UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

UMA PROPOSTA DE ARQUITETURA DE PILHA DE COMUNICAÇÃO EM REDE COM UM NÚMERO REDUZIDO DE CAMADAS

DISSERTAÇÃO DE MESTRADO

Josué Paulo José de Freitas

Santa Maria, RS, Brasil

2009

UMA PROPOSTA DE ARQUITETURA DE PILHA DE COMUNICAÇÃO EM REDE COM UM NÚMERO REDUZIDO DE CAMADAS

por

Josué Paulo José de Freitas

Dissertação apresentada ao Programa de Pós-Graduação em Informática,
da Universidade Federal de Santa Maria (UFSM, RS),
como requisito parcial para obtenção do grau de **Mestre em Computação**

Orientador: Prof. João Baptista dos Santos Martins

Santa Maria, RS, Brasil

2009

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação

UMA PROPOSTA DE ARQUITETURA DE PILHA DE COMUNICAÇÃO EM REDE COM UM NÚMERO REDUZIDO DE CAMADAS

elaborada por
Josué Paulo José de Freitas

como requisito parcial para obtenção do grau de
Mestre em Computação

COMISSÃO EXAMINADORA:

_____
João Baptista dos Santos Martins, Dr (UFSM)
(Presidente/Orientador)

_____
Rafael Ramos dos Santos, Dr. (UNISC)
(Examinador)

_____
Raul Ceretta Nunes, Dr. (UFSM)
(Examinador)

Santa Maria, 22 de Agosto de 2009

Dedico este trabalho aos meus pais, Aracy A. de Freitas e Argemiro José de Freitas, e a minha namorada Débora Franke por toda compreensão e carinho durante esta etapa.

AGRADECIMENTOS

Agradeço aos meus pais, Aracy A. de Freitas e Argemiro José de Freitas, pelo apoio, suporte e carinho sem os quais esta jornada n ao teria sido possível.

Agradeço aos meus colegas da Santa Maria Design House, Gustavo Fernando Dessbessel, Douglas Camargo Foster, João Eduardo Taffarel, Rafael Tambara e César Augusto Prior pelo companheirismo. Cabe neste momento um agradecimento especial a Gustavo Fernando Dessbessel por todas as conversas e colaborações técnicas que foram de imensa valia.

Agradeço também a todos meus colegas do Grupo de Microeletrônica da Universidade Federal de Santa Maria (Gmicro). Pela amizade e parceria destaco todos aqueles que trabalharam comigo de alguma forma nestes anos, são eles, Leandro Roberto Sehn, Fernando Luis Hermann, Paulo César Comassetto de Aguirre, Lucas Teixeira, Cristian Müller, Edinei Santin, Guilherme Perin, Sidinei Ghissoni, Leonardo Guedes da Luz Martins, Rodrigo Flores, Felipe Woulthers e Rafael Bertagnolli.

Agradeço ao meu orientador, João Baptista dos Santos Martins, pela amizade, orientação e por todas as oportunidades durante estes anos de Gmicro.

Agradeço também a Marinelma Aimi de Carvalho, secretária da Pós-graduação em Informática da UFSM, por sua competência e colaboração durante este período.

Agradeço também os professores Rafael Ramos do Santos e Raul Ceretta Nunes por seus comentários extremamente valiosos durante a a banca de avaliação desta dissertação.

O meu muito obrigado a todos que contribuíram de alguma forma com este trabalho. Dentre estes destaco Yan-Shun Li e Parimal Patel, da Xilinx, Bernard Aboba, da Microsoft, professores César Ramos Rodrigues e Giovani Baratto, do Gmicro, e Friedrich Nietzsche.

Por fim, mas não menos importante, agradeço a Débora Franke, minha namorada, por ser fonte de amor, conforto e inspiração tornando assim esta caminhada menos árdua.

Há tantas auroras que não brilharam ainda.

Rigveda

RESUMO

Dissertação
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria, RS, Brasil

UMA PROPOSTA DE ARQUITETURA DE PILHA DE COMUNICAÇÃO EM REDE COM UM NÚMERO REDUZIDO DE CAMADAS

AUTOR: JOSUÉ PAULO JOSÉ DE FREITAS
ORIENTADOR: JOÃO BAPTISTA DOS SANTOS MARTINS

Local da Defesa e Data: Santa Maria, 22 de Agosto de 2009.

Este trabalho apresenta uma proposta arquitetura de pilha de comunicação em rede com número reduzido de camadas. A redução do número de camadas visa fornecer um método de comunicação simples e eficaz para sistemas embarcados permitindo que o microprocessador, onde geralmente a Camada de Aplicação é implementada, execute apenas código de aplicação isentando-se assim de tarefas de comunicação em rede. A arquitetura foi implementada em placa de desenvolvimento FPGA e apresentou, em média, vazão cerca de 27 vezes superior em comparação com uma pilha de comunicação implementada em *software* e executada sobre um microprocessador embarcado.

Palavras-chave: Pilha de comunicação em rede, Rede de computadores, Latência, Vazão, FPGA

ABSTRACT

Master's Dissertation
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria, RS, Brasil

A NOVELL NETWORK STACK ARCHITECTURE WITH REDUCED NUMBER OF LAYERS

AUTHOR: JOSUÉ PAULO JOSÉ DE FREITAS
ADVISOR: JOÃO BAPTISTA DOS SANTOS MARTINS

Place and Date: Santa Maria, August 22th, 2009.

This work presents a network stack architecture proposal with a reduced number of layers. The reduction in number of layers aim to provided a simpler and efficient communication method to embedded systems by allowing the microprocessor, where usually application is implemented, run just application code and not running code related to network communication. The architetucture was implemented on and FPGA board and show, in average, throughput results around 27 times better in comparision with a network stack implemented in software and running over an embedded microprocessor.

Palavras-chave: Network Stack, Computer Network, Latency, Throughput, FPGA

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

API	Application Program Interface
ARP	Address Resolution Protocol
ASIC	Application Specific Integrated Circuit
Bcopy	Buffer Copy
BIP	Basic Interface for Parallelism
BRAM	Block Random Access Memory
CPU	Central Processor Unit
EDK	Embedded Development Kit
EMP	Ethernet Message Passing
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GCC	Gnu C Compiler
HCA	Host Channel Adapter
HDL	Hardware Description Language
HT	HyperThreading
IP core	Intelectual Property core
IP	Internet Protocol
lwIP	Lightweight IP
MAC	Medium Access Control
MTU	Maximum Tranference Unit
NIC	Network Interface Card
PC	Personal Computer
PCI	Peripheral Component Interconnect
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
RTAI	Real Time Application Interface
SATA	Serial Advanced Technology Attachment
SMP	Symmetric MultiProcessor
SDP	Sockets Direct Protocol

TCP	Transport Control Protocol
UDP	User Datagram Protocol
ULP	Upper Layer Protocol
VHDL	VHSIC Hardware Description Language
ZCopy	ZeroCopy

LISTA DE LISTAGENS

SUMÁRIO

1 INTRODUÇÃO

A largura de banda das redes de computadores tem aumentado rapidamente nos últimos anos. No ano 2000 mais informação podia ser enviada através de um único cabo em um segundo do que era enviado em toda a Internet durante um mês no ano de 1997 (GILDER, 2001).

Estudos demonstram que o *software* dos sistemas operacionais, onde a pilha de protocolos é implementada, não consegue lidar de maneira otimizada com uma alta vazão de dados, gastando assim vários ciclos de processamento codificando e decodificando informações relativas aos protocolos de comunicação (CLARK et al., 1989) (CLARK et al., 2002). Uma pilha TCP/IP tradicional, não otimizada, sendo executada sobre um processador de propósito geral apresenta uma relação de aproximadamente 1 MHz de frequência de operação para 1 Mb/s de vazão *full-duplex* (SMITH, 2002). Em (ROMANOW et al., 2005) é apresentado um caso onde, no ano de 2001, um processador Athlon de 1,2 GHz com 100% de utilização poderia processar uma vazão full-duplex máxima de 2,7 Gb/s, ou seja, restariam poucos ciclos de processamento para aplicações de usuário considerando a tecnologia Gigabit Ethernet já disponível na época.

A Lei de Moore afirma que o número de transistores em um processador dobra a cada 18 meses em média. Este fato proporcionou nos últimos anos que a frequência de operação também dobrasse. Por outro lado a Lei de Gilder afirma que a largura de banda disponível dobra a cada 6 meses (GILDER, 2001). Mesmo que a Lei de Gilder não tenha se confirmado com a mesma exatidão da Lei de Moore é possível vislumbrar o surgimento de um gargalo entre a largura de banda disponível e a capacidade dos processadores de propósito geral

Esta relação entre frequência de processamento e vazão não representou um problema até o

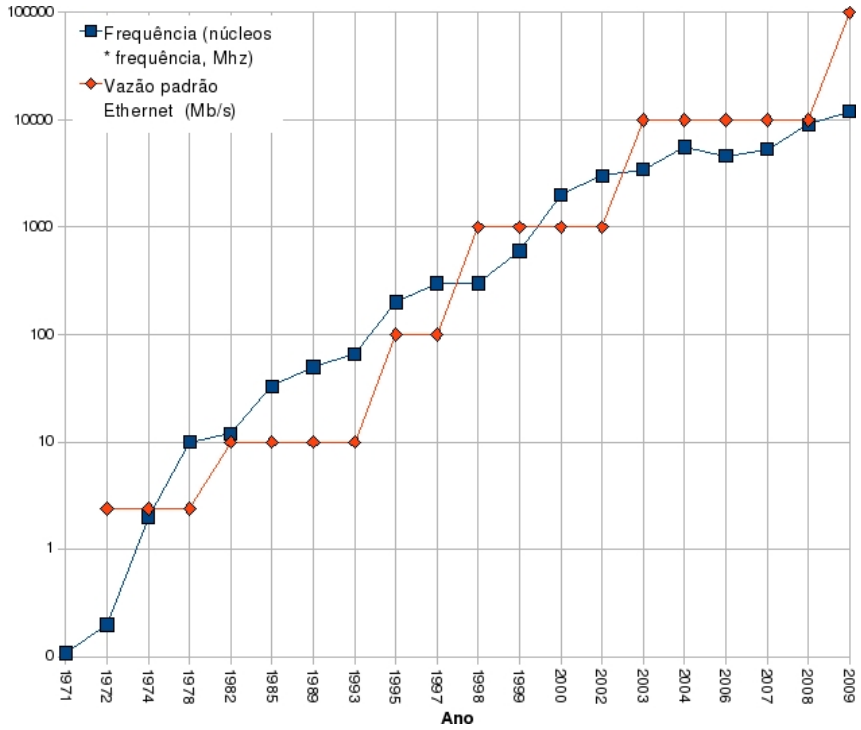

Figura 1.1: Vazão versus frequência dos processadores de propósito geral

ano de 1998 com o surgimento da tecnologia Gigabit Ethernet. A Figura 1.1[1] mostra um gráfico com os valores de frequência e vazão ao longo dos anos. O número de núcleos foi multiplicado pela frequência uma vez que cada núcleo possui capacidade de processamento e poderia assim executar tarefas relacionadas à comunicacação em rede.

Atualmente encontra-se em fase experimental o padrão Ethernet 802.3ba que permitirá uma vazão de 100 Gb/s em cada sentido o que representa um significativo gargalo considerando os processadores de propósito geral disponíveis neste ano.

Considerando que sistemas embarcados normalmente são dotados de microprocessadores com recursos limitados e baixa frequência, a relação frequência de operação e vazão torna-se ainda mais expressiva. Desta maneira, é possível notar um gargalo expressivo quando sistemas embarcados necessitam de uma maior vazão de dados em rede e utilizam pilhas de comunicação embarcadas para este fim.

[1] O gráfico foi concebido tendo como base a lista de processadores Intel lançados ao longo dos anos disponível em http://www.intel.com/pressroom/kits/quickreffam.htm, as datas de lançamento dos diferentes padrões Ethernet 802.3 e projeções disponíveis em http://www.ieee802.org/

1.1 Organização

A dissertação encontra-se organizada da seguinte maneira: o Capítulo 1 realiza uma introdução ao tema juntamente com Objetivos deste trabalho além de Justificativa para o mesmo. O Capítulo 2 apresenta uma revisão bibliográfica da área de redes com enfoque nas tecnologias e projetos de pesquisas relacionados com este trabalho. O Capítulo 3 apresenta o desenvolvimento da arquitetura proposta e o Capítulo 4 apresenta os resultados obtidos. Por fim o Capítulo 5 apresenta as conclusões e trabalhos futuros.

1.2 Objetivos

O principal objetivo deste trabalho é desenvolver uma nova arquitetura de pilha de comunicação e implementá-la em FPGA com um número reduzido de camadas e, desta maneira, prover maior vazão de dados para a Camada de Aplicação em sistemas embarcados.

Também é objetivo deste trabalho desenvolver uma camada de integração que atue entre um MAC (*Medium Access Layer*) Gigabit Ethernet e uma Camada de Aplicação. Esta camada de integração deve prover algumas funcionalidades de gerenciamento de conexão. Esta camada de integração recebe *Raw Ethernet Frames* diretamente da camada inferior, doravante denominados quadros Ethernet puros. Estes quadros possuem informações relativas apenas ao protocolo Ethernet de modo que quando utilizados não existem informações relativas às demais camadas ou protocolos como na suíte de protocolos TCP/IP. Este fato, apesar de tornar inviável sua utilização fora de redes locais, permite um custo computacional reduzido nos processos de encapsulamento e desencapsulamento uma vez que todas as verificações dos protocolos IP e TCP/UDP não são necessárias.

A Figura 1.2 mostra um quadro Ethernet tradicional (a), contendo dados de cabeçalho dos demais protocolos da pilha de comunicação tradicional, e um quadro Ethernet puro (b). Observando a Figura 1.2, é possível notar que a utilização de quadros Ethernet puros disponibiliza uma quantidade levemente maior de dados para Camada de Aplicação, o que pode ser relevante

conforme a necessidade da Camada de Aplicação. A garantia de integridade dos dados é provida por um cálculo CRC (*Cyclic Redudancy Check*) localizado no final do quadro (*Ethernet trailer*).

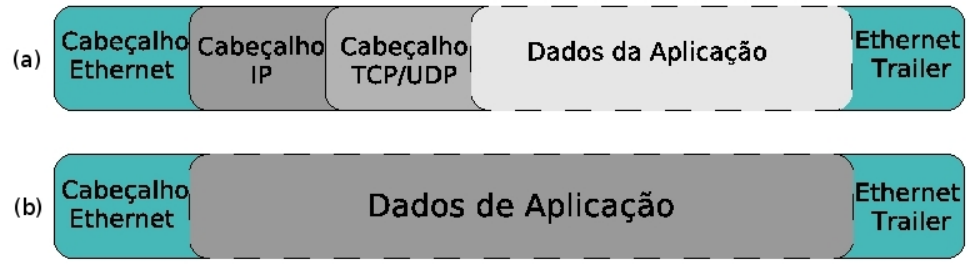

Figura 1.2: Quadro Ethernet tradicional contendo outros protocolos encapsulados (a) em comparação com um quadro Ethernet puro (b)

Por fim, também é objetivo deste trabalho realizar medições de latência e vazão na arquitetura proposta bem como em outras arquiteturas de pilha de comunicação com objetivo de comparação de resultados.

1.3 Justificativa

Com o objetivo de reduzir a carga do processamento de informações de rede sobre o processador, pilhas de comunicação em *software* fazem uso de alternativas que utilizam *hardware* dedicado para as tarefas relativas aos protocolos de rede. Desta maneira o processador pode utilizar mais ciclos de processamento com aplicações de usuários e menos ciclos com o processamento de protocolos de rede. As Seções 2.2.1 e 2.2.2 apresentam projetos de pesquisa e tecnologias que fazem uso de *hardware* dedicado com este objetivo.

Neste contexto, o processo de cópia de dados foi identificado como um significativo consumidor de ciclos de CPU (CLARK et al., 1989) (CLARK et al., 2002) (SKEVIK et al., 2001). Juntamente, o uso de técnicas de *ZeroCopy* (BROSE, 2005) e *One-Copy*, que visam otimizações na transferência de dados do usuário para o dispositivo de rede, tornaram-se uma alternativa. Assim, teoricamente, quanto menor o número de vezes em que um dado precisa ser copiado em *software* melhor será o desempenho da pilha de comunicação.

A utilização da técnica de ZeroCopy tem como objetivo não realizar cópia de dados na memória do nodo de rede de modo que os dados são transferidos, em determinados casos, diretamente através da utilização de RDMA (*Remote Direct Memory Access*)(GOLDENBERG et al., 2005). A técnica *One-Copy* tem como objetivo que uma única cópia de dados seja realizada dentro do nodo de rede. Durante a cópia dos dados a serem transferidos o cálculo do *checksum* é realizado.

Em (SARDA, 2003) o autor realizou modificações para que a biblioteca MPI (*Message Passing Interface*) utilizasse quadros Ethernet puros ao invés de TCP/IP. Esta modificação apresentou um significativo aumento de desempenho como é possível visualizar nos gráficos de vazão e latência nas Figuras 1.3 e 1.4, respectivamente.

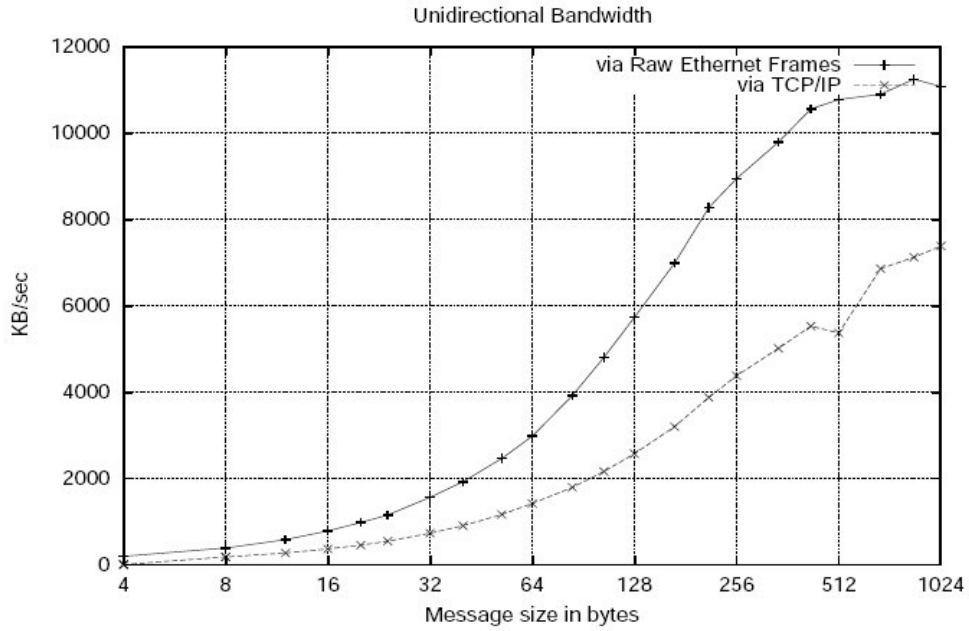

Figura 1.3: Vazão de quadros Ethernet puros versus TCP/IP (SARDA, 2003)

Como é possível observar na Figura 1.3, o ganho de vazão foi, para o melhor caso (mensagens com 512 bytes), em torno de 100%. Para mensagens pequenas (< 32 bytes) o ganho não foi muito significativo. Os resultados de latência mantiveram-se em torno de 50% de redução na maioria dos casos, apresentando melhor desempenho para mensagens de 512 bytes onde a redução foi em torno de 85%.

Em (VALENTIM; JUNIOR; OLIVEIRA, 2007) a utilização de quadros Ethernet puros

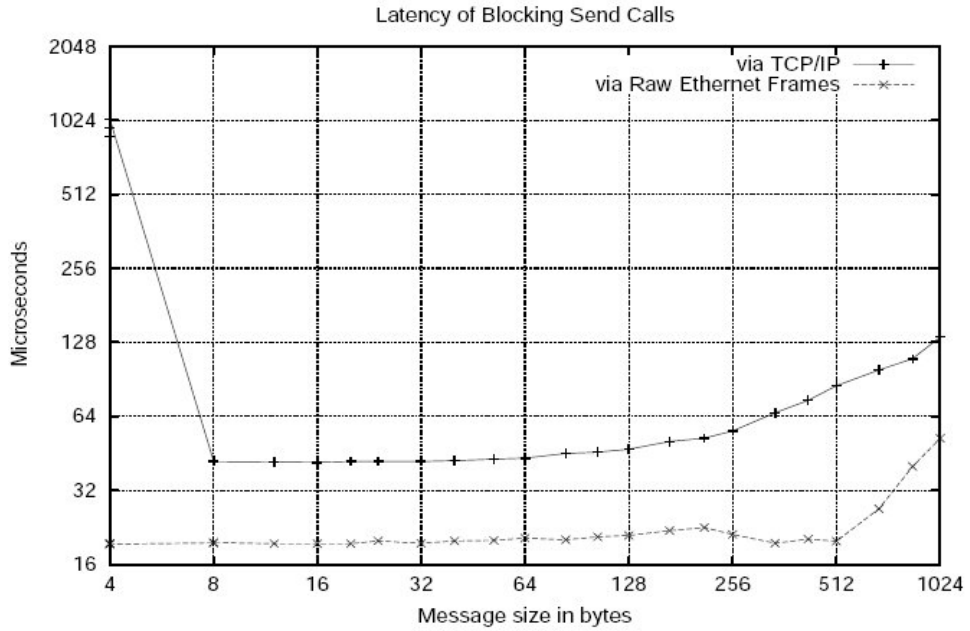

Figura 1.4: Latência quadros Ethernet puros versus TCP/IP (SARDA, 2003)

reduziu em 55% o tempo de envio de dados em comparação com UDP/IP.

Considerando o acima exposto as principais justificativas para este trabalho são a utilização de hardware dedicado para a realização de gerenciamento de conexões e a utilização de quadros Ethernet puros. Além disso, sistemas embarcados, de uma maneira geral, possuem microprocessadores com menos recursos e assim pilhas de comunicação desenvolvidas em software podem reduzir significativamente o desempenho do sistema, muitas vezes apresentando desempenho muito inferior a capacidade do *hardware* da camada física.

2 REVISÃO BIBLIOGRÁFICA

2.1 Camadas de comunicação em rede

Com o objetivo de reduzir a complexidade, a comunicação em rede foi dividida em camadas. Estas camadas quando empilhadas formam uma suíte de protocolos de comunicação, ou, em outras palavras, uma pilha de protocolos. Cada camada de uma pilha de comunicação implementa um protocolo que por sua vez é responsável por determinadas tarefas do processo de comunicação em rede, abstraindo os detalhes destas tarefas às camadas superiores.

A pilha de protocolos TCP/IP é o padrão *de facto* para comunicação na Internet. Cada camada da pilha TCP/IP é responsável por encapsular e desencapsular os dados de saída e entrada, respectivamente. Entende-se por encapsular, a adição de informações dos protocolos em questão, comumente um cabeçalho contendo estas informações, seguida pelo repasse destes dados encapsulados para a camada inferior. O desencapsulamento consiste no processo inverso, onde as informações do protocolo são removidas e os dados restantes repassados à camada superior. A Figura 2.1 ilustra a pilha TCP e o processo de encapsulamento e desencapsulamento.

O processo de cópia de dados consiste num grande consumidor de ciclos de CPU em uma pilha TCP/IP não otimizada (CLARK et al., 1989) (CLARK et al., 2002). Assim, muitas vezes recursos como *Remote Direct Memory Access* (RDMA) (RECIO, 2007) são utilizados em pilhas de comunicação para evitar o excessivo número de cópias através da memória. Juntamente, pilhas de comunicação alternativas à TCP/IP têm surgido, seja para comunicação otimizada em redes locais voltadas a agredados de computadores (GOLDENBERG et al., 2005) (MYRICOM, 2007) (MELAS; ZALUSKA, 1998), seja em implementações parciais ou completas da pilha

TCP/IP em *hardware* dedicado (HAMERSKI, 2008) (LU, 2003) (LOFGREN, 2004) ou em microprocessadores dedicados a este fim (SHIVAM; WYCKOFF; PANDA, 2001) (SHAFER; RIXNER, 2006).

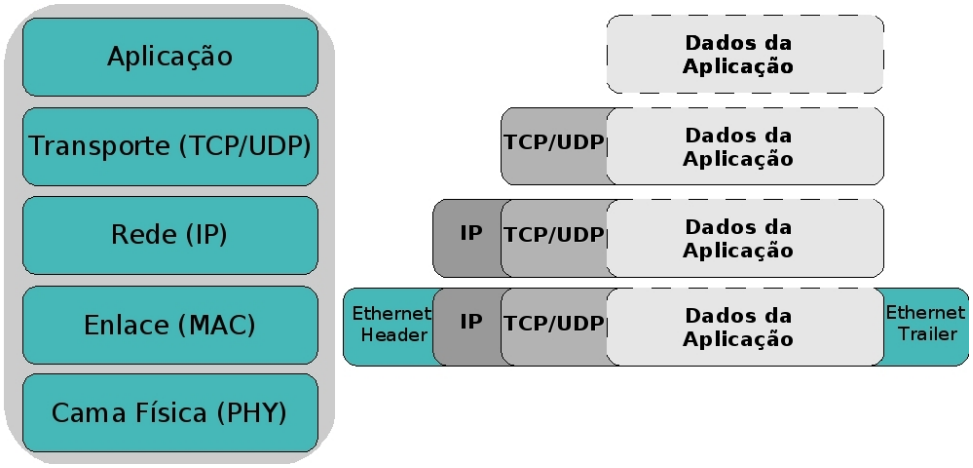

Figura 2.1: Pilha de comunicação TCP/IP e o processo de encapsulamento/desencapsulamento

2.2 Recursos de *hardware* dedicado e redes de computadores

Além da camada física, também conhecida como PHY (*PHYsical*), algumas tecnologias utilizam mais recursos de *hardware* com o objetivo de aumentar seu desempenho. Na tecnologia Gigabit Ethernet o PHY recebe o sinal através do cabo e o converte para um sinal digital onde o barramento de dados possui 8 bits que são fornecidos a cada ciclo de 125 MHz para a camada acima (Camada de Enlace, MAC). No caminho inverso, o PHY recebe 8 bits da Camada de Enlace a cada ciclo de 125 MHz e os envia no formato analógico pelo cabo de rede.

2.2.1 Projetos Acadêmicos

Em (SHIVAM; WYCKOFF; PANDA, 2001) é apresentado o projeto EMP (*Ethernet Message Passing*) onde foi desenvolvida uma solução para Gigabit Ethernet sem cópia através da memória. Este trabalho faz uso de placas de rede Gigabit Ethernet Alteon dotadas de um microprocessador programável onde foi implementado o código responsável pelos protocolos de comunicação, deixando o processador do computador livre para processamento da Camada de Aplicação. Este projeto, apresentou uma vazão máxima de 880 Mb/s e uma latência de 23 mi-

crossegundos. Comparativamente, a menor latência mensurada para TCP/IP foi superior a 100 microssegundos e a vazão não passou de aproximadamente 600 Mb/s. Também em (SHIVAM; WYCKOFF; PANDA, 2001) são apresentados resultados significativos do comportamento da vazão em relação a utilização do processador principal por programas de usuário. Dentre as três alternativas avaliadas EMP, TCP/IP e GM (*Myrinet Message Passing*) o EMP apresentou a menor queda na vazão em função do processamento requerido por programas de usuário. Uma comparação entre a arquitetura EMP e uma arquitetura tradicional encontra-se ilustrada na Figura 2.2.

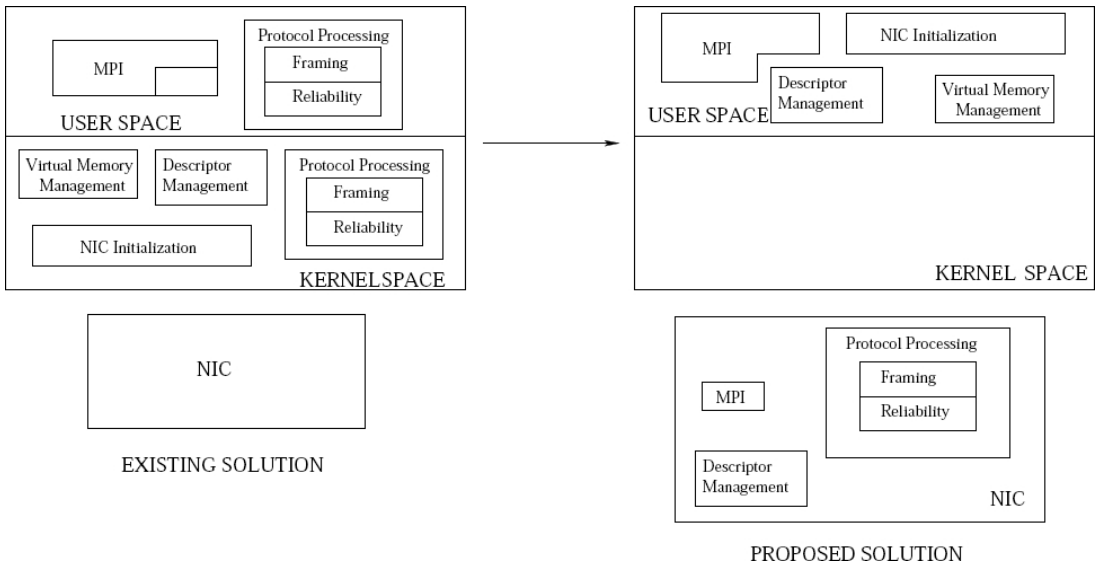

Figura 2.2: Arquitetura existente e a arquitetura proposta no projeto EMP (SHIVAM; WYCKOFF; PANDA, 2001)

No projeto RiceNIC (*Rice Network Interface Card*) (SHAFER; RIXNER, 2006) os autores utilizaram uma placa PCI dotada de dois FPGAs, um VirtexII Pro e um SpartanIIE. O FPGA VirtexII Pro foi responsável pela Camada de Enlace fazendo uso de um núcleo Gigabit da própria Xilinx, enquanto o FPGA SpartanIIE foi programado com um núcleo responsável pela comunicação PCI com o barramento do computador. As camadas superiores ao enlace foram implementadas em *software* rodando sobre o processador PowerPC disponível como um *hard-core* dentro do FPGA VirtexII Pro. Juntamente, também foram desenvolvidos os *drivers* do sistema operacional para que este pudesse utilizar os novos recursos. Utilizando tamanho de pacotes de 960 bytes este projeto aproximou-se da vazão máxima teórica para Gigabit Ethernet. Além disso, a utilização do processador principal do computador pessoal utilizado restringiu-se

ao envio de dados ao barramento PCI, permitindo que o processador ficasse livre para processos do usuário. A arquitetura do projeto RiceNIC encontra-se ilustrada na Figura 2.3.

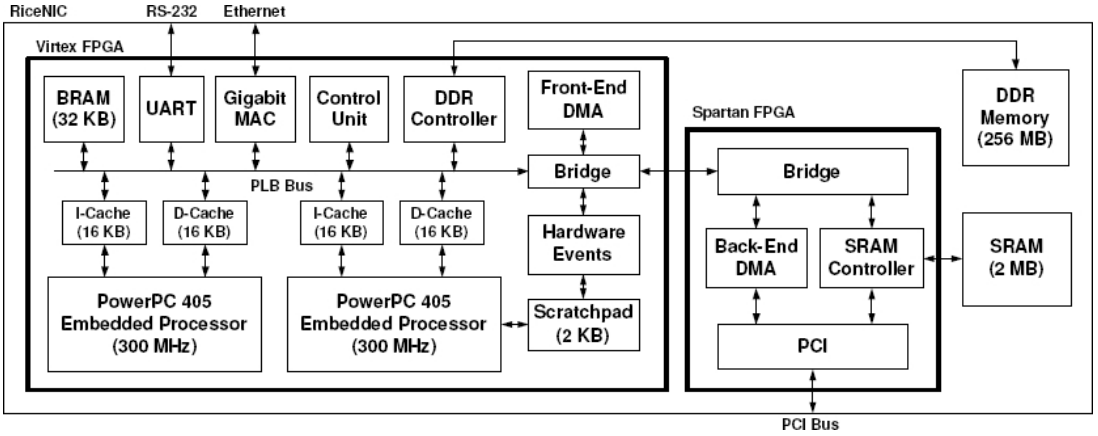

Figura 2.3: Arquitetura do projeto RiceNIC (SHAFER; RIXNER, 2006)

O projeto *Basic Interface for Parallelism* (BIP) apresentado em (MELAS; ZALUSKA, 1998) consiste em uma implementação sobre a tecnologia Myrinet. Este projeto provê recursos de ZeroCopy em nível de usuário. As latências medidas neste projeto são da ordem de poucos microsegundos e a vazão máxima gira em torno de 1008 Mb/s. A Figura 2.4 ilustra a arquitetura do projeto BIP e uma pilha de comunicação tradicional. Na Figura 2.4 (a) é possível observar que interface BIP realiza uma comunicação direta entre o dispositivo Myrinet e a Camada de Aplicação. Esta comunicação direta é o motivo de seu desempenho otimizado em comparação com uma pilha de comunicação tradicional uma vez que o caminho de dados é encurtado significativamente.

2.2.2 Estado da arte

Além destes projetos de pesquisa, as tecnologias Infiniband (GOLDENBERG et al., 2005) e Myrinet (BODEN et al., 1995) destacam-se no mercado de dispositivos de rede de alto desempenho utilizando recursos de ZeroCopy, One-copy e *hardware* dedicado.

Myrinet

A tecnologia Myrinet tem como objetivo realizar interconexão de rede em agregados de computadores (*clusters*). Dentre suas características destaca-se a baixa latência adquirida atra-

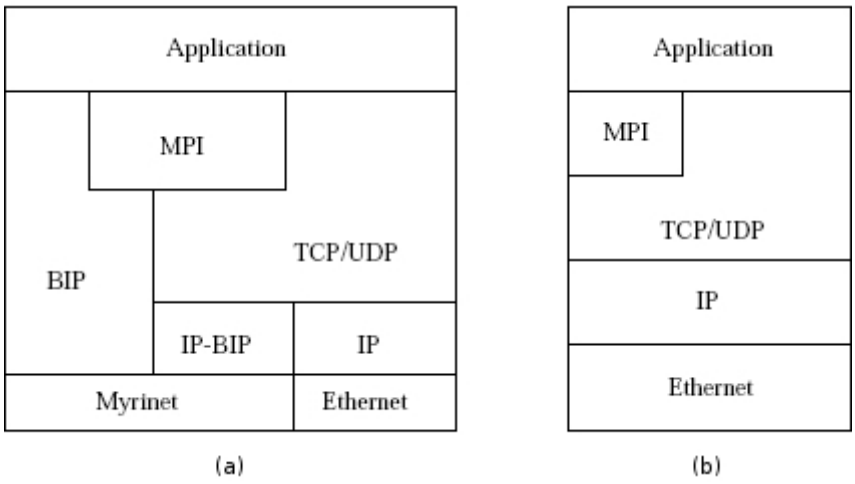

Figura 2.4: Comparação entre a pilha de comunicação do projeto BIP (a) e uma pilha de comunicação tradicional (b) (MELAS; ZALUSKA, 1998)

vés da capacidade de transpassar o sistema operacional (*OS bypass*). As placas de rede Myrinet executam um *firmware* com objetivo de isentar o sistema operacional da função de processar dados relativos aos protocolos de rede (*offload*).

Além disso, esta tecnologia também otimiza o processo de transferência de dados do usuário para a placa de rede. O diagrama mostrado na Figura 2.5 (BODEN et al., 1995) compara três tipos de transferências de dados da memória de usuário para o dispositivo de rede.

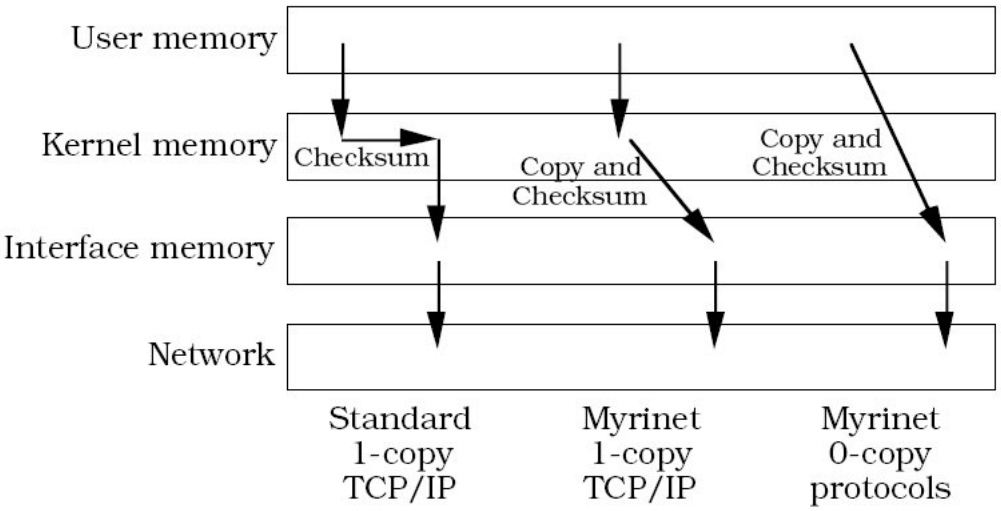

Figura 2.5: Diagrama de cópia de dados da memória de usuário ao dispositivos de rede (BODEN et al., 1995)

No primeiro modo, denominado “Standard 1-copy TCP/IP”, o sistema operacional realiza a cópia do bloco de dados da memória de usuário para a memória de *kernel* onde o checksum

é calculado e, por fim, os dados são enviados à interface com o dispositivo de rede. No segundo caso, denominado "Myrinet 1-copy TCP/IP", a cópia de dados ocorre com uso de *Direct Memory Access* (DMA) e o *checksum*, com o devido suporte do sistema operacional, é calculado pelo dispositivo Myrinet, deste modo reduzindo significativamente a carga do processador. Por fim, o terceiro caso apresenta a utilização de ZeroCopy da API Myrinet onde os dados são repassados diretamente da memória do usuário para a interface do dispositivo de rede Myrinet.

Como é possível observar na Tabela 2.1[1] a API Myrinet apresentou os melhores resultados em comparação com outras soluções para a arquitetura SPARC-2. Isto demonstra que um número reduzido de camadas e uma comunicação mais direta entre o nível aplicação e o dispositivo de rede produzem ganhos significativos de desempenho.

Tabela 2.1: Desempenho da API Myrinet em comparação com outras soluções de transferência de dados para a arquitetura SPARC-2

Método	Vazão máxima
Vazão máxima do barramento	640 Mb/s
Vazão através de DMA	444 Mb/s
Teste de transferência pura	380 Mb/s
API Myrinet	250 Mb/s
TCP/IP com ZeroCopy e checksum em hardware	70 Mb/s
UDP/IP com OneCopy e checksum em hardware	55 Mb/s

Atualmente a tecnologia Myrinet apresenta uma solução denominada Sockets-MX (MYRICOM, 2007), que, como a API original de (BODEN et al., 1995), utiliza ZeroCopy, aliando a isto transposição do *kernel* (*Kernel bypass*) e transferências entre nodos através de RDMA. A Figura 2.6 ilustra o conceito de Sockets-MX em comparação com o modelo TCP/IP tradicional.

Infiniband

A tecnologia Infiniband, assim como a Myrinet, também consiste numa solução de interconexão de rede voltada a agregados de computadores. A Infiniband apresenta baixa latência, alta vazão de dados e também realiza transferência de dados sem cópia através da memória.

A implementação sem cópia através da memória, denominada ZeroCopy, é realizada atra-

[1]Teste de transferência pura: *Raw speed test* no original de (BODEN et al., 1995)

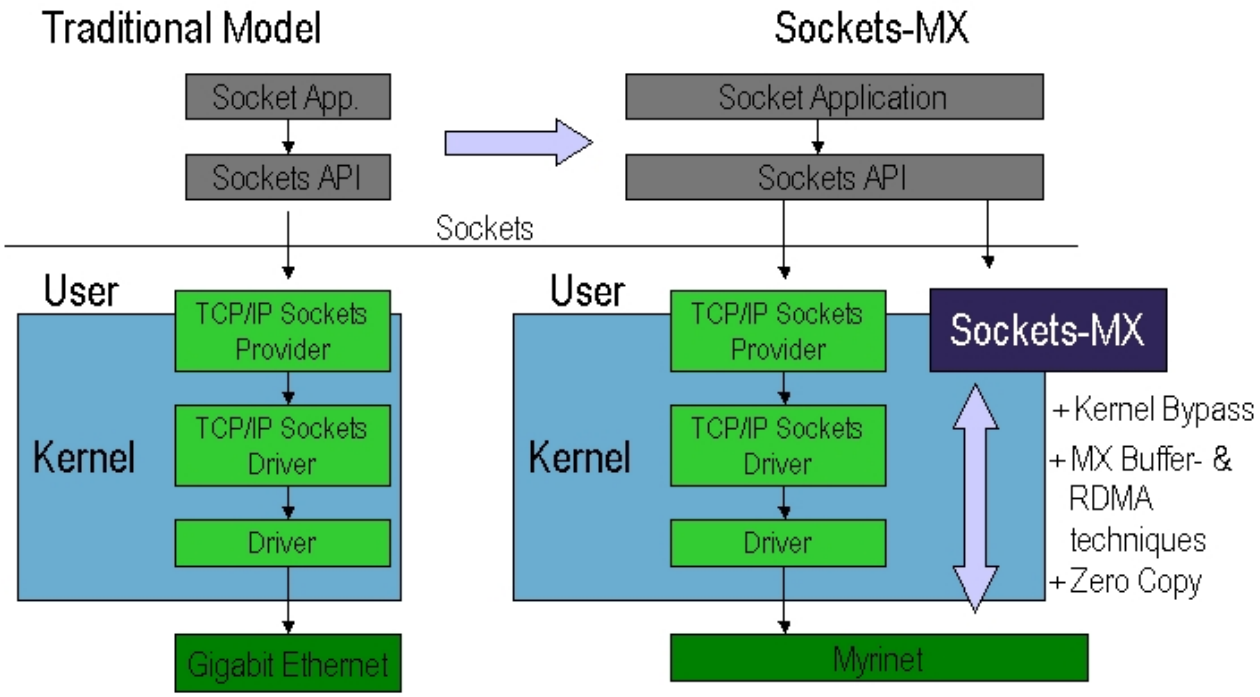

Figura 2.6: Conceito de Socktex-MX em comparação com o modelo TCP/IP tradicional (MYRICOM, 2007)

vés de RDMA. Com este recurso é possível que os nodos de rede realizem transferências de dados com o mínimo de utilização do processador, evitando a cópia para *buffers* intermediários. O recurso de transposição do sistema operacional permite que aplicações de usuário interajam diretamente com o *hardware* do dispotivo Infiniband para envio e recebimento de mensagens, diminuindo assim a sobrecarga de comunicação no caminho dos dados da aplicação até o dispositivo de rede.

A tecnologia Infiniband faz uso do protocolo *Sockets Direct Protocol* (SDP) (INFINIBAND, 2002) para a utilização dos recursos de ZeroCopy e de transposição do sistema operacional. O SDP é implementado como um módulo de *kernel* juntamente com uma opção de tipo de soquete em uma biblioteca em modo de usuário (libsdp) (GOLDENBERG et al., 2005), conforme ilustra a Figura 2.7.

Como é mostrado na Figura 2.7 a comunicação do protocolo SDP com a aplicação é feita através de uma biblioteca (*SDP sockets pre-load Library*) que interage diretamente com o *driver* do protocolo SDP, transpassando assim tanto biblioteca padrão do sistema operacional (Libc) quanto a pilha TCP/IP. Uma vez que o *hardware* Infiniband já possui transporte confiável, ou

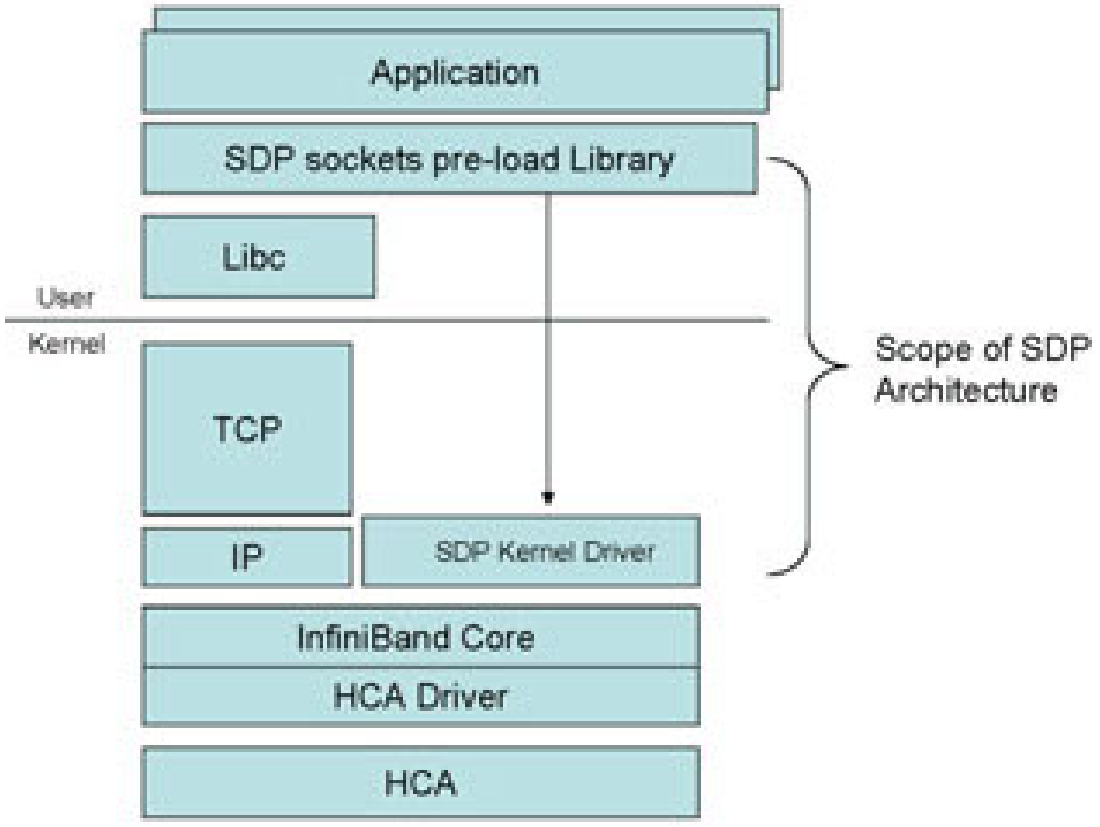

Figura 2.7: Componentes de uma pilha de comunicação TCP/IP e Inifiniband SDP (WOODRUFF, 2008)

seja, com garantia de entrega, é possível transpassar o TCP/IP sem perda de dados durante o processo de comunicação.

O protocolo SDP utiliza a família de endereçamento AF_SDP diferente da tradicional AF_INET utilizada para conexões convencionais. Desta maneira é possível fazer uso do protocolo SDP sem que seja necessário alterar as aplicações já existentes que fazem uso de soquetes. Assim, toda vez que uma aplicação utilizar AF_SDP ao invés de AF_NET para a família de endereçamento a biblioteca *SDP sockets pre-load Library* irá tratar a conexão ao invés da biblioteca Libc (WOODRUFF, 2008). Apenas o tipo de soquete SOCK_STREAM é suportado quando utilizado com o SDP.

Além disso, o SDP também possui a capacidade de utilizar a área de portas do TCP e endereçamento IP (IPv4 ou IPv6), características que facilitam o desenvolvimento de aplicações.

A interface do SDP com os protocolos de nível superior (ULP, *Upper Layer Protocol*)

encontra-se imediatamente acima da camada de transporte que implementa a entrega confiável da arquitetura Infiniband. O mapeamento do SDP para o protocolo de transporte da arquitetura Infiniband foi desenvolvido de modo a permitir que os dados sejam transmitidos ao ULP de uma das duas maneiras: pela utilização de *buffers* privados (Bcopy, *Buffer Copy*) ou diretamente entre os *buffers* ULP (Zcopy, Zero Copy) (INFINIBAND, 2002).

A utilização de RDMA no processo de transferência ZeroCopy permite que sejam feitos acessos remotos à memória de uma estação sem que o processador desta estação seja interrompido. Deste modo os dados são copiados diretamente do *buffer* do protocolo de aplicação (ULP *buffer*) de origem para o ULP *buffer* de destino. A Figura 2.8 ilustra o processo de transmissão de dados ZeroCopy através de RDMA, como é possível observar a estação de origem (*Data Source*) envia uma mensagem *SrcAvail* e a estação de destino (*Data Sink*) efetua a leitura através de RDMA. Após efetuar a leitura de todo o buffer RDMA indicado em *SrcAvail*, a estação de destino envia a mensagem *RdmaRdCompl* indicando que a transmissão foi completada.

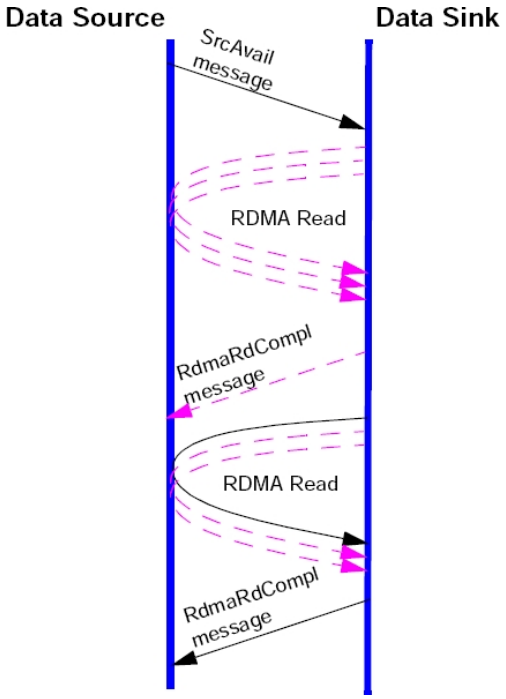

Figura 2.8: Transmissão de dados ZeroCopy com utilização de RDMA. As setas pontilhadas representam ações opcionais. (INFINIBAND, 2002)

Pelo processo Bcopy os dados são copiados para os *buffers* privados de recebimento e então repassados para os *buffers* do protocolo de aplicação (*ULP buffers*), uma mensagem de controle

de fluxo é enviada periodicamente uma vez que existe um número limitado de *buffers* privados de recebimento. A Figura 2.9 ilustra o processo de transmissão de dados através de Bcopy e o envio da mensagem de controle de fluxo.

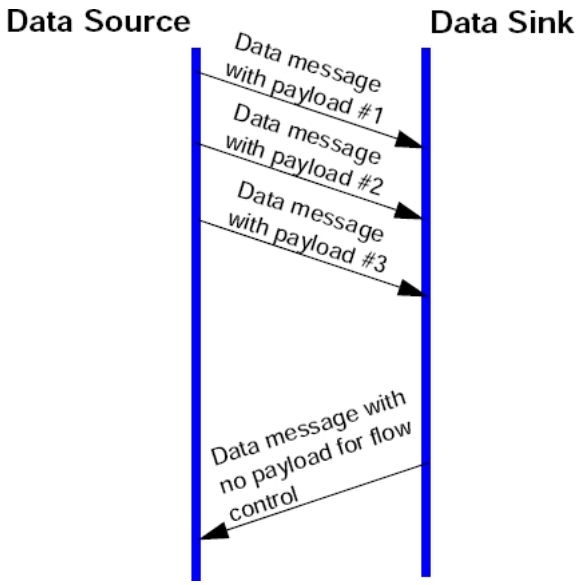

Figura 2.9: Transmissão de dados Bcopy. (INFINIBAND, 2002)

Os testes de desempenho em (GOLDENBERG et al., 2005) mediram a utilização de CPU e a vazão de dados dos métodos de transferência ZeroCopy e Bcopy. Através de Bcopy os dados são transmitidos por mensagens SDP, o que assemelha-se a uma transmissão de dados tradicional, enquanto com ZeroCopy a mensagem SDP enviada informa a estação de destino que deseja transmitir um dado e este, por sua vez, efetua a leitura no nodo de origem através de RDMA. Nos testes de desempenho foi utilizado um computador com dois processadores Xeon de 3GHz dotados de HyperThreading, totalizando assim quatro CPUs lógicas disponíveis para o sistema operacional. Deste modo os gráficos ao mostrarem 400% de utilização de CPU indicam que as 4 CPUs lógicas foram utilizadas em 100%.

O gráfico da Figura 2.10 apresentado em (GOLDENBERG et al., 2005) mostra que o recurso de Bcopy apresentou uma melhor vazão para mensagens de até 128 KB. A partir deste ponto, a transferência através de ZeroCopy apresentou uma melhor vazão. A utilização média de CPU através de ZeroCopy apresentou em todos os casos o melhor desempenho em relação a Bcopy onde mais de uma CPU por conexão foi necessária.

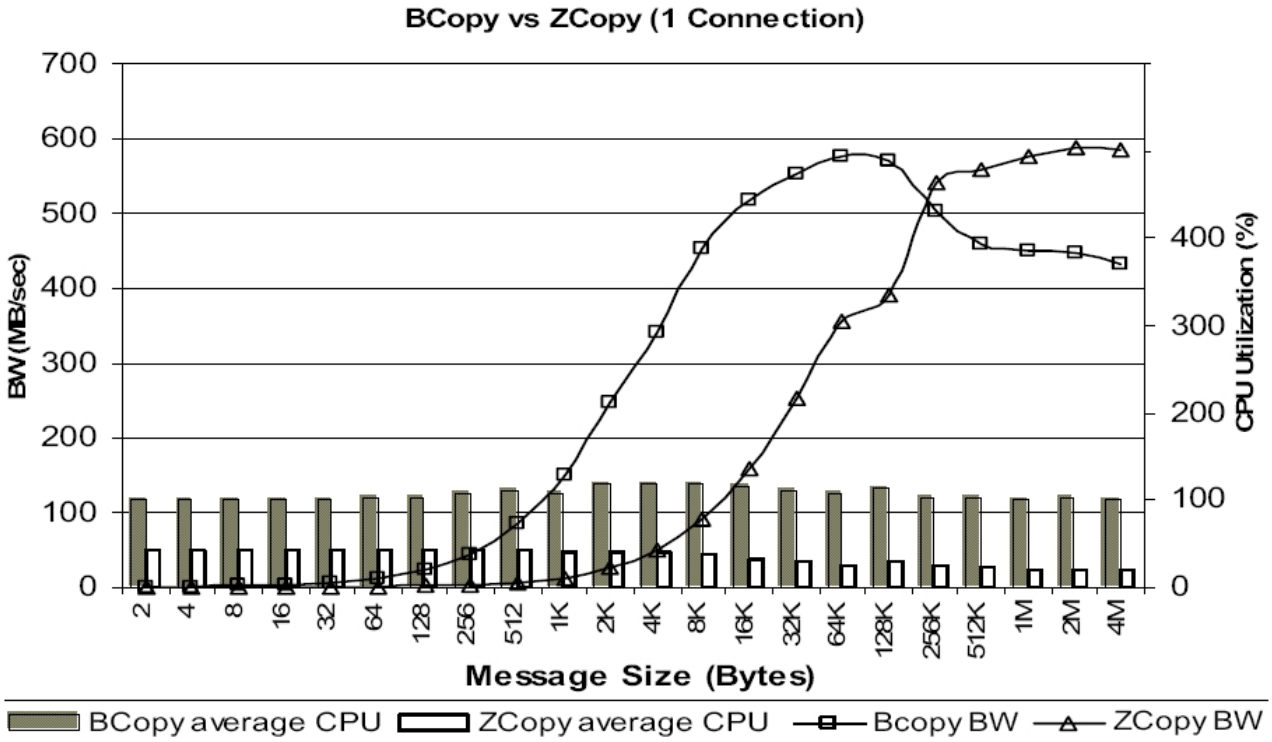

Figura 2.10: Utilização de CPU e vazão (*Bandwidth*) de Zerocopy e Bcopy para um única conexão

O gráfico ilustrado na Figura 2.11 (GOLDENBERG et al., 2005) demostra que ao aumentar o número de conexões de uma para quatro a utilização de CPU de Bcopy cresce linearmente na maior parte dos casos. É possível observar também que, considerando apenas a vazão, caso a utilização de CPU não seja um fator significativamente relevante para uma determinada aplicação, a transmissão de dados através de ZeroCopy torna-se vantajosa apenas para mensagens maiores que 32 KB.

Com os resultados apresentados em (GOLDENBERG et al., 2005) é possível observar que recursos que retiram do processador principal a função de transmissão de dados, como ZeroCopy por exemplo, apresentam um ganho significativo de desempenho em relação a soluções em que o processador é utilizado para cópia de dados (Bcopy).

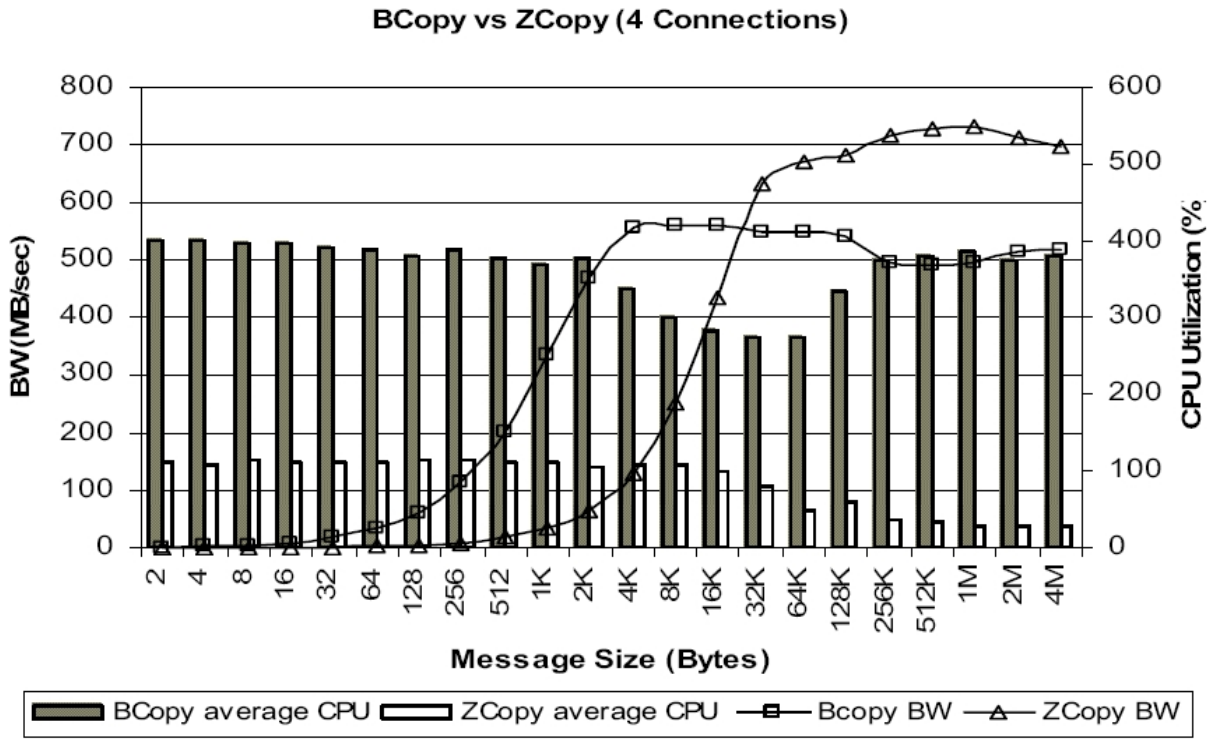

Figura 2.11: Utilização de CPU e vazão (*Bandwidth*) de ZeroCopy e Bcopy para 4 conexões simultâneas

3 ARQUITETURA DA PILHA DE COMUNICAÇÃO PROPOSTA

A arquitetura de pilha de comunicação proposta possui um número reduzido de camadas. As camadas da pilha de comunicação proposta são:

- Camada de Enlace: implementada através de uma MAC Gigabit Ethernet. O MAC que implementa a Camada de Enlace encontra-se descrito na Seção 3.2.1.
- Camada de Integração: foi desenvolvida com o objetivo de isentar da Camada de Aplicação qualquer processamento de dados de protocolos de rede.
- Camada de Aplicação: duas versões da Camada de Aplicação foram desenvolvidas. A primeira delas foi implementada em linguagem Verilog e utilizado como referência de desempenho devido sua simplicidade no recebimento e no repasse de dados. A segunda foi implementada com o microprocessador Microblaze e utiliza *software* embarcado leitura e escrita no barramento pelo qual comunica-se com a Camada de Integração.

A Figura 3.1 ilustra a arquitetura proposta. Entre as duas linhas pontilhadas da Figura 3.1 encontra-se a Camada de Integração. Esta camada implementa um protocolo responsável pela execução das 3 seguintes funcionalidades de rede: informação sobre o recebimento de quadros, gerenciamento das conexões e repasse dos dados à Camada de Aplicação. Este protocolo é consideravelmente simples, assemelhando-se a uma aplicação UDP/IP onde a confirmação de entrega de dados é realizada através dos pacotes UDP pela própria aplicação. Ressalta-se que nesta implementação não existe Camada de Rede ou de Transporte devido a utilização de

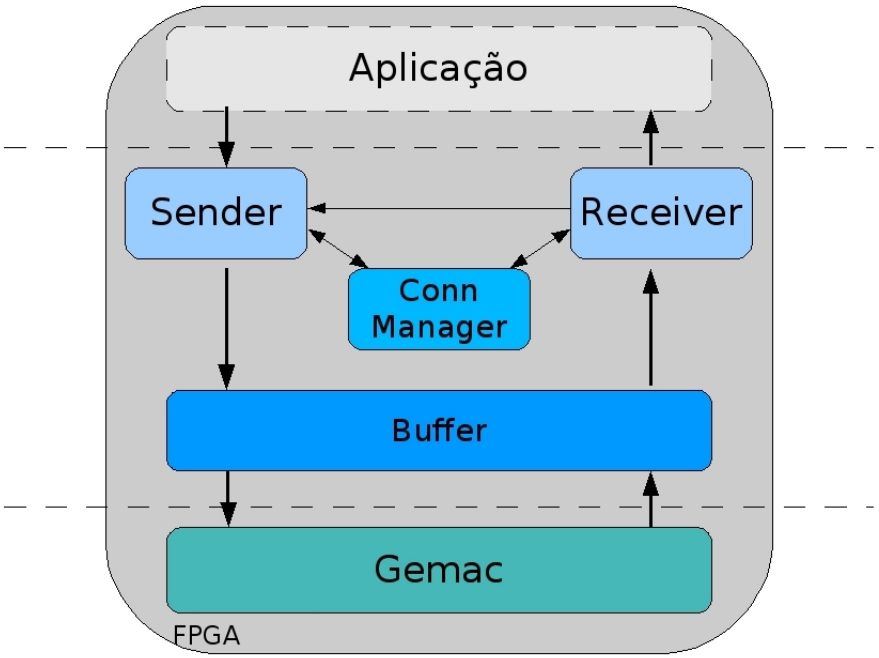

Figura 3.1: Pilha de comunicação do sistema proposto

quadros Ethernet puros. Neste caso, a camada de integração encontra-se posicionada onde o protocolo IP se encontraria em uma pilha tradicional.

3.1 Especificação da Arquitetura Proposta

A implementação foi realizada de modo que o comportamento *full-duplex* (envio e recebimento simultâneo) fosse apresentado na maior parte do tempo. Assim os blocos foram projetados para que pudessem realizar suas funções de maneira mais independente possível dos demais. Nas seções abaixo uma descrição detalhada de cada bloco.

A garantia de entrega é realizada utilizando quadros que confirmem o recebimento de dados (ACK). O gerenciamento das conexões dá-se através de uma chave exclusiva contendo endereço MAC de origem (MAC_ORIGEM) e identificador da conexão (C_ID) semelhante ao TCP/IP, que faz uso de uma chave contendo o endereço IP de origem e número da porta de origem. O C_ID é um número gerado pelo nodo de origem e deve ser único dentre as conexões ativas para este nodo. A ordem dos quadros enviados é realizada por um campo de seqüência (SEQ) que utiliza incremento de um. Embora estas soluções de gerenciamento de conexão sejam clara-

mente inseguras, elas representam uma maneira simples e eficiente para uma implementação em hardware e, como o foco é utilização em redes locais, isto não representa grande problema dado os objetivos do trabalho.

3.1.1 Campos do Protocolo

A Figura 3.2 apresenta o cabeçalho do protocolo implementado. Este cabeçalho segue imediatamente após o campo *Lenght/Type* do cabeçalho do quadro Ethernet.

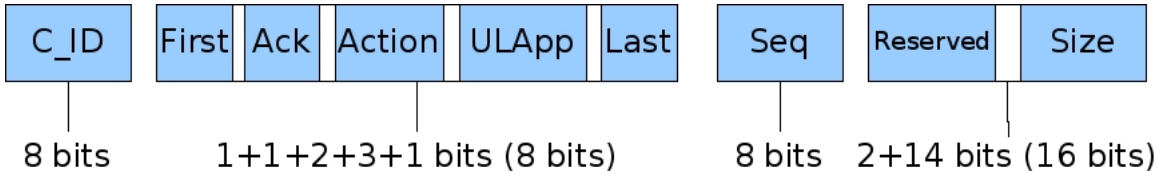

Figura 3.2: Cabeçalho do protocolo implementado

Os itens abaixo descrevem o protocolo implementado na Camada de Integração.

- *Seq*: Número de sequência do quadro. O valor inicial deste campo consiste num número inteiro aleatório entre 0 e 255 que é incrementado por um no nodo de origem a cada novo quadro enviado por este. Este campo é utilizado para conferência da sequência de quadros recebidos pelo nodo de destino. Também é utilizado para o envio de um quadro de confirmação de recebimento (ACK), este último devendo conter o mesmo valor de *Seq* do quadro que solicitou a confirmação. Tamanho: 8 bits. Seu valor então varia de 0 a 255 e é reiniciado para 0 após atingir 255.

- *Last*: este campo indica que o quadro em questão é o último de uma conexão. Em caso positivo deverá conter o valor '1', caso contrário, '0'. Tamanho: 1 bit.

- *First*: indica que o quadro atual demarca o de início de uma conexão. Sempre que um quadro com *First*='1' for recebido o nodo de destino deverá confirmar o seu recebimento com um quadro contendo o campo *ACK*='1' e sem dados válidos. Quadros com *First*='1' devem sempre satisfazer a chave única [C_ID,MAC_ORIGEM] caso contrário o quadro

será descartado e nenhum quadro contendo ACK será enviado. Após iniciada a conexão *First* deverá conter '0'. Tamanho: 1 bit.

- *Upper Level Application Number (ULApp)*: indica para qual aplicação acima da Camada da Integração os dados do quadro recebido estão destinados. Inicialmente este campo conterá o valor padrão '000' pois inicialmente haverá apenas uma aplicação acima da Camada de Integração. Tamanho: 3 bits.

- *Ack*: Indica que o nodo de origem solicitou a confirmação de recebimento do quadro. Para isto o nodo de destino gera um quadro do tipo ACK(Campo Action='01') e o envia para o nodo de origem. Tamanho 1 bit.

- *Action*: contendo valor '01' indica um quadro de confirmação de recebimento (ACK) em resposta a um quadro contendo *First*='1' ou um quadro contendo o campo *Ack*='1'. Valor '00' indica um envio normal de dados. Demais valores possíveis são reservados para uso futuro. Tamanho: 2 bits, sendo que 1 bit é reservado para uso futuro.

- *Connection ID (C_ID)*: é o identificador da conexão para o gerenciamento das conexões. O C_ID é escolhido pelo nodo de origem e deve ser único dentre as conexões ativas para um determinado MAC de origem, sempre satisfazendo a chave [C_ID,MAC_ORIGEM]. Tamanho: 8 bits.

- *Reserved*. Reservado para uso futuro, seja para suportar um tamanho de dados maior seja para outras opções. Tamanho 2 bits.

- *Size*: informa o tamanho em bytes da região de dados a ser repassada para a Camada de Aplicação. Tamanho: 14 bits. Este número é suficiente para representar um quadro jumbo de 16000 bytes tamanho de quadro este que é o máximo suportado no padrão Gigabit Ethernet.

Os algoritmos de recebimento e envio foram implementados em linguagem de descrição de *hardware* Verilog. Os fluxogramas do protocolo desenvolvido para o sistema proposto

encontram-se nas figuras 3.3 e 3.4 e representam os algoritmos de recebimento e envio, respectivamente.

3.2 Blocos do sistema

3.2.1 Gigabit Ethernet MAC (Gemac)

O Gigabit Ethernet Mac (Gemac) representa a Camada de Enlace. Este *core* possui a implementação completa de um MAC Gigabit e foi certificado pelo Laboratório de Interoperabilidade da Universidade of New Hampshire (UNH IOL). Este MAC é compatível com PHYs BASE-T, onde são utilizados cabos de rede tradicionais (Categoria 5 ou 6), e também com PHYs BASE-X que utilizam cabos de fibra ótica.

Uma vez que o núcleo Gemac é um *softcore*, ele pode ser alterado em várias características, como por exemplo filtro de endereços MAC. A Figura 3.5 mostra o conjunto de pinos de entrada e saída do núcleo Gemac para a configuração utilizada neste trabalho.

A lista abaixo mostra uma suscinta descrição de cada porta deste bloco.

- *gtx_clk*: clock de envio. Um clock de 125 MHz deve ser gerado e fornecido para o Gemac através desta porta de entrada.
- *tx_data[7:0]*: byte de dados para ser transmitido.
- *tx_data_valid*: um valor alto informa o Gemac a presença de um byte para ser transmitido em tx_data.
- *tx_ifg_delay[7:0]*: *interframe Gap Delay*. Indica o tempo de espera após a transmissão de um quadro.
- *tx_ack*: quando o lado cliente ativar tx_data_valid ele deverá aguardar até que um pulso alto de tx_ack seja gerado informando assim que é possível enviar os demais bytes de um quadro através de tx_data.

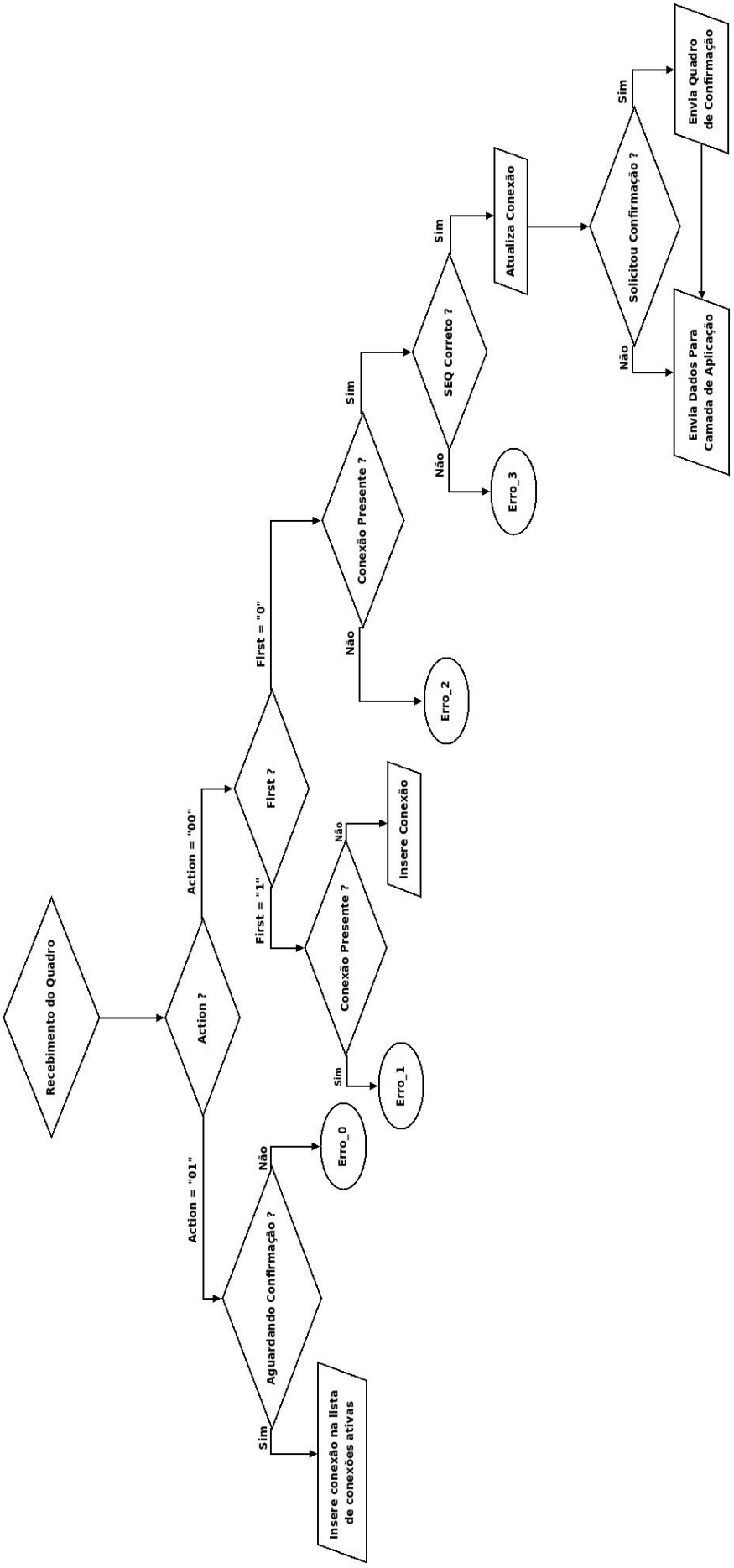

Figura 3.3: Fluxograma do algoritmo de recebimento de dados no qual foi baseada a máquina de estados do bloco *Receiver*

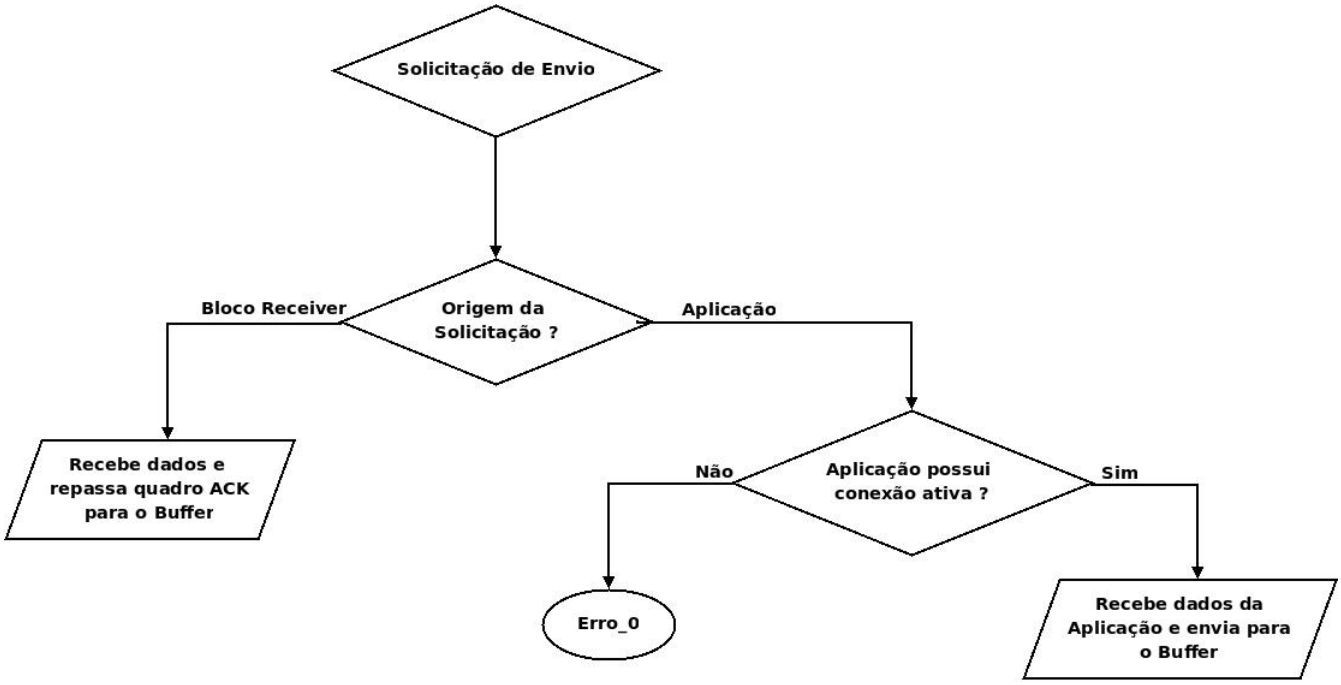

Figura 3.4: Fluxograma do algoritmo de envio de dados no qual foi baseada a máquina de estados do bloco *Sender*

- *tx_underrun*: quando o lado cliente decidir descartar um quadro antes que ele tenha sido enviado completamente tx_underrun deve ser ativada.
- *tx_statistics_vector[21:0]*: vetor contendo informações sobre o último quadro enviado.
- *tx_statistics_valid*: um pulso alto indica o instante em que o conteúdo de tx_statistics_vector é válido.
- *pause_req*: solicita ao Gemac a geração de um quadro de pausa.
- *pause_val[15:0]*: valor da pausa solicitada por pause_req.
- *rx_data[7:0]*: byte de quadro sendo recebido.
- *rx_data_valid*: informa a presença de dados válidos em rx_data.
- *rx_good_frame*: informa que o quadro recebido é válido. Um quadro é considerado válido sempre que possuir um CRC correto e suas demais características estiverem de acordo com a configuração do vetor de configuração (configuration_vector) como por exemplo, aceitar ou não quadros com endereço MAC diferente do configurado em mac_unicast_address, aceitar ou não quadros jumbo, dentre outras opções.

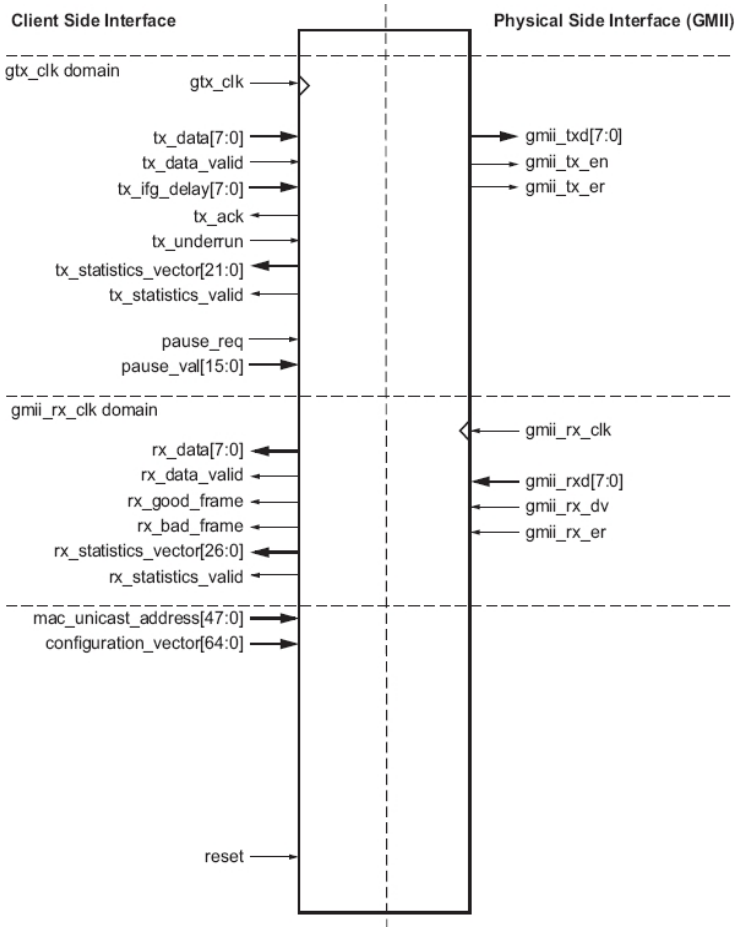

Figura 3.5: Pinos de entrada e saída do núcleo Gemac (XILINX, 2007b)

- *rx_bad_frame*: informa que o quadro recebido é inválido e deve ser descartado.
- *rx_statistics_vector[26:0]*: vetor contendo informações sobre o último quadro enviado.
- *rx_statistics_valid*: um pulso alto indica que o instante em que o conteúdo de rx_statistics_vector é válido.
- *mac_unicast_address[47:0]*: endereço MAC do bloco Gemac.
- *configuration_vector*: vetor de configuração. Através deste vetor é possível efetuar uma série de configurações dinâmicas no bloco Gemac. Mais destalhes podem ser encontras em (XILINX, 2007b);
- *reset*: reset ativo alto.
- *gmii_txd[7:0]*: bytes do quadro transmitidos ao PHY.

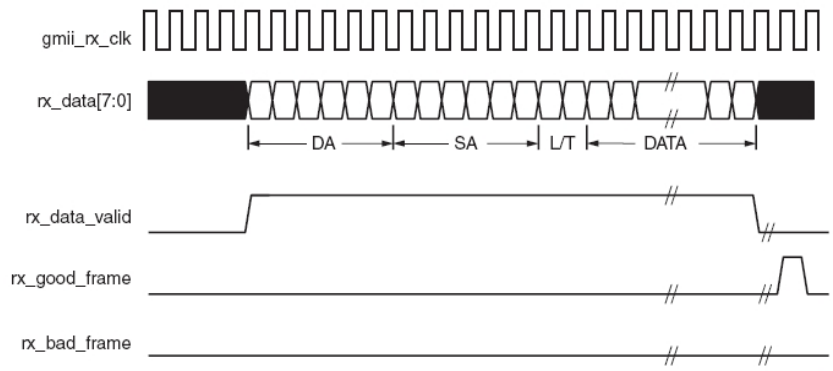

Figura 3.6: Recepção de um quadro Ethernet sem erros (XILINX, 2007b)

- *gmii_tx_en*: informa ao PHY a presença de dados válidos em gmii_txd.
- *gmii_tx_er*: informa ao PHY a ocorrência de um erro na transmissão.
- *gmii_rx_clk*: clock de recebimento. Este clock de 125 MHz é fornecido pelo PHY e usado para o recebimento dos bytes através de gmii_rxd.
- *gmii_rxd[7:0]*: bytes recebidos do PHY.
- *gmii_rx_dv*: informa ao bloco Gemac que existem dados válidos em gmii_rxd.
- *gmii_rx_er*: informa ao bloco Gemac sobre a ocorrência de um erro no recebimento.

As figuras 3.6 e 3.7 ilustram, respectivamente, a recepção e o envio de quadros através do Gemac. O comportamento apresentado nestas figuras é levado em consideração pelo Buffer para a execução das tarefas de recepção e envio de quadros.

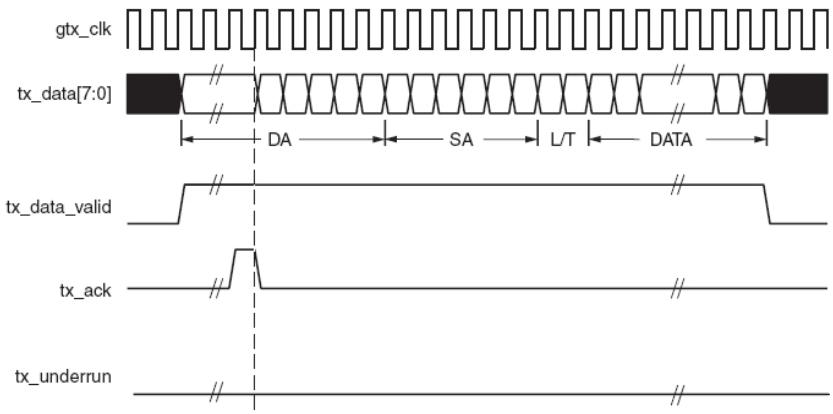

Figura 3.7: Envio de um quadro Ethernet (XILINX, 2007b)

3.2.2 Buffer

O *buffer* é subdividido em 2 FIFOs, sendo um para recebimento e outro para envio. Desta maneira é possível receber e enviar dados independentemente do estado dos blocos *Receiver* e *Sender*.

A memória dos FIFOs é composta por Block RAMs (BRAMs) que são memórias internas do dispositivo FPGA. Cada instância de BRAM, usualmente através de uma macrofunção, possui um tamanho de 2 KB assim foram utilizadas 8 BRAMs para cada FIFO de modo a armazenar um tamanho máximo de quadro de 16 KB o que é suficiente para armazenar um quadro do tipo jumbo de 16000 bytes. As BRAMs utilizadas possuem porta dupla de endereçamento e dados, ou seja, permitem que leitura e gravação sejam feitas simultaneamente.

Além do armazenamento de dados recebidos e de dados a serem enviados, o Buffer concatena os 8 bits vindos do Gemac em palavras de 32 bits, as quais são efetivamente gravadas nas BRAMs. Inversamente, o Buffer recebe as palavras de 32 bits oriundas do bloco Sender e as armazena nas BRAMs. Posteriormente, ao repassá-las para o Gemac, divide a palavra de 32 bits em palavras de 8 bits.

Este bloco utiliza o protocolo de barramento LocalLink (WEI; HUANG, 2007) para se comunicar com os blocos Receiver e Sender e o protocolo descrito nas figuras 3.6 e 3.7 para se comunicar com o Gemac. A Figura 3.8 ilustra o funcionamento do barramento LocalLink.

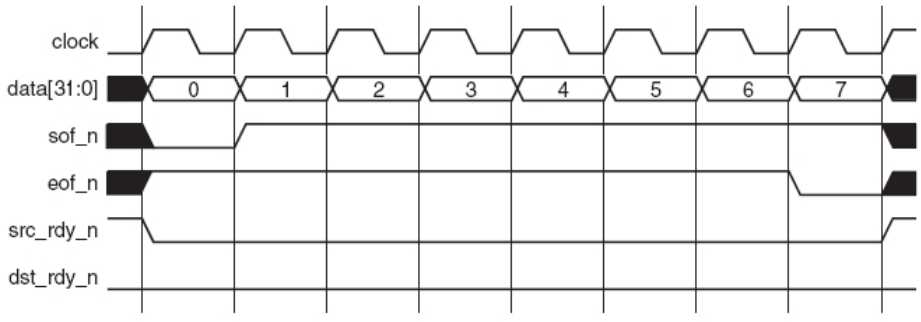

Figura 3.8: Funcionamento do protocolo LocalLink

Cada instância de BRAM possui capacidade de 2 KB podendo então armazenar 512 palavras de 32 bits. Assim, para o endereçamento foram utilizados 9 bits para cada instância de

BRAM. Para tornar o endereçamento transparente para cada FIFO, um endereço único de 12 bits foi gerado fazendo uso do sinais de habilitação de cada BRAM. Assim comparações entre o endereço de leitura e de escrita, necessárias para saber o quão cheio encontra-se o FIFO, foram bastante simplificadas. Desta maneira, como pode ser observado na tabela 3.1, os bits 9, 10 e 11 são utilizados para habilitação da BRAM a qual o endereço se refere, enquanto os demais bits (0 à 8), para endereçamento individual da BRAM habilitada.

Tabela 3.1: Habilitação de cada BRAM conforme o endereço

BRAM	Enable
0	$!addr[9]$ & $!addr[10]$ & $!addr[11]$
1	$addr[9]$ & $!addr[10]$ & $!addr[11]$
2	$!addr[9]$ & $addr[10]$ & $!addr[11]$
3	$addr[9]$ & $addr[10]$ & $!addr[11]$
4	$!addr[9]$ & $!addr[10]$ & $addr[11]$
5	$addr[9]$ & $!addr[10]$ & $addr[11]$
6	$!addr[9]$ & $addr[10]$ & $addr[11]$
7	$addr[9]$ & $addr[10]$ & $addr[11]$

3.2.3 Bloco Connection Manager

Este bloco é responsável pelo armazenamento das informações das conexões, bem como pelo fornecimento destas informações para os blocos *Receiver* e *Sender*.

As informações armazenadas por este bloco dizem respeito as conexões ativas e aguardando confirmação (ACK). Ele é consultado pelos blocos *Receiver* e *Sender* para que forneça informações sobre as conexões. Por exemplo, caso seja recebido um quadro de transferência de dados para uma conexão inexistente na lista de conexões ativas este bloco irá informar esta situação ao bloco *Receiver* que por sua vez irá descartar este quadro.

3.2.4 Bloco Receiver

Este bloco é responsável pelo recebimento de quadros, efetuando a leitura de dados do bloco *Buffer*, processo esse implementado pelo algoritmo descrito no fluxograma ilustrado na Figura 3.3. Este bloco também comunica-se diretamente com o bloco Sender para o envio de

quadros de ACK, o que permite que este tipo de quadro possa ser enviado com uma latência reduzida.

A comunicação com a Camada de Aplicação dá-se através do barramento Fast Simplex Link (FSL) (XILINX, 2007a). O barramento FSL facilita a integração com a Camada de Aplicação, neste caso um microprocessador Microblaze, bem como o controle de início e final de dados.

O barramento FSL possui um pino de entrada (*FSL_M_Control*, ativo baixo) que indica se a palavra no barramento é de controle ou de dados. Assim, este pino é ativado toda vez que um bloco de dados começa a ser enviado e ativado novamente no final deste bloco de dados. Desta maneira a Camada de Aplicação consegue facilmente determinar os limites de cada bloco de dados, pois a primeira e última palavras são dados comuns porém recebem o status de palavra de controle para demarcação de início e fim de quadro.

A gravação de dados no barramento dá-se através da ativação do pino *FSL_M_Write*. O Bloco Receiver monitora o pino *FSL_M_Full* proveniente do barramento FSL para verificar se o buffer do barramento está ou não cheio. Caso não esteja cheio o dado é colocado no barramento e a porta *FSL_M_Writ*e é ativada. Caso o buffer torne-se cheio antes do final de um quadro o bloco Receiver efetua uma pausa na leitura de dados do Buffer até que o buffer do barramento FSL possua espaço novamente, o que acontecerá conforme o software dentro da Camada de Aplicação efetua leitura de dados.

Mais detalhes sobre o barramento FSL encontram-se na seção 3.2.6.

3.2.5 Bloco Sender

Este bloco é responsável pelo envio de quadros, ou seja, efetua a gravação no bloco Buffer. O bloco Sender pode receber dados para enviar quadros tanto do bloco Receiver quanto diretamente da Camada de Aplicação. A máquina de estados deste bloco implementa o algoritmo apresentado na Figura 3.4.

Igualmente ao Bloco Receiver a comunicação com a Camada de Aplicação dá-se através do barramento FSL. O Bloco Sender monitora um pino de saída do barramento FSL que indica que

o dado presente no barramento é um dado de controle (*FSL_S_Control*, ativo baixo). Assim, monitorando este sinal de controle, o bloco determina o inicio e o fim de um bloco de dados a ser anexado ao cabeçalho de um quadro e então repassado ao bloco Buffer.

O Bloco Sender não possui memória interna, desta maneira todo dado recebido é imediatamente repassado ao Buffer. O controle de fluxo de dados do protocolo FSL é realizado através da porta de leitura (*FSL_S_Read*). Assim, sempre que esta porta estiver desativada o barramento FSL não fornecerá dados. A porta *FSL_S_Exists* indica a existência ou não de dados a serem lidos. Desta maneira, sempre que o Bloco Sender não estiver ocupado com envio de quadros ACK, *FSL_S_Exists* estiver ativa e o Buffer de saída não estiver cheio, o Bloco Sender efetuará leitura de dados do barramento FSL.

3.2.6 Aplicação

A Camada de Aplicação é onde ocorre o processamento dos dados do quadro. Sendo assim, esta camada não recebe nenhuma informação relativa aos protocolos de rede e, consequentemente, não necessita realizar nenhum processamento relativo aos protocolos de rede envolvidos.

Duas versões da Camada de Aplicação foram desenvolvidas neste trabalho. A primeira foi desenvolvida de linguagem Verilog e serviu como referência para aferir o impacto que segunda Camada de Aplicação trouxe para o sistema. A segunda foi desenvolvida utilizando o microprocessador Microblaze versão 7.00.a sobre o qual um aplicação em linguagem C foi desenvolvida.

A pilha de comunicação com a Camada de Aplicação descrita em Verilog é denominada Arq. Proposta (HDL App) nos gráficos de resultados. A pilha de comunicação com a Camada de Aplicação implementada em software sobre o microprocessador Microblaze é denominada Arq. Proposta (MB App) nos gráficos de resultados.

Barramento Fast Simplex Link (FSL)

O barramento FSL é um barramento unidirecional utilizado para implementar comunicação entre dois blocos de hardware em FPGA quanto estes implementarem a interface FSL (XILINX,

2007a). A interface FSL encontra-se disponíveis nas versões mais recentes do processador Microblaze.

Uma vez que o FSL é unidirecional duas instâncias do Core IP FSL_V20 foram utilizadas para integração da pilha de comunicação com o Microblaze. Desta maneira, foi possível que gravação de dados para o microprocessador, efetuada pelo bloco Receiver, e a leitura de dados vindos do microprocessador, efetuada pelo bloco Sender, pudessem ser realizadas em paralelo.

Uma vez que o microprocessador Microblaze é capaz de gerenciar apenas um fluxo de execução, a leitura e gravação de dados em paralelo nos barramentos FSL só foi possível por conta da existência de um FIFO interno de até 8192 bytes em cada barramento. Assim, os blocos Receiver e Sender podem gravar e ler desde que haja espaço e dados disponíveis nos FIFOs internos do barramento. Estes FIFOs podem ser implementados com BRAMs, o que reduz a área do FPGA utilizada com memória, fato este consideravelmente interessante para FIFOs com tamanho superior a 1024 bytes.

A Figura 3.9 ilustra os pinos de entrada e saída de cada interface do barramento FSL. Como é possível observar através do barramento *Master* (pinos *FSL_M_**) são realizadas as gravações de dados e através do barramento *Slave* (pinos *FSL_S_**) são realizadas as leituras. Desta maneira a integração dos blocos Receiver e Sender com as duas instâncias do *Core IP* FSL_V20 (fsl_0 e fsl_1) ocorre conforme ilustrado na Figura 3.10.

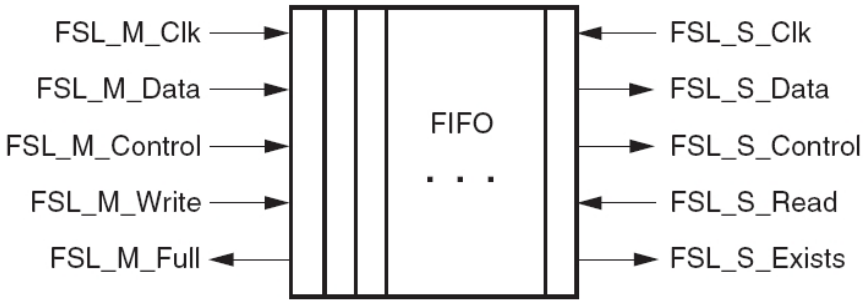

Figura 3.9: Pinos de entrada e saída do barramento FSL (XILINX, 2007a)

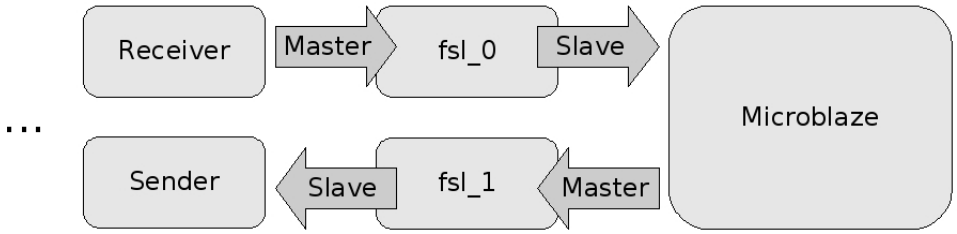

Figura 3.10: Conexão entre os blocos Receiver e Sender com as instâncias dos barramentos FSL e o microprocessador Microblaze

Software da Camada de Aplicação

Com a migração das funcionalidades de rede para uma camada dedicada em hardware e com a utilização do barramento FSL, o software da Camada de Aplicação pode ser bastante simplificado.

O acesso aos dados gravados pelo bloco Receiver no barramento FSL é realizado por intermédio das funções de aceso ao barramento FSL. Estas funções encontram-se no arquivo *header* fsl.h. O código abaixo ilustra a leitura de quadros:

Listagem 3.1: Código fonte C para leitura de quadros do barramento FSL

```
#include "fsl.h";

int main (void) {
    int value,is_error,eof;
    eof=0;
    while(1){
        getfslx(value,0, FSL_CONTROL);
        fsl_iserror(is_error);
//      xil_printf("value=%d | error=%d",value,is_error);
        if((is_error==16)){
                mtmsr(0);
                if(eof==0){
                  xil_printf("inicio de quadro\n");
                  eof=1;
                }
                else{
```

```
17                      xil_printf("fim de quadro\n");
                        eof=0;
19                    }
              }
21  }
```

A função *getfslx* efetua a leitura no barramento FSL. O primeiro parâmetro desta função retorna o valor lido, o segundo refere-se ao número do barramento FSL de onde deseja-se ler e o terceiro indica determinadas características desta operação. Neste caso a diretiva FSL_CONTROL indica a tentativa de ler um dado de controle de modo bloqueante. Independente da palavra lida ser ou não uma palavra de controle seu valor será retornado na variável *value*.

A diferenciação entre uma palavra de controle, que no caso indica início e final de um bloco de dados pertencente a um quadro, ocorre através do valor do registrador MSR (*Machine Status Registrer*) que é lido através da função fsl_iserror(). Se após a execução da função *getfslx(value,0, FSL_CONTROL)* a função fsl_iserror retornar o valor '0', a palavra lida foi gravada juntamente com a ativação de *FSL_M_Control*, ou seja, é um palavra de controle. Caso o valor retornado seja '16', isto indica que a palavra foi gravada com o sinal de controle baixo, no caso deste trabalho, ativo. Uma vez que o registrador MSR recebe algum valor através das funções FSL é necessário que ele seja resetado. A chamada *mtmsr(0)* atribui o valor '0' ao registrador novamente.

A chamada à função *xil_printf* mostrada acima pode ser utilizada para a visualização dos dados lidos. Porém, uma vez que a interface de E/S RS232 é por padrão utilizada para Entrada e Saída padrão de dados, o desempenho da aplicação tende a ser consideravelmente inferior. O código mostrado acima recebe constantemente quadros sendo função do desenvolvedor do software de aplicação sair do laço, manipular dados e enviar quadros. O código de exemplo para envio de quadros encontra-se abaixo.

Listagem 3.2: Código fonte C para escrita de quadros no barramento FSL

```
1  #include "fsl.h";
```

```
int main (void) {
  int y,quadros_a_enviar,tamanho,sof;
  y=0;
  quadros_a_enviar=2;
  tamanho=2000;
  sof=0;
  while(y < quadros_a_enviar ){
    x = 0;
    while(x<tamanho-1){
      if(sof==0){
        putfslx(x,0,FSL_DEFAULT);//FSL_S_Control=0,
                                 // começo do bloco de dados
        xil_printf("inicio do quadro gravado\n");
        sof=1;
      }
      else{
        putfslx(x,0,FSL_CONTROL);//FSL_S_Control=1,
                                 // dados normais
      }
//      xil_printf("gravando dado %d\n",x);
      x++;
    }
      putfslx(x,0,FSL_DEFAULT);//FSL_S_Control=0,
                               //fim do bloco de dados
      xil_printf("fim do quadro gravado\n");
      sof=0;
      y++;
  }
}
```

Igualmente à função *getfslx* a função *putfslx* também possui três parâmetros com o mesmo significado. Diferentemente, não é preciso verificar o status de sua operação, uma vez que os parâmetros FSL_DEFAULT e FSL_CONTROL realizam a operação de gravação de modo

bloqueante. A função ficará bloqueada até que possa ser executada, por exemplo, caso o FIFO do barramento esteja cheio e seja necessário aguardar pela liberação de espaço para gravação.

3.3 Aplicação de referência

Uma aplicação que implementa os algoritmos da pilha de comunicação proposta ilustrados nas Figuras 3.3 e 3.4 foi desenvolvida em linguagem C com objetivo de executar sobre um computador pessoal. Esta aplicação foi utilizada nas medições de desempenho para efeito de comparação com a arquitetura implementada em FPGA e também para efetuar a comunicação com a placa de desenvolvimento FPGA.

Para uma comparação mais justa, a aplicação utiliza dois fluxos de execução (*threads*) implementados através da biblioteca Pthreads. Esta aplicação executou sobre um processador Pentium IV HT que suporta a execução de mais de um fluxo de execução em paralelo juntamente com o sistema operacional Linux dotado de um kernel com suporte SMP (*Symmetric Multi Processing*). Além disto, recursos da biblioteca de tempo real RTAI (*Real Time Application Interface*)(TEAM, 2009), juntamente com um kernel modificado para suportá-los, foram utilizados de modo a garantir que a execução sofresse o mínimo de interrupções e praticamente não fosse preemptada pelo sistema operacional.

Ensaios foram realizados para aferir se o comportamento esperado descrito acima realmente ocorria com a utilização da RTAI. Dentre eles uma aplicação devia executar um laço com um determinado número de somas enquanto outros processos, fora do escopo desta aplicação e sem utilizar a biblioteca RTAI, também eram executados. Os testes confirmaram que a influência dos processos externos é insignificante se comparada com a mesma aplicação sob as mesmas condições porém sem a utilização dos recursos da RTAI.

3.4 Metodologia

Inicialmente, foi realizada uma implementação em software, através da linguagem C e do compilador Gcc (Gnu C Compiler), do mesmo protocolo de comunicação do sistema proposto.

Este protocolo faz uso de quadros Ethernet puros e possui um cabeçalho imediatamente após os campo *Length/Type* do quadro Ethernet. Esta implementação em *software* ajudou a validar o sistema implementado em linguagem de descrição de hardware (HDL) e também foi utilizada para comparação de desempenho com o sistema proposto. Detalhes sobre a validação do sistema encontram-se na Seção 3.4.1.

O sistema proposto foi implementado na placa de desenvolvimento ML402 (XILINX, 2006) dotada de um FPGA Xilinx XC4VSX35 da família Virtex-4. Para a síntese do sistema foi utilizado o *software* ISE da própria Xilinx e para simulação o *software* utilizado foi o Modelsim da empresa Mentor Graphics. Para o desenvolvimento de grande parte do sistema foi utilizado o software HDL Designer, também fornecido pela Mentor Graphics, que auxiliou na tarefa de codificação HDL, bem como na organização do trabalho. O microprocessador utilizado na integração foi Microblaze versão 7.00.a, integrado a arquitetura de pilha de comunicação proposta através do *software* XPS, fornecido pela Xilinx.

Para depuração e teste do sistema foi utilizado um analisador lógico Agilent modelo 1682AD dotado de 64 canais digitais.

As medições de desempenho foram feitas conectando um computador pessoal à placa ML402 e enviando dados através do *software* implementado em C. O computador pessoal utilizado foi um *notebook* Dell Vostro dotado de um processador Core 2 Duo T8300 de 2.4 GHz e 3MB de cache L2 com 4 GB de memória RAM e interface de rede Gigabit Ethernet. Para a realização das medições de desempenho apenas da versão em *software* foi utilizado um computador dotado de um processador Pentium IV 3.0 GHz HT com 1MB de cache L2 e 2 GB de memória. O *software* foi implementado com dois fluxos de execução (*threads*), um para envio e um para recebimento, utilizando a biblioteca Pthreads.

O sistema operacional utilizado foi o Gnu/Linux, distribuição Ubuntu 8.04.2, com kernel 2.6.24 e *pacth* para biblioteca de tempo real RTAI. A biblioteca RTAI permitiu que as medições de desempenho pudessem ser realizadas sem que outros processos interferissem no desempenho do processo que realizava a comunicação com a placa ML402.

3.4.1 Metodologia de verificação

A verificação funcional do sistema implementado teve como base a aplicação de referência desenvolvida em software para PC apresentada na seção 3.3. Os dados enviados e recebidos por esta aplicação foram capturados através de um software *sniffer* e salvos no formato Tcpdump. O dados no formato Tcpdump foram convertidos para a sintaxe da linguagem Verilog através de um *script* Perl desenvolvido neste trabalho. Este *script* fez uso do módulo Net::TcpDumpLog (GREGG, 2003) para facilitar o acesso aos dados no formato Tcpdump. Uma vez convertidos para Verilog, os dados adquiridos através do software *sniffer* foram utilizados em *testbenches* que injetavam estímulos na entrada da pilha de comunicação e analisavam as respostas obtidas na saída da pilha comparando-as então com com as saídas esperadas também disponíveis no arquivo gerado pelo software *sniffer*. A Figura 3.11 ilustra este processo.

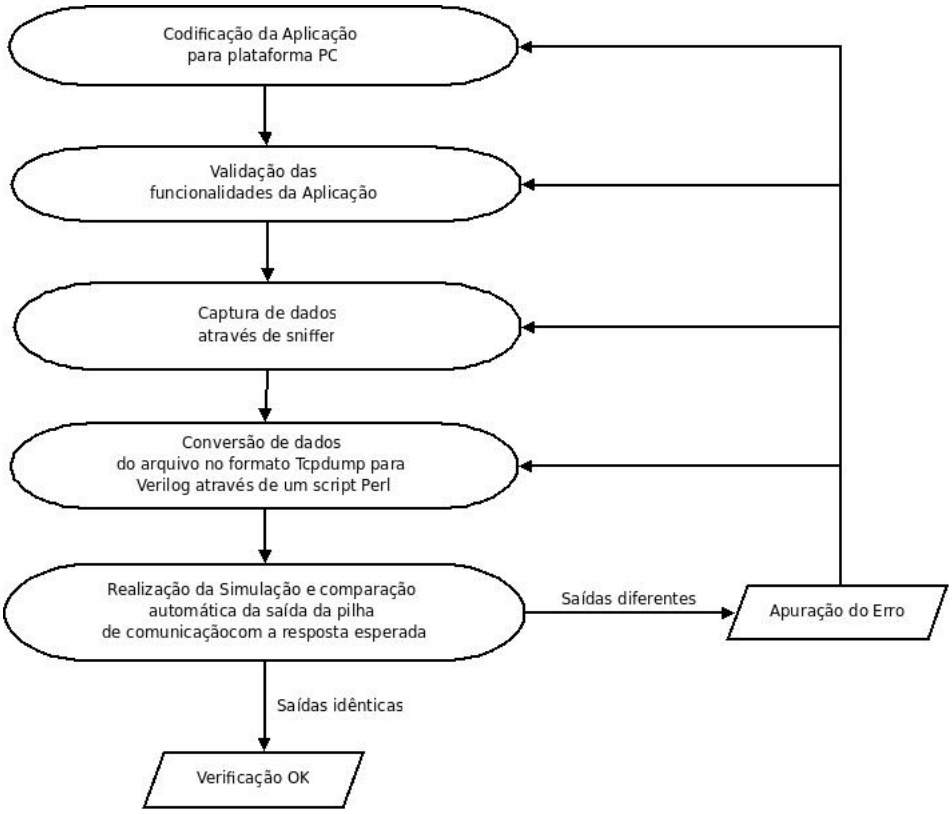

Figura 3.11: Fluxograma do processo de verificação

Desta maneira foi possível automatizar o processo de verificação, o que agiliza o processo e o torna menos suscetível a erros humanos comuns em inspeções visuais. Além da validação funcional o sistema também foi validado considerando os atrasos do FPGA utilizado (simulação

Post Place and Route) e então implementado na placa ML402 onde os detalhes finais foram testados através de um analisador lógico Agilent 1682AD.

O *script* Perl desenvolvido encontra-se na listagem A.1 no Apêndice A.

4 RESULTADOS

A arquitetura de pilha de comunicação foi implementada em uma placa de desenvolvimento ML402. Os resultados apresentados neste Capítulo foram obtidos conectando a placa ML402, cujo dispositivo FPGA encontrava-se configurado com a implementação descrita na Seção 3.1, a um computador pessoal através de um cabo de rede Categoria 6. A partir do computador pessoal os dados foram enviados e recebidos e as medições de tempo foram realizadas.

A Figura 4.1 ilustra o ambiente montado para a realização das medições de desempenho. Ao realizar o envio de um dado é realizada a tomada de tempo T0, quando este dado retornar é realizada a tomada de tempo T1. A diferença entre T0 e T1 e quantidade de dados tranferida neste período são utilizadas para o cálculo da vazão expressa em Mb/s.

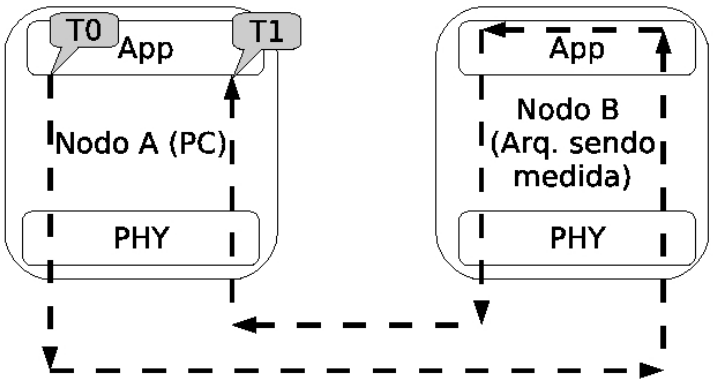

Figura 4.1: Ambiente de medição de desempenho

4.1 Comparações realizadas

A mesmas medições realizados na pilha de comunicação proposta neste trabalho também foram realizadas em outras pilhas de comunicação comumente utilizadas. A subseções seguin-

tes descrevem estas pilhas de comunicação.

4.1.1 Lightweight IP (lwIP)

Uma vez que o foco principal da arquitetura desenvolvida neste trabalho é fornecer maior vazão e menor latência para microprocessadores, a biblioteca lwIP é uma importante opção neste nicho. Ela implemententa os protocolos TCP, UDP, IP e ARP e é distribuída juntamente com o Embedded Development Kit (EDK) da empresa Xilinx, sendo assim nativamente suportada pelo microprocessador Microblaze.

A arquitetura completa de uma pilha de comunicação que emprega a lwIP contém a Camada de Enlace implementada em hardware dedicado. Esta arquitetura é ilustrada na Figura 4.2.

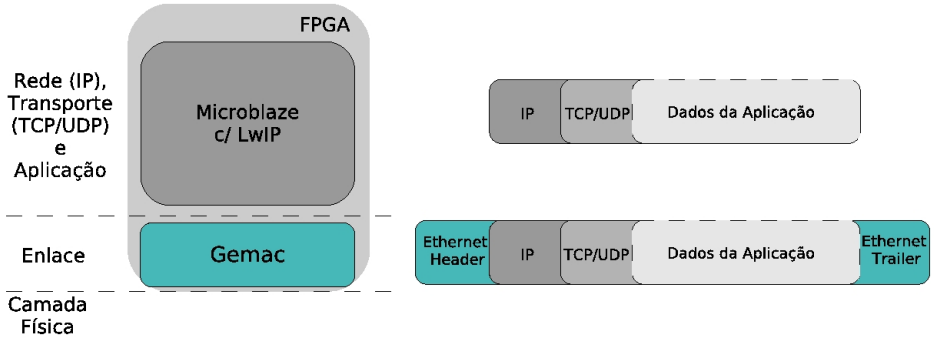

Figura 4.2: Arquitetura da pilha de comunicação utilizando a biblioteca lwIP

Para uma comparação justa o processador Microblaze foi configurado da mesma maneira tanto na arquitetura de pilha utilizando a lwIP quanto na arquitetura desenvolvida neste trabalho. Ambos foram configurados com 8 KB de memória cache de dados, 8 KB de memória cache de instruções e frequência de operação de 125 MHz.

A pilha de comunicação Lightweight IP é denominada lwIP UDP e lwIP TCP nos gráficos de resultados.

4.1.2 Pilha de comunicação tradicional do sistema operacional Linux

A pilha de comunicação tradicional do sistema operacional Linux também foi utilizada nas medições realizadas neste trabalho. No caso desta pilha o computador fazendo o papel de

cliente (Nodo A na Figura 4.1) comunicou-se com um outro computador pessoal (Nodo B) sob o qual o mesmo software, porém executando a função de servidor, ou seja, aguardando dados, encontrava-se em execução.

Como já afirmado anteriormente, este software utiliza dois fluxos de execução (*threads*) e faz uso de quadros Ethernet puros. Os dois fluxos de execução consistem nos processos de envio e recebimento e, uma vez que o processador do computador pessoal utilizado possui capacidade para executar dois fluxos de execução simultâneos, esta característica tornou a comparação mais justa fazendo melhor uso do hardware disponível no computador pessoal. A utilização de quadros Ethernet puros permite que a aplicação crie um *socket* de acesso a rede que possa receber dados diretamente da camada de enlace, ou seja, sem a necessidade de passar por nenhum outro protocolo intermediário tanto para envio quanto para recebimento. A Figura 4.3 ilustra a descrição acima.

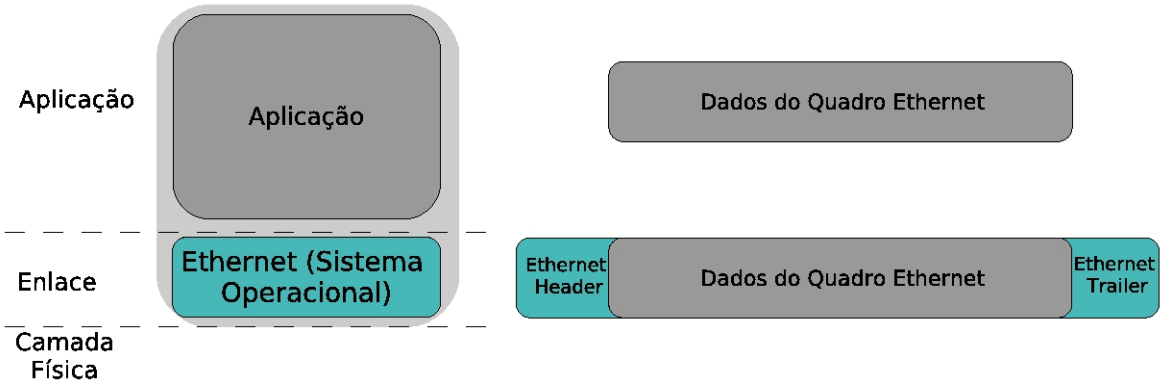

Figura 4.3: Pilha de comunicação tradicional do sistema operacional Linux com as camadas utilizadas neste trabalho

A pilha de comunicação radicional do sistema operacional Linux é denominada PC Software (Linux) nos gráficos de resultados.

4.2 Resultados de Latência

Os resultados de latência foram obtidos enviando um único quadro para a arquitetura que estava sendo avaliada e comparando o tempo inicial (T0) com o tempo de retorno do quadro (T1), conforme ilustrado na Figura 4.1.

Mesmo com a utilização da RTAI no Nodo A houve uma variação de 2% no tempo de resposta da pilha desenvolvida neste trabalho, tanto para a pilha com Camada de Aplicação em Verilog quanto a com o microprocessador Microblaze. Teoricamente, poderia ser esperado que, ao realizar estas medições as respostas seriam totalmente idênticas em todas as medições realizadas para o mesmo tamanho de quadro devido à utilização de hardware dedicado no Nodo B e a utilização da RTAI no Nodo A.

Porém, por fim, concluiu-se que estas variações eram realmente oriundas do comportamento do sistema operacional do PC. Corroborou para esta conclusão a documentação da RTAI que informa que pela simples utilização da RTAI e suas funções não é possível garantir a execução exclusiva de um processo caso este invoque funções que retirem o *kernel* do modo *Hard Real Time* e coloque o processo novamente sob o comando do *kernel* Linux tradicional, justamente o que ocorre quando da execução de alguma das funções de rede.

Para que esta variação de 2% no tempo de resposta pudesse ser minimizada, no caso da pilha proposta neste trabalho, foram enviados 1000 quadros de cada um dos tamanhos medidos para cada uma das arquiteturas que representaram o Nodo B neste *benchmark*. Adicionalmente uma média dos tempos de respostas foi realizada, reduzindo assim ainda mais a influência causada pelo sistema operacional nos resultados. A solução ideal para este caso seria a utilização da extensão RTnet da própria RTAI, a qual implementa protocolos de rede em tempo real. Esta solução, entretanto, é dependente de *drivers* específicos para cada placa de rede. O driver para a placa de rede do Nodo A não se encontrava disponível para a versão utilizada na época de realização deste trabalho.

O gráfico da Figura 4.4 apresenta os valores de latência para cada uma das arquiteturas de pilha de comunicação mensuradas. Como é possível observar os valores de latência para PC Software e para as duas versões da arquitetura proposta são bastante semelhantes.

A implementação de rede no *kernel* do sistema operacional consiste em uma parte relevante dos resultados apresentados. A maneira como o sistema operacional atende aos quadros recebidos é crucial para os resultados de latência (BENVENUTI, 2005). Assim, com a exis-

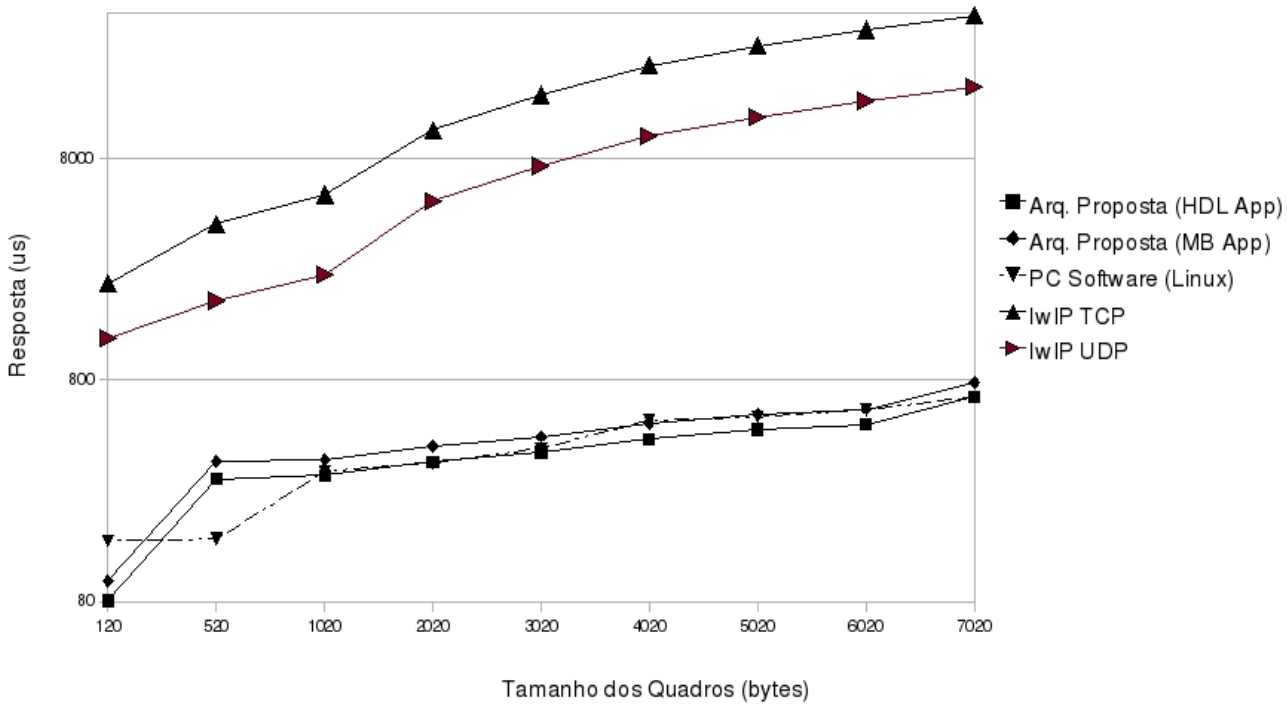

Figura 4.4: Latência por tamanho de quadro

tência da diferença entre a recepção real (recepção pelo PHY do Nodo A) e a efetiva recepção pelo sistema operacional (e consequente tomada de T1) resulta que os valores de T1 sejam determinados pelo mecanismo de recepção de quadros do sistema operacional. Ou seja, faz-se necessária uma significativa diferença entre T0 e T1 para que esta diferença possa efetivamente ser percebida pelo sistema operacional e não seja atenuada pela diferença entre recepção real e recepção do sistema operacional.

No caso das três comparações citadas é possível afirmar que as variações mais significativas em relação aos tamanhos de quadro de 120 e 520 bytes. Para quadros de 120 bytes a latência foi 51% melhor a Arquitetura proposta utilizando o microprocessador Microblaze quando comparada com a arquitetura PC. Considerando que a implementação em software no computador pessoal (PC Software) rodou sobre um processador muito mais robusto, frequência de operação 24 vezes maior e 128 vezes mais cache, este resultado, apesar de modesto, é relevante.

Para quadros de 520 bytes a Arquitetura proposta utilizando o processador Microblaze apresentou um desempenho 55% inferior. É importante notar que o desempenho da implementação em software executada sob um PC apresentou praticamente valores de latência praticamente

idênticos para 120 bytes (151 microssegundos) e para 520 bytes (154 microssegundos). Este fato é claramente relacionado à maneira como o sistema operacional de ambos os nodos atende aos quadros recebidos. Uma vez que a quantidade de bytes aumentou em mais de 4 vezes, o que faz com que o PHY da placa de rede do PC gaste 4 vezes mais ciclos de clock para o recebimento de dados, era esperado um aumento ao menos próximo de 4 vezes.

Comparando a arquiteura desenvolvida neste trabalho com uma pilha desevolvida em software embarcado (lwIP) os resultados foram muito superiores. A pilha embarcada lwIP é largamente disponível e de fácil implementação com a utilização das ferramentas da empresa Xilinx. As medições realizadas mostraram um ganho de mais de 51 vezes no melhor caso (quadros de 6020 bytes) e de mais de 32 vezes em média para conexões TCP/IP através da lwIP. Para conexões UDP/IP o ganho foi de mais de 24 vezes no melhor caso (igualmente quadros de 6020 bytes) e de mais de 15 vezes em média.

Este ganho significativo em comparação à lwIP deve-se justamente à grande quantidade de ciclos de processamento despendidos por parte do microprocessador Microblaze no processamamento de instruções relativas aos protocolos da pilha de comunicação (ARP, IP, UDP, TCP). Uma vez que arquitetura de pilha de comunicação desenvolvida neste trabalho implementa características de rede e gerenciamento de conexões fazendo uso de hardware dedicado, uma quantidade reduzida de software embarcado (apenas leitura e escrita no barramento FSL conforme ilustrado na Seção 3.2.6) faz-se necessária.

4.3 Resultados de Vazão

Os resultados de vazão foram obtidos utilizando o mesmo ambiente empregado na apuração dos resultados de latência. A Figura 4.5 ilustra os resultados de vazão para as arquiteturas mensuradas.

Comparando as duas versões da arquitetura proposta (com e sem o microprocessador na Camada de Aplicação) é passível de destaque o impacto que a adição do microprocessador causou à arquitetura. Em média, a vazão foi reduzida em 9 vezes o que deixa claro o impacto do

microprocessador de características modestas e, consequentemente de software, nesta arquitetura. Por outro lado, enquanto a vazão máxima da arquitetura proposta com o microprocessador na Camada de Aplicação foi de 141,04 Mb/s (quadros de 6020 bytes) a vazão máxima com a Camada de Aplicação em Verilog foi de 1311,63 Mb/s (quadros de 7020 bytes) o que demonstra que a arquitetura proposta é capaz de fornecer uma maior quantidade de dados desde que o microprocessador da Camada de Aplicação possua capacidade para processar esta vazão ou que mais microprocessadores sejam instanciados em paralelo.

Considerando a pilha de comunicação do sistema operacional Linux o desempenho da arquitetura proposta com microprocessador Microblaze foi em média 3,8 vezes inferior e 1,47 vezes inferior no melhor caso (quadros de 120 bytes). Tendo em vista que a capacidade do processador do PC sob o qual o sistema operacional Linux foi executado (frequência de operação 24 vezes maior e 128 vezes mais memória cache) a redução média de 3,8 vezes na vazão denota alguns benefícios desta arquitetura. Assim, uma menor utilização de software para funcionalidades de rede e a migração destas funcionalidades de rede para hardware dedicado colaboram significativamente no desempenho da arquitetura como um todo, uma vez que poderia se esperar uma redução maior no desempenho considerando a grande diferença de recursos disponíveis em cada processador e a igual utilização de quadros Ethernet puros.

Por outro lado, os resultados de vazão da arquitetura proposta com a Camada de Aplicação em Verilog foram em média 2,51 vezes superiores e 4,7 vezes no melhor caso (quadros de 120 bytes). Assim, é possível afirmar que caso o gargalo da Camada de Aplicação possa ser superado a arquitetura proposta possui potencial para um desempenho superior ao apresentado por sua versão com o processador Microblaze como Camada de Aplicação.

Analisando o desempenho da pilha embarcada lwIP, a arquitetura proposta mostrou uma vazão significativamente superior. A vazão média da arquitetura proposta com o microprocessador microblaze foi de 27,03 vezes maior com a utilização de TCP/IP pela lwIP. Com a utilização de UDP/IP a vazão média foi 18,56 vezes superior. Os melhores resultados da lwIP foram em quadros de 120 bytes para TCP/IP, apresentando 5,8 vezes menos vazão, e para quadros 1020 bytes UDP/IP, apresentando 5,24 menos vazão em comparação com a arquitetura proposta.

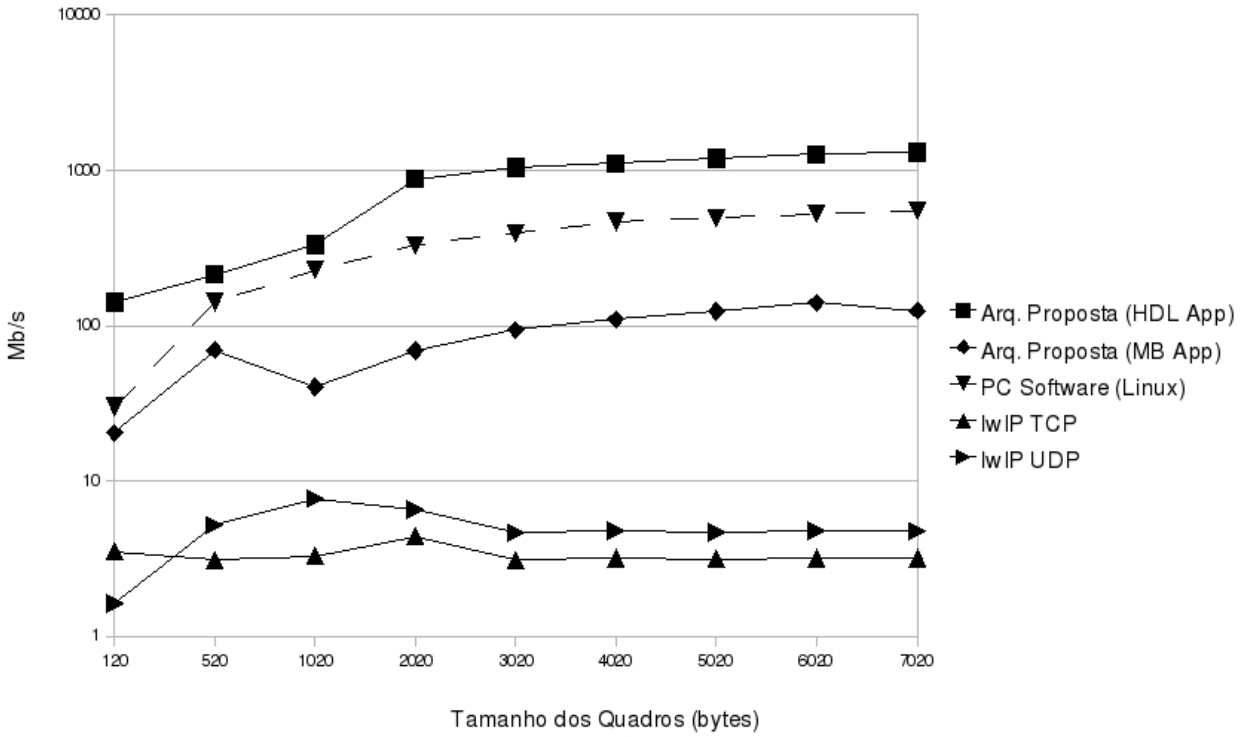

Figura 4.5: Vazão de dados por tamanho de quadro

5 CONCLUSÃO

Este trabalho apresentou um proposta de pilha de comunicação com um número reduzido de camadas. A pilha de comunicação foi desenvolvida em linguagem de descrição de hardware e implementada em uma placa de desenvolvimente dotada de interface Gigabit Ethernet e um dispositivo FPGA.

Os resultados apresentados no Capítulo 4 demostram que esta arquitetura é interessante para prover uma menor latência e maior vazão em comparação com pilhas de comunicação embarcadas. Além disso, considerando que tanto a arquitetura proposta, quanto a arquitetura com pilha embarcada fazem uso na camada de enlace de um Mac Ethernet Gigabit, o hardware adicional requiriu apenas 11% de área programável no dispositivo FPGA utilizado. Assim sendo, 1,11 vezes mais hardware gerou resultados significativamente superiores na maioria dos casos das comparações realizadas neste trabalho.

Os resultados demostraram também uma redução de desempenho quando a Camada de Aplicação é implementa em software. No melhor caso a vazão com a Camada de Aplicação em hardware dedicado foi de 1311,63 Mb/s enquanto a vazão máxima com a Camada de Aplicação em software foi de 141,04 Mb/s, demostrando uma subutilização da arquitetura de pilha proposta quando é utilizado software na Camada de Aplicação. Assim, no intuito de evitar esta subutilização da pilha de comunicação, torna-se interessante uma implementação com mais de um microprocessador Microblaze.

5.1 Trabalhos Futuros

Além da utilização de mais de um processador Microblaze para melhor aproveitamento da capacidade de vazão da pilha de comunicação proposta, os seguintes pontos podem ser alvo de trabalhos futuros.

- Implementação com várias camadas de aplicação. Considerando a redução na vazão gerada pela utilização do microprocessador Microblaze na Camada de Aplicação é considerável a utilização de mais de uma instância do microprocessador na Camada Aplicação. Para isto, pode ser utilizado o campo *Upper Level Application Number* (UllApp) para a seleção de saídas e entradas de demultiplexadores e multiplexadores para cada sinal do barramento FSL. A redução na vazão máxima com a utilização do microprocessador foi de cerca de 9 vezes, assim sendo, uma simples modificação nos blocos Sender e Receiver faz-se necessário pois o campo UllApp pode endereçar no máximo 8 aplicações. Esta modificação passaria a utilizar os bits Reservados como parte do campo UllApp.

- Desenvolvimento de um modelo através de *Network Calculus* (BOUDEC; THIRAN, 2004). *Network Calculus* é um conjunto de trabalhos recentes na área da Matemática que provêem recursos para a solução de problemas na área de redes de computadores, tais como controle de fluxo, escalonamento e dimensionamento de *buffer* e atraso. Um modelo adequado poderá informar qual o impacto de vazão e latência conforme a variação de determinados parâmetros do modelo.

- Implementação da pilha de comunicação em um placa interconectável a um computador pessoal. Considerando os resultados obtidos neste trabalho torna-se interessante um estudo, e a eventual implementação, desta pilha de comunicação como parte de um computador pessoal. Uma placa Gigabit Ethernet com um barramento PCI poderia ser utilizada para transpassar totalmente a pilha de comunicação do sistema operacional de modo que o processo que deseje receber e enviar dados através desta pilha deva apenas gravar e ler do barramento PCI. A tecnologia SATA também poderia ser utilizada como meio de in-

terconexão dos nodos de rede uma vez que placas de desenvolvimento em FPGA dotadas desta tecnologia já encontram-se disponíveis no mercado.

REFERÊNCIAS BIBLIOGRÁFICAS

BENVENUTI, C. *Understanding Linux Network Internals*. [S.l.]: O'Reilly, 2005.

BODEN, N. J. et al. Myrinet – a gigabit-per-second local-area network. In: . [S.l.: s.n.], 1995.

BOUDEC, J.-Y. L.; THIRAN, P. *NETWORK CALCULUS - A Theory of Deterministic Queuing Systems for the Internet*. [S.l.: s.n.], 2004.

BROSE, E. Zerocopy: Techniques, benefits and pitfalls. *Hot topics on OS (WS2005/06 TU Berlin)*, 2005.

CLARK, D. et al. An analysis of tcp processing overhead. v. 27, n. 6, p. 23–29, 1989. ISSN 0163-6804.

CLARK, D. et al. An analysis of tcp processing overhead. v. 40, n. 5, p. 94–101, 2002. ISSN 0163-6804.

GILDER, G. *Telecosmo - A Era Pos-Computador*. [S.l.: s.n.], 2001.

GOLDENBERG, D. et al. Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis. In: *Proc. 13th Symposium on High Performance Interconnects*. [S.l.: s.n.], 2005. p. 128–137. ISSN 1550-4794.

GREGG, B. *Net::TcpDumpLog - Read tcpdump/libpcap network packet logs. Perl implementation (not an interface).2*. [S.l.], 2003. Disponível em: <http://search.cpan.org/dist/Net-TcpDumpLog/TcpDumpLog.pm>.

HAMERSKI, J. C. *Desenvolvimento de uma arquitetura Parametrizavel para Processamento de Pilha TCP/IP em Hardware*. Dissertação (Mestrado) — UFRGS, 2008.

INFINIBAND. *Sockets Direct Protocol (SDP) - Supplement to Infiniband Architecture Specification*. [S.l.], 2002.

LOFGREN, A. e. a. An analysis of fpga-based udp/ip stack parallelism for embedded ethernet connectivity. 2004.

LU, W. *Designing TCP/IP Functions In FPGAs*. Dissertação (Mestrado) — TU Delft, 2003.

MELAS, P.; ZALUSKA, E. Performance of message-passing systems using a zero-copy communication protocol. In: *Proc. International Conference on Parallel Architectures and Compilation Techniques*. [S.l.: s.n.], 1998. p. 264–271.

MYRICOM. *Sockets-MX Overview and Performance*. [S.l.], 2007.

RECIO, R. e. a. *RFC 5040: A Remote Direct Memory Access Protocol Specification*. Outubro 2007.

ROMANOW, A. et al. Rfc 4297 - remote direct memory access (rdma) over ip problem statement. *IETF*, 2005.

SARDA, D. *Message Passing Using Raw Ethernet Frames In Parallel Computing*. [S.l.], 2003.

SHAFER, J.; RIXNER, S. *A Reconfigurable and Programmable Gigabit Ethernet Network Interface Card*. [S.l.], 2006.

SHIVAM, P.; WYCKOFF, P.; PANDA, D. Emp: Zero-copy os-bypass nic-driven gigabit ethernet message passing. In: *Proc. ACM/IEEE 2001 Conference Supercomputing*. [S.l.: s.n.], 2001. p. 49–49. Zerocopy.

SKEVIK, K.-A. et al. Evaluation of a zero-copy protocol implementation. In: *Proc. 27th Euromicro Conference*. [S.l.: s.n.], 2001. p. 324–330.

SMITH, M. J. S. Gigabit ethernet and transport offload: transport offload engines help relieve tcp processing burden for gigabit ethernet. *Computer Technology Review*, 2002.

TEAM, T. R. *RTAI - the RealTime Application Interface for Linux*. [S.l.], 2009. Disponível em: <www.rtai.org>.

VALENTIM, R. A. M.; JUNIOR, V.; OLIVEIRA, L. A. G. d. Performance analysis of protocols: Udp and raw ethernet to real-time networks. *In: International Workshop on Telecommunications - IWT 2007*, 2007.

WEI, W. Y.; HUANG, D. Parameterizable locallink fifo. *Xilinx Application Notes*, 2007.

WOODRUFF, R. J. *Access to InfiniBand* from Linux**. [S.l.], 2008.

XILINX. *ML401/ML402/ML403 Evaluation Platform - User Guide*. [S.l.], Maio 2006.

XILINX. Fast simplex link (fsl) bus. *Product Specification*, 2007a.

XILINX. *LogiCORE(TM) 1-Gigabit Ethernet MAC v8.3 User Guide*. [S.l.], 2007b.

APÊNDICE A – LISTAGEM DE CÓDIGOS FONTE

Listagem A.1: Script Perl utilizado para conversão de arquivos no formato TCPDump em vetores na sintaxe Verilog para utilização em *testbenches*

```
1  #!/usr/bin/perl
   #
3  # This program is free software: you can redistribute it and/or modify
   # it under the terms of the GNU General Public License as published by
5  # the Free Software Foundation, either version 3 of the License, or
   # (at your option) any later version.
7  #
   # This program is distributed in the hope that it will be useful,
9  # but WITHOUT ANY WARRANTY; without even the implied warranty of
   # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
11 # GNU General Public License for more details.
   #
13 # You should have received a copy of the GNU General Public License
   # along with this program.  If not, see <http://www.gnu.org/licenses/>.
15 #
   # gera_tb (v0.1): Extract bytes from sniffed tcpdump files and
17 # output them as Verilog vectors.
   #
19 # Based on Brendan Gregg example02.pl of 11-Oct-2003 -
   # Example of Net::TcpDumpLog. Prints out ethernet headers.
21 # Depends on Net::TcpDumpLog
   # (http://search.cpan.org/~bdgregg/Net-TcpDumpLog-0.11/TcpDumpLog.pm)
23 #
```

```
# Copyright (C) 2008  Josué Paulo José de Freitas <josue.freitas@gmail.com>
25
use Net::TcpDumpLog;
27
sub pos_arg { # Para que seja possível colocar os argumentos em qualquer
    ordem
29          my $cnt = 0;
            my $temp = shift(@_);
31      #     print $temp;
            foreach $arg (@ARGV){
33              # print $ARGV[$cnt];
                if($temp eq $arg){
35                      $posicao=$cnt+1;
                        # return $cnt+1;
37                  }
                else{ $posicao=0; }
39              $cnt++;
                # shift local(@ARGV);
41                  }
        return $posicao;
43      }

45 $pos=pos_arg("--help");

47 if ($pos != 0){
        printf "Usage: gera_tb.pl <tcp_dump_file> <options>\n";
49      printf "Options:\n";
        print  "        --gd   : gerar declaracao do FRAME_TYP";
51      print  "        --no-frame-valid: não imprime vetor de bytes válidos");
        print  "        --no-frame-error: não imprime vetor de erro");
53      exit;
        }
55
```

```
57
   $log = Net::TcpDumpLog->new();
59 $log->read($ARGV[0]); # Lê o arquivo tcpdump passado pela linha de comando.
   @Indexes = $log->indexes;
61 #printf "%5s %12s %12s %4s  %s\n","Frame","Source","Dest","Type","Length";


63
   $countFrame=0;
65 $maxData=0;

67 if(!(pos_arg('--gd') eq 0)){
       foreach $index (@Indexes) {
69         $data = $log->data($index);
           if(length($data) > $maxData){
71           $maxData=length($data);
           }
73     }
       goto frm_typ;
75 }

77 foreach $index (@Indexes) {
       $data = $log->data($index); # $index);
79 #    printf "DATA : %d\n\n",$data[1];
       $length = length($ether_data);
81
       $pos1=pos_arg("--no-frame-valid");
83     $pos2=pos_arg("--no-frame-error");

85     $data_count=0;
   #   printf "data_l".length($data);
87     while( $data_count < length($data)  ){

89         $data_nibble=substr($data, $data_count,1);
```

```
            printf "frame%d.data[%d]=8'h%X;",$countFrame,$data_count,ord(
                $data_nibble);
91
            if ($pos1 eq 0){ printf "  frame%d.valid[%d]  = 1'b1;",$countFrame,
                $data_count; }
93          if ($pos2 eq 0){ printf "  frame%d.error[%d]  = 1'b0;",$countFrame,
                $data_count; }

95          print "\n";

97          # printf "%x==%08b\n",ord($data_nibble),ord($data_nibble);
            # print "      assign frame[$count]=4'b$data_nibble;\n";
99          $data_count+=1;
        }
101
#printf "data=%s\n",$data;
103
    $countFrame+=1;
105
}
107
frm_typ:
109
if(!(pos_arg('--gd') eq 0)){
111     $cnt = 0;
        print "bits = {";
113     while($cnt < $maxData){
            print "data[$cnt], ";
115         $cnt=$cnt+1;
        }
117     print "\n";
        $cnt = 0;
119     while($cnt < $maxData){
            print "valid [$cnt], ";
```

```
121             $cnt=$cnt+1;
                }
123         print "\n";
            $cnt = 0;
125         while($cnt < $maxData){
                print "error[$cnt], ";
127             $cnt=$cnt+1;
                }
129         print "badframe }";
    }
```

APÊNDICE B – ARTIGOS PUBLICADOS

Artigo aceito no Fórum de Estudantes de Microeletrônica 2009.

A FPGA-based Network stack with a reduced number of layers

Josue P. J. de Freitas,Gustavo F. Dessbesell,Joao Baptista dos Santos Martins
UNIVERSIDADE FEDERAL DE SANTA MARIA (UFSM)
josue.freitas@mail.ufsm.br,gfd@mail.ufsm.br,batista@inf.ufsm.br

ABSTRACT

This paper presents a proposal regarding a Network stack in FPGA with reduced number of layers. The network stack uses is composed by a Gigabit Ethernet Mac, connected to an Application Layer by a Middle Layer. The Middle layer implements some network features, like connection management and buffering. The Middle layer aim to be responsible for all network processing this way letting the Application layer free to process application information. Our proposed network node show a maximum 1311.63 Mb/s full-duplex throughput. Besides, considering just the internal 32 bits data bus and operation frequency of 125 MHz it's possible to reach around 8000Mb/s of full-duplex internal throughput. Our architecture presents a throughput 4.7 times higher in the best case (2.5 times higher on average) when compared to a software implementation running over personal computer. Also, comparing with an embedded network stack our architecture, using a microprocessor in Application layer, shows in the best case 44 times more throughput.

1. INTRODUCTION

The available bandwidth is increasing very quickly nowadays. According to Gilder's law [4] the available bandwidth doubles every six months in average. Considering Moore's law, which says that the number of transistors in an integrated circuit doubles approximately every two years.

A common network stack, developed in operating system software, shows a ratio around of 1 MHz to 1 Mb/s of full-duplex throughput [7] [6]. On an RFC regarding RDMA (Remote Direct Memory Access) [6] is shown that, in 2001, a 1.2 GHz Athlon could handle a maximmum theoretical 2.7Gb/s of full-duplex throughput when using 100% of CPU capacity, i.e. the machine have too few CPU cycles to process user applications if we consider the Gigabit Ethernet technology available at that time. Copy through memory is considered a huge consumer of CPU cycles in software network stack implementations [2], this fact stimulated the development of hardware based network stacks and industry standard network stacks with a reduced number of layers [5] [1].

Although the relation between general purpose processor and available bandwidth wasn't a huge problem in the past, this problem is increasing since network bandwidth is increasing much faster than processors capacity, even considering the multi-core architectures. The graphic in figure 1 shows processors frequency and the available bandwidth through the years. The graphic was assembled based on Intel processors released through the years available at http://www.intel.com/pressroom/kits/quickreffam.htm and Ethernet 802.3 standards releasing dates, and projections, found at http://www.ieee802.org/. In this graphic is possible to notice that this relation wasn't problematic before 1998, when Gigabit Ethernet standard was released. After this year the available bandwidth kept steady above processors capacity and it might keep this way in a long future since 100Gb/s Ethernet, 802.3ba standard, is already in experimental state in 2009.

Considering that embedded system use to have simple microprocessors, with few cache memory and lower frequency, this even more significant. So, this paper addresses these issues by proposing an FPGA-based network stack with a Middle layer responsible with network processing connected to a Gigabit Ethernet MAC and to an Application layer, in order to avoid the microprocessor getting involved in network stack processing.

The paper is organized as follows: Section 2 explains the network node architecture and relevant information. Section 3 presents results where our network stack is compared with pure software running over a PC and with an embedded an embedded TCP/IP network stack, lwIP[3]. Section 4 shows conclusions.

2. PROPOSED NETWORK NODE ARCHITECTURE

The Network node is composed by three main parts: a Gigabit Ethernet Mac (Gemac), a Middle layer, and an Application 2 . The Gemac block communicates with the buffer and Receiver and Sender blocks read and write data from and to the buffer. The Middle layer, situated between Application and Gemac, is where some network features are implemented. Furthermore, the Application is where data is processed, it communicates with the Middle layer by writting and reading to and from Sender and Receiver blocks.

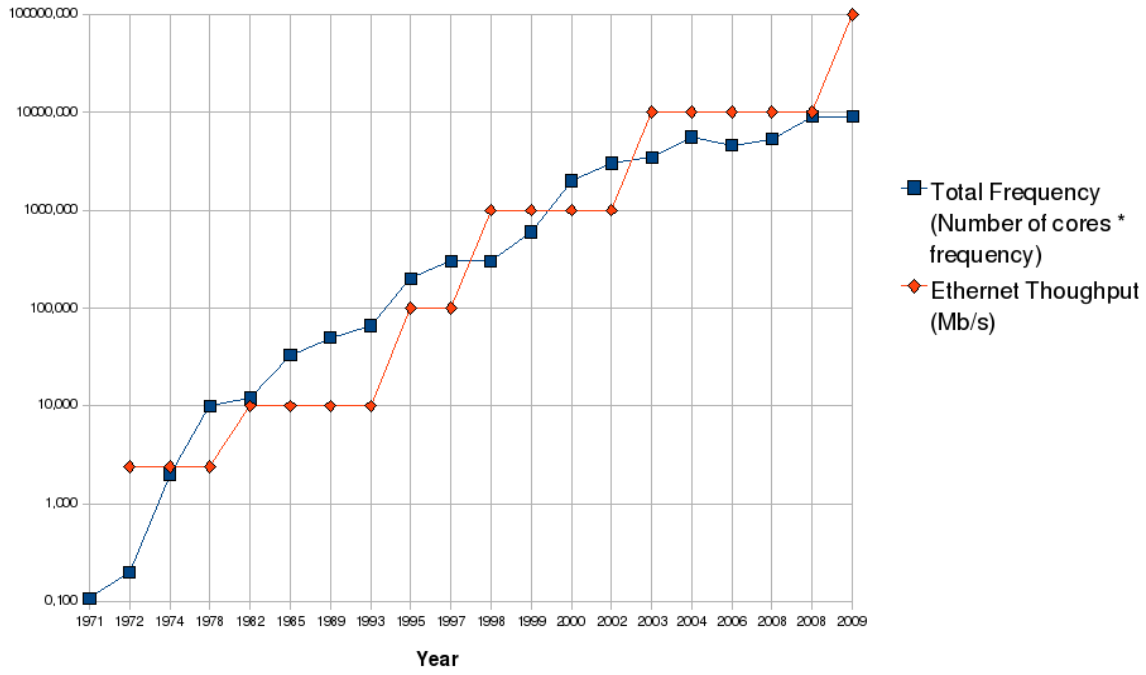

Figure 1: Available bandwidth versus Processors frequency

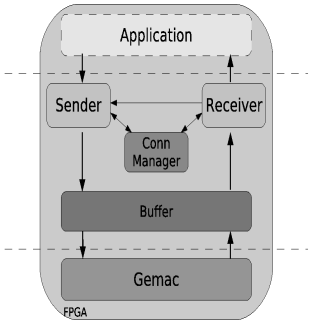

Figure 2: Proposal Network Node Architecture

A buffer is needed in order to support different applications that may have very different time to process the incoming data, this way avoiding discarding frames.

The Middle layer is composed by the following blocks: Sender, Receiver, Connection Manager and Buffer. The Receiver block checks some values in the control header and decides whether the inbound frame is correct or not. The control header is composed by 5 bytes after the Length/Type field of the Ethernet Frame and is shown in Figure 3. The fields First, Ack, Action, UlApp, Last, Seq were grouped into a single byte to make the bit manipulation easier in software, allowing the use of an unsigned char type to represent them.

- CID: Connection IDentification. An eight bit number that is chosen by the host that starts the communication;
- First: A '1' value indicates the communication first frame. The Receiver block checks if there is no other active connection with the same key [SOURCE_MAC,CID]. If not, the connection is inserted on the Active Connection List through Connection Manager block;
- Ack: A '1' value indicates that source host requests an acknowledgement frame;
- Action: Together with other field indicates the type of the frame. A '00' means data transfer operation and '01' means an Acknowledgment (ACK) response;
- UlApp: Indicates to which Upper Level Application the frame is destinated to;
- Last: A '1' value indicates communication end, causing the key [SOURCE_MAC,CID] to be removed from Active Connections List. If both First='1' and Last='1' the connection is not added to the Active Connection List;
- Seq: The Sequence field. It is used to check if the frames are sent correctly and without any loss.
- Reserved: Reserved for future use.
- Size: Size of Data in bytes. It was chosen to be 14 bits long since this way is possible to represent 16384 bytes of data, which is enough for the maximum Jumbo frame size that our architecture will support.

A finite state machine on the Receiver block realize all verifications. After the verifications from the Receiving algorithm, the header is stripped and stored in an entry on Connection Manager block and the data is forwarded to the Application layer. The Sender blocks communicates with the Connection Manager, getting information about the connection, and, in ACK frame sending cases, it communicates

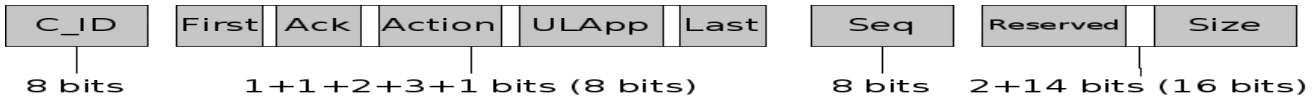

Figure 3: Control header

Table 1: Device utilization for the proposed network stack architecture

Resource	Used	Available	Used(%)
Number of Slice Flip Flops	3,529	30,720	11%
DCM autocalibration logic	7	3,529	<1%
Number of 4 input LUTs	4,141	30,720	3%
Number of occupied Slices	4,080	15,360	26%
Number of RAMB16s (BRAM)	16	192	8%
Number of DCM_ADVs	1	8	12%

directly with the Receiver block. The Sender block communicates with the Application in order to read data processed by the Application.

3. IMPLEMENTATION

Our network node has been implemented in a Xilinx ML402 evaluation board. This board has Gigabit Ethernet PHY and a Xilinx Virtex 4 SX35 FPGA. Our system achieves a maximum frequency of 128 MHz, which satisfies Gigabit Ethernet requirements, where design must reach at least 125 MHz of maximum frequency. Block RAM (BRAM) were used in the Buffer implementation. The buffer size is 32 KB, where one half stores receiving data from Gemac and the other half stores data to be sent.

Resource utilization is available on table 1. By looking at this table is possible to notice that there are enough resources to improve our design in many ways, for example, increasing buffer size and improving parallelization.

Although the bus size used to communicate with Gemac must be 8 bits wide, our proposed architecture uses an 32-bit internal bus. This way, considering the clock frequency employed in our system (125 MHz), it is possible to reach an internal 4 Gb/s throughput in each way, i.e. an 8 Gb/s full-duplex internal throughput. The buffer block also chops and assembles the 32 bits to 8 bits and vice-versa in order to communicate with Gemac.

The software implementation, developed in C language using raw ethernet frames, was build using two threads through library Pthreads. One thread handles the data sending process and the other the data receiving process. This way, the comparison between software version and our FPGA architecture can be considered more fair than using a single thread implementation.

4. RESULTS

The results have been obtained by sending frames of different sizes to our proposed network node and for the software version implemented in two PCs. Both PCs where the software version run were a Pentium IV HT with 3.0GHz over a Linux operational system with a SMP (Symmetric Multi-Processing) kernel, which allows the software application to use both threads in parallel.

The measurements have been made by using a PC connected through an Ethernet cable to the ML402 board. The same client software used in pure software measurements was used on the PC side to connect with our network stack on ML402 board. The ML402 board was configured with all architectures listed below, excepting PC Software running under Linux operational system. The Application layer on each architecture implements a pong application, which means that each application should response with the same data it was received.

- Proposal Arch (HW App): Our proposal architecture using an Application layer developed in Hardware Description Language.
- Proposal Arch (MB App): The same proposal architecture using a Microblaze©microprocessor, so this architecture have a software based Application layer.
- PC Software (Linux): A software developed in C language running over Linux operational system. This software implements the same features that our proposal architecture implements.
- lwIP TCP: A Gigabit Ethernet Mac connected with a Microblaze©microprocessor where lwIP embedded network stack was implemented, in this case using TCP sockets.
- lwIP UDP: The same as above instead using UDP sockets in communication.

The charts on Figures 4 and 5 show the latency and throughput results for ping-pong tests for our architecture and for other implementations. The latencies results show almost an equivalence between or proposal architecture and PC Software. The main reason behind these latency results is how latency test were made. One frame with specific size is sent (T0 time) and then the application keeps waiting until it comes back (T1 time) (see figure 6). Since just one frame is sent on this test, our architecture pipeline is empty, thus not using hardware resources like its inherent parallelism.

Although, comparing our architecture with lwIP embedded stack we achieve results more than 51 times better when using 6020 byte frame size. On average our latencies results were around 32 times better than TCP embedded software and around 15 times better than UDP.

On the other hand, throughput shows, on average, 2.5 times better results than the software implementation. The best result, for 120-byte frames, allows 4.7 times more throughput than in the software implementation, altough it represents just 111.54 Mb/s more throughput. Considering the

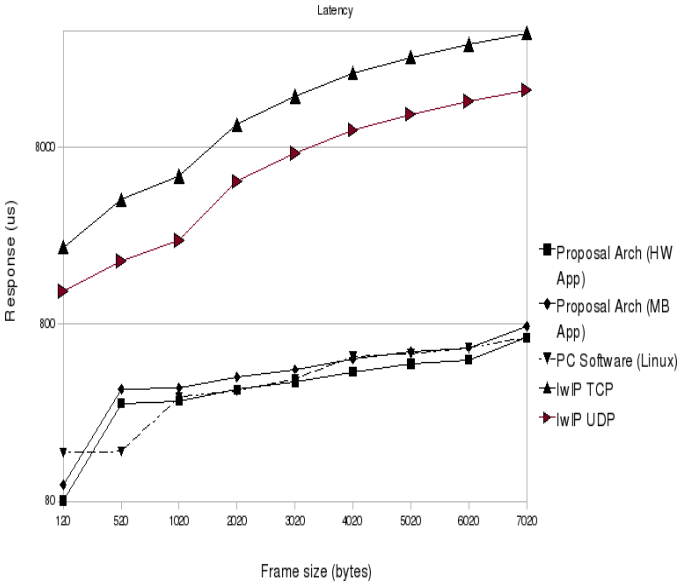

Figure 4: Proposal network stack architecture versus different network stacks' latencies

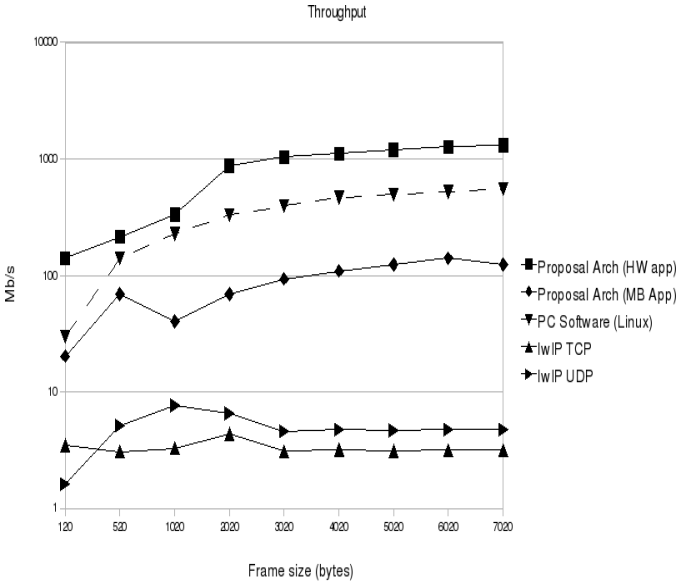

Figure 5: Proposal network stack architecture versus different network stacks' throughput results

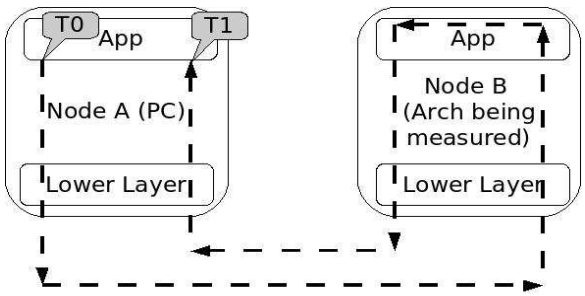

Figure 6: Benchmark environment

throughput, the best result provided by our architecture has been obtained using 7020-byte jumbo frames, when it is possible to transfer more 759.77 Mb/s than software implementation. In this configuration, our hardware is 2.38 times better than the software implementation. Also, comparing our architecture with lwIP implementation we had 27 times more throughput in TCP and 18 times on UDP.

This behavior, where throughput results are much better than latency results, is due our architecture pipeline is being fully functional most part of the time. Also, the buffer utilization allows an almost constant data rate between buffer and Gemac since we fill the buffer with 32 bits words at each 125MHz cycle and the buffer outputs 8 bits words to Gemac at the same frequency.

5. CONCLUSIONS

This paper presented a proposal of a network stack implemented in FPGA. The architecture has been implemented on an ML402 board and it's using around 13% (4,203 LUTs) of a Xilinx Virtex 4 SX35 FPGA. Its maximum frequency of operation is 128MHz, which satisfies the requirements for Gigabit Ethernet communication.

Our architecture showed, in the best case, a throughput performance of 4.7 times better in the best case and 2.5 times better in average compared with pure software implementation. Also, comparing with lwIP network stack it shows 44 times more throughput in our best case. Considering that in all frame sizes our architecture shows very best results than lwIP we could consider our proposal network stack a good solution to provide high bandwidth to embedded systems that pursue simple microprocessors.

6. ACKNOWLEDGEMENT

The authors would like to thank CNPq and the FINEP/SEBRAE program for supporting the project VoIPWIFI and this work.

7. REFERENCES

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet – a gigabit-per-second local-area network. 1995.

[2] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. 40(5):94–101, 2002.

[3] A. e. a. Dunkels. lwip: A lightweigth tcp/ip stack available at http://savannah.nongnu.org/projects/lwip/. 2009.

[4] G. Gilder. TELECOSM: How Infinite Bandwidth will Revolutionize Our World. 2001.

[5] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin. Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis. In Proc. 13th Symposium on High Performance Interconnects, pages 128–137, 2005.

[6] A. Romanow, J. Mogul, T. Talpey, and S. Bailey. Rfc 4297 - remote direct memory access (rdma) over ip problem statement. IETF, 2005.

[7] M. J. S. Smith. Gigabit ethernet and transport offload: transport offload engines help relieve tcp processing burden for gigabit ethernet. Computer Technology Review, 2002.