Escalonamento Baseado em Algoritmos Genéticos para Roteadores de Alto Desempenho

José Ricardo Hoffmann e Emilio Carlos Gomes Wille

***Resumo*— Os roteadores modernos empregam sofisticados comutadores para a transmissão de pocotes. A arquitetura de comutadores com filas de entrada exige um processo de escalonamento que estabelece a transferência de pacotes das portas de entrada às portas de saída. O desempenho do sistema depende diretamente do algoritmo de escalonamento, considerando sua vazão e complexidade. Este trabalho apresenta uma abordagem de escalonamento usando algoritmos genéticos. O algoritmo proposto, denominado GENIUS, apresenta baixa complexidade e os resultados alcançados mostram que o desempenho do algoritmo é próximo ao ótimo.**

***Palavras-Chave*— Roteadores, Comutador com filas de entrada, Algoritmo de escalonamento, Algoritmos Genéticos.**

***Abstract*— Modern routers employ input-queued crossbar switches that require sophisticated scheduling techniques for packet transmission. The performance of the router then directly depends on the scheduling algorithm, considering its throughput and complexity. In this paper, a new genetic-based scheduling algorithm is developed and tested for high-performance Internet router operation. The proposed algorithm, called GENIUS, presents low complexity and the results show a performance close to the optimal.**

***Keywords*— Router, Input-Queued switches, Scheduling algorithms, Genetic Algorithms.**

I. INTRODUÇÃO

A Internet é uma grande rede de comutação de pacotes, construída em torno de uma grande variedade de sistemas de transmissão e comutação. Os sistemas de comutação mais importantes da Internet são os roteadores. As principais funções de um roteador são receber os pacotes das portas de entrada, encontrar o destino adequado com base na tabela de encaminhamento e transferir os pacotes para as portas de saída. Estas duas funções básicas, roteamento e comutação, são difíceis de serem implementadas quando a taxa de transferência exigida é muito alta, uma vez que algoritmos complexos devem ser executados em um espaço de tempo muito curto. O desenvolvimento de roteadores de alto desempenho é particularmente importante, dado que a Internet, hoje, é composta por um número relativamente pequeno de redes dorsais muito rápidas, que interconectam um número muito grande de redes menores.

Os comutadores (*switches*) possuem uma arquitetura composta por *buffers* e pelo elemento de comutação (*switch fabric*). A Figura 1 mostra a estrutura lógica de um comutador com filas de entrada (*input-queued switch*) [1]. O comutador opera com unidades de dados de tamanho fixo (chamadas neste trabalho de pacotes). O principal gargalo de desempenho é dado pelo algoritmo de escalonamento, que seleciona pacotes a serem transferidos pelo elemento de comutação a cada intervalo de tempo chamado de *time slot*.

J. R. Hoffmann e E. C. G. Wille, Universidade Tecnológica Federal do Paraná. Av. Sete de Setembro 3165, 80230-901, Curitiba (PR), Brasil. E-mail: josericardo.hoffmann@gmail.com, ewille@utfpr.edu.br

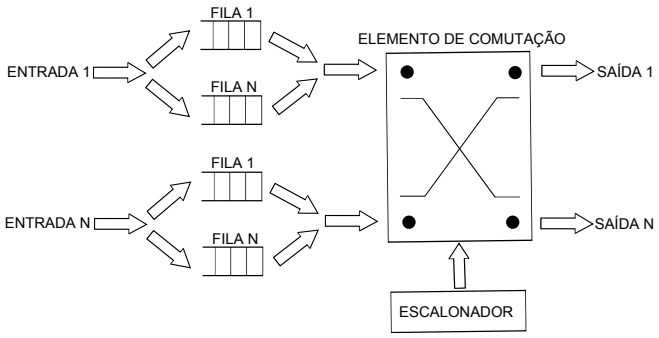

Fig. 1. Estrutura lógica de um comutador com filas de entrada.

Considera-se um comutador com N entradas e N saídas. Assume-se que todas as linhas de entrada e saída atuam com a mesma velocidade. Os pacotes são armazenados nas interfaces de entrada. Cada entrada gerencia uma fila para cada saída, portanto, um total de $N \times N$ filas são necessárias. Cada fila (*drop-tail*) pode armazenar até M pacotes. Esta separação de filas, chamada VOQ (*Virtual Output Queuing*), permite evitar degradações de desempenho devido ao bloqueio HoL (*Head-of-Line*) [2]. O elemento de comutação é sem bloqueio, sem memória, e sem atraso: no máximo, um pacote pode ser removido de cada entrada e, no máximo, um pacote pode ser transferido para cada uma das saídas em cada *time slot*. O algoritmo de escalonamento é responsável por decidir quais pacotes serão transferidas, a partir das entradas, para as saídas do comutador.

Os algoritmos genéticos (AGs) são exemplos de métodos de computação evolucionária usados para busca, otimização e aprendizado de máquina. Neste artigo, suas características naturais de busca em ambientes dinâmicos (*changing environments*) [3] são exploradas na construção de um algoritmo de escalonamento de ótimo desempenho denominado GENIUS (*Genetic Scheduling Algorithm for High-Performance Switches*). Duas versões são propostas e analizadas: a versão elitista GENIUS-E, visando alcançar um desempenho próximo ao ótimo, e a versão simplificada GENIUS-S de complexidade reduzida.

Este artigo está organizado da seguinte forma. A Seção II apresenta a arquitetura e o funcionamento do comutador bem como a notação matemática adotada. A Seção III define os algoritmos de escalonamento relevantes e relacionados ao proposto, e seus respectivos funcionamentos. A Seção IV descreve detalhadamente o escalonador proposto e suas

versões. Na Seção V, resultados numéricos são apresentados e discutidos. As conclusões são apresentadas na Seção VI.

II. Modelo de Comutador

Este trabalho considera um comutador com $N \times N$ filas de entrada. O *buffer* na entrada i é particionado em N "filas virtuais de saída" ($VOQs$), onde VOQ_{ij} armazena pacotes na entrada i para a saída j. Denota-se a dimensão de VOQ_{ij} no instante t por $q_{ij}(t)$. Sendo $Q(t) = [q_{ij}(t)]$ uma matriz $N \times N$ contendo a dimensão de todas as filas $VOQs$ no instante t.

Seja λ_{ij} a taxa média na qual os pacotes chegam na entrada i para a saída j, e $\Lambda = [\lambda_{ij}]$ a matriz de taxa de chegada, também chamada matriz de carga. Exige-se uma carga admissível, ou seja, $\sum_j \lambda_{ij} \leq 1$ para todo i, e $\sum_i \lambda_{ij} \leq 1$ para todo j. Em outras palavras, esta situação assegura que nenhuma entrada ou saída esteja sobrecarregada. A carga máxima de entrada é dada por $\rho = max_i(\sum_j \lambda_{ij})$.

Utilizam-se variáveis binárias $x_{ij}(t), i, j = 1, \ldots, N$, para representar conexões. A entrada i está conectada com a saída j no momento t, se e somente se, $x_{ij}(t) = 1$. Sem perda de generalidade, consideram-se apenas ligações completas, isto é, permite-se uma conexão entre a entrada i e saída j, mesmo se $q_{ij}(t) = 0$.

Uma configuração de conexão admissível pode ser vista como um casamento (*matching*) em um grafo bipartido. Entradas e saídas correspondem aos nós do grafo, e uma aresta entre entrada i e saída j indica uma conexão. Seja $X(t) = [x_{ij}(t)]$ a matriz de matching no instante t. Define-se o peso do matching $X(t) = [x_{ij}(t)]$ como $W(t) = \sum_{ij} q_{ij}(t)x_{ij}(t)$, onde o peso da aresta entre a entrada i e a saída j é igual ao tamanho da fila $q_{ij}(t)$. Para um comutador $N \times N$, o conjunto de todos os possíveis matchings, indicados por S, tem cardinalidade $N!$. É função do algoritmo de escalonamento determinar, em cada momento t, o matching particular a ser utilizado.

III. Algoritmos de Escalonamento

A. Algoritmo MWM

Um algoritmo de escalonamento de interesse particular é o *Maximum Weighted Matching* (MWM). O escalonador MWM escolhe, em cada instante t, o matching de maior peso. Mais precisamente, se $X^w(t)$ representa o matching determinado pelo MWM no instante t, então $X^w(t)$ é dado por:

$$X^w(t) = arg\ max_{X \in S_n} \{\sum_{i,j} x_{ij} q_{ij}(t)\} \qquad (1)$$

O algoritmo de escalonamento MWM garante 100% de transferência (*throughput*) para todos os padrões de tráfego (Bernoulli) admissíveis de entrada [4], [5]. A literatura mostra que o escalonador MWM apresenta valores baixos de atraso médio (por manter tamanhos de fila pequenos), entretanto, por ser o problema resolvido mediante aplicação do algoritmo Húngaro [6], apresenta complexidade temporal $O(N^3)$.

B. Algoritmo APSARA

O algoritmo de escalonamento APSARA (*A Parallel Scheduling Algorithm and its Random Aproximation*) foi proposto em [7], e por ser baseado em uma população de soluções é tomado como base de comparação neste trabalho.

Seja $X(t)$ o matching determinado pelo algoritmo APSARA no instante t e seja $Q(t+1)$ o tamanho de fila no início do instante t+1. No instante t+1, o APSARA procede da seguinte forma:

- Determina os vizinhos, $\mathcal{N}[X(t)]$, de $X(t)$ e o matching $Z(t+1)$ que corresponde a um caminho Hamiltoniano no instante $t+1$.
- Seja $S(t+1) = \mathcal{N}[X(t)] \cup Z(t+1) \cup X(t)$. Calcula o peso de cada matching $Y \in S(t+1)$ como $W(Y) = \sum_{ij} y_{ij} q_{ij}(t+1)$.
- Determina o matching no instante $t+1$ por $X^w(t+1) = arg\ max_{U \in S_{(t+1)}} \{W(U)\}$.

O algoritmo básico APSARA-B, requer o cálculo do peso para os matchings vizinhos. Esse cálculo é simples, porque um vizinho Y difere do matching $X(t)$ em exatamente duas arestas. No entanto, o cálculo dos pesos de todos os $\binom{N}{2}$ vizinhos, se for feito em paralelo como mostrado na Figura 2, necessitará de grande quantidade de espaço em hardware para grandes valores de N.

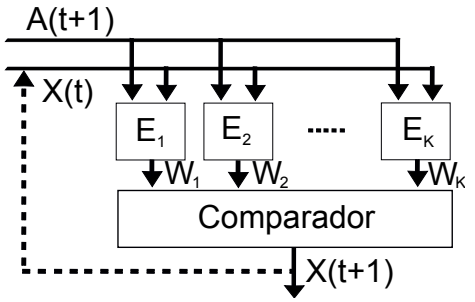

Fig. 2. Esquema para implementação do APSARA [7].

Nota-se que, para a construção de comutadores com grande número de portas, o número de módulos em paralelo faria o sistema proibitivamente caro. Uma solução consiste em manter um máximo de K ($<< N^2$) vizinhos escolhidos de forma aleatória a partir do conjunto $\mathcal{N}[X(t)]$. Essa consideração dá origem ao algoritmo randomizado APSARA-R [8], [9].

C. Max-APSARA

O APSARA gera todos os matchings no conjunto de vizinhança independentemente do tamanho das filas. O tamanho das filas é usada somente para selecionar o matching mais pesado da vizinhança. Por este motivo, é possível que o matching determinado pelo APSARA seja o melhor dentre os vizinhos, mas não o matching ótimo. Isto é, existe uma entrada i, que possui dados para a saída j, mas o matching $X(t)$ conecta a entrada i para alguma outra saída j' e conecta a saída j para alguma outra entrada i', e ambos $q_{ij'}(t)$ e $q_{i'j}(t)$ são nulos. Assim, a entrada i e a saída j não estão sendo corretamente conectadas. Deve-se, então, completar o

PROCEDIMENTO *Escalonador Genético*( );
(1) **INICIO**
(2) Insere parâmetros do sistema;
(3) Inicializa população aleatoriamente;
(4) Avalia o fitness de cada indivíduo;
(5) **PARA** cada *time slot* **FAÇA**
(6) Aplica seleção de pais;
(7) Aplica crossover com probabilidade p_c;
(8) Aplica mutação com probabilidade p_m;
(9) Avalia o fitness dos filhos produzidos;
(10) Atualiza população;
(11) *matching* = melhor indivíduo;
(12) **FIM PARA**
(13) **FIM**

Fig. 3. Pseudocódigo do algoritmo GENIUS.

matching determinado pelo APSARA, transformando-o em um matching de peso máximo. Essa opção é chamada de Max-APSARA [1].

IV. O Escalonador Genético

Os algoritmos genéticos (AGs) são heurísticas de otimização estocásticas, onde explorações no espaço de soluções são conduzidas imitando a genética de população estabelecida na teoria da evolução de Darwin. Os princípios básicos dos AGs foram estabelecidos de forma detalhada por Holland [10]. Para utilizar um algoritmo genético, é necessário representar uma solução para um problema como um *cromossomo*. *Seleção*, *operadores genéticos* e *substituição*, diretamente derivados dos mecanismos de evolução natural são, então, aplicados a uma população de soluções favorecem o nascimento e sobrevivência das melhores soluções.

Neste artigo, as características naturais de busca em ambientes dinâmicos dos AGs são exploradas na construção de um algoritmo de escalonamento de ótimo desempenho denominado GENIUS. A descrição em pseudo-código do escalonador proposto é mostrada na Figura 3.

A população compreende um grupo de K indivíduos (cromossomos), dos quais candidatos podem ser selecionados para a solução de um problema. Inicialmente, uma população é gerada aleatoriamente. Os valores de *fitness* de todos os cromossomos são avaliados através do cálculo da função objetivo. Um grupo particular de cromossomos (pais) é selecionado à partir da população para gerar os filhos através das operações genéticas *crossover* e *mutação*. A aptidão dos filhos é avaliada de forma semelhante a de seus pais. Os indivíduos da população corrente são substituídos pelos seus descendentes, com base em uma estratégia de substituição. A evolução da população avança ao longo da execução do escalonador, e a cada *time slot* o melhor indivíduo da população corresponde ao *matching* a ser utilizado pelo comutador.

A. Codificação e Operadores Genéticos

Os detalhes específicos associados ao GENIUS são apresentados nos parágrafos seguintes.

Esquema de codificação: Nesse trabalho, um *matching* é um vetor de inteiros, de tamanho N, usado para representar a correspondência de portas de entrada com portas de saída. Assim, um cromossomo corresponde, através dos índices, à porta de entrada e, através do conteúdo, à uma sequência de N portas de saída, como na Figura 4.

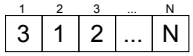

Fig. 4. Representação do cromossomo.

Avaliação do fitness: O fitness de cromossomo corresponde simplesmente ao peso do matching definido na seção III.

Seleção de pais: A seleção dos pais emula o mecanismo natural de "sobrevivência do mais apto". Neste trabalho foi utilizado o método de seleção torneio de tamanho k, onde k indivíduos são escolhidos aleatoriamente e aquele com o maior fitness (o que "vence o torneio") é usado como um pai. A seleção torneio é então repetida com uma nova seleção de k indivíduos a fim de encontrar outro pai de características diferentes [11], [12].

Operadores genéticos: O *crossover* é um operador de recombinação usado para produzir filhos. Neste trabalho é utilizado o crossover OX *(Order Crossover)* de dois pontos [13]. Dado dois pais, a partir do segundo ponto de corte de um dos pais, descreve-se a sequência e posteriormente remove-se, desta, os vértices existentes entre os cortes do filho, ao final os filhos herdam as sub-sequências entre os pontos de corte. Estes pontos são selecionados aleatoriamente. O processo é exemplificado na Figura 5.

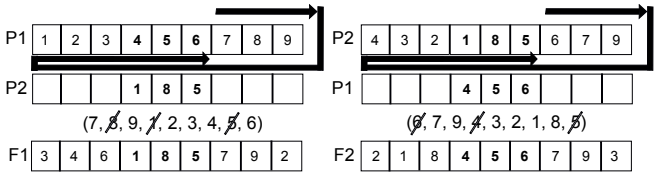

Fig. 5. Crossover OX (*Order Crossover*).

A *mutação* é usada para evitar a convergência das soluções para ótimos locais de baixa qualidade. Nesta proposta foram obtidos bons resultados usando o operador de mutação SIM (*Simple Inversion Mutation*) que faz seleção aleatória de pontos de mutação, e inverte os conjuntos de genes entre estes pontos para cada descendente produzido [10][14]. Considerando apenas um ponto de mutação a Figura 6 exemplifica a operação.

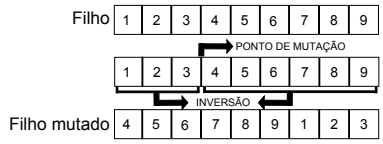

Fig. 6. Mutação SIM (*Simple Inversion Mutation*).

Estratégia de substituição 1: A fim de gerar uma nova população foi utilizada a *estratégia elitista* onde uma vez que a população de filhos é gerada, ela é mesclada com a população de pais segundo a seguinte regra: somente os melhores indivíduos presentes nas populações de pais e filhos entram na nova população. Para isso foi utilizado o

algoritmo de classificação *heapsort*, que em seu pior caso apresenta uma complexidade computacional $O(K\ logK)$ [15]. Essa estratégia é aplicada na versão elitista (GENIUS-E) do escalonador proposto.

Estratégia de substituição 2: A fim de reduzir a complexidade do algoritmo de escalonamento, a geração da nova população é através da comparação imediata realizada entre filhos e pais, e segue a seguinte regra: os melhores indivíduos presentes após a reprodução entre pais e filhos entram na nova população. Como não há qualquer classificação nesta estratégia a complexidade computacional exigida é $O(K)$. Essa estratégia é aplicada na versão simplificada (GENIUS-S) do escalonador proposto.

Tamanho da população: O tamanho da população K foi definido de forma empírica de modo a não elevar demasiadamente a complexidade do escalonador.

V. Análise de Desempenho

Nesta seção são apresentados resultados numéricos, obtidos via simulação (com base no método *batch means* [16]), considerando-se os algoritmos: GENIUS, APSARA e MWM.

A. Configurações da simulação

Comutador: Foi considerado um comutador com $N = 32$ portas. Cada VOQ pode armazenar até 10.000 pacotes, sendo que, pacotes em excesso são descartados.

Tráfego de entrada: A carga para todas as entradas foi normalizada, sendo $\rho \in (0, 1)$ seu valor. O processo de chegada é independente e identicamente distribuído seguindo um processo de Bernoulli. Foram utilizadas as seguintes matrizes de carga, onde $|i| = (i\ mod\ N)$, para testar o desempenho dos algoritmos:

- *Tráfego uniforme*: Neste caso $\lambda_{ij} = \rho/N\ \forall i, j$. Este é o teste de tráfego mais comumente usado na literatura.
- *Tráfego diagonal*: Neste caso $\lambda_{ii} = 2\rho/3N,\ \lambda_{i|i+1|} = \rho/3N\ \forall i$, and $\lambda_{ij} = 0$ para qualquer outro i e j. Esta é uma carga desbalanceada no sentido de que a entrada i possui pacotes para as saídas i e $|i + 1|$. É mais difícil tratar esse tráfego do que o tráfego uniforme.

B. Estudo de desempenho

Neste item são apresentados e analisados os resultados obtidos, explorando o espaço de parâmetros dos algoritmos de escalonamento. Foram comparados os desempenhos dos diferentes algoritmos, medindo os tamanhos médios das filas de entrada ($E[L]$). Claramente, quanto menor o valor de $E[L]$ melhor será o desempenho do sistema. O GENIUS foi configurado da seguinte forma: $p_c = 100\%,\ p_m = 65\%$ e $k = 0,25 \times K$.

1) Variação da População: Observando as Figuras 7 e 8 nota-se que, de forma geral, quanto maior a carga ofertada maior é a ocupação das filas de entrada. Com o aumento do tamanho da população há um aumento de desempenho dos escalonadores GENIUS e APSARA, como esperado. O escalonador GENIUS-S apresentou desempenho equivalente ao APSARA-R32. Entretanto o escalonador GENIUS-E apresentou desempenho superior ($K = 496$), principalmente quando se observa o tráfego diagonal.

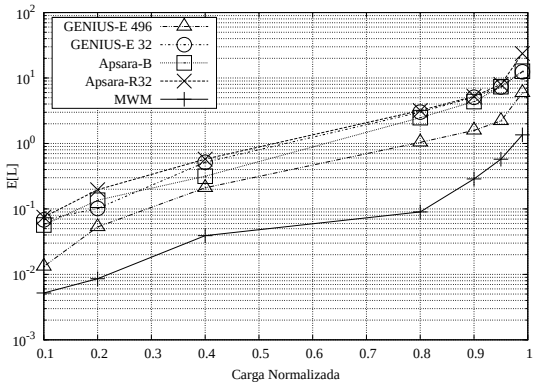

Fig. 7. Tráfego Uniforme – Variação do tamanho da população.

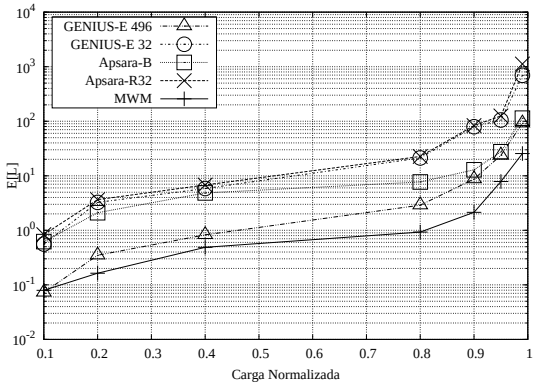

Fig. 8. Tráfego Diagonal – Variação do tamanho da população.

2) Impacto da Redução de Complexidade: As Figuras 9 e 10 comparam o desempenho dos escalonadores elitista (GENIUS-E) e simplificado (GENIUS-S), considerando somente o tráfego diagonal (mais problemático). O GENIUS apresentou um desempenho semelhante ou superior ao escalonador APSARA para tamanho de população $K = 496$. Para população com $K = 32$ as duas versões do escalonador proposto perdem desempenho.

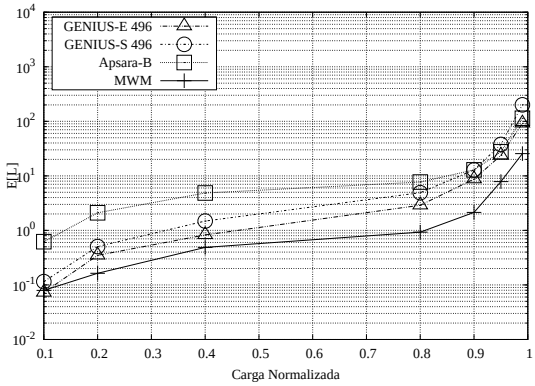

Fig. 9. Tráfego Diagonal – Alteração de complexidade.

3) Análise da Versão MAX: Para versões MAX dos escalonadores as Figuras 11 e 12 mostram desempenhos semelhantes entre GENIUS e APSARA. Como esperado, os algoritmos apresentam desempenho próximo do ótimo

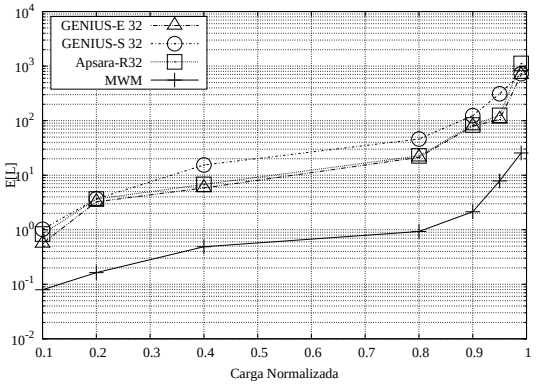

Fig. 10. Tráfego Diagonal – Alteração de complexidade.

(MWM). Entretanto, com a redução da população para $K = 32$ observa-se um afastamento do ótimo, principalmente para cargas mais elevadas.

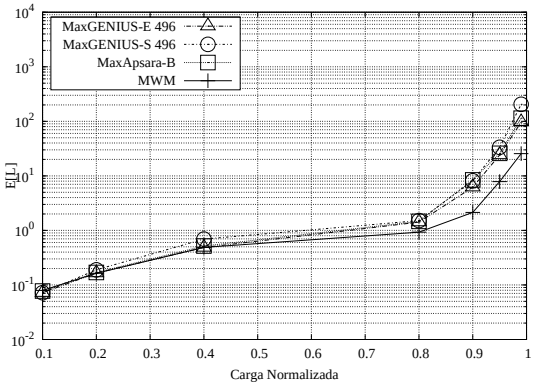

Fig. 11. Tráfego Diagonal – Versão MAX.

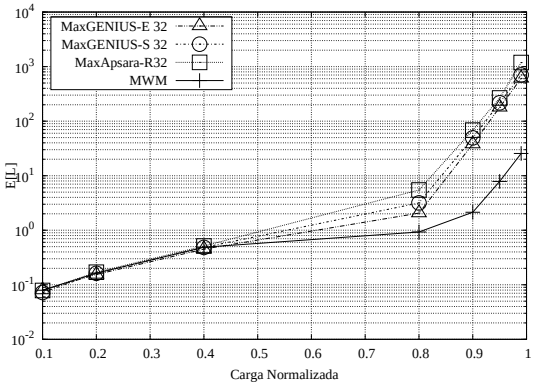

Fig. 12. Tráfego Diagonal – Versão MAX.

VI. CONCLUSÃO

Este trabalho apresentou uma abordagem para criação de um escalonador para comutadores com filas de entrada através de algoritmos genéticos. Com a aplicação do escalonador GENIUS foram encontrados ótimos resultados em relação a outros escalonadores relevantes.

Um ponto a ser esclarecido diz respeito a implementação física da proposta. O escalonador APSARA apresenta complexidade temporal $O(N)$ e complexidade espacial $O(K)$. Um implementação "tradicional" do escalonador GENIUS levaria a uma complexidade temporal mais elevada, em função da característica serial dos Algoritmos Genéticos. Entretanto, pode-se usar o conceito de Algoritmo Genético em "pipe-line" [17] onde K operações genéticas são realizadas ao mesmo tempo sobre uma população de K indivíduos produzindo K filhos. A operação de substituição é realizada após esta etapa para a criação da nova população. Desta forma, o GENIUS passaria a apresentar uma complexidade espacial $O(K)$ e complexidade temporal descrita pela Tabela I.

TABELA I
COMPLEXIDADES DAS VERSÕES DO ALGORITMO GENIUS.

Versão	$K = \binom{N}{2}$	$K = N$
GENIUS-S	$O(N)$	$O(N)$
GENIUS-E	$O(N^2\ logN)$	$O(N\ logN)$

Finalmente, o escalonador GENUIS-S é particularmente interessante por apresentar complexidade e desempenho equivalentes ao APSARA, podendo-se afirmar, portanto, que é um escalonador competitivo.

REFERÊNCIAS

[1] P. Giaccone, *Queueing and scheduling algorithms for high performance routers*, Tese de Doutorado, Politecnico di Torino, 2002.
[2] M. Karol, M. Hluchyj e S. Morgan, *Input versus output queuing on space division switch*, IEEE Trans. on Communications, Vol. 35, n. 12, 1987.
[3] J. J. Grefenstette, *Genetic Algorithms for Changing Environments*, Parallel Problem Solving from Nature 2, 1992.
[4] L. Tassiulas e A. Ephremides, *Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks*, IEEE Trans. on Automatic Control, 1992.
[5] N. McKeown, V. Anantharan e J. Walrand, *Achieving 100% throughput in an input-queued switch*, IEEE INFOCOM'96, San Francisco, 1996.
[6] H. W. Kuhn, *The Hungarian method for the assignment problem*, In Naval Research Logistics (NRL), Vol. 52, 7–21, 2005.
[7] P. Giaccone , D. Shah e B. Prabhakar, *An Implementable Parallel Scheduler for Input-Queued Switches*, In IEEE Micro, 2001.
[8] D. Shah, P. Giaccone e B. Prabhakar, *Efficient Randomized Algorithms for Input-Queued Switch Scheduling*, In IEEE Micro, 2002.
[9] P. Giaccone , B. Prabhakar e D. Shah, *Randomized Scheduling Algorithms for High-Aggregate Bandwidth Switches*, In IEEE Journal on Selected Areas in Communications, Vol. 21, 546–559, 2003.
[10] J. H. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, 1975.
[11] D. E. Goldberg, D. Thierens, e K. Deb, *Toward a better understanding of mixing in genetic algorithms*, Journal of the Society of Instrument and Control Engineers, 1993.
[12] B. L. Miller e D. E. Goldberg, *Genetic Algorithms, Tournament Selection, and the Effects of Noise*, IlliGAL Report No. 95006, July, 1995.
[13] M. C. Goldbarg e H. P. L. Luna, *Otimização Combinatória e Programação Linear: modelos e algoritmos*, Rio de Janeiro: Editora Campus, 2000.
[14] J. J. Grefenstette, *Incorporating Problem Specific Knowledge into Genetic Algorithms*, Los Altos, CA, 1987.
[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, *Introduction to Algorithms (2nd ed.)*, MIT Press, 2003.
[16] J. Banks, *Simulation - Principles, Methodology, Advances, Applications, and Pratice*, EMP, 1998.
[17] K. Pakhira e R. K. De, *Generational PipeLined Genetic Algorithm (PLGA) using Stochastic Selection*, Int. Journal of Computer Systems Science and Engineering, 2007.