UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

BRUNO POLICARPO TOLEDO FREITAS

ADAPTAÇÃO E ACELERAÇÃO DO MIDDLEWARE GINGA-NCL PARA O SISTEMA-EM-CHIP DO SBTVD

Porto Alegre
2014

BRUNO POLICARPO TOLEDO FREITAS

ADAPTAÇÃO E ACELERAÇÃO DO MIDDLEWARE GINGA-NCL PARA O SISTEMA-EM-CHIP DO SBTVD

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Engenharia de Computação - Processamento de Sinais

ORIENTADOR: Prof. Dr. Altamiro Amadeu Susin

Porto Alegre
2014

BRUNO POLICARPO TOLEDO FREITAS

ADAPTAÇÃO E ACELERAÇÃO DO MIDDLEWARE GINGA-NCL PARA O SISTEMA-EM-CHIP DO SBTVD

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Altamiro Amadeu Susin, UFRGS
Doutor em Informática pela INPG - Grenoble, França

Banca Examinadora:

Prof. Dr. Carlos Eduardo Pereira, UFRGS
Doutor pela Technische Universitat Stuttgart - Stuttgart, Alemanha)

Prof. Dr. Alexandre da Silva Carissimi, UFRGS
Doutor pela INPG - Grenoble, França

Prof. Dr. Marcelo Götz, UFRGS
Doutor pela Universität Paderborn - Paderborn, Alemanha

Coordenador do Curso: _____
Prof. Dr. Alexandre Sanfelice Bazanella

Porto Alegre, Maio de 2014.

AGRADECIMENTOS

Houve poucas vezes em minha vida em que a minha veia de escritor pulsou ativamente, e nestes momentos a motivação foi condição sine qua non para o sucesso da produção textual. Por essa razão, agradeço a todos que me incentivaram e me apoiaram ao longo do processo de escrita desta dissertação de Mestrado.

Agradeço, em especial, ao meu orientador, professor Altamiro Amadeu Susin, pela paciência, pelo acolhimento, pelas relevantes intervenções e contribuições que resultaram na concretização exitosa deste trabalho de pesquisa científica.

Agradeço a todos os professores das disciplinas que estudei no Curso de Mestrado, por terem compartilhado seus saberes e conhecimentos com competência, pois seus ensinamentos foram fundamentais para ampliar meus conhecimentos referentes ao tema desta pesquisa.

A meus colegas e amigos do grupo de pesquisa do LaPSI, agradeço pela troca de experiências e de conhecimentos como também pelos momentos de descontração durante o período de nossa convivência, os quais foram essenciais para recarregar as minhas baterias mentais.

Agradeço, ainda, à Coordenação do Programa de Pós-Graduação de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul por me ofertar a oportunidade de aperfeiçoamento profissional.

Por fim, agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior pela bolsa de estudo que me foi concedida durante o período de realização do Curso de Mestrado em Engenharia Elétrica.

RESUMO

Este trabalho tem por finalidade aprimorar o Sistema-em-Chip (SoC) desenvolvido para um "Set-Top Box"de Televisão Digital com a capacidade de executar aplicações segundo o Sistema Brasileiro de Televisão Digital (SBTVD) e melhorar o desempenho do novo sistema. A "Rede H.264"e o "GingaCDN", dois projetos desenvolvidos anteriormente relacionados à Televisão Digital, foram utilizados como base para esse trabalho. A Rede H.264 teve como principal objetivo o desenvolvimento de codificadores e decodificadores para o padrão brasileiro. O resultado foi um SoC para "Set-Top Box"que inclui uma interface de usuário, um processador e os decodificadores de áudio e vídeo com suas respectivas interfaces de saída. Por outro lado, o GingaCDN criou uma implementação de referência para o middleware do SBTVD, denominado Ginga. O primeiro passo foi adicionar regras para compilar o Ginga no ambiente de desenvolvimento do SoC, sendo necessárias diversas mudanças na infraestrutura do middleware. O desempenho do sistema é melhorado através de hardware-software codesign onde as primitivas do Ginga que consomem maior tempo de processamento e de processador foram implementadas em hardware. O ganho obtido ocorre devido a dois fatores: o sistema se torna mais rápido e os recursos da CPU são liberados para outras aplicações. Neste trabalho, o alvo foi o subsistema gráfico do middleware, onde o impacto é mais significativo. Um estudo das características do hardware do sistema foi realizado e, então, uma função gráfica foi escolhida e implementada em hardware. Todas as etapas para substituir uma função em software por outra equivalente em hardware são descritas no texto. Entre as contribuições deste trabalho, abre-se espaço para dar continuidade à expansão das capacidades do "Set-Top Box"por meio de módulos de hardware, melhorando a eficiência do SoC para esta aplicação, dito então "Ginga-ready". A experiência pode ser estendida também para auxiliar a geração de middleware para outras plataformas.

Palavras-chave: Televisão Digital, Set-Top Box, H.264, Middleware, Ginga, Ginga-NCL.

ABSTRACT

This work aims to enhance a System-on-Chip (SoC) designed for Digital Television Set-Top Box in order to run applications according to the Brazilian Television standard (SBTVD) and to increase the performance by hardware. Two previous projects related to the Digital Television, "Rede H.264" and "GingaCDN", were used as base for this work. The "Rede H.264" had as main objective the development of codecs for the Brazilian standard. The result was a SoC for a Set-Top Box which includes a processor, audio and video decoders with output drivers, and user interface. Otherwise, the "GingaCDN" created a reference implementation for the middleware of the SBTVD, called Ginga. The first step was to add rules to compile Ginga on the development environment of the SoC, for which some configuration of the Ginga middleware needed to be changed. Performance improvement was obtained by hardware-software codesign where Ginga primitives that are time and processor consuming could be implemented in hardware. The gain is twofold: the system becomes faster and CPU resources are freed for other applications. In this work, the target was the graphical subsystem primitives of the middleware, where the impact is more significant. A study of the hardware characteristics of the system was made, and then a graphical function was chosen and implemented on hardware. All the steps needed to substitute a software function by an equivalent one implemented in hardware are described in the text. Among the contributions of this work, the way is openned to continue the expansion of the capabilities of the Set-Top Box by efficient hardware modules on a so called "Ginga-ready" SoC. The experience may be useful also to help the generation of the middleware for other platforms.

Keywords: Digital Television, Set-Top Box, H.264, Middleware, Ginga, Ginga-NCL.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

AAC	Advanced Audio Coding
ABNT	Associação Brasileira de Normas Técnicas
API	Application Programming Interface
AHB	Advanced High-performace Bus
AMBA	Advanced Microcontroller Bus Architecture
ARGB	Alpha - Red - Green - Blue
APB	Advanced Peripheral Bus
AVC	Advanced Video Coding
CPU	Central Processing Unit
EPG	Electronic Program Guide
ES	Elementary Stream
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
FSF	Free Software Foundation
GCC	GNU Compiler Collection
GNU	GNU is Not Unix
GPL	GNU General Public License
HAL	Hardware Abstraction Layer
IP	Internet Protocol
IPTV	Televisão sobre IP
ISDB-T	Integrated Services Digital Broadcasting Terrestrial
ISDB-Tb	Integrated Services Digital Broadcasting Terrestrial, Brazilian version
LaPSI	Laboratório de Processamento de Sinais e Imagens
MDDR	Multi-channel Double Data Rate Memory
MMC	Multi-channel Memory Controller
MPEG	Moving Picture Experts Group

NCL	Nested Context Language
OSD	On-Screen Display
PnP	Plug and Play
RGB	Reg-Green-Blue
SBTVD	Sistema Brasileiro de Televisão Digital
SOC	System-On-Chip
STB	Set-Top Box
STL	Standard Template Library
TCP	Transmission Control Protocol
TS	Transport Stream
TVD	Televisão Digital
VHDL	VHSIC Hardware Description Language
XML	Extended Markup Language

SUMÁRIO

1 INTRODUÇÃO

A evolução dos equipamentos de aquisição e exibição de imagem e reprodução de áudio proporcionaram uma melhoria significativa da qualidade de cenas de televisão. Associada a esta melhor qualidade está o acréscimo do volume de informação a ser transmitido por um canal de televisão. Um canal de transmissão analógica não comporta o incremento demandado pela televisão de alta definição.

A digitalização possibilitou uma melhora significativa no Sistema de Televisão como um todo, através da compressão do sinal-fonte de vídeo e áudio e transmissão digital da informação com inclusão de procedimentos de correção de erros. Evidentemente, as ganhos com a redução do volume de informação foi aproveitado pelos processos de armazenamento que reduziram praticamente em duas ordens de grandeza o espaço utilizado em arquivos de computador, DVDs, etc.

A compressão do sinal-fonte baseia-se principalmente na redução da redundância espacial e temporal. Este processo é eficiente a ponto de permitir a transmissão de vários canais com a resolução da televisão analógica numa faixa do espectro utilizada por apenas um canal. Por outro lado, na mesma faixa do espectro de um canal de televisão analógica é possível transmitir um programa em alta resolução com vários canais de áudio, além de outro de baixa resolução para equipamentos móveis. Ainda, no fluxo digital são inseridas informações para o usuário como uma tabela de programação e outros aplicativos como serviços para o cidadão. Esses aplicativos podem trocar informações com a emissora ou com outros servidores por meio de um *canal de retorno*, normalmente a internet.

O sinal transmitido é um fluxo de dados, chamado Fluxo de Transporte (TS, do inglês Transport Stream) que contém uma multiplexação dos Fluxos Elementares (ES, do inglês Elementary Stream) de áudio e vídeo gerados pela emissora e dos dados de programação e serviços. Os Fluxos Elementares contêm as informações de áudio e vídeo digitalizadas e comprimidas. Os dados para o usuário, sinais de controle, programas (Aplicativos) e dados para interatividade são transmitidos no mesmo TS devidamente empacotados e identificados pelo Multiplexador. Na recepção os diversos pacotes são separados para remontar os elementos originais.

A capacidade de enviar aplicações por meio de um TS muda a maneira com que os programas televisivos são concebidos. Como ilustração, consideremos o caso das enquetes televisivas, por exemplo, a emissora está transmitindo um jogo de futebol e deseja saber do telespectador qual é o melhor jogador da partida. Atualmente, as emissoras brasileiras adotam uma solução em duas partes. Primeiro, elas transmitem a conteúdo televisivo sobreposto com os dados da enquete, contendo normalmente um endereço *online* para entrar com a resposta. O telespectador então acessa o endereço por meio de um dispositivo com acesso a Internet e entra com seu voto.

Na Televisão Digital, o jogo e a enquete são duas entidades diferentes, transmitidas

pelo mesmo canal e sobrepostas pelo STB no lado do telespectador. A enquete se torna então um *programa não-linear*. Dentro do contexto da TVD, em um programa não-linear o que a emissora envia não é necessariamente aquilo que o telespectador observa pois ele mesmo está alterando seu fluxo de execução.

Voltando ao exemplo, a pergunta agora é respondida por meio do controle remoto e sua resposta enviada por meio de um canal de retorno à emissora ou a um outro gerador de conteúdo. Ela pode, por exemplo, compilar as estatísticas temporárias e envia-las de volta por meio do mesmo canal junto a um comando para mostrar os resultados parciais.

Para que esse cenário seja possível, o STB torna-se mais complexo. Agora, ele deverá ser capaz de receber e executar as aplicações que são transmitidas junto com sinais de áudio e vídeo, o que requer um sistema computacional adequado. Esse sistema necessita de um hardware com periféricos para interface com o usuário (com controle remoto e capacidade gráfica) e um sistema operacional. Além disso, como mencionado em (BARBOSA; SOARES, 2008), programas não-lineares para TVD possuem requisitos específicos, como:

- **Separação entre definição de mídias e relacionamentos entre elas**. Na televisão analógica, o áudio e vídeo recebidos são compostos somente do lado da emissora. Já em um programa não-linear, a composição ocorre também no lado usuário. Isso requer que seja possível manipular as mídias, que antes vinham compostas da emissora, de forma separada. Uma vez que as mídias agora vêm separadas, torna-se necessário um mecanismo para relacionar como elas devem ser exibidas no lado do telespectador.
- **Suporte à sincronização de objetos de mídia**. Dados enviados multiplexados no TS, a princípio, não possuem sincronização com áudio e vídeo principais. Isso é um problema pois programas não-lineares são alterados pelo telespectador de acordo com esses eventos de áudio e vídeo. Portanto, é necessário que o programa não-linear implemente um mecanismo de sincronização.
- **Suporte à exibição em múltiplos dispositivos**. A TVD possibilita que a transmissão seja recebida por outros tipos de dispositivos além da televisão, como celulares ou computadores pessoais. Logo, é importante que o programa não-linear se adapte às características do dispositivo. No caso do jogo de futebol, por exemplo, ter a capacidade de redimensionar o tamanho do placar do jogo e posicioná-lo de acordo com a resolução da tela.
- **Suporte à edição ao vivo**. No exemplo acima, a enquete é atualizada pelo canal de retorno. Para que isso seja possível, a emissora deve ter a capacidade de alterar o programa ao vivo. Por exemplo, enviando um comando para redesenhar o gráfico toda vez que os dados da enquete são atualizados.
- **Adaptação de conteúdo**. Esta é a propriedade dos programas não-lineares de se adaptarem seguindo algum critério. O exemplo mais comum é de acordo com a localização do telespectador, por exemplo, uma propaganda que indica qual a loja mais próxima.

Além de cumprir os requisitos acima, é importante que as aplicações executem em qualquer STB e que haja uma interface de programação padrão (API, do inglês *Application Programming Interface*) para desenvolvimento. Para atingir todos esses objetivos, adiciona-se mais uma camada de software acima do sistema operacional, chamada *middleware*, cujo objetivo é executar as aplicações de TVD de acordo com um padrão de referência.

1.1 Motivação

Para entender a motivação deste trabalho, é necessário fazer o retrospecto da história da TVD no Brasil. O Sistema Brasileiro de Televisão Digital (SBTVD) foi criado oficialmente no ano de 2003, por meio de um decreto presidencial (BRASIL, 2003). Iniciou-se, então, uma etapa de estudos a fim de definir os padrões de vídeo, áudio, e aplicações a serem utilizados, de acordo com os objetivos estratégicos do governo.

O padrão brasileiro foi definido no ano de 2006 (BRASIL, 2006), baseado no padrão japonês ISDB-T modificado com atualizações brasileiras. Chamado de ISDB-Tb, o padrão brasileiro tem como principais novidades a utilização de codificadores e decodificadores (codecs) de áudio e vídeo atualizados e um novo middleware denominado *Ginga*. A Tabela 1 mostra os padrões utilizados no SBTVD segundo a ABNT (Associação Brasileira de Normas Técnicas):

Tabela 1: Padrões utilizados no SBTVD

Vídeo	MPEG4 AVC (ABNT, 2007a)
Áudio	MPEG4 AAC (ABNT, 2007b)
Middleware	Ambiente declarativo: Ginga-NCL (ABNT, 2007c) Ambiente procedural: Ginga-J (ABNT, 2012)

De 2006 em diante, iniciaram-se pelas universidades brasileiras diversos projetos relacionados ao SBTVD. Dois deles são importantes para este trabalho: a *Rede H.264*, iniciado em 2007, e a *GingaCDN*, iniciado em 2009.

A Rede H.264 foi um projeto envolvendo diversas universidades brasileiras, tendo como objetivo desenvolver tecnologia nacional compatível com os padrões do SBTVD. Uma de suas atribuições foi desenvolver codificadores e decodificadores para o sinal fonte de áudio e vídeo para um STB compatível com o SBTVD.

O GingaCDN (Ginga Code Developers Network) foi criado em 2009 com o objetivo de criar uma implementação de referência do middleware Ginga. O middleware Ginga é dividido em dois ambientes: uma máquina de execução chamada Ginga-J (SOUZA FILHO; LEITE; BATISTA, 2007) e uma máquina de apresentação chamada Ginga-NCL (SOARES; RODRIGUES; MORENO, 2007a). Antes da criação do GingaCDN, esses dois desenvolvimentos aconteciam de forma paralela, forçando desenvolvedores de aplicações a escolher ou um paradigma ou outro.

Um outro reflexo oriundo desse início disjunto entre as duas plataformas de middleware ocorreu no desenvolvimento do STB: no início deste trabalho, não era possível executar aplicações de Televisão Digital, limitando-o a funcionar como um televisor convencional. Porém, esse cenário muda com a implementação de referência, pois pode-se adaptá-la ao Sistema-em-Chip (SOC, do inglês *System-On-Chip*) do STB, permitindo enfim a execução de aplicações interativas para TVD.

Mas mais do que somente adaptar a implementação de referência para o STB, abre-se a possibilidade de expandir a arquitetura do hardware do STB. Estudando a execução do middleware dentro do novo ambiente de hardware e descobrindo seus gargalos, pode-se criar módulos em hardware para acelerar e, futuramente, otimizar a execução de aplicações de TVD para o SOC.

1.2 Objetivos

O objetivo geral deste trabalho é unificar a atual plataforma de hardware do STB com o middleware Ginga, ou seja, ter os componentes de software deste último executando no STB. Para tanto, optou-se por começar pelo middleware declarativo Ginga-NCL.

Com uma versão inicial dos componentes em software do middleware integrados, pode-se começar a estudar como propor um STB *Ginga-Ready*, ou seja, que acelere as funções mais lentas do middleware por meio da criação de módulos em hardware para elas. Esse tema foi abordado com foco na infraestrutura gráfica utilizada pelo middleware. Desse modo, foi escolhida uma função gráfica do middleware e um hardware acelerador para ela foi implementado, substituindo a original em software.

Dito isso, os objetivos específicos são:

1. Adaptar a implementação de referência do middleware Ginga-NCL para a plataforma de hardware do STB
2. Implementar um hardware para acelerar uma função gráfica utilizada pelo middleware Ginga-NCL

1.3 Organização do documento

Esta dissertação está organizada da seguinte forma: o Capítulo 2 tem a finalidade de apresentar a implementação de referência do middleware e a plataforma de hardware do STB. Antes, será feita uma apresentação de como é organizado sistema de TVD interativo, buscando relacioná-lo com as plataformas envolvidas. Logo após a descrição das plataformas, no Capítulo 3 encontra-se uma análise bibliográfica sobre o que já foi feito para executar o middleware Ginga-NCL em STBs.

O trabalho de adaptação da implementação de referência é tema do Capítulo 4. A implementação de referência é abordada de forma mais técnica, descrevendo o sistema de distribuição utilizado por ela e as ferramentas de desenvolvimento. Com isso, é apresentada a estratégia de adaptação e as soluções adotadas.

O Capítulo 5 discorre sobre a aceleração da função gráfica escolhida, começando com a escolha da função e seus motivos. Em seguida, é apresentado o caminho que deve ser trilhado desde a implementação em hardware até a substituição da implementação em software, abordando-os sequencialmente.

O Capítulo 6 informa e analisa os resultados obtidos. O capítulo iniciará fazendo uma análise do trabalho de porte do middleware com base em programas de sua suíte de testes de conformidade. Em seguida, serão apresentados os resultados obtidos das modificações feitas no hardware gráfico do STB a fim de que o middleware pudesse gerar sua saída gráfica. O capítulo então terminará mostrando os resultados obtidos com o hardware implementado para a função escolhida, comparando-o com a antiga implementação em software. Por fim, o Capítulo 7 irá abordar as conclusões deste trabalho.

2 DESCRIÇÃO DAS PLATAFORMAS

Este capítulo tem a finalidade de apresentar, de acordo com as necessidades deste trabalho, o middleware Ginga-NCL e a plataforma de hardware do STB. Antes, será feita uma introdução a conceitos de transmissão digital e ilustrado como um sistema de TVD interativa é organizado. Com isso, este capítulo é finalizado com a apresentação de uma visão geral do que é necessário para estender o STB atual para suportar interatividade, indicando a área de atuação adotada neste trabalho.

2.1 Fundamentos de Transmissão Digital

Esta seção tem a finalidade de familiarizar o leitor com as noções relacionadas à transmissão digital e, consequentemente, aos termos técnicos utilizados ao longo deste trabalho.

De forma geral, os processos envolvidos na transmissão digital visam explorar redundâncias em um vídeo com a finalidade de comprimir dados. Um vídeo é uma sequência de imagens que mudam a uma taxa maior do que aquela capaz de ser processada pelo olho humano, dando a impressão de movimento. Uma imagem é definida como uma função bidimensional de elementos de imagem conhecidos como *pixels*.

Explorar redundâncias consiste em atuar na informação que um vídeo carrega, com o objetivo de transmitir a mesma informação com menor quantidade de dados. Ela começa pelo *pixel*, através da escolha de um *espaço de cor* adequado para a transmissão digital. Um espaço de cor é uma representação matemática para a cor de um pixel. Um exemplo de espaço de cor bastante conhecido é o RGB, utilizado para a exibição em dispositivos de vídeo, e que consiste em representar a cor como a adição das três componentes primárias de cor vermelho, verde e azul.

Isso não significa que o RGB seja bom também para transmissão digital. O olho humano, por exemplo, possui maior sensibilidade a variações de intensidade luminosa do que de cor. Reduzindo a quantidade de informações de cor enviadas e mantendo as de intensidade, pode-se fazer a visão humana acreditar que a imagem recebida é a mesma que foi capturada. Esse processo é conhecido como *subamostragem de cor*.

Para explorar essa propriedade, o padrão H.264/AVC utiliza o formato de cor **YCbCr**: Luminância (Y), Crominância azul (Cb), Crominância vermelho (Cr). Com isso, pode-se agora reduzir a quantidade de informações de cor sendo enviadas reduzindo a resolução de cor verticalmente, horizontalmente ou ambas ao mesmo tempo. Esta última é a mais utilizada na transmissão digital e é chamada de subamostragem 4:2:0. Isso significa que, de um grupo de 4 pixels, serão enviadas as informações de luminosidade de cada um deles mas apenas um informação de cor. A Figura 1 ilustra esse tipo de amostragem, além de outros utilizados.

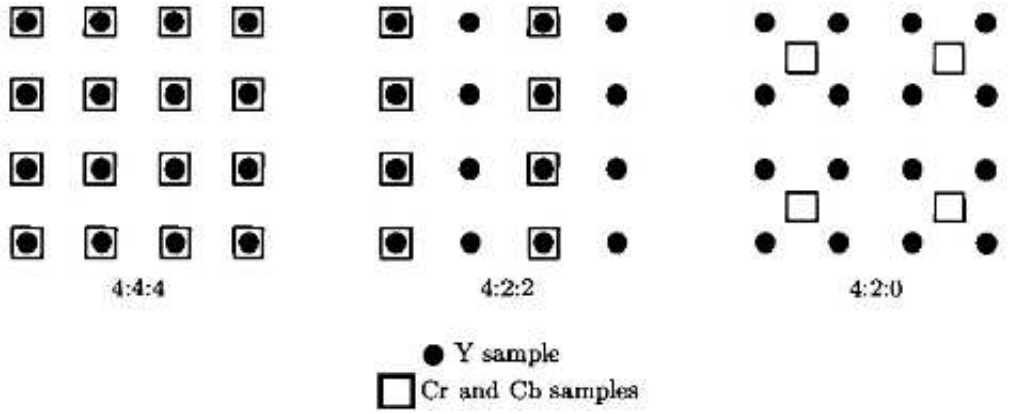

Figura 1: Amostragens de YCbCr (SAHAFI, 2005)

Aumentando o escopo, o próximo passo é pensar na imagem. Muitas vezes, uma imagem contém muitas redundâncias. Por exemplo, se tirarmos uma foto do céu, é de se esperar que os pixels serão majoritariamente azuis. Para explorar essa propriedade, a transmissão digital divide a imagem em vários blocos de dimensões iguais, de forma que ela possa descrever um bloco como sendo igual a outro. O tamanho de bloco utilizado possui 16 linhas por 16 colunas de pixels e é chamado *macrobloco*. O processo de explorar redundâncias dentro de uma imagem é chamado de *predição intraquadros* (BONATTO et al., 2013).

O próximo passo é explorar as redundâncias na sequência de imagens em um vídeo. No início desta seção foi definido que um vídeo é uma sequência de imagens que muda a uma taxa maior do que aquela que o olho humano pode processar. A velocidade com que essa mudança ocorre implica ter imagens em sequência muito semelhantes dependendo do que está sendo transmitido. Por exemplo, na transmissão de um apresentador de telejornal, a maior parte das diferenças entre as imagens capturadas estará nos movimentos dos lábios. O processo de explorar redundâncias entre várias imagens em sequência chama-se *predição interquadros*.

2.2 Sistema de Televisão Digital Interativo

O diagrama de um sistema de TVD interativo é mostrado na Figura 2. Do lado esquerdo da figura, a emissora compõe o TS que será transmitido como sendo o áudio e vídeo principais e o aplicativo interativo. Pode-se enviar o programa e todos os dados utilizados no TS, ou o apenas o programa principal e obter os demais dados por meio do canal de retorno, representado pelos blocos de "Encap. IP"e "Desenc. IP".

O lado direito da figura é o que realmente nos interessa. O fluxo começa com o sinal transmitido por radiodifusão sendo demodulado e o TS passado ao demultiplexador. O demultiplexador separa os ES de áudio e vídeo, enviando-os aos decodificadores e também o programa recebido. A imagem final vista pelo telespectador é a composição entre o vídeo e áudio decodificados com uma saída gerada pelo programa. Conforme a interação do telespectador, o sistema envia dados para a emissora por meio do canal de retorno, através do qual ela pode alterar ao vivo o programa recebido pelo TS.

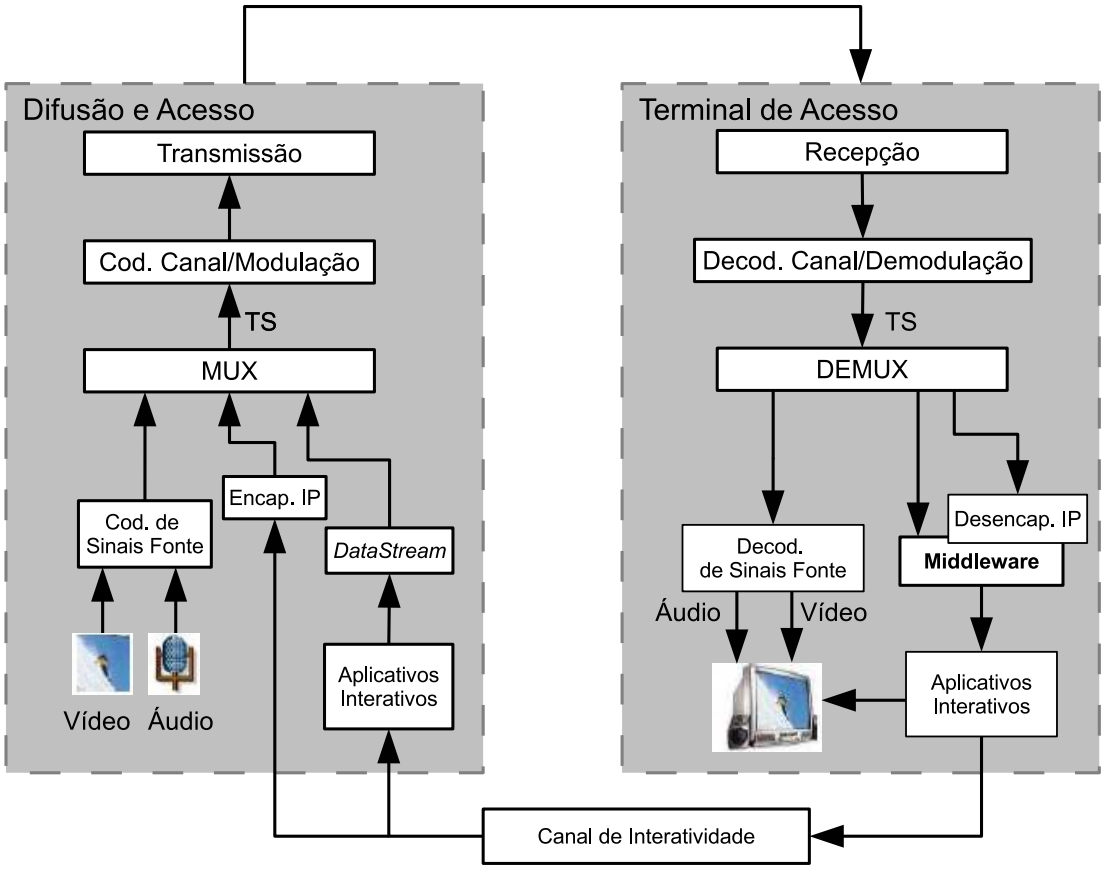

Figura 2: Organização de um Sistema de TVD Interativo

2.3 O Middleware Ginga-NCL

O SBTVD define o Ginga-NCL(SOARES et al., 2010) como o middleware para a execução de aplicações *declarativas* para TVD. Uma linguagem declarativa é aquela cujo foco está em descrever o *resultado* desejado ao invés do *algoritmo* para se chegar nele. A vantagem dessa abordagem em relação a linguagens imperativas, tais como o C, é que isto facilita sua utilização por parte dos produtores de conteúdo, os quais geralmente não possuem histórico como programadores. O Ginga-NCL é normatizado segundo a (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2007c) e também é Recomendação da ITU-T para IPTV (TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU (ITU-T), 2009).

Linguagens declarativas são voltadas a domínios específicos de aplicações, pois dessa forma elas podem definir modelos para descrever aplicações desse domínio. O modelo adotado pelo Ginga-NCL para descrever uma aplicação de TVD consiste em relacionar como diferentes tipos de mídia são relacionados no tempo e no espaço a fim de formar uma *apresentação multimídia*.

O nome Ginga-NCL deriva do fato da linguagem de desenvolvimento de aplicações chamar-se NCL (Nexted Context Language). Um programa NCL é um arquivo XML descrevendo como as mídias de um programa de TVD são relacionadas. O conceito de mídia aqui é mais amplo, sendo tudo aquilo que é permitido ser executado pela máquina de execução NCL.

Uma apresentação multimídia em NCL deve definir a **mídia** que deve ser apresentada, **onde**, **quando**, e **como**. Para ilustrar esse conceito, um programa NCL é apresentado na

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <ncl id="simpleNCL" >
3 <head>
4    <regionBase>
5       <region id="metade_esquerda_da_tela"
6          width="50%" height="100%" top="0%" left="0 %" />
7
8       <region id="metade_direita_da_tela"
9          width="50%" height="100%" top="0%" left="50%" />
10   </regionBase>
11
12   <descriptorBase>
13      <descriptor id="esquerda" region="metade_esquerda_da_tela"
     explicitDur="3s" />
14      <descriptor id="direita" region="metade_direita_da_tela"
     explicitDur="3s" />
15   </descriptorBase>
16
17   <connectorBase>
18      <causalConnector id="onEndStart">
19        <simpleCondition role="onEnd" />
20      <simpleAction role="start" />
21      </causalConnector>
22   </connectorBase>
23 </head>
24 <body>
25   <port component="azul" />
26
27   <media id="azul" src="azul.jpg" descriptor="esquerda" />
28   <media id="vermelho" src="vermelho.jpg" descriptor="direita" />
29
30   <link xconnector="onEndStart">
31    <bind component="azul" role="onEnd" />
32      <bind component="vermelho" role="start" />
33   </link>
34
35 </body>
36 </ncl>
```

Listagem 2.1: Um NCL que exibe uma imagem na metade esquerda da tela

Listagem 2.1. O programa exibe uma imagem azul na metade esquerda da tela durante 3 segundos, quando então passa a exibir uma imagem vermelha na outra metade por mais 3 segundos, terminando em seguida.

No programa acima, os locais **onde** as imagens devem aparecer são definidos nos nodos *region* entre as linhas 5 e 9. Com as regiões definidas, cria-se as descrições de **como** as mídias devem ser exibidas, feito pelos nodos *descriptor* entre as linhas 12 e 15 . Agora pode-se definir **o que** será executado, associando às mídias as descrições desejadas entre nas linhas 27 e 28 .

Por padrão, um programa NCL sempre inicia uma apresentação executando a mídia especificada no nodo *port* na linha 25, definindo o **quando**. Para definir quando as demais mídias deverão ser tocadas no resto da apresentação, elas deverão ser relacionadas entre si. Para isso, NCL define o conceito de *elos*.

Elos definem uma relação de causalidade e são especificados por meio de *conectores*.

Um conector deve definir, no mínimo, uma condição e uma ação. No exemplo acima, queremos exibir a imagem vermelha após o término da exibição da imagem azul. Para isso, define-se o conector *onEndStart* entre as linhas 18 e 21.

Definido o conector, define-se o elo entre as linhas 30 e 33, especificando as mídias envolvidas e os seus papéis dentro do conector: na linha 31, à imagem azul é associado o papel *onEnd*, que é uma condição, ou seja, quando ela for satisfeita irá ocorrer uma ação. Essa ação é iniciar a apresentação da imagem vermelha, o que é feito associando o papel *start* à imagem vermelha na linha 32.

A linguagem NCL apresenta muitos outros recursos, os quais serão apresentados neste trabalho conforme a necessidade. Para tutoriais mais detalhados, consulte (BARBOSA; SOARES, 2008; SANT'ANNA et al., 2009, BARBOSA et al.).

2.3.1 Arquitetura do Middleware

A Figura 3 mostra uma visão geral da arquitetura do middleware Ginga-NCL. Ele é dividido nas camadas *Ginga Common-Core* e na *Máquina de Apresentação* Ginga-NCL. A Máquina de Apresentação é responsável por orquestrar a apresentação multimídia descrita por um programa NCL. Já o *Ginga Common-Core* implementa os mecanismos necessários para a execução do um programa NCL de acordo com a configuração do sistema. O *Ginga Common-Core* é compartilhado tanto pela Máquina de Apresentação Ginga-NCL quanto pela Máquina de Execução Ginga-J.

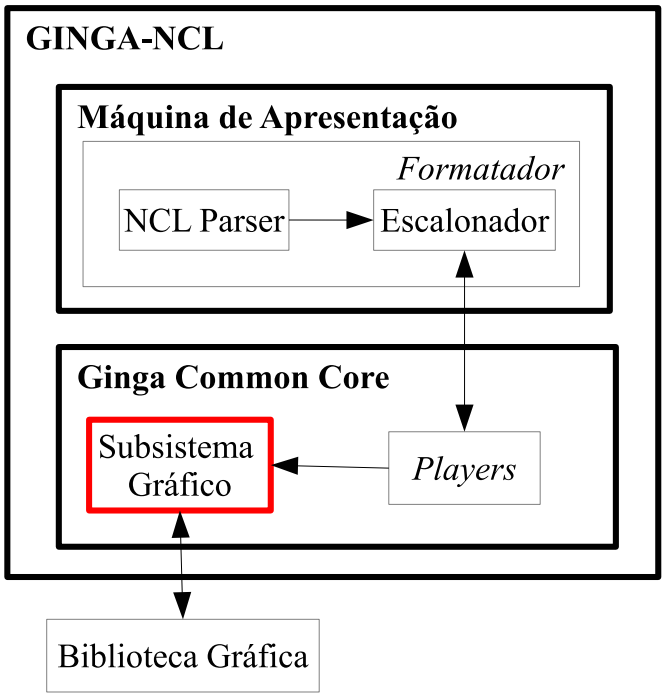

Figura 3: Arquitetura do Middleware Ginga-NCL

Um fluxo comum de execução de uma aplicação NCL começa pelo **Formatador**, responsável por realizar a interpretação da linguagem NCL, invocando interpretadores XML. Essa etapa aciona o **Escalonador**, responsável por transformar a apresentação em eventos como apresentar uma imagem em um dado instante de tempo. Esses eventos são então traduzidos em chamadas a bibliotecas e dispositivos através dos ***players*** implementados na camada *Ginga Common-Core*. Para que uma determinada mídia seja capaz de ser executada pela Máquina de Apresentação Ginga-NCL, um *player* deve estar implementado na

camada *Ginga Common-Core*. Os *players* comunicam-se com o Escalonador, sinalizando para ele quando que uma mídia inicia e termina sua apresentação, ou quando uma mídia é selecionada, a fim de que o Escalonador possa disparar eventos de mídia relacionados.

Neste trabalho, foi utilizada a capacidade do Ginga-NCL de realizar operações gráficas. Dentro desse contexto, e de acordo com o que foi dito no parágrafo anterior sobre o fluxo de execução de um programa NCL, as operações gráficas são traduzidas pelo *player* implementado para chamadas ao **Subsistema Gráfico**. O Subsistema Gráfico tem a finalidade de transformar as chamadas a rotinas gráficas do *player* implementado em chamadas a rotinas da biblioteca gráfica configurada pelo *Ginga Common-Core*.

2.3.2 Scripts NCLua e Interatividade

Como explicado na Seção 2.3.1, o conceito de mídia no NCL é tudo aquilo que pode ser executado pela Máquina de Apresentação do Ginga-NCL. O Ginga-NCL explora esse conceito para adicionar a interatividade implementando um *player* para a execução de *scripts NCLua* (TELEMIDIA LABS, 2014b).

Scripts NCLua são uma forma de embutir código imperativo em uma aplicação NCL, invocando um interpretador para Lua(IERUSALIMSCHY; FIGUEIREDO; CELES, 2006) com diretivas específicas para modificar a apresentação multimídia durante sua execução. Dentre essas diretivas está a capacidade de implementar um canal de retorno por meio de conexões TCP.

É pela utilização dos scripts NCLua que é possível utilizar chamadas a rotinas gráficas, utilizadas neste trabalho e citadas anteriormente na Seção 2.3.1. Este tópico voltará a ser abordado no Capítulo 5.

2.4 Arquitetura do Set-Top Box

Uma visão geral da plataforma de hardware do STB é mostrada na Figura 4, na página 22. Ela inclui um processador Leon-3; um controlador de memória multi-clientes (MMC); um demultiplexador de TS MPEG2; decodificadores de áudio, vídeo e legendas; e um módulo de pós-processamento de vídeo.

O funcionamento começa com o demultiplexador. Ele tem a finalidade de extrair do TS MPEG2 enviado pela emissora os ES de áudio, vídeo e legendas, enviando-os aos decodificadores apropriados. O decodificador de vídeo, o processador, e o módulo de pós-processamento gráfico compartilham uma memória externa cujo acesso é controlado pelo MMC. A saída de vídeo resultante é gerada pelo módulo de pós-processamento gráfico, a partir da saída de vídeo do decodificador (canal *VID* na figura), de gráficos gerados pelo processador (OSD, de *On-Screen Display*) e das legendas decodificadas.

Uma explicação detalhada da arquitetura do hardware é encontrada em (BONATTO et al., 2013). Para este trabalho, é importante entender o conceito de memória multi-clientes, apresentar o processador, e entender como é realizada a composição com a saída do decodificador de vídeo. Eles serão tema das próximas seções. Para mais informações sobre o decodificador de vídeo e aúdio, consulte (SOARES; BONATTO; SUSIN, 2011) e (RENNER; SUSIN, 2012).

2.4.1 Controlador de memória multi-clientes

Sistemas de processamento de vídeo necessitam de uma hierarquia de memória eficiente a fim de atingir o desempenho necessário para a decodificação de vídeos de alta resolução. Por exemplo, os macroblocos decodificados são utilizados como referência

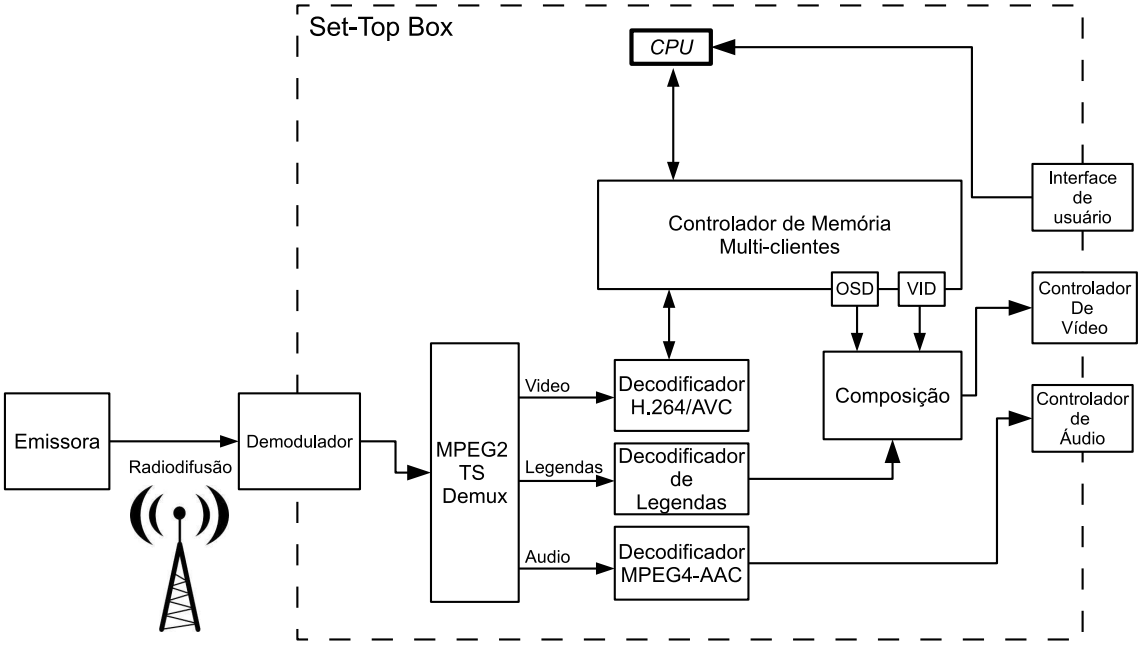

Figura 4: Arquitetura do Set-Top Box do SBTVD (NEGREIROS et al., 2012)

durante processo de decodificação e também pelo módulo de pós-processamento gráfico para gerar a saída de vídeo. Ou seja, pode-se aumentar a eficiência do sistema reutilizando os macroblocos lidos durante a decodificação também no processo de pós-processamento, economizando acessos de leitura.

Em (BONATTO; SOARES; SUSIN, 2011), é proposto o controlador de memória multi-clientes utilizado no STB. O controlador proposto cria uma hierarquia de memória em que acessos de leitura a macroblocos sejam bufferizados para uso, tanto pelo decodificador, quanto para a exibição. Além disso, o controlador implementa um *árbitro* para selecionar qual módulo tem a vez de utilizar a memória. Esse árbitro utiliza dois critérios: taxa de utilização de memória pelo módulo e uma classificação do módulo entre *sensível à latência* ou *sensível à banda*.

Módulos sensíveis à latência são aqueles que possuem requisitos de tempo real, onde atrasos oriundos das características das memórias DDR SRAM afetam o sistema. Já os módulos sensíveis à banda são aqueles que exigem alta troca de dados com a memória. Com isso, o árbitro prioriza módulos com requisitos de tempo real, dando-lhes à capacidade de interromper operações de módulos sensíveis a banda. Com a taxa de utilização da memória, obtida por meio de um estudo do padrão H.264 mostrada também em (BONATTO; SOARES; SUSIN, 2011), define-se a prioridade que o árbitro dará aos módulos sensíveis à banda.

O controlador de memória voltará a ser abordado no Capítulo 5, quando será explicada a interface de comunicação que o controlador possui com os clientes.

2.4.2 Pós-processamento gráfico

O pós-processador gráfico tem a finalidade de gerar a saída de vídeo final. Ele é apresentado em (NEGREIROS et al., 2012) e sua arquitetura é mostrada na Figura 5. O vídeo final é uma composição entre a saída do decodificador de vídeo (*Video* na figura), de uma saída gráfica gerada localmente pelo processador e escrita na região de memória *On-Screen Display* (OSD), e das legendas recebidas. O quadro descrito pela OSD possui

resolução de 720x480, com a descrição de uma imagem por macroblocos e pixel YCbCr com amostragem 4:4:4, com 8 bits para as amostras de Y, Cb e Cr.

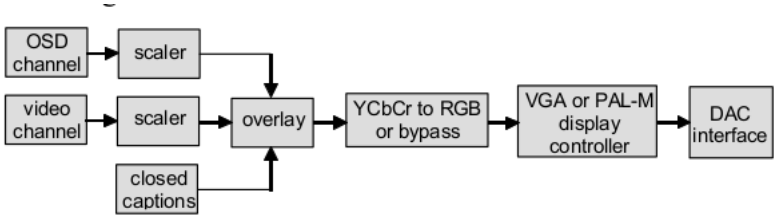

Figura 5: Arquitetura do módulo de pós-processamento gráfico (NEGREIROS et al., 2012).

O bloco *scaler* tem a finalidade de ajustar as resoluções da saída do decodificador e da saída do processador para a configuração atual de vídeo. A composição entre o gráfico da CPU e do decodificador é determinada pelos 2 bits menos significativos das amostras de Cb e Cr de cada pixel do quadro na OSD, resultando em 16 combinações possíveis de *alpha* para composição. O valor do pixel final enviado para a saída de vídeo é dado pela Equação 1. Por fim, são feitas as adaptações necessárias para a exibição, representados pelos blocos de conversão de espaço de cor YCbCr-RGB e para PAL-M.

$$Pixel\_Saida = Pixel\_OSD + (1 - Alpha) * Pixel\_Video \quad (1)$$

O módulo de pós-processamento voltará a ser será abordado novamente no Capítulo 4, quando serão apresentadas as mudanças necessárias para integrar a saída gráfica do Ginga-NCL com a saída de vídeo.

2.4.3 Processador

O processador é de vital importância para este trabalho, pois é devido a ele que é possível unificar o middleware ao STB, portando a implementação de referência para o STB. Esta seção tratará de apresentar com maiores detalhes este processador e o ambiente que o cerca: o Leon-3.

O Leon-3 é um processador de 32 bits com arquitetura SPARC v8 (SPARC INTERNATIONAL, 1992), que tem como principal diferencial a existência de unidades de multiplicação e divisão inteiras. O processador faz parte de um conjunto de *Intellectual Properties* (IPs) da Aeroflex Gaisler chamada GRLIB (AEROFLEX GAISLER, 2013a,b), distribuída sob licença GPL.

A principal característica do Leon-3 é ser centrado no barramento, ou seja, todos os outros IPs disponibilizados pela Gaisler se comunicam com o processador pelo mesmo barramento: o AMBA versão 2.0 (ADVANCED RISC MACHINES, 1999). A Figura 6 ilustra o Leon-3 segundo esse conceito, de acordo com a plataforma de hardware atual do STB. O barramento AMBA é dividido em dois tipos: o *Advanced High-performace Bus* (AHB) e o *Advanced Peripheral Bus* (APB).

O AHB é um barramento voltado a alto desempenho, tendo como características principais permitir escritas e leituras em rajadas e múltiplos *mestres*. Um mestre é um dispositivo capaz de iniciar operações de leitura ou escrita no barramento. Um árbitro é utilizado a fim de que somente um mestre utilize o barramento por vez. Um dispositivo mestre possui um ou mais dispositivos *escravos*. Dispositivos escravos podem comunicar-se entre si somente se possuírem diferentes mestres.

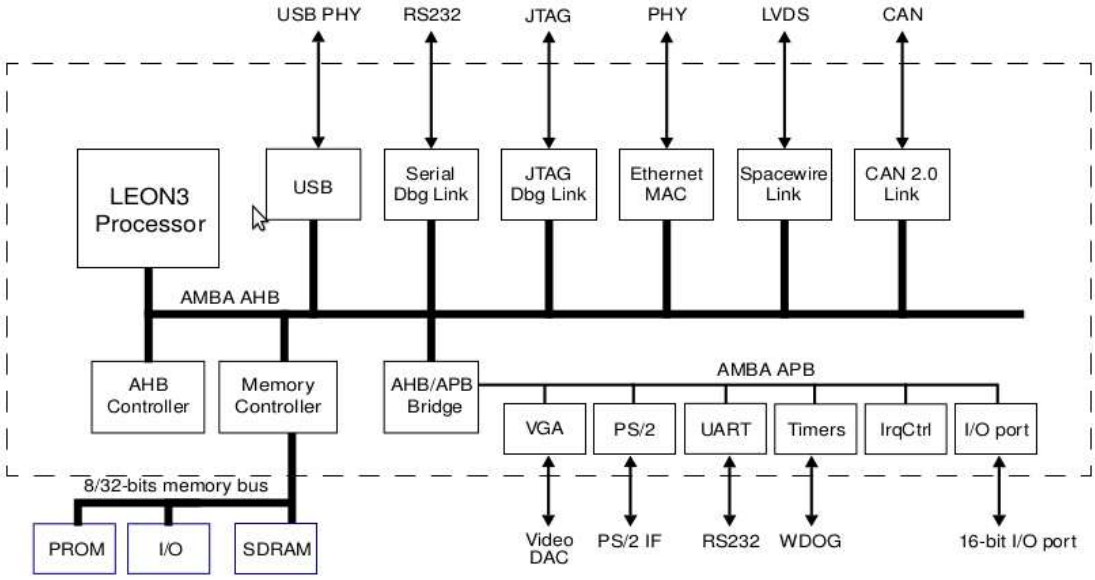

Figura 6: Processador Leon3 e o ambiente de IPs da GRLIB(AEROFLEX GAISLER, 2013a)

Já o APB tem como finalidade integrar periféricos que realizam baixa troca de dados com o processador. Diferentemente do AHB, o APB possui apenas um mestre, que funciona como uma "ponte"entre o APB e o barramento AHB. Os dispositivos conectados ao APB funcionam então como escravos da ponte.

Para adicionar um novo hardware ao Sistema-em-Chip (SoC, do inglês *System-on-Chip*) da GRLIB é necessário, primeiro, escolher em qual barramento colocá-lo. Isso requer implementar todo o protocolo de comunicação entre o barramento e o dispositivo. Este trabalho adotou um atalho para evitar essa necessidade, simplificando o processo de adição de hardware. Isso será abordado no Capítulo 5.

2.5 Contribuições do Trabalho

A Figura 7 mostra a arquitetura anteriormente vista na Figura 4 com as adições deste trabalho. Os componentes em software do middleware Ginga-NCL agora existem dentro do sistema e, por ser um pré-requisito para a adição deste último, também foi adicionado um sistema operacional. Além disso, um hardware acelerador para o middleware, chamado *Gingaccel*, foi criado e adicionado ao sistema, tendo sido criados e adicionados novos *drivers* de dispositivo no sistema operacional.

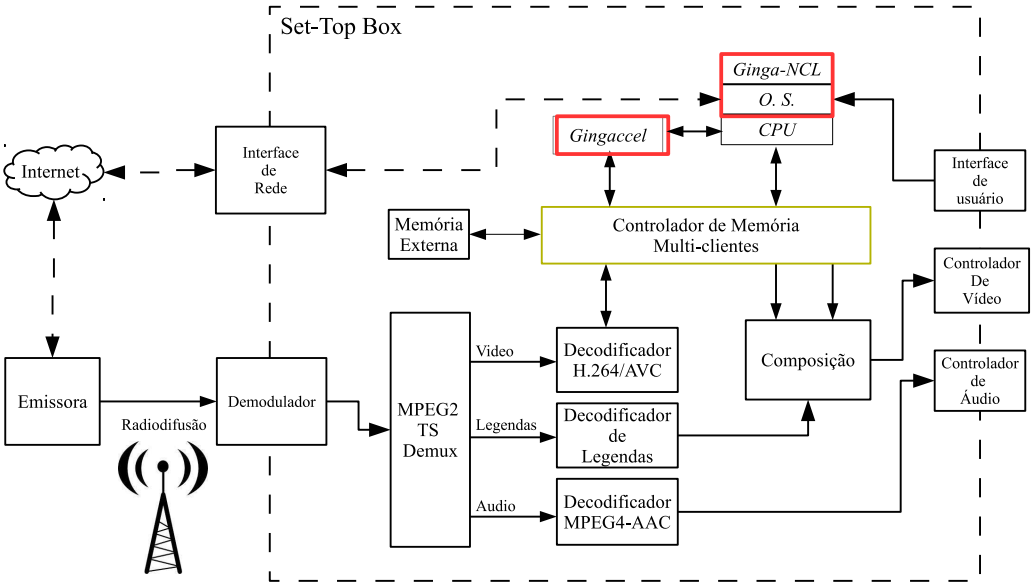

Figura 7: STB com as contribuições deste trabalho

3 TRABALHOS RELACIONADOS

Este capítulo tem a finalidade de mostrar trabalhos sobre a construção de sistemas de televisão digital interativos utilizando o middleware Ginga-NCL.

O trabalho (CRUZ; MORENO; SOARES, 2008) apresenta a primeira implementação de referência do Ginga-NCL para dispositivos portáteis, com o intuito de servir como referência a outras implementações do Ginga-NCL para esses dispositivos no futuro. Para isso, foi escolhido o sistema operacional para celulares Symbian, em detrimento aos sistemas Windows Mobile e Linux, com o argumento de que o Symbiam possuía uma comunidade maior e uma oferta de dispositivos para desenvolvimento maior na época do trabalho. O trabalho cita que a implementação foi portada e executada com sucesso *Nokia N81 1G* e *Nokia 5700 Express Music*.

A implementação utilizou a plataforma de desenvolvimento Symbian C++ e foi baseada no trabalho em (SOARES; RODRIGUES; MORENO, 2007b) para dispositivos fixos. Para isso, foi necessário adaptar as estruturas utilizadas originalmente por estruturas equivalentes no Symbian. Uma das principais diferenças encontradas foi que, no Symbian, não existia a biblioteca STL (*Standard Template Library*)do C++, largamente utilizada na implementação original. A solução inicialmente adotada no trabalho consistiu em mapear as chamadas à funções da STL por equivalentes no Symbian, consumindo tempo elevado de desenvolvimento. Posteriormente, um porte da biblioteca STL para o Symbian surgiu, chamado de STLPort (STLPORT, 2014), e, a partir desse ponto, o trabalho passou a utiliza-la.

Comparando com o desenvolvimento deste trabalho, não houve problemas devido a STL pois, como será visto mais adiante no capítulo 4, as ferramentas de desenvolvimento utilizadas eram compatíveis com a implementação de referência do middleware. Em contrapartida, é necessário que a *cross-compilação* dos códigos-fonte do middleware não entrem em conflito com o sistema-nativo. Ou seja, ao ser utilizada uma biblioteca, deve-se utilizar a biblioteca do sistema operacional embarcado, e não aquela do computador no qual se está trabalhando. Isso não foi observado no código original e teve que ser modificado.

Outra importante mudança feita na implementação foi no escalonador de tarefas do NCL. Originalmente, o escalonador era implementado utilizando a biblioteca *pthreads* e, embora houvesse suporte para ela, o trabalho cita que isso incorria em perda de desempenho e, por isso, a implementação foi adaptada para utilizar *Active Object*, uma estrutura específica do sistema operacional Symbian.

Para fins de trabalho nesta dissertação, é interessante a descrição de como é utilizada a biblioteca *pthreads* para implementar a sincronização temporal: para um dado intervalo de tempo, é criada uma *thread* indicando o início e o fim desse intervalo, de forma que esses eventos sejam informados ao formatador NCL. Essa característica será lembrada

novamente na Seção 6.1, quando serão abordados os testes utilizando programas NCL. .

Uma outra adaptação foi necessária para realizar o *parsing* da linguagem NCL. Como dito na Seção 2.3, a linguagem é baseada em XML e, segundo a argumentação do trabalho, a solução adotada inicialmente aloca o programa NCL inteiro na memória, o que era indesejável dada a limitação de memória dos dispositivos utilizados. Em contrapartida, nesta dissertação isso não foi um problema - enquanto que o máximo de memória dos dispositivos utilizados era 96 MB, na plataforma de desenvolvimento aqui utilizada é disponibilizado 512 MB.

Outra questão levantada foi a dos *players*. No trabalho, eles acabaram sendo limitados pelo Symbian, ou seja, foram implementados *players* somente de mídias com codecs disponibilizados pela plataforma. Assim como no caso da STL, o fato de terem sido utilizadas ferramentas compatíveis com a implementação de referência fez com que praticamente todos os *players* fossem implementados. As únicas exceções foram os de áudio e vídeo os quais irão requerer o desenvolvimento de drivers de dispositivo adequados.

O trabalho termina argumentando que, para implementações do Ginga-NCL, os três principais pontos a serem observados são 1) o escalonador de tarefas; 2) a manutenção do programa NCL na memória; e 3) as características gráfics do dispositivo embarcado. Dentre esses, os pontos 1 e 3 foram notados, como será visto com o decorrer deste texto.

O trabalho de (LIM; LEE, 2011) implementa um receptor Ginga-NCL para dispositivos móveis. A arquitetura do protótipo criado pode ser vista na Figura 8. Os principais componentes do receptor são o Formatador NCL (*NCL Formatter* na figura); o Gerenciador de Bases Privadas (*Private Base Manager*, ou PMB) e o Manipulador DSM-CC (*DSM-CC Handler*).

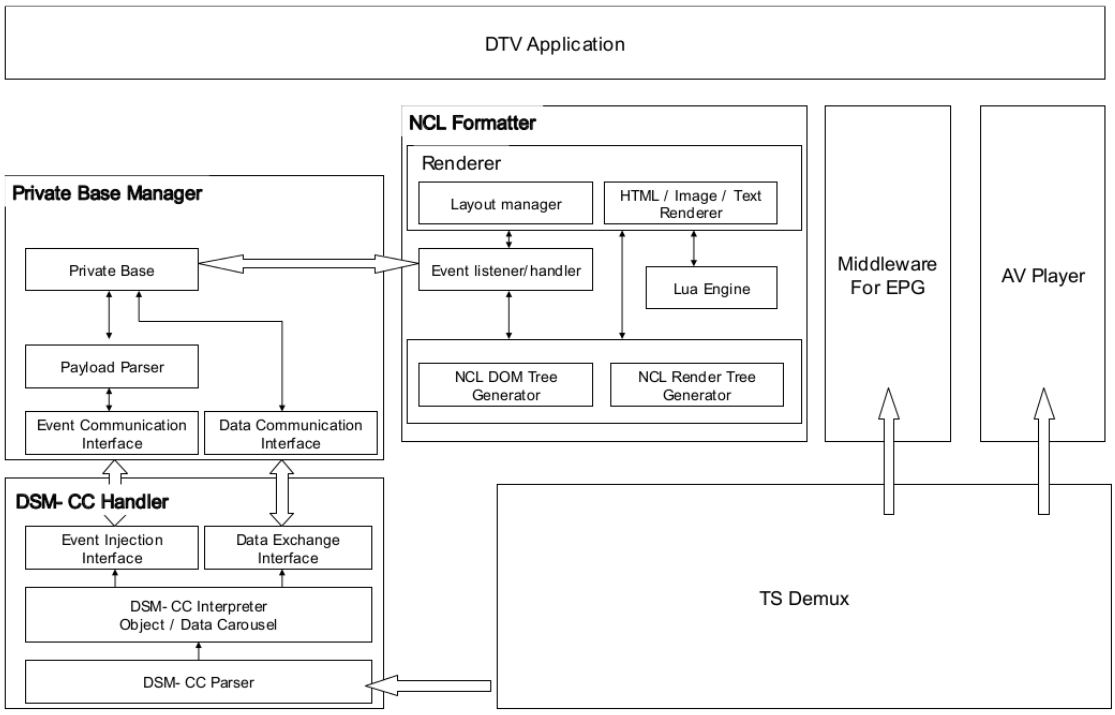

Figura 8: Arquitetura do receptor Ginga-NCL de (LIM; LEE, 2011)

O Formatador já foi apresentado anteriormente na Seção 2.3.1 e tem a finalidade de orquestrar a apresentação multimídia. No trabalho, o programa NCL é recebido da emissora via TS MPEG-2; nesse caso, o componente *middleware for EPG* analisa o TS recebido e,

quando detecta que há dados de um programa NCL sendo transmitidos, indica ao Demux para separar e passa-los ao Manipulador. O Manipulador então passa os dados referentes ao programa ao PMB, cuja responsabilidade é manter informações acerca do estado do programa NCL. O PMB comunica ao Formatador o término do recebimento dos programas, quando então inicia a apresentação. O Manipulador também tem a finalidade de receber o sistema de arquivos recebido pelo protocolo DSM-CC (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 1998), pelo qual são recebidos, além do programa NCL da emissora, as demais mídias necessárias para a execução do programa.

Comparando este trabalho com o de (LIM; LEE, 2011), aqui não há ainda preocupação com o recebimento de programas NCL por meio do TS MPEG-2. O principal objetivo foi a de adaptação de componentes de software já existentes para um novo hardware e, além disso, no versão projeto utilizado não havia um Demux, como dito anteriormente em 2.4.

O desempenho da implementação feita foi obtido segundo os parâmetros de QoE (*Quality of Service*) citados no trabalho. Esses parâmetros e os resultados são vistos na Tabela 2. A coluna nomeada "Sem Cache"indica o caso em que não há dados do programa já armazenados no sistema, enquanto que a "Com cache"indica o contrário. Uma vez que o recebimento de aplicações não foi abordado no trabalho, para fins de comparação foi utilizado somente o tempo de inicialização. Isso voltará a ser abordado no Capítulo 6.

Tabela 2: Resultados da implementação do receptor Ginga-NCL de (LIM; LEE, 2011)

	Sem Cache	**Com Cache**
Tempo de inicialização da aplicação de TVD	52.7 s	6.8 s
Tempo de *download* do documento NCL	300 s	0.5 s
Tempo de troca de canal	5.6 s	

4 ADAPTAÇÃO DO MIDDLEWARE

A primeira etapa deste trabalho concentrou-se em adequar a implementação de referência do Ginga para o ambiente de hardware do SoC. Para atingir esse objetivo, partiu-se do ambiente de desenvolvimento padrão para o processador Leon-3 fornecido pela Aeroflex Gaisler, que será apresentado em primeiro lugar, junto com a plataforma de desenvolvimento.

Logo após, será feita uma apresentação do sistema de distribuição de software utilizado pelo middleware. Isso é importante pois o trabalho de adaptação envolveu modificar configurações de instalação a fim de que elas se tornassem compatíveis com as ferramentas da Gaisler.

Apresentadas as ferramentas, partir-se-á para o estudo do método de instalação do middleware, com o objetivo de dominar todos os passos necessários para a sua instalação. Desse modo, inicia-se então a etapa de integração desse método com o ferramental da Gaisler. Isso envolve duas tarefas: escrever as regras estudadas de acordo para as ferramentas e corrigir problemas de compatibilidade encontrados no processo.

4.1 Plataforma de desenvolvimento

Antes de começar a falar sobre o trabalho de porte, esta seção irá apresentar a plataforma de hardware e o ambiente de desenvolvimento no qual ele foi desenvolvido.

A plataforma de hardware do STB utilizada neste trabalho é baseada na arquitetura de (SOARES; BONATTO; SUSIN, 2013), sem o decodificador de áudio AAC e o demultiplexador de TS MPEG2. A plataforma é implementada em VHDL numa placa de FPGA Virtex-6, utilizando 37,420 *slice registers* e *47,406 slice LUTs*, com uma ocupação final de 53% do FPGA XC6VLX240T. O processador Leon-3 e o decodificador funcionam a 75 MHz, enquanto que módulo MMC a 200 MHz.

O ambiente de desenvolvimento de software utilizado é *Linuxbuild* (AEROFLEX GAISLER, 2013c), fornecido pela Aeroflex Gaisler. Ele é composto por uma série de outras ferramentas que, juntas, visam criar uma imagem de uma distribuição Linux para desenvolvimento em FPGA. As ferramentas que compõem o sistema e os seus respectivos papéis dentro do Linuxbuild são:

- **Toolchain sparc-linux-gcc**
  Uma Toolchain permite desenvolver aplicações para um sistema embarcado e é composta por um *cross-compilador*, que gera código para a arquitetura do processador embarcado; uma biblioteca padrão para a linguagem C; e cabeçalhos e bibliotecas do sistema operacional embarcado.

Neste trabalho, a Toolchain utilizada chama-se *sparc-linux-gcc*, criada pela Gaisler com a finalidade de servir a diferentes versões e configurações do processador Leon. A configuração utilizada neste trabalho corresponde a arquitetura SPARC v8 do Leon-3, sem ponto flutuante.

- **Buildroot** (BUILDROOT, 2014)
  O Buildroot é um conjunto de scripts e Makefiles (FREE SOFTWARE FOUNDATION, 2014a) que visam gerar uma distribuição Linux embarcada. Ou seja, o sistema operacional junto com o conjunto final de aplicações e bibliotecas para que o sistema cumpra com seus objetivos. Dentro do Linuxbuild, o Buildroot gera o sistema de arquivos da distribuição, sem o kernel.
- **Linux versão 2.6.34.4**
  Após gerar a imagem do sistema de arquivos, o Linuxbuild se encarrega de gerar o Linux. Por sua vez, a compilação do Kernel utiliza a imagem do sistema de arquivos como um *RAM filesystem*. O RAM filesystem normalmente é utilizado para ajudar a inicializar o sistema real. Aqui, ele é utilizado como o sistema de arquivos final de desenvolvimento.
  Ao final desta etapa é gerado a distribuição completa, no entanto, sem o *bootloader*, ou o inicializador do sistema.
- **mkprom2** (AEROFLEX GAISLER, 2013d)
  Esse programa transforma a imagem final gerada pela compilação do Linux em uma imagem carregável no FPGA pela ferramenta *grmon* (AEROFLEX GAISLER, 2013e), também da Gaisler. Esse processo consiste principalmente em adicionar o bootloader que faltava no passo anterior.

Dentre essas ferramentas, a que ocupou maior destaque neste trabalho é o *Buildroot*, uma vez que o processo de porte consiste em adicionar suporte para a compilação do Ginga-NCL a essa ferramenta, tema da próxima seção.

4.2 Adição do Ginga-NCL ao Buildroot

O processo de porte do middleware para o Buildroot passa pela necessidade de entender o seu processo de compilação. Para realizar essa análise, foi utilizado o método de compilação para distribuições Linux baseadas no Debian (DEBIAN, 2014), encontrados nos Anexos A e B.

Por necessitar de um conjunto de bibliotecas e um sistema operacional, o primeiro passo é portar as dependências do Ginga-NCL. Esse trabalho será apresentado em primeiro lugar. Com as dependências satisfeitas, pode-se partir para o porte do middleware em si, que será abordado em seguida. Finalmente, será mostrada a interface gráfica criada com o trabalho realizado.

4.2.1 Dependências

Dentro do sistema estudado, a instalação das dependências é feita em duas etapas: primeiro, instalando aquilo que é disponibilizado pela ferramenta de instalação de softwares do Debian; e depois compilando o código fonte do restante. O script original utilizado para esta análise é o encontrado no Anexo A.

A Tabela 3 mostra a relação completa de bibliotecas e ferramentas instaladas pelo instalador do Debian segundo o script. Os nomes das dependências estão escritas na primeira coluna, de acordo com os nomes observados no script. A segunda coluna, intitulada "Status", indica a situação da referida dependência para a configuração final deste trabalho.

Tabela 3: Status do porte das bibliotecas dentro da ferramenta Buildroot

Nome	Status	Comentários
libssl-dev	Existente	
libcurl4-openssl-dev	Existente	
libreadline6-dev	Existente	
libexpat1-dev	Existente	
libxerces-c2-dev	Existente	
libmad0-dev	Existente	
libtiff4-dev	Existente	
libkrb5-dev	Portada	
libgpm-dev	Portada	
x11proto-xext-dev	Desabilitada	Biblioteca gráfica escolhida é o DirectFB
libxext-dev	Desabilitada	Parte do item anterior
libpng12-dev	Existente	
libjpeg62-dev	Existente	
libfreetype6-dev	Existente	
libavcodec-dev	Existente	Pertence à ffmpeg
libavformat-dev	Existente	Pertence à ffmpeg
libxine-dev	Desabilitada	Diversos codecs de vídeo em software
libxine1	Desabilitada	Igual ao anterior
libxine1-ffmpeg	Desabilitada	Igual ao anterior
libdirectfb-extra	Portada	
libtool	Desnecessária	Compilação automática pelo Buildroot
liblua5.1-0-dev	Existente	
libzip-dev	Existente	
libzip1	Existente	
libavcodec-extra-52	Desabilitada	Parte da ffmpeg
libavformat-extra-52	Desabilitada	Parte da ffmpeg
ffmpeg	Desabilitada	Codec MPEG em software
libvorbis-dev	Existente	

Dependências marcadas somente como "Existentes"existem no Buildroot e são selecionadas em sua interface de configuração para serem compiladas, fazendo parte do sistema embarcado final. Dependências marcadas como "Portadas"não existiam na ferramenta e tiveram que ser portadas, e também fazem parte do sistema final. Dependências marcadas como "Desabilitadas"não fizeram parte da configuração final e são devidamente acompanhadas de comentários explicativos.

Alguns casos especiais são devidamente assinalados. Por exemplo, as bibliotecas da ffmpeg existem no Buildroot mas, por serem o codec de vídeo utilizado pela implementação de referência, futuramente elas seriam substituídas pelos codecs em hardware do STB. Logo, optou-se por já removê-las da configuração final.

Com as dependências resolvidas, pode-se partir para a geração do middleware propriamente dito, tema da próxima seção.

4.2.2 Middleware

A compilação final do Middleware é feita com a ajuda do script encontrado no Anexo B. Os códigos-fonte utilizados foram obtidos de (TELEMIDIA LABS, 2014a). A informação mais importante a ser extraída do script em anexo é a ordem com que os componentes do Ginga-NCL devem ser compilados. Eles são mostrados abaixo, de acordo com o nome do componente obtido em (TELEMIDIA LABS, 2014a), e são:

1. telemidia-util-cpp
2. gingacc-system
3. gingacc-cm
4. gingacc-mb
5. gingacc-contextmanager
6. gingacc-ic
7. gingacc-um
8. gingacc-tuner
9. gingacc-tsparser
10. gingacc-dataprocessing
11. gingacc-multidevice
12. gingacc-player
13. ncl30-cpp
14. ncl30-converter-cpp
15. gingancl-cpp
16. gingalssm-cpp
17. ginga-cpp

Para adicionar um novo componente ao Buildroot, deve-se criar um arquivo de configuração *Config.in* e um Makefile apropriado. O arquivo de configuração exporta para o sistema a existência do componente e também opções de configuração selecionáveis pela interface gráfica. O arquivo de configuração utiliza a sintaxe dos arquivos de configuração do Kernel Linux (TORVALDS, 2014). Já os Makefiles exportam as variáveis que são utilizadas para gerar os passos de instalação de forma automática.

Logo, para portar o Ginga-NCL para o Buildroot teve que ser criado um Makefile e um Config.in para cada um dos componentes acima citados. As características gerais dos Makefiles criados para os componentes acima foram:

- **Versão do componente** Este valor é constante para todos os componentes, e corresponde a versão "0.13.2"do middleware, obtida no final do ano de 2012.
- **Subdiretório**. Necessário para os componentes do Common Core (prefixados por gingacc) e pelo interpretador NCL (prefixados por ncl30), pois o código-fonte de ambos contém todos os componentes, sendo necessário explicitar em qual pasta está o componente a ser gerado.
- **Dependências**. Cada componente tem como única dependência o item imediatamente anterior na ordem de compilação vista acima. O primeiro item "telemidia-util-cpp"constitui uma exceção, incluindo também as dependências do Ginga-NCL vistas na Seção 4.2.1.
- **Reconfiguração**. Os scripts de configuração e Makefiles do Ginga-NCL não são distribuídos por padrão, tendo que ser criados antes da compilação ser iniciada. Esta opção do Buildroot se encarrega de cumprir essa tarefa.
- **Opções de configuração**. Inicialmente, somente as opções observadas no script estudado foram adicionadas para configuração pela interface gráfica. As demais foram desabilitadas no Makefile de forma estática.

- **Variáveis de ambiente**. Uma das possibilidades de configuração que o Buildroot proporciona são a definição de variáveis de ambiente UNIX durante o processo de compilação. No trabalho de porte, elas fizeram parte da solução adotada para resolver as incompatibilidades das configurações das autotools durante a cross-compilação.

  As autotools são ferramentas para gerar o sistema de distribuição de software livre da GNU, e são compostas pelas ferramentas automake (FREE SOFTWARE FOUNDATION, 2014b) e autoconf (FREE SOFTWARE FOUNDATION, 2014c). A raiz das incompatibilidades encontradas está no fato de que as autotools devem ser genéricas, ou seja, independente de como as dependências estejam instaladas no sistema. Esse fato não foi observado nas configurações do Ginga-NCL, quebrando os mecanismos utilizados pelo Buildroot para cross-compilação.

  Para solucionar isso, as configurações antigas foram substituídas por configurações computadas pelo Buildroot durante o tempo de compilação da imagem Linux, utilizando para isso variáveis de ambiente. Para maiores detalhes, consulte o Apêndice A.

A interface gráfica resultante gerada após a adição de todos esses Makefiles e arquivos de configuração pode ser vista na Figura 9.

Figura 9: Interface gráfica criada para gerar o Ginga-NCL no Buildroot

4.3 Subsistema gráfico

Uma parte importante do Ginga que deve ser definida é a implementação de sua parte gráfica, pois ela é o primeira saída visível. O middleware oferece duas alternativas: utilizar ou a biblioteca gráfica DirectFB (DIRECTFB, 2014) ou a biblioteca SDL (SDL, 2014). O DirectFB é uma biblioteca gráfica voltada para sistemas embarcados, sendo que uma de suas características é estar preparada para substituição de suas implementações em software por equivalentes em hardware. Já a SDL é uma biblioteca que visa ao desenvolvimento de aplicações multimídia *cross-platform*, ou seja, portáveis entre plataformas. Para isso, ela implementa acesso ao hardware gráfico e de entrada e saída por meio de outras bibliotecas, como OpenGL, Direct3D e o próprio DirectFB.

Uma vez que o sistema-em-chip é um sistema embarcado, o DirectFB se propõe a facilitar , e o a SDL atuaria como mais uma camada desnecessária entre o hardware e o middleware, optou-se por utiliza o DirectFB. Para que isso seja possível, é necessário que haja um driver de dispositivo *framebuffer*(VENKATESWARAN, 2008) no sistema. O Framebuffer é uma camada de abstração de hardware (HAL, de *Hardware Abstraction Layer*) gráfico do Linux que corresponde a um buffer de pixels daquilo que é exibido na tela.

Dentro da arquitetura original do Set-Top Box, o Ginga-NCL escreve sua saída gráfica na região da memória OSD. Inicialmente, essa região era lida pelo módulo de pós-processamento gráfico da mesma maneira que o decodificador de vídeo H.264, o que traz dois inconvenientes:

- No H.264, os pixels são organizados em macroblocos, um mapeamento desconhecido pelo HAL do framebuffer.
- Dentro da arquitetura, os pixels da região OSD estão no formato YCbCr com subamostragem 4:4:4. Isso vai de encontro ao padrão utilizado pelo DirectFB, que utiliza espaços de cor derivados do RGB.

Uma opção para contornar esses inconvenientes seria adaptar o software de acordo com as características do hardware do Set-Top Box. Duas opções inicialmente foram levantadas: adaptar o framebuffer de forma com que as alterações nele fossem transparentes ao DirectFB ou adaptar o próprio DirectFB diretamente. Porém, elas possuem as seguintes desvantagens:

- O DirectFB mapeia diretamente a região de memória que corresponde ao framebuffer para então fazer as operações gráficas nela. Para utilizar mapeamentos de pixel customizados no framebuffer, é necessário implementar e chamar os métodos específicos *fb_write* e *fb_read*. Isso iria requerer alterar, além do framebuffer, a implementação de funções básicas da biblioteca - ou seja, se perderia a transparência desejada.
- Alterar somente o DirectFB, além de recorrer a mesma complexidade citada no parágrafo anterior com relação remapeamentos de pixels, iria adicionar a necessidade de ser criado um tipo de pixel específico para a arquitetura.
- Em qualquer uma das alternativas, é necessário implementar conversão de espaços de cor, o que resulta em uma queda de desempenho devido a inexistência de uma FPU.

Por outro lado, também existe a possibilidade de alterar o hardware do Set-Top Box de acordo com as características dos softwares utilizados. Isso traz as seguintes vantagens e desvantagens:

- A implementação do framebuffer se torna imediata pois apenas é necessário imple-

mentar o método de mapeamento de memória da OSD para o espaço de usuário, o único requisito do framebuffer.

- Não é necessário alterar o DirectFB, uma vez que o framebuffer seria alterado para um espaço de cor e organização de pixels conhecidos.
- Não seriam utilizadas operações com a FPU.
- Segundo a (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2007d), pode-se utilizar o tipo de pixel ARGB32 para a saída gráfica do middleware, um pixel que existe dentro do DirectFB.
- Em contrapartida, é necessário alterar o módulo de pós-processamento gráfico para explorar essas vantagens.

Dadas essas alternativas, em um trabalho conjunto com o LaPSI optou-se por alterar a maneira com que o módulo de pós-processamento gráfico lia a região de memória OSD. Tais mudanças foram:

- Pixels organizados no formato de *raster scan*, em substituição ao em macroblocos. O raster scan consiste em ler os pixels da esquerda para direita, de cima para baixo.
- Pixels individualmente lidos no espaço de cor ARGB32, onde "A"corresponde a um canal *Alpha*, e cada canal de cor ARGB possui 8 bits de profundidade. O canal Alpha passa a ser utilizado para determinar o grau de composição entre a saída gerada pelo processador e pelo decodificador de vídeo, seguindo a mesma fórmula mostrada na Equação 1

Com essas alterações, o trabalho de porte do middleware Ginga-NCL foi terminado com sucesso. Os resultados obtidos por esta mudança são mostrados mais adiante, na Seção 6.2. A próxima etapa é acelerar o middleware dentro da nova plataforma de hardware, tema do próximo capítulo.

5 ACELERAÇÃO

Com os componentes de software do middleware portados, abre-se a possibilidade de acelerar a execução das aplicações de TVD. Dentro desse contexto, o próximo objetivo deste trabalho foi acelerar uma função utilizada pelo middleware de acordo com as características do hardware do STB. Para isso, trabalhou-se sobre o subsistema gráfico do middleware, pois a utilização do DirectFB como base para sua implementação facilita adicionar um novo hardware e a substituir as implementações em software existentes. Este capítulo irá abordar todos os passos necessários para atingir esse objetivo em sequencia, os quais são:

1. Escolher a função a ser acelerada - esse será o assunto da Seção 5.1;
2. Estudo da arquitetura do STB. Necessário para entender onde se deve posicionar o hardware e para projetá-lo. Isso será tema das Seções 5.2 e 5.3.
3. Projeto e implementação do hardware, de acordo com o que foi estudado no item anterior. Isso tema da Seção 5.4.
4. Desenvolvimento de um Driver Linux para o novo Dispositivo. Neste trabalho, utilizou-se uma interface disponibilizada pela Gaisler chamada GRGPIO para ligar o hardware desenvolvido com o barramento APB. Logo, foi criado um driver para este dispositivo. Isso será tema da Seção 5.5
5. Substituição da implementação em software por uma utilizando o hardware. Isso é feito criando um driver do DirectFB que utilize o dispositivo adicionado ao STB pelo driver de dispositivo da GRGPIO criado no item anterior. Isso será tema da Seção 5.6.

5.1 Função acelerada

Esta seção tem a finalidade de apresentar a função que foi acelerada neste trabalho. Isso foi feito observando a lista de funções gráficas oferecidas pela API *NCLua*, que pode ser obtida em (SANT'ANNA et al., 2009; TELEMIDIA LABS, 2014b), e procurando o código fonte do middleware por sua implementação.

De todas as funções encontradas, escolheu-se a função *FillRectangle*, utilizada pelo *NCLua* na função *fillRect*. Os principais motivos para tal escolha foram:

- Maior simplicidade de implementação. Quando se escreve uma linha de pixels, o algoritmo é simplesmente uma série de escrita sequenciais na memória, e quando se passa para a linha de pixels seguinte o deslocamento é sempre constante em relação ao último pixel escrito.
- Oferece vantagens na implementação quando se considera o protocolo de escrita na MDDR, a ser apresentado na Seção 5.3.

- Possibilidade de estender o hardware para utilização em outras funções gráficas. Por exemplo, implementar uma função de limpeza na tela é escrever um retângulo preto nela. Pode-se também descrever algoritmos de renderização de outras figuras geométricas a partir da escrita de áreas de memória.
- Semelhança com o processo de leitura e escrita em blocos na memória. Isso permite estender o módulo no futuro para operações de escrita, não abordada neste trabalho. Com isso, abre-se a possibilidade de estender o módulo ainda mais, permitindo implementar a cópia de regiões de memória.

Com a função escolhida, começa-se o processo de projeto do hardware acelerador para ela. Primeiro, é necessário entender onde o módulo irá se posicionar dentro do sistema como um todo, tema da próxima seção.

5.2 Posicionamento do hardware dentro do SoC

A Figura 10 mostra como o novo hardware deve ser posicionado dentro do SoC. Independentemente do que será acelerado, para que o hardware utilize a memória é necessário que seja criado um novo *cliente* para ele na MDDR.

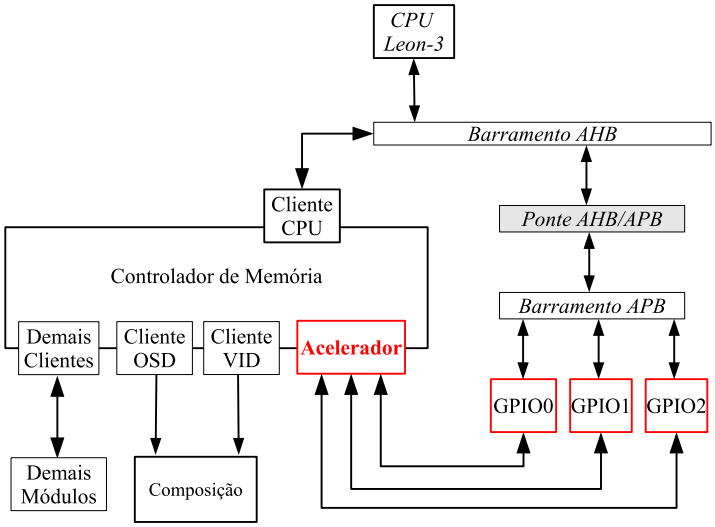

Figura 10: Posicionamento do hardware acelerador dentro do STB.

Além de criar um cliente, é preciso disponibilizar ao sistema operacional as interfaces do módulo necessárias para realizar a comunicação com o DirectFB. Neste trabalho, a solução adotada foi conectar essas interfaces ao módulo da Gaisler chamado de *GRGPIO* (*Gaisler Research General Purpose I/O*). A vantagem de utilizar essa interface é que ela já implementa a comunicação com o barramento AMBA do Leon-3. Em contrapartida, na época do trabalho o ambiente *Linuxbuild* não possuía um driver de dispositivo para a GRGPIO, torna1ndo necessária sua implementação. A implementação desse driver será vista mais adiante na Seção 5.5.

Antes, e assim como pode ser observado na Figura 10, para implementar o módulo acelerador é necessário entender como ocorre o processo de escritas e leituras na memória utilizando o controlador de memória. Esse assunto será abordado a seguir.

5.3 Interface com o controlador de memória

Como dito na Seção 5.2, o hardware utilizará a memória por meio de um cliente. Esta seção irá apresentar o protocolo de escrita na memória dos diversos clientes.

A interface de comunicação com o cliente pode ser observada na Figura 11. O sinal *clk_ddr* é o relógio da MDDR e o sinal *clk_ci* o relógio do cliente. O processo inicia com o cliente pedindo uma requisição (*ci_req=1*), sinalizando que a operação é de escrita (*ci_write=1*), já com os dados a serem escritos preparados. Isso corresponde a Etapa 1 e 2 sinalizada na figura.

Os dados constituem um vetor de 64 bytes (*ci_wr_data* na figura), o que traz a necessidade de definir quais desses dados serão de fato escritos. Isso é feito por meio de uma máscara (*ci_wr_mask* na figura) de 64 bits, onde cada bit corresponde a um byte do vetor de dados. Se o valor desse bit for igual a 1, o byte correspondente deve ser escrito.

Com a requisição feita, o comando volta ao controlador. O módulo deve, então, esperar uma sinalização indicando que ele já terminou de realizar a escrita (*mddr_ack=1*). Na figura, isso corresponde à Etapa 3. Quando o módulo detectar esse evento, ele sinaliza ao controlador o recebimento retirando a requisição (*ci_req=0*). Isso corresponde à Etapa 4. Por fim, o controlador recebe o último evento e retira a sinalização de recebimento (*ci_ack=0*), indicando que ele está pronto para receber novas requisições.

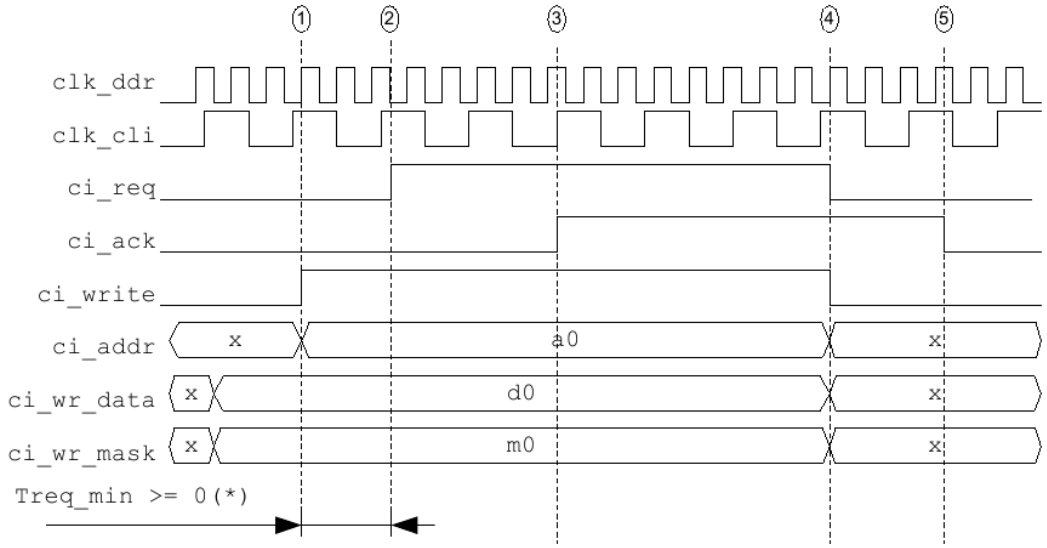

Figura 11: Protocolo de escrita do cliente para a MDDR

5.4 Projeto do hardware

Esta seção tratará sobre como a aceleração da função *fillRect* foi implementada em hardware.

A interface do cliente da MDDR aceita escritas de até 64 bytes. Isso significa que em um ciclo de escrita até 16 pixels podem ser escritos, já que o pixel do canal OSD possui 4 bytes, com cada byte correspondendo a um dos canais ARGB. Portanto, o problema de acelerar a função *fillRect* consiste em transformar as escritas que antes eram de pixel a pixel em escritas sequenciais de 16 pixels.

Uma vez que as escritas possuem exatamente 16 pixels, as máscaras podem ser definidas em termos de pixel: caso um determinado pixel tenha que ser escrito, o grupo de 4 bits correspondente na máscara terá nível lógico alto, representando os 4 canais utilizados por pixel na OSD.

Serão utilizadas, no máximo, 3 máscaras diferentes para escrita: uma para escrever o primeiro bloco de 16 pixels; outra para escrever o último bloco; e uma última para escrever quaisquer blocos intermediários. A máscara para os blocos intermediários é constante e possui todos os bits em nível lógico alto. Já as máscaras inicial e final podem ser calculadas a partir do primeiro endereço de escrita e pelo tamanho da linha de pixels.

A Figura 12 mostra o bloco acelerador implementado, indicando suas entradas, saídas e conexões dentro do STB. As principais entradas correspondem aos parâmetros de descrição de um retângulo da função *FillRectangle*. Já as saídas correspondem ao sinais para execução do protocolo de escrita na MDDR.

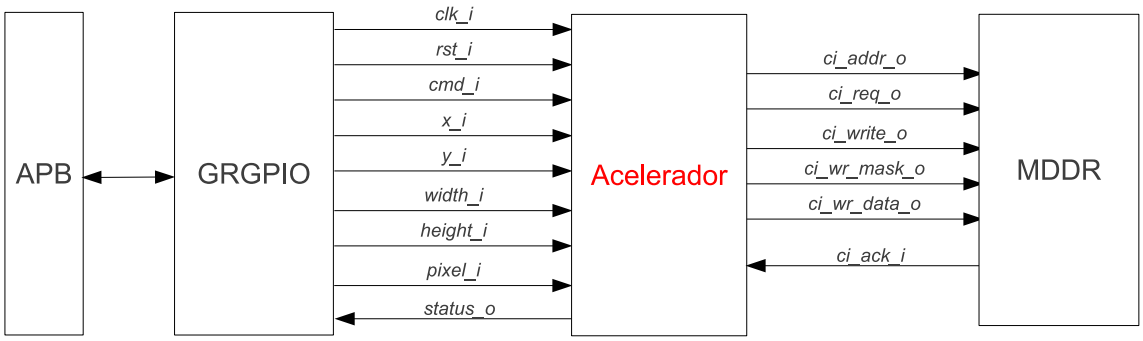

Figura 12: Diagrama de blocos do hardware acelerador.

Os sinais de entrada mostrados na Figura 12 são:

1. clk_i,rst_i : Os sinais de relógio e reinicialização do módulo.
2. x_i, y_i : O ponto superior esquerdo do retângulo.
3. width_i, height_i : As dimensões do retângulo a ser escrito, também em pixels.
4. pixel_i : A cor do retângulo, no formato ARGB, com 32 bits de profundidade.
5. ack_i : O sinal de ack do controlador de memória.
6. cmd_i : O sinal de comando para que o hardware inicie sua operação.
7. ci_rd_data_i : Dados lidos da memória. Não utilizado no trabalho mas adicionado a fim de completar a interface.

Já os sinais de saída são:

1. ci_addr_o: O endereço no qual deve ser feita a escrita
2. ci_write_o : Sinal que, quando igual a 1, indica que a operação é de escrita, e de leitura quando é 0.
3. ci_req_o : Sinal de requisição de operação à memória.
4. ci_wr_data_o : Dados a serem escritos.
5. ci_wr_mask_o : Máscara para indicar quais os bytes passados por ci_wr_data_o devem ser realmente utilizados para escrita.
6. status_o : Indica o status da operação iniciada com *cmd_i*. Quando tem valor em nível lógico alto, significa que a operação iniciada com "cmd_i=1"terminou.

5.4.1 Arquitetura

A arquitetura do hardware é mostrada na Figura 13. O processo *INIT* é combinacional e tem a função de transformar os parâmetros de entrada em coordenadas da MDDR. Ou

seja, ele transforma as operações de escritas por pixel em operações por *blocos de 16 pixels*. Dentro dessa lógica, o registrador *ncols* indica quantos blocos de 16 pixels ainda restam ser escritos na linha atual, enquanto que o registrador *nlines* indica quantas linhas ainda restam. Além disso, o registrador *ncols_reload* guarda o valor original calculado para *ncols* a fim de reinicia-lo a cada nova escrita de linha.

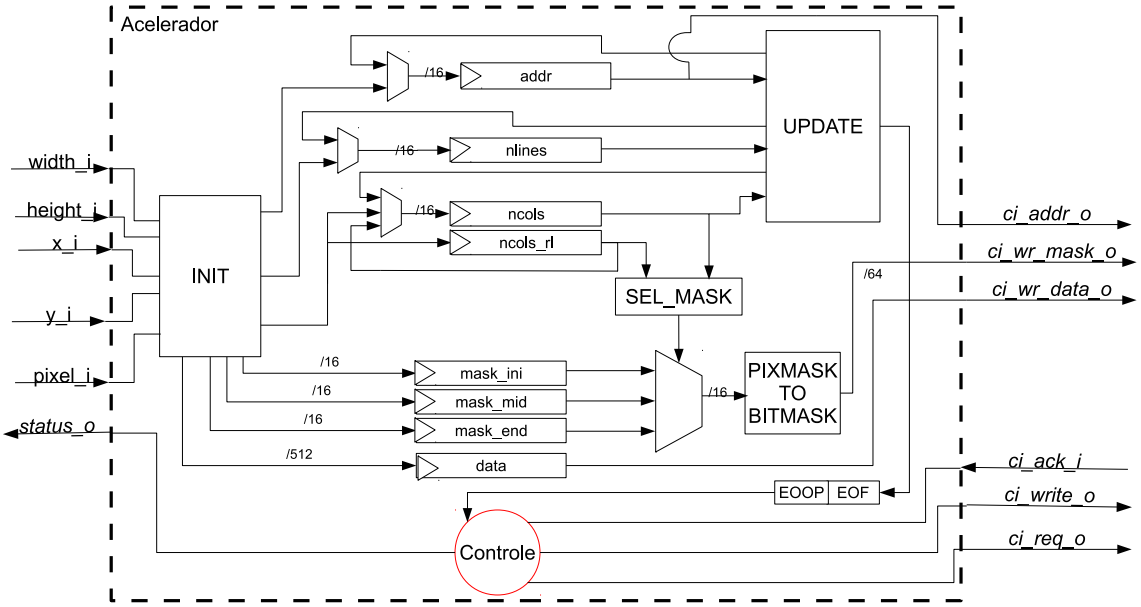

Figura 13: Arquitetura do Hardware

O registrador *data* contém os dados a serem escritos. Sua inicialização é feita no processo *INIT* e consiste em replicar o pixel de entrada, *pixel_i*, uniformemente ao longo do registrador.

O registrador *addr* tem a finalidade de guardar o endereço atual de escrita. Para incrementar para a próxima coluna de 16 pixels, os 6 bits menos significativos do endereço são desconsiderados, já que eles não são utilizados pela MDDR. Para incrementar para a próxima linha, ele soma ao primeiro endereço de escrita da linha o tamanho total de uma linha, em bytes.

As máscaras são inicializadas no processo *INIT*. A seleção de qual máscara será utilizada na escrita corrente é feita pelo bloco combinacional *SEL_MASK*, que toma tal decisão com base nos registradores *ncols* e *ncols_reload*. O bloco *PIXMASK_TO_BITMASK* então converte a máscara que está expressa em termos de pixels para bits.

Finalmente, o bloco combinacional *UPDATE* tem a finalidade de executar os incrementos de coluna e linha, atualizando os valores dos registradores *ncols*, *nlines*, e *addr*. É nesse bloco que são sinalizados ao controle do hardware quando que termina a escrita de uma linha e quando termina a operação: no registrador *status*, o bit EOL (End Of Line) sinaliza o fim de linha; e o bit EOOP (End Of Operation) sinaliza que a operação terminou.

Dada a visão geral da arquitetura, a próxima seção irá mostrar uma explicação mais detalhada de seu funcionamento, partindo de sua máquina de estados.

5.4.2 Máquina de estados

A explicação do hardware mostrado na Figura 13 irá partir da máquina de estados implementada para o controle, observada na Figura 14. A máquina segue a mesma ló-

gica de comunicação com a MDDR mostrada na Figura 11 (vide página 11). A título de simplificação, são mostrados apenas os sinais relevantes para as transições entre estados. Os demais sinais de controle envolvidos em cada estado são mostrados abaixo com a descrição detalhada do que um estado realiza. A principal fonte de estudo para a realização desta etapa foi (PEDRONI, 2010).

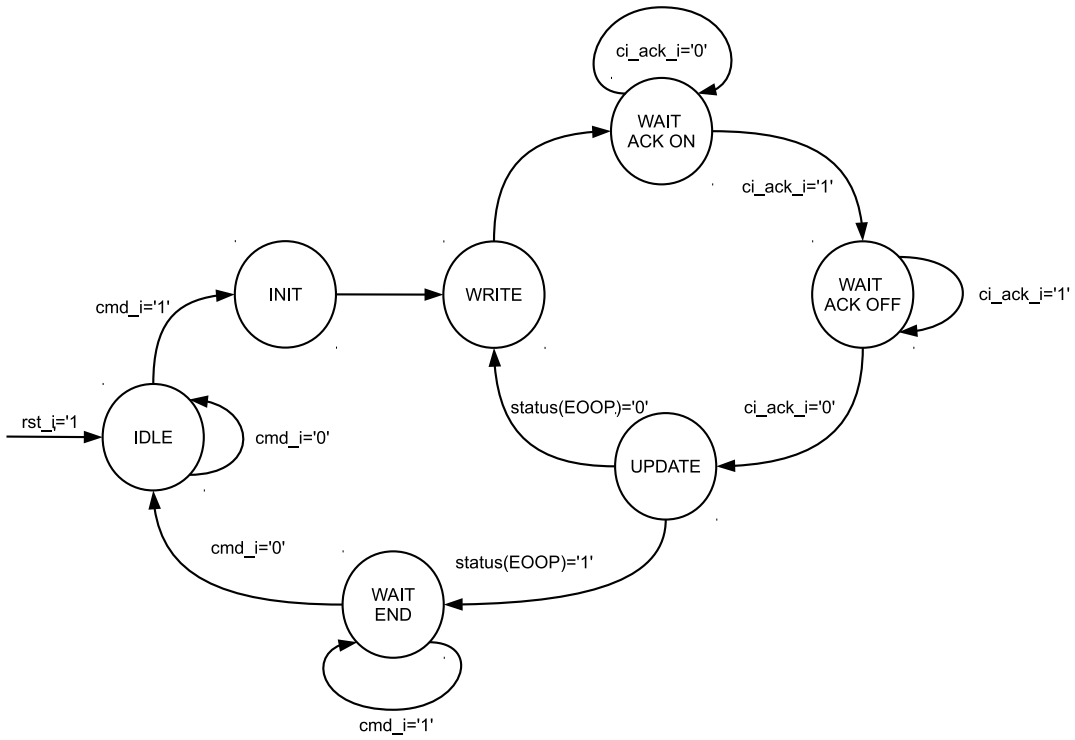

Figura 14: Máquina de estados para controle do hardware otimizador

O estado inicial é o estado *IDLE*, que tem a finalidade de esperar o comando para iniciar o desenho de retângulos. O papel dos demais estados seguem abaixo:

1. IDLE: O estado inicial apenas fica esperando pelo sinal de comando "cmd_i=’1’"para passar ao estado "INIT".
2. INIT: Este estado ativa a inicialização dos parâmetros do hardware.
   Como dito na Seção 5.4.1, são usados três registradores para controle de iterações: *ncols*, com o número de colunas de 16 pixels que serão escritas; *ncols_reload*, com o valor de recarga do registrador *ncols* quando muda-se de linha; e *nlines*, com o número de linhas. Esses registradores são inicializados segundo as Equações 2 a 4.

$$ncols = (width\_i/16) + 1 \quad (2)$$

$$ncols\_reload = ncols \quad (3)$$

$$nlines = (height\_i) \quad (4)$$

   O vetor de 64 bytes de dados é inicializado nesta etapa, replicando o pixel de entrada *pixel_i* uniformemente ao longo do registrador *data*. Isso é feito dessa maneira pois os dados não mudam e aquilo que será de fato escrito é dado pela máscara de escrita.

O endereço de escrita é guardado no registrador *addr*. Seu valor inicial é definido pelas entradas *x_i* e *y_i* e por constantes que representam a configuração da OSD. O cálculo é mostrado na Equação 5.

$$addr = BASE\_ADDR + (x\_i * WIDTH + y\_i) * PIXLEN \quad (5)$$

Os valores das constantes utilizadas acima são:

- BASE_ADDR: Endereço base do canal OSD. Seu valor neste trabalho é *0x43C00000*
- WIDTH: Largura, em pixels, do canal OSD. No trabalho, seu valor é igual a 720.
- PIXLEN: Largura, em bytes, de um pixel na OSD. O valor utilizado foi 4.

Com o endereço inicial de escrita calculado, pode-se calcular as máscaras que serão utilizadas na escrita. Para isso, observe a Figura 15, que mostra e explica características importantes do registrador *addr*. Os bits (31..6) indexam um bloco de 64 bytes escritos pela MDDR. Os bits (5..2) indexam um *pixel* dentro de um bloco de 64 bytes. Os bits (1..0) indexam um *canal de cor* (A, R, G, ou B) dentro de um pixel indexado anteriormente

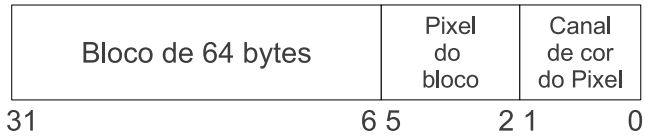

Figura 15: Registrador *addr* detalhado.

Embora os 6 últimos bits de *addr* não sejam utilizados, eles carregam uma informação importante: o primeiro pixel a ser escrito na primeira coluna de 16 pixels, indexado pelos bits (5..2). Chamando esse pixel de *pixel_ini*, a partir dele, utiliza-se o seguinte algoritmo para a definição das máscaras:

(a) Se $pixel\_ini + width\_i <= 16$:
Nesse caso, é feita apenas uma escrita, em apenas um bloco de 16 pixels. Portanto, é necessário apenas um registrador de máscara, que é o *mask_ini*. O índice do primeiro pixel a ser escrito corresponde ao valor *pixel_ini* visto anteriormente, e o índice do último pixel é dado por $pixel\_ini + width\_i$. Todos os pixels com índices entre esses dois valores serão escritos e, portanto, marcados com '1', enquanto que os demais são marcados com '0'. A Figura 16 ilustra esse caso.

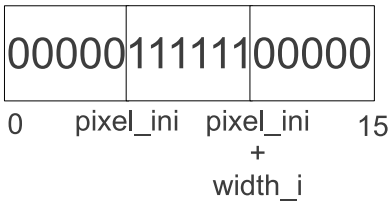

Figura 16: Registrador *mask_ini* para o caso em que é realizada apenas uma escrita por linha

(b) Se $pixel\_ini + width\_i > 16$:
Isso significa que serão feitas duas ou mais escritas por linha. O cálculo do primeiro pixel escrito na primeira colunas de 16 pixels continua igual ao anterior. Porém, o índice do último pixel a ser escrito agora é constante e igual a 15. A máscara utilizada neste caso continua sendo *mask_ini*.

Deve-se também utilizar sempre uma máscara para a última escrita de cada linha, *mask_end*. O índice do primeiro pixel a ser escrito é constante e igual a 0. O índice do último pixel, que aqui se chamará *pixel_end*, é calculado utilizando a *npix_1st*, que representa a quantidade de pixels escritos na primeira escrita por linha. O cálculo de *npix_1st* e *pixel_end* é mostrado nas Equações 6 a 7.

$$npix\_1st = 16 - pixel\_ini \tag{6}$$
$$pixel\_end = (width\_i - npix\_1st)\%16 \tag{7}$$

Para quaisquer escritas intermediárias entre a primeira e a última de cada linha, todas terão a mesma máscara constante, *mask_mid*. Todos os pixels são configurados para escrita nessa máscara. Por fim, a Figura 17 ilustra o formato das máscaras para o caso em que mais de uma escrita é realizada por linha.

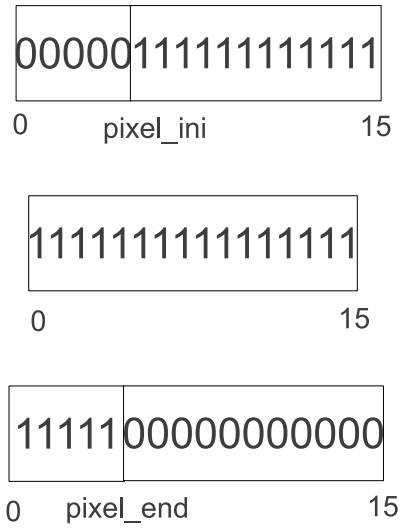

Figura 17: Formato das máscaras para o caso em que mais de uma escrita é realizada por linha. 17(a) Máscara inicial *mask_ini* 17(b) Máscara intermediária *mask_mid* 17(c) Máscara final *mask_end*

3. WRITE
Este estado inicializa o protocolo de escrita na memória. Primeiro, ele aciona o sinais de requisição e escrita *ci_req_o='1'* e *ci_wr_o='1'*. O circuito combinacional *SEL_MASK* seleciona, a partir dos valores dos registradores *ncols* e *ncols_reload*, a máscara pertinente para a escrita, utilizando a lógica da Listagem 5.1.

```
1 IF ncols = ncols_reload THEN
2    mask <= mask_ini;
```

```
3 ELSIF ncols = 1 THEN
4     mask <= mask_end;
5 ELSE
6    mask <= mask_mid
7 END IF;
```

Listagem 5.1: Lógica de seleção da máscara de escrita

Selecionada a máscara, o próximo passo é convertê-la do formato em pixel para o formato em bits. Isso é feito no bloco combinacional *PIXMASK_TO_BITMASK*. A lógica empregada constitui em replicar o valor do pixel quatro vezes, cada uma correspondendo a um canal de cor ARGB32.

4. WAIT_ACK_ON:
   Este estado espera o sinal *ci_ack_i* ter nível lógico alto, sinalizando que a MDDR recebeu a requisição. Enquanto esse sinal possui nível lógico baixo, o controle continua neste estado.
   Ao detectar nível lógico alto do sinal *ci_ack_i*, passa-se para estado WAIT_ACK_OFF, aproveitando para decrementar o contador de colunas *ncols* no processo. Se o novo valor do contador for zero, ativa-se a flag de status *EOL* (End Of Line).
5. WAIT_ACK_OFF:
   Esse passo da máquina é análogo ao anterior, com a diferença de que o controlador agora espera o nível lógico do sinal *ci_ack_i* tornar-se baixo.

   Quando for detectada a condição acima, a máquina testa a flag de status EOL. Caso ela esteja ativada, indicando final de escrita na linha, o registrador *nlines* é decrementado. Se o novo valor for nulo, ativa-se a flag no registrador de status "EOOP"(End Of Operation), indicando que o processo de escrita terminou.
6. UPDATE:
   Este estado é responsável por verificar o status da operação, atualizando o registrador de endereço *addr* conforme necessário.

   Se a flag EOOP estiver em nível lógico alto, a operação acabou e passa-se ao estado WAIT_END. Caso contrário, testa-se o nível lógico da flag EOL. Se for alto, isso significa que se terminou de escrever uma linha e, portanto, o novo endereço de escrita é incrementado de forma a passar para a próxima. Se EOL estiver com nível lógico baixo, incrementa-se o endereço de forma a passar para a próxima coluna de 16 pixels.
7. WAIT_END:
   Neste estado, o sinal *status_o* retorna nível lógico baixo, indicando que a operação terminou. Com isso feito, espera-se o sinal *cmd_i* possuir nível lógico baixo, indicando que o software recebeu a confirmação. Logo após, retorna-se ao estado IDLE, esperando mais escritas.

Com o hardware implementado, o próximo passo é substituir a implementação original em software da função *FillRectangle*. Para tanto, é necessário implementar um driver de dispositivo para o sistema operacional. Uma vez que foi utilizado o dispositivo da Gaisler GRGPIO para adicionar o novo hardware ao barramento AMBA, o driver criado foi para este último dispositivo, assunto a ser abordado na próxima seção.

5.5 Driver de dispositivo para a GPIO

Como foi apresentado na Seção 5.2, a maneira escolhida para comunicar o hardware criado com o sistema operacional foi utilizar o módulo *GRGPIO*, visto que isto simplifica a adição do módulo no barramento AMBA. Isso trouxe a necessidade de ser desenvolvido um driver para esse dispositivo dentro do ambiente Linuxbuild, já que não existia uma implementação disponível na época em que este trabalho foi desenvolvido. Esta seção irá apresentar o desenvolvimento desse driver.

Para implementar o driver, é necessário entender como o dispositivo funciona. A interface da GRGPIO possui 8 registradores de até 32 bits, com dois deles sendo os registradores principais, que são:

- *data*, o registrador com os dados de entrada;
- *output*, onde são escritos os valores de saída;

.

Esses registradores são mapeados em memória e suas posições-base dependem da configuração das portas no VHDL do hardware mas, uma vez configuradas, geram um mapeamento com deslocamento constante em relação ao endereço base. Esse mapeamento pode ser visto na Tabela 4.

Tabela 4: Deslocamento dos registradores da GRGPIO em relação ao endereço base

Deslocamento (em hexa)	Registrador
0x00	data
0x04	output

As características do driver da GRGPIO implementado nesta etapa são:

- Dispositivo orientado a caractere (CORBET; RUBINI; KROAH-HARTMAN, 2005)
- Leituras e escritas de 32 bits, independentemente da quantidade configurada no hardware

5.5.1 Estruturas de dados

As principais estruturas de dados utilizadas pela GRGPIO podem ser vistas na Listagem 5.2. As Linhas 4 a 7 declaram uma estrutura cujo propósito é guardar informações gerais, tanto as obtidas em tempo de inicialização quanto as estáticas. No caso, a estrutura apenas guarda a quantidade de módulos de GRGPIO detectados.

```
1 /*
2 General GRGPIO data
3 */
4 struct grgpio_dev_data {
5   int num_dev;
6   int num_bits;
7 } grgpio_data;
8
9 /*
10 GRGPIO Registers
11 */
12 struct grgpio_regs {
13   u32 data;
14   u32 output;
15   u32 direction;
16   u32 int_mask;
17   u32 int_polarity;
18   u32 int_edge;
19   u32 int_bypass;
20 };
21
22 /*
23 GRGPIO Devices Vector
24 */
25 struct grgpio_dev {
26   /* grgpio registers */
27   struct grgpio_regs *regs;
28
29   /* grgpio name*/
30   char name[10];
31
32   /* char device structure */
33   struct cdev cdev;
34 } *grgpio_devp[MAX_NUM_GPIO];
```

Listagem 5.2: Principais estrutura de dados do driver da GRGPIO

Das Linhas 12 a 20, é declarada a estrutura que representa os registradores de um dispositivo GRGPIO. Além dos registradores mostrados na Tabela 4, são mostrados os demais registradores presentes no dispositivo, não utilizados neste trabalho.

Das Linhas 25 a 34, é declarada a estrutura principal que representa um dispositivo GRGPIO. Ela, além de conter os registradores mostrados na Tabela 4, consiste em:

- Um ponteiro para os registradores mapeados em memória.
- O nome do dispositivo, utilizado para identificá-lo no sistema.
- Uma estrutura cdev, indicando o tipo de driver como sendo de caracteres.

Ao mesmo tempo, na Linha 34 aloca-se um vetor de "MAX_NUM_GPIO"ponteiros para esta estrutura do driver, que são alocadas conforme sua identificação durante a etapa de inicialização, que será vista a seguir.

5.5.2 Inicialização

A inicialização do dispositivo utiliza a mesma sistemática observada nos dispositivos com drivers que a Gaisler disponibiliza. Utilizar essa inicialização apresenta a vantagem de ser escalável, ou seja, se quisermos utilizar mais de uma GRGPIO não é necessário código adicional pois elas serão automaticamente identificadas.

Para utilizá-la, deve-se tomar os seguintes passos:

1. Declarar uma lista de identificadores do hardware em questão.
   A lista é vista na Listagem 5.3. São utilizadas duas declarações: uma declaração em formato do código hexadecimal que identifica o dispositivo dentro da GRLIB e outra em formato de string.

```
1  static struct of_device_id grgpio_of_match[] = {
2    {
3     .name = "GAISLER_GPIO",
4     },
5    {
6     .name = "01_01a",
7     },
8    {},
9  };
10
11 MODULE_DEVICE_TABLE(of, grgpio_of_match);
```

Listagem 5.3: Identificadores de hardware para a grgpio

2. Definir o driver da plataforma.
   A definição é mostrada na Listagem 5.4. Ela consiste em definir dentro da estrutura do driver identificadores como:
   - Nome do driver, na Linha 3.
   - Escopo de atuação, na Linha 4.
   - Identificadores de hardware, Linha 5.
   - Definição da função *probe*, que tem a função de realizar a inicialização de fato do hardware de acordo com o que for encontrado pela busca PnP.

```
1 static struct of_platform_driver grgpio_of_driver = {
2   .driver = {
3     .name = GRGPIO_NAME,
4     .owner = THIS_MODULE,
5     .of_match_table = grgpio_of_match,
6   },
7   .probe = grgpio_of_probe,
8   .remove = __devexit_p(grgpio_of_remove),
9 };
```

Listagem 5.4: Definição do driver da GRGPIO na plataforma

3. Registrar o driver dentro da plataforma.
   Isso é feito durante a inicialização, mostrado na Listagem 5.5, e irá disparar o processo de busca pelo hardware da GRGPIO no barramento AMBA. Além disso, é feita a inicialização das estruturas de dados acima: os ponteiros para os dispositivos GRGPIO são zerados e a quantidade de dispositivos é inicializada para zero.

```
1  static int __init soch264_gpio_init(void)
2  {
3    int i;
4    /* Initialize GRGPIO device pointers */
5    for (i=0;i<MAX_NUM_GPIO; grgpio_devp[i++]=NULL);
6
7    /*  Set number of grgpios implemented to 0 */
8    grgpio_data.num_dev=0;
9
10   /* Start searching... */
11   return of_register_platform_driver(&grgpio_of_driver);
12 }
```

Listagem 5.5: Inicialização do driver

Com esse esquema, toda vez que for identificado um hardware com o código da GRGPIO é chamado o método "probe". Ele é mostrado na Listagem 5.6 e recebe como parâmetros o início e o fim da região de memória correspondente aos seus registradores mapeados.

Com esses dados, esse método então toma os seguintes passos:

1. Registra o dispositivo com MAJOR igual a 212, se ele não estiver registrado ainda. Isso é feito das Linhas 12 a 17.
   O número MAJOR identifica unicamente um dispositivo dentro do Linux. Para checar isso, o método *probe* verifica a quantidade de GRGPIO's já existentes e configuradas: se for igual a 0, significa que é a primeira GRGPIO sendo configurada, logo deve-se registrar o MAJOR.
2. Configuração do índice do dispositivo dentro do vetor de ponteiros, que consiste no atual número de dispositivos encontrados e configurados, feito na Linha 22.
3. Alocação dinâmica de uma estrutura "grgpio_dev", feito das Linhas 27 a 32, utilizando o slot dentro do vetor pego no item anterior
4. Configuração do nome do dispositivo de acordo também com o slot que ele ocupa, feito na Linha 35.
5. Aquisição e remapeamento da região de memória retornada pelo registro do driver, feito nas Linhas 40 e 41.
   Quando é encontrado um hardware com a descrição da GRGPIO, é retornado junto no parâmetro "ofdev"os recursos associados a ela, que no caso é a área de memória onde estão os registradores mapeados em memória.
6. Por fim, se o dispositivo foi corretamente inicializado, na Linha 50 incrementa o contador de dispositivos, a fim de também indicar qual será a posição no vetor de dispositivos que um possível próximo GRGPIO ocupará.

Sendo um driver orientado a caractere, as operações disponibilizadas são as padrões de acesso a um arquivo genérico, que são: abrir, fechar, ler e escrever. As próximas seções irão apresentar as implementação desses métodos. Uma vez que a implementação de fechar o arquivo não realiza nada no driver até aqui implementado, ela será omitida.

```
1  static int __devinit
2  grgpio_of_probe(struct platform_device *ofdev, const struct
       of_device_id *match)
3  {
4    int err;
5    int i,dev_index;
6    dev_t dev;
7    struct grgpio_regs *regs;
8
9    /* Registering device if hasn't already */
10   if (!grgpio_data.num_dev){
11     if ( (err=register_chrdev(GRGPIO_MAJOR,GRGPIO_NAME,&gpio_fops)) < 0
       ){
12       printk(KERN_ERR "grgpio: failure registering devices");
13       return -1;
14     }
15   }
16
17   /* setting device index */
18   dev_index=grgpio_data.num_dev;
19
20   /* allocating device memory */
21   grgpio_devp[dev_index]=kmalloc(sizeof(struct grgpio_dev),GFP_KERNEL);
22
23   if (!grgpio_devp[dev_index]){
24     printk(KERN_ERR "grgpio: failure allocating device memory");
25     return -ENOMEM;
26   }
27
28   /* setting grgpio device name */
29   sprintf(grgpio_devp[dev_index]->name,"gpio%d",dev_index);
30
31   /* getting register mapping */
32   grgpio_devp[dev_index]->regs=(struct grgpio_regs*)of_ioremap(&ofdev->
       resource[0], 0, resource_size(&ofdev->resource[0]), strcat(
       grgpio_devp[dev_index]->name," regs"));
33
34   if (grgpio_devp[dev_index]->regs==NULL){
35     printk(KERN_ERR "%s: ioremap failure",grgpio_devp[dev_index]->name)
       ;
36     err = -EIO;
37     goto err1;
38   }
39
40   //incrementing number of grgpios devices found
41   grgpio_data.num_dev++;
42
43   return 0;
44
45 err2:
46   of_iounmap(&ofdev->resource[dev_index],regs,resource_size(&ofdev->
       resource[dev_index]));
47
48 err1:
49   kfree(grgpio_devp[dev_index]);
50   return err;
51 }
```

Listagem 5.6: Método probe para inserção de um driver da GRGPIO

5.5.3 Abertura do dispositivo

Uma vez que um driver Linux funciona como se fosse um arquivo, antes de começar sua utilização deve-se, primeiro, abri-lo. Porém, uma vez que é prevista a utilização de mais de um dispositivo GRGPIO, durante essa abertura deve-se implementar um mecanismo para que seja utilizado o dispositivo correto dentre os vários que podem coexistir.

É nesse ponto que entra a utilização do número MINOR do dispositivo. Enquanto o MAJOR tem o papel de identificar o dispositivo, o número MINOR é utilizado internamente pelo dispositivo da maneira que se desejar. No caso da GRGPIO, ele foi utilizado para obter o dispositivo dentro do vetor GRGPIO a ser utilizado.

A Listagem 5.7 mostra como é realizada essa implementação. O "inode"é uma estrutura que representa um arquivo no mundo UNIX. Com ele, obtém-se qual é o MINOR associado ao arquivo recém aberto na Linha 5. Em posse desse MINOR, configura-se o parâmetro *private_data* do descritor do arquivo recém-aberto *file* com o ponteiro apropriado do vetor.

```
1  int gpio_open(struct inode *inode,struct file *file){
2    struct grgpio_dev *this_grgpio_devp;
3
4    /* use the minor number to get the correct pointer */
5    if (MINOR(inode->i_rdev)>=grgpio_data.num_dev)
6      return -ENODEV;
7    this_grgpio_devp=grgpio_devp[MINOR(inode->i_rdev)];
8
9    /* set private_data to the correct grgpio_devp pointer */
10   file->private_data = this_grgpio_devp;
11
12   /* device data initialization */
13   this_grgpio_devp->datap=0;
14
15   return 0;
16 }
```

Listagem 5.7: Abertura do dispositivo GRGPIO para utilização

5.5.4 Leitura

A operação de leitura consiste em simplesmente ler o valor atual do registrador *output*. A implementação é mostrada na Listagem 5.8. Os principais pontos são:

1. Obter o ponteiro correto para o dispositivo GRGPIO desejado, feito na Linha 7
2. Verificar se o *buffer* passado como parâmetro possui pelo menos 4 bytes, o tamanho máximo do registrador, retornando um erro caso contrário. Isso é feito na Linha 10.
3. Ler o dado desejado, utilizando o método *iowrite32*, que lê 4 bytes do dispositivo de maneira portável. Isso é feito na Linha 13.
4. Corrigir o *endianess*, feito na Linha 15.
   Sendo um processador com arquitetura SPARC, o Leon3 utiliza o *big-endian*; porém, o dispositivo GRGPIO utiliza o *little-endian*. No momento de escrita desta dissertação, esse problema foi resolvido via software pelo driver. Porém, isso pode ser resolvido no projeto do hardware.
5. Retornar exatamente o tamanho do buffer passado como parâmetro. Dentro da infraestrutura de driver de dispositivo do Linux, esta é a melhor situação e retorna imediatamente ao processo que iniciou a leitura.

```
1 static int gpio_read(struct file* file,char *buf,size_t count,loff_t *
    f_pos)
2 {
3   int b;
4   u32 data;
5   struct grgpio_dev *grgpio_devp;
6
7   grgpio_devp=(struct grgpio_dev*)file->private_data;
8
9   // buffer must have at least 4 bytes
10  if (count<4)
11    return -ENOMEM;
12
13  data=ioread32((void*)&grgpio_devp->regs->data);
14
15  for (b=0;b<4;b++)
16    buf[b]=((char*)&data)[3-b];
17
18  return count;
19 }
```

Listagem 5.8: Leitura na GRGPIO

5.5.5 Escrita

O método de escrita é semelhante ao de leitura. Sua implementação pode ser vista na Listagem 5.9. Primeiro, ele inverte o *endianess* nas Linha 12 e 13, para então escrever os dados, na Linha 15.

```
1 static ssize_t gpio_write(struct file *file,const char *buf,size_t
    count,loff_t *ppos)
2 {
3   u32 output;
4   struct grgpio_dev *grgpio_devp;
5
6   grgpio_devp=(struct grgpio_dev*)file->private_data;
7
8   // at least 4 bytes must be written
9   if (count<4)
10    return -ENOMEM;
11
12  for (b=0,output=0;b<4;b++)
13    ((char*)(&output))[3-b]=buf[b];
14
15  iowrite32(output,(void*)&grgpio_devp->regs->output);
16
17  return count;
18 }
```

Listagem 5.9: Escrita na GRGPIO

5.6 Substituição da implementação em software

Com o hardware criado, o próximo passo é substituir a implementação em software original. Esse trabalho começa mapeando os sinais de controle e entradas do módulo acelerador nos dispositivos de GRGPIO. O dispositivo GRGPIO possui 8 registradores de 32 bits, sendo que dois deles são importantes para este trabalho: *data*, o registrador onde são recebidos os dados de entrada, e *output*, onde são colocados os dados de saída. Dito isso, foram instanciados três dispositivos GRGPIO, cada um dos quais tendo os seguintes mapeamentos do hardware criado:

- **GRGPIO0**: mapeia os sinais de saída *cmd_i*, *x_i* e *y_i* no registrador *output* e o sinal de controle *status_o* no registrador *data*, conforme o mostrado nas Tabelas 5 e 6.

Tabela 5: Mapeamento dos sinais do hardware *cmd_i*, *x_i* e *y_i* no registrador *output* da GRGPIO0

Bits		Sinal
Final	Inicial	
31	31	cmd_i
19	10	x_i
9	0	y_i

Tabela 6: Mapeamento dos sinais do hardware no registrador *data* da GRGPIO0

Bits		Sinal
Final	Inicial	
30	30	output_o

- **GRGPIO1**: mapeia os sinais de saída *width_i* e *height_i* no registrador *output* conforme o mostrado na Tabela 7.

Tabela 7: Mapeamento dos sinais do hardware *width_i* e *height_i* no registrador *output* da GRGPIO1

Bits		Sinal
Final	Inicial	
19	10	width_i
9	0	height_i

- **GRGPIO2**: mapeia a cor do pixel *pixel_i*, segundo a Tabela 8.

Tabela 8: Mapeamento do sinal *pixel_i* do hardware no registrador *output* da GRGPIO2

Bits		Sinal
Final	Inicial	
31	24	canal Alfa
23	16	canal R
15	8	canal G
7	0	canal B

Com esse mapeamento completo, para trocar a implementação padrão da FillRectangle é preciso criar um driver do DirectFB para o módulo. Esse driver deve configurar essa função para ser acelerada e implementar o algoritmo para substituir a implementação em software. O algoritmo utilizado segue na Listagem 5.10.

```
1  //Inicializa o pixel - feito pela GRGPIO2
2  gpio2_output=(long)( pixel_i );
3
4  //Inicializa width_i e height_i - feito pela GRGPIO1
5  gpio1_output=(long)( (0x000FFC00&(width_i<10))|(0x000003FF & height_i
     ) );
6
7  //Inicializa x_i, y_i, e manda escrever - feito pela GRGPIO0
8  gpio0_output=(long)( (0x000FFC00&(y_i<<10))|(0x000003FF & x_i)|0
     x80000000 );
9
10 //Escreve os dados e inicializa a operação
11 write(gpio2,&gpio2_output,4);
12 write(gpio1,&gpio1_output,4);
13 write(gpio0,&gpio0_output,4);
14
15 //Espera escrita terminar, o que ocorre quando o bit 30 do
     registrador data da GPIO0 eh igual a '1'
16 do {
17   read(gpio0,&gpio0_data,4);
18 }
19 while ( (gpio0_data&0x40000000) == 0 );
20
21 //Abaixa o sinal cmd_i, indicando que recebeu o comando
22 write(gpio0,&clear,4);
23
24 //Espera o bit 30 ser igual a '0'
25 do {
26   read(gpio0,&gpio0_data,4);
27 }
28 while ((gpio1_i&0x40000000)!=0);
```

Listagem 5.10: Algoritmo da implementação em hardware do FillRectangle

Finalmente, o que resta agora é executar um programa que utilize a função acelerada. Por exemplo, o teste de conformidade *area11.01.01.ncl*, mostrado na Figura 18. A comparação com a implementação em software é tema da próxima seção.

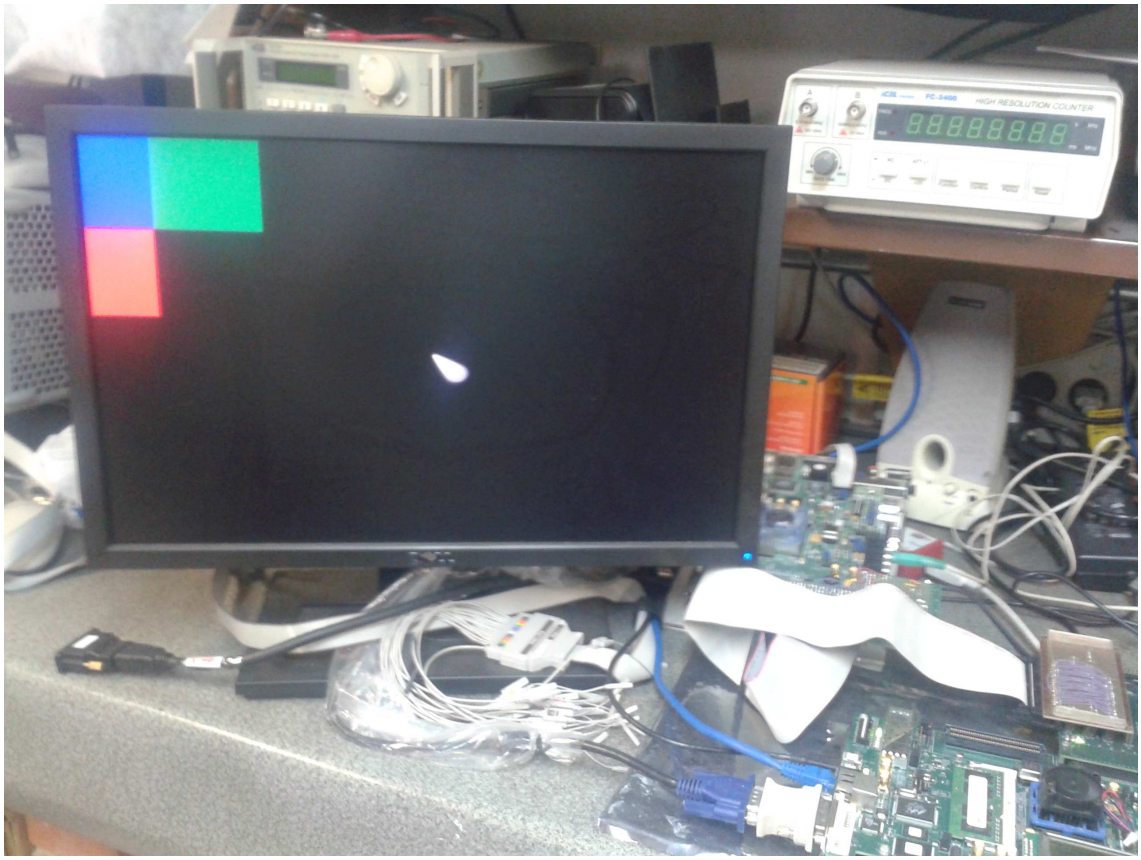

Figura 18: Execução do teste de conformidade *area11.01.01.ncl*, que utiliza a função *FillRectangle*

6 RESULTADOS

Este capítulo tem a finalidade de apresentar e analisar os resultados obtidos neste trabalho. O capítulo inicia com uma análise da execução do middleware dentro do STB, utilizando para tanto programas NCL obtidos da suíte de testes de conformidade do middleware Ginga-NCL. Em seguida, são informados os resultados relativos a adaptação do subsistema gráfico. Por fim, é analisado o ganho da implementação em hardware feita para a função FillRectangle em relação a original em software.

6.1 Programas NCL

Os programas NCL executados são um subconjunto dos testes de conformidade do middleware (ARAÚJO, 2011; TELEMIDIA LABS, 2014c). Esta suíte é composta por 400 testes, separados em *testes manuais* e *testes semiautomáticos*. Os testes manuais devem ser executados, seus resultados observados e comparados com o que se espera, e então terminados também manualmente. Já os testes semi-automáticos permitem utilizar o próprio middleware para implementar uma rotina automatizada de testes, devendo ser executados somente após a execução dos testes manuais. Todos esses testes possuem uma descrição do seu comportamento em seu início, de forma que, para saber se o teste passou, deve-se comparar o que aconteceu durante sua execução com essa descrição. Como exemplo, a Listagem 6.1, mostra o início do código do teste de conformidade *area01.01.01.ncl*, onde pode ser observado a descrição de seu comportamento.

```
1 <?xml version="1.0" encoding="ISO-8859-1" standalone="no"?><ncl xmlns="
    http://www.ncl.org.br/NCL3.0/EDTVProfile" id="area01.01.01">
2   <!-- Displays a blue jpg image during 3 seconds and then displays a
    red jpg image-->
3   <head>
```

Listagem 6.1: Descrição do teste de conformidade Ginga-NCL *area01.01.01.ncl*

A suíte de testes foi utilizada para calcular o tempo médio de inicialização dos programas NCL. Esse tempo foi calculado como o tempo transcorrido desde o momento em que se dispara o comando de execução do programa até o momento em que ele começa a realizar a tarefa de sua descrição. Esses tempos são obtidos por meio de um arquivo de *log* gerado pelo middleware: dentro desse log, é listado o instante de tempo exato em que se passa o comando para o escalonador de eventos.

Para simplificar a análise, dos 400 testes existentes na suíte, foram selecionados para execução 171, sendo excluídos os seguintes tipos de testes:

- Testes de mídia com áudio e vídeo, uma vez que a capacidade de executá-los não está presente na versão utilizada.

- Testes manuais, devido ao fato de eles requererem acompanhamento manual, dificultando a criação de um método automático para execução dos testes.

O script criado para obter as duas medições encontra-se no Apêndice B, assim como quais testes foram executados com sucesso. Nele, caso um teste falhe, ele é executado novamente até 10 vezes, sendo a cada vez seus *timestamps*. O tempo médio de inicialização dos programas obtidos é mostrado na Tabela 9, comparando com os resultados obtidos em (LIM; LEE, 2011). Comparando os dois valores, nota-se que o sistema aqui desenvolvido possui um tempo de inicialização maior. Porém, algumas considerações devem ser feitas:

Tabela 9: Tempo médio de inicialização de um programa NCL

	Tempo médio
(LIM; LEE, 2011)	6,8s
Obtido no trabalho	15,6s

- O trabalho citado não diz se o tempo de inicialização da aplicação leva em consideração a inicialização do sistema Ginga-NCL, ou seja, se o Ginga-NCL encontrava-se em execução. Essa diferenciação é importante pois, no melhor caso, em que o sistema já está em execução, o tempo mostrado no trabalho aumentaria.
- Embora não diga explicitamente, o trabalho citado sugere, por meio de fotos da aplicação sendo executada em um *smarthphone*, que os resultados foram extraídos de uma plataforma final, contrastando com os deste trabalho, obtidos em FPGAs.
- Semelhante ao item anterior, não há menção do hardware utilizado no trabalho citado.
- O trabalho citado calcula dois tempos: um para o caso em que o programa NCL já tenha sido recebido, chamando-o de *with cache*, e outro quando o programa não existe, rotulando-o como *no cache*. Como a questão do demultiplexador de TS não foi abordada neste trabalho, é apresentado somente o primeiro tempo, considerando-o mais próximo do que foi desenvolvido.

6.2 Subsistema gráfico

O DirectFB possui dois programas de benchmark: *df_dok* e *df_andi*. O primeiro é um programa com uma coleção de testes padronizados para cada uma das funções do DirectFB. Por padrão, o programa executa todos os testes, mas permite a execução individual. Já o segundo programa realiza uma avaliação do *framerate* que a parte gráfica pode atingir. Esta seção apresenta os resultados obtidos por ambos a fim de servir como referência para comparações futuras. Além disso, são feitos comentários acerca das características das modificações no módulo de pós-processamento gráfico, mostrada anteriormente na Seção 4.3.

A Tabela 10 mostra todos os resultados obtidos pelo *df_dok*. A Figura 19 mostra a execução do *df_andi*, exibindo o seu resultando de 1.4 *frames per second* no canto superior esquerdo da tela.

A tela inicial do *df_dok*, executando na máquina virtual de (TELEMIDIA LABS, 2014a), é mostrada na Figura 20. Observa-se que ela possui fundo preto. Ao executar o mesmo programa no STB, dentro do formato de cor ARGB utilizado pelo framebuffer, isso resulta em um valor "Alpha=0". Quando um pixel com esse valor de Alpha é lido pelo módulo de pós-processamento, este entende que deve exibir o pixel oriundo do decodificador de vídeo ao invés do pixel preto. Esse resultado é mostrado na Figura 21, em que aparece um vídeo de teste de um jogo de futebol ao fundo.

Em contrapartida, o logo DirectFB, sendo escrito totalmente em branco, possui valor do canal Alfa igual a 255. Para o módulo de pós-processamento, isso significa que deve ser exibido totalmente o pixel lido da região de memória OSD. Ou seja, a saída do decodificador de vídeo é sobreposta pelo logo.

Tabela 10: Resultados do *df_dok*, benchmark de funções do DirectFB, no ambiente portado. "MPixel/s"significa MegaPixel/segundo

Função	**Desempenho**	**Unidade**
Anti-aliased Text	4.986	(KChars/sec)
Anti-aliased Text (blend)	0.888	(KChars/sec)
Fill Rectangle	3.651	(MPixel/sec)
Fill Rectangle (blend)	0.242	(MPixel/sec)
Fill Rectangles [10]	3.825	(MPixel/sec)
Fill Rectangles [10] (blend)	0.242	(MPixel/sec)
Fill Triangles	2.745	(MPixel/sec)
Fill Triangles (blend)	0.236	(MPixel/sec)
Draw Rectangle (KRects/sec)	0.303	(KRects/sec)
Draw Rectangle (blend) (KRects/sec)	0.067	(KRects/sec)
Draw Lines [10] (KLines/sec)	1.362	(KLines/sec)
Draw Lines [10] (blend) (KLines/sec)	0.311	(KLines/sec)
Fill Spans (MPixel/sec)	3.091	(MPixel/sec)
Fill Spans (blend) (MPixel/sec)	0.237	(MPixel/sec)
Blit (MPixel/sec)	2.229	(MPixel/sec)
Blit 180 (MPixel/sec)	1.466	(MPixel/sec)
Blit colorkeyed (MPixel/sec)	1.302	(MPixel/sec)
Blit destination colorkeyed (MPixel/sec)	1.527	(MPixel/sec)
Blit with format conversion (MPixel/sec)	0.548	(MPixel/sec)
Blit with colorizing (MPixel/sec)	0.330	(MPixel/sec)
Blit from 32bit (blend) (MPixel/sec)	0.137	(MPixel/sec)
Blit from 32bit (blend) with colorizing (MPixel/sec)	0.123	(MPixel/sec)
Stretch Blit (MPixel/sec)	1.951	(MPixel/sec)
Stretch Blit colorkeyed (MPixel/sec)	1.769	(MPixel/sec)

Figura 19: Execução do teste de *framerate* do *df_andi* no Set-Top Box

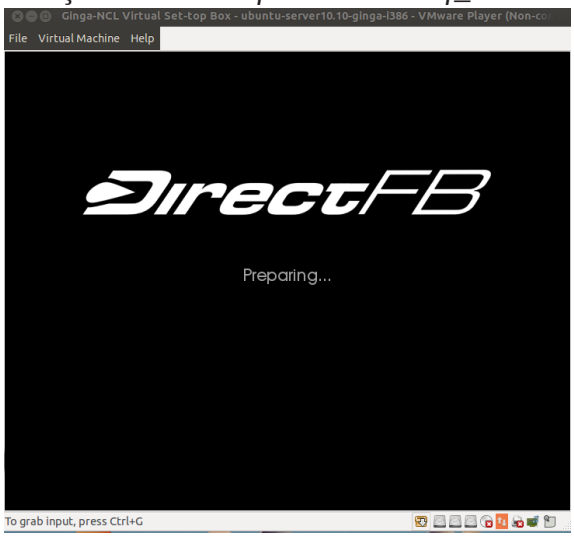

Figura 20: Tela inicial do *df_dok*, executando em Máquina Virtual Ginga-NCL (TELEMIDIA LABS, 2014a)

Figura 21: Tela inicial do *df_dok*, executando no Set-Top Box

6.3 Ganho do hardware

Para avaliar o ganho obtido com a implementação em hardware, foram feitos duas baterias de testes: em um primeiro momento sem a presença do sistema operacional; e depois utilizando com sistema operacional com benchmark do DirectFB *df_dok*.

Na avaliação sem sistema operacional, o algoritmo para utilização do hardware mostrado anteriormente na Seção 5.6, bem como o equivalente em software, foi feito com um programa em C utilizando o *Bare Cross-Compiler* da Gaisler (AEROFLEX GAISLER, 2013f). O programa desenvolvido pode ser encontrado no Apêndice C. A métrica de desempenho foi calculada da mesma maneira que o programa df_dok: desenha-se o maior número de retângulos possível dentro de um dado intervalo de tempo, obtendo o desempenho em MPixel/s pela razão entre a quantidade de pixels escritos pelo tempo utilizado.

Na implementação feita em software, utilizou-se a primitiva *clock()* do compilador BCC, a qual retorna o tempo passado em micro-segundos. O cálculo do desempenho das implementações é feito então segundo a Equação 8:

$$MPix/s = \frac{n_{rects} * width * height}{t_{final} - t_{inicial}} \tag{8}$$

Em que:

- n_{rects}: Número de retângulos escritos dentro do intervalo de tempo
- $width$: Largura do retângulo
- $height$: Altura do retângulo
- $t_{inicial}$: Tempo em que o teste iniciou
- t_{final}: Tempo em que o teste terminou

Já os testes feitos com o df_dok foram feitos por meio de um script, encontrado no Apêndice D. No caso do df_dok, é necessário apenas configurar os parâmetros pertinentes ao teste por linha de comando que o resultado é retornado já na métrica utilizada.

Tanto os testes com e sem sistema operacional foram executados 10 vezes, para diversas combinações de altura e largura de retângulos. As larguras utilizadas iniciam-se com 100 pixels, com incrementos de 100 pixels até o limite de 700. Já as alturas iniciam-se também com 100 pixels e incrementando de 100 em 100, porém terminando com 400 pixels. Tais limites foram definidos segundo as dimensões da região de memória OSD, com 720x480 pixels. Os resultados obtidos para os casos com e sem sistema operacional são mostrados nas Tabelas 11 e 12, nas páginas 63 e 64, respectivamente. A coluna "Ganho"nos dois casos foi calculada como a razão das médias dos resultados em hardware pela em software.

Para facilitar a análise dos resultados, dois gráficos de superfície foram criados a partir das tabelas, mostrados nas Figuras 22 e 23, os quais relacionam os ganhos com as dimensões utilizadas nos testes com e sem sistema operacional, respectivamente.

O primeiro fato observado é que o desempenho da implementação com sistema operacional é sempre menor do que a sem sistema operacional, o que é esperado dada a adição de *overheads* do processo de escalonamento e do acesso a infraestrutura de arquivos do driver implementado.

Os gráficos também apontam uma mesma tendência: quanto maiores as dimensões dos retângulos a serem escritos, o ganho tende a ser maior. Porém, enquanto o gráfico no caso sem sistema operacional apresenta um ganho mínimo, próximo de ser constante, o gráfico para o caso com sistema operacional apresenta ganhos maiores, inclusive ultrapassando os ganhos observados sem sistema operacional.

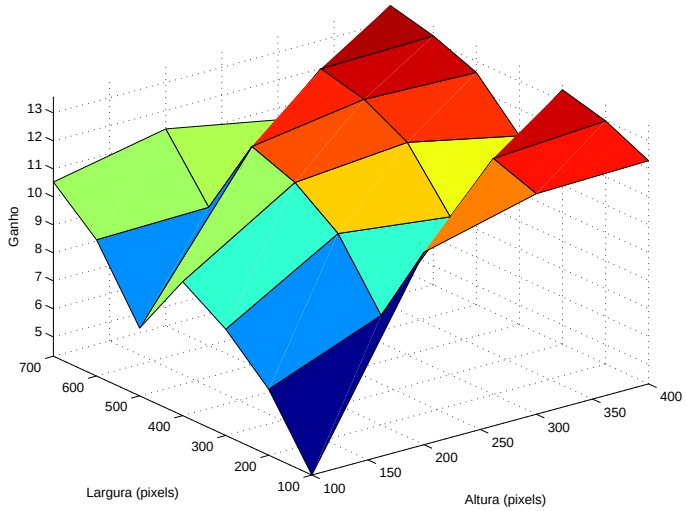

Figura 22: Superfície de ganho do hardware em relação ao software com sistema operacional, para a escrita de retângulos com diferentes dimensões

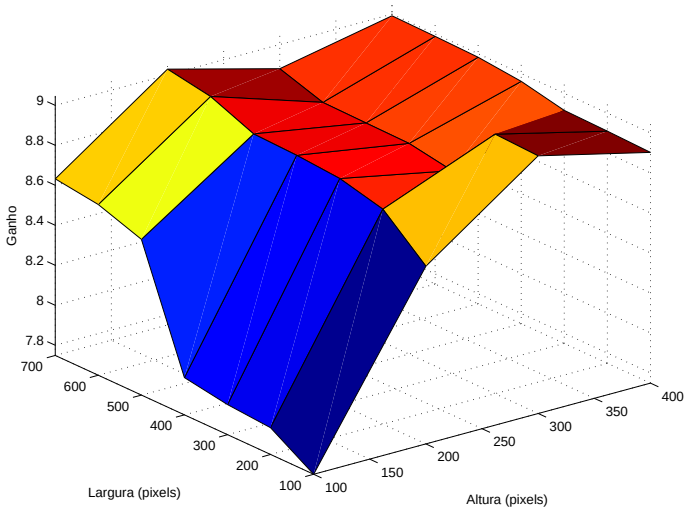

Figura 23: Superfície de ganho do hardware em relação ao software sem sistema operacional, para a escrita de retângulos com diferentes dimensões

Esse comportamento pode ser explicado também pelos *overheads* do sistema operacional. Nota-se que a implementação em software, nos dois casos, apresenta comportamento constante. Porém, a taxa de pixels escritos cai pela metade quando há sistema operacional. Ou seja, em ao menos metade do tempo em que antes se escrevia pixels agora são feitas outras tarefas, uma vez que o intervalo de tempo permanece constante nos dois casos. diminuindo pela metade a taxa de escrita.

Por outro lado, na implementação em hardware, quando este inicia o processo de escrita de um retângulo as únicas interações necessárias com o sistema operacional são as de controle, a fim de verificar se a operação terminou. Ou seja, a escrita de pixels é feita de forma independente, e durante o período que o software não escreve pixels, o hardware está trabalhando.

Isso traz mais uma questão: a alta variação observada no desempenho do hardware no caso com sistema operacional, contrastando quando não há. Uma hipótese para explicar esse fenômeno reside na implementação feita do algoritmo da Seção 5.6, a qual utiliza

espera ocupada, resultando em acessos à memória que competem com o hardware criado. Para validá-la, porém, seria necessário mudar a implementação da função em hardware nos dois casos a fim de eliminar a espera ocupada, um trabalho que fica para o futuro.

Tabela 11: Resultados da execução da função FillRectangle com sistema operacional, em MPixel/s

Largura	Altura	Software		Hardware		**Ganho**
		Média	Desvio Padrão	Média	Desvio Padrão	
100	100	3.25	0	17.3	0.106	4.31
100	200	3.37	0.0128	25.8	0.118	6.66
100	300	3.42	0.00871	31.2	0.158	8.11
100	400	3.44	0.009	34.8	0.224	9.1
200	100	3.48	0.0162	26.9	0.243	6.73
200	200	3.55	0.0157	36.1	0.326	9.18
200	300	3.6	0.011	41.5	0.122	10.5
200	400	3.63	0.0085	44.2	0.2	11.2
300	100	3.57	0.0084	32.9	0.16	8.22
300	200	3.64	0	41.6	0	10.4
300	300	3.67	0.006	46.1	0.156	11.6
300	400	3.69	0.006	48.4	0.174	12.1
400	100	3.6	4.44e-016	36.9	0.219	9.25
400	200	3.68	0.0068	45.5	0.211	11.3
400	300	3.7	0.0036	48.9	0.16	12.2
400	400	3.72	4.44e-016	51.1	0.163	12.7
500	100	3.65	0.0078	39.9	0.159	9.95
500	200	3.71	0.00686	47.8	0.229	11.9
500	300	3.73	0.0036	51.2	0.174	12.7
500	400	3.74	0.00343	52.8	0.228	13.1
600	100	3.67	0.0101	42.5	0.248	10.6
600	200	3.73	0.0036	49.5	0.162	12.3
600	300	3.74	0	52.4	0.151	13
600	400	3.75	0.00229	54.1	0.162	13.4
700	100	3.7	0.0057	44.6	0.123	11
700	200	3.74	0.0049	50.9	0.138	12.6
700	300	3.76	0.0018	53.3	0.134	13.2
700	400	3.76	0.00229	54.8	0.16	13.6

Tabela 12: Resultados da execução da função FillRectangle sem sistema operacional, em MPixel/s

Largura	Altura	Software		Hardware		**Ganho**
		Média	Desvio Padrão	Média	Desvio Padrão	
100	100	7.63	0.000632	66.7	0.00636	7.75
100	200	7.64	0.00064	67.8	0.00673	7.88
100	300	7.64	0.000663	68	0.00478	7.9
100	400	7.64	0.0003	68.3	0.00998	7.94
200	100	7.64	0.000663	72.8	0.00999	8.53
200	200	7.64	0.0003	73.4	0.00244	8.6
200	300	7.64	0.0008	73.6	0.005	8.63
200	400	7.64	0.000781	73.7	0.00732	8.64
300	100	7.64	0.00064	75	0.00424	8.82
300	200	7.64	0.000872	75.5	0.00957	8.88
300	300	7.64	0.0003	75.6	0.00988	8.9
300	400	7.64	0.000894	75.7	0.00914	8.9
400	100	7.64	0.0004	76.3	0.00964	8.99
400	200	7.64	0.000872	76.6	0.00939	9.03
400	300	7.64	0.000831	76.7	0.00765	9.04
400	400	7.64	0.000671	76.8	0.0107	9.05
500	100	7.64	0.000632	74.6	0.0096	8.77
500	200	7.64	0.000894	74.9	0.00704	8.8
500	300	7.64	0.000663	75	0.00988	8.81
500	400	7.64	0.000806	75	0.00765	8.81
600	100	7.64	0.000806	75.5	0.00972	8.88
600	200	7.64	0.000894	75.7	0.00752	8.9
600	300	7.64	0.0004	75.7	0.00951	8.91
600	400	7.64	0.0009	75.8	0.00634	8.92
700	100	7.64	0.000894	76.1	0.00519	8.96
700	200	7.64	0.000872	76.3	0.008	8.98
700	300	7.64	0.0004	76.3	0.00457	8.99
700	400	7.64	0.000748	76.4	0.0106	8.99

7 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou os primeiros passos em direção a um STB capaz de executar aplicações interativas para TVD de acordo com o padrão brasileiro Ginga partindo do ambiente de execução declarativo Ginga-NCL.

A implementação de referência do middleware Ginga-NCL foi alterada de forma a ser compatível com as ferramentas de desenvolvimento para o processador utilizado no STB, o Leon-3. Essa adaptação consistiu em alterar os arquivos responsáveis pela configuração middleware, sem alterar o código-fonte em si. Com essas correções, o middleware pôde ser integrado ao sistema de geração do Linux embarcado da Gaisler utilizando o sistema Buildroot.

Portanto, pode-se dizer que o primeiro objetivo deste trabalho de adequar da implementação de referência foi cumprido com sucesso. A partir dela, o trabalho de expansão pode ser continuado no futuro. Dentro deste objetivo, deve-se salientar a importância de modificar a implementação de referência a fim de que ela seja compatível com cross-compiladores derivados do GCC, não só por ser uma necessidade de trabalho mas também visando facilitar a utilização dela para outras plataformas embarcadas.

A experiência adquirida com a implementação em hardware de uma função do Ginga-NCL permitiu visualizar caminhos que devem ser trilhados para otimizar a execução do middleware Ginga no STB como um todo. A principal lição é que, uma vez que o protocolo de comunicação com o controlador de memória permite trocas de dados em até 64 bytes, implementar um hardware especializado em escritas e leituras da memória permitiria otimizar não só as função gráficas do middleware mas também todas as operações de cópia de dados do sistema.

Além de agora ser possível propor um STB compatível com o Ginga-NCL, este trabalho trouxe indiretamente muitos outros benefícios. Antes de ser iniciado, não havia nenhum ambiente de desenvolvimento para Linux embarcado, dificultando objetivos como controlar o hardware do STB e inviabilizando a utilização de outras funcionalidades do sistema tais como a interface de rede. Como o Ginga-NCL requer um sistema operacional, esse problema acabou tendo que ser resolvido.

A adição dos drivers para dispositivos GRGPIO da Gaisler trouxe a vantagem de facilitar a conexão dos dispositivos do STB com o sistema operacional, pois somente é necessário conectar o dispositivo, sem se preocupar com o barramento AMBA do Leon3. Uma outra vantagem é que aprimorar o driver é aprimorar também a utilização de todos os dispositivos conectados via GRGPIO.

Por fim, como trabalhos futuros, além de continuar trabalhando na otimização do middleware no STB, será necessário também trabalhar sobre o demultiplexador e no código-fonte do middleware. Deverá ser adicionado ao demultiplexador a capacidade de enviar ao middleware as aplicações recebidas via radiodifusão, o que consequentemente tornará

necessário que o middleware reconheça a nova fonte de programas televisivos.

REFERÊNCIAS

ADVANCED RISC MACHINES. **AMBA 2.0 Specification**, 1999. Disponível em : <http://www-micro.deis.unibo.it/ magagni/amba99.pdf>. Acesso em: 01 Jan. 2013.

AEROFLEX GAISLER. **GRLIB IP Library User's Manual**. Disponível em : <http://www.gaisler.com/products/grlib/grlib.pdf>. Acesso em: 01 Jan. 2013.

AEROFLEX GAISLER. **GRLIB IP Core User's Manual**. Disponível em : <http://www.gaisler.com/products/grlib/grip.pdf>. Acesso em: 01 Jan. 2013.

AEROFLEX GAISLER. **Download Linux**. Disponível em: <http://www.gaisler.com/index.php/downloads/linux>. Acesso em: 01 Jan. 2013.

AEROFLEX GAISLER. **MKPROM2 Overview**. Disponível em: <http://www.gaisler.com/doc/mkprom.pdf>. Acesso em: 01 Jan. 2013.

AEROFLEX GAISLER. **GRMON User's Manual**. Disponível em : <http://www.gaisler.com/doc/grmon.pdf>. Acesso em: 01 Jan. 2013.

AEROFLEX GAISLER. **BCC - Bare-C Cross-Compiler User's Manual**. Disponível em : <http://www.gaisler.com/doc/bcc.pdf>. Acesso em: 01 Jan. 2014.

ARAÚJO, E. C. **Nested Context Language 3.0 Parte 14 - Suíte de Testes de Conformidade para o Ginga-NCL**. 2011. 24 p. Trabalho de Conclusão de Curso (Monografia em Ciências da Computação) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2011.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15602-1** : televisão digital terrestre - codificação de vídeo, áudio e multiplexação: parte 1: codificação de vídeo. Rio de Janeiro, 2007.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15602-2** : televisão digital terrestre - codificação de vídeo, áudio e multiplexação: parte 2: codificação de Áudio. Rio de Janeiro, 2007.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15606-2**: televisão digital terrestre - codificação de dados e especificações de transmissão para radiodifusão digital: parte 2: ginga-ncl para receptores fixos e móveis: linguagem de aplicação xml para codificação de aplicações. Rio de Janeiro, 2007.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15606-1**: televisão digital terrestre - codificação de dados e especificações de transmissão para radiodifusão digital: parte 1: codificação de dados. Rio de Janeiro, 2007.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15606-4**: televisão digital terrestre - codificação de dados e especificações de transmissão para radiodifusão digital: parte 4: ginga-j - ambiente para a execução de aplicações procedurais. Rio de Janeiro, 2012.

BARBOSA, S. D. J.; SOARES, L. F. G. TV digital interativa se faz com Ginga: fundamentos, padrões, autoria declarativa e usabilidade. In: KOWALTOWSKI, T.; BREITMAN, K. (Ed.). **Atualizações em Informática**. Rio de Janeiro: PUC-RJ, 2008. p.105–174. Disponível em : <http://www.ncl.org.br/documentos/JAI2008.pdf>. Acesso em: 01 Jan. 2013.

BONATTO, A. et al. Hardware Decoding Architecture for H.264/AVC Digital Video Standard. In: AKSHAYA MISHRA ZAFAR NAWAZ, Z. S. (Ed.). **Image and Video Processing**: an introductory guide. [S.l.]: iConcept Press, 2013. p.27. Disponível em: <http://www.iconceptpress.com/books/image-and-video-processing-an-introductory-guide>. Acesso em: 01 Jan. 2013.

BONATTO, A.; SOARES, A.; SUSIN, A. Multichannel SDRAM controller design for H.264/AVC video decoder. In: SOUTHERN CONFERENCE ON PROGRAMMABLE LOGIC (SPL), 7., 2011, Córdoba. **Proceedings...** [S.l.: s.n.], 2011. p.137–142.

BRASIL. Decreto 4.901, de 26 de novembro de 2003. Institui o Sistema Brasileiro de Televisão Digital - SBTVD, e dá outras providências. **Diário Oficial da União**. Brasília, DF, 27 nov. 2003. Disponível em: <http://www.planalto.gov.br/ccivil_03/decreto/2003/D4901.htm>. Acesso em: 01 Abr. 2013.

BRASIL. Decreto 5820, de 29 de Junho de 2006. Dispõe sobre a implantação do SBTVD-T, estabelece diretrizes para a transição do sistema de transmissão analógica para o sistema de transmissão digital do serviço de radiodifusão de sons e imagens e do serviço de retransmissão de televisão, e dá outras providências. **Diário Oficial da União**. Brasília, DF, 30 jun. 2006. Disponível em: <http://www.planalto.gov.br/ccivil_03/_Ato2004-2006/2006/Decreto/D5820.htm>. Acesso em: 01 Abr. 2013.

BUILDROOT. **Buildroot**: making Embedded Linux easy. Disponível em: < http://buildroot.uclibc.org >. Acesso em: 01 Jan. 2013.

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. **Linux Device Drivers**. 3.ed. [S.l.]: O'Reilly Media, 2005.

CRUZ, V. M.; MORENO, M. F.; SOARES, L. F. G. Ginga-NCL: implementação de referência para dispositivos portáteis. In: BRAZILIAN SYMPOSIUM ON MULTIMEDIA AND THE WEB (WEBMEDIA '08), 14., 2012, New York. **Proceedings...** ACM, 2008. p.67–74.

DEBIAN. **Debian**: the universal operating system. Disponível em: <http://www.debian.org >. Acesso em: 01 Jan. 2013.

DIRECTFB. **DirectFB**. Disponível em : <http://www.directfb.org>. Acesso em: 01 Jan. 2014.

FREE SOFTWARE FOUNDATION. **GNU make**. Disponível em: <https://www.gnu.org/software/make/manual/make.html>. Acesso em: 01 Jan. 2013.

FREE SOFTWARE FOUNDATION. **GNU Automake**. Disponível em: <http://www.gnu.org/software/automake/manual/>. Acessado em: 01 Jan. 2013.

FREE SOFTWARE FOUNDATION. **GNU Autoconf**. Disponível em: <https://www.gnu.org/software/autoconf/manual/autoconf.html>. Acessado em: 01 Jan. 2013.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. d.; CELES, W. **Lua 5.1 Reference Manual**. [S.l.: s.n.], 2006. Disponível em: <http://www.lua.org/manual/5.1/pt/>. Acesso em: 01 Jan. 2013.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 13818-6**: Information Technology : Generic coding of moving pictures and associated audio information : Part 6 - Extensions for DSM-CC. Geneva, 1998.

LIM, C.; LEE, B.-D. Development of a GINGA-NCL receiver for Brazilian mobile broadcasting services. **Transactions on Consumer Electronics, IEEE**, [S.l.], v.57, n.4, p.1535–1540, November 2011.

NEGREIROS, M. et al. Towards a video processing architecture for SBTVD. In: SOUTHERN CONFERENCE ON PROGRAMMABLE LOGIC (SPL), 8., 2012, Bento Gonçalves. **Proceedings...** [S.l.: s.n.], 2012. p.1–6.

PEDRONI, V. A. **Circuit Design and Simulation with VHDL**. 2.ed. Massachusetts: MIT, 2010.

RENNER, A.; SUSIN, A. An MPEG-4 AAC decoder FPGA implementation for the Brazilian digital television. In: SOUTHERN CONFERENCE ON PROGRAMMABLE LOGIC (SPL), 8., 2012, Bento Gonçalves. **Proceedings...** [S.l.: s.n.], 2012. p.1–6.

SAHAFI, L. **Context-Based Complexity Reduction Applied to H.264/AVC Video Compression**. 2005. 24 p. Thesis (Master in Applied Sciences) — School of Enginering Science, Simon Frase University, Burnaby, 2005.

SANT'ANNA, F. et al. **Desenvolvimento de Aplicações interativas para TV Digital no Middleware Ginga com objetos imperativos NCLua**. 2009. Disponível em: <http://www.telemidia.puc-rio.br/sites/telemidia.puc-rio.br/files/MCNCLua.pdf>. Acesso em: 01 Jan. 2013.

SDL. **SDL**: Simple Direct Media Layer. Disponível em: <http://www.libsdl.org>. Acesso em: 01 Jan. 2014.

SOARES, A. B.; BONATTO, A. C.; SUSIN, A. A. Integration Issues on the Development of an H.264/AVC Video Decoder SoC for SBTVD Set Top Box. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 24., 2011, João Pessoa. **Proceedings...** [S.l.: s.n.], 2011. p.125–130.

SOARES, A. B.; BONATTO, A. C.; SUSIN, A. A. Development of a SoC for Digital Television Set-top Box: architecture and system integration issues. **International Journal on Reconfigurable Computing**, New York, v.2013, p.1–10, Jan. 2013.

SOARES, L. et al. Ginga-NCL: declarative middleware for multimedia iptv services. **Communications Magazine, IEEE**, [S.l.], v.48, n.6, p.74–81, 2010.

SOARES, L. F. G.; RODRIGUES, R. F.; MORENO, M. F. Ginga-NCL: the declarative environment of the brazilian digital tv system. **Journal of the Brazilian Computer Society**, [S.l.], v.13, n.1, p.37–46, 2007.

SOARES, L.; RODRIGUES, R.; MORENO, M. Ginga-NCL: the declarative environment of the brazilian digital tv system. **Journal of the Brazilian Computer Society**, [S.l.], v.13, n.1, p.37–46, 2007.

SOUZA FILHO, G. L. de; LEITE, L. E. C.; BATISTA, C. E. C. F. Ginga-J: the procedural middleware for the brazilian digital tv system. **Journal of the Brazilian Computer Society**, [S.l.], v.13, n.1, p.47–56, 2007.

SPARC INTERNATIONAL. **The SPARC Architecture Manual**. 1992. Disponível em : <http://www.gaisler.com/doc/sparcv8.pdf>. Acessado em: 01 Jan. 2013.

STLPORT. **STLPort**. Disponível em: <http://www.stlport.org >. Acesso em: 01 Jul. 2014.

TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU (ITU-T). **Nested context language (NCL) and Ginga-NCL**. Recommendation H.761. Disponível em: <http://www.itu.int/rec/T-REC-H.761>. Acesso em: 01 Jan. 2013.

TELEMIDIA LABS. **Implementação de Referência do Ginga-NCL**. Disponível em: <http://git.telemidia.puc-rio.br>. Acessado em: 01 Jan. 2013.

TELEMIDIA LABS. **Documentação NCLua**. Disponível em: <http://www.telemidia.puc-rio.br/ francisco/nclua/>. Acesso em: 01 Jan. 2013.

TELEMIDIA LABS. **Testsuite Ginga-NCL**. Disponível em <http://testsuite.gingancl.org.br>. Acessado em: 01 Jan. 2013.

TORVALDS, L. **Kconfig Language**. Disponível em: <http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt>. Acessado em: 01 Jan. 2013.

VENKATESWARAN, S. **Essential linux device drivers**. 1.ed. Upper Saddle River: Prentice Hall, 2008.

APÊNDICE A PROBLEMAS DE PORTABILIDADE DO GINGA-NCL

Este apêndice visa desenvolver com maiores detalhes os principais problemas encontrados durante o porte do middleware Ginga-NCL para o sistema da Gaisler e as soluções adotadas. O principal desafio encontrado durante o trabalho foram os arquivos de configuração dos diversos componentes do Ginga-NCL, os quais não seguiam os padrões do sistema de distribuição das Autotools.

A.1 configure.ac

Os scripts *configure.ac* são compostos por uma série de macros que são transformados pelo programa *Autoconf* (FREE SOFTWARE FOUNDATION, 2014c) da Autotools no script *configure*.

Um defeito encontrado nos scripts "configure.ac"do Ginga-NCL é que eles buscam e instalam o Middleware de forma estática. Isso vai de encontro aos mecanismos utilizados pelo Buildroot para cross-compilação e do próprio "autoconf".

Um exemplo de trecho de código com tais problemas é mostrado na Listagem A.1. O principal problema que ocorre ao utilizar a busca estática mostrada é que as Toolchains já utilizam por padrão um diretório de busca diferente daquele usado pelo sistema. Ao utilizar as inserções como mostradas, a Toolchain irá adicionar bibliotecas e cabeçalhos do sistema nativo, resultando em erros.

```
1 for spfx in /usr/lib /usr/local/lib /usr/local/lib/ginga; do
2   if test -d ${spfx}; then
3     LDFLAGS="-L${spfx} $LDFLAGS"
4   fi
5 done
```

Listagem A.1: Exemplo de caminho estático para busca. Note a adição estática do diretório /usr/lib, o que incluirá o sistema de arquivo nativo durante a cross-compilação.

A solução adotada neste trabalho envolve duas partes. A primeira delas é condicionar a inserção de diretórios de busca estáticas apenas quando não é detectado ambiente de cross-compilação, verificado pela variável *cross_compiling*. Essa variável é configurada automaticamente pelo script "configure"e indica se o ambiente está em modo de cross-compilação. A Listagem A.2 ilustra esse passo.

```
1 if [ x$cross_compiling = xno ]; then
2   for spfx in /usr/lib /usr/local/lib /usr/local/lib/ginga; do
3     if test -d ${spfx}; then
4       LDFLAGS="-L${spfx} $LDFLAGS"
5     fi
```

```
6     done
7 fi
```

Listagem A.2: Primeira parte da solução adotada para os diretórios de busca estáticos nos scripts "configure.ac"

A segunda parte é utilizar os Makefiles do Buildroot, criados para cada componente do Ginga-NCL, para inserir os caminhos adequados. Isso é feito definindo variáveis de ambiente antes da chamada pelo script, ilustrado na Listagem A.3. A solução consiste em definir variáveis de ambiente com os diretórios de busca adequados, o que no caso significa prefixa-los com o STAGING_DIR, o lugar onde os cabeçalhos são realmente instalados.

```
1 GINGACC_PLAYER_CONF_ENV = \
2     CPPFLAGS="-I$(STAGING_DIR)/usr/include/ginga" \
3     LDFLAGS="-L$(STAGING_DIR)/usr/lib/ginga"
```

Listagem A.3: Parte 2 da solução adotada para os diretórios de busca estáticos nos arquivos "configure.ac"

A.2 Makefile.am

Os arquivos Makefile.am são protótipos utilizados pela ferramenta *Automake* (FREE SOFTWARE FOUNDATION, 2014b) para gerar os Makefiles finais de compilação de um projeto. No caso do Ginga-NCL, muitos dos arquivos "Makefile.am"dos componentes do Ginga-NCL adicionam internamente diretórios utilizando a variável *prefix*. Um exemplo disso é mostrado na Listagem A.4. Dentro das Autotools, o "prefix"indica o diretório base para instalação do programa recém-compilado, normalmente padronizado para "/usr/local".

```
1 libgingaccdataprocessing_la_LIBADD = -ltelemidiautil -lgingaccsystem \
2       $(COMPONENTS_LIBS) -L$(prefix)/lib/ -L$(prefix)/lib/ginga/
```

Listagem A.4: Inserção da variável *prefix* nos Makefile.am

O Buildroot, por padrão, configura essa variável para "/usr". Consequentemente, durante a compilação acaba-se adicionando diretórios do sistema de arquivos da máquina hospedeira, entrando em conflito com o do sistema-alvo, recaindo no mesmo problema da Seção A.1.

Para resolver esse problema, bastou apagar todas essas referências.

APÊNDICE B TESTES COM PROGRAMAS NCL

B.1 Script de teste

```
#!/bin/bash

killall -9 ginga

while [ "$1" != "" ]; do
  case $1 in

  # clear stuff
  -c )
    shift
    case $1 in
      ts | timestamps )
        clear_timestamps=y
      ;;
    esac;
  ;;

  *)
    FILELIST="$1"
  ;;
  esac;

  shift
done

# initialize files
LOG_EXEC=`mktemp`
LOG_TIMESTAMPS=$PWD/log_timestamps.txt

# clear log of timestamps
if [[ "$clear_timestamps" == "y" || ! -f $LOG_TIMESTAMPS ]]; then
  rm -f $LOG_TIMESTAMPS | touch $LOG_TIMESTAMPS
fi

echo "ola"

# if file with list of tests is not passed or list is empty, exit with
    error
if [[ ! -f $FILELIST || `cat $FILELIST | wc -l` -eq 0 ]]; then
  echo "No file with list of tests to run! Aborting"
  exit 1
fi
```

```
42
43 # computes tests to run
44 tests_ran=`cat $LOG_TIMESTAMPS | wc -l`
45 all_tests=`cat $FILELIST | wc -l`
46 tests_to_run=$((all_tests-tests_ran))
47
48 echo "Tests already ran: $tests_ran"
49 echo "Total Tests: $all_tests"
50 echo "Tests to run: $tests_to_run"
51
52 # initializes timestamp multiplying factor
53 MULTFACTOR=1
54
55 # passes through all tests from testsuite
56 for fullncl in $(cat $FILELIST | tail -n $tests_to_run); do
57   root=`pwd`
58   fullpath=`find -name $fullncl`
59
60   echo "################################"
61   echo "root: $root"
62   echo "fullpath: $fullpath"
63
64   path=`dirname $fullpath`
65   ncl=`basename $fullpath`
66
67   echo "ncl: $ncl"
68   echo "path: $path"
69
70   cd $path
71
72   [ ! -d media ] && ln -s ../../medias media
73
74   RET=-1
75   while [[ $RET -neq  0 ]]; do
76
77     ginga --ncl test.ncl > $LOG_EXEC
78     RET=$?
79     if [[ $RET -neq 0 ]]; then
80       echo -n "$ncl falhou! Incrementando MULTFACTOR ... $MULTFACTOR ->
       "
81       MULTFACTOR=$((MULT_FACTOR+1))
82       echo $MULTFACTOR
83       cat $ncl | sed
84         -e "s/2s/$((MULTFACTOR*2))s/g"
85         -e "s/3s/$((MULTFACTOR*3))s/g"
86         -e "s/2s/$((MULTFACTOR*5))s/g"
87         -e "s/6s/$((MULTFACTOR*6))s/g" > test.ncl
88     else
89
90     fi
91   done
92
93   cat $LOG_EXEC | grep -m 1 "TimeStamp" | egrep -o "[[:digit:]]*\.?[[:
       digit:]]*" >> $LOG_TIMESTAMPS
94
95   cat $LOG_TIMESTAMPS | awk 'BEGIN { n=0; sum=0 } { n=n+1; sum=sum+$1 }
       END { print "Total de testes: ", n "\nSoma dos tempos:", sum }'
96
```

```
97    echo "Fator de multiplicaÃ§Ã£o parcial: $MULTFACTOR"
98
99    cd $root
100 done
101
102 echo -e "***Termino dos testes***\nResultados:"
103
104 cat $LOG_TIMESTAMPS | awk
105   'BEGIN { n=0; sum=0 } { n=n+1; sum=sum+$1 } END { print "Total de
        testes: ", n ,"\nSoma dos tempos:", sum,"Tempo mÃ©dio de
        inicializaÃ§Ã£o: ",sum/n }'
```

B.2 Testes executados com sucesso

area01.01.01.ncl area01.02.01.ncl area01.03.01.ncl area01.04.01.ncl area01.05.01.ncl area01.06.01.ncl area02.01.01.ncl area02.02.01.ncl area03.01.01.ncl area03.02.01.ncl area04.01.01.ncl area10.01.07.ncl bind01.11.01.ncl bind02.02.02.ncl bind02.02.02.ncl bind02.03.01.ncl bind02.04.01.ncl bind02.07.01.ncl bind02.08.01.ncl bind05.01.01.ncl bind05.02.01.ncl causalConnector01.01.01.ncl causalConnector01.02.01.ncl compoundCondition01.04.01.ncl compoundCondition01.05.01.ncl connectorParam01.01.01.ncl context01.01.01.ncl context03.01.01.ncl context03.02.01.ncl context04.01.01.ncl context06.01.01.ncl descriptor01.01.01.ncl descriptor01.01.02.ncl descriptor01.01.03.ncl descriptor01.01.04.ncl descriptor01.01.07.ncl descriptor01.01.08.ncl descriptor01.01.09.ncl descriptor01.01.10.ncl descriptor01.02.02.ncl descriptor03.01.01.ncl descriptor03.01.02.ncl descriptor03.01.03.ncl descriptor03.01.04.ncl descriptor03.01.07.ncl descriptor03.01.08.ncl descriptor03.01.09.ncl descriptor03.01.01.ncl descriptor04.01.02.ncl descriptor06.01.01.ncl descriptor06.01.02.ncl descriptor06.01.03.ncl descriptor06.01.04.ncl descriptor07.01.01.ncl descriptor07.01.02.ncl descriptor07.01.03.ncl descriptor07.01.04.ncl descriptor08.01.01.ncl descriptor08.01.02.ncl descriptor08.01.03.ncl descriptor08.01.04.ncl descriptor09.01.01.ncl descriptor09.01.02.ncl descriptor09.01.03.ncl descriptor09.01.04.ncl

B.3 Testes executados com falha

bind01.01.01.ncl bind01.02.01.ncl bind01.03.01.ncl bind01.04.01.ncl bind01.05.01.ncl bind01.06.01.ncl bind01.07.01.ncl bind01.09.01.ncl bind01.12.01.ncl bind02.05.01.ncl bind02.06.01.ncl bind02.09.01.ncl bind02.10.01.ncl bind04.01.01.ncl bind04.02.01.ncl bindParam01.01.01.ncl bindParam02.01.01.ncl causalConnector02.01.01.ncl compoundCondition01.01.01.ncl compoundCondition01.03.01.ncl compoundCondition02.01.01.ncl connectorParam02.01.01.ncl context02.01.01.ncl context05.01.01.ncl link02.01.01.ncl link02.02.01.ncl linkParam01.01.01.ncl linkParam02.01.01.ncl port01.01.01.ncl port01.02.02.ncl port01.03.01.ncl port01.03.02.ncl port02.01.01.ncl port02.03.01.ncl port01.03.01.ncl port01.03.02.ncl port02.01.01.ncl port02.03.01.ncl property09.01.01.ncl property09.01.02.ncl property09.01.03.ncl property09.01.04.ncl property09.01.07.ncl property09.01.08.ncl property09.02.02.ncl property09.11.01.ncl property10.01.01.ncl property10.01.02.ncl property10.01.03.ncl property10.01.04.ncl property10.01.07.ncl property10.01.08.ncl property10.01.09.ncl property10.01.10.ncl property10.02.01.ncl property10.02.02.ncl property10.02.03.ncl property10.02.04.ncl property10.02.07.ncl property10.02.08.ncl property10.02.09.ncl property10.02.10.ncl simpleAction01.01.01.ncl simpleAction01.04.01.ncl simpleAction01.07.01.ncl simpleAction02.01.01.ncl simpleAction02.02.01.ncl simpleAction02.05.01.ncl simpleAction02.07.01.ncl simpleAction02.08.01.ncl simpleAction07.01.01.ncl simpleAction08.01.01.ncl simpleAction08.02.01.ncl simpleAction08.03.01.ncl simpleAc-

tion08.04.01.ncl simpleAction08.05.01.ncl simpleAction08.06.01.ncl simpleAction08.07.01.ncl simpleAction08.08.01.ncl simpleAction08.09.01.ncl simpleAction08.10.01.ncl simpleAction08.11.01.ncl simpleAction08.12.01.ncl simpleAction09.01.01.ncl simpleAction10.01.01.ncl simpleAction10.02.01.ncl simpleCondition01.01.01.ncl simpleCondition01.02.01.ncl simpleCondition02.04.01.ncl simpleCondition02.05.01.ncl simpleCondition02.06.01.ncl simpleCondition02.07.01.ncl simpleCondition02.08.01.ncl simpleCondition02.09.01.ncl simpleCondition03.01.01.ncl simpleCondition04.01.01.ncl simpleCondition04.02.01.ncl simpleCondition04.03.01.ncl simpleCondition05.01.01.ncl simpleCondition06.01.01.ncl simpleCondition07.06.01.ncl simpleCondition07.07.01.ncl simpleCondition07.08.01.ncl simpleCondition08.01.01.ncl simpleCondition09.01.01.ncl simpleCondition09.02.01.ncl

APÊNDICE C TESTE DE DESEMPENHO DA FUNÇÃO FILLRECTANGLE SEM SISTEMA OPERACIONAL

```
1 //*******************************************************
2 // ATENCAO: Ajustar o define pra 0 quando rodar em placa
3 //*******************************************************
4 #define SIMULACAO_SOMENTE 0
5
6 #define OSD_WIDTH 720
7 #define OSD_HEIGHT  480
8
9 #define INI_WIDTH 100
10 #define END_WIDTH 700
11 #define INC_WIDTH 100
12
13 #define INI_HEIGHT  100
14 #define END_HEIGHT  400
15 #define INC_HEIGHT  100
16
17 #define TOTAL_PASSES  (((END_WIDTH-INI_WIDTH)/INC_WIDTH+1)*((END_HEIGHT
     -INI_HEIGHT)/INC_HEIGHT+1))
18
19 #define TOTAL_REPS 10
20
21 #define HW    0
22 #define SW_32   1
23 #define SW_64   2
24
25 #define DEBUG
26
27
28 const volatile long* gpio1_addr = (volatile long*) 0x80000B00;
29 const volatile long* gpio2_addr = (volatile long*) 0x80000D00;
30 const volatile long* gpio3_addr = (volatile long*) 0x80000E00;
31 const long *OSD = (long*)0x43C00000;
32
33 #include <asm-leon/timer.h>
34 #include <math.h>
35
36 static int fillrect_hw(long x_i,long y_i,long width_i,long height_i,
     long pixel_i);
37 static int fillrect_sw_32(long x_i,long y_i,long width_i,long height_i,
     long pixel_i);
38 static int fillrect_sw_64(long x_i,long y_i,long width_i,long height_i,
     long long pixel_i);
```

```
39
40 typedef struct {
41   long width,height;
42   float times[TOTAL_REPS];
43 } statistics;
44
45 statistics results[TOTAL_PASSES][3];
46
47 main()
48 {
49   long colors_32[]= {0xFFFFFFFF,0xFFFFFF00,0xFF00FFFF,0xFF00FF00,0
     xFFFF00FF,0xFFFF0000,0xFF0000FF,0xFF000000};
50   long long colors_64[]={0xFFFFFFFFFFFFFFFF,0xFFFFFF00FFFFFF00,0
     xFF00FFFFFF00FFFF,0xFF00FF00FF00FF00,
51    0xFFFF00FFFFFF00FF,0xFFFF0000FFFF0000,0xFF0000FFFF0000FF,0
     xFF000000FF000000};
52
53   if (SIMULACAO_SOMENTE)
54     report_start();
55
56   int l,c,w,h,run,x,y,i,j;
57
58   int width,height,pass;
59
60   long mean_fillrect_hw,mean_fillrect_sw_32,mean_fillrect_sw_64;
61   long dev_fillrect_hw,dev_fillrect_sw_32,dev_fillrect_sw_64;
62
63   w=720/8;
64   h=480;
65
66   leonbare_init_ticks();
67
68   for (pass=0,width=INI_WIDTH;width<=END_WIDTH;width+=INC_WIDTH)
69     for (height=INI_HEIGHT;height<=END_HEIGHT;height+=INC_HEIGHT,pass
     ++)
70     {
71       printf("*****************************************\n");
72       printf("*** Benchmark %0.2d (WIDTH=%0.2d,HEIGHT=%0.2d) ***\n",
     pass+1,width,height);
73       printf("*****************************************\n");
74
75       results[pass][HW].height=results[pass][SW_32].height=results[pass
     ][SW_64].height=height;
76       results[pass][HW].width=results[pass][SW_32].width=results[pass][
     SW_64].width=width;
77
78       printf("-------------------------------------------------\n");
79       printf("1) Benchmarking fillrect_hw(%d,%d)\n",width,height);
80       fillrect_hw(0,0,720,480,0x00000000);
81       for (run=0;run<TOTAL_REPS;run++){
82         int n,ts,tn,te;
83         float MPix,t;
84
85         ts=clock();
86         for (n=0;(clock()-ts)<1000000;n++)
87           fillrect_hw(0,0,width,height,colors_32[n%8]);
88         te=clock();
89
```

```
90          MPix=(float)(width*height*n)/1000000.0;
91          t=(float)(te-ts);
92          t/=1000000.0;
93
94          results[pass][HW].times[run]=(MPix/t);
95
96          printf("\tPass %d: time=%0.3f, nrects=%d, MPix=%0.3f, MPix/s =
        %.3f \n",run+1,t,n,MPix,results[pass][HW].times[run]);
97        }
98
99        printf("-------------------------------------------------------\n");
100       printf("2) Benchmarking fillrect_sw32(%d,%d)\n",width,height);
101       fillrect_hw(0,0,720,480,0x00000000);
102       for (run=0;run<TOTAL_REPS;run++){
103         int n,ts,tn,te;
104         float MPix,t;
105
106         ts=clock();
107         for (n=0;(clock()-ts)<1000000;n++){
108           //printf("%u\n",clock());
109           fillrect_sw_32(0,0,width,height,colors_32[n%8]);
110         }
111         te=clock();
112
113         MPix=(float)(width*height*n)/1000000.0;
114         t=(float)(te-ts);
115         t/=1000000.0;
116
117         results[pass][SW_32].times[run]=(MPix/t);
118
119         printf("\tPass %d: time=%0.3fs, nrects=%d, MPix=%.3f, MPix/s =
        %.3f \n",run+1,t,n,MPix,results[pass][SW_32].times[run]);
120
121       }
122
123       printf("-------------------------------------------------------\n");
124       printf("3) Benchmarking fillrect_sw_64(%d,%d)\n",width,height);
125
126       for (run=0;run<TOTAL_REPS;run++){
127         unsigned n,ts,tn,te;
128         float MPix,t;
129
130         ts=clock();
131         for (n=0;(clock()-ts)<1000000;n++)
132           fillrect_sw_64(0,0,width,height,colors_64[n%8]);
133         te=clock();
134
135         MPix=(float)(width*height*n)/1000000.0;
136         t=(float)(te-ts);
137         t/=1000000.0;
138
139         results[pass][SW_64].times[run]=(MPix/t);
140
141         printf("\tPass %d: time=%0.3fs, nrects=%u, MPix=%0.3f, MPix/s =
         %.3f \n",run+1,t,n,MPix,results[pass][SW_64].times[run]);
142       }
143
144       printf("\n");
```

```
145     }
146
147   printf("Tests ended. Printing results: \n\nSTART_HW\n");
148   for (i=0;i<pass;i++){
149     printf("%d\t%d\t",results[i][HW].width,results[i][HW].height);
150
151     for (j=0;j<TOTAL_REPS;j++){
152       printf("%0.3f\t",results[i][HW].times[j]);
153     }
154
155     printf("\n");
156   }
157   printf("END_HW\n\n");
158
159   printf("START_SW_32\n");
160   for (i=0;i<pass;i++){
161     printf("%d\t%d\t",results[i][SW_32].width,results[i][SW_32].height)
      ;
162
163     for (j=0;j<TOTAL_REPS;j++){
164       printf("%0.3f\t",results[i][SW_32].times[j]);
165     }
166
167     printf("\n");
168   }
169   printf("END_SW_32\n\n");
170
171   printf("START_SW_64\n");
172   for (i=0;i<pass;i++){
173     printf("%d\t%d\t",results[i][SW_64].width,results[i][SW_64].height)
      ;
174
175     for (j=0;j<TOTAL_REPS;j++){
176       printf("%0.3f\t",results[i][SW_64].times[j]);
177     }
178
179     printf("\n");
180   }
181   printf("END_SW_64\n");
182
183
184   if (SIMULACAO_SOMENTE)
185     report_end();
186 }
187
188 static
189 int fillrect_sw_32(long x_i,long y_i,long width_i,long height_i,long
      pixel_i){
190   int x,y;
191
192   long *ptr=(long*)OSD;
193
194   for (x=x_i;x<x_i+height_i;x++)
195     for (y=y_i;y<y_i+width_i;y++){
196       *(ptr+720*x+y)=pixel_i;
197     }
198
199   return 0;
```

```
200 }
201
202 static
203 int fillrect_sw_64(long x_i,long y_i,long width_i,long height_i,long
      long pixel_i){
204   int x,y;
205
206   long long *ptr=(long long *)OSD;
207
208   for (x=x_i;x<x_i+height_i;x++)
209     for (y=y_i;y<y_i+width_i/2;y++){
210       *( ptr+360*x+y)=pixel_i;
211     }
212
213   return 0;
214 }
215
216
217 int fillrect_hw(long x_i,long y_i,long width_i,long height_i,long
      pixel_i){
218   /*
219   EndereÃ§os das portas de GPIO
220   */
221   volatile long* gpio1 = (volatile long*)gpio1_addr;
222   volatile long* gpio2 = (volatile long*)gpio2_addr;
223   volatile long* gpio3 = (volatile long*)gpio3_addr;
224
225   /*
226   offsets dos registradores da GPIO
227   */
228   const long off_data   = 0x00; //data: porta de entrada
229   const long off_output = 0x01; //output: porta de saida - long tem 4
      bytes e offset eh de 4 bytes
230
231   //Inicializa o pixel
232   *(gpio3+off_output)=pixel_i;
233
234   //Inicializa width_i e height_i - feito pela GPIO2
235   *(gpio2+off_output)=(long)( (0x000FFC00&(width_i<<10))|(0x000003FF &
      height_i) );
236
237   //Espera o sinal de finalizacao da operaÃ§Ã£o anterior ser baixado
      pelo hardware
238   while ((*(gpio1+off_data)&0x40000000) != 0);
239
240   //Inicializa x_i, y_i, e manda escrever - feito pela GPIO1
241   *(gpio1+off_output)=(long)( (0x000FFC00&(x_i<<10))|(0x000003FF & y_i)
      |0x80000000 );
242
243   //Espera escrita terminar - indicado quando bit 30 do registrador
      data da GPIO1 eh igual a 1
244   while ((*(gpio1+off_data)&0x40000000) == 0);
245
246   //Abaixa o sinal cmd_i, indicando que recebeu o comando
247   *(gpio1+off_output)=(long)(0);
248
249   return 0;
250 }
```

APÊNDICE D TESTE DE DESEMPENHO DA FUNÇÃO FILLRECTANGLE COM SISTEMA OPERACIONAL

```
#!/bin/bash


INI_WIDTH=100
INC_WIDTH=50
END_WIDTH=600

INI_HEIGHT=100
INC_HEIGHT=50
END_HEIGHT=300

REPETITIONS=2

DFB_OPTS="--dfb:no-debug --dfb:no-trace --dfb:quiet --noresults"

tmpfile=`mktemp`

fileout_hw="resultados_fillrect_com-SO_HW.dat"
fileout_sw="resultados_fillrect_com-SO_SW.dat"

echo "" > $fileout_hw
echo "" > $fileout_sw

for width in $(seq $INI_WIDTH $INC_WIDTH $END_WIDTH)
do
  for height in $(seq $INI_HEIGHT $INC_HEIGHT $END_HEIGHT)
  do
    echo "Benchmarking with width=$width and height=$height"

    results_accel=""
    results_noaccel=""

    for rep in $(seq $REPETITIONS)
    do
      df_dok $DFB_OPTS --fill-rect --size ${width}x${height} --accelonly > $tmpfile
      accel=`cat $tmpfile | grep "Fill\ Rectangle" | egrep -o "\(.*\)" | egrep -o "[[:digit:]]*\.[[:digit:]]*"`
      results_accel="$results_accel\t$accel"

      df_dok $DFB_OPTS --fill-rect --size ${width}x${height} --noaccel > $tmpfile
```

```
40        noaccel=‘cat $tmpfile | grep "Fill\ Rectangle" | egrep -o "\(.*\)
      " | egrep -o "[[:digit:]]*\.[[:digit:]]*"‘
41        results_noaccel="$results_noaccel\t$noaccel"
42
43        echo -e "\tPass $rep: HW=$accel\tSW=$noaccel"
44      done
45
46      echo -e "${width}\t${height}${results_accel}" >> $fileout_hw
47      echo -e "${width}\t${height}${results_noaccel}" >> $fileout_sw
48    done
49 done
50
51 echo "Done. Printing results"
52
53 echo "START_WITH_OS_SW"
54 cat $fileout_sw
55 echo "END_WITH_OS_SW"
56
57 echo "START_WITH_OS_HW"
58 cat $fileout_hw
59 echo "END_WITH_OS_HW"
```

ANEXO A INSTALAÇÃO DAS DEPENDÊNCIAS DO GINGA

Abaixo, o script utilizado como base para o estudo das dependências do Ginga.

```
#!/bin/bash

OK="\033[30;32mOK\033[m"
FAILED="\033[30;31mFAILED\033[m"
UNCHANGED="\033[30;32mUNCHANGED\033[m"
ERROR_MSG1="Erro durante a compilação do Ginga Common Core. Você pode compilar à partir do diretório gingacc/ para obter mais informações. [ cd gingacc/ && make]"
OUT='>/dev/null 2>&1'
AND_TEST=" && echo -e $OK || cd ../ echo -e $FAILED && exit 1;"
OUT_TEST="$OUT $AND_TEST"
BASEDIR=`pwd`

#info:
echo -e -n "Instalando dependencias do  OpenGinga. Esta operação pode levar vários minutos. Aguarde..."

#aptitude:
echo -e -n "Instando dependencias do aptitude ..."
sudo apt-get update -y
sudo apt-get upgrade -y
sudo apt-get install -y automake subversion git-core build-essential patch libssl-dev libcppunit-dev autoconf libcurl4-openssl-dev libreadline6-dev libexpat1-dev libxerces-c2-dev libmad0-dev libtiff4-dev libkrb5-dev libgpm-dev  x11proto-xext-dev libxext-dev libpng12-dev libjpeg62-dev libfreetype6-dev libavcodec-dev libavformat-dev libxine-dev libxine1 libxine1-ffmpeg libdirectfb-extra libtool liblua5.1-0-dev libzip-dev libzip1 libavcodec-extra-52  libavformat-extra-52 ffmpeg libvorbis-dev >/dev/null 2>&1 && echo -e $OK || (cd ../ echo -e $FAILED && exit 1);

#fusion:
echo -e -n "Instalando linux-fusion ..."
if ! test -e linux-fusion
then
  tar -xzf linux-fusion.tar.gz&& \
  cd linux-fusion &&\
  git checkout af15f7396d96c5d7079ca75703ccb128065a5196 &&\
  patch -Np0 < ../fusion_kernel_version.patch &&\
  make &&\
  sudo make install &&\
  sudo depmod -a &&\
  sudo modprobe fusion &&\
```

```
33   sudo chmod 777 /dev/fusion* &&\
34   sudo chmod 666  /etc/initramfs-tools/modules &&\
35   sudo echo -e "fusion" >> /etc/initramfs-tools/modules &&\
36   sudo update-initramfs -u &&\
37   cd ../ && echo -e $OK || (cd ../ echo -e $FAILED && exit 1);
38 else
39   echo -e $UNCHANGED
40 fi
41
42 #directfb:
43 echo -e -n "Instalando DirectFB-1.4 ..."
44 if ! test -e DirectFB-1.4.5
45 then
46   sudo rm -rf DirectFB-1.4.5  && tar -xvzf DirectFB-1.4.5.tar.gz&&\
47   cd DirectFB-1.4.5 &&\
48   ./configure --enable-multi --enable-x11 --with-gfxdrivers=none&&\
49   make&&\
50   sudo make install&&\
51   cd ../ && echo -e $OK || (cd ../ echo -e $FAILED && exit 1);
52 else
53   echo -e $UNCHANGED
54 fi
55
56 #fusionsound:
57 echo -e -n "Instalando FusionSound ..."
58 if ! test -e FusionSound
59 then
60   tar -xzf FusionSound.tar.gz &&\
61   cd FusionSound &&\
62   git checkout 6c9f3e777453b0cdbdf3e1db57ae4bfd7dd922d5 &&\
63   ./autogen.sh&&\
64   make &&\
65   sudo make install&&\
66   cd ../ && echo -e $OK || (cd ../ echo -e $FAILED && exit 1);
67 else
68   echo -e $UNCHANGED
69 fi
70
71 #dfbextra:
72 echo -e -n "Instalando DirectFB Extras..."
73 if ! test -e DirectFB-extra
74 then
75   tar -xzf DirectFB-extra.tar.gz &&\
76   cd DirectFB-extra &&\
77   git checkout 9e63dda95c580f72f8b54de7cae79ded76823853 &&\
78   ./autogen.sh &&\
79   make &&\
80   sudo make install&&\
81   cd ../ && echo -e $OK || (cd ../ echo -e $FAILED && exit 1);
82 else
83   echo -e $UNCHANGED
84 fi
85
86
87 #lua:
88 echo -e -n "Instalando Lua ..."
89 if ! test -e lua-5.1.4
90 then
```

```
91        tar -xvzf lua-5.1.4.tar.gz &&\
92        cd lua-5.1.4 &&\
93        make linux &&\
94        sudo make install &&\
95    cd ../ && echo -e $OK || (cd ../ echo -e $FAILED && exit 1);
96  else
97    echo -e $UNCHANGED
98  fi
99
100 #luasocket:
101 echo -e -n "Instalando LuaSocket ..."
102 if ! test -e luasocket-2.0.2
103 then
104   tar -xzf luasocket-2.0.2.tar.gz &&\
105   cd luasocket-2.0.2 &&\
106   make &&\
107   sudo make install &&\
108   sudo rm -rf  /usr/local/lib/lua/5.1/socket/libcore.so &&\
109   sudo ln -s /usr/local/lib/lua/5.1/socket/core.so /usr/local/lib/lua
      /5.1/socket/libcore.so &&\
110   cd ../ && echo -e $OK || (cd ../ echo -e $FAILED && exit 1);
111 else
112   echo -e $UNCHANGED
113 fi
114
115 #jlibcpp:
116 echo -e -n "Instalando jlibcpp ..."
117 if ! test -e jlibcpp
118 then
119   tar -xzf jlibcpp.tar.gz &&\
120   cd jlibcpp &&\
121   make&&\
122   sudo make install&&\
123   cd ../ && echo -e $OK || (cd ../ echo -e $FAILED && exit 1);
124 else
125   echo -e $UNCHANGED
126 fi
127
128 #flexcmee:
129 echo -e -n "Instalando flexcm ..."
130 if ! test -e flexcm
131 then
132    tar -xzf flexcm2.tar.gz &&\
133     make -sC flexcm&&\
134     sudo make install -sC flexcm&&\
135     cd ../ && echo -e $OK || (cd ../ echo -e $FAILED && exit 1);
136 else
137   echo -e $UNCHANGED
138 fi
139
140 #jdk:
141 echo -e -n "Instalando jdk 1.4.2 ..."
142 if ! test -e j2sdk1.4.2_19
143 then
144   tar -xzf j2sdk1.4.2_19.tar.gz &&\
145   sudo cp -r j2sdk1.4.2_19 /opt
146   cd ../ && echo -e $OK || (cd ../ echo -e $FAILED && exit 1);
147 else
```

```
148   echo -e $UNCHANGED
149 fi
150
151 #directfbrc:
152 if test -e ~/.directfbrc
153 then
154   rm  ~/.directfbrc
155 fi
156 echo -e -n "Instalando ~/.directfbrc ..." &&\
157   echo -e "system=x11 \nmode=960x540 \npixelformat=ARGB \nquiet \nno-
      debug \nno-trace" >> ~/.directfbrc && echo -e $OK || (echo -e
      $FAILED && exit 1) ;
```

ANEXO B INSTALAÇÃO DO GINGA-NCL

Este anexo tem por finalidade mostrar os scripts utilizados para o estudo do processo de geração do middleware. Esses scripts foram feitos visando a instalação na distribuição Linux *Debian*

B.1 Script: *build.sh*

Este é o script principal, responsável por compilar e instalar o middleware. Para isso, é necessário que exista uma pasta chamada *src* no mesmo diretório, contendo o código fonte do middleware. Então, por linha de comando, utiliza-se o comando "*sudo ./build.sh install*", compilando e instalando o middleware.

```
1 #!/bin/bash
2 # $Id: build.sh,v 1.2 2010/02/22 20:54:17 gflima Exp $
3 # build.sh -- Build and install source packages.
4 # This is a bash-only script.
5
6 PROGRAM_NAME='build.sh'
7 PROGRAM_VERSION='$Id: build.sh,v 1.2 2010/02/22 20:54:17 gflima Exp $'
8 AUTHORS='Guilherme Lima'
9 BUGS_TO='gflima@telemidia.puc-rio.br'
10
11 # Argv[0] -- initialized at main().
12 program_name=""
13
14 # Version string (printed when --version is given).
15 version_string="\
16 $PROGRAM_VERSION
17 Written by $AUTHORS."
18
19 # Current working directory.
20 cwd=$(pwd)
21
22 # Default configuration file.
23 build_conf="$cwd/build.conf"
24
25 # Sources top-level dir.
26 src_dir='.'
27
28 # Log file name.
29 log_file="$cwd/build.log"
30
31 # Packages build order.
32 packages=''
```

```
33
34 # Show configure/make output.
35 verbose='true'
36
37 # Selected action (all, clean, distclean, install or uninstall).
38 # Defaults to 'all'.
39 action='all'
40
41 # List of environmnet variables (eg. CC, CFLAGS...)
42 # passed to configure.  Specify them as VAR=VALUE.
43 conf_variables=''
44
45 # func_error STATUS MESSAGE
46 # Print a message to stderr.
47 # If STATUS is nonzero, terminate the program with 'exit STATUS'.
48 func_error()
49 {
50     local status=$1
51     local message=$2
52
53     echo "$PROGRAM_NAME: $message" 1>&2
54     if test $status -ne 0; then
55         exit $status
56     fi
57 }
58
59 # func_info() MESSAGE
60 # Print a message to stdout and sends a copy to log_file.
61 func_info()
62 {
63     local message="$1"
64     echo "$message" | tee -a "$log_file"
65 }
66
67 # func_exec COMMAND ARGS
68 # Execute COMMAND with arguments ARGS.
69 # The function exit status is the status of 'eval COMMAND ARGS'.
70 # If verbose == true, then command output is redirected to stdout/err,
71 # otherwise output is redirected to log_file.
72 func_exec()
73 {
74     local command="$1"
75     local args="$2"
76
77     if test $command = 'make'; then
78         args='CFLAGS="-g -O0" CXXFLAGS="-g -O0" '"$args"
79     fi
80
81     # Check if COMMAND exists.
82     which "$command" &> /dev/null
83     if test $? -ne 0; then
84         func_error 1 "command '$command' not found, aborting"
85     fi
86
87     if test "$verbose" = 'true'; then
88         eval "$command $args"
89     else
90         eval "$command $args" 1>> "$log_file" 2>&1
```

```
91      fi
92      return $?
93  }
94
95  # func_apply_action ACTION
96  # Iterates over all packages applying the specified ACTION.
97  # If successful return 0, otherwise return 1.
98  func_apply_action()
99  {
100     local action="$1"
101     local actioning=""
102     local actionlong=""
103
104     # Action long name.
105     if test "$action" = 'all'; then
106         actioning='Building'
107         actionlong='Build'
108     else
109         actioning="`echo ${action:0:1}|tr [:lower:] [:upper:]`${action
    :1}ing"
110         actionlong="`echo ${action:0:1}|tr [:lower:] [:upper:]`${action
    :1}"
111     fi
112
113     local i=0
114     while test $i -lt $array_size; do
115         local pkg="${pkg_array[$i]}"
116         local options="${conf_array[$i]}"
117
118         func_info "(($i+1))/$array_size) $actioning $pkg"
119         if ! test -d "$src_dir/$pkg"; then
120             func_error 0 "package \`$pkg' not found"
121             return 1
122         fi
123         cd "$src_dir/$pkg"
124
125         func_exec 'make' "$action"
126
127         # Action failed.
128         if test $? -ne 0; then
129
130             # If is `all' or `install', reconfigure and try again;
131             # otherwise skip package.
132
133             if test "$action" = 'all' -o "$action" = 'install'; then
134
135                 # Check for Makefile.
136                 if test -f './Makefile'; then
137                     func_info "$actionlong failed, maybe Makefile is
    corrupted"
138                     func_info 'I will run configure and try again'
139                     rm -f './Makefile'
140                 else
141                     func_info 'Makefile is missing, running configure'
142                 fi
143
144                 # Configure package.
145                 {
```

```
146                     local script=''
147
148                     # if test -x './configure'; then
149                     #     script='./configure';
150                     if test -x './autogen.sh'; then
151                         script='./autogen.sh'
152
153                     # autogen.sh exists but it is not executable;
154                     # try to toggle it executable.
155                     elif test -f './autogen.sh'; then
156                         chmod u+x './autogen.sh'
157                         if test $? -ne 0; then
158                             func_error 0 "can't force autogen.sh
    execution"
159                             return 1
160                         fi
161                         script='./autogen.sh'
162
163                     else
164                         func_error 0 "can't find configure script"
165                         return 1
166                     fi
167
168                     func_exec "$script" "$options $conf_variables"
169                     if test $? -ne 0; then
170                         func_error 0 'configure failed'
171                         return 1
172                     fi
173                 }
174     # changing libtool version
175     # echo "Making link to older libtool"
176     # rm -rf libtool
177     # ln -s $libtool_path libtool
178                 # Trying again.
179                 func_exec 'make' "$action"
180                 if test $? -ne 0; then
181                     func_error 0 "$actionlong failed"
182                     return 1
183                 fi
184             else
185                 func_info 'Makefile is missing, skipping'
186             fi
187         fi
188
189         cd "$cwd"
190         i=$(($i + 1))
191     done
192     return $?
193 }
194
195 # func_create_pkg_array PKG_STRING
196 # Create packages (and associated configuration) arrays.
197 # This function defines the following globals:
198 #   pkg_array[]    packages names in build order.
199 #   conf_array[]   packages configure parameters.
200 #   array_size     {pkg,conf}_array size.
201 func_create_pkg_array()
202 {
```

```
203     local pkg_string="$1"
204     local i=0
205     local s=""
206 
207     # Remove blanks.
208     pkg_string=`echo "$pkg_string" | tr '\n' ' '`
209     pkg_string=`echo "$pkg_string" | sed -e 's/[ ]*//g'`
210 
211     while test ${#pkg_string} -ne 0; do
212 
213         # Get first package in string.
214         s=`echo "$pkg_string" | sed -e 's/;.*//'`
215         pkg_string=${pkg_string:$((${#s} + 1))}
216 
217         # Split string into package name and configure options.
218         pkg_array[$i]=`echo "$s" | sed -e 's/--.*$//'`
219         s=${s:${#pkg_array[$i]}}
220         conf_array[$i]=`echo "$s" | sed -e 's/--/ --/g'`
221 
222         # Next step.
223         i=$(($i + 1))
224     done
225     array_size=$i
226 }
227 
228 # func_usage STATUS
229 # Print usage message and terminate the program with `exit STATUS'.
230 func_usage()
231 {
232     local status=$1
233 
234     if test $status -ne 0; then
235         echo "Try \`$program_name --help' for more information." 1>&2
236     else
237         echo "\
238 Usage: $program_name [ACTION] [OPTION]... [VAR=VALUE]...
239 Build and install source packages.
240 
241 Actions:
242   all                 build sources (default action)
243   clean               clean sources
244   distclean           clean sources and configuration info
245   maintainer-clean    clean all generated files
246   install             build and install sources
247   uninstall           uninstall sources
248 
249 Options:
250       --help              print this help and exit
251       --version           print version information and exit
252 
253   -c, --config=FILE       configuration FILE (defaults to \`build.conf')
254   -d, --src-dir=DIR       sources DIR (defaults to \`.')
255   -l, --log-file=FILE     log FILE name (defaults to \`build.log')
256   -p, --packages=<LIST>   semi-colon separated LIST of packages
257                             and configure parameters
258   -v, --verbose           show configure/make output
259 
260 To assign environment variables (e.g., CC, CFLAGS...),
```

```
261 specify them as VAR=VALUE.
262
263 Report bugs to <$BUGS_TO>."
264     fi
265     exit $status
266 }
267
268 # func_parse_args ARGV
269 # Parse program arguments ARGV.
270 # This function defines the following global:
271 #   action  selected action (all, clean, distclean, install or
    uninstall).
272 func_parse_args()
273 {
274     # Parse action.
275     case "$1" in
276         all | clean | distclean | maintainer-clean | install |
    uninstall)
277             action=$1
278             shift
279             ;;
280         *)
281             # Check if is an option or variable assignment.
282             if test -n "$1" \
283                 -a "${1:0:1}" != '-' \
284                 -a -z "`echo \"$1\"|grep '[^ ][^ ]*=[^ ]*'`"; then
285                 func_error 0 "unknown action \`$1'"
286                 func_usage 1
287             fi
288             action='all'
289             ;;
290     esac
291
292     # Parse options.
293     while test $# -gt 0; do
294         case "$1" in
295             --help)
296                 func_usage 0
297                 ;;
298
299             --version)
300                 echo "$version_string"
301                 exit 0
302                 ;;
303
304             -c | --config)
305                 shift
306                 build_conf="$1"
307                 if ! test -f "$build_conf"; then
308                     func_error 0 "$build_conf: no such file"
309                     func_usage 1
310                 fi
311                 # Path is relative, prefix cwd.
312                 if test ! "${build_conf:0:1}" = '/'; then
313                     build_conf="$cwd/$build_conf"
314                 fi
315                 shift
316                 ;;
```

```
317
318             -d | --src-dir)
319                 shift
320                 src_dir="$1"
321                 if ! test -d "$src_dir"; then
322                     func_error 0 "$src_dir: no such directory"
323                     func_usage 1
324                 fi
325                 if test -z "$src_dir" -o "${src_dir:0:1}" = '-'; then
326                     func_error 0 "option requires an argument -- '-d'"
327                     func_usage 1
328                 fi
329                 # Path is relative, prefix cwd.
330                 if test ! "${src_dir:0:1}" = '/'; then
331                     src_dir="$cwd/$src_dir"
332                 fi
333                 shift
334                 ;;
335
336             -l | --log-file)
337                 shift
338                 log_file="$1"
339                 if test -z "$log_file" -o "${log_file:0:1}" = '-'; then
340                     func_error 0 "option requires an argument -- '-l'"
341                     func_usage 1
342                 fi
343                 # Path is relative, prefix cwd.
344                 if test ! "${log_file:0:1}" = '/'; then
345                     log_file="$cwd/$log_file"
346                 fi
347                 shift
348                 ;;
349
350             -p | --packages)
351                 shift
352                 packages="$1"
353                 if test -z "$packages"; then
354                     func_error 0 "option requires an argument -- '-p'"
355                     func_usage 1
356                 fi
357                 shift
358                 ;;
359
360             -v | --verbose)
361                 verbose='true'
362                 shift
363                 ;;
364
365             --) # Stop option processing.
366                 shift; break
367                 ;;
368
369             -*)
370                 func_error 0 "unknown option \`$1'"
371                 func_usage 1
372                 ;;
373
374             *)
```

```
375                 # Check if is a variable assignment.
376                 if test -z "`echo \"$1\"|grep '[^ ][^ ]*=[^ ]*'`"; then
377                     func_error 0 "invalid variable assignment \`$1'"
378                     func_usage 1
379                 fi
380                 {
381                     local var=`echo $1 | sed 's/\([^ ][^ ]*\)=.*/\1/'`
382                     local value=`echo $1 | sed 's/.*=\(.*\)/\1/'`
383                     conf_variables="$conf_variables $var=\"$value\""
384                 }
385                 shift
386                 ;;
387         esac
388     done
389 }
390
391 func_main()
392 {
393     program_name="$0"
394
395
396     func_parse_args "$@"
397
398     # Load build.sh configuration.
399     source $build_conf
400
401     # Use absolute paths.
402     if test ! "${build_conf:0:1}" = '/'; then
403         build_conf="$cwd/$build_conf"
404     fi
405     if test ! "${log_file:0:1}" = '/'; then
406         log_file="$cwd/$log_file"
407     fi
408     if test ! "${src_dir:0:1}" = '/'; then
409         src_dir="$cwd/$src_dir"
410     fi
411
412     # Parse package list.
413     func_create_pkg_array "$packages"
414
415     # Update log_file.
416     echo `date` >> "$log_file"
417
418     # Apply action.
419     func_apply_action "$action"
420     if test $? -ne 0; then
421         if test "$verbose" = 'false'; then
422             echo "$PROGRAM_NAME: $log_file last lines:"
423             echo "========== >"
424             eval "tail $log_file"
425             echo "<=========="
426         fi
427         echo "Aborted." 1>&2
428         exit 1
429     fi
430
431     func_info "Finished"
432     exit 0
```

```
433 }
434
435 # Run main.
436 func_main "$@"
```

B.2 Arquivo: *build.conf*

Este arquivo contém a configuração utilizada durante a compilação e instalação do Ginga-NCL. A partir dele, a ordem com que os componentes do Ginga-NCL deve ser compilados é desoberta, bem como os parâmetros de configuração que foram utilizados inicialmente.

```
1  # Sources top-level dir.
2  src_dir=src
3
4  # Packages install order (with configure options, if any).
5  # Packages names must be separated by ';'.
6  packages="\
7    telemidia-util-cpp;
8    gingacc-cpp/gingacc-system;
9    gingacc-cpp/gingacc-cm;
10   gingacc-cpp/gingacc-mb --disable-sdl --enable-dfb --enable-
       xineprovider;
11   gingacc-cpp/gingacc-contextmanager;
12   gingacc-cpp/gingacc-ic;
13   gingacc-cpp/gingacc-um;
14   gingacc-cpp/gingacc-tuner;
15   gingacc-cpp/gingacc-tsparser;
16   gingacc-cpp/gingacc-dataprocessing;
17   gingacc-cpp/gingacc-multidevice;
18   gingacc-cpp/gingacc-player --disable-links --enable-berkelium;
19   ncl30-cpp/ncl30;
20   ncl30-cpp/ncl30-converter;
21   gingancl-cpp;
22   gingalssm-cpp --enable-tuner --enable-tsparser --enable-
       dataprocessing;
23   ginga-cpp;"
24
25 conf_variables='CFLAGS="-g -O0" CXXFLAGS="-g -O0"'
26 export LD_LIBRARY_PATH=`find /usr/local/lib/ginga -type d -print0 \
27   | xargs -i -0 echo -n "{}:"`/usr/local/lua/5.1/socket:/usr/lib/5.1/
       socket
```

B.3 Script: *sourceme*

A finalidade deste script é preparar o ambiente para a execução do Ginga logo após ele ter sido instalado.

Para utiliza-lo, deve-se entrar por terminal Linux o comando "*source sourceme*".

```
1 export LD_LIBRARY_PATH=`find /usr/local/lib/ginga -type d -print0 |
    xargs -i -0 echo -n "{}:"`/usr/local/lua/5.1/socket:/usr/lib/5.1/
    socket
```