Capítulo 2
Técnicas de comutação em redes gigabit*

Vinícius M. Romão, Pedro H. Prudêncio, Rudy M. Braquehais, Joaquim Celestino Jr., Jorge Luiz C. Silva, Marcial P. Fernandez

[1]Laboratório de Redes de Comunicação e Segurança (LARCES)
Universidade Estadual do Ceará (UECE)
Av. Paranjana, 1700 – Itaperi – 60.740-000 – Fortaleza – CE

{vinicius,pedro,rudy,celestino,jlcs,marcial}@larces.uece.br

Abstract. *The progresses in optical technologies have been increased the speeds and capacity of the telecommunication infrastructure. Now the most promising technology to obtain high switching rates is the purely optic switches(OXC), that in spite of the high capacity, it still has a lot of technological limitations. In any way, we will still need electronic switches, at least in the access networks connection. This minicourse presents a study of hardware and software techniques for packet switching and presents the new approachs to reach gigabits rates, necessary for the new optical networks.*

Resumo. *Os últimos avanços das tecnologias óticas tem propiciado o aumento cada vez maior das velocidades e capacidade da infra-estrutura de comunicação. Atualmente a tecnologia mais promissora para se obter altas taxas de comutação são os comutadores puramente óticos (OXC), que apesar da alta capacidade ainda tem muitas limitações. De qualquer forma, ainda vamos necessitar de comutadores eletrônicos, pelo menos na ligação rede de núcleo com rede de acesso. Esse minicurso apresenta um estudo de mecanismos de hardware e software para comutação e apresenta as linhas para se atingir velocidade gigabits, necessárias para as novas redes óticas.*

1. Introdução

Nos últimos anos acompanhamos o desenvolvimento das tecnologias óticas que possibilitaram atingir taxas de transmissão inimagináveis até então. A utilização de laser NLM (Nonlinear Loop Mirror) e a multiplexação por comprimento de onda (WDM) aumentaram a capacidade de transmissão de dados em uma fibra ótica para taxas de Tbps. Da mesma forma, a utilização de amplificadores óticos permitiu o aumento da capacidade pela eliminação da conversão ótico-elétrico-ótico (OEO).

Atualmente o grande desafio é desenvolver técnicas para comutação puramente ótica, com o objetivo de eliminar a conversão OEO nos comutadores. Com isso poderemos dispor de uma infra-estrutura ótica completa para transmissão de dados a velocidades acima de Gbps.

No entanto, os usuários ainda continuam (e continuarão por muito tempo) utilizando interfaces elétricas para se conectar à rede. Por isso, é necessário um comutador

*Esse trabalho foi realizado com recursos da Padtec S/A referentes à Lei nº 8.248 de 23.10.91 (Lei de Informática).

eletrônico capaz de comutar dados a velocidades gigabit. Isso nos coloca em um novo desafio - construir novos equipamentos capazes de velocidades gigabit. As arquiteturas de comutadores atuais ainda utilizam técnicas e algoritmos que não suportam essa taxa de transmissão.

Esse texto é organizado da seguinte forma: a seção 2. apresenta uma visão das arquiteturas de roteadores, a seção 3. apresenta os principais algoritmos de busca em tabela de roteamento, a seção 4. mostra os algoritmos de classificação de pacotes, a seção 5. apresenta o dispositivo Processador de Rede (NP-Network Processors), a seção 6. mostra o dispositivo FPGA (Field Programmed Gate Array), a seção 7. apresenta alguns estudos de casos sobre o uso de NPs e FPGAs para implementar funções de um comutador de pacotes, e, finalmente, a seção 8. apresenta as conclusões e comentários finais.

2. Arquiteturas

A popularidade da Internet tem causado um drástico aumento no tráfego de dados ao redor mundo. Para suportar esse aumento, as taxas de transmissão tem sido aumentadas para o nível de gigabit por segundo, exigindo cada vez mais equipamentos mais potentes e rápidos. Muita atenção tem de ser dada as novas arquiteturas desses equipamentos (roteadores) que surgem a cada instante como forma de suportar essa demanda de tráfego cada vez mais intensa.

O aumento no tráfego IP começa a saturar os processadores dos equipamentos atuais e traz novos desafios para o desenvolvimento da arquitetura dos roteadores dessa nova geração. Processadores eram tradicionalmente implementados por software. Por esse motivo o desempenho do roteador era limitado pelo desempenho do processador e também pela eficiência dos algoritmos. Para alcançar um roteamento na velocidade das interfaces (*wire-speed*) é necessário alto desempenho de todos os componentes dos comutadores, trazendo novos desafios aos projetistas[Aweya 1997].

2.1. Funcionalidades Básicas de um Roteador IP

Basicamente um roteador é composto pelos seguintes componentes: várias interfaces de rede, uma ou mais unidades de processamento, uma ou mais unidades de armazenamento (buffer) e uma unidade de interconexão interna (barramento) ou *switch fabric*[Baker 1995].

O funcionamento de um roteador é dividido em duas funções distintas: roteamento e encaminhamento de pacotes. O encaminhamento de pacotes é a função de recebimento de pacotes em uma interface de entrada, o armazenamento, a análise das informações (por exemplo, endereços de destino), busca na tabela de roteamento, cálculo do checksum e encaminhamento para a interface de saída. O roteamento de pacotes envolve atividades de suporte ao encaminhamento, como a execução de protocolos de roteamento, por exemplo, OSPF (*Open Shortest Path First*), RIP(*Routing Information Protocol*), etc, coleta das informações de gerenciamento e execução de protocolos de gerenciamento, por exemplo SNMP (*Simple Network Management Protocol*).

2.1.1. Roteamento (*Route Processing*)

A função de roteamento cuida da construção e manutenção da tabela de roteamento através de protocolos como RIP, OSPF ou MPLS(*Multiprotocol Label Switch*), ou ainda

roteamento estático. Outra função importante é a supervisão e controle do equipamento como um todo, através de testes de funcionamento de cada interface, estado das comunicações, temperatura do chassi do equipamento, entre outras. Acompanhando essa supervisão temos os protocolo de gerenciamento (SNMP). A atualização das tabelas de roteamento nas diversas interfaces pode ser realizada pela função de roteamento.

2.1.2. Encaminhamento de pacotes (*Packet Forwarding*)

O encaminhamento de pacotes consiste em enviar um pacote de uma interface de entrada para a interface de saída apropriada. Esta etapa pode ser dividida em:

Validação do pacote IP o roteador deve checar se o pacote IP recebido sofreu algum dano antes que o mesmo seja processado. Isto envolve checagem da versão, checagem do tamanho do cabeçalho e cálculo do checksum.

Busca na tabela o endereço de destino o roteador percorre a tabela de roteamento para determinar a porta de saída que deve direcionar o pacote e para onde este pacote dever ser enviado (*next hop*). Esta definição baseia-se no endereço IP de destino do pacote e nas máscaras de rede associadas às tabelas de entrada. O roteador deve determinar também o mapeamento do endereço de rede de destino para o endereço do enlace e a porta de saída (ARP), quando a interface está ligada a uma rede local.

Classificação de pacotes Um pacote pode precisar de um tratamento diferenciado, assim, é necessário classifica-los conforme regras de comparação do valor de vários campos de um pacote (quando comparamos apenas um campo, por exemplo, endereço IP essa função se transforma em uma simples busca em tabela).

Controle do tempo de vida do pacote o roteador ajusta o tempo de vida (TTL) dos pacotes para prevenir uma circulação interminável e desnecessária dos mesmos. O campo de TTL é decrementado a medida que o pacote é encaminhado. Pacotes com tempos de vida negativos são descartados pelo roteador, podendo ou não ser enviada uma mensagem de erro ao seu destinatário.

Cálculo do Checksum o checksum do cabeçalho IP deve ser recalculado sempre que o campo de TTL for modificado.

De todas essas funções as mais críticas são a busca em tabelas e classificação, principalmente em função da quantidade de entradas e complexidade da decisão. As outras funções são mais simples e não tem grande impacto no tempo total de processamento. Essas funções serão tratadas com mais profundidade nas seções 3. e 4..

2.2. Caminho rápido X Caminho lento (*Fast-Path* X *Slow-Path*)

Para um eficiente processamento de dados pelo roteador, é necessário uma distribuição de tarefas entre seus diversos mecanismos. Isto é feito com o objetivo de encontrar uma disposição adequada das tarefas a serem implementadas no *Fast-Path* e no *Slow-Path* que proporcionem a maior taxa de processamento de pacotes possível.

Assim, pacotes de dados que possuem características de processamento esporádicos devem ser processados no *Slow-Path*, pois, normalmente, os roteadores não precisam processar esses pacotes na velocidade de transmissão da interface. Dessa maneira, não é necessário manter um código de tratamento de pacotes em uma interface,

a qual não dispõe de vastos recursos de memória, para tratar condições de pequena incidência.

Para que um roteador encaminhe pacotes na velocidade de transmissão da interface (*wire speed*), o tempo que o processador de rede, em uma interface, dispõe para processar um pacote é intervalo de chegada entre dois pacotes consecutivos. Para tanto, as tarefas que compõem o *Fast-Path* devem ser pequenas e otimizadas.

Rotinas que demandem maior tempo de execução no *Fast-Path* fazem com que o tempo gasto no processamento por pacote aumente. Isso compromete a valor da taxa de pacotes processados por segundo.

Dessa forma, duas abordagens podem ser adotadas na implementação desta rotina para que não haja prejuízos no desempenho do roteador. A primeira, no caso de desejar-se implementá-la em *Fast-Path*, é buscar uma arquitetura que ofereça um co-processador que auxilie a execução das atividades desta rotina. Isto reduziria seu tempo de execução e viabilizaria sua implementação. A segunda abordagem é levar a rotina para o *Slow-Path*. Alguns exemplos de tarefas e protocolos implementados em *Fast-Path* e *Slow-Path*, bem como as justificativas, são apresentados em [Aweya 1997]

2.3. Arquiteturas

Apresentamos na figura 1 duas arquiteturas básicas de roteadores: memória compartilhada e múltiplos processadores. A arquitetura de memória compartilhada, mostrada na figura 1(a), qualquer encaminhamento de pacote usa a memória comum como buffer intermediário, causando congestionamento no barramento compartilhado.

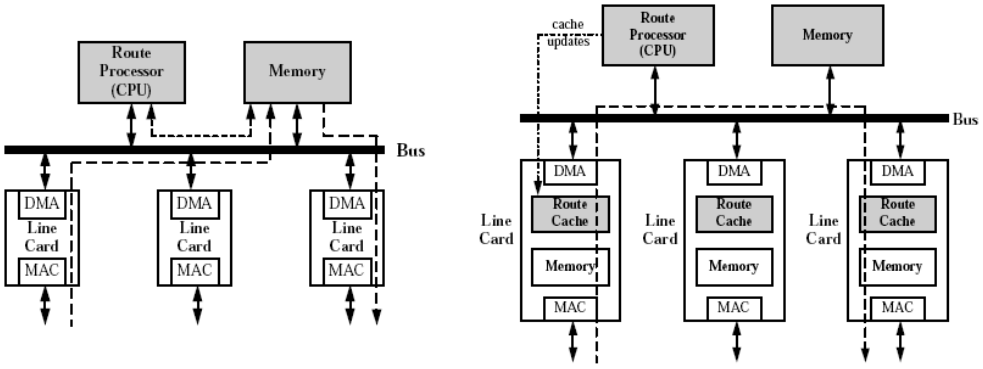
A arquitetura de múltiplos processadores, mostrada na figura 1(b), não utiliza memória compartilhada. Cada interface dispõe de uma memória cache local e os pacotes são encaminhados diretamente para a interface de saída, sem passar pela memória comum. Nessa arquitetura as funções de roteamento e encaminhamento podem ser alocadas a processadores diferentes e o tráfego de dados é significativamente menor. Todas as funções de encaminhamento são executadas nas interfaces e o processador central executa apenas as funções de roteamento. Sendo assim, a única comunicação entre o processador central e as interfaces é a atualização das tabelas de roteamento/encaminhamento.

Mesmo com o aumento da capacidade da arquitetura de múltiplos processadores, ainda há uma limitação de desempenho devido ao barramento compartilhado. Para isso, foi desenvolvida a arquitetura *crossbar*, que consiste em múltiplos barramentos que possibilitam conexão exclusiva entre todos os pares de interfaces (matriz malha completa)

3. Busca em Tabelas de Roteamento (IP-Lookup)

Para que um roteador possa repassar os pacotes recebidos, faz-se necessário que saiba para onde encaminhá-los. As informações de encaminhamento são coletadas através de um protocolo de roteamento e guardadas em uma **tabela de encaminhamento** (*Forwarding Table*, tabela 1). Para cada pacote recebido, o roteador deverá consultá-la para obter o endereço do próximo roteador ao qual será enviado o pacote (*next-hop*). Esta tarefa é denominada *IP-Lookup*.

As tabelas de encaminhamento nos roteadores de borda cresceram exponencialmente nos últimos anos. Atualmente, o número de entradas na tabela é da ordem de



(a) Barramento Compartilhado (b) Barramento Múltiplos Processadores

Figura 1. Arquiteturas de um roteador

Tabela 1. Tabela de encaminhamento

Prefixo de destino	Next-Hop	Porta de saída
24.40.32/20	192.41.177.148	C
130.86/16	192.41.177.181	D
208.12.16/20	192.41.177.241	B
208.12.21/24	192.41.177.196	A
167.24.103/24	192.41.177.3	B

250.000 (Figura 4), o que torna o *IP-Lookup* um fator limitante para a velocidade do encaminhamento[Varghese 2005].

3.1. Arquitetura de endereçamento da internet

Apresentamos nesta seção um resumo sobre os esquemas de endereçamento na Internet.

3.1.1. Classfull Adressing Scheme (esquema de endereçamento por classes)

Antes de 1993, o esquema de endereçamento utilizado na internet era o **Esquema de Endereçamento por Classes** (*Classfull Adressing Scheme*). Esse esquema usa uma hierarquia simples de dois níveis. As redes são o nível mais alto e os hosts o nível mais baixo. Isso é baseado no fato de que o IP consiste em duas partes: a parte que representa a rede (*netid*) e a parte que representa o host (*hostid*). A figura 2, mostra a divisão de classes, bem como os tamanhos do **netid** e **hostid** em cada classe.

Devido ao *netid* iniciar o endereço IP, ele também é conhecido como prefixo de endereço (*address prefix*). Os prefixos de endereço podem ser representados por uma seqüência de bits seguida de um asterisco. Por exemplo, $1001101101100000*$ representa todos os endereços que iniciam com 1001101100000. Outra maneira é representá-los usando a notação **decimais com ponto** (*dotted-decimal*). Então, o mesmo prefixo poderia ser representado por $155.96/16$, onde 16 é o tamanho do prefixo de endereço de rede. Durante o início da Internet, esse esquema funcionava bem, mas com o seu crescimento (figura 3), as tabelas aumentaram muito de tamanho[Gupta 2000].

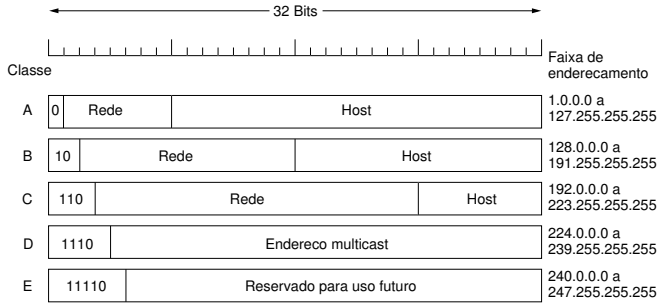


Figura 2. Distribuição das faixas de IP

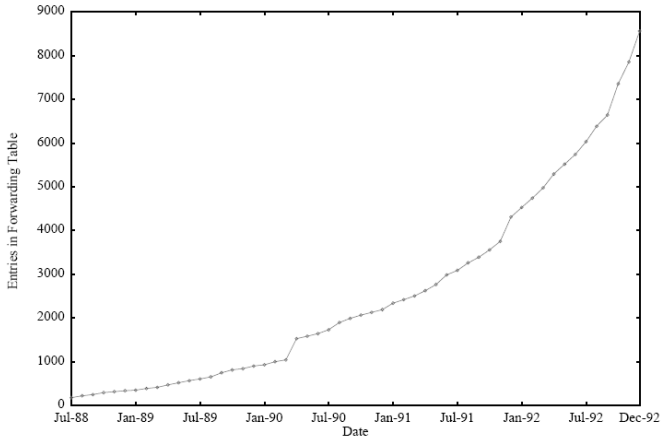


Figura 3. Crescimento das tabelas durante o uso do Classfull Adressing Scheme

Esse esquema de endereçamento era muito ineficiente. Uma rede da Classe A dificilmente preencheria todos os 2 milhões de hosts disponíveis. O tamanho das redes de Classe B também era grande, 65000 hosts. Já os da classe C eram muito pequenos para maioria dos casos. Para resolver esse problema foi criado uma nova arquitetura de endereçamento chamada CIDR (*Classless Interdomain Routing*), que é utilizada atualmente. Durante a época de utilização desse esquema de endereçamento, a tarefa de *table lookup* era relativamente simples, que é apresentada na seção 3.2..

3.1.2. CIDR Esquema de Endereçamento sem Classes (*Classless Interdomain Routing*)

O *Classfull Adressing Scheme* adotava tamanhos de prefixo 8, 16 e 24. No CIDR, o tamanho dos prefixos é arbitrário para aproveitar melhor o espaçamento. Assim, o número de *hostids* pode se ajustar melhor ao o tamanho da rede, evitando-se o desperdício de endereços. O uso do CIDR não impediu o aumento do número de hosts na internet mas permitiu uma maior flexibilidade na utilização dos endereços disponíveis. O CIDR, no entanto, não resolve o problema de exaustão de endereços IP, no qual uma das soluções é a migração para o IPv6. O topo do gráfico na figura 4 mostra o número de entradas nas tabelas de roteamento em função do tempo[Huston 2006].

Durante o uso do *Classfull Adressing Scheme* era simples determinar o tamanho

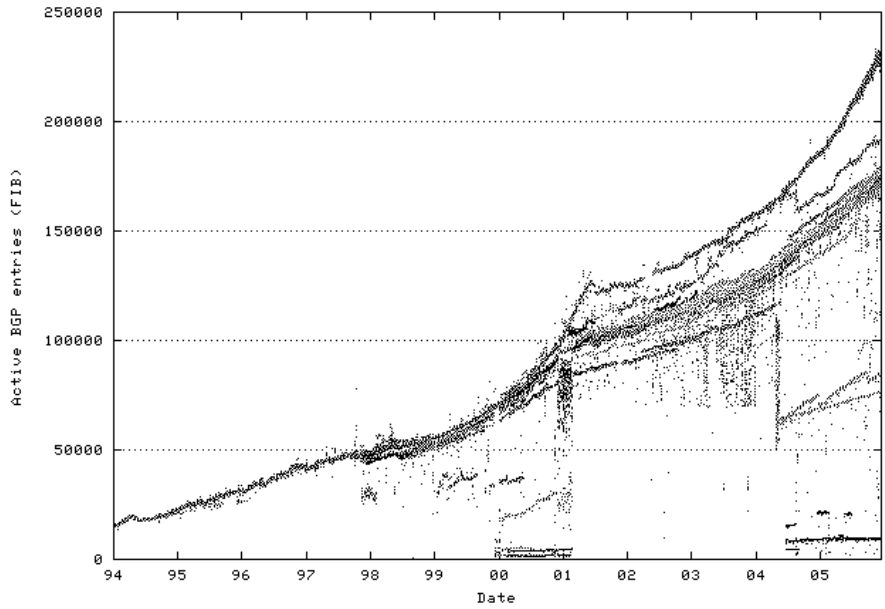


Figura 4. Crescimento das tabelas após o surgimento do CIDR

do prefixo de um IP. Bastava-se verificar os primeiros bits para determinar a qual classe ele pertencia, e conseqüentemente, separar o *netid* e *hostid*. No CIDR, não há como determinar o tamanho do prefixo sem se fazer uma busca na tabela por um prefixo compatível. O problema é que pode existir mais de um prefixo válido para um mesmo IP. Nesses casos o maior deles prevalece e o *next-hop* é aquele associado a tal prefixo. Então, mesmo que seja encontrado um prefixo válido no início da tabela, é necessário ir até o final dela, pois não se sabe se haverá um prefixo maior. Esta busca por um prefixo válido de maior tamanho possível é denominada *Best Match Lookup* e é apresentada na seção 3.3..

3.2. Busca em Tabela por Prefixo Exato (*Exact Match Lookup*)

É a forma mais simples de busca em tabela, onde uma entrada é exatamente igual ao IP de destino do pacote. Ela é realizada quando se organiza a tabela usando o *Classfull Adressing Scheme* em tabelas com poucas entradas, como por exemplo em pontes(*bridges*). As duas maneiras mais importantes de se realizar um *exact-match lookup* é o uso de *hashing* ou busca binária[Varghese 2005].

3.2.1. *Hashing*

A técnica de *hashing*(figura 5) é a mais utilizada para *Exact Match Lookups*[Gupta 2000]. É muito eficiente, se consegue uma busca de apenas um acesso a memória na maioria dos casos. Porém, em alguns casos especiais onde ocorre **colisão** (vários prefixos coincidem no mesmo lugar da tabela) ela pode se tornar ineficiente.

Uma solução para melhoria são as funções de *hashing* parametrizáveis. A cada novo prefixo inserido na tabela, novos parâmetros são calculados de modo a tornar o *hashing* homogêneo, fazendo com que o pior caso tenha em torno de 4 acessos a memória. A desvantagem é que, apesar de improvável, o tempo para achar os parâmetros da função de *hashing* pode ser muito grande.

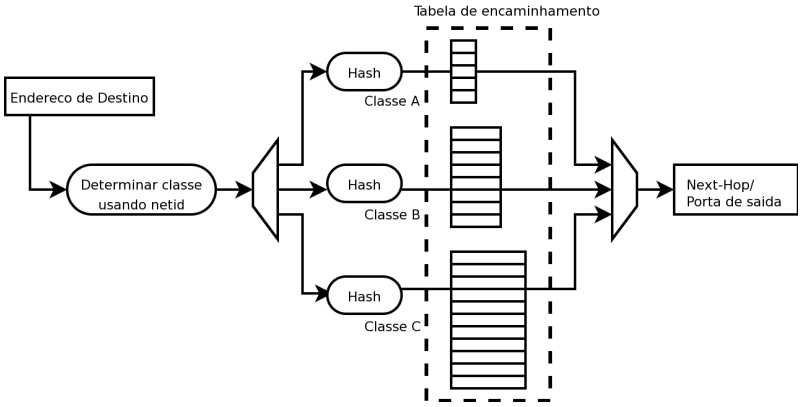


Figura 5. Busca em tabela usando-se *hashing*

3.2.2. Busca Binária

O problema com o *hashing* é que não se tem certeza sobre o pior caso. Outra solução interessante para os *Exact Match Lookups* é o uso de uma busca binária. No entanto, sua complexidade é logarítmica[Cormen et al. 2001], o que a torna muito ineficiente. Em uma tabela com 128 entradas serão necessários 8 acessos na memória para se encontrar um *next-hop*.

Esta técnica pode ser melhorada usando-se um paralelismo em hardware, mas ainda assim a busca é bem mais lenta do que usando *hashing*. A vantagem em relação a técnica anterior é o menor tempo de atualização da tabela.

3.3. Best Match Lookup

Para a busca em tabela para o esquema de endereçamento *Classless Interdomain Routing* é necessário um processo mais complexo que o anterior. Para um determinado endereço de destino poderão existir mais de duas entradas válidas na tabela e neste caso devemos escolher a entrada com o **maior prefixo válido** (*best matching prefix*, BMP). Apresentamos em seguida as várias categorias de *best-match lookups*[Gupta 2000, Ruiz-Sanchez et al. 2001].

3.3.1. Árvores

A maneira clássica de se representar as tabelas de encaminhamento é usando árvores. Vários métodos de *lookup* se baseiam nesta idéia, dentre eles estão: *unibit tries*, *multibit tries*, *level-compressed tries*, *lulea-compressed tries* e *path-compressed tries*. Todos eles são facilmente implementados em hardware. O termo *trie* vem de *information re**trie**val*. Uma trie[Skiena 1997] é definida como um tipo especial de árvore em que se guarda *strings*, que neste caso, são sequencias de bits.

As *Binary tries*, também conhecidas como *unibit tries*, podem ser entendidas como sendo árvores binárias. Um exemplo de *binary trie* é apresentado na figura 6. Ao se procurar por endereços nesse tipo de tabela, percorre-se as arestas de acordo com o bits do IP equivalente aos níveis da árvore. O endereço do *next-hop* que estiver no nível mais profundo da tabela durante o percurso será o endereço ao qual será enviado o pacote.

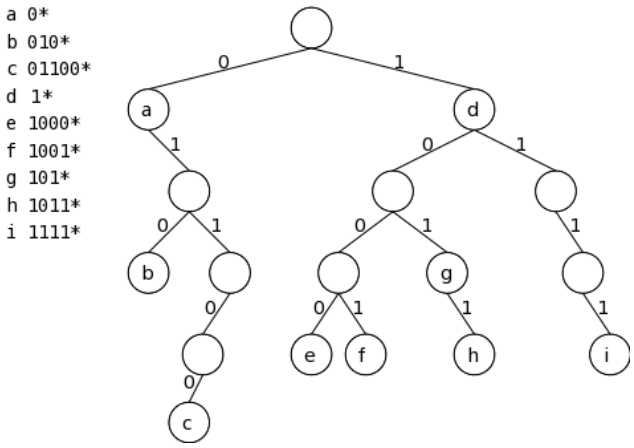


Figura 6. Binary Trie

Por exemplo, ao se procurar pelo *next-hop* de um IP iniciado por 1010 na tabela da figura 6, a busca é iniciada a partir da raiz. Como o primeiro bit é 1, a aresta percorrida é a da direita. É encontrado um nó com o *next-hop* "d" que é guardado como o mais profundo da tabela. A busca continua, como o segundo bit 0, a aresta da esquerda é escolhida. Terceiro bit 1, escolhemos a aresta da direita, onde é encontrado um nó com o *Next-Hop* "g", que passa a ser o mais profundo da tabela. O quarto bit é 0, porém como não há aresta para a esquerda, a busca é finalizada. No final da busca o endereço escolhido será o último *Next-Hop* encontrado, neste caso, o "g".

Comparada as outras estruturas de dados apresentadas nesse artigo, ela é a mais lenta para a busca. A cada nó atravessado, tem-se um acesso a memória. Sua complexidade é $O(W)$, onde W é o número de bits do IP. No caso do IPv4, tem-se 32 acessos a memória no pior caso.

A atualização de uma *Binary Trie* ocorre semelhantemente a uma busca, a diferença é que quando não se encontra uma aresta equivalente ao IP, são adicionadas arestas até que se tenha um caminho equivalente ao prefixo.

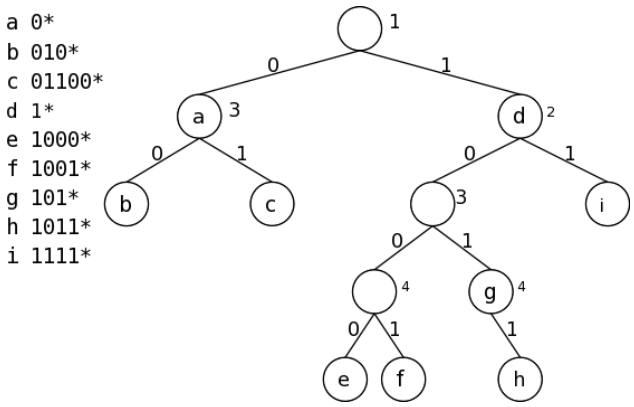


Figura 7. Path-Compressed Trie

Um dos problemas do esquema baseado em *binary tries* é que em alguns passos da busca é desperdiçado um acesso a memória sem que seja feita alguma decisão importante (nós com apenas um filho). Para resolver esse problema, é feita uma **compressão de**

caminho (*path-compression*). Nós com apenas um filho são eliminados. Ao se fazer isso deve ser adicionada uma nova informação em cada nó: o próximo bit a ser inspecionado. Do contrário, não se saberia qual bit inspecionar quando se chegasse aquele nó. As tabelas que se utilizam esse recurso são denominadas *path-compressed tries*. Um exemplo deste tipo de *trie* é mostrado na figura 7.

As *path-compressed tries* são baseadas no algoritmo PATRICIA (*radix trie*), inicialmente proposto por [Morrison 1968]. Uma implementação muito conhecida das *path-compressed tries* é a *BSD-trie*, que difere apenas no fato de que a busca na tabela ocorre até uma folha, depois retornando até se alcançar o *Next-Hop* mais profundo da tabela, o que não altera a complexidade do algoritmo.

Assim como nas *binary tries* ainda se tem um pior caso de busca com 32 acessos a memória(IPv4), porém, em média, o número de acessos é menor.

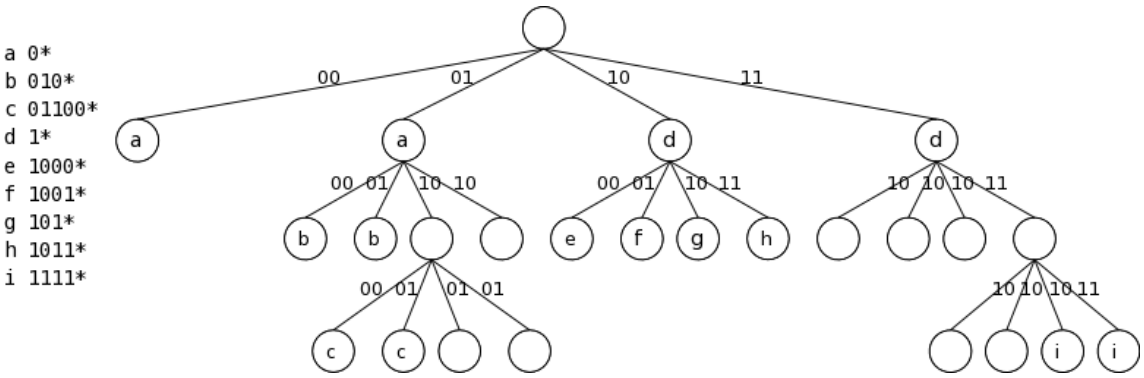


Figura 8. Multibit Trie

Outro tipo de esquema semelhante ao das *binary tries* é o das *multibit tries*. A diferença é que durante a busca inspeciona-se mais de um bit por vez, conseqüentemente pode-se ter mais de dois caminhos para escolher a cada decisão. O termo que denomina o número de bits a serem inspecionados a cada escolha é chamado *stride*, que pode ser fixo ou variável. A Figura 8 exemplifica uma tabela de *stride* fixo. A complexidade da busca nesse esquema é $O(W/k)$, onde W é o tamanho do IP, e k é o tamanho dos *strides*. A complexidade da atualização é $O(W/k + 2^k)$.

Outra variação das *multibit-tries* bastante conhecida são as *lulea-compressed tries*[Brodnik et al. 1997]. Ela utiliza *strides* de tamanho 16, 8 e 8. Para diminuir o tamanho de armazenamento da tabela, não é permitido que os *next-hops* sejam guardados em nós internos da tabela e é feita uma compactação usando um algoritmo especial. O tempo de busca é pequeno, de 3 acessos a memória (3 níveis da árvore). Já o tempo de atualização é bem grande.

O uso das *multibit tries* é bastante eficiente em termos de *best match lookup*. Porém, não é escalável, pois com a introdução do IPv6 (128 bits) haveria uma perda grande no tempo de busca, já que ele é dado em função do número de bits do IP.

O *Level-compressed Trie*[Nilsson and Karlsson 1999] é um tipo especial de árvore baseada nas *multibit tries* que usa *path-compression*. Ele está entre os mais eficientes e seu funcionamento é bastante simples. Primeiro inicia-se com uma *binary trie*, e sempre que aparece uma sub-árvore quase completa ela é transformada em *multibit trie*. Quando

há nós com apenas um filho é feita uma compressão de caminho.

3.3.2. Busca a Partir dos Tamanhos (*Search on prefix lengths*)

Inicialmente proposta em [Waldvogel et al. 1997] com nome de *Search Of Hash Tables*. Seu funcionamento se baseia na separação dos prefixos em várias tabelas *hash* conforme o tamanho. Então pode-se procurar nestas tabelas através de uma busca linear ou binária.

A busca linear ocorre procurando-se por prefixos válidos primeiro nas tabelas *hash* de prefixos maiores, depois nas de prefixos menores. Quando se encontra algum prefixo válido a busca é terminada, já que é garantido que não hajam prefixos maiores dado o sentido da busca. Levando-se em consideração um *hash* perfeito, o tempo de busca é $O(W)$. No caso do IPv4, 32 acessos a memória. Na prática esse número é menor, pois geralmente os prefixos são grandes, próximos de 32 bits. As tabelas com os prefixos menores raramente são acessadas.

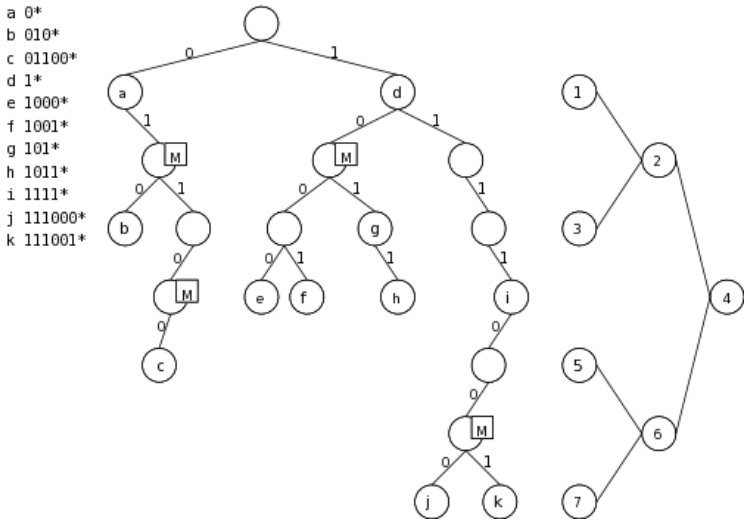


Figura 9. Binary Search On Prefix Lengths

Uma estratégia de busca mais apropriada seria usar uma busca binária nas tabelas *hash* (*Binary Search On Prefix Lenghts*). O problema é que em alguns passos da busca (nível 2 da figura 9) não se sabe se deve-se seguir na direção dos prefixos menores ou dos maiores, pois não é certo se será encontrado um prefixo válido nesta direção. Se fosse escolhida a direção dos maiores e nenhum prefixo fosse encontrado teria que ser feito um *backtracking*. Para evitar o *backtracking* são inseridos **marcadores** (*markers*) para indicar que vai haver um prefixo válido na direção dos maiores. Na figura 9 por motivos didáticos os prefixos estão representados através de uma árvore, entenda cada nível da árvore como uma tabela *hash*. O cálculo para a inserção de marcadores torna a complexidade da atualização da tabela um pouco lenta, $O(Nlog_2w)$.

Usando os marcadores a busca binária ocorre da seguinte forma: se há um prefixo válido ou um marcador, a direção escolhida é a dos prefixos maiores; se não, a direção escolhida é a dos prefixos menores. Tem-se assim um método de busca bastante eficiente de complexidade $O(log_2w)$. Esse método é bastante escalável, com IPv4 temos 5 acessos a memória e com IPv6 temos 7 acessos.

3.3.3. Busca por Intervalos (*Search on ranges*)

Nessa técnica cada prefixo é representado através de um intervalo de endereços, então busca-se pelo menor intervalo que enclausura o endereço de destino do pacote. Dentre as formas de busca por intervalos estão: *binary range search*, *multiway range trees* e *multiway range search*[Lampson et al. 1999, Ruiz-Sanchez et al. 2001].

4. Classificação de Pacotes

Durante o início da internet os pacotes eram processados na ordem em que chegavam no roteador (*first-come-first-served*). A diferenciação de tratamento dos pacotes era feita unicamente de acordo com seus endereços de destino.

Um classificador é um conjunto de regras, isto é, ações a serem aplicadas aos pacotes que seguirem determinados pré-requisitos. Um exemplo de classificador é visto na tabela 2. No caso de um pacote coincidir com mais de duas regras, a que tiver maior prioridade é a que prevalece. Na tabela 2, por convenção: $pri(R_n) > pri(R_{n+1})$.

Tabela 2. Classificador

Regra	Prefixo de origem	Prefixo de destino	Porta	Protocolo (transporte)	Ação
R_1	*	192.168.1.85/32	22	tcp	Negar
R_2	157.88.33.7/32	*	80 (http)	tcp	Negar
R_3	160.200.30/24	212.32/16	intervalo 30-40	udp	Permitir
R_4	123.255.32/20	123.254/16	80	*	Permitir
R_5	160.200.30/24	212.32/16	intervalo 20-80	udp	Negar
R_6	*	192.168/16	8080	*	Negar
R_7	123.255.32/20	123.254/16	*	*	Negar
R_8	*	*	*	*	Permitir

Tabela 3. Exemplo de classificação de alguns pacotes usando o classificador da tabela 2

Pacote	Endereço de origem	Endereço de destino	Porta	Protocolo (transporte)	Regra aplicada
1	160.200.30.11	212.32.132.50	35	udp	C, permitir
2	192.168.1.3	192.168.1.85	22	tcp	A, negar
3	157.88.33.7	192.168.1.85	80	tcp	B, negar

Vários autores [Gupta 2000, Varghese 2005] consideram o IP-Lookup (seção 3.) como um caso particular de classificação na qual se considera apenas um campo (endereço de destino). A classificação é uma tarefa difícil, pois ela ocorre baseando-se em vários campos (dimensões). A cada dimensão adicionada, aumenta-se sua complexidade. Outro problema é que algumas dimensões são avaliadas através de intervalos, portanto mais de uma regra pode ser aplicada a um mesmo pacote. Os classificadores vem crescendo junto com o aumento de serviços e número de usuários da Internet.

Devido as dificuldades apresentadas anteriormente, até hoje não se encontrou um esquema algoritmicamente eficiente para a classificação de pacotes. Porém, os classificadores do "mundo real" apresentam uma estrutura especial. Ela consiste em: geralmente não há especificações de intervalos, e as que existem podem ser facilmente convertidas em vários prefixos com um baixo custo de memória[Srinivasan et al. 1998]; dificilmente há interseção entre intervalos; prefixos geralmente são grandes, isto é, se aproximam do tamanho máximo do campo, o que diminui as interseções. Esta estrutura permite que

esquemas baseados em heurísticas sejam bastante eficientes. Este tópico visa resumir as técnicas de classificação de pacotes mais utilizadas atualmente[Varghese 2005].

4.1. Técnicas Simples

Essas técnicas possuem algumas limitações em seu uso, porém ainda hoje são utilizadas, dada sua simplicidade de implementação. São apresentadas por referência. Dentre elas estão a Busca Linear, o uso de CAMs (*Content Adressable Memories*), o algoritmo *PathFinder* e o *Caching*.

4.1.1. Busca linear

A técnica mais simples para a classificação de pacotes é uma busca linear. Uma lista ligada de regras ordenadas de acordo com sua prioridade. Este esquema é claramente ineficiente, tendo a complexidade da consulta em $O(N)$.

4.1.2. Ternary CAM

CAM é um tipo de memória em que se pode fazer consultas em um tempo praticamente constante. É passado para CAM um vetor de bits, então é feita uma busca em todas as posições da memória simultaneamente. Classicamente as CAMs só permitem buscas de uma seqüencia definida de bits (Ex: 1011101). Um tipo especial de CAM, ternary CAM, é mais apropriado para classificação de pacotes, pois ele permite procurar por uma string de bits que contenha valores indefinidos, *"don't care bits"* (Ex: 1*11*01), o que é importante na classificação, já que podemos ter campos indefinidos.

O uso de CAMs é uma técnica simples e eficiente para classificação de pacotes. Sua desvantagem é o custo alto, que torna sua capacidade de armazenamento limitada.

4.1.3. PathFinder

Outra técnica bastante eficiente que pode ser utilizada é o uso do algoritmo PathFinder[Bailey et al. 1994]. Este algoritmo usa a idéia de árvores e não efetua *backtraking*. Para isso ele estabelece uma limitação: não podem haver chaves indefinidas em dimensões intermediários da tabela. No caso do classificador da tabela 2, apenas as regras R_4, R_7, R_8 seriam classificáveis.

4.1.4. Caching

Uma técnica muito conhecida e usada em várias áreas da computação é o *caching*, que consiste em guardar os cabeçalhos dos últimos pacotes recebidos em uma tabela (*cache*). Estudos feitos por [Newman et al. 1997] mostram que a taxa de ocorrência dos cabeçalhos dos pacotes recebidos no *cache* varia entre 50% e 90%. O uso desta técnica pode vir a melhorar a eficiência dos outros vários esquemas de classificação, aqui apresentados.

4.2. Adaptações para Geometria

O problema de classificação de pacotes pode ser adaptado para geometria. Cada campo da classificação se torna uma dimensão do espaço(figura 10), então o problema se reduz em

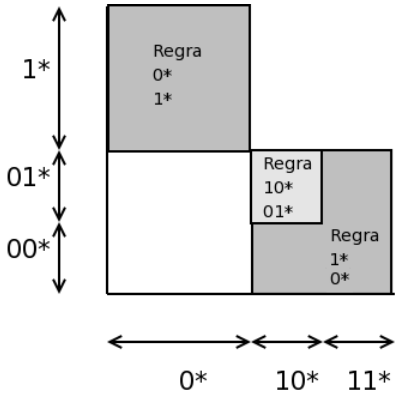


Figura 10. Visão geométrica da classificação

encontrar o local do espaço em que o pacote se encontra. Algoritmos baseados em geometria funcionam bem em classificações de duas dimensões[Lakshman and Stiliadis 1998, Gupta and McKeown 2001].

A visão geométrica da classificação também serve como base para algoritmos de outras classes, como é o caso do *HiCuts*[Gupta and McKeown 1999] e *HyperCuts*[Singh et al. 2003], que são abordados na seção 4.3.4..

4.3. Árvores

Generalizações dos algoritmos de árvore utilizados no IP *Lookup* são bastante utilizadas para classificação de pacotes. As mais usadas são as *Set Pruning Tries*[Qiu et al. 2001]. Outros algoritmos de árvore de decisão[Woo 2000, Gupta and McKeown 1999, Singh et al. 2003] foram criados a partir da interpretação geométrica da classificação. Esta seção resume os principais deles.

Tabela 4. Classificador de 2 dimensões

Regra	Endereço de destino	Endereço de origem
R_1	*	0*
R_2	*	00*
R_3	*	01*
R_4	0*	0*
R_5	0*	01*
R_6	01*	11*
R_7	11*	01*

4.3.1. Set Pruning Tries

As *Set-Pruning Tries* são baseadas em uma idéia interessante: uma árvore de árvores. Esta estrutura de dados é construída a partir de uma árvore binária (*binary trie*) de endereços de destino a partir das regras do classificador. Cada nó desta árvore aponta para uma árvore de endereços de origem. As árvores de endereços de origem, por sua vez, vão conter apenas as regras (e todas elas) que batem com os endereços de destino de seu respectivo nó (figura 11).

A busca nesta árvore é feita percorrendo a árvore de endereços de destino como em um *IP-Lookup*. O diferencial é que quando um nó é encontrado passa-se para a árvore

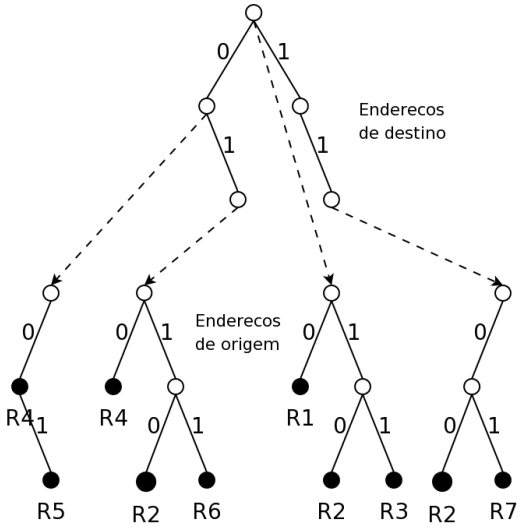


Figura 11. Set Pruning Trie do classificador da tabela 4

de endereços de destino para que a busca continue. O tempo de busca nesta árvore leva $O(dW)$, onde d é o número de dimensões da busca. A quantidade de memória gasta é $O(N^d dW)$, dado que uma regra, pode ser armazenada várias vezes.

4.3.2. Hierarchical Tries

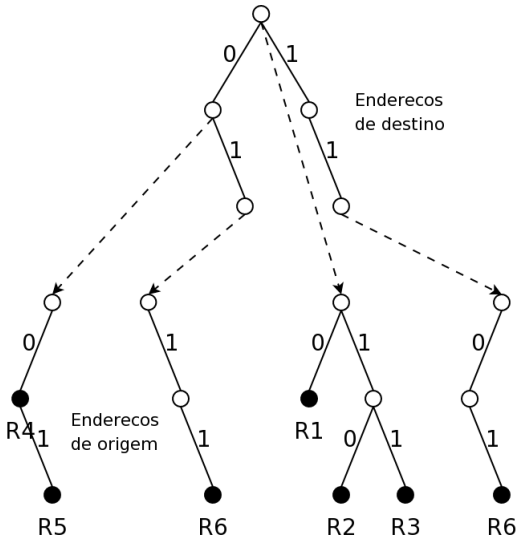


Figura 12. Hierarchical Trie do classificador da tabela 4

Hierarchical Trie é uma estrutura de dados semelhante a anterior, onde se troca tempo de busca por espaço de armazenamento. Nas *Set-Pruning Tries* uma regra pode aparecer mais de uma vez nos nós das árvores de destino, já neste algoritmo não. Sendo assim, como se pode garantir que a regra correta seja encontrada? Um *backtracking*[Cormen et al. 2001] é feito para a árvore original (end. destino), para que a busca continue normalmente. Se forem encontradas outras árvores de endereço de origem o processo se repete. Por causa do *backtracking* feito neste algoritmo o tempo de busca é

exorbitante, $O(W^d)$, em compensação o gasto de memória é bem pequeno, $O(NW)$.

4.3.3. Grid of Tries

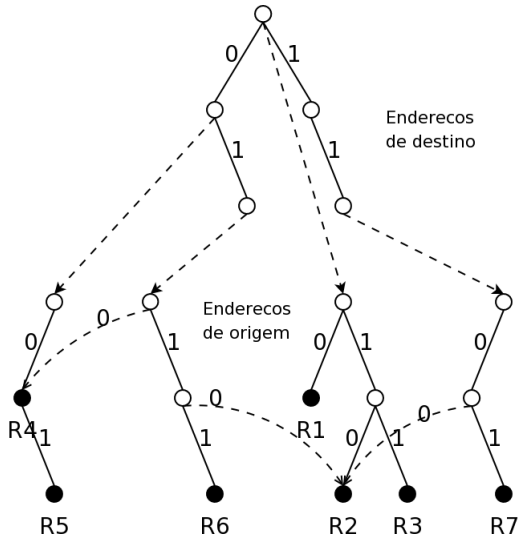


Figura 13. Grid Of Tries do classificador da tabela 4

Apesar do nome a estrutura *Grid of Tries* não é bem uma árvore e sim um grafo. Mais precisamente um **grafo direcionado acíclico**[Cormen et al. 2001]. Esta técnica consegue juntar a idéia de economia de tempo das *Set-Pruning Tries*, não realizando *backtracking*, e de economia de espaço das *Hierarchical Tries*, não duplicando as regras.

Para conseguir alcançar esse objetivos o algoritmo faz alguns cálculos durante a atualização da tabela para evitar informações redundantes nas árvores. No lugar onde haveria repetições de vértices, ou *backtracking*, são adicionados ponteiros para outras árvores de endereços de origem. De modo que se há uma regra válida, ela será encontrada pelo caminho(figura 13). A complexidade da busca e armazenamento é $O(dW)$. No lugar de usar *Binary Tries*, podem ser usadas *Multibit Tries* para melhorar ainda mais a complexidade da busca, $O(dW/k)$. A otimização por *Multibit Tries* também funciona com as *Set-Pruning Tries* e *Hierarchical Tries*.

4.3.4. Árvores de decisão - HiCuts e HyperCuts

Um algoritmo bastante eficiente que foi gerado a partir da visão geométrica da classificação, o HiCuts[Gupta and McKeown 1999] (*Hierarchical Intelligent Cuttings*), funciona transformando cuidadosamente o classificador em uma árvore de decisão a partir de cortes no ”espaço”, gerado pela interpretação geométrica do classificador. Os cortes são paralelos aos eixos do expaço, e são calculados de modo a minimizar o ”custo” da árvore. As folhas desta árvore por sua vez, contém apenas poucas regras, tornando viável uma busca linear.

O algoritmo de HyperCuts[Singh et al. 2003] está para o algoritmo de HiCuts, assim como as *Multibit Tries*, estão para as *Binary Tries*. A melhoria está na forma com que são feitos os cortes.

4.4. Divisão e conquista

Os algoritmos de divisão e conquista são bastante eficientes, principalmente o *Recursive Flow Classification* (RFC). Classifica-se cada campo independentemente e então mescla-se os resultados. Dentre eles: **Bit Vector Linear Search**, **Cross Producting** e **RFC**[Gupta 2000].

4.4.1. Bit Vector Linear Search

Supondo uma busca em apenas um dos campos do classificador da tabela 2. Esta busca elimina todas as regras que **não coincidem** com o cabeçalho do pacote. Fazendo-se isso com todos os campos, sobram bem menos regras no classificador. Se o classificador não é muito grande, apenas uma busca linear resolve o problema de encontrar a regra válida.

Tabela 5. Classificador

Regra	Campo X	Campo Y	Campo Z
R_1	B	C	E
R_2	A	C	E
R_3	B	*	E
R_4	*	*	*

Tabela 6. Bit-Vectors

Campo X	Campo Y	Campo Z
A 0101	C 1111	E 1111
B 1011	* 0011	* 0010
* 0010		

Em classificadores um pouco maiores, pode ser usada uma outra técnica, o *Bit Vector Linear Search*[Lakshman and Stiliadis 1998]. A tabela 6 mostra a representação utilizada pelo *Bit Vector Linear Search* do classificador da tabela 5. O número de bits em cada célula (bitmap) é equivalente ao número de regras do classificador. O n-ésimo bit, é associado a n-ésima regra, e indica se aquela regra bate com o valor associado ao bitmap.

Para encontrar a regra associada a um cabeçalho basta se procurar pelo *best match* de cada campo individualmente. Então, realizando-se um AND nos bitmaps associados a cada *best match* é fácil ver que isto vai retornar um bitmap que mostra todas as regras válidas para um pacote a partir da posição dos bits. A regra associada ao pacote seria a de maior prioridade dentre este grupo. Este esquema foi implementado sem problemas em FPGA[Lakshman and Stiliadis 1998].

4.4.2. Crossproducting

Foi observado que existem bem menos possibilidades de valores nos campos do que especificações de regras em si. Ou seja, em um classificador com 50 regras, podem existir em torno de 5 possibilidades de endereços de origem, 8 possibilidades de endereços de destino e 3 intervalos de portas.

Tomando isto como base, o algoritmo de *Crossproducting*[Srinivasan et al. 1998] funciona pré-computando todas as possibilidades de “misturas” (*crossproducts*) entre os

Tabela 7. Alguns dos *crossproducts* gerados com o classificador da tabela 5

Crossproduct	Regra Válida de maior prioridade
A,C,E	R_2
A,C,*	R_4
A,*,E	R_4
A,*,*	R_4
B,C,E	R_1
B,C,*	R_4
B,*,E	R_3

campos independentes do classificador. A tabela 7 mostra exemplos de vários *crossproducts* do classificador da tabela 5. Para encontrar a regra que se aplica ao pacote a partir dos *Best Matchs* independentes de cada campo, basta achar o *crossproduct* gerado pela combinação deles. A busca neste esquema leva tem complexidade de $O(dW)$. O gasto de memória é $O(N^d)$.

4.4.3. RFC - Recursive Flow Classification

A partir de várias otimizações feitas no *Crossproducting* foi criado o algoritmo RFC (*Recursive Flow Classification*)[Gupta 2000], também conhecido pelo nome de *Equivalenced Crossproducting*[Varghese 2005]. Difere do anterior pela forma de combinar os *crossproducts*, que são combinados aos pares. Há um decréscimo no tempo de busca me relação ao seu antecessor, $O(d)$, enquanto o gasto de memória permanece o mesmo.

5. Processador de Rede (Network Processor)

No início da Internet, boa parte dos equipamentos de redes (roteadores) eram baseados em uma arquitetura semelhante aos PC's. A CPU (unidade central de processamento) gerenciava e executava tarefas de controle das portas de entrada e saída, restrição de acessos, processamento de rotinas de atualização e encaminhamento de pacotes. A gerência dos processos efetuados pela CPU, dependia do sistema operacional presente no roteador, cuja pequena parte das instruções era armazenadas em memória ROM, e a maioria armazenada em memória RAM. Existia, já nessa época, grande facilidade de atualizações e inclusão de funcionalidades ao produto. Essa flexibilidade se devia ao fato de que as instruções/tarefas eram feitas em software, bastando para isso atualizar a versão do sistema operacional. Porém, esses roteadores tinham uma significativa desvantagem: a incapacidade de suportar o aumento da banda de transmissão. É comum notarmos a perda de desempenho, por exemplo, em roteadores ao executar políticas baseadas em estatísticas ou filtragem de tráfego.

O *Switch* representou uma mudança na arquitetura convencional empregada até então, agora ele era capaz de funcionar como uma bridge no reconhecimento dos dispositivos conectados a cada porta, direcionando o tráfego para os dispositivos de destino. O processamento baseado em software não foi eliminado, porém vários avanços tecnológicos permitiram o desenvolvimento de funções específicas e embutidas diretamente em hardware. Muitos equipamentos empregaram então, os *Application Specific Integrated Circuits* (ASICs). Os *switchs* de nível 3 são equipamentos que utilizam dispositivos para encaminhamento de pacotes com tecnologia *ASIC* e gerenciamento baseado em software para controle de funções (tais como cálculo de rotas, caminho mínimo, etc).

Porém essa arquitetura enfrenta alguns problemas, como a dificuldade de introdução de novas funcionalidades e/ou mudanças em arquitetura ASIC. Processadores de rede (NP - Network Processors) foram criados como uma solução para sanar esses problemas, na conseqüente substituição dos ASICs, não programáveis, e diferentemente dos roteadores e switches baseados em software, nos quais uma nova funcionalidade pode ser acrescida com uma simples atualização do sistema operacional.

5.1. Processador de rede (NP - *Network Processor*)

Um Processador de Rede (*network processor*) é um circuito integrado, altamente especializado, modelado para a manipulação do fluxo de dados na velocidade de transmissão (*wire speed*) e reconhecimento/execução dos protocolos de classificação e análise. Diferencia-se dos dispositivos ASIC por ser programável e baseado em software específico, e implementa funcionalidades diretamente em hardware, como um ASIC. Fica situado no caminho dos dados entre a interface física e o dispositivo de comutação, ou seja, nos cartões de interface do sistema, conectado ao cabo através do processador de sinal e internamente através do dispositivo de comutação[Sica and Torres 2005]. Abaixo, descrevemos algumas funções dos processadores de rede.

5.1.1. Segmentação, Montagem e Desmontagem (SAR - *Segmentation, Assembly and Reassembly*)

Os quadros são desmontados, processados e remontados para encaminhamento. Nessa etapa são verificados o enquadramento, atualizadas as informações de segmentação e calculados os códigos de verificação de erros.

5.1.2. Recodificação e classificação de protocolos

Os quadros são identificados com base nas informações do tipo de protocolo, tamanho, destino ou outra informação de aplicação ou protocolo. Pacotes classificados são colocados em filas que são manipuladas pelas estruturas de processamento do roteador. Nesta etapa, a classificação de protocolos pode ser usada também para a determinação da classe de processamento e da qualidade de serviço de um fluxo de dados.

5.1.3. Filas e controle de acesso

Os quadros identificados são colocados em filas especiais para processamento (de prioridade, classificação, segurança etc). Um dos mais desafiadores problemas enfrentados no processamento de rede se encontra no gerenciamento de filas e *buffers* no sistema. Hardware especializado pode ser usado nestas operações a fim de melhorar seu desempenho.

5.1.4. Qualidade de Serviço

Adicionalmente, uma formatação apropriada dos quadros visa obter adequação ao tráfego baseado em QoS. É tratado através do reconhecimento de um conjunto de parâmetros que

denotam a qualidade do fluxo de dados, por exemplo: largura da banda requerida, prioridade e espaço de buffer, eficiência, eficácia, exatidão, alteração e correção de parâmetros. Permite aos provedores de serviço oferecer classes de QoS com custos diferenciados.

5.2. Características básicas de Processadores de rede

Apresentaremos nesta seção algumas características dos processadores de rede: roteamento, gerenciamento de fila, desempenho e programabilidade.

5.2.1. Roteamento

As funções de roteamento são classificadas em duas categorias[Spalink et al. 2000]: classificação e execução.

Classificação é o processo pelo qual o NP examina os dados e determina como deve ser o processamento e o roteamento, baseado em programação de usuário, de acordo com múltiplas regras para pacotes, diferentes protocolos e inúmeros fluxos simultâneos. Esta etapa pode ser dividida em:

Remontagem para a classificação dos pacotes o processador de rede decodifica os dados de acordo com regras programadas pelo usuário, necessitando, às vezes, remontá-lo em virtude da fragmentação dos pacotes no momento da transmissão.

Policiamento o processador de rede deve classificar os pacotes de entrada e adequá-los de acordo com a largura da banda.

Classificação define parâmetros gerais sobre roteamento, encaminhamento.

Estatísticas o processador de rede deve apresentar relatórios estatísticos sobre seu fluxo de dados.

Execução é o processo pelo qual o NP atua sobre o resultado da classificação, de acordo com diversos padrões de gerenciamento da indústria e programados pelo usuário. Esta etapa pode ser dividida em:

Gerenciamento de buffer o processador de rede deve ter um buffer de gerenciamento que permite o dispositivo tomar decisões a respeito dos pacotes.

Moldagem de tráfego dispositivo encarregado do monitoramento do tráfego para o correto encaminhamento de pacotes.

Modificação do fluxo deve haver a possibilidade de efetuar a edição de dados dentro do fluxo, para incluir ou remover cabeçalhos e rodapés, encapsulamento de dados de acordo com protocolos específicos e segmentação de dados em quadros específicos dependendo do meio.

Estatística geração de estatísticas acerca de todo processo de execução.

5.2.2. Gerenciamento de fila

Uma variedade de funções da rede cai sob a categoria genérica de gerência de fila. Estas funções variam desde conjunto e da segmentação do pacote à qualidade do serviço que enfileira-se, onde os dados de ingresso são requisitados novamente na fila do chassi de acordo com a prioridade. Os algoritmos de descarte de dados, classificação e gerenciamento antecedem esta função. Tais funções requerem o *buffering*, onde os dados da rede são armazenados em filas lógicas, e um algoritmo de decisão determina a taxa, a prioridade e o tipo de serviço de manutenção requerido para cada fila.

5.2.3. Desempenho

É crucial o fator de desempenho num processador de rede, tendo em vista as elevadas taxas de larguras de banda existentes atualmente, na faixa de 10 a 40 Gbits/seg. As diversas velocidades são reconhecidas e classificadas pelo dispositivo de processamento do meio físico que repassa ou recebe do processador de rede. Deve-se ressaltar porém, o fato de que o NP é implementado parcialmente em software, existindo assim, soluções ainda mais rígidas, implementadas totalmente em hardware.

5.2.4. Programabilidade

A programabilidade veio com um dos fatores de avanço dos equipamentos de processamento de rede e surgimento dos NPs, que possuem diferentes formas de serem programados. Em geral, as instruções dadas pelo usuário em cima do *fast-path* devem ser as mais concisas possíveis, como forma de minimizar o tempo de leitura e execução de cada instrução e não afetar a velocidade de encaminhamento. O equipamento deve prover ao usuário a capacidade de implementar a adicionar novas aplicações, tarefas, ou até mesmo a aplicação de um novo protocolo, com a simples adição de instruções ou atualização do sistema operacional. Para uma fácil e eficaz manutenção do sistema, deve ser provido a capacidade de adicionar ou remover rotas, conexões, regras, sem que o desempenho e o fluxo de dados do sistema seja afetado.

5.3. Composições de um processador de rede

A típica arquitetura de um processamento de rede é mostrado na figura 14. O tema central por trás do desenvolvimento dos processadores de rede está no emprego de múltiplos pequenos processadores que desempenham funções com largo processamento individual.

Um processador de rede contém diversos processadores de pacotes individuais. Esses, desempenham várias importantes funções nesta arquitetura como: classificação, encaminhamento, comutação e modificação de cabeçalhos.

Esta arquitetura também conta com um processador central de gerenciamento. Ele pode executar funções realocadas e não-críticas de processamento de pacotes. Esse processador também executa função de carregamento de código dos objetos nos processadores de pacotes.

Em adição aos múltiplos núcleos de processamento programável, os processadores de rede podem ainda contar com os co-processadores, que são especializados em funções específicas, tais como: pesquisa na tabela de roteamento, criptografia, gerenciamento e balanceamento de tráfego, etc.

A interface de um processador de rede comunica-se com o CPU através de um barramento PCI ou similar. Também contam com unidades de memória SDRAM/SRAM para implementar tabelas de roteamento (*table lookups*), e alocar *buffer* e *cache*.

5.4. Arquitetura dos Processadores de Rede

A técnica mais utilizada nos processadores de rede para tirar proveito do paralelismo da carga de trabalho, a qual estão submetidos, é o **multiprocessamento**. A maioria dos processadores de rede contém várias unidades de processamento confinadas na própria

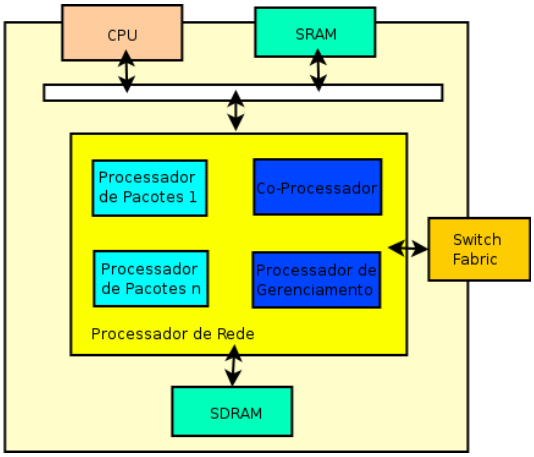
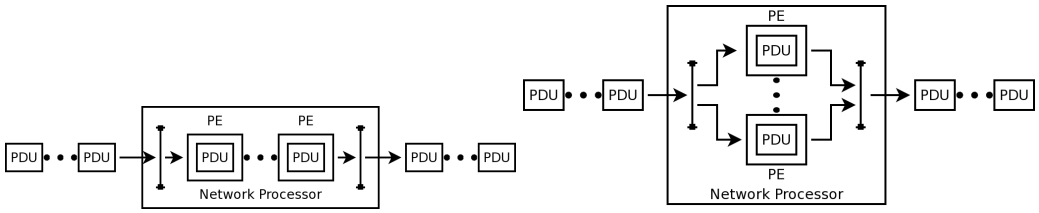


Figura 14. Arquitetura Convencional de um NP

pastilha. Unidade de processamento (UP) é um processador de conjunto de instruções, capaz de decodificar seu próprio fluxo de instruções.

As duas formas mais comuns de organização de unidades de processamento em um NP são em *pipeline* e em paralelo, que determinam o fluxo dos pacotes através do NP. Na organização em **pipeline**, mostrada na figura 15(a), cada unidade de processamento desempenha uma tarefa específica no encaminhamento do pacote. Assim o pacote vai sendo passado por cada UP, que realiza sua tarefa sobre o pacote e encaminha-o para a próxima UP. O processo, visto como um todo, lembra uma linha de montagem, onde cada estágio (UP) realiza sua atividade (tarefa) sobre o produto a ser montado (pacote).



(a) Pipeline (b) Paralelo

Figura 15. Organização de unidades de processamento

Como pode ser observado, existe um PDU em cada unidade de processamento. Assim, se existirem n UPs, existirão n PDUs sendo processadas simultaneamente, porém, cada uma em um estágio diferente. Em geral, esta arquitetura é mais fácil de se programar, uma vez que a comunicação entre os programas, de diferentes unidades de processamento, é restringida pela própria arquitetura[Shah and Keutzer 2002]. Contudo, a dificuldade está em ajustar o tempo de processamento em cada estágio do *pipeline*, pois diferenças de desempenho entre os estágios afetarão o desempenho geral. Por exemplo, baixo desempenho em um estágio causa ociosidade nos estágios posteriores e ociosidade ou adição de estruturas para armazenar os trabalhos recebidos de estágios anteriores.

Na organização em **paralelo** as unidades de processamento desempenham tarefas similares, normalmente executam o mesmo código fonte, independentemente umas

das outras. Cada unidade de processamento recebe um pacote diferente, processa-o e encaminha-o. Diferentemente do que acontecia na organização em *pipeline*, onde o processamento era distribuído em estágios, na organização em paralelo o processamento do pacote é responsabilidade da respectiva unidade de processamento. A figura 15(b) ilustra uma organização em paralelo.

Contrariamente a organização em *pipeline*, esta arquitetura não apresenta o problema de ajuste de tempo de execução de pacote por unidade de processamento, já que todas as unidades executam o mesmo código sobre o seu pacote paralelamente. O tempo de processamento irá depender do conteúdo e da operação a ser realizada sobre o pacote. Apesar de mais flexível esta arquitetura é mais difícil de ser programada, pois existem problemas de concorrência por recursos da máquina entre as unidades de processamento. Assim, mecanismos para arbitração de acesso a recursos compartilhados são necessários. Por não existirem estágios especializados e otimizados na execução de tarefas específicas, como na organização pipeline, esta organização normalmente conta com a presença de co-processadores para acelerar o processamento das UPs.

Alguns processadores de rede permitem que o desenvolvedor configure a organização das suas unidades de processamento em paralelo ou em *pipeline*. Outros, porém, não oferecem esta flexibilidade e possuem organização fixa. Portanto, a equipe de desenvolvimento deve estar atenta na hora de optar pela organização que melhor se adapta a suas necessidades.

5.4.1. Arquitetura da Unidade de Processamento de NP's

Enquanto um processador de uso geral é projetado para executar um programa o mais rápido possível, um processador de rede tem como objetivo processar o maior número possível de unidades de dados de protocolo (PDUs) por unidade de tempo. Portanto, os NPs devem possuir arquiteturas adaptadas à carga de trabalho a qual estão submetidos.

Uma característica básica da carga de trabalho dos processadores de rede é a seguinte: pacotes de dados ou mensagens, que são unidades básicas de trabalhos para estas aplicações, são freqüentemente independentes e podem ser processadas simultaneamente[Crowley et al. 2000]. Este paralelismo ao nível de pacotes deve ser explorado em nível de arquitetura para atingir o desempenho desejado.

Um estudo foi realizado para investigar qual arquitetura de processador responde melhor a carga de trabalho de um roteador. Foram comparadas as seguintes arquiteturas: Superscalar (SS), Fine-Grain Multithread (FGMT), Chip-Multiprocessor (CMP) e Simultaneous Multithread (SMT). Os gráficos da figura 16 mostram o desempenho apresentado por cada arquitetura[Crowley et al. 2000].

As arquiteturas foram submetidas a três avaliações de desempenho. O primeiro, relativo ao encaminhamento de pacotes IP, denominado IP4. O segundo e o terceiro são relativos à algoritmos de criptografia sobre IP, como MD5 e 3DES.

O *benchmark* IP4 realiza busca de endereços IP em uma tabela *lookup*, baseada em uma estrutura de dados em árvore. Este *benchmark* é representativo da categoria de aplicações que realizam processamento apenas sobre o cabeçalho dos pacotes recebidos, ou seja, o tamanho dos pacotes não influência no processamento. Normalmente estas

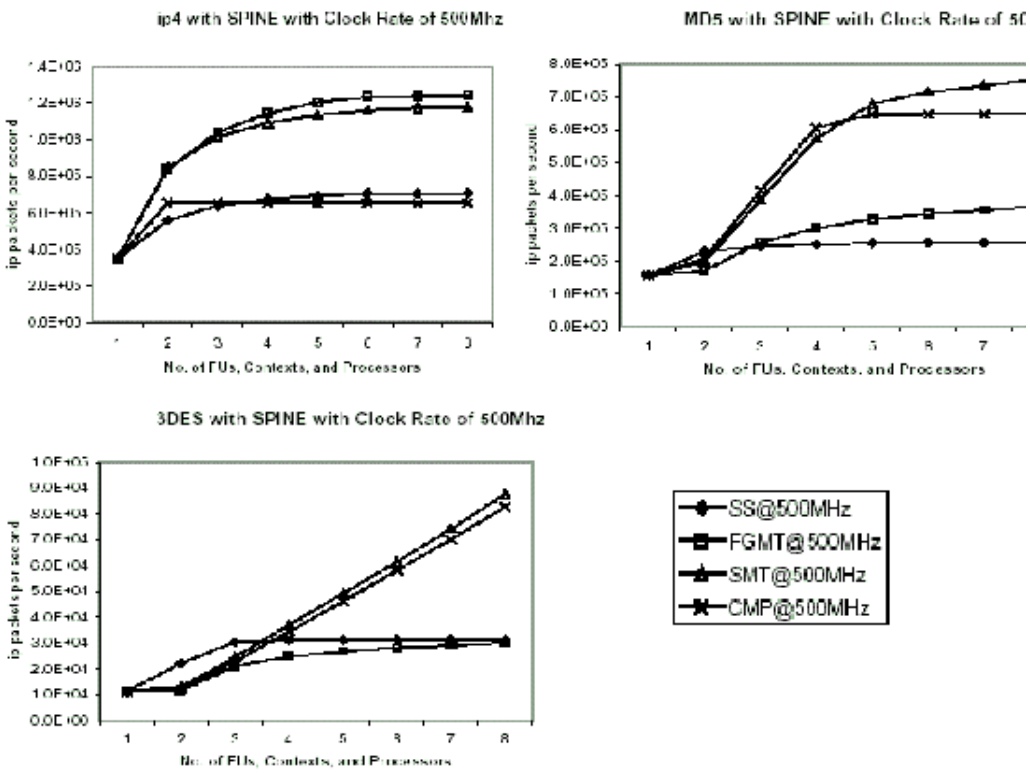


Figura 16. Gráficos de comparativos de *benchmarcks* entre arquiteturas

aplicações mantêm tabelas de informações, em estruturas de dados complexas, que são acessadas a cada pacote recebido.

O *benchmark* MD5 é usado para identificar mensagens. Ele calcula uma assinatura única para cada pacote com base em seu conteúdo. O *benchmark* 3DES é uma rotina de criptografia usada para encriptar o conteúdo do pacote. Estes *benchmarks* são representativos da categoria de aplicações que realizam processamento sobre todo o conteúdo do pacote. Normalmente estas aplicações demandam uma quantidade significativa de capacidade de processamento para despachar pacotes à velocidade de transmissão do cabo. Em geral, nestas aplicações, o processamento de um pacote é independente de qualquer outro, podendo-se processar pacotes em paralelo para atingir melhor desempenho.

Observando os resultados apresentados nos gráficos da figura 16 verifica-se que as arquiteturas CMP e SMT demonstraram-se as mais indicadas para explorar o paralelismo inerente ao processamento de pacotes. Contudo, a arquitetura SMT conseguiu os melhores resultados. Isto se deve ao fato desta arquitetura conseguir combinar duas características fundamentais: paralelismo ao nível de instrução e paralelismo ao nível de *thread*.

Na prática, observa-se que várias arquiteturas mistas de projeto têm sido usadas por fabricantes. Apesar das várias soluções, todas elas buscam explorar o processamento simultâneo de pacotes através de técnicas de processamento paralelo. Assim, arquiteturas CMP e SMT podem ser combinadas, como no INTEL IXP1200, e ainda encontradas ao lado de técnicas de processamento superescalares, como no Agere Swith Router Processor[Systems 2000].

Considerações também devem ser feitas quanto ao conjunto de instruções de processadores e unidades de processamento. Existem processadores de conjunto de instruções complexas (CISC) e processadores de conjunto de instruções reduzidas (RISC).

A principal características dos processadores CISC é o conjunto de instruções com formatos variados[Freitas and Martins 2000]. Dessa forma, cada instrução possui grande flexibilidade quanto aos modos de endereçamento. As principais vantagens desta arquitetura são: a maior flexibilidade e o maior número de facilidades oferecidas ao programador, os programas possuem tamanho reduzido.

A principal característica dos processadores RISC é que as instruções que compõem seu conjunto de instruções são simples e poucas delas referenciam a memória principal.Assim, a maioria das instruções executa operações sobre os registradores. As principais vantagens desta arquitetura são: exige arquitetura de hardware mais simples e o número de ciclos de *clock* gastos por instrução é pequeno.

De um modo geral pode-se dizer que os processadores CISC tentam reduzir o número de instruções para um programa enquanto um RISC tenta reduzir o número de ciclos de *clock* por instrução. Na prática, observa-se que a maioria dos processadores de rede utiliza tecnologia RISC. Eles possuem um conjunto de instruções otimizadas para processamento de pacotes gastando a pequena quantidade de ciclos de *clock* por operação.

5.4.2. Hardware de propósito especial

Uma técnica utilizada para melhorar o desempenho de operações críticas, no processador de rede, é implementá-las em hardware, ao invés de executá-las em software. Normalmente usam-se co-processadores e unidades funcionais especiais para este fim. Co-processadores são blocos computacionais que são disparados por unidades de processamento (isto é, não têm unidade de decodificação de instruções) e computam resultados assincronamente[Shah and Keutzer 2002].

Co-processadores podem ser acessados via memória mapeada, instruções especiais ou barramento. Normalmente executam tarefas complicadas, as quais podem armazenar estados. Também podem acessar diretamente a memória ou barramentos. Por serem entidades mais complexas geralmente são compartilhados entre unidades de processamento.

As tarefas executadas em co-processadores possuem as seguintes características: são tarefas com escopo bem definido, são caras e/ou inadequadas para executar no conjunto de instruções e proibitivamente caras para serem implementadas como unidades funcionais especiais independentes [Shah 2001]. Funções comumente executadas por co-processadores são: busca em tabela de *lookup*, gerenciamento de filas, busca por padrões em pacotes, cálculo de *checksum* e CRC, encriptação e autenticação.

As unidades funcionais especiais são blocos computacionais especializados que computam um resultado dentro de um estágio do *pipeline* ou de uma unidade de processamento[Shah and Keutzer 2002]. A maioria dos processadores de rede tem unidades funcionais para executar operações comuns como: busca por padrões em pacote ou manipulação de bits. As computações desempenhadas por estas operações são fáceis de

serem implementadas em hardware, mas incômodas ou propensa a erros caso seja implementada em software.

O uso de hardware especial para execução de tarefas no processador de rede tem como vantagem o ganho de desempenho, contudo o software desenvolvido para estes processadores fica amarrado a suas funcionalidades. A conseqüência imediata é a perda de portabilidade.

Para um software apresentar ganhos reais de desempenho o desenvolvedor deve conhecer quais funcionalidades de hardware estão disponíveis e usá-las intensamente. Caso elas não sejam usadas, as tarefas terão que ser implementadas em software, apresentando menor desempenho, e parte do hardware estará sendo sub-utilizado.

Por exemplo, o processador Intel IXP2850 possui um co-processador para operações de *hashing*. Este co-processador será útil para busca em tabelas de *lookup*, caso o software empregue um algoritmo baseado em *hashing*. Portanto co-processadores que implementam algoritmos específicos oferecem melhor desempenho destas operações, mas limitam a liberdade de implementação do software.

5.5. Software para Processadores de Rede

Um processador de rede é um dispositivo programável por software. No entanto, para se obter o máximo do desempenho é necessário implementar o software adequadamente.

5.5.1. Arquitetura de software em camadas sobre NPs

O modelo de programação de softwares para equipamentos que utilizam NPs pode ser dividido em três camadas. A primeira integra o plano de encaminhamento de dados e é executada no NP, a segunda é uma interface de programação entre as camadas do plano de controle e do plano de dados e a terceira integra o plano de controle e é executada pelo processador principal do roteador. A figura 17 mostra este modelo.

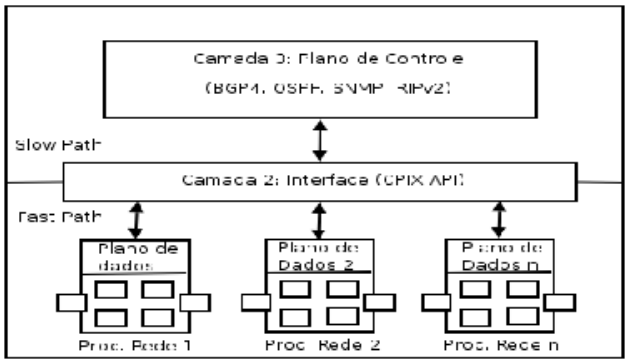


Figura 17. Arquitetura de software em um NP

A camada do **plano de encaminhamento de dados** é altamente dependente do hardware do NP. Esta parte do software apresenta baixa ou nenhuma portabilidade entre diferentes NPs. Além disso, seu código é pequeno e mais simples, se comparado ao software do plano de controle. Ele não executa, por exemplo, algoritmos de roteamento

ou políticas. Suas operações estão ligadas a busca em tabelas, reconhecimento de padrões e gerenciamento de filas[Herity 2005].

A segunda camada, **plano de interface**, fornece uma abstração do processador de rede. Isto permite que o software do plano de controle seja independente do processador de rede. Desta forma, sua portabilidade é maior que na primeira camada. Alterações e otimizações das funções na camada de encaminhamento não comprometem o plano de controle.

A camada do **plano de controle**, normalmente, constitui a maior e mais complexa parte do software. É esta parte de software que trata dados de pacotes de controle, atualiza tabelas de rotas, modifica políticas e etc. As características inerentes desta camada, bem como o fato dela ser programada para executar em um processador de uso geral tornam seu código bastante portável.

A portabilidade da camada do plano de controle é uma parte chave da construção de soluções para NP. Um aspecto crítico para vendedores de software de NPs é fornecer camadas de plano de controle facilmente portáveis e integráveis a qualquer implementação de camada de encaminhamento[Limited 2005]. Isto encurta o tempo de desenvolvimento da solução e aumenta a confiabilidade do produto.

O Network Processing Forum está desenvolvendo uma API, chamada CPIX API, como proposta de padrão para o plano de interface, entre o plano de controle e o plano de encaminhamento. O objetivo é modelar uma grande faixa de processadores de rede com uma API comum[Forum 2006].

5.5.2. Linguagens e ferramentas

Desenvolver software para NPs não é uma atividade trivial, pois para explorar as funcionalidades oferecidas pelo hardware é necessário conhecê-lo profundamente. Além disso, também é necessário conhecer as ferramentas e linguagens de desenvolvimento, oferecidas pelo fabricante, para o NP específico.

Os fabricantes de NPs oferecem várias facilidades para o desenvolvedor. Entre as mais comuns estão: Ambientes de Desenvolvimento Integrados (IDEs), compiladores C/C++ e debugadores. Outros fabricantes, ao invés de usarem linguagens imperativas (C/C++ e similares) para desenvolvimento, usam linguagens funcionais. Nestas linguagens o desenvolvedor especifica o que acontece quando condições preestabelecidas são verificadas. O fabricante Agere, por exemplo, disponibiliza uma linguagem para busca de padrões e classificação de alto nível[Systems 2003].

Apesar dos fabricantes disponibilizarem compiladores C/C++, linguagens funcionais e de alto nível, eles sugerem que programando em nível de micro-código (*Assembler*) poderão ser obtidos resultados melhores. A sintaxe e semântica de um micro-código são específicas de cada NP e existe pouquíssima reusabilidade entre códigos de NPs diferentes[Limited 2005].

Portanto o desenvolvimento de software otimizado exige estudos criteriosos sobre a arquitetura do processador de rede e sobre as ferramentas de programação oferecidas pelo fabricante do NP. Esses fatores vão influenciar o custo final da solução adotada.

5.6. A escolha de um Processador de Rede

A escolha do processador de rede, que será utilizado para desenvolver um equipamento, deve seguir critérios técnicos e critérios empresariais. É necessário verificar se o NP corresponde às necessidades do equipamento e nas características da equipe de desenvolvimento.

Que serviços e níveis de carga o processador deve suportar? Ou seja, é preciso identificar se o equipamento será de núcleo, borda ou de acesso. A correta classificação é fundamental, pois cada categoria de equipamentos tem suas próprias características funcionais e de desempenho. Existem linhas de NPs apropriados para cada categoria.

A arquitetura e funcionalidades do NP são apropriadas para a carga de trabalho? Deve-se, por exemplo, verificar: a existência de co-processadores que otimizem o processamento de pacotes, a arquitetura e a organização das unidades de processamento e a presença de interfaces para meio físico e para *switch fabrics*.

O quão fácil, rápido e passível de otimização é a programação e uso do NP? Deve-se verificar se o vendedor disponibiliza ferramentas eficientes de desenvolvimento de software para o NP. Isto facilita e agiliza o trabalho da equipe de programadores e reduz o tempo de desenvolvimento do produto.

Relação custo/benefício Cada linha de NP tem características especificas e o custo total de desenvolvimento e de produção devem ser avaliados.

Portanto, escolha do NP é difícil pois afeta diretamente todas as equipes que trabalham no desenvolvimento do produto: a equipe de hardware, a equipe de desenvolvimento e a equipe de finanças e marketing. Assim, a escolha correta do NP é fundamental no desenvolvimento de um produto comercialmente viável.

6. Field Programmable Gate Arrays (FPGAs)

FPGAs são circuitos digitais integrados, que possuem blocos lógicos programáveis interligados por blocos de conexão configuráveis. Esta arquitetura interna permite que um mesmo dispositivo seja configurado de diferentes formas podendo desempenhar diversas funções.

A expressão *Field Programmable*, Programável em Campo, do nome FPGA vem do fato da configuração interna dos blocos lógicos e de conexão, que compõem a arquitetura do dispositivo, serem programáveis “em campo”, isto é, a organização interna não é determinada durante a fabricação. Esta característica permite que um projetista configure um mesmo FPGA para executar diversas atividades diferentes. Conceitualmente os FPGAs situam-se entre os Dispositivos Lógico Programáveis (*Programmable Logical Device* - PLD) e Circuitos Integrados Específicos de Aplicação (*Application-Specific Integrated Circuit* - ASIC).

PLDs são dispositivos eletrônicos cuja a arquitetura é determinada pelo fabricante, contudo sua organização interna é possível de ser programada pelo projetista, em campo, para executar uma variedade de tarefas diferentes. Apesar de FPGAs e PLDs possuírem características semelhantes, os PLDs possuem um número muito menor de portas lógicas e reduzida possibilidade de configuração, limitando-os a implementações de problemas pequenos.

ASICs oferecem centenas de milhões de portas lógicas e possibilitam a implementação de funções grandes e complexas com excelente performance. Contudo, os projeto de construção destes dispositivos, em geral, são caros e demandam uma parcela considerável de tempo. Além disso, ASICs não são passíveis de programação como são os PLDs e FPGAs.

Os **FPGAs** conciliam característica destas duas tecnologias, pois possuem organização interna programável, como os PLDs, e são capazes de conter milhões de portas lógicas podendo implementar problemas complexos, como os ASICs. Isto faz dos FPGAs dispositivos versáteis e eficientes.

O custo do desenvolvimento de um projeto baseado em FPGAs é muito menor que o projeto de um ASIC, embora o custo de fabricação de ASICs em larga escala, em geral, seja menor. Além disso, implementar alterações de projeto é mais fácil e rápido usando FPGAs, pois são dispositivos reconfiguráveis e o tempo de desenvolvimento (*time to market*) é muito menor.

Atualmente existem várias áreas onde são implementadas soluções baseadas em FPGAs. Tradicionalmente o processamento digital de sinais tem sido feito por Processadores Digital de Sinais (DSPs), contudo devido ao aumento do poder de processamento dos FPGAs, recentemente estes têm sido bastante utilizados[Tessier and Burleson 2001]. Eles têm se tornado uma alternativa cada vez mais interessante no lugar de ASICs, além de serem uma boa opção para micro-controladores embarcados. FPGAs também são bastante utilizados na camada física de comunicação de dados e em Computação Reconfigurável.

6.1. Princípio de Funcionamento

Pode-se afirmar que um FPGA é composto basicamente pelos seguintes elementos: blocos lógicos, blocos de interconexão (blocos de conexão e de comutação), blocos de I/O e canais de roteamento. As formas de organizações e detalhes construtivos destes elementos variam bastante e são eles que determinam a arquitetura do FPGA.

Uma arquitetura típica de FPGAs pode ser visualizada na figura 18. O FPGA possui várias “ilhas” de blocos lógicos cercados por um “mar” de blocos de interconexão.

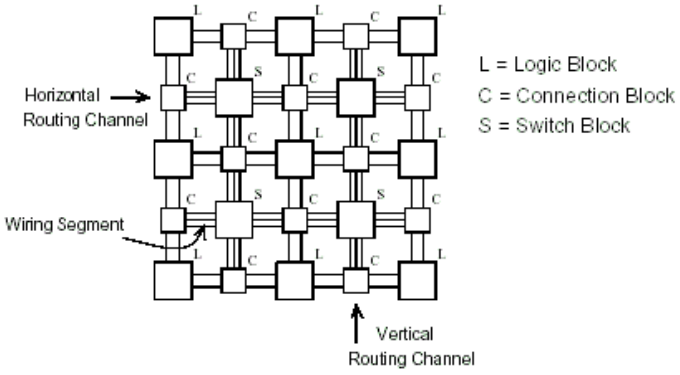


Figura 18. Arquitetura de um FPGA estilo ilha

As aplicações de circuitos projetados, que são carregadas no FPGA, são particionadas em funções menores, possíveis de serem mapeadas em blocos lógicos. As funções

lógicas completas são montadas no FPGA a partir da associação das funções lógicas, presentes nos blocos lógicos. Estas associações são feitas através de ligações configuráveis realizadas pelos blocos de interconexão.

Os blocos lógicos são usados para implementar funções lógicas combinacionais ou seqüenciais. Eles são constituídos de uma ou mais células lógicas. Quanto maior o número de células lógicas em um bloco, mais funções lógicas podem ser implementadas por bloco, e menos blocos serão necessários. Por outro lado, um bloco lógico maior exigirá mais espaço interno no FPGA. Portanto é necessário equilibrar o número de células contidas em um bloco e o seu tamanho final. Estudos indicam que os blocos lógicos devem ser constituídos de *clusters* de 8 células por razões de desempenho, redução de retardos e área utilizada[Marquardt et al. 2000]. Para interligar as células lógicas, no interior de cada bloco, são usadas matrizes de interconexão.

Existem várias abordagens de construção de células lógicas: pares de transistores, portas lógicas básicas NAND e XOR, tabelas de *lookup* e multiplexadores. A figura 19 mostra a representação de uma típica célula lógica baseada em uma tabela de *lookup* de quatro entradas. Uma tabela de *lookup* de tamanho N é, basicamente, uma memória que quando programada apropriadamente pode computar qualquer função de N entradas. O *flip-flop* tipo D pode ser usado como registrador, para *pipeline*, para guardar o estado de máquinas de transição ou em qualquer situação onde operações com o *clock* sejam necessárias. O elemento de lógica *carry* é um recurso especial utilizado para acelerar as operações baseadas em *carry*, por exemplo, a adição. Estudos realizados acerca do tamanho e complexidade das células lógicas apontam que o tamanho ótimo para tabelas de *lookup* varia entre 4 e 6 entradas[Ahmed and Rose 2000, Rose et al. 1989].

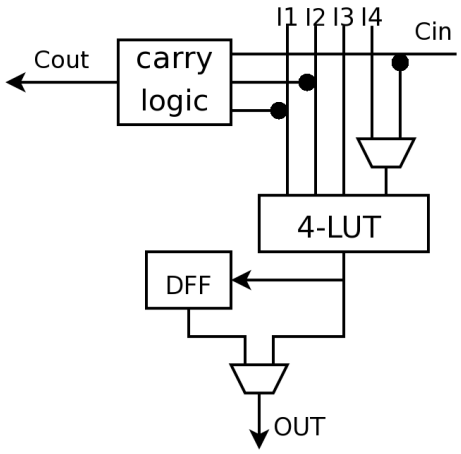


Figura 19. Célula Lógica padrão

A arquitetura de roteamento interliga blocos de I/O a blocos lógicos e blocos lógicos entre si. Ela é composta pelos blocos de interconexão (blocos de conexão e comutação) e por canais de roteamento. Os canais de roteamento são compostos por trilhas metálicas, que estão dispostas horizontalmente e verticalmente. Cada canal possui o mesmo número de trilhas.

Os blocos de conexão interligam as saídas e entradas de cada bloco lógico à malha de canais de roteamento. Eles são constituídos basicamente de transistores de passagem e de multiplexadores programáveis. Estes dispositivos funcionam como comutadores que

são configurados para habilitar a conexão entre os terminais dos blocos lógicos e as trilhas que constituem o canal.

Os blocos de comutação interligam as trilhas de um canal de roteamento vertical as trilhas de um canal horizontal e determinam todas as conexões possíveis. Eles são constituídos de comutadores programáveis que ativam as conexões desejadas. Existem várias topologias: bloco de comutação de disjunção, bloco de comutação universal, bloco de comutação de Wilton[Masud 1999]. A figura 20 apresenta detalhadamente todos os elementos descritos.

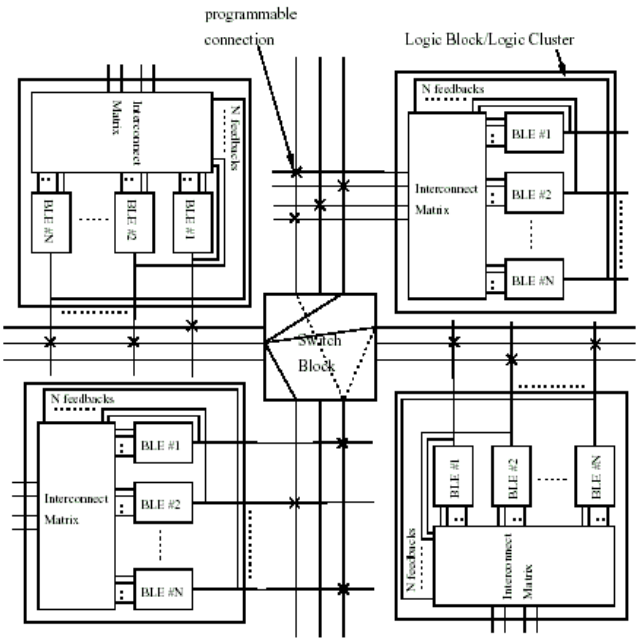


Figura 20. Arquitetura detalhada de FPGA estilo ilha.

Existem muitas variações desta arquitetura de FPGAs, cada uma delas objetivando melhorar diferentes aspectos do dispositivo. A arquitetura do Xilinx Virtex II Pro[Xilinx 2005] e a arquitetura Stratix-II[Lewis et al. 2005] apresentam alto desempenho e elevada densidade lógica, por outro lado, a arquitetura Cyclone[Leventis et al. 2003] apresenta baixo custo e desempenho razoável. Propostas de arquiteturas de baixo consumo energético são apresentados em [Gayasen et al. 2004, Mondal and Memik 2005].

Outra arquitetura interna que vem sendo estudada é a arquitetura de FPGA baseada em *pipeline*[Sharma et al. 2004]. O objetivo das pesquisas com esta arquitetura é melhorar o *clock* interno do dispositivo.

6.2. Tecnologias de Blocos e Matrizes de Interconexão

Os blocos e matrizes de interconexão utilizam três tipos de tecnologias para guardar as configurações das células de comutação: SRAMs, Antifusos ou E2PROM/Flash.

A principal vantagem dos blocos e matrizes de interconexão baseados em **SRAMs** é que eles podem ser reconfigurados várias vezes e o tempo de programação e a leitura é bastante rápida. Assim, melhorias, correções e mudanças no projeto são facilmente implementadas no FPGA. Além disso, as SRAMs apresentam-se na vanguarda da tecnologia

de densidade de fabricação, atualmente em 90nm[Telikepalli 2005].

A desvantagem dos dispositivos baseados em SRAMs é que eles tem que ser reconfigurados toda vez que são inicializados. Isto implica que deve existir uma memória externa para guardar a configuração e um circuito extra para carregá-la no FPGA. Uma conseqüência desta abordagem está relacionada a segurança dos dados na memória externa. Pessoas interessadas na codificação do dispositivo podem aplicar engenharia reversa sobre os dados, assim, obter o código fonte de determinada função. Uma solução para esse problema é o uso de técnicas de encriptação.

O **Antifuso** é um dispositivo elétrico que executa uma função oposta a do fusível. Inicialmente ele apresenta uma alta impedância (isolador), quando submetido a uma voltagem elevada passa a se comportar definitivamente como um condutor (baixa impedância). Dispositivos baseados em antifusos são programados apenas uma vez, e esta é sua principal desvantagem. Contudo, suas configurações não são voláteis, ou seja, não é necessário carregar configurações. Estes dispositivos, no entanto, não apresentam problemas relativos a segurança dos dados da configuração e não são susceptíveis a radiação.

Dispositivos baseados em **EEPROM** e **Flash** também são reprogramáveis. Porém, além da programação ser feita através de um dispositivo de programação externo, o tempo de gravação e leitura são bem maiores do que o tempo gasto em dispositivos baseados em SRAM. As configurações e dados carregados neste dispositivo não são voláteis. A tabela 8 apresenta um quadro comparativo resumido das tecnologias.

Tabela 8. Tecnologias de Blocos e Matrizes de Interconexão [Maxfield 2004]

Característica	SRAM	Antifuso	EEPROM/FLASH
Densidade Tecnológica	Estado da Arte	Uma ou mais gerações atrás	Uma ou mais gerações atrás
Reprogramável	Sim (no sistema)	Não	Sim (fora ou no sistema)
Velocidade de gravação	Rápida		3x menor que SRAMs.
Volatilidade	Sim	Não	Não
Requer arquivo de configuração externo	Sim	Não	Não
Bom para Prototipação	Sim (Muito Bom)	Não	Sim (Razoável)
Segurança de IP Core	Aceitável	Muito Bom	Muito Bom
Consumo de energia	Médio	Baixo	Médio
Resistente a radiação	Não	Sim	Não totalmente

6.3. Processadores Embutidos

FPGAs de alto desempenho oferecem a possibilidade de conter processadores embutidos, eles são chamados de "*microprocessor cores*". Esta característica traz algumas vantagens. Com um microprocessador presente no próprio FPGA o microprocessador externo é dispensável, reduzindo-se no número de dispositivos e o tamanho, custo e complexidade da placa de circuito impresso. Processadores embutidos dividem-se em: *Hard Microprocessador Cores* e *Soft Microprocessor Cores*.

6.3.1. Hard Microprocessador Core

O *Hard Microprocessador Core* é um bloco de silício, definido em tempo de fabricação no interior do FPGA, que implementa um processador. Cada um dos principais fabricantes de FPGAs adota um processador diferente: Altera embute processadores ARMs nos

seus FPGAs, a QuickLogic optou por soluções baseadas da MIPS e a Xilinx adotou o PowerPC.

Existem duas abordagens para a disposição interna dos processadores no FPGA[Maxfield 2004]. A primeira é colocar o processador e recursos necessários em uma faixa na lateral da malha principal do FPGA. A figura 21 mostra em detalhes.

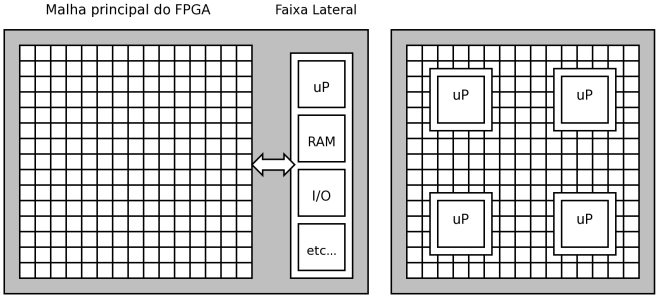


Figura 21. À esquerda o processador e demais dispositivos estão concentrados na linha lateral. À direita o processador e dispositivos estão espalhados pela malha principal.

Uma das vantagens é a fácil adição de dispositivos que complementam o processador na linha lateral. Além disso, a malha principal de FPGAs com e sem processadores embutidos é a mesma, simplificando as ferramentas de desenvolvimento. Os fabricantes Altera e QuickLogic utilizam esta técnica.

Outra abordagem é embutir processadores diretamente na malha principal do FPGA. A figura 21 mostra em detalhes. Devido ao fato dos processadores estarem muito próximos das funções implementadas na malha principal do FPGA, existe um ganho de desempenho. Porém, as ferramentas de desenvolvimento devem estar preparadas para reconhecer as áreas alocadas aos dispositivos embutidos. O fabricante Xilinx utiliza esta técnica.

6.3.2. Soft Microprocessor Core

O *Soft Microprocessor Core* é constituído de um grupo blocos lógicos configurados para agirem como um processador. Novamente, cada fabricante possui uma implementação diferente. A Altera disponibiliza o NIOS II, ele é um processador configurável de 32 bits e suporta *clocks* de até 135MHz. A Xilinx utiliza o MicroBlaze, também é um processador configurável de 32 bits e suporta clocks de até 150MHz.

Esta solução é muito flexível e apresenta um custo bem menor, pois o projetista pode carregar o processador apenas se necessário, além de poder fazer maiores customizações. Contudo, *Soft Microprocessor Cores* são mais lentos e gastam mais energia que os *Hard Microprocessor Cores*. Técnicas de particionamento de harware/software dinâmica têm sido utilizadas para diminuir estas diferenças de desempenho[Lysecky and Vahid 2005].

6.4. Interfaces de I/O de propósito geral e Gigabit Transceivers

FPGA's contém interfaces de I/O de propósito geral, estas interfaces podem ser configuradas para operarem conforme vários padrões de I/O. Caso estas interfaces não tivessem

esta característica, vários blocos de I/O, um para cada padrão diferente, deveriam ser colocados no FPGA. A figura 22 mostra a distribuição destes blocos em um FPGA.

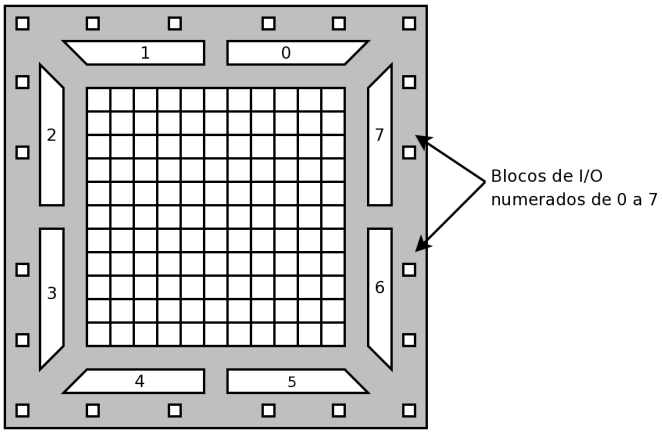


Figura 22. Disposição interna dos blocos de I/O

Para viabilizar a configuração dos diversos padrões de I/O diferentes, os blocos se utilizam de impedâncias internas configuráveis além de pinos próprios de alimentação. Neste último caso, os pinos são utilizados quando a alimentação do padrão de I/O é diferente da tensão interna do núcleo do FPGA.

Gigabit transceivers, configurados em tempo de fabricação, também são disponibilizados em FPGA's. Estas interfaces trabalham em velocidades bastante elevadas, possibilitando a troca de grandes quantidades de dados. O RocketIO X Transceiver [Xilinx 2004] é utilizados em alguns modelos de FPGA's da Xilinx, a tabela 9 mostra os modos de operação e as respectivas taxas de transmissão atingida pelo dispositivo.

Tabela 9. Modos de operação do RocketIO X Transceiver.

Modos de Operação	Taxa de Transmissão Atingida (Gb/s)
SONET OC-48	2,488
PCI Express	2,5
Infiniband	2,5
XAUI (10 Gigabit Ethernet)	3,125
XAUI (10 Gigabit Fibre Channel)	3,1875
SONET OC-192	9,95328
Aurora (Protocolo Xilinx)	2,488 - 10,3135
Modo Padrão	2,488 - 10,3135

6.5. Núcleos de propriedade intelectual (IP Core)

Núcleo de Propriedade Intelectual (*IP Core*) é um bloco funcional pré-projetado para desempenhar uma função específica. São exemplos de IP Cores: núcleos microprocessados, interfaces gigabit, interfaces de barramentos, algoritmos para DSP etc. *IP Cores* dividem-se em: *Hard IP*, *Soft IP* e *Firm IP*.

Hard IPs são blocos funcionais implementados no FPGA em tempo de fabricação. Eles são projetados para serem eficientes e otimizarem o espaço utilizado o tamanho e consumo de energia. Apresentam baixa flexibilidade por serem implementados diretamente na malha do FPGA.

Soft IPs são blocos funcionais projetados para serem carregados no FPGA. Eles são programados utilizando-se uma linguagem de descrição de hardware (HDL) como Verilog ou VHDL. São disponibilizados na forma de bibliotecas de blocos funcionais e funções que podem ser incluídas em um projeto. *Soft IPs* apresentam grande flexibilidade, porém desempenho menor.

Firm IPs estão entre *Hard IPs* e *Soft IPs*. Apesar de serem disponibilizados na forma de bibliotecas de blocos funcionais, eles já vêm mapeados e roteados em blocos lógicos. *Firm IPs* buscam oferecer melhorias de desempenho através de mapeamento e roteamento otimizados.

IP Cores podem diminuir bastante o tempo de desenvolvimento de um projeto, pois viabilizam o reuso de blocos funcionais. Pode-se afirmar que *IP Cores* estão para projetos de hardware, assim como DLLs estão para projetos de software. Desta forma o projetista pode utilizar *IP Cores* para implementar funcionalidades básicas e gastar mais tempo com a implementação das funcionalidades específicas do projeto.

Bibliotecas de *IP Cores* são disponibilizadas pelos próprios fabricantes de FPGAs ou por empresas especializadas neste tipo de desenvolvimento. Existe uma comunidade projetando e publicando *IP Cores* a licença LGPL, mais detalhes do projeto e dos IP Cores estão disponíveis em www.opencores.org.

6.6. Programação e configuração de FPGAs

A implementação de um projeto em um FPGA consiste de alguns passos. Primeiramente é necessário entrar com a descrição do projeto do circuito lógico. Normalmente são utilizadas linguagens de descrição de hardware (*Hardware Description Language - HDL*), também podem ser usados editores esquemáticos.

A síntese é a atividade de converter o código HDL em uma *netlist*. Uma *netlist* é a descrição de um projeto eletrônico em termos de portas lógicas e suas interconexões. A tarefa de realizar esta conversão é do sintetizador lógico, pode-se afirmar que ele está para as HDLs, assim como o compilador está para as linguagens de programação.

Após a síntese vem as atividade de *mapping, packing* e *place-and-routing*. A atividade de *mapping* consiste em associar as funções, expressas por portas lógicas na *netlist*, às tabelas de *lookup* presentes nas células lógicas. Em seguida vem a atividade de *packing* que agrupa as funções mapeadas em células lógicas em blocos lógicos. Por fim, a atividade de *place-and-routing* determina a configuração das conexões entre os blocos lógicos. A partir destas configurações, a ferramenta de implementação gera um *bitstream* contendo a configuração das matrizes e blocos de interconexão. Por fim, o *bitstream* é carregado no FPGA. Os comutadores eletrônicos responderão às configurações do *bitstream*. Ao final do *download* para o FPGA, este passa a funcionar conforme o especificado pela HDL.

As funcionalidades de um projeto de circuito eletrônico podem ser representadas em diferentes níveis de abstração: estrutural, funcional, comportamental e sistêmico. As HDLs buscam suportar tais abstrações[Maxfield 2004].

O **nível estrutural** é composto de dois sub-níveis: comutadores e de portas. No sub-nível de comutadores os circuitos são descritos como uma *netlist* de transistores. No sub-nível de portas o sistema é descrito como uma *netlist* de portas lógicas.

O **nível funcional** permite que uma função seja descrita utilizando equações booleanas. Ele também compreende o nível transferência de registradores (*Register Transfer Level - RTL*). O RTL permite que sinais e registradores sejam declarados e combinados utilizando operadores da lógica booleana.

O **nível comportamental** permite descrever o comportamento de um circuito usando construções abstratas como *loops* e processos. Permite também que operações como soma e multiplicação sejam usadas em equações. São exemplos de HDLs deste nível: Verilog e VHDL [Smith 1996].

O **nível de sistema** surgiu da necessidade de lidar com projetos de grandes portes e complexidades. As ferramentas deste nível tentam conciliar, usando uma abordagem *top-down*, os estágios de projeto funcional e arquitetural. O estágio funcional se preocupa com o desenvolvimento da aplicação do sistema, em geral sem se preocupar com o hardware. O estágio arquitetural busca descrever o hardware do sistema. Uma questão crítica para as ferramentas deste nível é o particionamento do sistema em hardware e software. São exemplos de HDLs deste nível: SystemVerilog[Accellera 2004] e SystemC.

A figura 23 apresenta um gráfico comparativo entre algumas das principais HDLs e os níveis de abstração suportados.

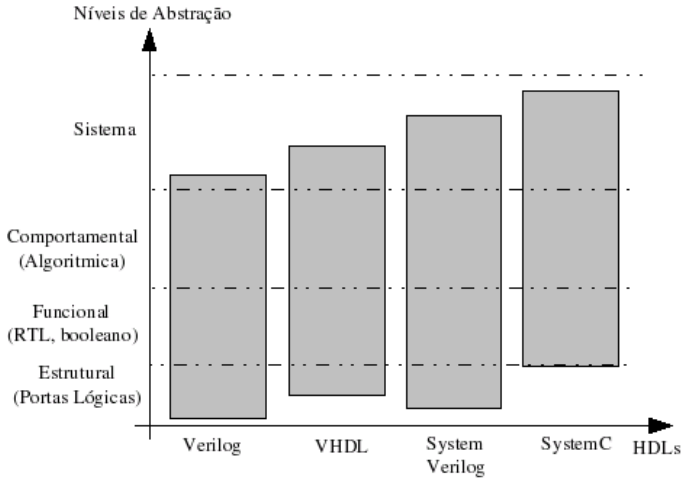


Figura 23. Níveis de abstração de HDLs

6.7. Ferramentas Open Source de desenvolvimento

Existem várias ferramentas open-source desenvolvimento para FPGA. **Icarus Verilog** é uma ferramenta de síntese e simulação de projetos. O compilador é projetado para trabalhar com a especificação da linguagem Verilog IEEE 1364-2001, o projeto pode ser encontrado em www.icarus.com/eda/verilog.

SystemC pode ser considerado um framework de desenvolvimento, oferece ferramentas de síntese e simulação. A linguagem de programação é considerada uma HDL, mas possui um grande poder de abstração nos níveis comportamentais e de sistema. Baseada em C/C++, o padrão SystemC IEEE 1666, foi aprovado em dezembro de 2005. O projeto é mantido por um grupo, composto de empresas e universidades, sem fins lucrativos. Detalhes do projeto estão disponíveis em www.systemc.org.

Verilator é uma simulador que compila código sintetizado Verilog em código C++ ou SystemC. Ele é projetado para grandes projetos onde o tempo de simulação é restritivo. Detalhes do projeto estão disponíveis em www.veripool.com/verilator.html.

DinoTrace e **GTKWave** são ferramentas de visualização de arquivos produzidos por ferramentas de simulação, normalmente em formato VCD. Estas ferramentas podem ser usadas para visualizar os arquivos gerados pelo Icarus Verilog, por exemplo. DinoTrace pode ser encontrado em www.veripool.com/dinotrace e GTLWave pode ser encontrado em www.geda.seul.org/tools/gtkwave.

Ferramentas grátis de desenvolvimento podem ser encontradas nas páginas dos fabricantes de FPGAs. É o caso das ferramentas ISE da Xilinx e Quartus da Altera.

6.8. Computação Reconfigurável (RC)

Computação reconfigurável é a tecnologia na qual circuitos integrados têm suas malhas de interconexão interna reprogramadas em tempo de execução. Esta tecnologia possibilita que várias funcionalidades diferentes sejam carregadas em um mesmo CI de capacidade limitada.

O conceito de computação reconfigurável é baseado no conceito de hardware virtual, que é semelhante ao conceito de memória virtual. O hardware físico é menor do que o necessário para implementar todas as funções lógicas da aplicação. Contudo, ao invés de reduzir o número de funções a serem mapeadas, elas são carregadas e descarregadas do dispositivo à medida que são necessárias[Compton and Hauck 2002].

Esta técnica proporciona um ganho de desempenho da aplicação, pois mais partes de software pode ser mapeada em hardware. À medida que parte da aplicação esta sendo intensamente executada ela é carregada para o hardware. Desta forma o tempo gasto na execução é menor.

O hardware e o software empregados em RC apresentam características particulares. O hardware normalmente baseia em CIs de FPGAs, associados com diferentes componentes externos, como processadores. Os FPGAs podem ser de contexto simples ou multi-contexto. FPGAs de contexto simples têm sua malha de interconexão totalmente reconfiguradas para qualquer alteração a ser realizada. Já os FPGAs multi-contexto permitem que apenas uma região específica da matriz de interconexão seja alterada para realizar mudanças de configurações. FPGAs multi-contexto são mais indicados para situações onde reconfigurações de malhas de interconexão em tempo de execução são necessárias, pois não é preciso reconfigurar a malha inteira.

O correto particionamento da aplicação entre execução em hardware e software é um fator determinante para o desempenho do sistema projetado. Em geral, seqüências de controle complexas e com muitos desvios são melhor implementadas em software; operações de seqüência de comandos bem definidas são melhor implementadas em hardware, por exemplo, algoritmos de criptografia. Técnicas de particionamento dinâmico de hardware/software, também conhecidas por *Co-design* são utilizadas para realizar a carga de descarga de partes da aplicação que precisam ser executadas[Lysecky and Vahid 2004]. Esta técnica tem apresentado bons ganhos de desempenho.

A rápida reconfiguração e carga de uma funcionalidade em um FPGA é fundamental para a aplicação apresentar um bom desempenho. Várias técnicas podem ser

empregadas para acelerar esta tarefa [Compton and Hauck 2002]: busca antecipada de configuração, compressão de configurações, relocação/desfragmentação de sistemas reconfiguráveis e *cache* de configurações.

7. Estudo de Casos

Apresentamos nessa seção três estudos de caso de implementação de roteadores utilizando Processadores de Rede e FPGA. O primeiro exemplo, mostrado na seção 7.1., mostra o projeto FPX, um roteador baseado em FPGA e RC. O segundo exemplo, mostrado na seção 7.2., mostra a implementação de algoritmos de *lookup* em FPGAs. E, finalmente, a seção 7.3., mostra uma estudo comparativo de implementação de roteador baseado em FPGA e NP.

7.1. Projeto de roteador baseado em FPGA e RC

O projeto *FPX Plataform* tem como objetivo a construção de roteadores de alto desempenho, capaz de acompanhar o crescimento das taxas de transmissão. O projeto é baseado na utilização de hardware reconfigurável. Atualmente ele está sendo desenvolvido na Universidade de Washington, em St. Louis, com o apoio de pesquisadores de outras universidades e empresas parceiras. Detalhes do projeto podem ser encontrados www.arl.wustl.edu/projects/fpx.

A estrutura geral do roteador pode ser vista na figura 24 [Lockwood et al. 2000]. O roteador é constituído de um switch central (*Washington University Gigabit Switch - WUGS*), vários processadores de porta (*Field-Programmable Port Extender - FPX*) e associado a cada processador de porta está uma interface de rede ótica.

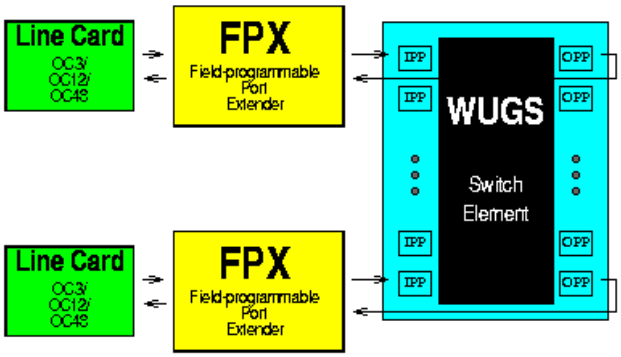


Figura 24. Estrutura geral roteador FPX

O *switch* interliga os processadores de porta. O WUGS pode ser configurado para operar com um número de portas variando de 8 a 4096, e permite que cada interface trabalhe a 2.4 Gb/s.

O processador central[Taylor et al. 2001], não mostrado na figura, é o responsável por executar funções de gerenciamento de interfaces e processadores de portas, manutenção de estruturas de dados de classificação de pacotes e busca de rotas. Funções do *Slow Path* são executadas neste processador. Ele é conectado ao *switch* central.

O Processador de Porta, FPX, fica entre a interface de rede ótica e o *switch*. Ele é quem realiza o encaminhamento, enfileiramento, escalonamento e classificação de pa-

cotes. Também são executadas funções de processamento sobre pacotes, por exemplo, funções criptográficas.

O FPX é constituído basicamente de 6 bancos de memória e 2 FPGAs Xilinx Virtex. Os bancos de memória se dividem em: 3 de SRAMs, 2 de SDRAMs e 1 de memória EPROM. Os FPGAs se dividem em: *Network Interface Device(NID)* implementado em um Xilinx XCV600E e *Reconfigurable Aplication Device(RAD)* implementado em um Xilinx XCV-2000E. A figura 25 mostra o diagrama lógico de um FPX. A placa de circuito impresso mede 20cm x 10.5cm. O esquema físico pode ser encontrado na página oficial do projeto.

O NID controla a comunicação e o fluxo de pacotes entre a interface de rede, o *switch* central e os módulos do RAD. Ele também controla o mecanismo de carga de configurações do RAD. Sua configuração é guardada em uma memória EPROM e é carregada no início de sua operação [Lockwood et al. 2001].

A arquitetura interna do FPGA NID é composta de um *switch* de quatro portas que transfere os dados entre as interfaces externas e o RAD. Também faz parte do NID um processador de células de controle que é o responsável por realizar a programação do RAD. Circuitos virtuais de tabelas de lookup, não apresentados na figura 25, são utilizados para controlar o fluxo de pacotes no *switch* interno do NID.

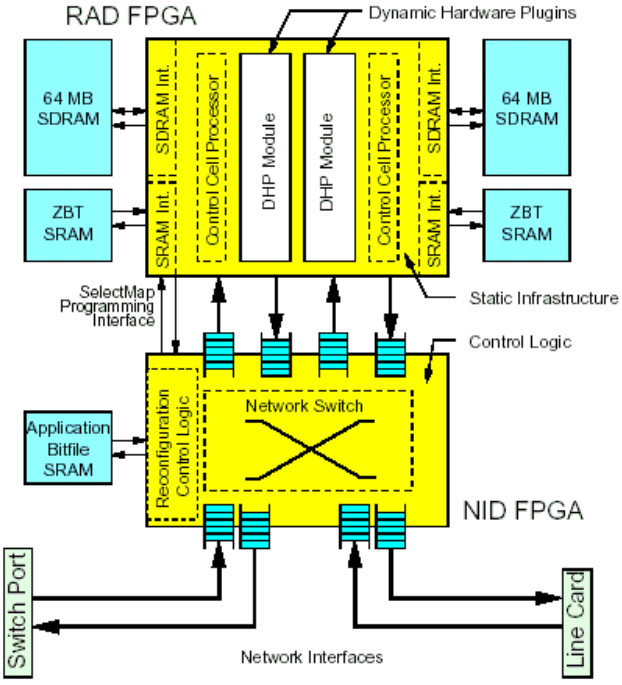


Figura 25. Organização lógica do FPX

O FPGA do RAD é o responsável por executar as funcionalidades específicas da aplicação. Elas são implementadas na forma de *Dynamic Hardware Plugins - DHPs*. DHPs são módulos que podem ser carregados ou removido de um FPGA ativo sem gerar qualquer distúrbio sobre os demais circuitos em operação [Horta et al. 2002]. Múltiplos DHPs podem podem estar carregados e em execução no FPGA do RAD.

A arquitetura do DHP se divide em duas partes principais: *plugins* de hardware

e infra-estrutura[Taylor et al. 2001]. Os *plugins* contêm as implementações das funcionalidades programadas pelo usuário. A infra-estrutura fornece um conjunto de serviços comuns a todos os *plugins* de hardware.

A infra-estrutura comporta-se como uma API, fornecendo uma interface padrão para os desenvolvedores, o que facilita o projeto modular de DHPs. Suas principais funções são: rotear pacotes e dados entre plugins e portas de I/O, reconfigurar dinamicamente *plugins* de hardware e fornecer interface para memórias e dispositivos externos.

Na figura 25 algumas funcionalidades fornecidas pela infra-estrutura podem ser ressaltadas. Cada módulo possui uma interface de comunicação com o NID e duas interfaces para os barramentos das memórias externa, uma para SRAM e outra para a SDRAM. Além de processadores de célula de controle responsáveis pela configuração de *plugins*.

Devido à velocidade de acesso o módulo deve utilizar a SRAM para armazenar estruturas de dados de controle e ponteiros para dados. Normalmente o conteúdo da tabela de *lookup* é armazenado nesta memória. A SDRAM, devido ao fato de possuir tempo de acesso e capacidade de armazenamento maiores, é utilizada para guardar dados e pacotes a serem processados.

O *plugin* de hardware é constituído, basicamente, de interfaces para infra-estrutura e da funcionalidade a ser executada. Para que uma funcionalidade seja eficientemente implementada em um DHP ela deve possuir algumas características: deve ser computacionalmente intensiva e necessitar operar sobre *streams* de dados com baixos tempos de resposta, deve possuir instruções que sejam passíveis de serem executadas em paralelo ou *pipelined*. Aplicações puramente seqüenciais não tiram proveito do paralelismo inerente à implementações em hardware.

A configuração e reprogramação dos FPGAs do FPX são realizadas da seguinte forma. Quando o sistema é inicializado pela primeira vez, as configurações do NID são carregadas da EPROM. A partir daí, o NID assume a responsabilidade de reconfigurar o RAD. Ele recebe os dados de configuração, na forma de células/pacotes de controle, através da interface de rede. Estes pacotes são organizados seqüencialmente e armazenados na memória *Aplication Bitfile SRAM*, figura 25. Quando o último pacote de configuração é recebido, o NID dá início a reprogramação do RAD.

O RAD permanece ativo durante o ciclo de reprogramação, ou seja, caso um módulo esteja sendo reconfigurado, os demais permanecem em execução, processando normalmente os pacotes. Isto é possível devido ao fato do NID controlar a porta de reprogramação (*JTAG port*) do RAD e enviar um *stream* de reconfiguração (*bitfile*) que contém apenas os comandos necessários para reprogramar partes específicas do dispositivo.

A ferramenta PARBIT gera arquivos *bitfile* a partir do *original bitfile*, do *target bitfile* e parâmetros fornecidos pelo usuário[Horta et al. 2002]. Os arquivos gerados pela ferramenta são também chamados *partial bitfiles(parbitfiles)*, porque configuram regiões determinadas do FPGA. Além desta facilidade, *parbitfiles* são menores que *bitfiles* completos, diminuindo o tempo gasto para transferir as configurações. A ferramenta facilita o processo de configuração e carga de DHPs.

A *FPX Plataform* apresenta-se como uma solução escalável, versátil e robusta

para construção de roteadores multi-portas. A utilização de computação reconfigurável no projeto permitiu alcançar uma flexibilidade só disponível em implementações baseada em software e uma performance só atingida em projetos baseados em ASICs.

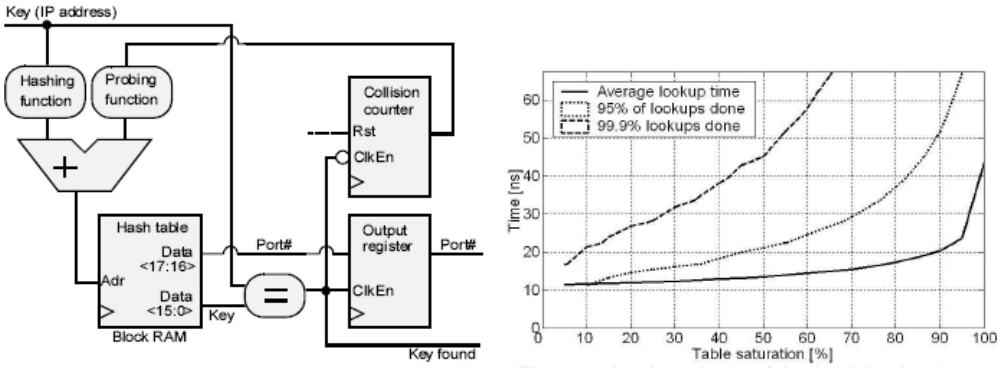
7.2. Comparação de Implementações de algoritmos *IPLookup* em FPGAs

Este estudo compara duas soluções de *IP-Lookup* em FPGA. São apresentadas idéias de implementação para *hashing* e árvores. Percebe-se a existência de vários prós e contra, por exemplo: hashing apresenta melhor desempenho para a busca, porém, gasta bem mais memória do que um algoritmo de árvore. A partir destas comparações são tiradas conclusões sobre custo-benefício em várias situações e são indicadas algumas soluções que melhoram o desempenho em cada uma delas[Taylor et al. 2002, Bemmann 2005a].

7.2.1. Implementação de algoritmos de *Hash* em FPGA

Para alcançar resultados satisfatórios com tabelas *hash*, a escolha da função *hash* adequada é crucial. O algoritmo aqui demonstrado usa portas XOR, função *hash* que dá excelentes resultados e é facilmente computável.

A implementação é feita associando-se cada chave a um número da porta de destino do roteador, mostrada na figura 26(a). A RAM é configurada para 18 bits (1024 entradas), assim uma chave de 16 bits e a porta de 2 bits podem armazenar juntas sob o mesmo endereço. A figura 26(a) não mostra a lógica adicional para detectar uma chave nova na entrada e o circuito para controlar colisões. Enquanto o hardware executar o *lookup*, um circuito externo tem a função de manter a tabela *hash*. Isto é viável pois as tabelas são atualizadas raramente, tipicamente uma atualização para alguns milhões de *lookups*[Bemmann 2005b].



(a) Hardware simplificado de *lookup* baseado em *hash*

(b) Tempos de lookup em um hardware *hash*

Figura 26. Implementação de *hash* em FPGA

Enquanto os tempos médios do *lookup* forem baixos com *hashing*, os *lookups* individuais podem consumir significativamente mais tempo. As experiências mostram que o tempo do *lookup* é independente do tamanho da tabela se a relação entre o tamanho da tabela e o número das entradas (saturação da tabela) permanecer constante. Assim, pode-se determinar com base na figura 26(b) quantos blocos da RAM são necessários para

que um percentual razoavelmente elevado dos *lookups* encontrar uma resposta dentro de tempo desejado em uma determinada aplicação.

7.2.2. Implementação de algoritmos de árvore (*tries*) em FPGA

Os algoritmos baseados em árvore têm um desempenho médio muito mais baixo do que tabelas *hash*, porém podem garantir um desempenho satisfatório no pior caso e, por isso, são mais previsíveis. Os algoritmos de árvore que comparam uma parcela diferente da chave em cada nível de profundidade, não podem ser balanceados desde que a profundidade dependa das chaves pais. Uma *trie*, por exemplo, é uma árvore binária com chave de comparação "bitwise". Sua profundidade máxima é a largura da chave. O algoritmo de *PATRICIA* é uma variação do *trie* que elimina filhos em um sentido único. É muito usado para *lookup*.

Para implementação foi analisado um esquema baseado em uma árvore binária onde os filhos da esquerda fossem menores e os filhos da direita fossem maiores que seu pai. A motivação é que, neste caso, a profundidade da árvore depende somente do número das entradas e não do comprimento da chave. Isto reduz o tempo de execução do pior caso de 16 para 9 iterações, porém requer o balanceamento da árvore. Cada nó tem um endereço de memória fixo. Se a chave não for encontrada, o algoritmo se ramifica à esquerda ou à direita, dependendo do resultado da comparação. O algoritmo permite uma execução consideravelmente eficiente em termos de área e garante uma árvore perfeitamente equilibrada, porém a tabela de roteamento inteira tem que ser reconstruída na inserção de uma nova entrada. Este esforço extra cresce com a freqüência de atualização e com o tamanho da tabela de roteamento. O hardware é semelhante à figura 26(a) com registros para o endereço do nó e distância, retirando-se o *hash*, as funções de varredura e o contador de colisões.

Ao contrário da aproximação em *hashing*, o algoritmo de árvore otimiza o uso do recurso. Os tempos do *lookup* dependem do número das entradas na tabela de roteamento. Seu tamanho real não tem nenhum efeito no desempenho, isto é a memória não pode ser negociada para uma latência mais baixa. O desempenho para quantidades diferentes de entradas aleatórias foi determinado nas simulações, figura 27. O tempo médio do *lookup* segue uma curva logarítmica. Na curva máxima do tempo do *lookup*, as etapas desobstruídas marcam onde a quantidade alcança um poder de dois e a profundidade da árvore aumenta por um nível.

7.2.3. Comparação

Enquanto FPGAs evolui para implementar sistemas mais complexos, a relação custo-benefício do projeto tornam-se cada vez mais importante. A área, a memória ou a latência são geralmente contrárias, porém esta análise indica uma melhor solução global. Os gráficos da figura 28 são baseadas em um cenário não trivial onde 700 entradas aleatórias sejam armazenadas e a latência da inclusão não é considerada. A aproximação baseada em árvore (E e F) é a mais eficiente em termos de memória, mas os tempos de *lookup* são altos. O hashing (A, B, C e D) oferece tempo significativamente menores, mas requer mais memória, de 50% a 500% maior conforme a aplicação[Bemmann 2005b].

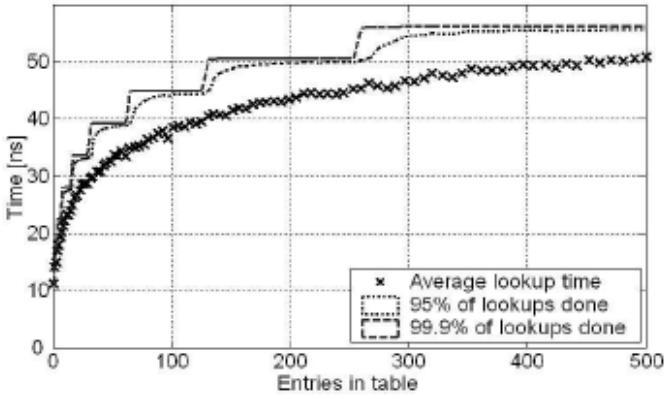
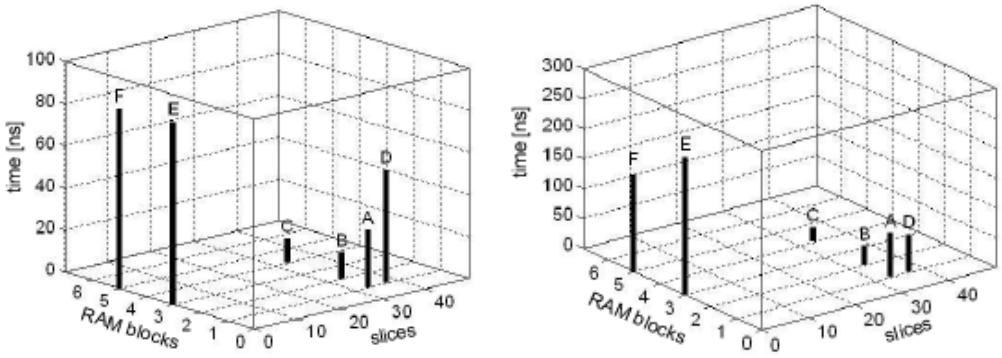


Figura 27. Tempos de *lookup* em uma implementação de árvores



(a) Média de tempos de *lookup*

(b) Tempo para completar 99% dos *lookups*

Figura 28. Comparações de implementações de *lookup* em FPGAs

Também as aplicações que requerem "don't care" favorecerão as árvores, tendo em vista que as tabelas de *hash* não permitem combinar o prefixo no teste padrão. Onde uma garantia no pior caso são necessárias, *hashing* deve ser evitado completamente, ou um cuidado especial deve ser tomado nas entradas para evitar colisões. A principal vantagem do *hashing* é que, em média, seus tempos de *lookup* são baixos, geralmente menores que os das árvores. Alocando mais memória, a porcentagem dos casos com tempos longos do *lookup* podem ser reduzida. As aplicações que visam o rendimento elevado favorecerão assim algoritmos baseados em *hashing*.

7.3. Estudo de caso comparativo: *Network Processors (NP)* x FPGAs

O objetivo deste tópico é apresentar a metodologia e os resultados de um estudo comparativo[Ravindran et al. 2005] entre uma solução de roteamento baseada em FPGA e uma outra baseada em NP (Intel IXP-2800)[Meng et al. 2003]. A intenção deste estudo é confrontar a eficiência do desenvolvimento do projeto de um ASIC e de um projeto baseado em FPGAs.

Em um primeiro momento são apresentados detalhes da construção do roteador baseado em FPGAs, depois uma breve descrição da metodologia dos testes e, por fim, são apresentados dos resultados obtidos.

O projeto do roteador baseado em FPGAs consiste, basicamente, na implementação de um sistema multiprocessado, baseado em *Soft Microprocessor Cores*, para encaminhamento de pacotes IPv4. O FPGA utilizado para a construção do roteador foi o Xilinx Virtex-II Pro 2VP50.

O sistema multiprocessado é composto pelos seguintes blocos construtivos: 14 Xilinx MicroBlaze *soft microprocessor IP Cores*, 2 IBM PowerPC 405 *hard microprocessor cores* e blocos de memória RAM (BRAM) embutidos no dispositivo. O diagrama esquemático da arquitetura interna pode ser visto na figura 29.

A interconexão entre os blocos é suportada por barramentos de comunicação. Barramentos *On-Chip Peripheral Bus (OPB)* permitem que MicroBlazes acessem memórias compartilhadas e outros periféricos. O barramento CoreConnect *Processor Local Bus (PLB)* serve aos núcleos PowerPCs. Os barramentos ponto-a-ponto *Fast Simplex Links (FSL)* funcionam como FIFOS e interconectam as MicroBlazes. Duas interfaces Gigabit Ethernet (GEMAC) recebem e enviam pacotes.

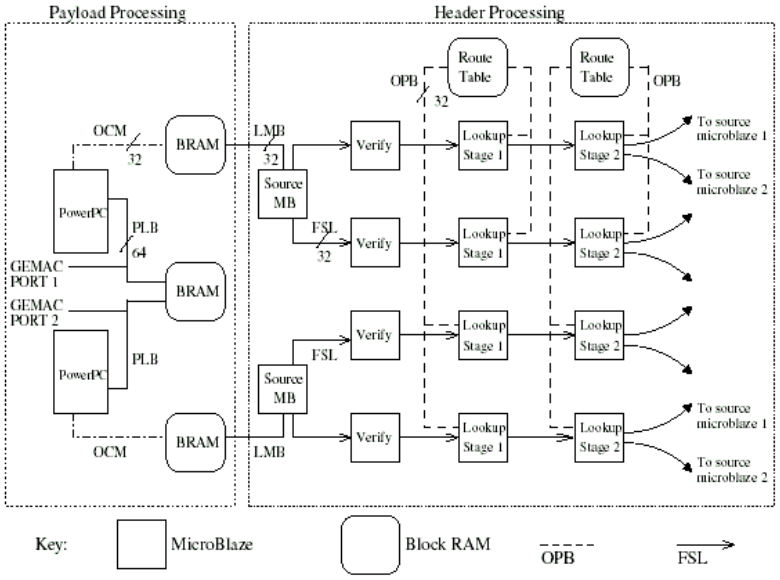


Figura 29. Arquitetura geral do roteador baseado em FPGA

A estratégia para implementação da tabela de *lookup* foi baseada em árvores *multi-bit tries* com *stride* fixo. A ordem dos *strides* é a seguinte: 12, 4, 4, 4, 4 e 4. Ou seja, o *stride* do nível 1 inspeciona os primeiros 12 bits do endereço, e os *strides* subseqüentes inspecionam 4 bits por vez. Assim, são necessários, no máximo, seis acessos à memória para realizar a busca por um endereço, mais um acesso extra para determinar a porta de saída correspondente, totalizando sete acessos à memória. O objetivo da implementação é conseguir equilibrar o número de acessos à memória e espaço utilizado.

A tabela de *lookup* é armazenada em blocos de memória internos ao chip (BRAMs), ocupando 300Kb, dos 522Kb disponíveis. Ela tem capacidade para aproximadamente 5000 registros e é acessada pelos MicroBlazes através dos barramentos OPBs.

O plano de encaminhamento de dados, implementado no FPGA, tem dois componentes principais: IPv4 *header processor* e *packet payload transfer*. O primeiro é responsável pelo processamento dos dados do cabeçalho do pacote. O segundo é responsável

pela recepção, envio e armazenamento dos pacotes, e é também quem extrai o cabeçalho e encaminha-o para o *header processor*. O funcionamento geral e os detalhes construtivos de cada um são apresentados a seguir.

As transferências e recepções de pacotes, realizadas pelo *packet payload transfer*, são controladas pelos PowerPCs *Cores*. Durante a recepção, as interfaces GEMACs transferem os cabeçalhos e os pacotes de dados para a memória BRAM através do barramento PLB. Uma cópia do cabeçalho e um ponteiro para o pacote são passados para a memória compartilhada entre o PowerPC e o *header processor*, através do barramento *On-Chip Memory(OCM)*.

Existe um MicroBlaze *soft processor* para cada porta do roteador (*Source MB*). Ele lê cabeçalhos do OCM, encaminha-os para processamento e recebe-os de volta para escrevê-los no OCM de destino. A partir daí o PowerPC lê o cabeçalho da memória compartilhada remonta o pacote e transmite-o através da respectiva GEMAC.

O *header processor* é constituído de um *array* de *pipelines*. Cada *pipeline* é responsável pelo processamento de um cabeçalho diferente. Eles são constituídos de um *array* de 3 MicroBlaze *soft processor IPs* interligados por links FSL que transferem o cabeçalho entre os estágios do *pipeline*. O primeiro estágio é responsável pela verificação da integridade do cabeçalho IP. O segundo estágio realiza 3 dos seis acessos necessários para a descoberta da rota (*strides*: 12 4 4 4 4 4). O terceiro realiza os outros 3 acessos e mais o sétimo para descobrir a porta de destino. Uma vez que a porta é identificada o cabeçalho processador é enviado para o *Source MB* apropriado.

A capacidade de operação de cada pipeline é de 0.5 Gbps. A capacidade total do roteador é de 1.8 Gbps. Este valor é um pouco menor que o quádruplo da capacidade máxima de cada pipeline devido à concorrência de acesso à tabela de *lookup* na memória compartilhada.

Esta taxa de transmissão foi calculada multiplicando-se a taxa de pacotes transmitidos pelo tamanho do pacote. Para modelar o cenário de pior caso, as seguintes medidas foram adotadas: os pacotes possuíam tamanho de 64 bits (tamanho mínimo do quadro ethernet), todos os endereços da tabela de rotas possuíam 32 bits de comprimento (são necessários 7 acessos à memória para descobrir o next-hop), resultados da busca de rotas não são salvos em cache (busca da rota é realizada para cada pacote) e não foram considerados processamentos do plano de controle.

A tabela 7.3. apresenta os resultados comparativos encontrados no estudo. São mostrados os índices para as soluções baseadas em FPGA e NP.

	Xilinx Virtex-II Pro 2VP50	Intel IXP2800
Tecnologia (t micron)	0,13	0,13
Freqüência do clock (MHz)	100	1400
Área (A mm2)	130	280
Taxa Transmissão (T Gbps)	1,8	10
Taxa normalizada (T / (A / t2))	1	2,6

Tabela 10. Resultado comparativo da performance das soluções de roteamento.

O estudo sugere que a comparação entre os desempenhos das soluções deve ser feita tomando como base as taxas de transmissão normalizadas. Elas são normalizadas em relação à área (A) de silício utilizada para realizar a aplicação e a tecnologia construtiva

(t) empregada. A fórmula da normalização pode ser observada na última linha da tabela 7.3..

O Intel IXP2800 apresentou um desempenho 2,6 vezes melhor que o do Xilinx Virtex-II Pro. Isto se deve ao fato do IXP2800 ser um ASIC especialmente projetado para aplicações de roteamento de pacotes.

Contudo a vantagem das soluções baseadas em FPGAs fica evidente quando o custo total de fabricação de um produto é considerado. Este custo é composto pelo custo não recorrente de desenvolvimento e pelo custo recorrente por parte. O custo recorrente por parte está associado ao valor de aquisição final do produto. O estudo aponta que o custo por parte de cada solução gira em torno de $1000.

A grande diferença está no custo de desenvolvimento. O projeto de construção de um ASIC (0,18 micron) está na faixa dos $20 milhões. Contudo, FPGAs são dispositivos padronizados, projetos que utilizam-nos eliminam gastos com o desenvolvimento de circuitos integrados.

Projetos baseados em FPGAs apresentam baixo custo e menor tempo de desenvolvimento. Além disso, FPGAs são dispositivos reprogramáveis, isto facilita a incorporação de mudanças no projeto e desenvolvimento de novas funcionalidades.

O estudo sugere que no caso de não haver uma plataforma de desenvolvimento, baseada em ASIC, perfeitamente adaptada às necessidades do projeto, ou no caso de existirem restrições de tempo e/ou de custo, projetos baseados em FPGAs são boas soluções, ao preço de uma modesta perda de desempenho.

8. Conclusões

Como conclusão desse minicurso podemos identificar a necessidade do desenvolvimento de novas técnicas para construir equipamentos para taxas gigabit por segundo. É inegável também a necessidade de se repensar as arquiteturas dos equipamentos de comutação em camada 2 e 3. Outro ponto importante é que as redes óticas gigabit por segundo vão exigir cada vez mais implementações de mecanismos de encaminhamento de pacotes em hardware.

No entanto, também observamos que as rápidas mudanças na Internet inviabilizam a utilização de dispositivos de hardware fixos (não programáveis) como os ASICs. Por outro lado a implementação com software em processadores de uso geral também está descartada por causa do limitado desempenho oferecido por essa arquitetura.

Sendo assim, vislumbramos que os dispositivos mais adequados são os dispositivos de propósito específico programáveis para a construção dos novos equipamentos, como por exemplo: processadores de rede (NP) e FPGA. Ambos dispositivos possibilitam atingir o desempenho desejado, porém os NPs são mais complexos, isto é, permitem implementar mais funções e mais sofisticadas, enquanto os FPGAs são mais simples, possibilitando apenas funções mais básicas. Podemos observar, no entanto, que as últimas gerações de FPGAs tem capacidade de implementar funções bastante avançadas, se aproximando muito dos NPs.

Quanto a questão de custo, comparando dispositivos com capacidade semelhantes, os NPs são (e deverão ser) mais caros que os FPGAs. Caso a demanda por NPs aumente e

a consequente aumento da produção, é possível que eles fiquem com um preço semelhante aos FPGAs.

Os fabricantes de NPs dizem que devido a utilização de software nos NPs eles mais fáceis de programar e de dar manutenção no código, os FPGAs por outro lado, por ter um código muito próximo ao nível lógico (portas lógicas) tem uma dificuldade maior para o desenvolvimento. Em nosso estudo notamos que o kit de desenvolvimento dos principais fabricantes de NPs, geralmente baseado em C, são extremamentes específicos, exigindo programação específica para se obter o desempenho apregoado. O aproveitamento do código para NPs de outros fabricantes, ou mesmo outras linhas de produto de um mesmo fabricante, são praticamente impossíveis. Não nos pareceu ser possível o aproveitamento de código desenvolvido para um NP.

Por outro lado, no caso de FPGAs, geralmente especificado em linguagens de descrição como VHDL ou Verilog, apesar de ser considerada de mais baixo nível que o software de NP, já é largamente padronizada. Todos os fabricantes de FPGAs oferecem suporte para essas linguagens que são padronizadas. Para melhorar esse ambiente de programação, já estão sendo lançadas extensões dessas linguagens que implementam um nível mais alto de abstração, incluindo orientação a objetos e funções avançadas de programação: o SystemC e o SystemVerilog. Esse ambiente torna o kit de desenvolvimento FPGA muito semelhante aos kits de desenvolvimento de software de alto nível. A utilização de bibliotecas *IP Core* desenvolvidas pelos fabricantes de FPGA melhora o desempenho de funções críticas e facilita o desenvolvimento.

Concluindo, a implementação de técnicas de comutação e roteamento em dispositivos otimizados para obter taxas gigabit por segundo ainda necessita de muito estudo e a evolução será rápida em função da crescente adoção desses equipamentos pelas operadoras de telecomunicações. Essa área exige um conhecimento tanto de redes como especificação de hardware o que torna o desafio maior.

Referências

Accellera (2004). SystemVerilog 3.1a Language Reference Manual. www.systemverilog.org.

Ahmed, E. and Rose, J. (2000). The effect of lut and cluster size on deep-submicron fpga performance and density. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 3–12, Monterey, USA.

Aweya, J. (1997). IP Router Architectures: An Overview. Technical report, Nortel Networks, Ottawa - Canada.

Bailey, M., Gopal, B., and Pagels, M. (1994). PATHFINDER: A Pattern-Based Packet Classifier. In *Usenix OSDI*.

Baker, F. (1995). Requirements for IP Version 4 Routers. RFC 1812 (Proposed Standard). Updated by RFC 2644.

Bemmann, D. (2005a). IP Lookup on a Platform FPGA: A Comparative Study. In *Proc. of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 166.

Bemmann, D. (2005b). IP Lookup on a Platform FPGA: a Comparative Study. In *IPDPS*.

Brodnik, A., Carlsson, S., Degermark, M., and Pink, S. (1997). Small forwarding tables for fast routing lookups. In *SIGCOMM '97 Symposium*, pages 3–14. ACM.

Compton, K. and Hauck, S. (2002). Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (CSUR)*, 34(2):171–210.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). *Introduction To Algorithms*. MIT Press.

Crowley, P., Fiuczynski, M. E., Baer, J., and Bernshad, B. N. (2000). Characterizing Processor Architectures for Programmable Network Interfaces. In *International Conference on Supercomputing*.

Forum, N. P. (2006). http://www.npforum.org/.

Freitas, H. C. and Martins, C. A. P. S. (2000). Processadores de Rede: Conceitos, Arquiteturas e Aplicações. In *III Escola Regional Informática ES/RJ*.

Gayasen, A., Lee, K., Vijaykrishnan, N., Kandemir, M., Irwin, M. J., and Tuan, T. (2004). A dual-vdd low power fpga architecture. In *Proceedings of the International Conference on Field-Programmable Logic and its applications*.

Gupta, P. (2000). *Algorithms For Routing Lookups And Packet Classification*. PhD thesis, Stanford University.

Gupta, P. and McKeown, N. (1999). Packet Classification on Multiple Fields. In *SIGCOMM*, pages 147–160.

Gupta, P. and McKeown, N. (2001). Algorithms for packet classification.

Herity, D. (2005). Network processor programming. http://www.embedded.com/story/OEG20010730S0053.

Horta, E. L., Lockwood, J. W., Taylor, D. E., and Parlour, D. (2002). Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration. In *Proc of 39th Design Automation Conference (DAC'02)*, page 343.

Huston, G. (2006). Bgp routing table analysis reports. http://bgp.potaroo.net/.

Lakshman, T. V. and Stiliadis, D. (1998). High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching. In *SIGCOMM*, pages 203–214.

Lampson, B., Srinivasan, V., and Varghese, G. (1999). IP lookups using multiway and multicolumn search. *IEEE/ACM Trans. Netw.*, 7(3):324–334.

Leventis, P., Chan, M., and Chan, M. (2003). Cyclone: a low-cost, high-performance FPGA. In *Proceedings of the IEEE CICC 2003*, pages 49–52.

Lewis, D., Betz, V., and Jefferson, D. (2005). The Stratix II routing and logic architecture. In *Proc. ACM Int. Symp. Field-Programmable Gate Arrays*.

Limited, F. S. (2005). Chalanges in Building Network Processor Based Solution. White Paper. Technical report, Future Software Limited.

Lockwood, J. W., Naufel, N., Turner, J. S., and Taylor, D. E. (2001). Reprogrammable network packet processing on the field programmable port extender (fpx). In *ACM International Symposium on Field Programmable Gate Arrays*, pages 87–93.

Lockwood, J. W., Turner, J. S., and Taylor, D. E. (2000). Field Programmable Port Extender (FPX) for Distributed Routing and Queuing. In *Proceedings of the 2000 ACM/SIGDA 8th Symposium on Field Programmable Gate Arrays*.

Lysecky, R. and Vahid, F. (2004). A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. In *Design, Automation and Test in Europe (DATE'04)*, volume 1, pages 10480–10486.

Lysecky, R. and Vahid, F. (2005). A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning. In *Design, Automation and Test in Europe (DATE'05)*, volume 1, pages 18–23.

Marquardt, A., Betz, V., and Rose, J. (2000). Speed and area trade-offs on cluster-based fpga architectures. *IEEE Transaction on VLSI*.

Masud, M. I. (1999). Fpga routing structures: A novel switch block and depopulated interconnect matrix architecture. Master's thesis, University of British-Columbia.

Maxfield, C. (2004). *The Design Warrior's Guide to FPGAs (Ed Series for Design Engineers)*. Newnes.

Meng, D., Gunturi, R., and Castelino, M. (2003). Ixp2800 intel network processor ip forwarding benchmark full disclosure report for oc192-pos, intel corporation, tech. Intel Corporation, Tech. Report.

Mondal, S. and Memik, S. O. (2005). A Low Power FPGA Routing Architecture. In *IEEE International Symposium on Circuits and Systems (ISCAS 2005)*, Kobe, Japan.

Morrison, D. (1968). PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *JACM*, 15.4:514 – 534.

Newman, P., Minshall, G., Lyon, T., and Huston, L. (1997). IP Switching and Gigabit Routers. *IEEE Communications Magazine*.

Nilsson, S. and Karlsson, G. (1999). IP-Address Lookup Using LC-Tries. *IEEE Journal on Selected Areas in Communication*.

Qiu, L., Varghese, G., and Suri, S. (2001). Fast firewall implementations for software and hardware-based routers. In *IEEE ICNP 2001*.

Ravindran, K., Satish, N., Jin, Y., and Keutzer, K. (2005). An FPGA-Based Soft Multiprocessor System for IPv4 Packet Forwarding. In *Proc. of 15th International Conference on Field Programmable Logic and Applications (FPL-05)*, pages 487–492.

Rose, J., Francis, R., and P. Chow, D. L. (1989). The Effect of Logic Block Complexity on Area of Programmable Gate Arrays. In *Proc. of IEEE Custom Integrated Circuits Conference*, pages 531–535.

Ruiz-Sanchez, M., Biersack, E., and Dabbous, W. (2001). Survey and taxonomy of IP address lookup algorithms. *IEEE Network*, 15(2):8–23.

Shah, N. (2001). Understanding Network Processors. Master's thesis, Dept. of Electrical Engineering and Computer Sciences, Univ. of California, Berkeley.

Shah, N. and Keutzer, K. (2002). Network processors: Origin of species. In *The Seventeenth International Symposium on Computer and Information Sciences*.

Sharma, A., Compton, K., Ebeling, C., and Hauck, S. (2004). Exploration of pipelined FPGA interconnect structures. In *Proceeding of the 2004 ACM/SIGDA 12th international Symposium on Field Programmable Gate Arrays*, Monterey, USA.

Sica, C. F. and Torres, G. G. (2005). Notas sobre a evolução dos Processadores de Rede (network processors): um estudo preliminar. http://www.lecom.dcc.ufmg.br/cursos/.

Singh, S., Baboescu, F., Varghese, G., and Wang, J. (2003). Packet classification using multidimensional cutting.

Skiena, S. (1997). *The Algorithm Design Manual*. Springer-Verlag.

Smith, D. J. (1996). VHDL and Verilog Compared and Contrasted - Plus Modeled Example Written in VHDL,Verilog and C. In *Annual Conference on Design Automation (DAC 96)*.

Spalink, T., Karlin, S., and Peterson, L. (2000). Evaluating Network Processors in IP Forwarding.

Srinivasan, V., Varghese, G., Suri, S., and Waldvogel, M. (1998). Fast Scalable Level Four Switching. In *SIGCOMM*, pages 191–202.

Systems, A. (2000). The Challenge for Next Generation Network White Paper. Technical report, Agere, www.agere.com.

Systems, A. (2003). The Case for a Classification Language. Technical report, Ucent Technologies.

Taylor, D. E., Lockwood, J. W., Sproull, T. S., Turner, J. S., and Parlour, D. B. (2002). Scalable IP Lookup for programmable routers. In *IEEE INFOCOM 2002 - The Conference on Computer Communications*, volume 21, pages 562–570.

Taylor, D. E., Turner, J. S., and Lockwood, J. W. (2001). Dynamic Hardware plugins (DHP): exploiting reconfigurable hardware for high-performance programmable routers. In *IEEE Open Architectures and Network Programming Proceedings*, pages 25–34.

Telikepalli, A. (2005). Power vs. performance: The 90 nm inflection point, version 1.1. www.xilinx.com.

Tessier, R. and Burleson, W. (2001). Reconfigurable Computing for Digital Signal Processing: A Survey. *Journal of VLSI Signal Processing*.

Varghese, G. (2005). *Network Algorithmics: An Interdisciplinary Approach to design Fast Networked Devices*. Morgan Kaufmann.

Waldvogel, M., Varghese, G., Turner, J., and Plattner, B. (1997). Scalable High Speed IP Routing Lookups. In *Proceedings of ACM SIGCOMM '97*, pages 25–36.

Woo, T. (2000). A modular Approach To Packet Classification: Algorithms and results. In *INFOCOM 2000*.

Xilinx (2004). RocketIO X Transceiver User Guide - version 1.5. www.xilinx.com.

Xilinx (2005). Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet - version 4.5. www.xilinx.com.