Sistemas de Armazenamento Compartilhado com Qualidade de Serviço e Alto Desempenho

Pedro Eugênio Rocha, Luis Carlos Erpen de Bona

Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 – CEP 81.531-980 – Curitiba – PR – Brazil

{pedro,bona}@inf.ufpr.br

Abstract. *Shared storage systems must provide performance isolation so that QoS guarantees are not affected by concurrent applications. This paper proposes a new non-work-conserving disk scheduling algorithm called HTBS, which is based on BFQ and pClock. HTBS merges these algorithms in order to provide QoS guarantees without decreasing the system overall performance. We show through experiments that HTBS can provide both high-throughput and QoS guarantees to applications with different requirements.*

Resumo. *Servidores de armazenamento compartilhado devem possuir mecanismos que forneçam isolamento de performance, de modo que as garantias de QoS não sejam influenciadas por aplicações concorrentes. Este trabalho propõe um novo algoritmo de escalonamento de disco não-conservativo, chamado HTBS, baseado nos algoritmos BFQ e pClock. O HTBS utiliza partes destes algoritmos com o objetivo de fornecer garantias de QoS, sem que a performance do disco seja degradada. Mostramos através de experimentos que o HTBS pode atender aplicações com requisitos variados e ao mesmo tempo fornecer alto desempenho.*

1. Introdução

A centralização da capacidade de armazenamento em servidores dedicados é um paradigma cada vez mais utilizado no gerenciamento de dados organizacionais. Dentre os diversos benefícios trazidos por essa centralização estão a redução de custos e complexidade de manutenção, utilização otimizada do recurso, simplificação no gerenciamento e em políticas de backup e flexibilidade para a alocação de espaço de armazenamento. Diante deste paradigma e da popularização da computação em nuvens, algumas empresas (como Amazon [Amazon EC2 2011] e Google [Google 2011]) criaram serviços para terceirização da administração desses servidores de armazenamento – serviços conhecidos como *Storage as a Service*. Por ser fortemente baseada em virtualização, a computação em nuvens também aumentou a necessidade por mecanismos que possibilitem a criação de infraestruturas de armazenamento virtualizadas.

Para que esta centralização seja viável, é desejável que o servidor de armazenamento possa garantir atributos de QoS aos seus usuários, definidos em documentos chamados SLA's (*Service Level Agreements*) [Vaquero et al. 2009], que especifiquem e mensurem as garantias fornecidas. Tais atributos são baseados em

conceitos de Qualidade de Serviço ou QoS, como largura de banda, latência e rajadas. O sistema de armazenamento deve também prover *isolamento de performance* [Seelam e Teller 2006], ou seja, garantir que o desempenho percebido por um usuário não seja influenciado por aplicações concorrentes.

A maioria dos sistemas de armazenamento atuais são baseados em discos rígidos, ainda que camadas intermediárias, como RAID (*Redundant Array of Independent Disks)* ou LVM *(Logical Volume Manager*) [LVM 2011], sejam utilizadas. Neste tipo de dispositivo é difícil fornecer garantias de QoS, pois o tempo necessário para atender requisições de leitura ou escrita é altamente variável, dependendo de fatores como *seek time,* latência de rotação, posição dos dados na superfície do disco e caches [Jian et al. 2009]. A ordem em que as requisições são executadas também tem grande impacto no desempenho do dispositivo. Ainda assim, podem existir diversas aplicações com características e requisitos de QoS variados competindo pela utilização do disco.

Outro problema enfrentado por este tipo de dispositivo, é o fato de que grande parte das aplicações cria apenas uma requisição por vez, de forma que a criação de uma próxima requisição dependa do término da anterior [Valente e Checconi 2010]. Estas requisições são conhecidas como síncronas. Escalonadores baseados em *timestamps* ou *tags* [Gulati et al. 2007]*,* em geral selecionam a próxima requisição a ser atendida com base no tempo de chegada, que pode ser afetado pelo atraso no atendimento da requisição anterior. Assim, o atraso no atendimento de uma requisição de uma aplicação pode causar atraso em todas as requisições seguintes, prejudicando as garantias fornecidas.

Como há relação de dependência entre requisições síncronas de uma mesma aplicação, existe um curto intervalo de tempo entre o término do atendimento de uma requisição e a criação da próxima, durante o qual a aplicação é dita *deceptive idle* [Iyer 2003], ou enganosamente ociosa. Neste intervalo, como ainda não há novas requisições pendentes da mesma aplicação, um algoritmo de escalonamento conservativo serviria outras aplicações, e assim sucessivamente. Este comportamento, conhecido como *deceptive idleness*, prejudica o desempenho do disco por não executar requisições com a localidade inerente a requisições de uma mesma aplicação.

Neste trabalho é proposto um novo algoritmo de escalonamento de disco chamado HTBS *(High-throughput Token Bucket Scheduler),* baseado em dois algoritmos propostos anteriormente: BFQ (*Budget Fair Queueing*) [Valente e Checconi 2010] e pClock [Gulati el at. 2007]. O HTBS tem o objetivo de fornecer garantias de QoS, causando o menor impacto possível no desempenho do disco. O algoritmo pClock é utilizado pois provê bom isolamento de performance (como mostrado em [Gulati et al. 2007]), além da possibilidade de ajustar os parâmetros largura de banda, rajadas e latência de forma independente. Apesar disso, mostramos através de experimentos que o algoritmo pClock falha em fornecer as garantias na presença de requisições síncronas. O algoritmo proposto incorpora os mecanismos para tratar requisições síncronas e prevenir a ocorrência de *deceptive idleness* presentes no algoritmo BFQ, que, por sua vez, não permite a configuração dos atributos largura de banda e latência de forma individual.

O algoritmo HTBS foi implementado como um módulo para o Kernel Linux

[Linux 2011], e experimentos foram realizados utilizando a ferramenta de *benchmark* de disco *fio* [Axboe 2011]. Através dos testes realizados, mostramos que o algoritmo HTBS aumenta o desempenho geral do disco na presença de aplicações que utilizem requisições síncronas e sequenciais, se comparado ao algoritmo pClock, provendo isolamento de performance e sem degradar de maneira significativa a latência das requisições. Mostramos também que o HTBS fornece desempenho próximo ao BFQ que possui mecanismos para prevenção de *deceptive idleness*, mas não possibilita a configuração dos atributos largura de banda e latência de forma explícita e individual.

O restante do artigo está organizado da seguinte forma. A Seção 2 apresenta uma visão geral do problema tratado por este trabalho, bem como definições que serão utilizadas no decorrer do artigo. Na Seção 3 são descritos os trabalhos relacionados. O funcionamento do algoritmo HTBS e seus parâmetros são detalhados na Seção 4, enquanto a Seção 5 apresenta os resultados obtidos através de experimentos. Finalmente, a Seção 6 conclui este trabalho.

2. Visão Geral

O objetivo deste trabalho é definir um novo algoritmo de escalonamento que garanta tanto isolamento de performance como alto desempenho. Assumimos neste trabalho que um *sistema de armazenamento compartilhado* é composto por três entidades, ilustradas na Figura 1: 1) um disco, capaz de atender requisições de leitura e escrita a um ou mais blocos de armazenamento contíguos, chamados *setores*; 2) um escalonador de requisições; e 3) *n* filas de requisições, referentes às *n* aplicações competindo pelo sistema. Aplicações são quaisquer entidades que possam utilizar o disco, como processos, grupos de processos, usuários, grupos de usuários, *threads* ou máquinas virtuais. Do ponto de vista do escalonador, o disco é composto por uma grande e única sequência de setores, numerados de forma crescente.

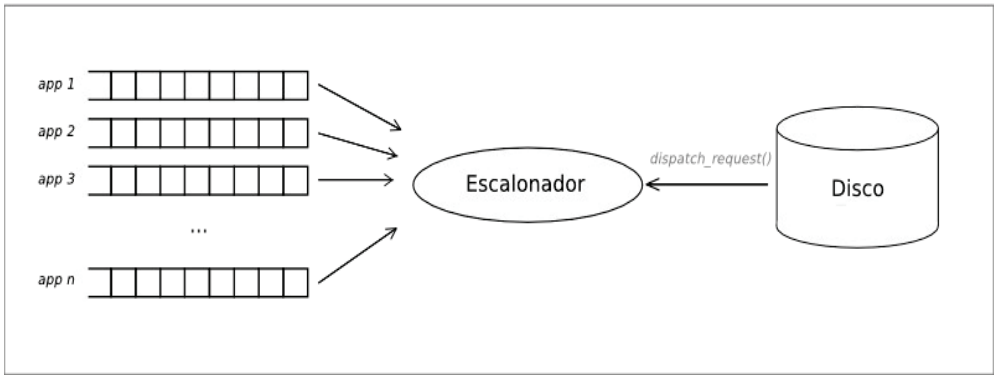

Figura 1: Modelagem do sistema.

Como definido em [Gulati et al. 2007], uma *reserva* de disco é composta por uma das *n* filas e seus três atributos de desempenho, σ_i, ρ_i e δ_i, sendo σ_i o tamanho máximo de rajadas, ρ_i a largura de banda e δ_i o limite máximo de latência da *i*-ésima reserva. O *backlog* de uma reserva *i*, representado por B_i, é o número de requisições pendentes, ou seja, requisições aguardando atendimento. Uma reserva *i* está *backlogged*

se $B_i > 0$. É denominada *ativa* a reserva que atualmente está sendo servida pelo disco. No restante deste trabalho, assumimos que os termos *reserva*, *aplicação* e *fluxo* se referem à definição de reserva discutida neste parágrafo.

Duas requisições são ditas sequenciais se a posição do final de uma requisição é adjacente ao início da próxima; caso contrário elas são denominadas não-sequenciais ou aleatórias. Caso a emissão de uma requisição dependa do término da anterior, elas são denominadas síncronas. Da mesma forma, requisições assíncronas não possuem relações de dependência. De acordo com [Valente e Checconi 2010], definimos T_{wait} como o tempo máximo que o escalonador aguardará ociosamente por nova requisição síncrona e sequencial de uma mesma reserva. Por último, B_{max} limita o número de requisições síncronas consecutivas que podem ser servidas a uma reserva.

Um escalonador que nunca desperdice capacidade, ou seja, nunca mantenha o disco ocioso enquanto houver requisições pendentes, é dito conservativo, ou *work-conserving*. Entretanto, como é mostrado em [Iyer 2003], há casos em que aguardar a requisição seguinte de uma aplicação, desde que síncrona e sequencial, pode aumentar o desempenho total do sistema de armazenamento. Neste caso, como o disco permanece ocioso por um breve intervalo de tempo aguardando novas requisições, estes escalonadores são ditos não-conservativos, ou *non-work-conserving*.

Embora paradoxal, manter o disco ocioso por curtos intervalos de tempo pode aumentar significativamente o desempenho em alguns casos. Isto ocorre porque a maioria das aplicações atuais utilizam requisições síncronas intercaladas com pequenas quantidades de processamento, como, por exemplo, alocação ou liberação de memória, processamento dos dados obtidos ou atualização de variáveis de controle [Valente e Checconi 2010]. Caso não haja uma nova requisição imediatamente após o término da anterior, um escalonador conservativo passará a atender uma nova aplicação. Essa constante troca de atendimento entre aplicações é prejudicial ao desempenho do disco, porque ocasiona *seek time* e latência de rotação em excesso, devido à falta de localidade no acesso. Esta perda de desempenho é denominada *deceptively idleness* (ociosidade enganosa) [Iyer 2003], e uma aplicação que se encontre neste tempo entre requisições é dita *deceptively idle* (enganosamente ociosa).

Escalonadores não-conservativos podem manter o disco ocioso por um curto intervalo de tempo após o término de uma requisição, com o objetivo de aguardar o processamento da aplicação e a chegada de uma nova requisição. Caso uma nova requisição seja criada neste intervalo de tempo, ela será atendida imediatamente; caso contrário, a aplicação poderá perder o direito de manter o disco em estado ocioso.

3. Trabalhos Relacionados

A abordagem atualmente mais utilizada para fornecer qualidade de serviço no acesso a disco é o desenvolvimento de novos algoritmos de escalonamento. Os trabalhos realizados nesta área podem ser classificados em dois principais grupos: 1) escalonadores de tempo real e 2) escalonadores baseados em algoritmos de enfileiramento justo (*fair queueuing*).

Os escalonadores de tempo real atribuem um *deadline* para cada requisição, reorganizando-as em função deste atributo. Em geral as requisições são atendidas de

acordo com o seu *deadline*, priorizando as que possuem o *deadline* mais próximo. A partir deste esquema, também chamado de EDF, ou *Earliest Deadline First,* foram desenvolvidas diversas variações, visando reduzir o *seek-time* e a latência de rotação. Alguns exemplos de escalonadores de disco de tempo real são PSCAN, FD-SCAN e SCAN-EDF [Reddy 2005]. Apesar de garantir *deadlines* para as requisições, estes algoritmos não têm mecanismos para limitar a largura de banda utilizada pelas aplicações ou possuem suporte a rajadas.

Algoritmos de enfileiramento justo foram utilizados inicialmente no contexto de redes de computadores, com o objetivo de alocar de forma justa a capacidade de um canal de comunicação compartilhado entre diversos fluxos de dados [Tanenbaum 2002]. Exemplos destes algoritmos são: WFQ, WF^{2}Q e WF^{2}Q+ [Bennett e Zhang 1997]. Nos escalonadores de disco baseados nesta classe de algoritmos, são atribuídas *tags* ou *timestamps* a todas as requisições, baseados em tempo real ou tempo virtual. As requisições são atendidas de acordo com seus *timestamps*, proporcionando boa distribuição da banda disponível. Alguns exemplos destes algoritmos são YFQ [Bruno et al. 1999], EYFQ, BFQ e pClock.

Este trabalho é baseado em dois algoritmos de escalonamento baseados em enfileiramento justo, BFQ e pClock, descritos em detalhes nas subseções seguintes.

3.1. Budget Fair Queueing

O BFQ [Valente e Checconi, 2010], ou *Budget Fair Queueing*, é um escalonador baseado em algoritmos de enfileiramento justo, que possibilita a alocação de uma fração da capacidade do disco a cada aplicação, proporcional ao seu *peso*. Quando enfileiradas, as aplicações recebem determinada quantidade de *budgets,* que representa a quantidade de setores de disco que elas poderão transferir. Essa disciplina de alocação de recursos também é conhecida como *Token Bucket*, ou balde de símbolos [Tanenbaum 2002].

Uma vez selecionada, uma aplicação terá acesso exclusivo ao disco, até que se esgotem seus *budgets* ou esvazie seu *backlog*. Em seguida, uma nova aplicação será escolhida de acordo com o algoritmo de enfileiramento justo chamado *Budget*-WF^{2}Q+, ou BWF^{2}Q+, uma extensão do algoritmo WF^{2}Q+. Para evitar a ocorrência de inanição, existe um limite máximo e configurável de *budgets* que uma aplicação pode armazenar, representado por $B_{max.}$

Diferentemente de outros escalonadores, o BFQ pode atender requisições de uma mesma aplicação em lotes, cujo tamanho é igual ao número de *budgets* da aplicação em determinado instante. Este esquema aumenta o desempenho do disco, devido à localidade do acesso e possibilita o agrupamento de requisições. Contudo, esse comportamento só é possível em aplicações que utilizem requisições assíncronas.

No BFQ o disco é mantido ocioso por um pequeno instante após o atendimento de requisições síncronas, evitando a ocorrência de *deceptively idleness*. Por este motivo, ele é classificado como um algoritmo não-conservativo. Essa otimização, presente também nos algoritmos Anticipatory [Iyer 2001] e CFQ [Axboe 2007], garante boa utilização da capacidade do disco, visto que a maioria dos acessos a disco é feita de forma sequencial e síncrona [Valente e Checconi, 2010].

Entretanto, assim como em outros escalonadores baseados em algoritmos de

enfileiramento justo, no BFQ não é possível configurar a largura de banda e latência de forma independente, nem o suporte a rajadas.

3.2. pClock

O pClock [Gulati et al. 2003] é um algoritmo de escalonamento baseado em curvas de chegada, ou *arrival curves*. Tais curvas são definidas através de três atributos: largura de banda, latência e rajadas, que controlam os parâmetros do *Token Bucket* de cada aplicação e as *tags* atribuídas às requisições. O atendimento das aplicações é realizado através de uma espécie de EDF, servindo requisições de acordo com seus *deadlines*, ou *finish tags*.

Por ser um escalonador conservativo, o pClock aloca a banda ociosa entre aplicações, e não as penaliza pela utilização adicional do disco. A banda ociosa também pode ser alocada a aplicações em *background*, conhecidas como aplicações *best-effort*. É provado que aplicações que cumpram suas curvas de chegada, ou seja, não excedam os limites preestabelecidos, nunca perderão seus *deadlines*, independentemente dos padrões de acesso de outras aplicações.

Para que seja possível atender os requisitos de desempenho de todas as aplicações, o pClock define matematicamente um limite mínimo de capacidade que o sistema de armazenamento deve possuir. Entretanto, é muito difícil estimar a capacidade real de um disco, pois esta depende de inúmeros parâmetros, como o padrão de acesso, localidade e até posição dos dados na superfície do disco [Bruno et al. 1999].

Como no algoritmo pClock o atendimento das requisições é baseado exclusivamente no *deadline*, o algoritmo não se beneficia da localidade das requisições, o que pode resultar na subutilização da capacidade do disco. Ainda, como o algoritmo é conservativo, pode haver *deceptive idleness* na presença de requisições sequenciais e síncronas.

4. Solução Proposta

Este trabalho propõe o HTBS, um novo algoritmo de escalonamento baseado nos algoritmos BFQ e pClock, descritos nas subseções 3.1 e 3.2, respectivamente. O objetivo é utilizar partes destes algoritmos para fornecer garantias de QoS, causando o menor impacto possível na performance do disco. Tomamos inicialmente o algoritmo pClock, por prover bom isolamento de performance (como mostrado nos experimentos presentes em [Gulati et al. 2007]), além da possibilidade de ajustar os parâmetros largura de banda, rajadas e latência de forma independente. A partir deste algoritmo, foram realizadas modificações visando a utilização do disco de forma mais eficiente.

Diferentemente do pClock, o HTBS é não-conservativo, pois implementa as políticas de prevenção de *deceptive idleness* do algoritmo BFQ. Por este motivo, algumas das garantias matemáticas expostas em [Gulati et al. 2007] referentes ao algoritmo pClock não se aplicam diretamente ao novo algoritmo. Contudo, mostramos através de experimentos que é possível manter as garantias relativas ao isolamento de performance e ao mesmo tempo aumentar o desempenho do disco.

O algoritmo é apresentado e explicado em detalhes na subseção 4.1, enquanto considerações a respeito de seu funcionamento e parametrização são discutidas em 4.2.

4.1. High-throughput Token Bucket Scheduler

O algoritmo proposto neste trabalho é baseado em *tags* ou *timestamps*. Cada requisição criada recebe duas tags: a *tag de início* (S_j^i) e a *tag de término* (F_j^i), onde i representa a reserva e j o identificador da requisição. Além disso, cada reserva i possui duas outras tags: $MinS_i$, que representa a menor tag de início entre as requisições pendentes de i e $MaxS_i$ que representa a soma da maior tag de início em i e $1/\rho_i$.

Novas requisições são criadas através da função *add_request,* enquanto a função *dispatch_request* retorna a próxima requisição a ser atendida. O algoritmo principal é mostrado em pseudo-código na Figura 2.

```
add_request (i, r)
   if active_app == i and i is waiting for the next request then
      unset_timer ()
   update_num_tokens ()
   check_and_adjust_tags ()
   compute_tags ()

dispatch_request ()
   if active_app == nil or
   active_app dispatched more than B_max then
      w = request with minimum finish tag F_j^w
      active_app = application j who issued w
   else
      if active_app is backlogged then
         w = request with minimum finish tag F_j^w from active_app
      else
         set_timer ()
         return nil
   MinS_k = S_k^w
   return w
```

Figura 2: Algoritmo principal do HTBS.

A função *dispatch_request* realiza dois procedimentos: definir a aplicação ativa (*active_app*) e a partir desta selecionar uma requisição pendente para atendimento. A aplicação ativa é alterada somente em três casos: 1) a aplicação ativa anterior atingiu o limite de requisições consecutivas B_{max}, 2) não foram criadas novas requisições no intervalo de tempo T_{wait} ou 3) o padrão de acesso da aplicação não é sequencial. Quando houver necessidade de alteração, será selecionada a aplicação que possuir a requisição pendente com menor tag de término.

Em seguida, caso a aplicação ativa esteja *backlogged*, será atendida sua requisição pendente com menor tag de término. Caso contrário, o disco será mantido ocioso por no máximo T_{wait} milissegundos, aguardando novas requisições desta aplicação, evitando a ocorrência de *deceptive idleness*. A função *set_timer* é responsável por iniciar o período de ociosidade do disco; *unset_timer,* por interrompê-lo.

Quando uma nova requisição é criada, através da função *add_request*, existem dois cenários possíveis. Se o disco está sendo mantido ocioso e a requisição pertence à

aplicação ativa, o algoritmo força a execução da função *dispatch_request*, onde a requisição recém-criada será selecionada imediatamente para atendimento. Por outro lado, se o disco não estava sendo mantido ocioso, a nova requisição é enfileirada junto a outras requisições da mesma aplicação. Em ambos os casos, três funções são executadas: *update_num_tokens, check_and_ajust_tags* e *compute_tags*. Essas três funções foram retiradas do algoritmo pClock e são ilustradas na Figura 3.

```
update_num_tokens ()
   Let Δ be the time interval since last request
   numtokens_i += Δ x ρ_i
   if numtokens_i > σ_i then
      numtokens_i = σ_i

check_and_adjust_tags ()
   Let C be the set of all backlogged reservations
   if ∀ j ∈ C, MinS_j > t_r then
      mindrift = min_{j∈C} {MinS_j − t_r}
      ∀ j ∈ C, subtract mindrift from MinS_j, MaxS_j and all start and finish
tags

compute_tags ()
   if numtokens_i < 1 then
      S_i^r = max {MaxS_i , t}
      MaxS_i = S_i^r + 1 / ρ_i
   else
      S_i^r = t
   F_i^r = S_i^r + δ_i
   numtokens_i -= 1
```

Figura 3: Funções retiradas do algoritmo pClock.

A função *update_num_tokens* atualiza o número de tokens de uma reserva. Os novos tokens disponibilizados são proporcionais ao tempo decorrido desde a última atualização, bem como a quantidade de banda ρ alocada para a reserva. Tokens regulam a quantidade de requisições que podem ser criadas por uma aplicação, sem que os atributos de desempenho sejam violados. Esta função também controla as rajadas, representadas pelo atributo por σ, por limitar a quantidade máxima de tokens armazenados em uma reserva.

Tags são atribuídas a novas requisições através da função *compute_tags*. A menos que a aplicação exceda os seus atributos de desempenho, a tag de início de uma requisição será igual ao tempo atual. Caso a aplicação ultrapasse os seus atributos de desempenho, isto é, o seu número de tokens esteja negativo, à tag de início será atribuído um valor maior que o tempo atual. Na prática, atribuir um tempo futuro à tag de início, tem como objetivo aproximar o valor que esta mesma tag teria se a aplicação criasse o mesmo número de requisições sem exceder os limites estabelecidos. A tag de término sempre corresponde à soma do valor da tag de início e do atributo de latência da reserva (δ).

Por último, a função *check_and_adjust_tags* evita que fluxos sejam penalizados pela utilização da banda ociosa do disco. Como explicado, requisições de aplicações

que excedam seus atributos recebem tags de início no futuro. Desta forma, quanto mais banda ociosa uma aplicação utilizar, mais distante do tempo atual serão suas tags de início. Seja um sistema com três reservas *a, b* e *c*, por exemplo, onde *a* e *b* estejam ociosas e *c* esteja utilizando mais do que seus atributos de desempenho. Como *c* está excedendo seus limites, suas tags de início estarão cada vez mais distantes no futuro. Quando *a* e *b* voltarem a criar requisições, suas requisições terão tags de início no tempo atual, fazendo com que as requisições da aplicação *c,* com tags de início no futuro, sofram inanição. Esta função impede que isso ocorra, evitando que as tags *MinS* de todas as aplicações *backlogged* estejam no futuro.

4.2. Parâmetros

O HTBS possui dois parâmetros ajustáveis: T_{wait}, que limita a quantidade de tempo que o disco pode ser mantido ocioso enquanto estiver aguardando novas requisições da aplicação ativa, e B_{max}, que representa o número máximo de requisições consecutivas que uma aplicação pode executar.

O valor adequado para T_{wait} depende fortemente do sistema e das características de cada aplicação. É necessário aumentar este intervalo caso as aplicações realizem uma quantidade maior de processamento entre requisições. Contudo, atribuir um valor muito alto para T_{wait} pode diminuir o desempenho do disco, pois também será alto o tempo máximo que o disco poderá ser mantido ocioso desnecessariamente, no caso de uma aplicação ter finalizado o seu acesso ao disco, por exemplo. Além disso, se o tempo que uma aplicação leva para criar sua próxima requisição é grande o suficiente, pode ser mais eficiente atender uma requisição pendente de outra aplicação, mesmo perdendo localidade. O valor ideal para T_{wait} é o menor tempo necessário para as aplicações síncronas criarem suas próximas requisições, sem causar grande impacto no desempenho do sistema. Nos experimentos realizados, o valor utilizado para T_{wait} é de 10 milissegundos.

Como exposto, B_{max} regula o número máximo de requisições síncronas e consecutivas que o disco atenderá de uma mesma aplicação. Quanto maior o valor de B_{max}, maior será o número de requisições executadas com localidade, sendo que a localidade é um dos fatores com maior influência no desempenho do disco. Entretanto, como o disco atenderá exclusivamente requisições de uma aplicação se o valor de B_{max} for alto, pode ocorrer inanição de requisições de outras aplicações. Da mesma forma, quanto maior o valor de B_{max}, mais defasadas serão as garantias de latência e maior será a flutuação (*jitter*) percebida. Na prática, quanto maior o valor de B_{max}, maior a largura de banda do disco, mas maior também é a latência média entre as requisições. O atributo deve ser configurado de forma adequada, ponderando entre alto desempenho e baixa latência.

5. Resultados Experimentais

Para a realização dos experimentos, os algoritmos HTBS e pClock foram implementados como módulos para o Kernel Linux 2.6.35 [Linux 2011]. Para o algoritmo BFQ, foi utilizada a implementação oficial que pode ser obtida em [BFQ 2011]. Todos os testes foram executados em um PC equipado com um processador AMD Athlon X2 240, 2800 MHz, dual-core, e 4 GB de memória RAM DDR3. O disco

utilizado é um Samsung HD080HJ SATA, 80 GB, 7200 rpm e 8 MB de cache *onboard*, sem suporte a NCQ (*Native Command Queueing*).

Todos os *workloads* utilizados nos experimentos são sintéticos, gerados pela ferramenta de benchmark fio [Axboe 2011]. O fio permite a realização de testes de benchmark de workloads específicos de I/O, sendo possível especificar parâmetros como a API utilizada para as requisições (síncrona ou assíncrona), padrão de acesso (sequencial ou aleatório), tipo da requisição (leitura ou escrita), tamanho das requisições, taxa de criação de requisições, dentre outros parâmetros. Esta ferramenta permite também a execução de jobs concorrentes com parametrização individual.

A seguir são apresentados os resultados de três experimentos: 1) comparação do algoritmo proposto com o algoritmo pClock na presença de duas aplicações com diferentes atributos de desempenho, 2) impacto do parâmetro B_{max} na latência das requisições de uma aplicação e 3) comparação da largura de banda acumulada utilizando diversas aplicações.

5.1. Comparação com pClock

O algoritmo proposto neste trabalho é semelhante ao algoritmo pClock. A principal diferença entre os algoritmos é que, no algoritmo proposto, foram criados mecanismos para que as garantias de QoS não fossem prejudicadas por aplicações com requisições síncronas, pois, segundo [Valente e Checconi 2010], grande parte das requisições em sistemas reais são síncronas. Neste primeiro experimento são criadas duas aplicações (jobs): *app1* com largura de banda igual a 200 KB/s e latência 50 milissegundos, e *app2*, com largura de banda de 800 KB/s, e latência 100 milissegundos. *App1* tem o padrão de acesso aleatório, enquanto *app2* é sequencial; ambas síncronas. O experimento foi executado por 5 minutos, com requisições de 4k e B_{max} igual a 20.

A Figura 4 mostra a largura de banda alcançada por cada aplicação durante o teste. De acordo com os gráficos, o algoritmo pClock (b), além de não cumprir o requisito de largura de banda de *app2*, alocou uma quantidade maior de banda para o *app1* que o esperado. Desta forma, mesmo não alocando a quantidade de banda adequada a *app2*, *app1* foi capaz de utilizar banda de disco ociosa. Isto ocorre pois, por serem síncronas, logo após o término do atendimento a uma requisição, o *backlog* de sua respectiva aplicação estará vazio. Há um intervalo de tempo entre o término de uma requisição e a criação da próxima, durante o qual o escalonador, por ser conservativo, atenderá uma requisição pendente de outra aplicação. Logo, o escalonador pClock atenderá as aplicações em política semelhante à *round-robin* neste cenário, prejudicando o desempenho total do disco e provendo serviço semelhante independente dos atributos de desempenho atribuídos.

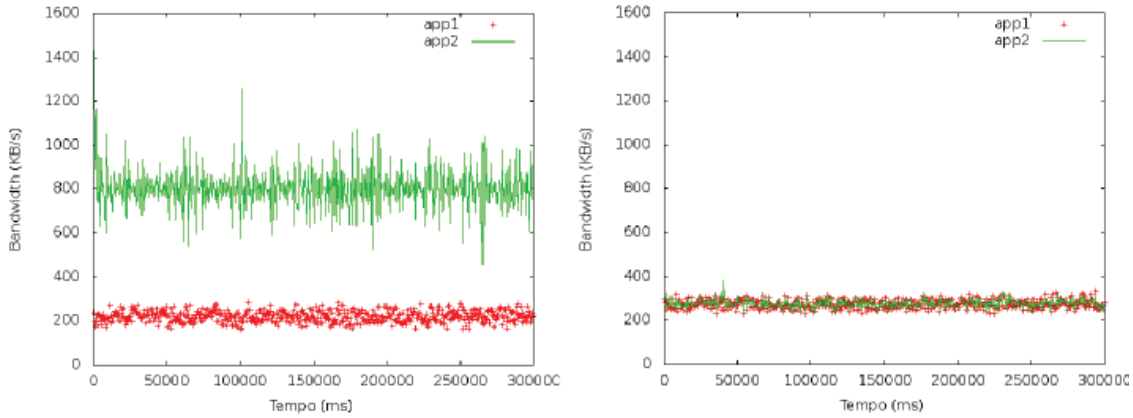

Figura 4: Largura de banda para (a) HTBS e (b) pClock

O HTBS (a) é capaz de aguardar a próxima requisição de uma aplicação síncrona, possibilitando o cumprimento do atributo largura de banda.

5.2. Impacto do Atributo B_{max} na Latência

Neste experimento é analisado o impacto do atributo B_{max} sobre as garantias de latência do sistema. O atributo B_{max} limita a quantidade máxima de requisições consecutivas atendidas de uma mesma aplicação. Por um lado, quanto maior o valor do atributo, maior o benefício pela localidade do acesso; por outro, um valor muito grande pode prejudicar as garantias de latência e até causar inanição.

Como no experimento anterior, são criadas duas aplicações com os mesmos padrões de acesso e atributos de desempenho – *app1* aleatória, largura de banda igual a 200 KB/s e latência 50 milissegundos, e *app2,* sequencial, 800 KB/s de largura de banda e 100 milissegundos de latência. Os gráficos apresentados na Figura 5 mostram a latência sofrida pelas requisições de ambas as aplicações quando executadas com (a) B_{max} igual a 1 e (b) B_{max} igual a 20. Na prática, utilizar o algoritmo HTBS com B_{max} igual a 1 é equivalente a utilizar o algoritmo pClock, logo, este experimento compara a latência sofrida por ambos os algoritmos. O tempo total de execução do teste é de 10 segundos.

A latência sofrida pelas requisições de *app1* e *app2* tem maior variação com o aumento do valor de B_{max}. Com este aumento mostrado em (b), as requisições de *app2*, que é sequencial, têm maior variação, apresentando valores muito baixos quando são atendidas as suas requisições consecutivas, e valores altos quando a aplicação ativa é alterada, causando *seek time*. *App1* também tem um pequeno aumento em sua latência média em (b), pois até 20 requisições de *app2* podem ser atendidas antes que a sua próxima requisição pendente seja servida. Mesmo assim, apesar do aumento na latência das requisições, nenhuma requisição de *app1* ou *app2* estourou o seu *deadline* neste experimento. O *jitter*, contudo, é maior em (b) do que em (a).

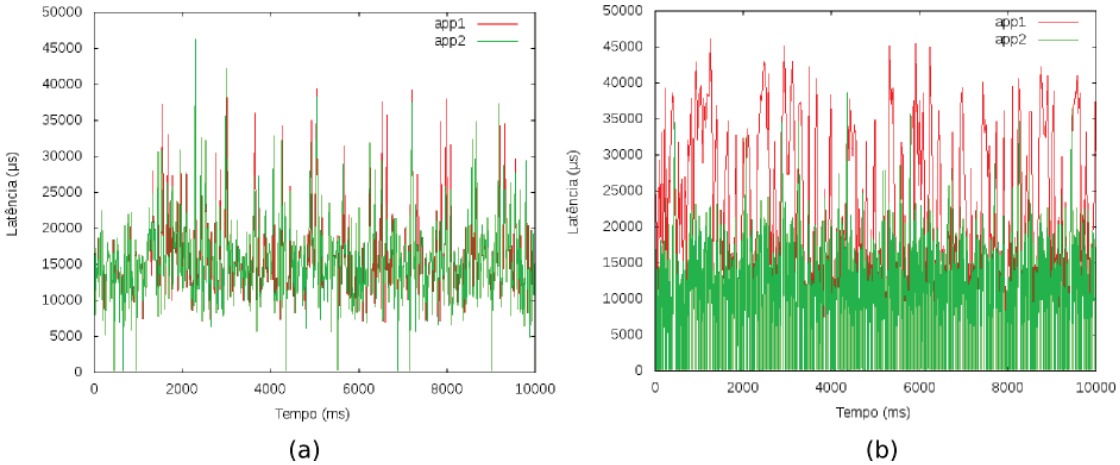

Figura 5: Latência utilizando o algoritmo HTBS com (a) B_{max} = 1 e (b) B_{max} = 20

5.3. Largura de Banda Acumulada

Neste experimento apresentamos a largura de banda total acumulada por cada escalonador. Com este objetivo, foram criados três casos de teste, sendo cada teste executado com os três algoritmos de escalonamento – pClock, BFQ e HTBS. Todos os casos de teste são compostos por sete aplicações executando leituras síncronas e sequenciais em diferentes posições do disco. A diferença entre os casos de teste consiste em que o primeiro executa somente as sete aplicações sequenciais; o segundo executa duas aplicações aleatórias concorrentemente às sete aplicações sequenciais; no terceiro, sete aplicações sequenciais e quatro aleatórias.

Cada teste foi executado por 30 segundos e o resultado mostrado corresponde à média aritmética de três execuções. Todas as aplicações aleatórias foram limitadas no *benchmark* a enviar requisições à taxa de 40 KB/s, enquanto as sequenciais foram mantidas ilimitadas. Estabelecer um baixo limite para o número de requisições aleatórias é uma forma de garantir que, em média, a mesma quantidade destas requisições sejam atendidas em todos os casos, por todos os escalonadores. Caso fossem mantidas ilimitadas, um escalonador poderia servir mais requisições aleatórias que outros, por exemplo, e prejudicar os resultados do experimento. Como o algoritmo BFQ não possui controle explícito para atributos de desempenho, a todas as aplicações foram atribuídos os mesmos parâmetros: largura de banda ilimitada (salvo requisições aleatórias) e 100 milissegundos de latência. Para os algoritmos BFQ e HTBS, o valor de B_{max} utilizado é 20.

A Figura 6 apresenta os resultados encontrados. No primeiro caso de teste, como não há requisições aleatórias, o sistema tem o melhor ganho de desempenho com os algoritmos que implementam controle de *deceptive idleness*: cerca de 25%. No segundo teste pode ser observado que mesmo na presença de requisições aleatórias, como ainda é grande o número de requisições sequenciais, HTBS e BFQ apresentam desempenho superior. No último teste, com quatro jobs aleatórios, é perceptível a degradação do desempenho do sistema. Neste caso, o HTBS mostra desempenho semelhante ao pClock, pois o ganho decorrido das execuções seguidas de requisições sequenciais é pequeno se comparado ao *overhead* causado pelas requisições aleatórias. Através destes

experimentos vemos também que a performance do HTBS é, em média, semelhante ao BFQ, mesmo que o BFQ não possua os mecanismos de controle por aplicação presentes nos outros dois algoritmos.

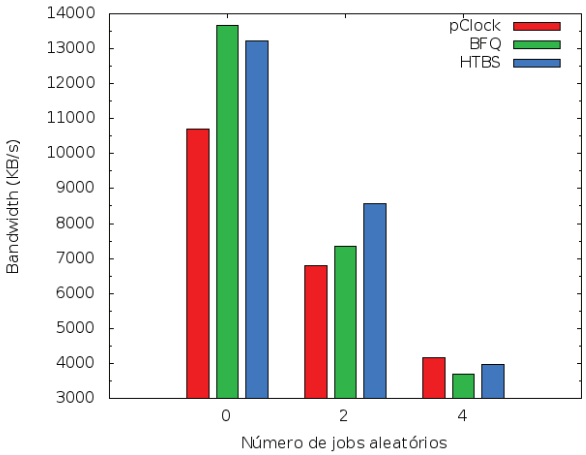

Figura 6: Largura de banda acumulada para os algoritmos pClock, BFQ e HTBS

6. Conclusão

Este trabalho apresentou o HTBS, um novo algoritmo de escalonamento de disco, baseados em dois algoritmos propostos anteriormente (pClock e BFQ), com o objetivo de fornecer isolamento de performance a aplicações concorrentes com diferentes atributos de QoS. Através de experimentos, mostramos que o algoritmo pClock, que é a base do algoritmo proposto, não fornece boas garantias de desempenho na presença de requisições síncronas e sequenciais, utilizadas por grande parte das aplicações atuais. Por isso, o HTBS incorpora alguns mecanismos utilizados pelo algoritmo BFQ para tratar este tipo de requisições, em especial o mecanismo de prevenção de *deceptive idleness*.

Os experimentos realizados com o protótipo implementado para o Kernel Linux mostram que o novo algoritmo aumenta o desempenho geral do disco, se comparado ao seu antecessor, pClock, na presença de requisições síncronas e sequenciais, sem que as garantias de latência sejam violadas. Mostramos também que em alguns casos, o desempenho geral provido pelo novo algoritmo supera o algoritmo BFQ, que apesar de prevenir a ocorrência de *deceptive idleness*, não implementa mecanismos para garantir atributos de largura de banda e latência de forma individual.

Dando continuidade ao trabalho, criaremos mais casos de teste para simular a execução de aplicações reais, como servidores Web, SGBD's e servidores de e-mail. Pretendemos também analisar o comportamento do novo algoritmo quando utilizado em conjunto com ferramentas como LVM e RAID, para que possa ser adaptado para utilização em servidores de armazenamento de grande porte.

Referências

Valente, P. e Checconi, F. (2010) “High Throughput Disk Scheduling with Fair Bandwidth Distribution”, *IEEE Transactions on Computers*, vol. 59, páginas 1172-1186. IEEE Computer Society.

Gulati, A., Merchant, A. e Varman, P. (2007) “pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems”, *SIGMETRICS Perform. Eval. Rev.*, New York, NY, USA, vol. 35, páginas 13-24. ACM.

Iyer, S. (2003) “The effect of deceptive idleness on disk schedulers.”, Dissertação de Mestrado, Rice University.

Seelam, S. e Teller, P. (2006) “Fairness and Performance Isolation: an Analysis of Disk Scheduling Algorithms”, *IEEE International Conference on Cluster Computing*, páginas 1-10. IEEE Computer Society.

Bruno, J., Brustoloni, J., Gabber, E., Ozden, B. E Silberschatz, ª (1999) “Disk Scheduling with Quality of Service Quarantees”, *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, volume 2. IEEE Computer Society.

Reddy, A., Wyllie, J. e Wijayartne, K. (2005) “Disk Scheduling in a Multimedia I/O System”, *ACM Trans. Multimedia Comput. Commun. Appl.,* páginas 37-59, volume 1. ACM.

Iyer, S. e Druschel, P. (2001) “Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O”, *ACM Symposium on Operating Systems Principles*, páginas 117-130. ACM.

Jian, K., Dong, Z., Wen-wu, N., Jun-wei, Z., Xiao-ming, H., Jian-gang, Z., Lu, X. (2009) “A Performance Isolation Algorithm for Shared Virtualization Storage System”, *Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage,* páginas 35-42, IEEE Computer Society.

Bennett, J. e Zhang, H. (1997) “Hierarchical Packet Fair Queueing Algorithms”, *IEEE/ACM Trans. Networks*, páginas 675-689, volume 5. IEEE Press.

Vaquero, L., Merino, L., Caceres, J. e Lindner, M. (2009) “A Break in the Clouds: Toward a Cloud Definition”, *SIGCOMM Comput. Commun*, páginas 50-55.

Tanenbaum, A. (2002) “Computer Networks”, volume 4. Prentice Hall Professional Technical Reference.

Axboe, J. (2007), “CFQ I/O Scheduler”, http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.pdf. Acessado em 3 de maio de 2011.

Axboe, J. (2011), “Fio – flexible I/O tester”, http://freshmeat.net/projects/fio. Acessado em 3 de maio de 2011.

Linux Kernel (2011), http://www.kernel.org. Acessado em 3 de maio de 2011.

BFQ and related stuff on disk scheduling (2011), http://algo.ing.unimo.it/people/paolo/disk_sched/sources.php. Acessado em 3 de maio de 2011.

LVM Resource Page (2011), http://sourceware.org/lvm2/. Acessado em 3 de maio de 2011.

Amazon EC2 (2011), http://aws.amazon.com/ec2/. Acessado em 3 de maio de 2011.

Google Storage for Developers (2011), http://code.google.com/intl/pt-BR/apis/storage/. Acessado em 3 de maio de 2011.