biblioteca possui a implementação de FFT utilizada na especialização **`BF_FFT`** da EMCA.

Tendo em vista que os SOs Linux e μClinux utilizados nos testes são multitarefa, a seguinte metodologia de teste foi adotada para reduzir o impacto do sistema sobre a avaliação de desempenho: cada decodificação foi executada dez vezes para cada configuração do decodificador em cada arquitetura. A execução mais rápida, com menor número de ciclos, e a execução mais lenta, com maior número de ciclos, foram desprezadas no cálculo da média do número de ciclos. Desta forma, o tempo de execução aqui apresentado para cada configuração é uma média de 8 execuções distintas.

4.4 Impacto do uso da EMCA no FAAD2

A tabela 4.2 mostra o tempo de execução médio, números em milhares de ciclos, de uma decodificação de um arquivo de áudio AAC LC de 5s de duração com FAAD2 na arquitetura x86. A coluna **`Kcyc FAAD`** mostra os números médios de ciclos de cada bloco do decodificador sem o uso da EMCA. A coluna **`Kcyc EMCA`** mostra os números médios de ciclos de cada bloco do decodificador FAAD2 com o uso da EMCA. O número médio total de ciclos do decodificador não é a soma exata das parciais apresentadas, pois alguns blocos como **`output`** não foram analisados por estarem fora do escopo deste estudo.

Como mostrado na tabela 4.2, o tempo médio de execução da decodificação aumentou 16% com o uso da EMCA (de 90 milhões para 105 milhões de ciclos). Os tempos médios dos blocos internos do decodificador mostram que o bloco mais impactado foi o banco de filtros inverso (*IFilterBank*) que teve um aumento médio de 51% no número de ciclos. Dentro do *IFilterBank* é executada a IMDCT e, consequentemente, a IFFT.

Como citado na seção 4.1, um decodificador AAC LC suporta dois tamanhos de janelas de coeficientes diferentes (janelas de 2048 amostras, que geram 1024 coeficientes, e de 256 amostras, que geram 128 coeficientes). O desempenho dos blocos IMDCT e IFFT foi avaliado para os dois casos isoladamente.

O desempenho da IMDCT foi parecido na média comparando-se as duas implementações do FAAD2, porém, o desempenho da IFFT foi bastante impactado pelo uso da EMCA. Para blocos de 128 coeficientes (IFFT[64]), o desempenho do decodificador que utiliza EMCA foi em média 178% pior que o FAAD2 original (2,7 vezes o número de ciclos). Para

	Kcyc FAAD	Kcyc EMCA	Relação EMCA/FAAD
Requant	9,458	9,987	1.05
Spectral Data	20,812	18,223	0.87
Scale Factors	2,944	2,843	0.96
IFilterBank	35,724	54,215	1.52
IMDCT[256]	668	677	1.01
IFFT[64]	783	2,179	2.78
IMDCT[2048]	9,228	9,111	0.99
IFFT[512]	16,941	27,691	1.63
Decoder total	**90,819**	**105,831**	**1.16**

Tabela 4.2: Tempo médio de decodificação de um arquivo de áudio AAC LC de 5s, utilizando diferentes versões do FAAD2, na arquitetura x86 em milhares de ciclos

blocos de 1024 coeficientes (IFFT[512]), o desempenho do decodificador que utiliza EMCA foi em média 63% pior que o FAAD2 original. Como é descrito na seção 4.5, estes resultados são coerentes, pois o algoritmo de IFFT original do FAAD2 é otimizado para este tipo de aplicação, e o algoritmo de IFFT da EMCA é genérico para qualquer aplicação, suportando qualquer tamanho de bloco.

A tabela 4.3 mostra o tempo de execução médio, em milhares de ciclos, de uma decodificação de um arquivo de áudio AAC LC de 5s de duração com FAAD2 na arquitetura Blackfin sem o uso da EMCA, coluna **`Kcyc FAAD`**, e com o uso da EMCA com implementação especializada para o Blackfin, coluna **`Kcyc EMCA_BF`**.

A implementação original do FAAD2 para arquiteturas ponto fixo utiliza tipo de dado **`int`** para representar e manipular coeficientes que são a entrada da IFFT, as operações de divisão e multiplicação sobre estes dados possuem suas implementações modificadas. Estas rotinas são isoladas e realizam operações em *Assembly*, definidas como Macros C, para manipular os coeficientes armazenados em dados inteiros. Esta implementação também está disponível em [BAK 00].

A segunda coluna da tabela 4.3, **`Kcyc EMCA_BF`**, mostra o tempo de execução do decodificador utilizando a especialização da EMCA para a arquitetura Blackfin. As características da implementação desta especialização da EMCA são descritas na seção 3.5. O tempo médio de decodificação total utilizando a **`EMCA_BF`** foi aproximadamente 8% inferior

	Kcyc FAAD	Kcyc EMCA_BF	Relação EMCA/FAAD
Requant	35,782	35,658	0.99
Spectral Data	36,836	36,395	0.99
Scale Factors	6,099	6,100	1
IFilterBank	149,344	122,877	0.82
IMDCT[256]	2,117	21,271	10.05
IFFT[64]	2,609	818	0.31
IMDCT[2048]	33,178	52,273	1.57
IFFT[512]	58,372	11,284	0.19
Decoder total	**322,581**	**296,679**	**0.92**

Tabela 4.3: Tempo médio de decodificação de um arquivo de áudio AAC LC de 5s, utilizando diferentes versões do FAAD2, na arquitetura Blackfin em milhares de ciclos.

(296 milhões contra 322 milhões de ciclos) quando comparado com a implementação original do FAAD2 para arquiteturas ponto fixo. A tabela 4.3 demonstra que a diferença de desempenho deve-se principalmente à mudança do desempenho do bloco banco de filtros (**`IFilterBank`**).

Como já explicado, o banco de filtros é o bloco onde é realizada a IMDCT no decodificador AAC. O tempo de execução médio do bloco de IMDCT foi bastante impactado com o uso da **`EMCA_BF`**. Para IMDCT em blocos de 256 coeficientes, o tempo de execução médio da versão com a **`EMCA_BF`** foi aproximadamente 10 vezes o tempo de execução médio da implementação original do FAAD2. Enquanto que para IMDCT em blocos de 2048 coeficientes, o tempo médio aumentou aproximadamente 1,7 vezes. Esta queda de desempenho se deve principalmente às conversões necessárias para utilização da **`EMCA_BF`**. A implementação de **`E_Complex`** utilizada **`EMCA_BF`** é a **`complex_fract16`**, disponível na biblioteca **`libbfdsp`** [ANA 09], que faz parte do conjunto de ferramentas GCC para arquitetura Blackfin. O FAAD2 utiliza o tipo **`int`**, 32 bits, para representação dos dados em implementações ponto fixo, portanto, para a utilização do **`complex_fract16`**, que utiliza representação 1.15 de números fracionais 16 bits, é necessário realizar um truncamento dos dados. Para amostras de áudio, bem como para seus coeficientes, este truncamento não acarretou em perda de precisão capaz de danificar o sinal, sendo esta conversão de tipo de dados saturada.

A tabela 4.3 mostra que a implementação de IFFT da **`EMCA_BF`** é

mais eficiente que a implementação ponto fixo original do FAAD2. Para blocos de 64 coeficientes, a **E_FFT** é em média 3 vezes mais rápida (818 mil contra 2.609 milhões de ciclos) que a implementação original do FAAD2 e, para blocos de 512 coeficientes, em média 5 vezes mais rápida (11 milhões contra 58 milhões de ciclos). A **EMCA_BF** utiliza a implementação de IFFT provida pela **libbfdsp** [ANA 09]. Esta implementação é acessada pelo método **ifft_fr16** que fornece uma interface para implementação *Assembly* da IFFT otimizada para a arquitetura Blackfin, como descrito na seção 3.5.

4.5 Análise dos resultados

As tabelas 4.2 e 4.3 mostram o impacto no desempenho do decodificador FAAD2 resultante do uso da EMCA. Ambas tabelas mostram o tempo da decodificação de um arquivo de áudio AAC no perfil LC com 5s de duração em milhares de ciclos.

A tabela 4.2 lista os tempos médios de execução de cada bloco do decodificador na arquitetura x86 com e sem o uso da EMCA. A implementação da EMCA utilizada neste teste emprega um algoritmo genérico em código C++ disponível em [CHE 09] na implementação da **E_FFT**. O impacto da EMCA no desempenho do decodificador na arquitetura x86 foi um acréscimo de aproximadamente 16% no tempo total de decodificação. O bloco de IFFT foi o principal responsável pela queda de desempenho. Isto deve-se ao fato de que a EMCA não possui especialização para a arquitetura x86 e, portanto, utiliza a implementação C++ não especializada, que pode ser utilizada em qualquer tamanho de bloco de IFFT que seja múltiplo de 2. Esta implementação não é otimizada para nenhuma arquitetura.

A tabela 4.3 lista os tempos médios de execução de cada bloco do decodificador na arquitetura Blackfin com e sem o uso da EMCA. Para esta arquitetura, duas versões da EMCA foram avaliadas, sendo uma genérica, como a utilizada na arquitetura x86, e uma versão especializada para a arquitetura Blackfin. A versão genérica da EMCA mostrou desempenho muito inferior no Blackfin devido à utilização de ponto flutuante na implementação da interface **E_Complex**. Como citado, esta arquitetura não possui FPU, portanto, todas as operações ponto flutuante são emuladas em software. A versão especializada para a arquitetura Blackfin mostrou desempenho superior à implementação original do decodificador para arquiteturas ponto fixo. Isto se deve a utilização de uma biblioteca para operações DSP fornecida pela Analog Devices, fabricante deste processador. Para tornar possível a invocação desta biblioteca pela

interface da EMCA, também foi necessária a utilização de uma versão especializada da interface **`E_Complex`** para esta arquitetura, como descrito em 3.5.

Este estudo de caso mostrou a utilização da EMCA no decodificador FAAD2 e analisou o impacto no seu desempenho. A especialização da EMCA para arquitetura Blackfin mostra que o seu objetivo, portabilidade com eficiência, foi cumprido com sucesso. É importante salientar que as mesmas modificações realizadas no código fonte do FAAD2 para arquitetura x86 foram realizadas para o Blackfin. A interface da EMCA é a mesma utilizada em ambas arquiteturas, permitindo implementações especializadas que favoreçam o desempenho de acordo com as extensões de hardware existentes.

Capítulo 5

Conclusões

A evolução das tecnologias de fabricação de semicondutores permite a criação de aplicações multimídia embarcadas, com novas características e requisitos, como videogames portáteis, vídeo *players* pessoais, televisores móveis, etc. que há uma década não seriam tecnologicamente e economicamente viáveis. Um reprodutor de vídeo móvel, por exemplo, possui hoje um preço atrativo e uma autonomia de bateria razoável, favorecendo a comercialização destes produtos em grande escala. A busca por produtos mais eficientes incentiva a criação de novas arquiteturas, e consequentemente novas plataformas, mais adequadas às necessidades de processamento dessa gama de aplicações.

O desenvolvimento de aplicações multimídia é geralmente realizado em linguagens de alto nível, como C e C++. Contudo a compilação para diferentes plataformas embarcadas nem sempre gera código binário que faça uso de toda potencialidade da plataforma alvo. É comum que desenvolvedores destas aplicações façam uso de rotinas em *Assembly* ou chamadas a API do fabricante da plataforma para otimizar partes críticas do código da aplicação.

O uso de bibliotecas específicas de rotinas fornecidas pelo fabricante da plataforma alvo, através de uma API específica, frequentemente supera o desempenho de rotinas desenvolvidas por terceiros. No entanto, o código fonte da aplicação otimizado para uma determinada arquitetura, empregando diretamente chamadas à API do fabricante, não será portável para outras sem que o desenvolvimento volte para o código C não otimizado.

Aplicações multimídia fazem uso de CODECs para permitir a compressão e descompressão de sinais de áudio ou vídeo a serem captados ou reproduzidos. Para permitir o processamento destes sinais, os CODECs utilizam algoritmos DSP como a FFT. Em alguns casos, como em um decodificador de áudio AAC, este bloco é responsável por aproximadamente

um terço do tempo de execução da aplicação.

Trabalhos de pesquisa anteriores abordaram a portabilidade de algoritmos DSP de forma eficiente em diferentes fases do desenvolvimento. A proposta de Chia-Jui Hsu visa uma representação intermediária do algoritmo adotado, e esta representação seria importada para o ambiente de desenvolvimento de diferentes plataformas e então executada a síntese de software para o código da linguagem alto nível utilizada na aplicação [HSU 05]. Uma outra proposta seria a adequação dos compiladores às estruturas de hardware de cada plataforma, como proposto por Sudarsanam [SUD 98] e, posteriormente aperfeiçoado por Hohenauer [HOH 09]. Uma terceira abordagem, proposta por Rojas, que busca portabilidade em nível de aplicação prevê a utilização de macros C que seriam definidas para cada plataforma que a aplicação fosse compilada [ROJ 03].

Dentre as abordagens existentes para prover a portabilidade de aplicações multimídia em sistemas embarcados, a mais difundida é a utilização de APIs independentes de plataforma. Algumas destas APIs são criadas por fóruns, compostos por diferentes fabricantes e usuários das arquiteturas, como a VSIPL, que é uma API para processamento de vetores e imagens criada pelo VSIPL Forum. Um outro exemplo de API para aplicações multimídia e também algoritmos DSP, é a OpenMAX, criada pelo grupo Khronos, que é uma organização para criação e manutenção de padrões livres. A proposta da OpenMAX sugere uma divisão em três níveis de software, desenvolvimento, integração e aplicação. Cada um destes níveis pode ser implementado e distribuído separadamente por terceiros.

Este trabalho apresentou a *Embedded Multimedia Cross-Platform API* (EMCA), que é uma API multiplataforma para algoritmos DSP utilizados em aplicações multimídia. A implementação da EMCA prevê dois níveis de interfaces: os mediadores de hardware, que fornecem para o usuário e para os algoritmos que compõem a API uma interface uniforme para acesso a componentes de hardware independentemente de plataforma; e a interface para algoritmos DSP, representada pela FFT neste trabalho. A interface de algoritmos faz uso dos mediadores em sua implementação, porém, assim como apresentado na seção 3.5, esta pode ser otimizada para arquiteturas específicas que proveem rotinas prontas para a função. A EMCA define uma interface única de acesso ao hardware e de acesso a algoritmos DSP, os conceitos da metodologia ADESD permitem que esta interface seja implementada utilizando características específicas de cada plataforma para otimizar a aplicação.

A EMCA não requer a utilização de funções específicas de inicialização de biblioteca, como a VSIPL, resultando em baixo impacto

no desempenho da aplicação. A interface DSP da EMCA busca suportar algoritmos utilizados na criação de CODECs multimídia, empregando mediadores de hardware, otimizados para cada plataforma em que esta é realizada. Esta abordagem permite o uso da mesma implementação de um algoritmo em diferentes plataformas, permitindo a sua otimização com o uso dos mediadores específicos da plataforma alvo.

O capítulo 4 relata um estudo de caso da EMCA, mostrando as modificações necessárias para a sua utilização na implementação de um decodificador de áudio AAC. Decodificadores de áudio AAC utilizam a transformada inversa de cosseno modificada (IMDCT) para transformar coeficientes, representados no domínio da frequência, para dados de amostras de áudio, representados no domínio do tempo. Esta transformada é responsável por aproximadamente um terço do tempo de execução do decodificador, sendo um algoritmo alvo de otimizações. O estudo de caso realizado mostrou que a especialização da interface de FFT da EMCA, empregada na implementação da IMDCT, utilizando as unidades funcionais e características arquiteturais da plataforma alvo foi capaz de acelerar o desempenho da aplicação mantendo a portabilidade para outras arquiteturas.

O uso da EMCA permite ao desenvolvedor a otimização de rotinas críticas em plataformas específicas sem a necessidade de tornar a aplicação dependente da plataforma. Isto é possível pois é criado um "contrato"da aplicação com a interface da EMCA, que pode ser especializada para diferentes plataformas, com implementações otimizadas para cada uma delas.

5.1 Trabalhos Futuros

Plataformas embarcadas atuais disponibilizam alguns algoritmos utilizados em CODECs multimídia implementados em hardware. É razoável avaliar o uso da EMCA e a possibilidade da sua especialização para estas plataformas. Assim como, é importante avaliar a possibilidade da especialização da EMCA para plataformas com CODECs inteiramente em hardware.

O artefato de mediadores de hardware foi empregado por Marcondes para a modelagem e implementação de componentes híbridos de hardware e software [MAR 09]. Nesse contexto, seria interessante avaliar diferentes configurações de componentes híbridos no domínio de aplicações multimídia.

O estudo de caso utilizado neste trabalho buscou provar a conformidade da proposta, colocando em prática a metodologia de separação

do hardware das interfaces de algoritmos. Seria plausível avaliar a metodologia aqui utilizada em diferentes plataformas, com diferentes aplicações, afim de provar suas propriedades e eficiência.

Referências Bibliográficas

[AMD 00] AMD. **3DNow! Technology Manual**, 2000.

[AMM 09] **JSR 234: Advanced Multimedia Supplements**. http://www.jcp.org/en/jsr/detail?id=234.

[ANA 09] ANALOGDEVICES. **VisualDSP++ 5.0: C and C++ Compiler and Library Manual for Blackfin Processors**, 2009.

[ARA 98] ARAÚJO, G.; MALIK, S. Code generation for fixed-point dsps. **ACM Transactions on Design Automation of Electronic Systems**, New York, NY, USA, v.3, n.2, p.136–161, 1998.

[ARM 97] ARM. **ARM® Architecture Reference Manual ARM®v7-A and ARM®v7-R edition**, 1997.

[ARM 08] **ARM OpenMAX DL Libraries**. http://www.arm.com/products/multimedia/openmax/index.html.

[BAK 00] BAKKER, M. **Freeware Advanced Audio Decoder 2**. http://www.audiocoding.com/faad2.html.

[BAR 98] BARGEN, B.; TEAM, D. **Inside DirectX**. Microsoft Programming Series. 1. ed. Microsoft Press, 1998.

[BHA 00] BHATTACHARYYA, S. S.; LEUPERS, R.; MARWEDEL, P. Software Synthesis and code generation for signal processing systems. **IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMSII. ANALOG AND DIGITAL SIGNAL PROCESSING**, [S.l.], v.47, n.9, p.849–875, Setembro, 2000.

[BIL 95] BILSEN, G. et al. Cyclo-static data flow. In: INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, 1995. ICASSP-95., 1995. **Proceedings...** Washington, DC, USA: [s.n.], 1995. v.5, p.3255–3258 vol.5.

[CHE 09] CHERNENKO, S. **Fast Fourier transform - FFT**. http://www.librow.com/articles/article-10.

[COD 09] **Sourcery VSIPL++**. http://www.codesourcery.com/vsiplplusplus/.

[COO 65] COOLEY, J. W.; TUKEY, J. W. An algorithm for the Machine Calculation of Complex Fourier Series. **Mathematics of Computation**, Washington, DC, USA, v.19, n.90, p.297–301, 1965.

[DEV 08] DEVICES, A. **Blackfin Processor Programming Reference. Revision 1.3**, 2008.

[DEV 09] DEVICES, A. **uClinux distribution project for the Blackfin processor**, 2009.

[DUH 91] DUHAMEL, P.; MAHIEUX, Y.; PETIT, J. A fast algorithm for the implementation of filter banks based on 'time domain aliasing cancellation'. In: 1991

INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, ICASSP-91., 1991. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1991. p.2209–2212.

[FAA 09] **Programs using FAAD2 (libfaad2)**. `http://faac.sourceforge.net/oldsite/programs.php`.

[FRE 07] FREESCALE. **ColdFire DSP Library Reference Manual**, 2007.

[FRÖ 00] FRÖHLICH, A. A.; SCHRÖDER-PREIKSCHAT, W. Scenario Adapters: Efficiently Adapting Components. In: PROCEEDINGS OF THE 4TH WORLD MULTICONFERENCE ON SYSTEMICS, CYBERNETICS AND INFORMATICS, 2000. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2000.

[FRÖ 01] FRÖHLICH, A. A. **Application-Oriented Operating Systems**. Sankt Augustin, Germany: Sankt Augustin: GMD - Forschungszentrum Informationstechnik GmbH, 2001. 220 p. Tese de Doutorado.

[FSF 09a] FSF. **GNU Binary Utilities Manual**, 2009.

[FSF 09b] FSF. **GNU compilers collection manual**, 2009.

[HAY 04] HAYS, W. P. DSPs: Back to the Future. **ACM Queue**, [S.l.], v.2, n.1, Março, 2004.

[HOH 09] HOHENAUER, M. et al. A simd optimization framework for retargetable compilers. **ACM Trans. Archit. Code Optim.**, New York, NY, USA, v.6, p.2:1–2:27, April, 2009.

[HSU 04] HSU, C.-J. et al. DIF: An Interchange Format for Dataflow-Based Design Tools. **Lecture Notes in Computer Science**, Berlin / Heidelberg, Germany, v.3133/2004, p.423–432, 2004.

[HSU 05] HSU, C.-J.; BHATTACHARYYA, S. S. Porting DSP Applications across Design Tools Using the Dataflow Interchange Format. In: RSP 05: PROCEEDINGS OF THE 16TH INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, 2005. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2005. p.40–46.

[INT 99] INTEL. **Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual**, 1999.

[INT 09a] INTEL. **Intel 64 and IA-32 Architectures Software Developers Manual**, 2009.

[INT 09b] INTEL. **Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture**, 2009.

[JAN 01] JANKA, R. S. et al. VSIPL: an object-based open standard API for vector, signal, and image processing. In: ICASSP 01: PROCEEDINGS OF THE ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, 200. ON IEEE INTERNATIONAL CONFERENCE, 2001. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2001. p.949–952.

[JUN 01] JUNG, S.; PAEK, Y. The very portable optimizer for digital signal processors. In: CASES '01: PROCEEDINGS OF THE 2001 INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE, AND SYNTHESIS FOR EMBEDDED SYSTEMS, 2001. **Proceedings...** New York, NY, USA: ACM Press, 2001. p.84–92.

[KER 08] KERR, A.; CAMPBELL, D.; RICHARDS, M. **GPU VSIPL: High-Performance VSIPL Implementation for GPUs**. `http://gpu-vsipl.gtri.gatech.edu/docs/gpuvsipl_hpec2008.pdf`.

[KHR 09] **The Khronos Group - Media Authoring and Acceleration**. http://www.khronos.org/.

[KIC 97] KICZALES, G. et al. Aspect-oriented programming. **In Proceedings of the European Conference on Object-oriented Programming**, [S.l.], v.1241, p.220–242, June, 1997.

[KUU 00] KUULUSA, M. **DSP Processor Core-Based Wireless System Design**. Tampere, Finland: Tampere University of Technology, 2000. Tese de Doutorado.

[LEE 87] LEE, E. A.; MESSERSCHMITT, D. G. Synchronous Data Flow. Washington, DC, USA, v.75, n.9, p.1235–1245, September, 1987.

[LEE 03] LEE, E. A. Overview of the ptolemy project. July, 2003. Relatório TécnicoTechnical Memorandum No. UCB/ERL M03/25.

[LEU 97a] LEUPERS, R. **Retargetable Code Generation for Digital Signal Processors**. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[LEU 97b] LEUPERS, R.; MARWEDEL, P. Retargetable compilers for embedded DSPs. In: EMMSEC 97: PROCEEDINGS OF THE 7TH EUROPEAN MULTIMEDIA, MICROPROCESSOR SYSTEMS AND ELECTRONIC COMMERCE CONFERENCE, 1997. **Proceedings...** New York, NY, USA: ACM Press, 1997.

[LEU 00] LEUPERS, R. Code generation for embedded processors. In: PROCEEDINGS OF THE 13TH INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, 2000. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2000. p.173–178.

[LEV 97] LEVY, M. **C compilers for DSPs flex their muscles**. http://www.ednmag.com.

[LYO 04] LYONS, R. G. **Understanding Digital Signal Processing**. Prentice Hall PTR, Mar, 2004. 688 p.

[MAR 09] MARCONDES, H.; FRÖHLICH, A. A. A hybrid hardware and software component architecture for embedded system design. In: PROCEEDINGS OF THE INTERNATIONAL EMBEDDED SYSTEM SYMPOSIUM, 2009. **Proceedings...** : IESS, 2009. p.259–270.

[MMA 09] **Mobile Media API (MMAPI); JSR 135**. http://java.sun.com/products/mmapi/.

[MPE 96] MPEG. **Information technology – Generic coding of moving pictures and associated audio information: Systems, ISO/IEC 13818-1**, 1996.

[MPE 97] MPEG. **Information technology – Generic coding of moving pictures and associated audio information – Part 7: Advanced Audio Coding (AAC), ISO/IEC 13818-7**, 1997.

[MPE 99] MPEG. **Information technology – Coding of audio-visual objects – Part 1: Systems, ISO/IEC 14496-1**, 1999.

[MPE 01] MPEG. **ISO/IEC 14496-5:2001 Reference Software**, 2001.

[NUC 09] **Nucleus Multimedia leverages the OpenMAX IL multimedia standard to facilitate portability, interoperability, and reuse**. http://www.mentor.com/products/embedded_software/nucleus_rtos/multimedia/.

[OPE 08a] **OpenSL ES - The Standard for Embedded Audio Acceleration**. http://www.khronos.org/opensles/.

[OPE 08b] **The Standard for Media Library Portability**. http://www.khronos.org/openmax/.

[PAR 76] PARNAS, D. L. On the Design and Development of Program Families. **IEEE Transactions on Software Engineering**, [S.l.], v.SE-2, n.1, p.1–9, Março, 1976.

[POL 04] POLPETA, F. V.; FRÖHLICH, A. A. Hardware Mediators: A Portability Artifact for Component-Based Systems. In: LECTURE NOTES IN COMPUTER SCIENCE. PROCEEDINGS OF THE INTERNATIONAL CONFERENCE EMBEDDED AND UBIQUITOUS COMPUTING, TRACK 4: HARDWARE/SOFTWARE CO-DESIGN AND SYSTEM-ON-CHIP, 2004. **Proceedings...** Berlin, Germany: Springer, 2004. v.3207/2004, p.271–280.

[QUA 99] QUACKENBUSH, S.; TOGURI, Y.; HERRE, J. Revised report on complexity of mpeg-2 aac tools. MPEG Audio Subgroup, 1999. Relatório técnico.

[RAN 05] RANKIN, K. **Linux Multimedia Hacks: Tips and Tools for Taming Images, Audio, and Video**. O'Reilly Media, Nov, 2005. 336 p.

[RIN 09] RINTALUOMA, T. **Optimizing H.264 Decoder for Cortex-A8 with ARM NEON OpenMax DL Implementation**. `http://www.on2.com/file.php?232`.

[RIT 93] RITZ, S. et al. Optimum vectorization of scalable synchronous dataflow graphs. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC ARRAY PROCESSORS, 1993., 1993. **Proceedings...** Washington, DC, USA: [s.n.], 1993. p.285–296.

[ROJ 03] ROJAS, J. C.; LEESER, M. Programming portable optimized multimedia applications. In: MULTIMEDIA '03: PROCEEDINGS OF THE ELEVENTH ACM INTERNATIONAL CONFERENCE ON MULTIMEDIA, 2003. **Proceedings...** New York, NY, USA: ACM, 2003. p.291–294.

[SCH 08] SCHWARTZ, D. A. et al. **VSIPL 1.3 API**, January, 2008.

[SFM 09] **Simple and Fast Multimedia Library**. `http://www.sfml-dev.org/`.

[SOL 00] SOLOMON, D. **Data Compression: The Complete Reference**. 2. ed. Springer, 2000.

[SUD 98] SUDARSANAM, A. **Code optimization libraries for retargetable compilation for embedded digital signal processors**. Princeton, NJ, USA: Princeton University, 1998. 209 p. Tese de Doutorado.

[TEX 08] TEXASINSTRUMENTS. **C62x/C64x Fast Run-Time Support (RTS) Library**, 2008.

[ZAR 04] ZARETSKY, D. et al. Overview of the FREEDOM compiler for mapping DSP software to FPGAs. In: FCCM '04: PROCEEDINGS OF THE 12TH ANNUAL IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 2004. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2004. p.37–46.

[ZIV 95] ZIVOJNOVIC, V. et al. Dsps, gpps, and multimedia applications - an evaluation using dspstone. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING APPLICATIONS AND TECHNOLOGY, 1995. **Proceedings...** ICSPAT: [s.n.], 1995.