Evgueni Dodonov

Um Mecanismo Integrado de Cache *e* Prefetching *para Sistemas de Entrada e Saída de Alto Desempenho*

Orientador:
Prof. Dr. Hélio Crestana Guardia

UNIVERSIDADE FEDERAL DE SÃO CARLOS
DEPARTAMENTO DE COMPUTAÇÃO

15 de setembro de 2004

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

D646mi

Dodonov, Evgueni.
Um mecanismo integrado de cache e prefetching para sistemas de entrada e saída de alto desempenho / Evgueni Dodonov. -- São Carlos : UFSCar, 2004.
115 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2004.

1. Sistemas distribuídos. 2. Arquivos paralelos. I. Título.

CDD: 005.43 (20ª)

Dedicatória

Dedicado ao *200.18.98.120*

Agradecimentos

Antes de tudo, queria agradecer o Prof. Dr. Hélio Crestana Guardia, pela ajuda e suporte contínuo em todas as partes deste trabalho.

Queria agradecer a Carla, Edilson e Paulo, cujos projetos de mestrado foram de grande importância para a realização deste trabalho. Também queria agradecer aos companheiros e ex-companheiros dos laboratórios da pós-graduação (*AQUÁRIO*, vulgo *LAB-GSDR*, e *MUNDINHO*, vulgo *LAB-ELOTOS*) pela ajuda e convivência durante o desenvolvimento do trabalho, principalmente ao Bruno [1], Joelle [2], Alessandro [3], Érico [4] e Camillo [5] [6].

Este trabalho não seria o mesmo sem a ajuda dos meus amigos, e queria aproveitar este espaço para agradecer a eles por tudo: o pessoal do mestrado (Bruno Amaral, Bruno Zanetti, Luiz Garcia, Luis Gustavo, Alessandro Quadrado, Camilla Martins e Camillo Hobeika); da turma de *EnC2K* em geral, principalmente Joelle, Fabiana, Eduardo Toledo, Raphael Cortez, Márcio Santos, Bruno Ferguson, Bruno Gregolin, Rodrigo Murta e Daniel Kawakami; a turma *BCC98* em geral; Fernanda, Elen e Tici da turma de *BCC2000*; Eduardo e Ulisses Jensen da *ENGEC*, Cíntia do Amaral de *PMMM* e Ana Luisa de Oliveira Palma.

Queria agradecer ao *200.18.98.120* que foi o principal culpado pelo fato deste trabalho ficar do jeito que ele ficou, e ao Darli que forneceu os equipamentos e foi o principal responsável, embora involuntário, do meu aprendizado sobre a manutenção de computadores...

Além disto, queria agradecer a *CNPQ* pela bolsa de mestrado concedida e ao Departamento de Computação que viabilizou o desenvolvimento deste trabalho.

Um agradecimento especial é dedicado ao *200.18.98.96*, por permanecer o servidor insubstituível do *aquarioNET*, e ao *ReiserFS* por salvar as coisas quando tudo parecia perdido.

Um grande *thank you* vai para Patrick Volkerding, Linus Torvalds e Bram Moolenaar, por *Slackware Linux*, *Linux Kernel* e *ViM*.

E finalmente, queria a agradecer **@Dodonov* por tudo.

---

[1] *200.18.98.123*
[2] *venus.dc.ufscar.br @ Lab-Mundinho*
[3] antigo *200.18.98.113*
[4] novo *200.18.98.113*
[5] *roaming*
[6] *www.motorola.com*

Epígrafe

The good thing about standards is that there are so many to choose from...
Andrew S. Tannenbaum

Resumo

Um mecanismo integrado de *cache* e *prefetching* para um sistema de arquivos paralelos em rede tem o potencial para aumentar a velocidade e o desempenho das operações de entrada e saída de dados. Vários tipos de *cache*, com políticas e algoritmos variados, podem ser empregados. Além disto, mecanismos distintos de *prefetching* de dados podem ser utilizados. Outras técnicas, como a escrita assíncrona, a manutenção da consistência do *cache* e a organização e distribuição do espaço no *cache* também podem influenciar no desempenho destas operações. Este trabalho apresenta a implementação de uma arquitetura integrada de mecanismos de *cache* e *prefetching* para sistemas de arquivos distribuídos. Além disto, este trabalho elabora um algoritmo adaptativo de determinação do padrão de acesso para ser utilizado em mecanismos de *prefetching*, *CPS*, e duas estratégias de *prefetching*, denominadas *prefetch-on-empty* e *limited aggressive*. Como estudos de caso, o trabalho apresenta dois servidores multimídia – um servidor de arquivos multimídia *on-demand*, e um servidor de mídia contínua (*streaming*). A influência dos mecanismos propostos nos aplicativos estudados é discutida. O trabalho apresenta e discute a funcionalidade dos mecanismos integrados de *cache* e *prefetching*, avaliando os algoritmos implementados e a influência da escolha das políticas adequadas no funcionamento de aplicações do usuário.

Abstract

An integrated caching and prefetching mechanism for a parallel network file system has the potential to improve both the speed and performance of I/O operations. Different cache policies and algorithms can be used in the system. Different prefetching mechanisms can also be used in the system. The performance of the I/O operations can also be influenced by other techniques, such as the assynchronous write, cache consistency and cache space organization and distribution. This work presents the implementation of an integrated caching and prefetching architecture, intended to be used in distributed file systems. Also, this work elaborates an adaptive access pattern discovery algorithm to be used in prefetching mechanisms, denominated *CPS*, and two prefetching strategies, called *prefetch-on-empty* and *limited aggressive*. As the case studies, this work presents two multimedia servers – a on-demand multimedia server, and a streaming media server. The influence of the proposed cache and prefetching mechanisms on tested applications is discussed. This work presents and discusses the functionality of integrated caching and prefetching mechanisms, benchmarking the implemented algorithms and overviewing the influence of chosen policies on the applications behavior.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

A necessidade de armazenamento de grandes volumes de dados e de mecanismos para a transferência desses dados entre disco e memória em velocidades compatíveis com o processamento das aplicações levou à criação dos arquivos paralelos. Arquivos paralelos tratam do particionamento e da distribuição dos dados entre diversos discos, localizados em servidores de rede ou discos locais, buscando combinar as taxas de transferência no acesso em paralelo aos discos para o aumento do *throughput* e a presença de diversos níveis de buffers para a diminuição da latência no acesso aos dados sendo manipulados.

Os diversos sistemas de arquivos paralelos podem, de maneira geral, ser divididos em sistemas controlados pelo *hardware (RAID* [1] [2]*)* e pelo *software (PFS* [3]*, software RAID* [4]*, NPFS* [5]*)*. O uso dos arquivos paralelos pode ser explorado de uma maneira mais eficiente na presença de diversos discos *locais*, oferecendo um desempenho maior e latência menor das requisições. Entretanto, esta solução requer a utilização de dispositivos específicos, aumentando o custo operacional do sistema. Além disto, a utilização de um sistema local implica na centralização do sistema de arquivos paralelos. A utilização de um sistema de arquivos paralelos em rede tem o potencial da distribuição de dados pelos diversos servidores, aumentando a tolerância a eventuais falhas do sistema [6].

A criação de um sistema de arquivos paralelos distribuído permite utilizar servidores distintos para armazenar os dados, levando os benefícios de um sistema de arquivos paralelo para redes de estações existentes, sem custo adicional.

A geração de um sistema de arquivos paralelos eficiente, contudo, envolve a otimização das políticas do sistema e variações na forma de distribuição dos dados, buscando explorar o paralelismo nas operações da melhor forma para cada aplicação.

Assim, a eficiência de um sistema de arquivos paralelo depende, principalmente, da baixa latência de requisições e da alta taxa de transmissão de dados. Porém, como uma grande parte das redes existentes utiliza meios de transmissão compartilhados, os benefícios que o sistema de arquivos pode oferecer são limitados pela utilização do meio de comunicação. Além disto,

a competição pelo uso da rede é um outro fator que limita o desempenho das operações de entrada e saída. É necessário utilizar o meio de transmissão da maneira eficiente e, para isso, é possível utilizar algumas técnicas que otimizam o acesso ao meio de comunicação, tais como os mecanismos de *cache* e *prefetching*.

1.1 Arquivos Paralelos

Atualmente, existem aplicações que manipulam conjuntos de dados da ordem de centenas de gigabytes, ou até terabytes, que podem ser, por exemplo, dados científicos ou multimídia. Porém, os dispositivos de armazenamento comuns não são capazes de oferecer tempo de acesso suficientemente pequeno para manipular os dados de maneira efetiva. Isso se torna um problema grande quando dados científicos ou dados multimídia são processados em tempo real. Considerando ainda que estes dados não podem ser armazenados na memória, é necessário criar mecanismos para manipulá-los entre arquivos temporários e memória eficientemente.

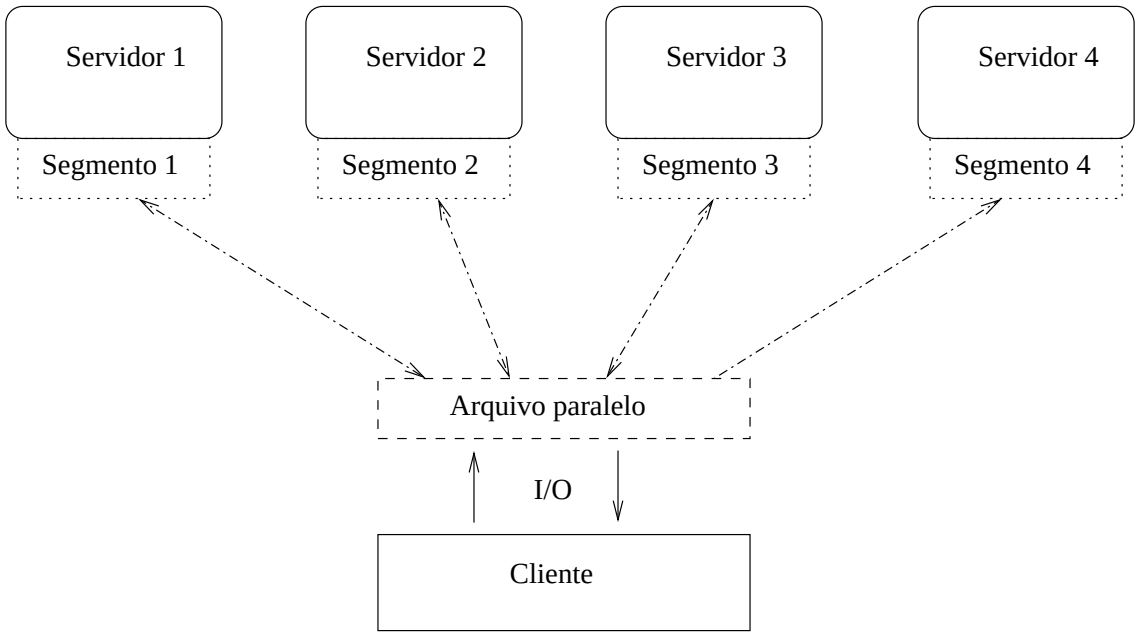

Figura 1: Arquivo paralelo

Uma solução encontrada foram os arquivos paralelos — nesses arquivos, os dados são divididos em *segmentos*, que podem ser acessados em paralelo, diminuindo assim o tempo de acesso. Da mesma forma, nas operações de escrita, os dados são transmitidos aos servidores, que os gravam em segmentos locais. A distribuição do arquivo paralelo em segmentos é feita de maneira transparente, e ocorre conforme demonstrado na figura 1 [7].

Os arquivos paralelos consistem de *segmentos* e *faixas (stripes)*, compostos por unidades de

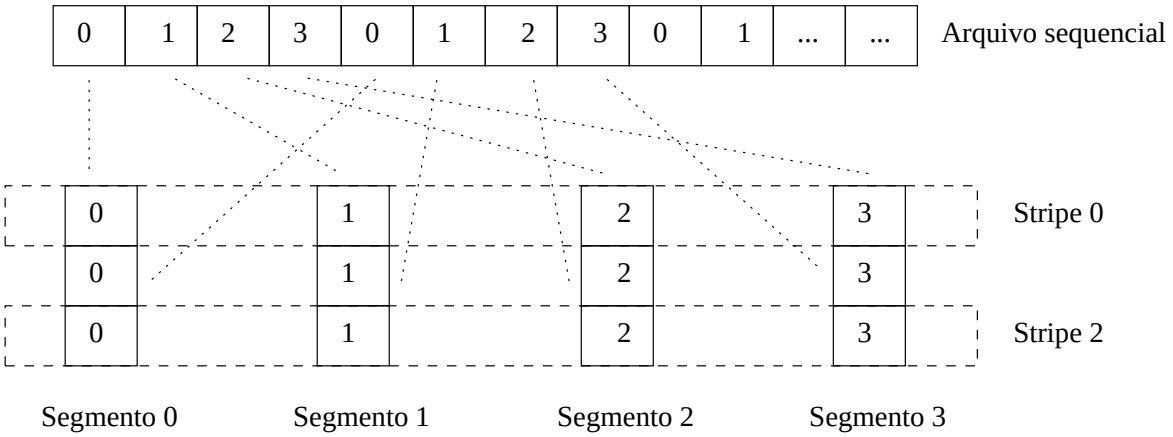

Figura 2: Estrutura do arquivo paralelo

distribuição (*striping units*), de acordo com a figura 2. O processo de manipulação dos arquivos paralelos é denominado de *Entrada/Saída paralela (Parallel I/O)*.

Considerando a organização dos dados, arquivos paralelos podem estar armazenados sob controle do *hardware* (por exemplo, utilizando *RAID*) ou sob o controle de um sistema de arquivos paralelos (*NPFS*).

1.2 Aplicações de Arquivos Paralelos

Na área da computação paralela e distribuída, as aplicações normalmente precisam obter um desempenho alto, ao mesmo tempo mantendo a latência de requisições baixa, para ter um funcionamento eficiente. Alguns exemplos de aplicações que utilizam os arquivos paralelos e distribuídos são apresentados a seguir:

- **Bancos de dados** — as possíveis vantagens oferecidas pelos sistemas de bancos de dados paralelos estão sendo estudados há várias décadas [8]. Atualmente, existem diversos exemplos de bancos de dados que utilizam mecanismos de entrada e saída paralelos, tais como *Oracle*[1], *Informix*[2] e *Paradise*[3].

  Um sistema de arquivos paralelos oferece grandes benefício para este tipo de aplicações por oferecer um maior espaço de armazenamento por um custo menor, além de otimizar a eficiência das requisições, oferecendo uma taxa de transferência maior e latência menor nas requisições.

[1] *http://www.oracle.com.*
[2] *http://www-3.ibm.com/software/data/informix/xps/.*
[3] *http://www.cs.wisc.edu/paradise/.*

- **Aplicativos científicos** — as aplicações que trabalham com conjuntos grandes de dados são bastante comuns hoje em dia [9]. Alguns dos projetos mais conhecidos que manipulam conjuntos grandes de dados são *SETI@HOME* [4] e o projeto *GENOMA* [10].

  O projeto *SETI@HOME* [11] utiliza o rede Internet como um super-computador, destinado a processar os dados obtidos através dos telescópios localizados em Porto Rico. Como a quantidade de dados obtidos é muito superior ao poder computacional disponível para processá-los localmente, os dados são disponibilizados para diversos computadores pessoais conectados a Internet, formando uma espécie de um *GRID* computacional [12]. Os dados são distribuídos entre os participantes do *grid* para o processamento, e os resultados são agrupados posteriormente.

  O projeto *GENOMA* foi criado para seqüenciar o *genoma* humano. Para isso, é necessário processar um volume grande de dados [10] de maneira eficiente, e a solução encontrada foi a utilização de bancos de dados distribuídos. A utilização de arquivos paralelos é uma das soluções utilizada no projeto, por oferecer taxas de transferência e latência de requisições eficientes.

- **Servidores de aplicações** — atualmente, os servidores de aplicações, como servidores *WWW* (*apache* [5], *Microsoft IIS* [6]), servidores de arquivos (*SMB* [7], *Microsoft Windows NT/2000 Server* [8]), entre outros, possuem suporte a operações paralelas.

  Tanto a alta taxa de transferência quanto latência baixa de requisições são os fatores favoráveis para este tipo de aplicações.

- **Aplicações especializadas** — Ron Oldfield e David Kotz demonstram em [9] que existe uma área de pesquisa ativa envolvendo os arquivos paralelos, voltando tanto para aplicações especializadas (como processamento de DNA [13], processamento de dados obtidos por satélites [14], ente outros), aplicações e estudo detalhado de diferentes padrões de acesso de entrada e saída paralela (como modelagem do relevo de mar [15], aplicações sísmicas [16], entre outras), aplicações que discutem tópicos relacionados à entrada e saída paralela (como técnicas de implementação de aplicações out-of-core [17], discussão de APIs e interfaces [18] [19]), além de diversas outras pesquisas.

A utilização de um sistema de arquivos paralelos pode melhorar o desempenho destes aplicativos, principalmente daqueles que precisam processar um volume significativo de dados.

[4] http://setiathome.ssl.berkeley.edu/
[5] *http://www.apache.org*.
[6] *http://www.microsoft.com*.
[7] *http://www.samba.org*.
[8] *http://www.microsoft.com*.

Para isso, é necessário algum mecanismo que possibilite o acesso aos arquivos paralelos.

1.3 Métodos de acesso aos Arquivos Paralelos

Entre as maneiras de acesso a arquivos paralelos, podemos destacar duas abordagens:

- **Acesso transparente** — desta maneira, a aplicação acessa os arquivos paralelos como se eles fossem arquivos comuns. Normalmente, isto é feito com a ajuda de algum dispositivo que possibilita acesso direto pelo sistema operacional aos arquivos paralelos. A maioria dos sistemas operacionais possui algum *driver* (no caso do sistema operacional *Windows*) ou *módulo do kernel* (no caso dos sistemas UNIX; por exemplo, *Linux* e *FreeBSD*) para acessar arquivos paralelos da maneira transparente. A desvantagem desta abordagem é a impossibilidade de explorar o paralelismo dos dados da maneira mais específica e adequada para cada aplicação, por oferecer a mesma interface de acesso aos dados para todos os tipos de sistemas de arquivos, paralelos ou não.

  A utilização de *módulos* nos sistemas *Unix*, da mesma forma que a utilização de diversos *drivers* nos sistemas *windows*, torna possível a integração de diversos sistemas de arquivos paralelos ao sistema sem a necessidade de intervenção no funcionamento do mesmo.

- **Interfaces de entrada e saída** — nestes casos, cada sistema de arquivos paralelos possui uma interface própria para acessar os dados. Assim, é possível utilizar da melhor maneira possível os recursos que o sistema de arquivos oferece. Isto é feito com uma maior troca de informações entre as aplicações e o sistema de arquivos.

  Entre as interfaces de entrada e saída paralela podemos destacar *MPI-IO* [20] [21], *PIOUS* [22] e *NPFS* [23].

  Existem projetos que visam implementar as funções destes sistemas de entrada e saída dentro do kernel do Linux [24] [25], possibilitado acesso *transparente* a estes sistemas de arquivos.

1.4 Acesso aos arquivos paralelos

No acesso a um sistema de arquivos, os dados são acessados freqüentemente. As situações nas quais diversos processos podem requisitar os mesmos dados ao mesmo tempo ou em seguida são bastante comuns durante os processos de entrada e saída de dados. Quando isto acontece sem uma política de *cache*, o sistema de arquivos deverá efetuar várias consultas ao

disco, podendo envolver a comunicação com a rede para ler os mesmos dados várias vezes, sendo que isso é desnecessário se os dados forem mantidos na memória principal, numa região de acesso rápido. A competição para o uso de um meio de comunicação normalmente compartilhado, como, por exemplo, uma rede *ethernet* para comunicação e acesso aos dados em redes de estações também sugere a redução do número de transmissões.

Na escrita de dados acontece o mesmo processo. Os dados podem ser escritos, lidos e depois escritos novamente. Neste caso, se nenhum mecanismo de *cache* for utilizado, os dados serão escritos no disco para serem lidos posteriormente. Utilizando uma política de *cache*, é possível reduzir o número de consultas ao disco, ou à rede, significativamente: os dados podem permanecer em memória para futuros acessos, tirando, assim, a necessidade de acessar novamente o disco. Como a memória é muito mais rápida que o disco rígido, um mecanismo de *cache* possibilita um aumento significativo do desempenho das operações de entrada e saída.

Uma outra maneira de aumentar o desempenho de um sistema de arquivos é através de mecanismos de busca antecipada de dados (*prefetching*). Com essa técnica, conhecendo-se *a priori* as necessidades futuras das aplicações, é possível antecipar as requisições de leitura dos dados. Isso pode diminuir o tempo de espera entre a leitura e o processamento de dados.

Assim, um mecanismo integrado de *cache* e *prefetching* possibilita processamento mais eficiente dos dados [3].

1.5 Desempenho das operações

O crescimento contínuo da velocidade dos processadores a das taxas de acesso à memória não é acompanhado pelas taxas de transmissão oferecidas pelos meios de comunicação convencionais, tais como redes de computadores. Enquanto o limite atual de transmissão de dados por uma rede de computadores chegou até as velocidades em torno de *10Gbps* [26], possibilitando a transferência de até 1.25GB de dados por segundo, os processadores mais recentes já operam nas freqüências superiores e *3GHz* [9] [10], possibilitando a manipulação de até 96GB de dados por segundo (considerando uma arquitetura de 32bits). Este crescimento da velocidade dos processadores foi acompanhado de perto pelo crescimento das interfaces de memória [11] (RDRAM [12], DDR [13] e DDR2 [14]), que operam nas taxas de transferência de até *9.6* (no caso de *DDR2*) e *10.7*

[9] http://www.intel.com
[10] http://www.amd.com
[11] http://www.memoryx.net/generic-memory.html
[12] http://www.rambus.com/products/rdram/
[13] http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_893,00.html
[14] http://www.infi neon.com/cgi/ecrm.dll/ecrm/scripts/promotion_start_page.jsp?oid=34746

(no caso de *RDRAM*) *GBytes por segundo* [27].

Desta forma, enquanto o meio de comunicação evoluiu de *10Mbits* (redes *ethernet*) para velocidades de em torno de *100Mbits* (redes *fast ethernet* e *ATM*) e até para velocidades maiores, tais como *1Gbit* (*gigabit ethernet*) e *10Gbits* ultimamente, a taxa de transferência de dados entre os discos rígidos, memória principal e *CPU* evoluiu de uma maneira que todo o crescimento das velocidades dos meios de comunicação foi insuficiente para acompanhar os outros pontos do sistema, tornando o meio de comunicação um verdadeiro "gargalo" nas operações de entrada e saída de dados.

Como foi visto anteriormente, os principais fatores que influenciam no desempenho das operações de entrada e saída de dados são a *latência* das requisições, isto é, a demora entre a requisição de dados e o momento no qual os dados tornam-se disponíveis para o processamento; e a *taxa de transferência* de dados, isto é, a quantidade de dados que podem ser transferidos em um período de tempo. A utilização de um meio de comunicação mais lento continua sendo o ponto crucial para o acesso de dados.

Desta forma, a necessidade de algum mecanismo para a redução e otimização dos acessos ao meio de comunicação torna-se evidente. A utilização de mecanismos de *cache* e *prefetching*, apresentados no próximo capítulo, permite resolver o problema apresentado, otimizando a utilização do meio de comunicação pelas aplicações do usuário e reduzindo o número de acesso à rede.

Este trabalho teve como objetivo o projeto e a implementação de mecanismos de *cache* e *prefetching* para serem usados em sistemas de arquivos paralelos. Neste texto, os mecanismos de *cache* e *prefetching* são apresentados e discutidos na seção 2. A seção 3 introduz o sistema de arquivos *NPFS*, utilizado neste trabalho. Uma revisão de trabalhos relacionados é apresentada na seção 4. Na seção 5 é apresentado o trabalho desenvolvido, e a seção 6 apresenta os testes e estudos de caso realizados. Finalmente, a seção 7 conclui o trabalho e apresenta os planos para os trabalhos futuros.

2 Cache e Prefetching

Dispositivos de armazenamento de dados utilizados por sistemas de arquivos geralmente apresentam tempo de acesso consideravelmente inferior ao da memória principal [28]. Considerando que nas aplicações com predominância das operações de entrada e saída (*IO-BOUND*) a velocidade de acesso ao dispositivo de armazenamento é o parâmetro mais importante, podemos ver que o "gargalo" destas aplicações é definido pela velocidade destes dispositivos [29]. Considerando ainda que o número de acessos aos dispositivos de armazenamento é relativamente alto nos sistemas de arquivos paralelos, devido à necessidade de troca intensiva de mensagens entre clientes e servidores, é possível ver que o desempenho das operações de entrada e saída é prejudicado pela utilização de dispositivos de armazenamento lentos.

Uma solução mais óbvia seria a utilização da memória principal para o armazenamento de todos os dados necessários para uma aplicação. Por outro lado, embora apresentasse tempo de acesso significativamente menor que os discos, a memória principal é utilizada em quantidades pequenas devido ao seu custo num sistema. A otimização do seu uso é, portanto, essencial para o desempenho de um sistema de arquivos paralelos.

Neste contexto, podemos definir *cache* como uma técnica que possibilita armazenar os dados mais relevantes na memória, reduzindo a necessidade de acessar os dispositivos de armazenamento e aumentando o desempenho das operações de leitura e escrita de dados.

Apesar de oferecer diversas vantagens, um sistema de *cache* traz problemas que devem ser resolvidos para a implementação eficiente do mesmo:

- Como decidir quais dados devem ser colocado no *cache*? Para isto, é necessário determinar a relevância destes dados para a aplicação em questão.
- Quando dados mantidos em memória devem ser salvos no disco? Ou seja, quais dados permanecem relevantes para a aplicação depois de serem armazenados em *cache*, e como determinar a relevância deles?
- Como escrever os dados armazenados em *cache* de volta aos discos? É necessário deter-

minar uma política para a escrita dos dados de acordo com estado do *cache* e a relevância dos dados em questão.

- Como utilizar o *cache* de maneira eficiente, para que somente os dados requisitados estejam disponíveis? Para isso, é necessário determinar o organização do espaço de *cache* para sempre ter espaço para os dados necessários.
- Como manter a consistência dos dados no *cache*? Isto é, qual é o procedimento para atualizar os dados mantidos em *cache* quando os dados armazenados são modificados?

Estas são algumas das perguntas que devem ser respondidas antes da implementação de um sistema de *cache*. Neste capítulo, as possíveis respostas para os problemas apresentados serão apresentadas e a implementação dos mesmos será discutida.

Por outro lado, um sistema de *cache* pode, além de servir para o armazenamento de dados lidos anteriormente, prover um espaço para o armazenamento de dados que podem ser requisitados no futuro, mas não foram solicitados ainda. A técnica de leitura de dados a serem requisitados no futuro é denominada de *prefetching*.

Prefetching, também como conhecido como *leitura antecipada* ou *redução de tempo ocioso do sistema* [28], é um mecanismo que procura antecipar leitura de dados, para carregá-los dados mais rapidamente na memória principal ou no *cache*, visando oferecer um acesso mais rápido aos mesmos quando solicitados. Com a utilização de um mecanismo de *prefetching* é possível reduzir o tempo decorrido entre uma requisição de leitura de dados e o acesso aos dados propriamente ditos, caso os dados solicitados já estejam em memória [30].

O mecanismo de *prefetching* também possui questões que devem ser discutidas para sua implementação:

- Como descobrir quais dados serão requisitados por um processo? Isto é, como determinar a seqüência de blocos lidos (também conhecida como *padrão de acesso*) [31].
- Como efetuar a consulta para ler os dados? Os dados podem ser lidos de maneira síncrona (isto é, de uma maneira controlada pela aplicação), ou de maneira assíncrona, através de um mecanismo que utiliza todo o tempo ocioso para requisitar os dados antecipadamente.
- Os dados devem ser lidos pelo processo do usuário diretamente ou devem ser mantidos em um *cache* na memória principal? Ou seja, onde os dados lidos antecipadamente devem ser armazenados?

- Como calcular a eficiência de um mecanismo de *prefetching*? É necessário avaliar diversas características do mecanismo de *prefetching* para determinar a sua eficiência [32].

- Quais são as técnicas que ajudam a descobrir as seqüências de leituras de dados (isto é, os padrões de acesso aos dados)? Como determinar os padrões de acesso e adaptar-se às possíveis mudanças da seqüência de leitura de dados [33]?

- Como determinar quando utilizar cada algoritmo de *prefetching*? A eficiência dos algoritmos de *prefetching* pode variar de acordo com a aplicação em questão [34] [30].

Dada a limitação do espaço em memória para conter os dados buscados e a possibilidade de interferências entre o mecanismo de *prefetching* e a leitura normal de dados, é necessário escolher uma política adequada para o *prefetching*, uma vez que um algoritmo inadequado pode prejudicar o desempenho do aplicativo, efetuando requisições antecipadas de dados que não serão utilizados no futuro e descartando dados úteis. Um algoritmo de *cache* e *prefetching* tem que seguir algumas regras básicas [30], apresentadas na seção 2.4, para ser eficiente, caso contrário, poderá piorar o desempenho das operações de entrada e saída.

2.1 Organização do *cache*

Visando uma estrutura geral de sistemas de arquivos paralelos, podemos dividir as entidades presentes em um sistema em:

- *Servidores* – dispositivos, ou servidores de rede, que armazenam os dados propriamente ditos.

- *Clientes* – processos que requisitam os dados.

Desta maneira, é possível representar um mecanismo de *cache* como um cooperação entre *servidores* que armazenam os dados em *cache* e os *clientes* que requisitam dados propriamente ditos. Esta terminologia será utilizada nesta seção para ilustrar o processo de interação com o mecanismo de *cache*.

Em sistemas de arquivos paralelos, os *servidores* geralmente podem comunicar-se com vários *clientes* que, por sua vez, podem acessar vários *arquivos* simultaneamente. Assim, para projetar um espaço de *cache* é necessário levar em consideração a distribuição da memória para todas as entidades. Para isto, é relevante determinar como o espaço de *cache* deve ser dividido para armazenar os dados.

Desta maneira, é possível pensar em um mecanismo de *cache* tanto no *cliente*, quanto no *servidor*. Assim, o *cache* do cliente torna-se conhecido com o *client-side cache* e o do servidor – *server-side*.

Para simplificar a terminologia utilizada, o termo *cache no servidor* será utilizado para determinar a entidade que efetua as consultas ao espaço de *cache*, e o termo *cliente* – entidade que requisita os dados dele.

É possível pensar em várias maneiras de distribuir o espaço do *cache* dos *servidores* entre os *clientes*: criar um *cache* separado para cada um dos clientes, dividir um único *cache* em blocos de tamanho fixo para cada um dos clientes, dividir o *cache* de acordo com o número de clientes ou alocar o espaço do *cache* dinamicamente.

Uma das maneiras de distribuir o espaço do *cache* é criar um *cache* separado para cada cliente. Assim, os *caches* serão independentes entre si, tornando a manutenção e a organização dos dados mais fácil. Por outro lado, a utilização de um espaço de *cache* independente para cada cliente pode prejudicar a eficiência dos mecanismos de *cache*, uma vez que os clientes podem utilizar o *cache* de maneiras diferentes.

Uma outra maneira seria utilizar um *cache* único para todos os clientes, dividido estaticamente entre os mesmos. Assim, cada cliente terá um número pré-determinado de blocos, independente dos outros clientes. Este método é semelhante ao método anterior, porém ele não é tão flexível devido à necessidade de re-organização completa de espaço de *cache* de acordo com o número de clientes presentes no sistema.

Um outro método de distribuição de *cache* consiste em alocar seu espaço dinamicamente, de acordo com as necessidades de cada um dos clientes. Desta maneira, cada cliente pode requisitar um espaço de memória que for necessário para ele. O maior problema deste método é a complexidade de gerenciamento de memória compartilhada entre os clientes, uma vez que cada cliente utiliza a memória alocada do jeito que ele julga melhor. Uma vantagem deste método é uma independência maior entre os *caches* dos clientes.

A identificação dos dados presentes no *cache* é um aspecto fundamental para a funcionalidade correta dos mecanismos de *cache*. Desta maneira, os dados mantidos no *cache* devem ser organizados para facilitar o acesso a eles: por exemplo, se o *cache* for muito grande, pode ser necessário percorrer a área de *cache* inteira para achar um determinado bloco. Isto poderá influenciar no desempenho dos acessos e, conseqüentemente, reduzir a eficiência do uso da memória para o armazenamento de dados.

Geralmente, são utilizados os métodos de organização simples [35], como *listas ligadas*, *listas duplamente ligadas*, *árvores binárias* ou *tabelas hash*, uma vez que estes métodos são de fácil implementação e apresentam uma eficiência alta. Além disto, alguns dos sistemas de arquivos atuais utilizam algumas técnicas mistas para obter resultados melhores [36].

Para facilitar o acesso aos dados mantidos no *cache*, é possível manter dois conjuntos de dados: um para especificar quais são os blocos que estão sendo utilizados atualmente e um outro para listar os blocos vazios, que podem ser utilizados para armazenar novos dados. Diversas estruturas de dados podem ser utilizadas para estas tarefas – a lista de blocos vazios, por exemplo, pode ser representada por uma lista ligada, uma vez que não há necessidade de percorrê-la inteira para achar um bloco vazio. Os dados armazenados em *cache*, por sua vez, precisam de alguma estrutura de dados mais eficiente, sendo que eles são acessados com maior freqüência.

2.2 Consistência do *cache*

Para manter os dados mais relevantes em *cache*, é necessário determinar as políticas necessárias para manter o *cache* consistente continuamente, independentemente do seu conteúdo. Para isso, é necessário determinar o comportamento dos mecanismos de *cache* frente à possível falta de espaço, e as ações a serem tomadas para escolher os blocos a serem descartados e a maneira de fazer isso.

2.2.1 Atualização de Cache

Como o *cache* tem tamanho finito, é necessário atualizá-lo com os dados mais adequados, ou seja, com dados que possuem uma maior probabilidade de utilização. Quando existe espaço livre em *cache*, a tarefa é trivial: é necessário apenas encontrar um espaço vazio e preenchê-lo. Entretanto, quando o *cache* está completamente ocupado, a situação fica mais complicada, sendo necessário retirar algum bloco do *cache* para liberar espaço. A identificação de quais dados podem ser retirados do *cache* requer uma política eficiente para manter o *cache* sempre atualizado e com os dados mais relevantes.

Existem várias políticas de gerenciamento de *cache*, tais como *FIFO*, *LRU*, *LFU* [37], *Segmented FIFO* [38], *CLOCK* [39], *2Q* [40], *LRU-K* [41], *EELRU* [42], *SEQ* [43], *LIRS* [44], *ARC* [45], *MRU* e *SIZE-MRU* [46], entre outros, que são descritas a seguir:

- ***FIFO (First In First Out)*** – possivelmente, é o algoritmo mais trivial de gerenciamento de *cache*. O algoritmo retira os blocos do *cache* na mesma seqüência que os mesmos

foram inseridos no mesmo. Desta maneira, o algoritmo oferece o melhor desempenho, já que ele não precisa analisar os dados antes de descartar os blocos correspondentes. Por mesmo motivo, este algoritmo pode descartar os blocos relevantes [37].

- ***LRU (Least Recently Used)*** – este algoritmo visa determinar a relevância dos blocos baseando-se em tempo de utilização dos mesmos. Desta maneira, o bloco que foi utilizado recentemente tem maior relevância do que um bloco que não foi utilizado por algum tempo [37].

- ***LFU (Least Frequently Used)*** – o algoritmo determina a relevância dos blocos baseando-se na freqüência do acesso a eles. Assim, quanto mais vezes um bloco foi acessado maior a relevância dele [37].

- ***Segmented FIFO*** – este algoritmo é uma variação do algoritmo *FIFO* original, que visa diminuir a possibilidade de remover um bloco relevante do *cache*, ao mesmo tempo mantendo o desempenho do algoritmo *FIFO* original [38].

  O algoritmo utiliza duas filas de blocos. A primeira fila de blocos comporta-se da mesma maneira que o algoritmo *FIFO* original. Quando um bloco é removido da fila original, ele é adicionado na segunda fila. Caso ele é acessado novamente enquanto ele encontra-se nesta fila, ele é retornado para a fila inicial.

  Desta maneira, é possível diminuir o número de blocos retirados do *cache* que podem ser utilizados no futuro.

  Variando o tamanho das filas, é possível alterar o comportamento do algoritmo. Por exemplo, se o tamanho da segunda fila for menor do que o da primeira, o algoritmo comporta-se como um algoritmo *FIFO* clássico. Entretanto, se as duas filas tiverem o mesmo tamanho, o comportamento do algoritmo fica muito similar ao do *LRU*. Quando o tamanho da segunda fila ultrapassa o tamanho da primeira fila, o algoritmo comporta-se da maneira idêntica ao algoritmo *LRU* [47].

- ***CLOCK*** – este algoritmo utiliza uma lista circular para determinar os blocos a serem removidos do *cache* [39].

  O algoritmo representa a lista de blocos como sendo uma lista circular, sendo que a posição atual desta lista é indicada pelo *ponteiro do relógio*, dando nome para o algoritmo. Cada bloco utilizado é marcado por um *bit de utilização*. De maneira similar a um relógio convencional, o algoritmo cicla pelos blocos nas direções horária ou anti-horária, procurando por algum bloco disponível e removendo o *bit de utilização* dos blocos consultados.

De acordo com o passo do algoritmo, o comportamento do mesmo pode ser similar ao *FIFO* ou *LRU* [47].

- ***LRU-K*** – o algoritmo é uma variação do algoritmo *LRU* clássico que suporta um histórico de acessos para determinar a relevância dos blocos [41].

  O algoritmo *LRU* clássico, na verdade, é um caso específico do algoritmo *LRU-K*, com $K = 1$. O "*K*" específica quantas referências passadas para o mesmo bloco devem ser armazenadas na memória. Desta maneira, como o algoritmo *LRU* clássico armazena apenas o número de referências atual, o valor do *K* no caso dele é igual a 1, ou seja, algoritmo *LRU* clássico é um algoritmo *LRU-1*.

  A vantagem deste algoritmo é a possibilidade de determinar um comportamento mais detalhado de cada um dos blocos presentes em *cache*, levando em conta tanto a relevância atual quanto o número de acessos passados aos mesmos.

  A desvantagem, entretanto, é a complexidade e o custo computacional alto deste algoritmo.

- ***2Q (Two Queues)*** – este algoritmo é um dos algoritmos que utiliza o histórico de acessos aos blocos para determinar a relevância dos mesmos. O algoritmo apresenta uma variação do algoritmo *LRU-2* (algoritmo *LRU-K*, com $K = 2$), com duas listas de blocos [40].

  O algoritmo possui duas listas de blocos, uma com blocos "quentes" e outra com blocos "frios". Uma das listas armazena os blocos que são utilizados com intervalos maiores (blocos "frios"), e outra – blocos que são utilizados por intervalos pequenos de tempo (blocos "quentes").

  Quando os blocos "frios" passam a ser utilizados freqüentemente, eles são transferidos para a lista de blocos quentes. Da mesma maneira, os blocos quentes, a não serem utilizados mais, são transferidos para a lista de blocos "frios". Os blocos "frios", a não serem mais utilizados, são removidos do *cache*.

  Para obter melhor desempenho, a lista de blocos "frios" é gerenciada pelo algoritmo *FIFO*, e dos "quentes" quentes pelo algoritmo *LRU*.

  Desta maneira, o *cache* apresenta funcionalidade adequada tanto para blocos que são utilizados por períodos curtos de tempo, quanto para blocos que são utilizados por períodos prolongados.

- ***EELRU (Early Eviction LRU)*** – o algoritmo *EELRU* é uma variação do algoritmo *LRU* que visa tornar o algoritmo *LRU* mais eficiente nos casos de seqüências de dados que não cabem inteiramente em um *cache* [42].

O algoritmo funciona da maneira similar ao algoritmo *LRU* clássico até detectar que um número grande de blocos foi retirado do *cache*. Neste caso, o algoritmo aplica um algoritmo de *fallback* para determinar os blocos que serão utilizados mais vezes e remover os blocos de menor relevância, mesmo que os mesmos forem recém-inseridos em *cache*. Para isso, o algoritmo armazena informações tanto sobre os blocos presentes em *cache* atualmente quanto sobre os blocos que já foram removidos do *cache*.

- ***SEQ*** – este algoritmo é diferente dos demais algoritmos devido ao fato dele processar *seqüências de blocos* ao invés de blocos individuais [43].

  O comportamento do algoritmo é similar ao algoritmo *LRU* nos sistemas que não manipulam seqüências de blocos em *cache*. Entretanto, quando uma seqüência de blocos é retirada de *cache*, o algoritmo começa a tratar esta seqüência como um todo. A seqüência detectada é tratada de maneira similar ao algoritmo *MRU*. Desta maneira, o algoritmo possibilita antecipar as requisições futuras aos blocos de uma seqüência.

  O algoritmo apresenta uma eficiência superior à do algoritmo *LRU* clássico, entretanto, o desempenho é inferior devido a um número maior de dados analisados [43].

- ***LIRS (Low Inter-reference Recency Set*** – este algoritmo utiliza os intervalos de acessos entre diferentes blocos em seqüência para determinar a relevância dos blocos [44].

  O algoritmo funciona da maneira contrária ao algoritmo *LRU*, que utiliza o tempo de último acesso a um bloco para determinar a relevância do mesmo – o algoritmo *LIRS* leva em conta os intervalos de acesso aos blocos para determinar a relevância dos mesmos.

  De acordo com os testes efetuados, o algoritmo oferece funcionalidade e desempenho superiores a outros algoritmos de substituição de blocos [44].

- ***ARC*** – este algoritmo visa juntar as vantagens dos algoritmos de substituição de blocos *LRU-2*, *2Q* e *LIRS* e automatizar o funcionamento deles, possibilitando a adaptação do algoritmo de substituição de blocos ao comportamento da aplicação [45].

  Além disto, o algoritmo *ARC* visa oferecer uma complexidade constante, independentemente do número de requisições, enquanto os algoritmos baseados em *LRU-K* possuem crescimento *logarítmico* de complexidade.

  Visando manter o *cache* preenchido com os dados mais relevantes, o algoritmo tem tolerância para os mecanismos de *scanning* – isto é, os dados que são lidos apenas uma vez não entram em *cache* para não retirar os dados a serem requisitados posteriormente.

- ***MRU (Most Recently Used)*** – este algoritmo visa determinar quais blocos tem foram utilizados maior número de vezes, determinando a relevância dos mesmos de acordo com o número de utilizações dos mesmos [46].

- ***SIZE-MRU*** – este algoritmo é uma variação do algoritmo *MRU*, que opera sobre blocos de tamanhos diferentes. Os tamanhos de blocos são utilizados para determinar a relevância dos blocos, baseando-se no custo necessário para remover ou inserir um bloco em *cache*.

A política de atualização do *cache* depende dos tipos de dados processados, ou seja, um algoritmo pode ser o mais efetivo em alguns casos e ter um péssimo desempenho em outros [46] [48].

Dependendo do algoritmo utilizado, a implementação das estruturas de dados pode variar de uma lista simples de blocos (como, por exemplo, no caso do algoritmo *FIFO*), duas listas (uma contendo os blocos de dados, e uma outra determinando a relevância dos mesmos), listas ordenadas por ordem de relevância de dados para facilitar a manutenção dos mesmos ou outras estruturas [35].

2.2.2 Escrita de dados

Eventualmente, os dados do *cache* devem ser escritos de volta no disco. Os principais métodos utilizados para isso são denominados *writethrough*, *writeback*, *writefree* e *writefull* [31].

A eficiência de cada política depende da aplicação em questão e da maneira da utilização do *cache*.

O funcionamento de cada política é descrito a seguir:

- **Método *writethrough*** — Este método escreve os dados do *cache* de volta no disco toda vez que os dados forem atualizados. Isso possibilita a consistência dos dados, mantendo as informações do disco sempre atualizadas. Porém, este método é relativamente lento, uma vez que toda atualização do *cache* deve ser escrita no disco, não oferecendo suporte às operações assíncronas de escrita de dados.

- **Método *writeback*** — Este método mantém os dados no *cache* até que este seja esvaziado. Os dados são atualizados apenas no *cache*. Assim que um bloco de dados for escolhido para sair do *cache*, o seu conteúdo é escrito no disco.

Este método é significativamente mais rápido do ponto de vista da aplicação, uma vez que os dados são considerados escritos logo depois de serem colocados em cache; porém, como os dados são mantidos apenas na memória, os dados podem se tornar inconsistentes caso ocorra uma falha.

- **Método *writefree*** — Este método escreve o bloco de volta no disco apenas quando algum processo requer um bloco de *cache* disponível. Este método é um balanceamento entre *writethrough* e *writeback*.

- **Método *writefull*** — De acordo com este método, todos os blocos marcados para a escrita são escritos no disco quando o buffer está "cheio", ou seja, não há espaço disponível. Desta maneira, todo o *cache* é escrito de uma vez para o disco.

2.2.3 Consistência de dados

Em sistemas de arquivos paralelos a leitura de um mesmo arquivo por diversos processos é bastante comum. Entretanto, se um dos processos modificar os dados lidos e gravá-los de volta ao sistema de arquivos, os dados lidos por outros processos serão diferentes daqueles que foram armazenados recentemente no servidor. Ao modificar estes dados e gravá-los de volta ao disco, as modificações feitas pelo processo anterior serão perdidas. Este problema é denominado de problema de coerência, ou consistência, do *cache* (*cache coherence problem*).

Algumas possíveis soluções para este problema são apresentadas em [49]:

- **Detecção da inconsistência** — os blocos possivelmente inconsistentes são detectados pelo sistema automaticamente em tempo de execução.

  Neste caso, o sistema mantém uma lista de todos os blocos possivelmente inconsistentes, decidindo as ações a serem tomadas frente às requisições aos blocos que podem ser inconsistentes.

  Este método é o mais eficiente, uma vez que os blocos possivelmente inconsistentes não são removidos do *cache*. Entretanto, a complexidade de implementação e necessidade de maiores recursos computacionais para manter informações sobre todos os blocos na memória aumenta a complexidade geral deste método.

- **Reforço da consistência** — os blocos que contêm dados inconsistentes são atualizados ou invalidados.

  Desta maneira, assim que o sistema detectar uma possível inconsistência de dados, os blocos em questão são removidos do *cache* para evitar futuros acessos a eles.

Outra técnica que pode ser utilizada é a atualização de um bloco, utilizando uma re-leitura do mesmo. Desta maneira, o bloco em *cache* permanece atualizado.

A *invalidação* do bloco em questão também é possível. Desta maneira, o bloco fica marcado com um *flag*, e é atualizado assim que houver uma tentativa de acesso a ele.

Entretanto, para poder exercer as ações sobre os blocos inconsistentes é necessário algum mecanismo para determinar quais são os blocos que apresentam inconsistência. Para isso, é possível utilizar os seguintes métodos [49]:

- ***Snoopy coherence*** – este mecanismo possibilita a troca constante de informações entre todos os clientes e servidores, atualizando constantemente os seus *caches*.

  O mecanismo apresenta a maior eficiência, uma vez que a atualização ocorre diretamente entre as entidades do sistema. Entretanto, devido à necessidade de atualizar todos os *caches* presentes no sistema, este método apresenta uma maior complexidade e precisa de um meio de propagação mais rápida das informações. Desta maneira, ele torna-se ineficiente em um sistema de arquivos em rede, onde os dados são transmitidos por uma rede de computadores.

- **Mecanismos baseados em diretório** – este método visa a criação de um *diretório* que armazena as informações entre os dados presentes nos *caches* do sistema.

  O *diretório* criado pode ser um processo que coordena as requisições de acesso aos *caches* diversos. Desta maneira, assim que um bloco torna-se atualizado em um dos *caches*, o *diretório* é informado sobre a possível inconsistência deste bloco, caso o mesmo estiver presentes em outros *caches*. Da maneira semelhante, para determinar a inconsistência de um bloco, o *diretório* é consultado.

  Este método requer um número muito menor de informações a serem trocadas entre diversas entidades do sistema, tornando-se mais eficiente em sistemas de arquivos em rede.

2.3 *Prefetching*

Existem várias maneiras de ler um arquivo: um arquivo pode ser lido seqüencialmente, do começo até o fim, por um ou vários processos; vários processos podem ler um arquivo, segmentando-o em pedaços distintos, ou apenas um segmento do arquivo pode ser lido, em seqüência pré-determinada ou aleatória.

Se o sistema de arquivos souber *a priori* as requisições que serão feitas pelas aplicações, pode ser possível carregar os dados na memória com antecedência [33]. Desta maneira, seria possível diminuir o tempo de espera entre a requisição de leitura de dados, a leitura em si e o processamento dos dados requeridos, melhorando o desempenho da operação de entrada de dados. A técnica que possibilita efetuar leitura antecipada dos dados é denominada de *prefetching*, ou *leitura antecipada*. Como o mecanismo de *prefetching*, geralmente, é utilizado nos intervalos entre as requisições de leitura de dados por parte da aplicação, este mecanismo também é conhecido como a *redução do tempo ocioso* [28].

A forma e seqüência em que os dados são lidos caracteriza os diferentes *padrões de acesso* e, conhecendo estes padrões, um *algoritmo de* prefetching pode ser utilizado para efetuar a leitura antecipada dos dados.

2.4 Regras de *prefetching*

Para obter um funcionamento eficiente dos mecanismos de leitura antecipada de dados, algumas regras básicas devem ser seguidas [30]:

- **Optimal Prefetching** — toda operação de *prefetching* deve carregar no *cache* ou na memória o próximo bloco da seqüência a ser lida, desde que este bloco ainda não tinha sido carregado.

  Ou seja, durante a operação de leitura antecipada um bloco a ser requisitado no futuro deve ser preparado para o futuro acesso pela aplicação. Um mecanismo de *prefetching* deve evitar ler os blocos que não serão requisitados pela aplicação.

- **Optimal Replacement** — se a operação de *prefetching* precisar retirar um bloco do *cache* ou da memória, o bloco a ser retirado deverá ser o de menor relevância.

  Ou seja, um bloco a ser retirado da memória não deve prejudicar o desempenho e a operação correta da aplicação.

- **Do No Harm** — um certo bloco *A* nunca deverá ser descartado para carregar o bloco *B*, se o bloco *A* será requisitado antes do *B*.

  Desta maneira, esta regra é um complemento das regras de *optimal prefetching* e *optimal replacement*. Enquanto a primeira regra especifica *quais* são os blocos a serem lidos durante a operação do *prefetching* e a segunda define as ações a serem tomadas para liberar espaço para armazenar o bloco lido, a regra de *do no harm* especifica como determinar as ações mais adequadas para o funcionamento correto das delas.

- **First Opportunity** — uma operação de leitura e substituição do bloco do *cache* ou memória nunca deve ser efetuada se ela poderia ter sido efetuada anteriormente e foi preterida por outra operação.

Utilizadas em conjunto, estas quatro regras permitem especificar o que deve ser carregado ou descartado do *cache* pelo mecanismo de *prefetching*.

2.5 Agressividade de *prefetching*

Para determinar as ações que devem ser tomadas por um mecanismo de leitura antecipada, uma política de *prefetching* deve ser empregada. Enquanto existem diversas políticas de *prefetching*, o comportamento delas é determinado de acordo com a *agressividade* [30] das mesmas, variando entre *prefetching agressivo* e *prefetching conservativo*, ou *passivo*:

- *Prefetching agressivo* - esta política executa o mecanismo de *prefetching* sempre, para qualquer leitura de dados. Isto pode trazer vantagens (dados podem sempre ser lidos antecipadamente) e desvantagens (como o tamanho do *cache* é limitado, dados antigos devem ser retirados para inserir novos dados). No pior caso, o algoritmo de *prefetching* irá retirar dados mais relevantes para inserir dados com menor relevância. Assim, em alguns casos, dados que serão acessados no futuro serão substituídos por outros dados, necessitando, assim, de mais acessos ao disco para obter os dados requisitados.

  O objetivo desta política é armazenar o máximo de dados em *cache*.

- *Prefetching passivo* - esta política executa o *prefetching* o mínimo de vezes possível. Assim, não há um grande ganho de desempenho mas, por outro lado, uma aplicação que utiliza esta política nunca irá precisar de mais acessos ao disco do que uma aplicação que não utiliza nenhum mecanismo de *prefetching*.

  O objetivo desta política é prover o mínimo de acessos adicionais aos discos.

Resumindo-se as duas políticas, podemos dizer que o *prefetching agressivo* procura executar a leitura antecipada sempre que for possível, e o *prefetching passivo* – sempre que for necessário.

De acordo com testes efetuados [50] [51] [52], foi descoberto que as políticas mais *agressivas*, em média, são mais eficiente que as políticas *passivas*.

Entretanto, dependendo da aplicação estudada e do *padrão de acesso* da aplicação, cada política pode oferecer resultados diferentes.

2.6 Padrões de acesso

Para determinar os próximos blocos a serem requisitados pelo mecanismo de leitura antecipada, é necessário determinar um *padrão* nas operações de leitura da aplicação. Este padrão é conhecido como *padrão de acesso* e tem papel fundamental para o funcionamento dos mecanismos de *prefetching* [33].

Os principais padrões de acesso seqüencial são *one-block look-ahead, n-block look-ahead* e *infinite-block look-ahead* [53]. Acesso sobre posições aleatórias dos arquivos não caracterizam um padrão de acesso, tornando difícil a atuação correta dos mecanismos de *prefetching*.

O funcionamento destes padrões de acesso é descrito a seguir:

- ***one-block look-ahead***: a aplicação lê o arquivo seqüencialmente, requisitando um bloco de cada vez.

  Neste padrão de acesso, o algoritmo de *prefetching* deve carregar o próximo bloco em *cache* durante a operação de leitura antecipada.

- ***n-block look-ahead***: a aplicação lê o arquivo em grupos de *n* blocos de cada vez.

  De acordo com este padrão de acesso, o *algoritmo de prefetching* deve carregar os *n* blocos em *cache* durante a operação de leitura antecipada.

- ***infinite-block look-ahead***: a aplicação lê o arquivo inteiro de uma vez.

  O comportamento do algoritmo de *prefetching* neste caso consiste em preenchimento do *cache* com os blocos ainda não lidos, requisitando um número de blocos suficiente para preencher o *cache*.

- **leitura não-seqüencial**: a aplicação lê apenas alguns dos blocos do arquivo, sem nenhuma seqüência pré-determinada.

  Neste caso, o comportamento do algoritmo de *prefetching* é indefinido, sendo que não é possível determinar uma seqüência de acessos aos blocos. É possível utilizar heurísticas para determinar um padrão de acesso não-linear, entretanto, este processo é bastante complexo [52].

A determinação dos padrões de acesso pode ser feita analisando a seqüência de blocos lidos pela aplicação [53]. Entretanto, na alteração entre um padrão de acesso e outro é possível perder o desempenho, visto que o mecanismo de *prefetching* pode identificar da maneira errada o novo padrão de acesso, requisitando blocos que não serão utilizados posteriormente.

2.7 Algoritmos de *prefetching*

Para cada padrão de acesso, o método utilizado para ler os dados é diferente. Enquanto é possível utilizar a mesma técnica para efetuar a leitura antecipada para todos os padrões de acesso, a eficiência desta abordagem será inferior a uma abordagem que leva em questão o tipo da aplicação e o do padrão de acesso utilizado.

A técnica utilizada para efetuar a leitura antecipada de dados de acordo com o padrão de acesso determinado é denominada de *algoritmo* de *prefetching*.

Entre os algoritmos de *prefetching* mais conhecidos podemos citar os seguintes:

- ***Aggressive*** – este algoritmo implementa um algoritmo de *prefetching* que segue a política de *prefetching* agressivo [30] [32].

  Este algoritmo procura executar a leitura antecipada sempre que for possível, requisitando um maior número de blocos.

- ***Passive*** – este algoritmo implementa um algoritmo de *prefetching* que visa executar o menor número de requisições adicionais ao disco, seguindo a política de *prefetching passivo* [30].

  O algoritmo procura oferecer um menor número de acessos adicionais ao disco, diminuindo a taxa de requisição de blocos antecipadamente.

- ***TIP2*** – este algoritmo utiliza as *dicas* das aplicações para determinar as ações a serem tomadas pelo mecanismo de *prefetching* [54].

  Desta maneira, as aplicações especificam qual é o padrão de acesso esperado, a utilização do *cache* prevista e outras informações, e o mecanismo de *prefetching* utiliza estes dados para executar as operações de leitura antecipada.

- ***Fixed horizon*** – o algoritmo *fixed horizon* visa determinar as operações a serem executadas pelo mecanismo de leitura antecipada analisando o tempo necessário para obter um bloco de dados e tempo requerido para processá-lo [32].

  Desta maneira, a relação entre tempo necessário para obter um bloco, e o tempo que a aplicação vai demorar para processar o bloco lido é utilizada para determinar a probabilidade deste bloco ser lido durante a operação de leitura antecipada.

- ***Reverse aggressive*** – este algoritmo visa obter a mesma eficiência do algoritmo *aggressive*, baseando as suas decisões no histórico de acessos passados e visando balancear melhor a carga entre diversos discos do sistema [32].

O algoritmo, entretendo, apresenta uma complexidade muito maior comparando-se com outros algoritmos de *prefetching* devido a um número maior de informações a serem analisadas para tomar uma decisão de leitura antecipada.

- ***Forestall*** – este algoritmo visa oferecer uma solução de *prefetching* balanceada entre o algoritmo *reverse aggressive* e *fixed horizon* [32].

  O algoritmo utiliza tanto as operações de *prefetching agressivo* para obter os blocos de maneira eficiente quanto as do algoritmo *fixed horizon* para requisitar os blocos de maneira apropriada, procurando determinar os blocos mais relevantes a serem lidos pelo mecanismo de *leitura antecipada*.

  O algoritmo visa evitar as possíveis substituições de blocos que podem ser feitas pelo algoritmo *aggressive*, limitando o "alcance" do mesmo. Desta maneira, até uma certa distância entre o "ponteiro" de leitura atual e o bloco a ser requisitado o algoritmo *aggressive* é utilizado, e depois desta distância é empregado o algoritmo *fixed horizon*, visando limitar a escolha de blocos a serem lidos.

- ***NOM*** – este algoritmo utiliza um *look-ahead global* para determinar a seqüência dos blocos a serem requisitados [55].

  No caso de diversos servidores, o algoritmo, à cada operação de leitura, requisita o próximo bloco não-lido de cada um dos servidores. O algoritmo utiliza um *look-ahead global* para ler os blocos de todos os servidores, sem levar em conta os *caches* locais dos mesmos. Desta maneira, os cliente sempre terão os blocos prontos para a leitura antecipada.

  O algoritmo utiliza um *look-ahead global* dos servidores, requisitando os blocos em seqüência da leitura pela aplicação. Desta maneira, se um dos discos tiver blocos que serão acessados mais no futuro, estes blocos somente serão lidos depois de todos os outros blocos dos outros servidores, que devem ser requisitados anteriormente, forem lidos. Desta maneira, os dados de cada servidor permanecem em *cache* para posterior leitura por um período curto de tempo.

- ***GREED*** – este algoritmo utiliza um *look-ahead global* em conjunto com um *look-ahead local* de cada servidor para determinar os blocos a serem requisitados [55].

  O funcionamento do algoritmo é semelhante ao do algoritmo *NOM*, com a diferença de que o algoritmo *GREED* somente lê os blocos de cada servidor se houver espaço disponível em *cache* local do servidor correspondente. O *look-ahead local* é utilizado para fazer uma leitura dos blocos antecipada para cada servidor.

Desta maneira, o algoritmo irá requisitar os blocos de cada servidor assim que for possível, mantendo-os em *cache* por mais tempo.

- ***SUPERVISOR*** – este algoritmo visa atribuir prioridades às requisições para posterior reorganização das mesmas, visando obter um desempenho melhor [56].

  O algoritmo visa reorganizar e agrupar os blocos a serem lidos de acordo com a localização dos mesmos. Desta maneira, ele procura agrupar as requisições a servidores (ou discos) diferentes para oferecer um melhor desempenho das operações, de maneira semelhante ao protocolo *SCSI-2* [57], utilizado nos discos rígidos.

- ***Full-file-on-open*** – o algoritmo visa executar uma leitura completa do arquivo na hora de abertura do mesmo [58].

  Esta abordagem é especialmente interessante para os arquivos pequenos, cujo tamanho é menor do que o tamanho do *cache*. Desta maneira, a aplicação consegue ler todos os dados requisitados diretamente da memória, aumentando o desempenho das operações de leitura de dados.

- ***Adaptive++*** – este algoritmo utiliza um histórico de acessos à memória para determinar o comportamento da aplicação e efetuar a leitura antecipada de dados de maneira correta [59].

  Esta técnica de *prefetching* utiliza o histórico de acessos à memória para determinar o padrão de acesso da aplicação. Os padrões de acesso determinado pelo algoritmo são os padrões conhecidos como *repeated-stride* (o intervalo entre os acessos consecutivos é repetitivo, ou seja, a determinação do próximo bloco na seqüência é possível) e *repeated-phase* (o conjunto de dados requisitados na operação é repetitivo, ou seja, é possível determinar o conjunto de blocos que devem ser requisitados baseando-se no bloco atual). O algoritmo de *prefetching* não é executado quando a aplicação não apresenta nenhum destes modos de acesso aos dados para evitar requisições adicionais de leitura de dados.

- ***B+*** – este algoritmo utiliza as *invalidações* dos blocos em *cache* para determinar os próximos blocos a serem requisitados pela aplicação [60].

  Este técnica de *prefetching* analisa a seqüência de blocos invalidados para determinar os blocos que devem ser requisitados. A técnica assume que, se um bloco foi requisitado recentemente, porém foi invalidado, é provável que o mesmo bloco seja requisitado novamente.

A criação de um algoritmo de *prefetching* adaptativo permite resolver o problema de escolha do algoritmo mais apropriado: analisando o padrão de acesso, um algoritmo ideal consegue determinar o método que deve ser utilizado para ler os dados. Este algoritmo deve ter capacidade para determinar a mudança no padrão de acesso, alterando o método da leitura de dados.

2.8 Integração de *Cache* com *Prefetching*

É possível utilizar os mecanismos de *cache* e *prefetching* de maneira mais eficiente caso eles sejam integrados [61].

Sem nenhuma integração entre *cache* e *prefetching*, o *cache* é utilizado apenas para conter os dados já lidos, e o *prefetching* carrega novos dados diretamente para a memória. Isto diminui as vantagens do *cache*, já que é possível que os dados raramente sejam acessados novamente, e piora o desempenho do mecanismo de *prefetching*, por necessitar de um local separado na memória para cada leitura de dados.

Em um mecanismo integrado de *cache* e *prefetching*, os dados obtidos através das leituras antecipadas são colocados no *cache*, de onde são lidos posteriormente pelo aplicativo. Assim, é possível aumentar o desempenho destes mecanismos, de forma que o *cache* mantenha os dados que serão acessados e o *prefetching* não precise alocar um novo segmento da memória toda vez que os dados forem carregados.

O desempenho e a eficiência de um mecanismo integrado de *cache* e *prefetching* são, geralmente, superiores aos sistemas independentes [61] [62].

3 *Sistema de arquivos* NPFS

3.1 Funcionamento do sistema

NPFS (Network Parallel File System) [23] é um sistema de arquivos paralelos distribuído, que é controlado por software e funciona sobre uma rede de computadores. A arquitetura alvo considerada por este projeto consiste de um conjunto de estações de trabalho interligadas em rede, cada uma com seu próprio disco local, que pode ser compartilhado.

Neste sistema de arquivos, os dados são divididos em *segmentos*, distribuídos pelos diversos servidores para serem lidos em paralelo, de maneira similar à figura 2, vista anteriormente.

A distribuição dos dados é decidida na hora de criar um arquivo. É possível especificar o número de segmentos e o tamanho das unidades de distribuição.

O sistema de arquivos utiliza o modelo cliente-servidor para criar servidores espalhados na rede. Existem três tipos de processos: *mestre, servidor* e *cliente*, como mostrado na figura 3.

O **mestre** é o processo responsável pelas funções de iniciação do sistema; ele cuida da ativação e da manutenção dos servidores, e trata também da abertura e fechamento de arquivos.

Os **servidores** provem acesso aos segmentos de dados que estão armazenados em seus discos locais. O processo servidor é executado em cada computador cujo disco local é compartilhado no sistema.

Os **clientes** são os processos dos usuários. Suas interações com o resto do sistema são efetuadas através das funções disponíveis em uma biblioteca de funções.

A ativação do sistema consiste da criação de um processo *mestre*. Depois disto, o mestre se encarrega de iniciar os *servidores* e espera as requisições dos *clientes*. Uma vez abertos os arquivos, os clientes interagem diretamente com os servidores.

A *API* (*Application Programming Interface*) do *NPFS* é baseada na sintaxe padrão da linguagem *C*, que utiliza as funções *open(), read(), write(),* entre outras. Assim, é possível conver-

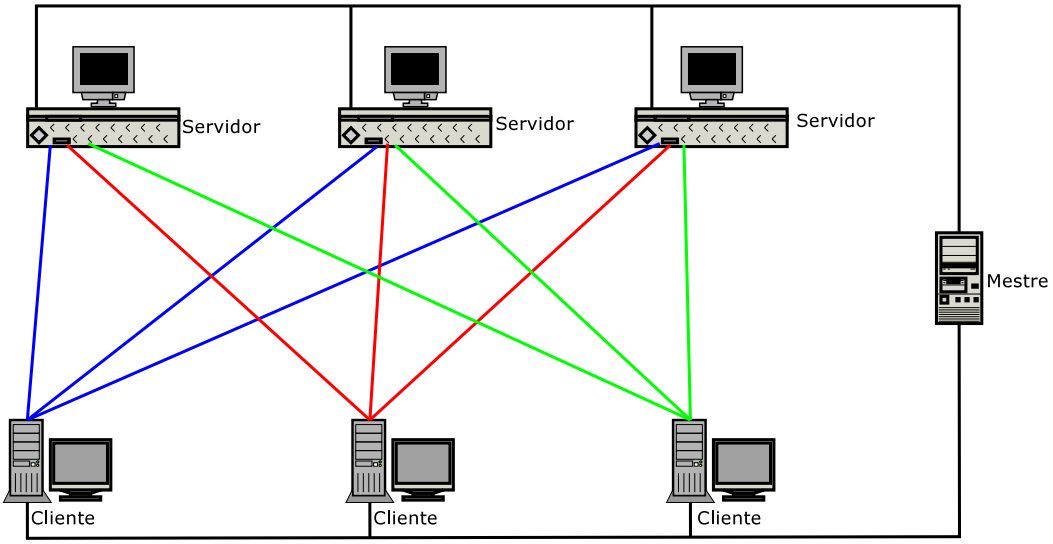

Figura 3: Arquitetura do *NPFS*

ter aplicações que utilizam funções comuns da linguagem *C* em aplicações que utilizam funções *NPFS*.

Nas operações de leitura, as funções da *API* executadas no cliente determinam automaticamente quais servidores serão necessários e enviam diretamente a cada um deles as requisições relevantes. O mesmo ocorre na escrita. Na versão original do *NPFS*, ambas as operações são *síncronas* e não há mecanismos de *cache* ou *prefetching*.

3.2 Interação entre processos do NPFS

Para possibilitar a interação entre vários processos do NPFS (mestre, servidores e clientes), um protocolo de comunicação foi desenvolvido. Nesta seção apenas o funcionamento básico do protocolo é apresentado; a descrição detalhada pode ser encontrada em [5].

- **Mestre** — Processo *Mestre* é responsável pelas funções globais do sistema, tais como:
  - **Ativação e Desativação do sistema** — os *servidores* são criados pelo processo *mestre* durante a ativação do sistema. Para desativar o sistema, o *mestre* desliga os servidores. Desta maneira, é possível ter um controle centralizado sobre todos os servidores do sistema.
  - **Abertura e Fechamento de arquivos** — ao receber uma mensagem de abertura de arquivos, o mestre verifica a existência do arquivo e, caso ele for encontrado, o *mestre* contata os servidores nos quais este arquivo é armazenado. Para fechar um

arquivo, ocorre um processo semelhante – *mestre* contata os servidores correspondentes para fechar o arquivo.

- **Servidores** — Os *servidores* se encarregam apenas de ler e escrever os dados requisitados pelos clientes.

  Os servidores possuem somente dois estados para cada arquivo – *espera* e *tratamento*.

  Inicialmente, os servidores permanecem no estado de espera até receber uma requisição de abertura de arquivo ou de leitura de dados. Ao receber esta requisição, os processos passam para o estado de tratamento do arquivo e permanecem neste estado até receber uma requisição de fechamento de arquivo.

  O tratamento da requisição depende da operação requisitada, ou seja, leitura de dados, escrita, remoção do arquivo, entre outras.

- **Clientes** — Por motivos de desempenho, a maior parte do processamento está concentrada nos *clientes*. As operações de escrita, leitura, remoção e troca de nome dos arquivos são efetuadas pelos clientes.

  Desta forma, para cada requisição o cliente deve determinar os parâmetros do arquivo, tais como o ponteiro atual de leitura ou escrita de dados, servidores correspondentes, segmentos e *stripes* necessários para a requisição e outras informações. Estas informações são repassadas para os servidores correspondentes da maneira agrupada – isto é, as mensagens são enviadas simultaneamente, sem esperar pela resposta dos servidores correspondentes. Depois do envio das mensagens, as respostas recebidas são ordenadas e processadas.

  Caso ocorre *timeout* no recebimento das mensagens, as mensagens cuja resposta não foi recebida são re-enviadas. Um mecanismo de determinação automática de *timeouts* das requisições em função da utilização da rede e dos discos é suportado no sistema [63].

  A comunicação entre os processos é efetuada através da troca de *PDUs*[1], que utilizam o protocolo *UDP*[2], desenvolvidas para o NPFS. As mensagens consistem de um cabeçalho de tamanho variável, seguido pelos dados. O cabeçalho possui vários campos, sendo que os tamanhos dos campos não são fixos. A separação entre campos adjacentes é feita através de um caracter especial.

[1] *Protocol Data Unit.*
[2] *User Datagram Protocol.*

3.3 API do NPFS

As funções do *NPFS* visam seguir o padrão *POSIX* das funções para a entrada e saída básica. Para diferenciar as funções do *NPFS* das funções de entrada e saída padrão foi criada uma *API*.

As funções oferecidas pela *API* tem como objetivo ser o mais semelhantes possível com as funções nativas do sistema, geralmente recebendo o mesmo número de parâmetros e retornando os valores esperados.

As seguintes funções são introduzidas na *API* do NPFS:

- **p_open** — a função *p_open* é utilizada para configurar os servidores e o *mestre* para a abertura de um arquivo.

  A função aceita os seguintes parâmetros:

  - **Nome do arquivo** — nome completo do arquivo a ser aberto.
  - ***Flags*** — modo de abertura de arquivo – escrita, leitura, etc.
  - **Permissões de abertura** — especifica se apenas o usuário pode ler ou escrever no arquivo, se o grupo a qual o usuário pertence pode ler ou escrever no arquivo, etc.
  - **Visão do arquivo** — em um sistema de arquivos paralelo, um arquivo pode ser aberto por vários processos, sendo que cada um deles terá uma visão sobre o arquivo.
  - **Número de segmentos** — indica em quantos segmentos o arquivo será dividido para ser guardado nos servidores.
  - **Tamanho do *stripe*** — indica o tamanho do cada segmento.
  - **Lista de servidores** — é possível especificar os servidores nos quais o arquivo será armazenado. Se esta variável não for especificada, o processo *mestre* irá distribuir o arquivo entre todos os servidores.

  A função *p_open* retorna o identificador do arquivo se o arquivo foi aberto com sucesso ou o código de erro se ocorrer algum erro na abertura do arquivo.

- **p_close** — a função *p_close* é utilizada para fechar um arquivo.

  Os seguintes parâmetros são aceitos pela função:

  - **Identificador do arquivo** — identificador do arquivo é retornado pela função *p_open*.

  - **Parâmetros de fechamento de arquivo** — é possível especificar se o programa deve esperar até que todos os servidores confirmarem o fechamento do arquivo ou se é possível retornar ao programa sem ver a resposta dos servidores.

  Depois de fechar o arquivo, o seu identificador pode ser reutilizado.

- **p_read** — esta função é utilizada para ler dados do arquivo aberto.

  Os parâmetros aceitos pela função são os seguintes:

  - **Identificador do arquivo** — identificador do arquivo é retornado pela função *p_open*.
  - **Local de memória** — dados lidos serão armazenados neste local de memória.
  - **Número de bytes** — especifica o número de bytes a serem lidos pela função.

  A função *p_read* retorna o número de bytes lidos ou o código de erro, caso ocorra alguma falha na leitura de dados.

- **p_write** — esta função escreve dados no arquivo aberto.

  Os parâmetros e os valores de retorno desta função são os mesmos da função anterior, *p_read*.

- **p_lseek** — esta função efetua o *seek* para uma determinada posição do arquivo.

  Os seguintes parâmetros são aceitos pela função:

  - **Identificador do arquivo** — o identificador do arquivo é retornado pela função *p_open*.
  - ***Offset*** — especifica quantos bytes tem que ser percorridos.
  - **Método** — especifica se é necessário percorrer o arquivo desde o começo dele, do final ou da posição atual.

- **p_rename** — esta função renomeia o arquivo.

  Os seguintes parâmetros são aceitos pela função:

  - **Nome antigo** — especifica o nome antigo do arquivo.
  - **Novo nome** — especifica novo nome do arquivo em questão.

- **p_unlink** — esta função remove o arquivo em questão do sistema. Único parâmetro recebido é o nome do arquivo a ser removido.

Além destas funções, existe uma série de funções de sistema que se encarregam de criar os processos, servidores; efetuar a troca de mensagens e ativar e desativar o sistema em si. Estas funções são *add_server, rm_server, f_query, s_query, spawn, get_pind, get_group, g_query, p_kill, g_kill, g_free, p_send e p_receive*.

3.4 Utilização do NPFS

Como exemplos de utilização do NPFS podemos destacar dois trabalhos em áreas diferentes — uma aplicação de banco de dados [64] e um servidor de vídeo sob demanda distribuído [65].

O primeiro trabalho visou criar um banco de dados distribuído utilizando um sistema de arquivos paralelo. De acordo com o trabalho, um ganho significativo no desempenho das operações foi atingido, utilizando um sistema de arquivos paralelo e distribuído.

O segundo trabalho visa criar um servidor de vídeo sob demanda distribuído, que possa aproveitar as vantagens oferecidas por um sistema de arquivos paralelos para oferecer um desempenho maior no processamento dos dados. O servidor criado baseou-se em *mplayer* [66], e teve como objetivo determinar a influência de utilização do sistema de arquivos *NPFS* no desempenho das operações de entrada e saída para arquivos multimídia.

De maneira geral, o uso dos arquivos paralelos em rede mostra-se viável. A latência das requisições e a perda de desempenho no acesso concorrente a arquivos compartilhados, entretanto, ainda são problemas que precisam ser resolvidos. A utilização de um mecanismo de *cache* e *prefetching* é uma potencial solução para esses casos.

4 Trabalhos relacionados

O objetivo desta seção é a apresentação dos trabalhos relacionados às áreas estudadas, visando descrever o estado da arte tanto nas áreas de *caching* e *prefetching*, quanto na utilização dos sistemas de arquivos paralelos.

Desta forma, podemos dividir os trabalhos apresentados nesta seção em três conjuntos: trabalhos relacionados a projetos e utilização de mecanismos de *cache*; trabalhos relacionados a algoritmos e mecanismos de *prefetching* e trabalhos relacionados aos sistemas de arquivos paralelos como um todo.

Entre os trabalhos relacionado tanto à área de *cache* e *prefetching* quanto aos estudos dos sistemas de arquivos paralelos em sim, podemos destacar principalmente os trabalhos de David Kotz e Carla Shlatter Ellis ([31] [33] [67] [68] [69] [70] [71] [72]). Os trabalhos abrangem uma grande área, mostrando estudos tanto sobre os mecanismos de *caching* ([31]) e *prefetching* ([33] [67]), quanto sobre a integração deles ([70]). Estudos sobre sistemas de arquivos paralelos em geral também são de grande relevância ([68] [71]). Além disto, uma seleção de bibliografia da área de entrada e saída paralela é apresentada em [69], sendo que a mesma é atualizada constantemente.

Os conceitos de importantes para a determinação do padrão de acesso para um algoritmo de *prefetching*, vistos na seção anterior, são apresentados em [31]. Mecanismos de escrita de dados contidos em *cache*, tais como *writethrough, writeback, writefree e writefull* também são discutidos neste trabalho. Além disto, a eficiência dos algoritmos discutidos é analisada.

Podemos citar [33] como um complemento do trabalho anterior. O trabalho discute os padrões de acesso e apresenta métodos para a determinação dos mesmos pela aplicação, estudando diversos padrões de acesso, tais como *one-block look-ahead, infinite-block look-ahead e portion recognition*. Um algoritmo *preditor* para a determinação do tipo de acesso utilizado é apresentado no trabalho.

Os trabalhos que discutem as aplicações dos arquivos paralelos em geral, os possíveis problemas e as soluções para elas e as técnicas para melhorar o desempenho são discutidos em [9], [68], [71] e [72].

As aplicações de arquivos paralelos são discutidas por Ron Oldfield e David Kotz em [9], apresentando as diferentes classes de aplicações para os arquivos paralelos.

Alguns dos problemas provenientes dos sistemas de arquivos paralelos são estudados, e uma solução para os mesmos é proposta em [68]. Neste trabalho, é apresentada uma divisão da funcionalidade de um sistema de arquivos paralelo em duas partes – um núcleo fixo que é padrão para todas as variações de sistemas de arquivos, e uma *API* de alto nível que visa oferecer a funcionalidade necessária para cada uma das aplicações. Uma abordagem semelhante a essa já vem sendo utilizada nos sistemas *unix*, onde a funcionalidade básica de entrada e saída é oferecida pelos processos de sistema, e é estendida por uma *API* de mais alto nível. A solução semelhante é utilizada no sistema de arquivos *NPFS* [23], utilizado neste trabalho, onde o sistema de arquivos oferece uma *API* básica que pode ser estendida pelas aplicações do usuário.

Um estudo sobre as características de acesso ao sistema de arquivos em sistemas multiprocessados é apresentado em [71] e [72]. No trabalho, foram analisados os acessos de todos os processos presentes no sistema por um período de duas semanas. Baseando-se nos resultados obtidos, foi demonstrado que um mecanismo de *caching* tem a capacidade de oferecer ótimos resultados e a determinação de um padrão de acesso tem grande relevância para a melhoria do desempenho do sistema ([71]).

Um grupo de trabalhos relacionado à integração dos mecanismos de *cache* e *prefetching* para sistemas de arquivos distribuídos são de autoria de Toni Cortes, Sergi Girona e Jesús Labarta ([73] [61] [52] [58] [62] [51]).

Esses trabalhos apresentam estudos sobre mecanismos cooperativos de *caching* e *prefetching* para sistemas de arquivos paralelos e distribuídos ([62]), junto com o sistema de arquivos *PAFS* ([73] e [51]) e um sistema de *cache* para máquinas paralelas, denominado de *PACA* ([58]).

O sistema de arquivos *PAFS*, apresentado em [73] e [51] é um sistema de arquivos paralelo, que pode ser utilizado tanto em redes de computadores quanto em máquinas paralelas. A integração dos mecanismos de *cache* dos diversos servidores em um *cache global*, que é um *cache* único e compartilhado, é discutida nos trabalhos referenciados. A consistência dos *caches* é mantida utilizando "*tokens*" repassados entre *caches* distintos. O trabalho apresenta as influências do tamanho do *cache*, algoritmo de *prefetching* e a atualização do *cache* no desempenho geral do sistema de arquivos.

A discussão do *PAFS* é continuada em [61]. O trabalho introduz os diferentes tipos de acesso ao *cache – local cache hit*, *remote cache hit* e *global cache hit*. A utilização de servidores de *cache*, denominados de *cache servers*, é apresentada e discutida no trabalho. Visando manter

a coerência entre diversos *caches* e diminuir a comunicação entre servidores, é utilizado um mecanismo de *hashing* para determinar de maneira única em qual dos servidores de *cache* é localizado um determinado bloco de dados. Desta forma, a coerência dos dados é mantida removendo a replicação de dados entre *caches* diferentes. Algoritmos de distribuição são empregados para coordenar acesso a regiões diferentes de arquivos por meio de servidores distintos de *cache*.

Um mecanismo de tolerância a falhas, cujo funcionamento é similar ao funcionamento do *RAID 5* [74], é introduzido em [73] e [51]. O algoritmo de *cache* cooperativo, *N-Chance Forwarding*, também é introduzido nos trabalhos.

O algoritmo de substituição de páginas em *cache* denominado *LRU-interleaved* junto com um algoritmo agressivo de *prefetching* é apresentado em [50]. O trabalho descreve um mecanismo cooperativo de *cache* para máquinas paralelas, cujo sistema operacional é baseado na arquitetura de *microkernels* [75].

O algoritmo *LRU-interleaved* é uma variação do algoritmo *LRU* clássico otimizado para sistemas de arquivos paralelos. O desempenho superior do algoritmo proposto nas operações de entrada e saída de dados foi comprovado pelos testes realizados no trabalho. Um outro algoritmo apresentado no trabalho é denominado de *full-file-on-open*. O algoritmo *full-file-on-open* é um algoritmo que tem como objetivo reduzir o número excessivo de requisições desnecessárias, carregando o maior número de dados em *cache* antecipadamente.

Em [52], é feito um estudo das técnicas de cache utilizadas atualmente nos sistemas de arquivos paralelos, descrevendo também algoritmos de prefetching existentes. Um novo algoritmo de prefetching, *IS_PPM*, é apresentado. Este algoritmo utiliza técnicas de estatística, tais como *as cadeias de Markov*, para determinar os blocos a serem lidos durante as operações de leitura antecipada.

Além disto, este trabalho apresenta estudo sobre a transformação dos algoritmos simples de *prefetching* em algoritmos mais avançados, capazes de tratar um número maior de informações adicionais, a serem utilizadas em análises estatísticas, para aumentar a eficiência das operações de leitura antecipada.

Uma série de estudos sobre a implementação de um mecanismo de *cache* distribuído é apresentada em [76]. Neste trabalho, o mecanismo de *cache* é baseado na existência de uma rede de alta velocidade, utilizada como um meio de armazenamento comum pra todos os participantes de uma rede de *workstations* (ou para todos os nós de um *GRID* [12]). Uma adaptação automática das aplicações do usuário à velocidade da rede é possibilitada através da utilização dos

mecanismos de qualidade de serviços (*QOS*).

O trabalho introduz um sistema de armazenamento paralelo e distribuído *DPSS* [1], que consiste de um conjunto de *workstations* que funcionam como servidores de blocos de dados propriamente ditos. Cada workstation contém diversos controladores de discos com vários discos ligados a cada um deles. A implementação e o funcionamento do sistema é semelhante ao sistema de arquivos *NPFS* [23], visto anteriormente.

A utilização do *Globus Metacomputing Directory Service* [77], que faz parte do projeto *Globus* [78] possibilita a alteração do número de servidores utilizados e do tamanho dos discos *on-the-fly*, possibilitando uma re-configuração automática do sistema. Outros aspectos interessantes apresentados neste trabalho são a utilização do protocolo de comunicação *TCP/IP* com ajuste automático do tamanho de *buffers* e dos *timeouts* junto com mecanismos de balanceamento de carga.

Entre os trabalhos voltados para a comparação das políticas de *cache* para diversos tipos de aplicações podemos relacionar [46] e [48]. O primeiro trabalho visa avaliar as políticas de *cache* para servidores de armazenamento de dados multimídia. O foco do trabalho são os sistemas de *cache* baseados em documentos, diferentemente dos sistemas de *cache* convencionais, baseados em blocos de dados.

O trabalho descreve e compara diferentes algoritmos de *cache – Space x Age* (com e sem a utilização de um *cache* cooperativo), *LRU*, *MRU*, com as duas variações deste último – *AVI-MRU*, utilizado preferencialmente para arquivos multimídia, e *SIZE-MRU*, que leva em questão o tamanho do documento para a lógica das as operações. Uma política de *caching* cooperativo, denominada *greedy forwarding* é apresentada e estudada no trabalho.

O segundo trabalho ([48]) já é mais voltado para mecanismos de *cache* utilizados em servidores *web*. No trabalho, são apresentados e comparados os algoritmos *LRU-TH*, *LRU-K-TH* e *LRU-MIN*, que são variações do algoritmo *LRU* clássico. Um outro algoritmo apresentado é denominado *LFU-SIZE*. O algoritmo utiliza uma fila de prioridades para os documentos, que é definida de acordo com o tamanho dos mesmos. Algoritmos estáticos (tabelas *hash*) são utilizados para procurar os documentos. O trabalho apresenta e discute um estudo sobre a influência do algoritmo de *cache* na utilização de memória e na carga do sistema. Os algoritmos propostos são avaliados, visando determinar as condições ótimas de funcionamento para cada um deles em aplicações estudadas no trabalho.

Os ganhos obtidos devido a utilização de um sistema de *cache* unificada são discutidos

[1] http://www-didc.lbl.gov/DPss/

em [79]. No trabalho, um sistema unificado de *caches* e *buffers* é utilizado em um sistema operacional UNIX (*FreeBSD* [80]). O trabalho demonstra que a unificação das operações sobre os *caches* presentes no sistema resulta em um desempenho melhor devido à redução do número de cópias necessárias na memórias.

O sistema proposto é validado em aplicações que realizam freqüentes acessos à memória, tais como servidores *web*. De acordo com os testes realizados, o mecanismo integrado de *cache* apresenta um ganho médio entre 40 e 80% no desempenho geral das operações de entrada e saída em relação a um sistema comum.

Tratando-se da integração de mecanismos de *cache* e *prefetching*, é de grande importância o trabalho [30], que discute as possíveis implementações dos mecanismos de *prefetching*, discutindo os algoritmos de *prefetching agressivo* e *passivo* e apresentando as vantagens e desvantagens de cada abordagem. Neste trabalho, as principais abordagens que podem ser utilizadas para a implementação de um mecanismo de *prefetching* são discutidas, definindo as regras básicas que um mecanismo de leitura antecipada deve seguir para ser eficiente. As quatro regras básicas de *prefetching* (*Optimal Prefetching, Optimal Replacement, Do No Harm, First Opportunity*) são apresentadas e comprovadas com estudos e exemplos. A eficiência dos algoritmos *conservativos* de *prefetching* (como o *prefetching passivo*) e dos algoritmos mais *agressivos* é comparada, focando-se tanto na taxa de utilização de *cache* quanto em eficiência geral das operações.

Trabalhos de grande relevância para a área de leitura antecipada são [28], [32] e [34]. Os trabalhos definem o *prefetching* como os *mecanismos para a redução do tempo ocioso* e discutem a implementação dos mecanismos de *cache* e *prefetching*, além de introduzir vários algoritmos de *prefetching*, tais como *forestall*, *fixed horizon*, *aggressive* e *reverse aggressive*. Cada um destes algoritmos torna-se mais efetivo em alguns casos – no caso do algoritmo *fixed horizon*, que é baseado no algoritmo *TIP2* [54], o algoritmo determina as ações a serem tomados pelo mecanismo de *prefetching* baseando-se no tempo necessário para obter um bloco de dados e tempo necessário para processá-lo. O algoritmo *aggressive* descrito no trabalho é um algoritmo de *prefetching agressivo* clássico, que tenta executar as operações de leitura antecipada sempre que for possível.

Um algoritmo de *prefetching* interessante é o *reverse aggressive*, que é semelhante ao algoritmo *aggressive*, com a diferença de que este toma as decisões sobre os blocos a serem lidos antecipadamente baseando-se em um número grande de informações adicionais. O algoritmo tem como objetivo um melhor balanceamento da carga entre os discos do sistema. Este algoritmo, de acordo com os testes realizados [32], é um algoritmo que se aproxima de um *algoritmo*

ótimo de prefetching.

Último dos algoritmos descritos nestes trabalhos é o algoritmo *forestall*, que tem como o objetivo a incorporação das vantagens dos três algoritmos anteriores, procurando juntar o desempenho do *reverse aggressive* com a facilidade de implementação do *fixed horizon* e *aggressive*.

De acordo com os testes realizados, os 4 algoritmos de *prefetching* estudados foram muito superiores à leitura seqüencial sem prefetching. Os algoritmos *fixed horizon* e *forestall* tiveram o melhor desempenho nas aplicações CPU-bound, enquanto o algoritmo *agressivo* teve um bom desempenho nas aplicações IO-bound.

Além disto, estes trabalhos discutem a funcionalidade dos mecanismos de *prefetching* em função da carga do disco que contém o bloco referenciado. Desta maneira, um mecanismo para balanceamento automático de carga é utilizado para obter um desempenho melhor nas operações de entrada e saída.

Além disto, os trabalhos introduzem o sistema de *prefetching* para memória global (*PGMS - Prefetching Global Memory System*) – um sistema que integra os *caches locais* de cada servidor em um *cache global*, visível para todos os servidores, facilitando a utilização de *cache* e a troca de informações entre os servidores.

Um trabalho que tem como objetivo a discussão da influência do *prefetching* baseando-se em *cache* local dos discos rígidos é apresentado em [81]. Os algoritmos de *prefetching* baseados na estrutura física do disco são analisados. A divisão de um *cache* em diversos *segmentos* visando servir diversos arquivos ao mesmo tempo é discutida. No trabalho, o termo *declustering* é utilizado para descrever o mecanismo de *striping*.

Uma avaliação geral dos algoritmos de *prefetching* existentes é apresentada em [55]. No trabalho são propostos dois novos algoritmos de *prefetching – NOM* e *GREED*. O algoritmo *NOM* é um algoritmo de *prefetching global* e *GREED* é um algoritmo de *prefetching* que utiliza o *look-ahead* local para determinar os próximos blocos da seqüência a serem lidos.

Um novo algoritmo de *prefetching – SUPERVISOR* é apresentado em [56]. O algoritmo apresentado associa prioridades diferentes para todas as requisições de leitura de dados, possibilitando um desempenho maior das operações de entrada e saída através da re-organização das requisições.

O trabalho demonstra que tanto os mecanismos de *cache* quanto os de *prefetching* devem ser adaptados para sistemas de arquivos paralelos; os algoritmos de *cache* e *prefetching* seqüenciais comuns podem prejudicar significativamente o desempenho das operações de entrada e saída.

A integração entre as técnicas de consistência do *cache* e os mecanismos de leitura antecipada é discutida em [60] e [59]. Os trabalhos apresentam os algoritmos de *prefetching* denominados de *B+* e *Adaptive++*, que utilizam o histórico de acessos à memória e as invalidações dos blocos mantidos em *cache* para determinar os blocos a serem lidos antecipadamente pelo mecanismo de *prefetching*.

E finalmente, um trabalho que descreve a modelagem e a implementação de um sistema de *prefetching* para ser utilizado em um sistema de arquivos paralelos é apresentado em [3]. O trabalho apresenta um sistema de *prefetching* para o sistema de arquivos *PFS - Intel Paragon Parallel File System* [82] [83]. O sistema é desenvolvido por *Intel* e suporta múltiplos modos de funcionamento. O trabalho demonstra que a utilização de um mecanismo de *prefetching* possibilitou um aumento significativo de desempenho das operações de entrada e saída na maioria dos casos estudados.

5 *Projeto de um sistema integrado de* cache *e* prefetching

5.1 Motivação

Considerando os benefícios potenciais dos mecanismos de *cache* e *prefetching*, este trabalho apresenta uma implementação destes mecanismos para ser em utilizada sistemas de arquivos paralelos, como o *NPFS*. Uma vez que o meio de transmissão em redes locais é, geralmente, compartilhado, este trabalho visa a criação dos mecanismos para melhorar a utilização do meio de transmissão e, conseqüentemente, aumentar o desempenho das operações de entrada e saída.

Uma vez que os dados são distribuídos entre os servidores, a minimização dos tempos para sua requisição é o maior desafio a ser enfrentado.

Tendo uma rede de dados com meio físico compartilhado, a concorrência no acesso ao meio é um grande desafio, podendo inibir os benefícios da operação em paralelo dos diversos servidores.

Assim, o uso de *caches* pode prover melhoras significativas nos tempos de resposta. Da mesma maneira, a leitura antecipada dos dados, se feita nos momentos de ociosidade da rede, é outra fonte de otimizações.

Diferentes padrões de acesso na leitura dos dados pelas aplicações também podem influir no desempenho da E/S.

Deste modo, o melhor desempenho de um sistema de arquivos paralelos distribuído está vinculado à descoberta do padrão de acesso em uso e à seleção das políticas mais adequadas de *prefetching* e organização de dados em *cache*, bem como à determinação dos momentos apropriados para realizar as transferências entre *clientes* e *servidores*.

O objetivo deste trabalho foi a criação de uma arquitetura de *cache* e *prefetching* independente da arquitetura do sistema de arquivos utilizado; isto é, a mesma arquitetura pode trabalhar com diferentes sistemas de arquivos e uma arquitetura única de *cache* e *prefetching* pode ser

utilizada tanto em *clientes* quanto nos *servidores*.

Para proporcionar uma maior facilidade de integração do sistema de *cache* e *prefetching* proposto neste trabalho com sistemas de arquivos existentes, foi criada uma série de funções intermediárias, cujo objetivo é possibilitar o acesso do sistema do *cache* e *prefetching* aos detalhes internos do sistema de arquivos utilizados. Desta forma, uma maior independência entre os mecanismos de *cache* e *prefetching* e diferentes sistemas de arquivos foi obtida.

A arquitetura interna e o funcionamento do sistema de *cache* e *prefetching* são apresentados a seguir.

5.2 Arquitetura do sistema

A arquitetura do sistema é apresentada na figura 4.

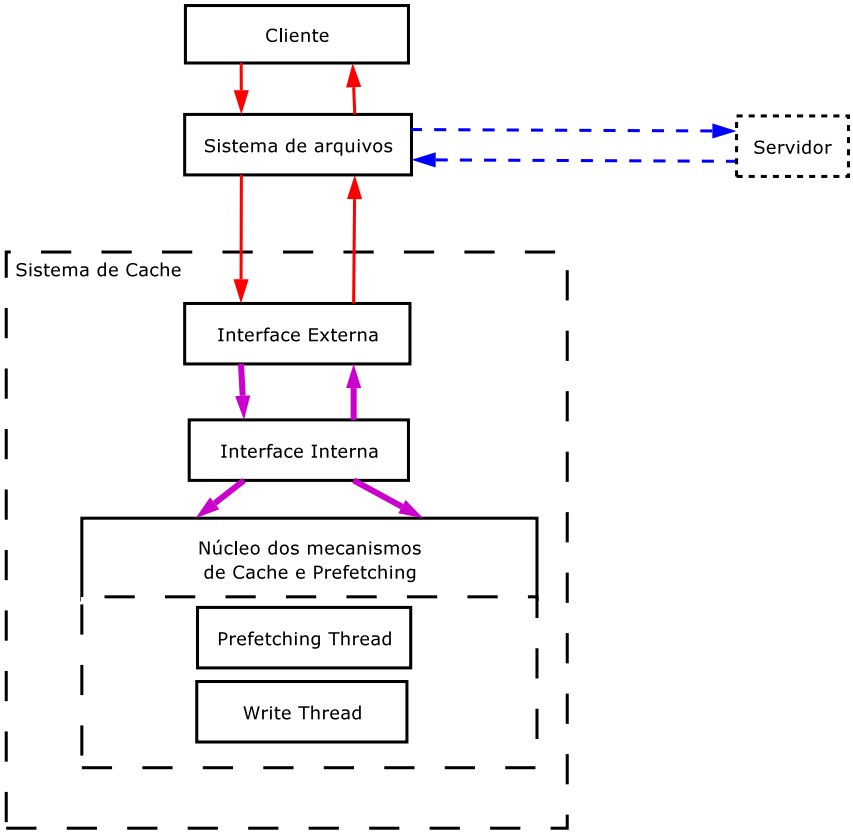

Figura 4: Arquitetura do sistema

Como pode ser visto na figura, 4, a interação do cliente com o sistema ocorre através de uma interface do sistema de arquivos, que por sua vez interage com os mecanismos de *caching* e *prefetching*

O sistema consiste de uma série de *caches* independentes, disponibilizando cada um deles

para cada um dos arquivos, sendo que o espaço de *cache* é dividido em blocos de tamanho variado. Cada *cache* pode possuir sua própria política de funcionamento, independente dos outros *caches* presentes no sistema.

Internamente, o sistema consiste de três módulos, denominados de *interface interna*, *interface externa* e *núcleo*. O núcleo do sistema contém as *threads* responsáveis por operações assíncronas, tais como leitura antecipada (*prefetching*) e escrita atrasada, e a implementação dos algoritmos de *cache* e *prefetching*. A interface interna, por sua vez, é utilizada pelo sistema para ter acesso às estruturas permanentes do *cache*, e a interface externa trata dos parâmetros variáveis do sistema.

A arquitetura modular permite a utilização do mesmo sistema de *cache* e *prefetching* em processos distintos, tais como *clientes* e *servidores*, sem modificações nas operações internas no sistema. Além disto, o acesso a diversos sistemas de arquivos também é facilitado devido à implementação da arquitetura apresentada.

5.2.1 Arquitetura interna

Internamente, o sistema de *cache* e *prefetching* implementado consiste de 3 módulos, sendo que cada um deles pode ser localizado em um arquivo diferente:

- **Interface interna: cache.h** – este arquivo contém todas as definições das estruturas e operações do cache. O arquivo é utilizado internamente pelo sistema de cache.

  Neste módulo, são definidas as estruturas de dados necessárias para o funcionamento correto do *cache*.

- **Núcleo do sistema: cache.c** – este é o arquivo principal do sistema. Ele contém as implementações das funções pertencentes aos mecanismos *cache* e *prefetching*. Devido a utilização de *threads*, este arquivo deve ser compilado e *linkado* com a biblioteca de *threads* do sistema.

- **Interfaces externas: client_cache.h, server_cache.h** – estes arquivos são os arquivos que devem ser utilizados pelas aplicações para obter acesso aos mecanismos de *cache* e *prefetching*. Desta maneira, é possível ter diferentes tipos de processos e sistemas de arquivo utilizando o mesmo sistema simultaneamente, alterando apenas os parâmetros necessários, tais como:

  – Tamanho do *cache*.

  - Tamanho de blocos, em quais o *cache* é dividido internamente.
  - *Delay* da escrita atrasada.
  - Número máximo de arquivos em *cache*.
  - Parâmetros e políticas de prefetching.
  - etc

  Deste modo, é possível configurar os parâmetros que são utilizados por processos distintos sem interferir diretamente na implementação interna de *cache* e *prefetching*.

A integração das duas interfaces em um arquivo somente é possível. Desta maneira, o núcleo do sistema permanece inalterado e a funcionalidade da interface externa é oferecida por um módulo apenas, caso não exista necessidade de uma re-utilização do sistema por diversos processos.

5.2.2 Funcionamento do sistema

O funcionamento do *cache* ocorre de acordo com três algoritmos:

- Algoritmo de leitura de dados. Este algoritmo é utilizado na leitura dos dados propriamente ditos. No caso do *NPFS*, ele é executado pela função *p_read*.

  Este algoritmo é responsável por:

  - Determinação dos blocos atualmente presentes em *cache*.
  - Alocação do espaço necessário para novos blocos, possivelmente removendo blocos antigos do *cache*.
  - Atualização da "relevância" dos blocos presentes em *cache*.
  - Atualização do padrão de acesso para o algoritmo de *prefetching*, levando em questão o número e a seqüência dos blocos lidos pelo processo da leitura.

- Algoritmo de *prefetching*. Este algoritmo determina os possíveis blocos a serem requisitados pela aplicação no futuro.

  Para tanto, o algoritmo analisa o histórico dos acessos da aplicação em questão visando determinar as alteração no padrão de acesso. De acordo com o comportamento do algoritmo de *prefetching*, o mecanismo pode requisitar os blocos logo após o término do processo de leitura (funcionamento síncrono) ou pode esperar até o fim de todos os processos de leitura para requisitar os blocos necessários (funcionamento assíncrono).

- Algoritmo de escrita de blocos. Este algoritmo é executado pela função de escrita de dados (por exemplo, a função *p_write* no caso de *NPFS*). Ele determina os blocos a serem escritos e os marca como “sujos”. De acordo com a política escolhida, o algoritmo pode escrever os blocos instantaneamente (*writethrough*), depois de um certo tempo (*write-back*) ou assim que não houver nenhum bloco disponível em *cache* (*writefull* e *writefree*).

5.2.3 Threads

Com o objetivo de oferecer suporte a mecanismos assíncronos de entrada e saída, sem as possíveis interferências pelas aplicações do usuário, é necessária a criação de algum mecanismo de suporte a operações paralelas.

Durante o trabalho, foram consideradas duas maneiras de oferecer um paralelismo às operações paralelas – a utilização de *processos* [75] e a utilização de mecanismos de *threads* [75]:

- ***Processos*** – esta é primeira idéia utilizada para oferecer a possibilidade de funcionamento simultâneo de diversas aplicações em paralelo.

  A técnica consiste em criação de um outro processo responsável pelas funções executadas em plano de fundo.

  Entretanto, a criação de processos e a comunicação entre eles são um processo relativamente lento [84] devido à necessidade de criação de estruturas de memória completas para cada um dos processos e à necessidade de processamento de mensagens trocadas entre os processos.

- ***Threads*** – a utilização de *threads* [75] possibilita eliminar os maiores problemas de criação de processos separados - velocidade de criação e o desempenho da comunicação [84].

  Visto que as *threads* possuem memória compartilhada, esta pode ser utilizada para a comunicação entre elas, tirando a necessidade da utilização de algum mecanismo de troca de mensagens entre diversas *threads* de um mesmo processo. Porém, é necessário utilizar mecanismos de bloqueio ao acessar as regiões críticas, no caso, o espaço de *cache*.

A utilização de *threads* foi a abordagem escolhida devido às vantagens oferecidas.

Para oferecer suporte a mecanismos assíncronos de entrada e saída foram criadas duas threads, uma *write thread*, responsável pela escrita de dados e uma *prefetching thread*, responsável pela leitura antecipada de dados. O funcionamento delas é apresentado a seguir.

5.2.4 Funcionamento assíncrono

Visando oferecer um paralelismo de operações de entrada e saída, as *threads* são executadas de maneira assíncrona. Entretanto, para não sobrecarregar o meio de transmissão, as operações de escrita de dados e de leitura antecipada ocorrem de uma maneira dirigida pelo sistema de *cache*.

As operações de leitura antecipada assíncrona acontecem nos intervalos entre a leitura de blocos, procurando não sobrepor as operações de leitura efetuadas pela aplicação (através da função *p_read*) com as operações de leitura efetuadas pelo mecanismo de *prefetching*.

Para isso, a *prefetching thread* fica aguardando as instruções da *thread* principal para começar o processo de *prefetching*.

A operação de *prefetching* ocorre quando:

- Algoritmo de prefetching é habilitado.
- Não há nenhuma operação de leitura sobre o determinado arquivo em execução.
- O padrão de acesso da aplicação é determinado.

Desta maneira, o número de possíveis interferências entre mensagens de leitura normal (*p_read*) e mensagens de leitura antecipada é relativamente baixo. Porém, caso ocorra alguma colisão, o mecanismo de *prefetching* não pede a retransmissão do bloco perdido, visando não prejudicar o desempenho geral das operações de entrada e saída. Desta maneira, o mecanismo de *prefetching* procura utilizar a rede de uma maneira eficiente, evitando colisões e sobreposições de mensagens.

Por outro lado, a operação de escrita não depende apenas da taxa de utilização da rede. Os dados mantidos em *cache* devem ser escritos de volta ao disco de acordo com a política de escrita empregada, que pode ser descrita pelos algoritmos *writethrough*, *writeback* ou *writefull*.

Caso o algoritmo *writethrough* seja utilizado, os dados são escritos de volta aos servidores logo depois da execução da função *p_write* pelo cliente. Desta maneira, a operação de escrita ocorre de maneira síncrona.

No algoritmo *writeback*, à cada operação de escrita (*p_write*) a função *p_write* notifica a *write thread* sobre a existências de dados a serem escritos após um período de tempo pré-determinado. Neste algoritmo, a *write thread* utiliza um contador para a escrita atrasada de

dados. Uma vez que os dados a serem escritos permanecem em *cache* por um período de tempo maior, eles são escritos de volta ao disco.

Algoritmo *writefull* efetua a escrita de todos os blocos marcados como “sujos” (isto é, dados que foram modificados e devem ser escritos de volta aos discos) assim que não existe nenhum bloco em *cache* que possa ser utilizado para a entrada de novos dados. Desta maneira, todos os blocos “sujos” são escritos simultaneamente no disco de uma vez.

5.3 Operação do *cache*

5.3.1 Processo de leitura

O esquema de funcionamento do cache é apresentado na figura 5.

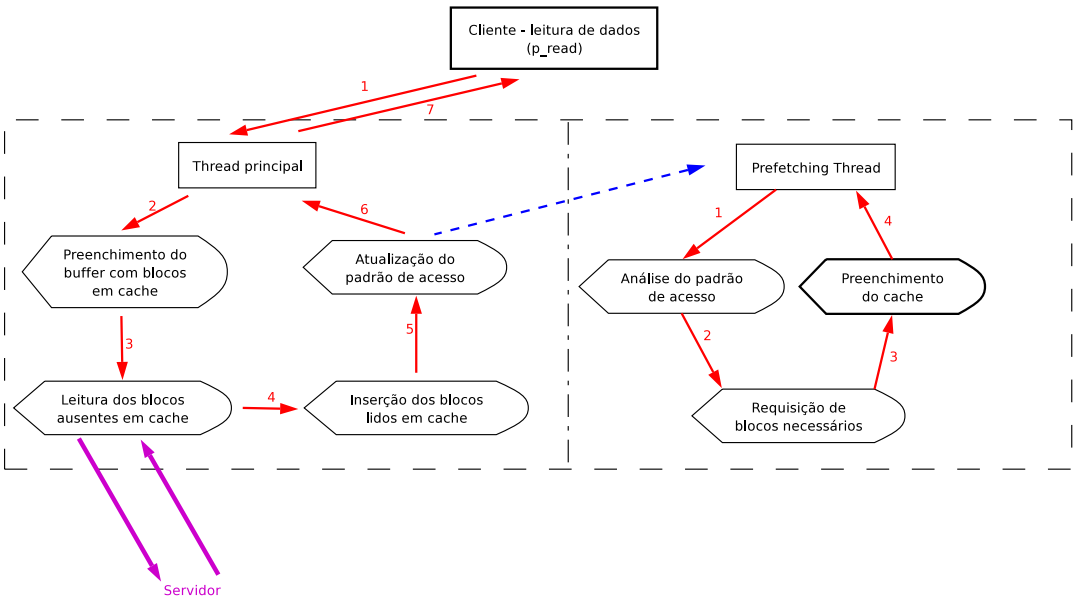

Figura 5: Funcionamento do cache

Como pode ser visto na figura 5, o funcionamento da função de leitura (*p_read* no caso do sistema de arquivos *NPFS*) ocorre da seguinte maneira:

- É determinada a quantidade de blocos a serem lidos durante a operação de leitura, em função da posição atual no arquivo (*offset*) e do tamanho da requisição.
- Uma lista de blocos é montada, contendo todos os blocos que devem ser lidos.
- Para cada bloco da lista, é determinada a presença do mesmo em cache, e as ações necessárias são tomadas de acordo com o esquema mostrado na figura 6.

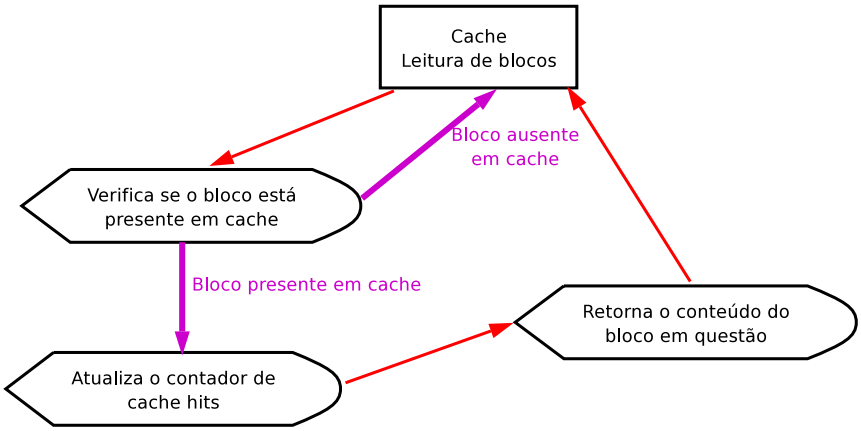

Figura 6: Leitura de blocos

  - Caso o bloco não esteja presente em cache, a função indica a ausência do bloco retornando *NULL*.
  - Caso o bloco esteja presente em cache, a aplicação:
    * Atualiza o número de cache hits para o determinado bloco.
    * Retorna o conteúdo do bloco em questão.
    * Copia o conteúdo do bloco obtido do cache para o *buffer* de leitura, marcando o bloco como lido.

- Todos os blocos marcados como não-lidos são requisitados dos servidores corespondentes e inseridos no cache, liberando espaço necessário através das políticas empregadas, tais como FIFO, LRU, como demonstrado na figura 7.

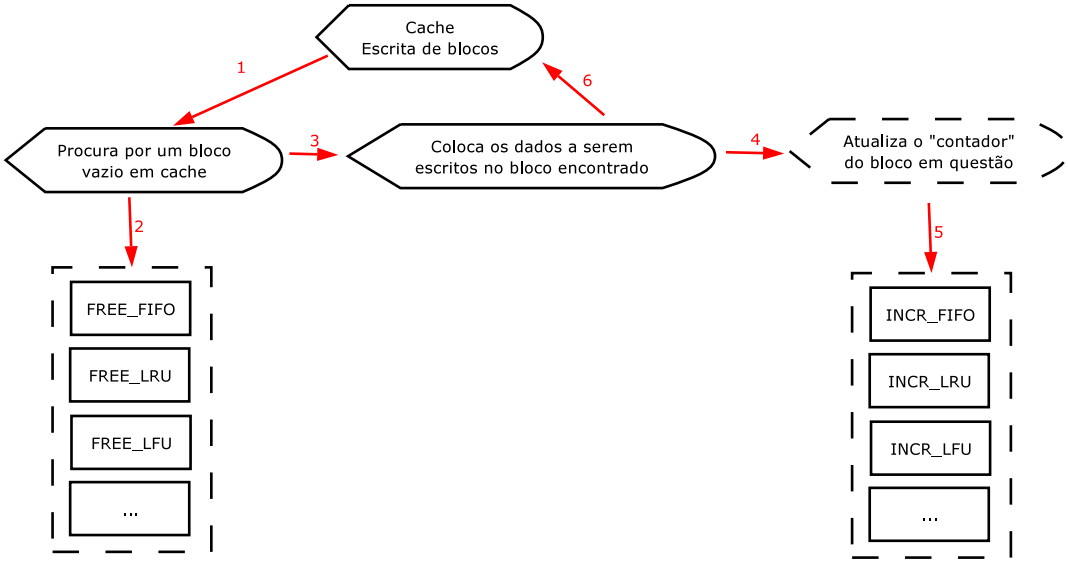

Figura 7: Escrita de blocos

- O padrão de acesso é atualizado.

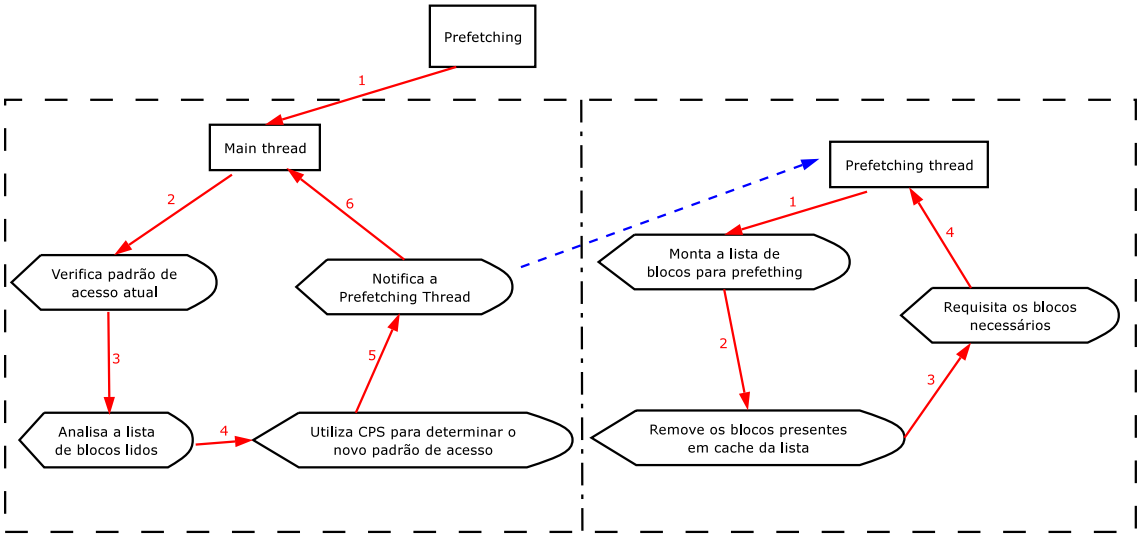

Figura 8: Operações de Prefetching

- Caso o *prefetching síncrono* for habilitado, os mecanismos de leitura antecipada são executados pela *thread principal*. Caso contrário, a *prefetching thread* é notificada sobre a atualização do padrão de acesso do arquivo em questão. A *prefetching thread* funciona de acordo com a figura 8:
  - Analisa o padrão de acesso do arquivo, levando em questão os blocos lidos pelo último processo de leitura.
  - Algoritmo CPS é empregado para determinar o padrão de acesso atual, possivelmente modificando o padrão de acesso empregado.
  - Caso um padrão de acesso conhecido for descoberto, a prefetching thread monta uma lista de blocos a serem lidos e começa a leitura dos mesmos.
  - Visando um desempenho maior das operações, não ocorre retransmissão dos blocos não recebidos devido a algum *timeout* ou colisão ocorrida.
  - O número de blocos a serem lidos e a posição deles é determinada através de algum algoritmo de *prefetching* (*aggressive, limited aggressive, passive* ou *prefetch-on-empty*).
  - Os blocos lidos são inseridos no cache.

5.3.2 Processo de escrita

A escrita de dados pode acontecer de três maneiras, dependendo do algoritmo utilizado, que pode ser *writethrough*, *writeback* ou *writefull*. A alteração entre os algoritmos é feita utilizando a função *cache_fcntl()*, apresentada nas seções a seguir.

O funcionamento destes algoritmos é descrito a seguir:

- **Algoritmo *writethrough*** – utilizando este algoritmo, os dados são escritos nos discos imediatamente, sem nenhum atraso.
- **Algoritmo *writeback*** – caso este algoritmo seja utilizado, os dados são escritos da maneira assíncrona. Durante a chamada à função de escrita, os dados são inseridos em *cache*, e são escritos em disco efetivamente nos intervalos entre requisições. Os blocos a serem escritos são marcados como "sujos" para a escrita posterior.
- **Algoritmo *writefull*** – este algoritmo escreve os dados de volta aos discos assim que o espaço do *cache* estiver cheio; ou seja, algum bloco tem que ser removido de *cache*, e existem blocos marcados como "sujos". Desta maneira, todos os blocos "sujos" são escritos simultaneamente.

A alteração entre os algoritmos pode ser feita durante a execução, utilizando o parâmetro *CACHE_WRITE_OPERATION* da função *cache_fcntl*.

5.3.3 Algoritmos de *prefetching*

Este trabalho introduz dois algoritmos de *prefetching* que procuram manter uma boa utilização do espaço em *cache* junto com uma carga balanceada da rede – *prefetch-on-empty* e *limited aggressive*.

O objetivo destes algoritmos é o balanceamento entre a utilização de rede e o preenchimento do espaço do *cache* com blocos lidos pelo algoritmo de leitura antecipada.

Ao contrário do algoritmo de *prefetching* agressivo, que procura efetuar a leitura antecipada de dados sempre que for possível, e do *prefetching* passivo, cujo objetivo é obter apenas os blocos necessários para a próxima operação de leitura, os algoritmos propostos procuram balancear as operações de leitura antecipada. Desta forma, enquanto o algoritmo de *prefetching* agressivo procura preencher todo o espaço de *cache* com blocos a serem lidos, e o algoritmo de *prefetching* passivo procura manter os blocos já presentes em *cache* o maior tempo possível, os algoritmos propostos apresentam o seguinte comportamento:

- ***Prefetch-on-empty*** – este algoritmo procura minimizar os períodos ociosos da rede, efetuando as operações de *prefetching*. O comportamento dele é descrito pelo algoritmo a seguir:

  - Se os próximos blocos a serem lidos pelo algoritmo de *prefetching* estão presentes em *cache*, o algoritmo não requisita blocos adicionais.
  - Se os próximos blocos não estão presentes em *cache*, o algoritmo requisita os blocos e entra em *modo de requisição de blocos*.
  - Em *modo de requisição de blocos*, a cada execução do algoritmo o mesmo requisita os blocos restantes até que o número de blocos já requisitados ultrapasse um certo valor (denominado de *prefetching lookahead*). Se o número de blocos ultrapassar o *prefetching lookahead*, o algoritmo volta a operar em *modo normal*.

  Desta maneira, o *prefetch-on-empty* procura intercalar as fases de acesso intensivo ao meio de comunicação (enquanto em modo de *requisição de blocos*) e fases de atividade relativamente baixa (modo normal).

- ***Limited aggressive*** – a idéia deste algoritmo é a possibilidade de juntar a eficiência das estratégias agressivas de *prefetching* com utilização eficiente do espaço em *cache*.

  O algoritmo comporta-se da maneira similar ao algoritmo *prefetch-on-empty*, com a diferença de que, quando o número de blocos já requisitados ultrapassa o valor do *prefetching lookahead*, o algoritmo continua requisitando os blocos, mantendo sempre um número de blocos igual ao valor do *prefetching lookahead* para possíveis acessos futuros.

Além destes algoritmos, o sistema de *cache* e *prefetching* suporta algoritmos de *prefetching passivo* e *prefetching agressivo*.

5.3.4 Algoritmo CPS

Visando a integração dos mecanismos de *cache* e *prefetching*, foi proposta a criação do algoritmo de *Contagem da Probabilidade de Sucesso (CPS)*, que é um algoritmo utilizado para efetuar a decisão das próximas operações de leitura antecipada. O algoritmo proposto é bastante simples e o seu objetivo é possibilitar a escolha transparente do padrão de acesso utilizado.

O algoritmo funciona da seguinte maneira:

- Quando o cliente abre o arquivo, é iniciado um contador, uma variável local do cliente, denominada de contador de probabilidade de sucesso (CPS).
- Após a primeira requisição de leitura de arquivo, o algoritmo de *prefetching* é iniciado. Ele analisa o contador de probabilidade de sucesso para determinar o tipo da requisição.

Se o contador for maior do que um certo valor, que é definido em tempo de compilação ou durante a execução, é enviada uma requisição de carregamento do próximo bloco de dados no *cache*, para ser lido posteriormente. Os dados carregados no *cache* podem não ser utilizados imediatamente, uma vez que o contador de probabilidade de sucesso ainda não é grande o suficiente.

- Para cada requisição de leitura seguinte, o contador de probabilidade de sucesso é analisado, para verificar se o mecanismo de *prefetching* conseguiu prever os próximos blocos a serem lidos. Para cada sucesso de *prefetching*, o contador de probabilidade de sucesso é incrementado; para cada falha, ele é decrementado.

- O algoritmo de *prefetching* avalia o número de sucessos e falhas, para determinar um algoritmo mais adequado (i.e., leitura seqüencial do arquivo, leitura de alguns blocos na ordem certa, etc.). A cada mudança do algoritmo de *prefetching*, o contador de probabilidade de sucesso é zerado.

- Quando o algoritmo de *prefetching* perceber que o contador de probabilidade de sucesso é suficientemente grande, ele começa a leitura de dados propriamente dita, carregando-os em *cache* para os acessos posteriores.

- Se o algoritmo de *prefetching* descobrir que o número de falhas é grande, ou seja, se ele perceber que o algoritmo de *prefetching* utilizado não é o mais adequado, ele termina de enviar as requisições de leitura de dados e volta a analisar o novo padrão de acesso. Assim, ele começa a utilizar o mecanismo *prefetching* dos dados efetivamente, procurando evitar leituras antecipadas desnecessárias.

A determinação do padrão de acesso atual é feita analisando a seqüência dos blocos requisitados anteriormente. Assim, é possível utilizar o mecanismo de *prefetching* diminuindo a taxa de requisição de blocos não utilizados, uma vez que o algoritmo envia os pedidos de leitura de próximos blocos de dados apenas quando ele acredita que esses dados são realmente aqueles que serão consumidos em seguida.

Uma vez que os padrões de acesso são tratados de maneira independente para cada cliente, o compartilhamento de um arquivo não altera a identificação do padrão em uso.

Para oferecer maior suporte aos diferentes padrões de acesso e algoritmos de *prefetching*, o algoritmo CPS pode iniciar o contador de probabilidade de sucesso com um valor diferente de zero. Isto facilita a escolha do algoritmo de *prefetching* quando o padrão de acesso é conhecido

ou especificado pelo usuário. Assim, o algoritmo inicia a utilização dos algoritmos de prefetching para efetuar a leitura antecipada dos dados logo depois de ser iniciado, sem uma análise prévia do padrão de acesso.

Para se adaptar mais rapidamente aos padrões de acesso das aplicações, o valor do contador de probabilidade de sucesso pode ser incrementado ou decrementado exponencialmente, ou com valores especificados pelo usuário. Isto possibilita uma escolha mais balanceada entre *prefetching agressivo*, quando o contador de probabilidade de sucesso é incrementado muito rápido, e *prefetching passivo*, quando a velocidade de decremento do contador a cada falha é maior que o valor de incremento.

Desta maneira, é possível especificar o comportamento do algoritmo CPS sem utilizar muitas variáveis e, mesmo assim, é possível definir o funcionamento do mecanismo de *prefetching*.

Uma vez que o algoritmo CPS proposto tem como o foco principal os padrões de acesso sequencias, a utilização de outros algoritmos de determinação de padrão de acesso especializados é prevista.

5.4 *Funcionamento e utilização do sistema*

O sistema de *cache* e *prefetching* consiste de três *módulos*: módulo de acesso externo, que é utilizado pelas aplicações para obter acesso aos mecanismos de *cache* e *prefetching*; módulo interno, que contém as definições das estruturas e funções do sistema, e de núcleo, que contém as implementações de funções utilizadas. Além disto, o sistema de *cache* e *prefetching* precisa de algumas funções que devem ser implementadas pelos clientes.

Nesta seção, a descrição e o funcionamento da *API* do sistema são apresentados.

5.4.1 Módulo de acesso externo

Este módulo consiste de um arquivo *.h*, que contém os valores-padrão das variáveis e estruturas de dados utilizadas no mecanismo de *cache* e *prefetching*.

Os principais valores que podem ser modificados no módulo externo são:

- **BITMAP_BLOCK_SIZE** – tamanho padrão de um bloco de *cache*.
- **DEFAULT_LOCAL_CACHE_SIZE** – tamanho padrão do *cache*, especificado em número de blocos.

- **NUM_CACHES** – número de *caches* que podem existir no sistema simultaneamente.
- **Algoritmo de *Prefetching*** – valores padrão do mecanismo de *prefetching*, que especificam a profundidade de análise dos acessos passados, número de blocos a serem lidos por cada chamada ao mecanismo de *prefetching*, algoritmo de *prefetching* utilizado, entre outros.
- **Algoritmo *CPS*** – valores que definem o comportamento do algoritmo *CPS*.

Além deste valores, é possível especificar detalhadamente o funcionamento do *cache* através de variáveis secundárias.

5.4.2 Módulo interno

Este módulo contém as definições das estruturas internas, responsáveis pela representação dos dados para os mecanismos de *cache* e *prefetching*.

As seguintes estruturas são representadas neste módulo:

- Definição de blocos.

  Os blocos são representados internamente por uma estrutura do tipo *blocks_type*.

  Dentro desta estrutura, são definidos:

  – Um apontador para a lista de blocos presentes em *cache*.

  – Número de blocos existentes.

  – Tamanho total do arquivo.

  Estes dados são utilizados na determinação de número de blocos necessários para um arquivo durante a criação e inicialização do espaço de *cache*.

- *Bitmap* para acesso rápido aos blocos presentes em *cache*.

  Buscando otimizar a velocidade de acesso ao espaço de *cache*, é criado um *bitmap* do tipo *bitmap_type*, contendo os seguintes dados:

  – Mapa de bytes representando os blocos existentes. O conteúdo dos blocos, informando se os mesmos são blocos vazios ou utilizados é representado por uma variável inteira.

Este *bitmap* é utilizado para determinar se um determinado bloco é vazio ou utilizado. Caso o bloco for utilizado (isto é, o valor da mapa de bytes para este bloco é maior que zero), o valor da mapa de bytes nesta posição indica a *relevância* deste bloco.

O valor da relevância para cada entrada no *bitmap* é atualizado de acordo com a política utilizada: *FIFO, LRU, LFU*...

- Lista de blocos "sujos" – isto é, blocos que foram alterados durante o processo de escrita e devem ser escritos de volta aos servidores.
- Tamanho dos blocos. O *cache*, mesmo tendo tamanhos fixos para cada bloco, pode possuir blocos que não utilizam por completo o espaço reservado, principalmente no caso dos arquivos pequenos ou de blocos localizados no fim do arquivo.
- Lista de referências entre o *bitmap* e a posição física dos blocos correspondentes no espaço de *cache*. Esta lista é utilizada para acessar rapidamente os dados de cada bloco sem precisar percorrer todo o espaço do *cache*.

- Funções responsáveis pela operação sobre os blocos.

  Esta estrutura, denominada *data_operations*, contém referências para as funções responsáveis pelas operações sobre os blocos presentes em *cache*.

  As funções são:

  - **free_space** – esta função é responsável por liberação de espaço em *cache* para um ou mais blocos. Ela é responsável pela análise da *relevância* dos blocos mantidos em *cache* e eventual retirada dos mesmos do *cache*
  - **init_counter** – esta função inicializa o valor de relevância dos blocos em *cache*.
  - **incr_counter** – finalmente, esta é a função responsável pelo aumento do valor de relevância dos blocos em *cache*. Esta função é executada para cada bloco encontrado em *cache* (isto é, na ocorrência de um *cache hit*).

  Estas funções são definidas de acordo com a política utilizada (*FIFO, LRU, LFU*...) e podem ser re-definidas durante a execução.

- Estatísticas do *cache*.

  Esta estrutura, denominada *cache_stats*, é responsável pela coleta de dados estatísticos sobre o funcionamento do *cache*, tais como:

  - Número de *cache hits*, isto é, número de blocos requisitados do *cache*.

  - Número de *cache misses* – número de blocos que foram requisitados em *cache* porém não foram encontrados em *cache*.
  - Número total de acessos ao *cache*.
  - Padrão de acesso atual, utilizado pelo mecanismo de *prefetching*.
  - Histórico de acessos passados, representado por uma lista de blocos, que é analisada visando determinar o padrão de acesso atual.
  - Intervalo entre os blocos. Este número representa intervalo entre blocos lidos. Por exemplo, caso uma operação de leitura requisitou os blocos *1, 3, 5, 7...*, este valor é igual a 2. Normalmente, este valor é igual a 1 (isto é, leitura seqüencial).
  - Valor do algoritmo *CPS*. Este valor é utilizado para determinar o funcionamento atual do algoritmo de *prefetching*.
  - Lista de blocos a ser requisitada durante a próxima execução do mecanismo de *prefetching*. Esta lista é montada de acordo com o histórico de acessos, padrão de acesso atual, intervalo entre blocos lidos e valor do algoritmo *CPS*.

- Definição das estruturas necessárias para o funcionamento das *threads*.

  Para as *threads* utilizadas no mecanismo de *cache* e *prefetching*, são alocadas os seguintes dados, descritos pela estrutura *cache_threads_type*:

  - *PID* da *threads*, identificando unicamente cada uma das *threads* existentes.
  - Semáforos e bloqueios de acesso às regiões críticas.
  - *Flags* de bloqueio, indicando se uma operação de *prefetching* (no caso da *prefetching thread*) ou de escrita (no caso da *write thread*) deve ser efetuada.
  - Valor de tempo, em segundos, especificando o atraso na execução periódica da *thread*. Este valor é utilizado para executar automaticamente a *write thread* efetuando a escrita atrasada, caso a política *writeback* for empregada.

  As *threads* são criadas na inicialização dos mecanismos de *cache* e *prefetching* com os parâmetros pré-definidos. Entretanto, a alteração destes parâmetros durante a execução é prevista.

- Definição da estrutura do *cache* em si.

  O espaço de *cache* é representado internamente através de uma estrutura do tipo *cache_type*. Dentro dela, são definidos os seguintes parâmetros do *cache*:

  - Espaço do *cache* propriamente dito.

    O espaço de *cache* é representado através de uma região de memória alocada. O tamanho da memória alocada é determinado em função de:

    * Número de blocos em *cache*.
    * Tamanho de cada bloco.

    Desta maneira, se o *cache* for configurado para utilizar *100* blocos, cada um com tamanho de *16Kb*, o tamanho total do espaço do *cache* é igual a $100 * 16Kb = 1.6Mb$.

    O tamanho do espaço do *cache* pode ser alterado durante a execução, sem necessidade de recompilação ou finalização da aplicação.

  - *Bitmap* representando dados em *cache*, definido por estrutura *bitmap_type* e descrito anteriormente.
  - Lista de blocos propriamente ditos, representados por estrutura *blocks_type*, descrita anteriormente.
  - Dados estatísticos do funcionamento do *cache*, definidos na estrutura *cache_stats* descrita anteriormente.
  - Status *atual* da leitura. Este *flag* indica se algum processo de leitura está em andamento atualmente. Caso este flag esteja habilitado, o processo de *prefetching* não é executado.
  - Estado atual do *cache*, indicando se o *cache* é inicializado, habilitado ou desabilitado.
  - Parâmetros do *cache*, representados pela estrutura *cache_params*. Esta estrutura contém todos os valores que podem ser alterados pelo *módulo externo*, descrito acima.
  - Referência para as funções de gerenciamento de blocos, descritas pela estrutura *data_operations*, apresentada acima.
  - Flag *need_prefetching*, indicando se a execução do mecanismo de *prefetching* é necessária.
  - Flag *need_write*, indicando se escrita imediata de dados é necessária.

- *Visões* sobre o(s) *cache(s)* presentes no sistema.

  Esta estrutura é uma lista ligada que representa todos os *caches* existentes e habilitados no sistema.

- Parâmetros da função de controle e manutenção do *cache*, *cache_fcntl*. A função *cache_fcntl* é responsável pela alteração dos parâmetros do *cache*. Esta função aceita três parâmetros:
  - Identificador do *cache* cujo parâmetro deve se alterado.
  - Parâmetro a ser alterado, que pode ser um dos seguintes:
    - ***CACHE_USE_CACHE*** – habilita ou desabilita o sistema do cache. O valor deste parâmetro e de todos os parâmetros lógicos (*booleanos*) pode ser 0 (parâmetro desabilitado) ou 1 (parâmetro habilitado).
    - ***CACHE_CACHE_SIZE*** – altera o tamanho do *cache*. O valor deste parâmetro é o novo tamanho do *cache*, em blocos, sendo que tamanho total do *cache* depende do tamanho do bloco.
    - ***CACHE_CACHE_POLICY*** – determina a política do *cache*. O valor deste parâmetro por ser:
      - *POLICY_FIFO* – determina a utilização da política *first-in forst-out*.
      - *POLICY_LRU* – determina a utilização da política *least recently used*.
      - *POLICY_LFU* – determina a utilização da política *least frequently used*.
    - ***CACHE_CPS_STEP*** – determina o valor do *incremento* do algoritmo *CPS*, descrito acima.
    - ***CACHE_CPS_MAX*** – determina o limite máximo do algoritmo *CPS*.
    - ***CACHE_AP_SUCCESS_COUNTER*** – determina o contador de sucessos necessários para determinar um padrão de acesso.
    - ***CACHE_AP_COUNTER*** – determina número total de blocos a serem analisados para determinar o padrão de acesso.
    - ***CACHE_PREFETCH_COUNTER*** – contador de blocos a serem requisitados pelo algoritmo de *prefetching*.
    - ***CACHE_USE_PREFETCHING*** – habilita ou desabilita a utilização da leitura antecipada (*prefetching*).
    - ***CACHE_PREFETCH_ASYNC*** – habilita ou desabilita a utilização de *prefetching* assíncrono (isto é, habilita ou desabilita a utilização da *prefetching thread*).
    - ***CACHE_AUTO_PREFETCHING*** – habilita ou desabilita utilização automática do *prefetching*. Caso o *prefetching* automático for desabilitado, a aplicação do usuário deverá chamar manualmente o mecanismo de *prefetching*.

    - ***CACHE_PREFETCH_OPER*** – determina o algoritmo do *prefetching*. O valor deste parâmetro pode ser:
        - ***CACHE_PREFETCH_AGGR*** – habilita o algoritmo *limited aggressive*.
        - ***CACHE_PREFETCH_PASS*** – habilita o algoritmo passívo.
        - ***CACHE_PREFETCH_ON_EMPTY*** – habilita o algoritmo *prefetch-on-empty*.
        - ***CACHE_PREFETCH_AGGR_NL*** – habilita o algoritmo de *prefetching* agressivo não limitado.
    - ***CACHE_PREFETCH_LOOKAHEAD*** – determina o *look-ahead* do algoritmo de *prefetching*, isto é, número de blocos que se deve buscar antecipadamente, utilizado nos algoritmos *limited aggressive* e *prefetch-on-empty*.
    - ***CACHE_WRITE_OPERATION*** – especifica o algoritmo de escrita a ser utilizado, através dos parâmetros *CACHE_WRITE_WRITETHROUGH*, *CACHE_WRITE_WRITEBACK* e *CACHE_WRITE_WRITEFULL*.

  Estes parâmetros possibilitam a alteração do funcionamento do sistema durante a execução.

- Códigos de retorno das funções:
    - ***STATUS_OK*** – finalização normal da operação.
    - ***STATUS_ERROR*** – erro durante a execução da operação.
    - ***STATUS_REINIT*** – tentativa de inicialização do mecanismo de *cache* previamente inicializado.
    - ***STATUS_OOM*** – memória insuficiente.
    - ***STATUS_FULL*** – não há espaço suficiente em *cache* para adicionar dados.
    - ***STATUS_DISABLED*** – tentativa de operação com *cache* desabilitado.
    - ***STATUS_BADOPT*** – parâmetro desconhecido.
    - ***STATUS_NO_PREFETCH*** – tentativa de operação com *prefetching* desabilitado.

Além disto, neste módulo são definidos os protótipos das funções implementadas no núcleo do sistema e das funções externas, que serão apresentadas a seguir.

5.4.3 Núcleo do sistema

Este módulo contém a implementação das funções responsáveis pelo funcionamento e manutenção do *cache*. Estas funções são divididas nos seguintes sub-módulos:

- **Inicialização do sistema** – este sub-módulo contém duas funções, *CacheInit* e *CacheSetup*:
  - ***CacheInit*** – esta função é responsável pela inicialização do *cache* em si. Ela é executada uma vez só, durante a inicialização do sistema. No caso do sistema de arquivos *NPFS*, ela é executada dentro da função *ParioInit*.

    Esta função cria as *threads*, aloca a memória necessária para as estruturas de dados, inicializa os bitmaps e blocos do sistema e atribui os valores definidos pelo módulo externo aos parâmetros do *cache*.

    Caso tudo der certo, a função retorna código de retorno *STATUS_OK*, caso contrário, ela retorna o código de erro correspondente.
  - ***CacheSetup*** – esta função é responsável pela inicialização de *caches* individuais. Ela calcula os tamanhos dos blocos, número de blocos necessários, baseando-se no tamanho do arquivo. Além disto, a função ajusta o tamanho de memória necessária para as estruturas levando em questão os possíveis ajustes feitos pela função *cache_fcntl*.

    A função retorna *STATUS_OK* se tudo der certo, *STATUS_REINIT* se o *cache* foi inicializado previamente ou *STATUS_OOM* caso não há memória suficiente no sistema para a habilitação do *cache*.

- **Manipulação dos blocos em *cache*** – este sub-módulo contém as funções responsáveis pelo tratamento dos blocos propriamente ditos. Ele contém as seguintes funções:
  - ***put_block*** – esta função coloca dados especificados pelo usuário em *cache*. Se for necessário, o espaço em *cache* é liberado, escrevendo os blocos "sujos" e removendo os blocos menos relevantes.

    Para o bloco em questão, o índice de relevância do mesmo é inicializado.
  - ***get_block*** – esta função retorna o conteúdo do bloco requisitado, incrementando o índice de relevância do mesmo.
  - Funções de manutenção – *get_block_status*, *get_block_size* e *mark_dirty*. Estas funções são utilizadas para consultar ou alterar os parâmetros dos blocos individuais,

tais como a presença do bloco em *cache*, tamanho do bloco e marcação de um bloco como “sujo”, para ser escrito durante a operação de escrita.

- **Implementação de *threads*** – este sub-módulo define a operação das *threads* utilizadas – *write thread* e *prefetching thread*, cujo funcionamento foi descrito anteriormente.

- **Leitura antecipada (*prefetching*) e escrita de blocos** – este sub-módulo implementa o funcionamento dos mecanismos de *prefetching* e de escrita de blocos. As seguintes funções são incluídas neste sub-módulo:

  – *update_pattern* – esta função analisa o histórico dos últimos acessos buscando determinar o padrão de acesso atual, atualizando ele se for necessário. Esta função monta uma lista de blocos a serem lidos pelo mecanismo de *prefetching*.

  – *prefetch* – esta função requisita a lista dos blocos montada previamente pela função *update_pattern*.

  Os blocos são lidos através de uma função externa *request_blocks*, que será descrita posteriormente.

  – *do_write* – esta função efetua a escrita dos blocos propriamente ditos aos servidores correspondentes.

  Os blocos são escritos por uma função externa *write_blocks*, que será descrita posteriormente.

- **Funções estatísticas** – este sub-módulo contém as funções responsáveis pelo oferecimento de informações sobre o estado atual do *cache* para o usuário.

  As funções contidas neste módulo são: *cache_get_hits*, *cache_get_misses*, *cache_get_reqs*, *cache_get_ap*, *cache_get_prefetch_count*, *cache_get_cps* e *cache_get_auto_prefetch*. Estas funções retornam para o usuário os parâmetros correspondentes, descritos anteriormente.

- **Sub-módulo de *algoritmos*** – este sub-módulo contém a implementação dos algoritmos de *cache*, tais como *FIFO*, *LRU*, *LFU*, entre outros. As funções definidas são *free_fifo*, *incr_fifo*, *init_fifo* e assim por diante. O funcionamento destas funções foi descrito anteriormente.

- **Funções globais de *cache*** – este sub-módulo contém as funções que alteram o comportamento global dos mecanismos de *cache* e *prefetching*. As funções definidas são:

– ***CachePurge*** – este função desaloca o espaço de *cache* em questão, liberando a memória alocada e voltando todos os parâmetros alterados para os valores originais. Todo o conteúdo do *cache* liberado é desalocado.

– ***cache_fcntl*** – esta é a função que permite alterar os parâmetros de funcionamento dos mecanismos de *cache* e *prefetching* durante a execução. O comportamento desta função foi descrito anteriormente.

5.4.4 Funções externas

Para possibilitar um funcionamento independente do sistema de arquivos utilizado, é necessário definir uma série de funções genéricas que possibilitam acesso transparente a diversos sistemas de arquivos.

Estas funções devem ser alteradas de acordo com a necessidade do sistema de arquivos em questão, sendo que o protótipo das mesmas deve permanecer inalterado.

As seguintes funções externas são necessárias para o funcionamento do mecanismo de *cache* e *prefetching*:

- ***get_file_size*** – esta função é utilizada para determinar o tamanho do arquivo em questão.

  Único argumento recebido é o identificador do arquivo.

  O valor de retorno desta função é o tamanho do arquivo em bytes.

- ***get_file_block_size*** – esta função é utilizada para determinar o tamanho de blocos do arquivo em questão. No caso de sistemas de arquivos paralelos, o tamanho de bloco é normalmente determinado em função de *striping unit*; nos sistemas de arquivos locais o tamanho de cada bloco pode ser determinado em função do tamanho físico dos blocos no disco ou em função do tamanho dos blocos do sistema de arquivos físico. O valor de cada bloco será utilizado para representar o tamanho do espaço do *cache*.

  Único parâmetro recebido por esta função é o identificador do arquivo.

  O valor de retorno é o tamanho de bloco a ser utilizado em bytes.

- ***request_blocks*** – esta função é utilizada pelo mecanismo de *prefetching* para efetuar a leitura antecipada de blocos. Esta função é chamada pela *prefetching thread* para efetuar a leitura antecipada dos blocos propriamente dita.

  Os parâmetros passados para a função são:

  - Identificador do arquivo.
  - Uma lista de blocos a ser lida pelo mecanismo de *prefetching*. O conteúdo dos blocos nesta lista deve ser lido e adicionado em *cache* através da função *put_block*.
  - Número de blocos a serem lidos de uma vez.

  Os valores de retorno desta função são *0* caso todos os blocos forem lidos com sucesso ou *-1* se algum bloco foi omitido.

- ***write_blocks*** – esta função exerce trabalho oposto ao da função descrita anteriormente – ela escreve os blocos mencionados de volta aos seus respectivos dispositivos de armazenamento.

  Os parâmetros desta função são idênticos aos da função anterior, com a diferença de que os blocos mencionados devem ser escritos nos respectivos lugares ao invés de lidos.

  Nenhum valor de retorno é esperado desta função.

6 *Resultados e discussões*

Com o objetivo de avaliar a eficiência dos mecanismos de *cache* e *prefetching* implementados, testes foram realizados com diferentes aplicações com os seguintes padrões de acesso de dados:

- **Transferência linear de dados**, visando determinar a influência dos mecanismos de *caching* e *prefetching* na leitura seqüencial de dados sem intervalos entre as requisições.
- **Transferência de dados com pequenos intervalos entre requisições.** O objetivo deste teste foi a verificação da influência do tempo decorrido entre as requisições consecutivas na operação dos mecanismos de *caching* e *prefetching*.

  Este tipo de aplicação é caracterizado pela necessidade de troca constante de informações entre os elementos do sistema.
- **Transferência de dados com grandes intervalos entre as requisições.**. Neste tipo de aplicações, os intervalos entre as requisições consecutiva são muito superiores ao tempo decorrido nas requisições propriamente ditas.

  Este tipo de aplicação não apresenta necessidade de interação constante entre os elementos do sistema. Isto é, os elementos do sistema não possuem necessidade de uma troca constante de informações entre eles.

Os padrões de transferência foram avaliados usando as seguintes aplicações:

- **NPFSTOOLS** – *NPFSTOOLS* é um conjunto de utilitários utilizados para transferir arquivos entre o sistema de arquivos *NPFS* e o sistema de arquivos local. *NPFSTOOLS* consiste atualmente de dois aplicativos – *npfsget*, utilizado para transferir arquivos do *NPFS* para o sistema de arquivos local, e *npfsput*, utilizado para armazenar arquivos locais no *NPFS*.

  Estes aplicativos foram utilizados para medir o desempenho dos mecanismos de *cache* e *prefetching* propostos, e para avaliar a eficiência dos algoritmos de *prefetching*.

- **PHTTPD** – servidor de arquivos multimídia *on demand*. O servidor opera sobre o protocolo *HTTP/1.1* [85] de maneira similar aos servidores *web* convencionais, tais como *Apache* [86] e *Tiny Httpd* [87]. Os arquivos multimídia utilizados nos testes foram codificados nos formatos *mp3* [88] e *ogg vorbis* [89].

  Este aplicativo foi utilizado como um exemplo de servidor com pequenos intervalos entre a execução das operações de entrada e saída.

- **PSHOUTCAST** – servidor de *streams shoutcast* [90]. Utiliza o servidor *icecast* [91]. Os arquivos multimídia utilizados foram codificados utilizando o formato *mp3* e *xvid/divx* [92].

  Este aplicativo foi utilizado como um exemplo de servidor com atrasos significativos entre operações de entrada e saída consecutivas.

As seções a seguir discutem a influência do mecanismo de *prefetching* empregado na utilização do espaço de *cache*, a influência do meio de comunicação na latência das requisições e as vantagens da utilização dos mecanismos propostos nas aplicações específicas.

6.1 Estudo de caso: NPFSTOOLS

Este teste procurou determinar a influência dos mecanismos de *caching* e *prefetching* nas operações de leitura de dados que buscam obter a taxa máxima de transferência de dados num fluxo contínuo.

Isto é comum principalmente nas aplicações de acesso seqüencial aos dados, tais como servidores de arquivos em rede local. Neste caso, como não há reutilização dos dados, o uso do *cache* não é adequado. Havendo tempo para *prefetching* entre as requisições, pode haver uma melhora no tempo de resposta; caso contrário, *prefetching* e primitivas de leitura de dados competem pelo acesso aos servidores, prejudicando o tempo de acesso.

6.1.1 Resultados esperados

Variando-se a operação dos mecanismos de *cache* e *prefetching*, esperava-se obter os seguintes resultados nos testes realizados:

- ***cache: off, prefetching: off*** – devido à não reutilização dos dados já lidos e ao pequeno intervalo entre requisições, esperava-se obter o melhor desempenho neste teste, uma vez

que ele não utiliza os mecanismos de *cache* e *prefetching*. Desta maneira, como o teste em questão simplesmente transfere os dados, o mecanismo de *prefetching* não conseguiria executar as suas operações no tempo previsto, introduzindo atrasos e colisões das mensagens.

- ***cache: on, prefetching: off*** – esperava-se obter resultados um pouco inferiores aos do teste anterior devido à necessidade de uma cópia extra de dados na memória. Entretanto, desta maneira é possível disponibilizar os dados para acessos futuros das aplicações, uma vez que a necessidade de acessos adicionais ao dispositivo de armazenamento é eliminada.

- ***cache: on, prefetching: synchronous*** – esperava-se obter um desempenho um pouco inferior devido à necessidade de adaptação do mecanismo de *prefetching* ao arquivo lido e pela transferência extra de dados. *Prefetching síncrono* não utiliza a *prefetching thread*, multiplexando as requisições aos dados com as requisições de leitura antecipada na aplicação principal. Desta maneira, o número de possíveis colisões é reduzido devido à natureza dos acessos. Entretanto, o funcionamento síncrono não oferece um paralelismo equivalente ao *prefetching* assíncrono.

- ***cache: on prefetching: assynchronous*** – esperava-se obter resultados melhores do que com o *prefetching síncrono*, uma vez que os dados são requisitados de maneira independente. Além disto, esperava-se obter um número de *timeouts* e colisões maior devido a transferência assíncrona de dados. Entretanto, este mecanismo explora o paralelismo das operações de *E/S* de maneira mais efetiva.

A influência das taxas de transferência dos discos dos servidores e dos valores utilizados para os *timeouts* das requisições são discutidos em [63] e fogem do escopo deste trabalho.

A diferença entre o funcionamento do mecanismo de *prefetching* de maneira síncrona e assíncrona é discutida na seção 6.4.2.

Os testes foram executados em um ambiente distribuído (*cluster* composto por 4 nós, todos com processador *pentium III dual 1GHz*, 256Mb de memória e disco *SCSI*), sendo que cada teste foi executado três vezes, utilizando a média dos resultados para fins de comparação. Os testes foram feitos utilizando *fast ethernet* e *gigabit ethernet* como o meio de comunicação.

6.1.2 Resultados obtidos

Os resultados obtidos utilizando a rede *fast ethernet* são apresentados na tabela 1. Neste teste, um arquivo de 500Mb foi transferido do sistema de arquivos *NPFS*.

Tipo do teste	**Tempo utilizado**	**Utilização do cache**
cache: off, prefetching: off	72.847 segundos	—
cache: on, prefetching: off	67.879 segundos	—
cache: on, prefetching: sync, passive	97.453 segundos	87.82%
cache: on, prefetching: sync, aggressive	159.616 segundos	99.97%
cache: on, prefetching: sync, on-empty	107.096 segundos	99.93%
cache: on, prefetching: sync, limited aggressive	108.853 segundos	99.97%
cache: on, prefetching: async, passive	104.053 segundos	87.82%
cache: on, prefetching: async, aggressive	169.116 segundos	99.97%
cache: on, prefetching: async, on-empty	114.822 segundos	99.74%
cache: on, prefetching: async, limited aggressive	115.499 segundos	99.97%

Tabela 1: Transferência linear, *fast ethernet, 500Mb*

Tipo do teste	**Tempo utilizado**	**Utilização do cache**
cache: off, prefetching: off	32.56 segundos	—
cache: on, prefetching: off	44.63 segundos	—
cache: on, prefetching: sync, passive	37.87 segundos	87.82%
cache: on, prefetching: sync, aggressive	79.42 segundos	99.97%
cache: on, prefetching: sync, on-empty	47.56 segundos	99.93%
cache: on, prefetching: sync, limited aggressive	53.04 segundos	99.97%
cache: on, prefetching: async, passive	46.69 segundos	87.82%
cache: on, prefetching: async, aggressive	125.97 segundos	99.97%
cache: on, prefetching: async, on-empty	47.56 segundos	99.74%
cache: on, prefetching: async, limited aggressive	53.04 segundos	99.97%

Tabela 2: Transferência linear, *gigabit ethernet, 500Mb*

Como pode ser visto nas tabelas 1 e 2, os resultados obtidos confirmaram as expectativas deste teste na maioria dos casos. O comportamento diferente do esperado ocorreu durante a utilização do mecanismo de *cache* sem nenhum mecanismo de *prefetching*, sendo que neste caso o tempo utilizado na transferência dos dados foi *menor* do que esperado.

Uma possível explicação para isto decorre da utilização de requisições com tamanho diferente do tamanho dos blocos utilizados no mecanismo de *cache*. Desta maneira, na leitura de algum bloco de maneira parcial, o mecanismo de *cache* poderia requisitar o bloco completo, tornando-o disponível para futuros acessos.

A transferência de dados sem o uso de mecanismos de *prefetching* ofereceu o melhor tempo de transferência de todos os outros testes, conforme foi esperado neste tipo de aplicações.

Para verificar o funcionamento dos mecanismos propostos usando a rede *gigabit ethernet*, um arquivo de 2Gb foi transferido. Os resultados podem ser vistos na tabela 3.

Tipo do teste	**Tempo utilizado**	**Utilização do cache**
cache: off, prefetching: off	162.725 segundos	—
cache: on, prefetching: off	164.182 segundos	—
cache: on, prefetching: on-empty	246.479 segundos	99.99%
cache: on, prefetching: limited aggressive	246.048 segundos	99.99%
cache: on, prefetching: passive	222.718 segundos	87.74%
cache: on, prefetching: aggressive	588.165 segundos	99.99%

Tabela 3: Transferência linear, *gigabit ethernet, 2Gb*

O objetivo deste teste foi a verificação da influência da utilização de um meio de comunicação mais rápido. Como pode ser visto na tabela 3, os resultados obtidos confirmaram os resultados esperados.

Este teste demonstrou que o algoritmo *agressive* clássico possui um tempo de execução que aumenta quase exponencialmente de acordo com a quantidade de dados transmitidos. O algoritmo passivo de *prefetching*, por sua vez, ofereceu o melhor desempenho, porém a pior utilização do espaço de *cache*.

A diferença entre algoritmos *limited aggressive* e *prefetch-on-empty* foi mínima, e o tempo foi praticamente igual. Desta maneira, é possível sugerir que a utilização de cada um dos algoritmos depende da aplicação em questão, uma vez que, mesmo oferecendo comportamento diferente, ambos os algoritmos obtêm resultados semelhantes.

6.1.3 Conclusões

Resumindo os testes de transferência linear, podemos chegar às seguintes conclusões:

- Para a transferência linear dos arquivos, um mecanismo de *cache* e *prefetching* não oferece nenhum ganho de desempenho devido à falta de tempo entre as requisições para executar as operações de leitura antecipada.
- A utilização do *cache* sem nenhum mecanismo de *prefetching* complementar praticamente não prejudica o desempenho das operações.
- O tempo necessário para a execução das operações de *prefetching* depende do algoritmo utilizado.

Como foi visto nos testes realizados, os algoritmos que tendem a uma execução mais *conservativa* (i.e., algoritmos que requisitam o mínimo de blocos na leitura antecipada, tais como *prefetching passivo*) são mais *rápidos* do que os algoritmos mais *agressivos* (tais como *prefetch-on-empty*, *limited aggressive* e *aggressive*). Entretanto, aqueles algoritmos não utilizam o espaço de *cache* de maneira eficiente, aumentando o número de *cache misses*.

De maneira geral, podemos ver que a eficiência do *prefetching* depende do intervalo entre requisições consecutivas.

Nos mecanismos de *cache* e *prefetching* implementados neste trabalho, o acesso a um determinado bloco em *cache* ocorre de maneira sincronizada entre as operações de leitura normal e a leitura antecipada, para eliminar a possibilidade de sobreposição dos dados já presentes em *cache* e manter a consistência dos mesmos.

O estudo detalhado dos resultados obtidos pela utilização de diversos algoritmos de *prefetching* é apresentado na seção 6.4.

6.2 Estudo de caso: *PHTTPD*

Um servidor *web* é um aplicativo bastante interessante para testar os mecanismos de *cache* e *prefetching* propostos, por causa dos seguintes fatores:

- Pequenos intervalos entre requisições consecutivas de entrada e saída, torna-se interessante medir a eficiência dos mecanismos propostos para este tipo de aplicativos.
- Devido à natureza de acessos *HTTP*, é possível determinar um mecanismo mais apropriado de *prefetching* logo na requisição do cliente. Assim, é possível iniciar as atividades do mecanismo de *prefetching* antecipadamente, configurando os parâmetros dos algoritmos utilizados. Ou seja, a aplicação pode informar a política de *prefetching*, ao invés de aguardar a detecção automática pelo algoritmo *CPS*.
- O padrão de acesso do mecanismo de *prefetching*, na maioria das vezes, é seqüencial, tornando mais eficiente o algoritmo *CPS*.

Visando prover uma melhoria de tempo de acesso e da latência de leitura de dados, o mecanismo de *caching* e *prefetching* foi implementado em um servidor multimídia sob demanda, que opera utilizando o protocolo *HTTP* para distribuir os arquivos entre os clientes.

6.2.1 Arquitetura do aplicativo

A arquitetura do servidor criado é apresentada na figura 9.

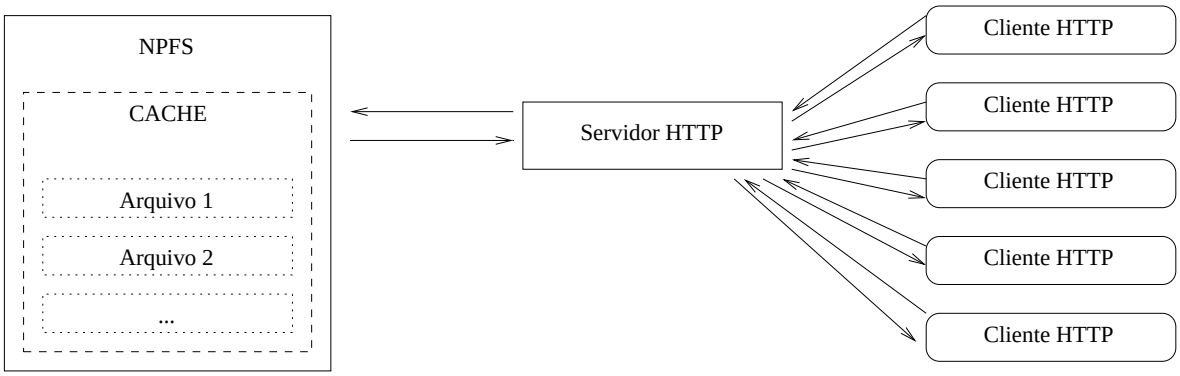

Figura 9: Servidor *phttpd*

O servidor é executado através de um *wrapper* de conexões (por exemplo, *inetd*, *xinetd* ou *tcpserver*), visando oferecer um serviço mais compatível com a arquitetura *UNIX*. Desta maneira, os parâmetros de conexão de rede entre os clientes e o servidor ficam independentes do funcionamento do servidor propriamente dito.

Para cada requisição do cliente o servidor pode retornar duas possíveis respostas, definidas no protocolo *HTTP*. Se o arquivo não foi encontrado, o código de erro *404* é retornado junto com uma mensagem de erro. Caso contrário, uma mensagem de sucesso (*200*) é retornada. Neste caso, as características do arquivo multimídia são especificadas (por exemplo, *Content-type: audio/mpeg*, no caso de um arquivo *mp3*) e o conteúdo do arquivo é enviado.

Durante o processamento da requisição do cliente, os mecanismos de *cache* e *prefetching* são inicializados, ativando uma política estática de *prefetching*

6.2.2 Resultados obtidos

A aplicação em questão, *phttpd*, depende da latência das requisições de entrada e saída de dados para um funcionamento mais efetivo. O processo de interação entre o servidor (*phttpd*) e o cliente é apresentado na figura 10.

Como pode ser visto na figura 10, a latência da aplicação cliente é calculada da seguinte maneira:

$$T_{total} = S_1 + S_2 + L_1 + L_2$$

Onde S_1 e S_2 representam o tempo da requisição dos dados pelo servidor e a respectiva resposta (os dados lidos), e L_1 e L_2 são os valores de tempo decorridos entre requisição pelos dados e a

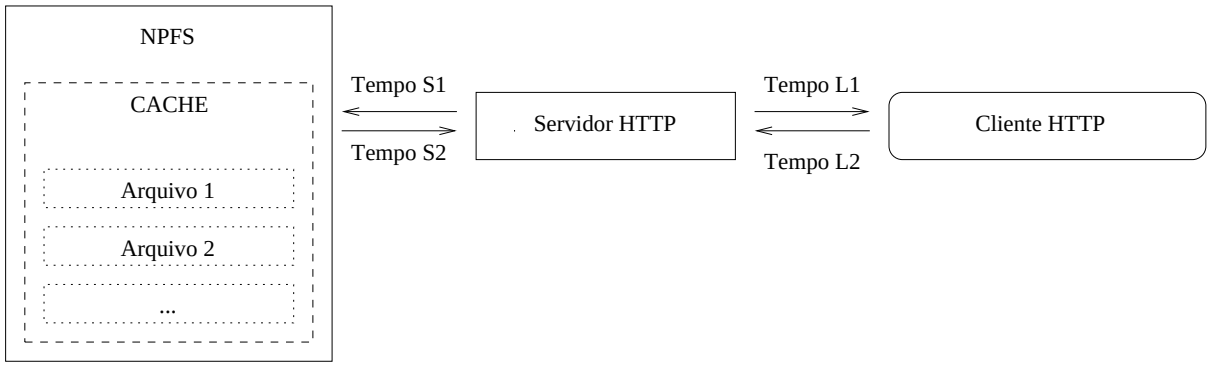

Figura 10: Interação cliente-servidor

respectiva resposta do servidor.

Desta maneira, sem a utilização dos mecanismos de *caching* e *prefetching* existe uma dependência linear entre as requisições do cliente e o tempo necessário para atendê-las, ou seja, tempo decorrido durante o processo de leitura de dados permanece contínuo.

A utilização dos mecanismos de *cache* e *prefetching* teoricamente permite reduzir os valores de *S1* e *S2*, eliminando a necessidade de consulta ao sistema de arquivos *NPFS*, caso o bloco requisitado esteja presente em *cache*. Para isso, um mecanismo de leitura antecipada deve ser empregado, com o objetivo de manter os blocos em *cache* para acessos futuros. Desta maneira, a utilização dos mecanismos de *cache* e *prefetching* permite reduzir a latência das requisições para

$$T_{total} = T_{cache} + L1 + L2$$

sendo que

$$T_{cache} < S_1 + S_2$$

e o T_{cache} representa o tempo de acesso ao *cache*.

Assim, os objetivos dos testes foram:

- Determinar o valor de T_{cache} e a influência do meio de comunicação na diferença entre $T_1 + T_2$ e T_{cache}.
- Determinar a influência dos mecanismos de *cache* e *prefetching* propostos na latência das requisições de entrada e saída.

De acordo com os testes feitos, os seguintes resultados foram obtidos:

Este teste verificou o tempo médio necessário para transferir um bloco de dados de *16Kb* utilizando o *cache* local, rede *gigabit ethernet* e *fast ethernet*. O tamanho do bloco foi esco-

Meio de comunicação	**Tempo de resposta**
T_{cache} (*Cache* local)	0.0032s
$T_1 + T_2$ (*gigabit ethernet*)	0.0052s
$T_1 + T_2$ (*fast ethernet*)	0.0500s

Tabela 4: Latência de acesso

lhido devido à utilização de 4 servidores, visando utilizar completamente o tamanho máximo disponível para transmissão (*64KBytes*, correspondentes ao maior datagrama *UDP*).

Como pode ser visto na tabela 4, o tempo T_{cache} obtido foi menor do que o tempo $T_1 + T_2$, conforme esperado.

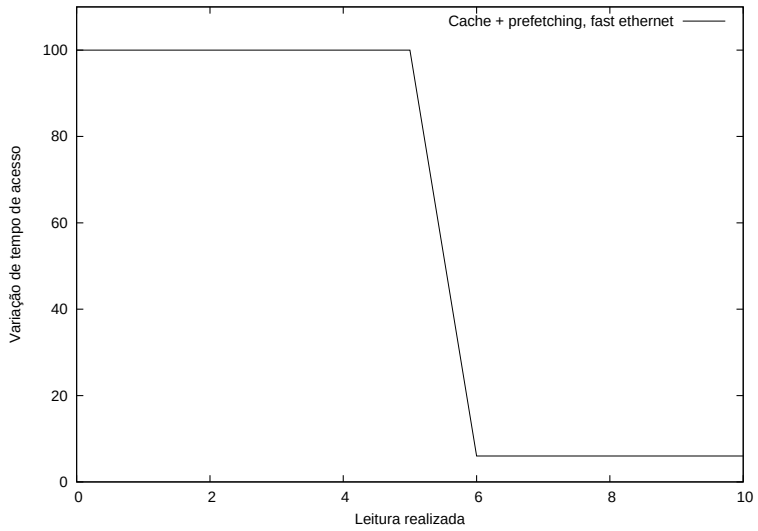

Figura 11: Latência com *cache* e *prefetching*, *fast ethernet*

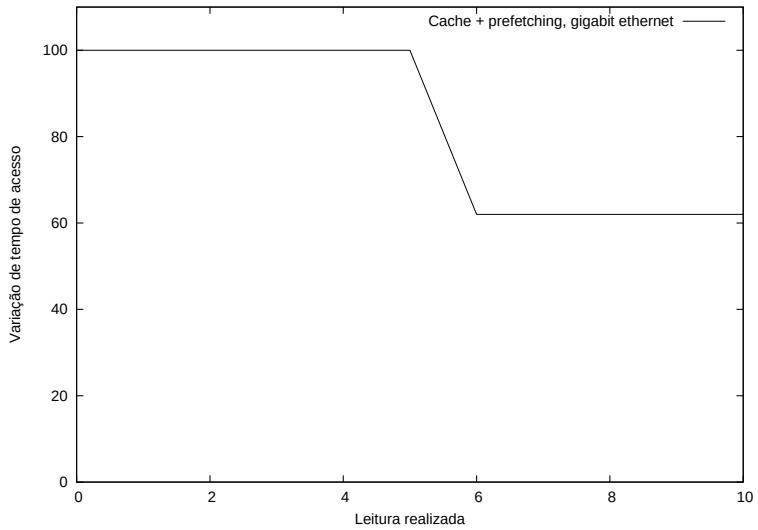

Figura 12: Latência com *cache* e *prefetching*, *gigabit ethernet*

A influência da utilização dos mecanismos de *cache* e *prefetching* no acesso aos dados é demonstrada nas figuras 11 e 12. Como pode ser visto nos gráficos, assim que o mecanismo de *prefetching* consegue preencher o espaço de *cache* com os dados necessários, a latência das requisições decresce até atingir um tempo correspondente ao tempo de acesso à memória, quando todas as requisições são atendidas do *cache*. Entretanto, para cada *cache miss* causado por tempo insuficiente para a operação do mecanismo de *prefetching* a latência é aumentada.

As figuras 11 e 12 mostram a variação relativa da latência decorrente da utilização dos mecanismos de *cache* e *prefetching*. O ganho obtido no caso da rede *gigabit ethernet* é inferior ao da rede *fast ethernet* devido à velocidade mais rápida do meio de transmissão *gigabit ethernet*. Como pode ser visto, o algoritmo *CPS* ativa as operações de leitura antecipada durante a 5a leitura realizada pela aplicação, levando a um decremento de tempo de acesso devido à leitura direta dos dados mantidos em *cache*.

De acordo com os resultados obtidos, é possível deduzir que o tempo necessário para a operação do mecanismo de *prefetching* é calculado em função de L_1 e L_2, sendo que a condição

$$T_{prefetching} < L_1 + L_2$$

deve ser cumprida; caso contrário, um mecanismo de *prefetching* não oferece nenhum ganho de desempenho e prejudica o tempo de acesso, possivelmente resultando na interferência das operações de leitura convencional. Assim, é possível concluir que, para evitar atrasos ou perdas na transmissão de mensagens, causadas por atuação simultânea de requisições de dados com o mecanismo de *prefetching*, e reduzir o número de *cache misses* causados por tempo insuficiente, a seguinte condição deve ser cumprida:

$$T_{prefetching} - T_{cache} < L_1 + L_2$$

ou

$$T_{prefetching} < L_1 + L_2 + T_{cache}$$

Esta condição tem que ser satisfeita principalmente durante o funcionamento *síncrono* do mecanismo de *prefetching*, quando a aplicação efetua as operações de leitura antecipada. No caso de *prefetching assíncrono*, a *prefetching thread* se responsabiliza por encontrar intervalos entre as requisições de leitura da aplicação para requisitar os blocos necessários antecipadamente.

Desta maneira, é possível ver que o funcionamento mais eficiente do mecanismo de *prefetching* depende do tempo da transferência e processamento de dados pela aplicação cliente,

sendo que o tempo de acesso ao *cache* permanece constante independente do meio de comunicação.

Concluindo esta seção, é possível ver que, no caso de aplicativos com pequenos intervalos entre as requisições, os mecanismos de *cache* e *prefetching* possibilitam uma melhoria no desempenho através de uma latência menor de entrada e saída. Entretanto, os mecanismos propostos oferecem ganhos de desempenho somente para casos em que o tempo necessário para a execução do mecanismo de *prefetching* é *menor* do que o tempo decorrido entre requisições consecutivas.

6.3 Estudo de caso: *PSHOUTCAST*

Para determinar a relevância dos mecanismos propostos para as aplicações que apresentam um atraso relativamente grande (onde o intervalo entre as requisições é superior ao tempo necessário para efetuar a leitura de dados propriamente dita) entre as requisições consecutivas, uma classe de aplicativos de distribuição de conteúdo multimídia foi escolhida. Trata-se das aplicações que oferecem *streams* de dados multimídia [93].

Usualmente, para acessar dados multimídia, o cliente deve esperar até que o arquivo inteiro com os dados requisitados esteja transferido até o computador local para processamento e apresentação posteriores. A tecnologia de *streaming* oferece uma outra solução para a transferência e armazenamento de dados multimídia – os dados são transmitidos simultaneamente para todos os clientes conectados, da maneira semelhante ao processo utilizado nas transmissões de rádio/televisão e são processados durante o recebimento. Desta maneira, os dados nunca ficam armazenados nos computadores clientes, oferecendo maior flexibilidade e um modelo de controle sobre a transferência de dados mais avançado [94].

Para manter o fluxo de dados sincronizado com todos os clientes conectados, o servidor de mídia contínua deve esperar um tempo equivalente à duração da mídia transferida. Desta forma, para cada transmissão de, por exemplo, 64Kb de um arquivo codificado em formato *mp3* com taxa de bits fixa em 48000 bits por segundo, a aplicação deve esperar $65535/(48000/8) = 10.9$ segundos antes da próxima requisição. Como foi visto na seção anterior, este tempo é consideravelmente maior do que o tempo necessário para requisitar os blocos a serem utilizados na próxima requsição de dados.

Um mecanismo de *cache* e *prefetching* tem o potencial de melhorar significativamente o desempenho das operações de entrada e saída oferecendo uma melhor latência de acessos para este tipo de aplicação. Uma vez que o tempo necessário para transferir os dados para os clientes

de maneira síncrona é um processo demorado, o funcionamento de mecanismos de *caching* e *prefetching* oferece a possibilidade de funcionamento assíncrono, mantendo os dados necessários para as futuras requisições dentro de um *cache* local do cliente continuamente.

Este caso é típico de um padrão de acesso seqüencial que pode beneficiar-se de um mecanismo de *prefetching*.

6.3.1 Arquitetura da aplicação

Para desenvolver a aplicação, foi utilizado o protocolo *Shoutcast*, desenvolvido por *Nullsoft Inc.*, que oferece uma solução *streaming* para arquivos *mp3*. A aplicação foi feita utilizando a biblioteca *libshout* [91].

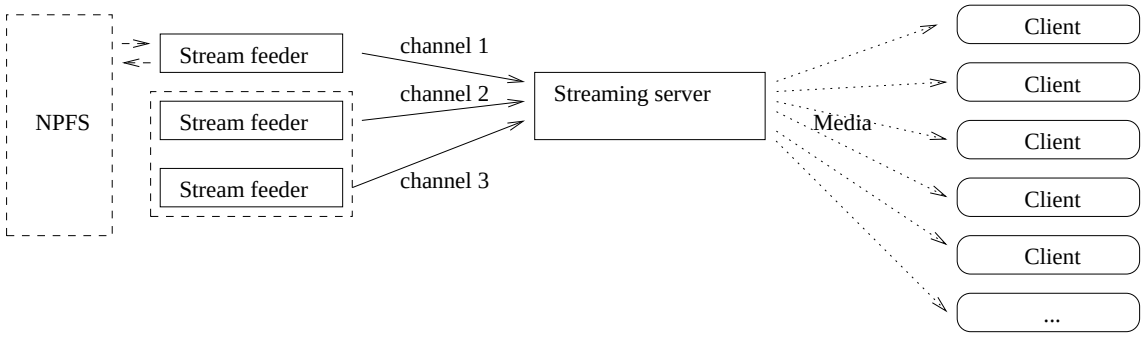

Figura 13: Arquitetura do *pshoutcast*

A arquitetura da aplicação é apresentada na figura 13 e, como pode ser visto na figura, o servidor consiste de dois processos:

- *Stream feeder* – este processo é responsável por requisitar os dados multimídia a serem transmitidos e enviá-los ao servidor de distribuição de conteúdo (*streaming server*).
- *Streaming server* – este processo gerencia as conexões dos clientes e é responsável pela sincronização dos dados multimídia entre os clientes.

Na arquitetura apresentada na figura 13, o *streaming server* utilizado foi o *icecast 1.4*, e o *stream feeder* foi implementado com o suporte ao sistema de arquivos *NPFS*.

O algoritmo de funcionamento do servidor *pshoutcast* criado é apresentado nas tabelas 5 e 6.

Como foi visto na seção anterior, os mecanismos de *cache* e *prefetching* funcionam da maneira mais eficiente quando o intervalo entre as requisições é maior do que o tempo necessário

main thread:

```
  Inicializa a conexão com o stream server
  Especifica o ponto de montagem utilizado
  Determina o arquivo a ser transmitido
  Abre o arquivo e inicializa os mecanismos de cache e prefetching
  Repete
    Lê bloco de dados do arquivo
    Envia bloco lido para o stream server
    Notifica à prefetching thread
    Espera até terminar a transmissao
  Até fim do arquivo
```

Tabela 5: Funcionamento do *pshoutcast – main thread*

prefetching thread:

```
  Repete
    Espera notificação da main thread
    Requisita blocos necessários
    Atualiza padrão de acesso
  Até fechamento de arquivo
```

Tabela 6: Funcionamento do *pshoutcast – prefetching thread*

para a execução do mecanismo de leitura antecipada. Desta maneira, podemos ver que no caso da aplicação em questão, o tempo demorado entre as requisições consecutivas é suficientemente grande para possibilitar uma utilização efetiva do *cache*. Dependendo do grau de agressividade, a aplicação pode ser atendida diretamente do cache. Neste caso, o uso da rede para acessar os servidores fica apenas para o mecanismo de *prefetching*.

Desta maneira, podemos deduzir que o desempenho da aplicação depende da eficiência do mecanismo de *prefetching* para manter o *cache* preenchido continuamente.

De acordo com os testes feitos com dois arquivos diferentes, o algoritmo *CPS* utilizado para determinar as ações a serem tomadas pelo mecanismo de *prefetching* conseguiu os resultados demonstrados na tabela 7, usando os valores padrão.

Arquivo	**Tamanho do arquivo**	***Cache misses***	***Cache hits***
media1.mp3	5305573 Bytes	19	305
media2.mp3	2654372 Bytes	19	144

Tabela 7: Utilização do *cache* pelo *pshoutcast*

Como pode ser visto na tabela 7, o número de *cache misses* permanece constante independentemente do arquivo utilizado. Isso acontece devido aos parâmetros utilizados pelo algoritmo *CPS* como padrão. Desta maneira, enquanto por padrão o algoritmo *CPS* analisa uma seqüência

de 5 blocos para determinar o padrão de acesso da aplicação, é possível reduzir o número de blocos analisados para um número menor. Caso a aplicação saiba antecipadamente o padrão de acesso utilizado, a utilização da função do *API cache_fcntl*, descrita anteriormente, torna possível a redução do número de *cache misses* para um valor menor, sem prejudicar a eficiência das operações de *prefetching*.

6.4 Comparação das estratégias de *prefetching*

Visando avaliar os diferentes algoritmos de *prefetching*, um teste foi feito comparando as diferentes estratégias. De maneira geral, buscava-se determinar quando é melhor utilizar cada um dos algoritmos implementados neste trabalho: *prefetching agressivo*, *passivo*, *limited aggressive* e *prefetch-on-empty*.

Neste teste, um arquivo de 640 blocos foi transferido para determinar o tempo necessário para a operação dos algoritmos de leitura antecipada e a eficiência dos mesmos. Este tamanho do arquivo foi escolhido de forma a prover resultados comparativos dos algoritmos, uma vez que, como pode ser visto na tabela 3, para arquivos grandes os resultados dos algoritmos ficam bastante semelhantes.

Os resultados obtidos são apresentados na tabela 8.

Estratégia de *prefetching*	**Diferença de tempo**	**Utilização do *cache***
Nenhum	-	0%
Passivo	+42%	87.82%
On-empty	+56%	99.74%
Limited aggressive	+57%	99.96%
Agressivo	+360%	99.98%

Tabela 8: Comparação das estratégias de *prefetching*

Como foi visto nas tabelas 1 e 3 da seção 6.1, o funcionamento de cada um dos algoritmos de *prefetching* ocorre de maneira semelhante, independentemente do meio de comunicação utilizado. O tempo decorrido pelos algoritmos de *prefetching* altera-se linearmente de acordo com a quantidade de dados transferidos, com a exceção do algoritmo de *prefetching agressivo*, que apresenta um crescimento quase-exponencial.

Como pode ser visto na tabela 8, a utilização efetiva do *cache* é proporcional à *agressividade* dos algoritmos à e ao tempo necessário para a operação dos mesmos.

6.4.1 Comportamento dos algoritmos

A seguir o comportamento detalhado dos algoritmos de *prefetching* é apresentado, para cada um dos algoritmos em questão. Nos gráficos apresentados, o eixo vertical representa a posição do bloco sendo lido, e o eixo horizontal – o número da operação de leitura realizada. Os dados lidos pelo mecanismo de *prefetching* podem ser representados como a área entre os dois gráficos – o da leitura normal, e o da leitura antecipada. Nas figuras, o comportamento tanto do mecanismo de *prefetching* quanto o da leitura normal é igual até um certo ponto, no qual o algoritmo *CPS* detecta um padrão de acesso seqüêncial e ativa o mecanismo de leitura antecipada e começa a requisitar os dados.

Os algoritmos implementados e estudados neste trabalho são *passive*, *agressive*, *limited aggressive* e *prefetch-on-empty*.

6.4.1.1 *Prefetching Passivo*

O comportamento de algoritmo de *prefetching* passivo é apresentado na figura 14.

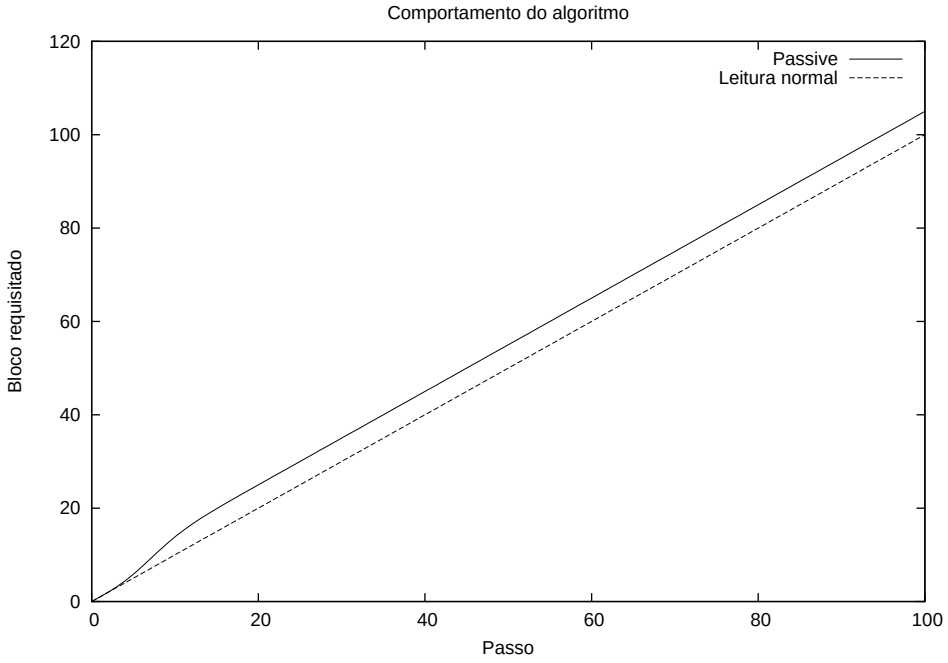

Figura 14: *Prefetching passivo*

Como pode ser visto na figura 14, o algoritmo de *prefetching* passivo limita-se a requisitar os dados mais adjacentes à posição de leitura atual. Desta maneira, o número de dados buscados antecipadamente pelo algoritmo de *prefetching* e ainda não utilizados é mínimo. Entretanto, como foi demonstrado nos testes, este *look-ahead* não é suficiente para manter o *cache* cheio

com os blocos a serem lidos tempo todo.

Devido a um número menor de dados transferidos, este algoritmo oferece o melhor desempenho de todos os algoritmos de *prefetching*, como foi observado nas tabelas 1, 3 e 8.

Como conclusão, é possível afirmar que este algoritmo é mais adequado para as seguintes áreas de aplicação:

- Transferência de dados em que os intervalos entre as requisições são pequenos e o aspecto mais importante é o tempo das operações de entrada e saída.
- Aplicações que utilizam pouco o espaço de *cache* e concentram-se na transferência de dados com pouco processamento dos mesmos.

Dado o pequeno intervalo entre requisições, o uso de uma política de *prefetching* mais agressiva nestes casos muito provavelmente geraria interferências com as requisições normais, prejudicando o tempo de resposta.

6.4.1.2 *Prefetching Agressivo*

O comportamento deste algoritmo é demonstrado na figura 15.

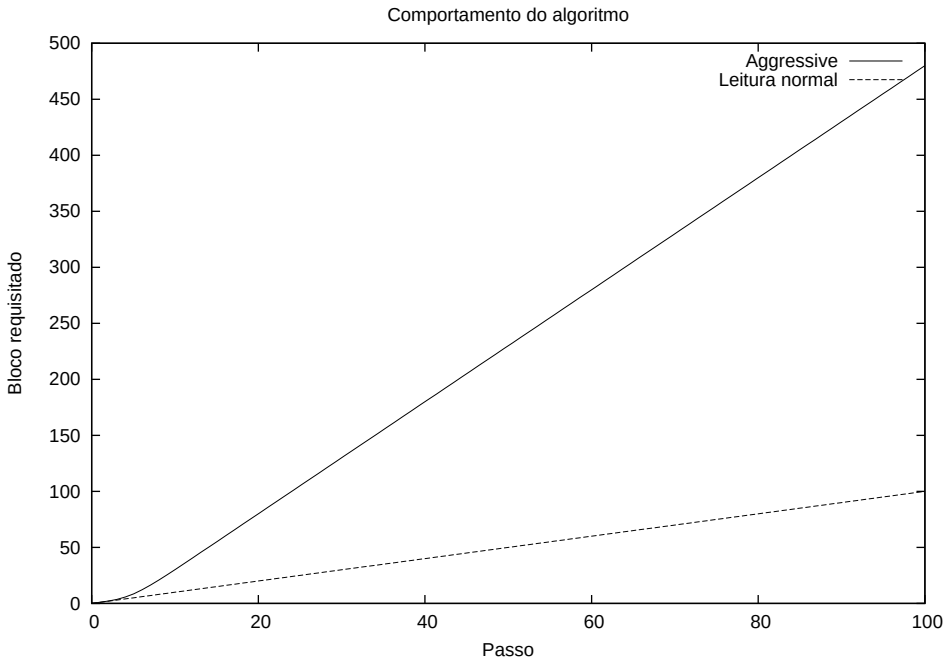

Figura 15: *Prefetching agressivo – funcionamento ideal*

O *prefetching agressivo* tem como objetivo requisitar o número máximo de blocos para

leitura posterior. O comportamento do algoritmo quando o espaço de *cache* disponível para armazenar os dados é ilimitado é demonstrado na figura 15.

Entretanto, quando o número de blocos lidos ultrapassa o tamanho do *cache*, este algoritmo passa a prejudicar o desempenho das operações de entrada e saída, efetivamente aumentando o número de requisições necessárias para obter um bloco de dados.

Desta forma, o algoritmo agressivo funciona de maneira adequada nas situações em que o tamanho do arquivo é inferior à quantidade de memória disponível em *cache*. Caso contrário, assim que a memória disponível chegar ao fim, o comportamento do algoritmo torna-se menos eficiente, como apresentado na figura 16.

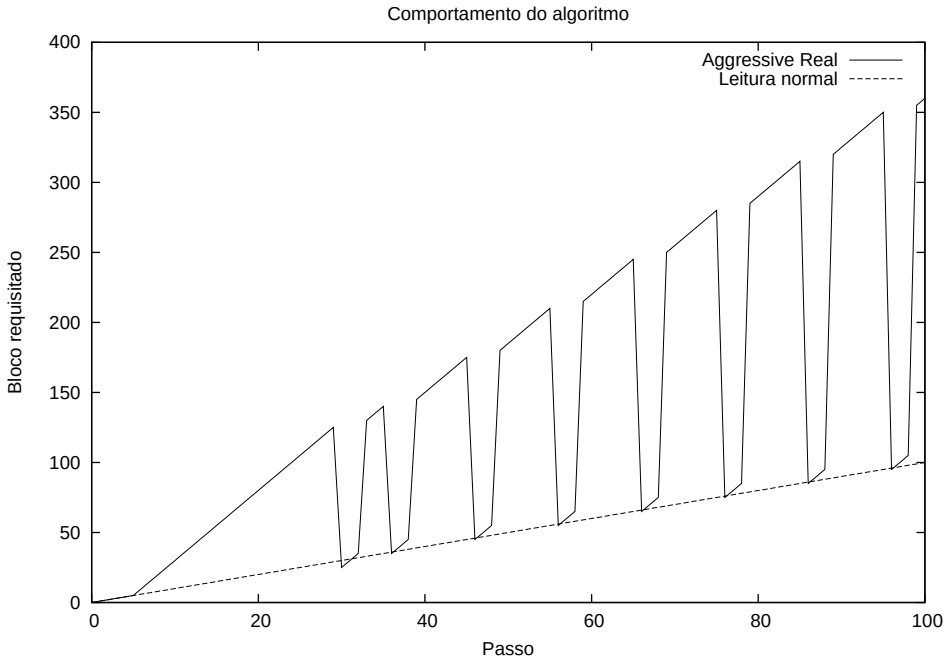

Figura 16: *Prefetching agressivo – funcionamento real*

Como pode ser visto na figura 16, o algoritmo começa a re-requisitar os blocos na medida em que os blocos requisitados anteriormente são sobrescritos por blocos novos, prejudicando o desempenho das operações. Assim, o *cache* permanece sempre cheio com os blocos a serem processados, ao custo de diversas leituras do mesmo bloco pelo algoritmo de *prefetching*. Os picos e as quedas bruscas apresentados na figura representam os momentos nos quais uma operação de leitura de *cache* não encontra um bloco requisitado (*cache miss*). Neste caso, o algoritmo de *prefetching* volta a requisitar o bloco necessário e retoma as operações de leitura antecipada logo em seguir.

Por exemplo, na figura 16 enquanto a operação de leitura normal requisita o bloco 35, o mecanismo de *prefetching* está lendo o bloco 140. Como o tamanho de *cache* neste exemplo

é de 100 blocos, a requisição de leitura antecipada descarta um dos blocos presentes em *cache* para a inclusão de um novo bloco. Como a "distância" entre as posições de leitura da aplicação e do algoritmo de *prefetching* excede o tamanho do *cache*, o bloco descartado é um dos blocos a ser lidos ainda pela aplicação. Logo, a requisição de leitura deste bloco resulta em um *cache miss*, tornando necessária uma segunda leitura do mesmo bloco pelo algoritmo de *prefetching*.

Uma política de substituição de páginas baseada no algoritmo *FIFO* pode ser crítica neste caso, enquanto políticas que analisam o histórico de acesso às páginas ou seqüências de acessos, tais como *LIRS* e *SEQ* podem suavizar as quedas de desempenho do algoritmo apresentado.

Desta maneira, este algoritmo teria aplicação mais adequada para as seguintes áreas:

- Aplicações que apresentam intervalos grande entre as requisições, uma vez que nestas aplicações o algoritmo apresentado consegue requisitar todos os dados necessários antecipadamente.
- Aplicações que reutilizam os dados do *cache* freqüentemente, sendo que os dados permanecem sempre os mesmos (como, por exemplo, em arquivos que cabem inteiramente em *cache*).

6.4.1.3 Algoritmo *Limited aggressive*

Este algoritmo visa eliminar algumas das deficiências do algoritmo agressivo, tais como tempo de execução alto, ao mesmo tempo procurando manter a alta utilização do espaço de *cache*. A operação deste algoritmo é apresentada na figura 17.

Como pode ser visto na figura 17, o comportamento deste algoritmo é semelhante ao comportamento do algoritmo agressivo, visto anteriormente. Entretanto, devido à limitação do *look-ahead* do algoritmo, a possibilidade de descarte de blocos ainda não utilizados é menor do que no algoritmo de *prefetching agressivo*.

O *look-ahead* do algoritmo é limitado para não ultrapassar o tamanho de *cache*, e ao atingir este limite, o algoritmo passa a se comportar como um algoritmo de *prefetching passivo*, como pode ser visto na figura 17 no *passo 25* da leitura. Neste caso, como o *cache* já possui um número grande de blocos buscados pelo algoritmo de *prefetching*, o algoritmo procura manter a mesma distância entre os blocos requisitados e buscados pelas operações de leitura antecipada.

A taxa de utilização do *cache* no caso deste algoritmo é praticamente igual à taxa obtida pelo algoritmo agressivo. Entretanto, os problemas que o algoritmo agressivo de *prefetching*

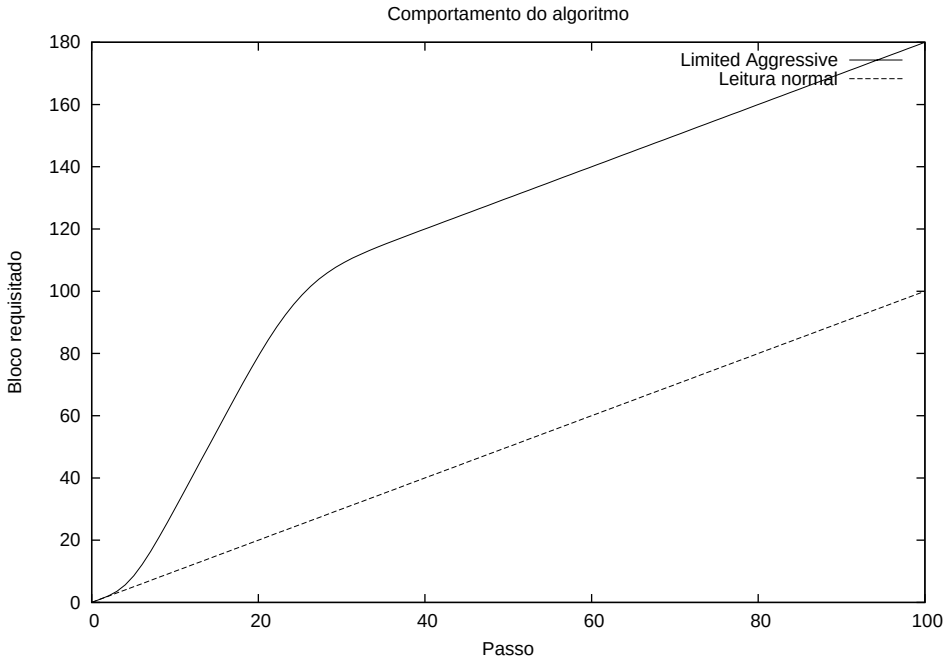

Figura 17: *Limited aggressive*

apresenta, tais como a remoção dos dados ainda não lidos do *cache* para a inserção de novos blocos, são eliminados.

Podemos concluir que este algoritmo é mais adaptado para as seguintes aplicações:

- Aplicações que precisam tanto de tempo de acesso (*latência*) baixo quanto da utilização alta do espaço de *cache*. Desta maneira, este algoritmo é adequado para a maioria das aplicações cujo padrão de acesso e o comportamento dos mecanismos de entrada e saída são indeterminados.
- Aplicações que podem alterar o padrão de acesso freqüentemente, uma vez que o balanceamento entre a taxa de preenchimento de *cache* alta e adaptação rápida a uma mudança no padrão de acesso torna este algoritmo bastante interessante para este tipo de aplicações.

6.4.1.4 Algoritmo *Prefetch-on-empty*

Este algoritmo foi criado visando atender as aplicações cujo comportamento pode mudar ao longo do tempo, tornando necessário algum mecanismo de ajuste rápido. Outro tipo de aplicações que podem obter um desempenho melhor com este tipo de algoritmo são as aplicações que intercalam os intervalos de transferência de dados com intervalos de processamento dos mesmos.

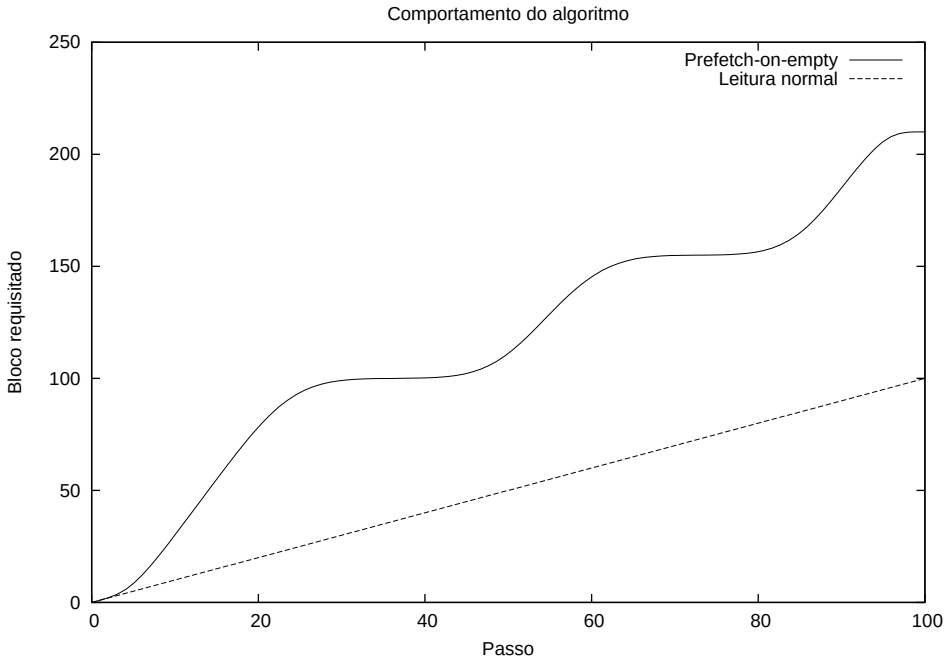

Figura 18: *Prefetch-on-empty*

O comportamento deste algoritmo é apresentado na figura 18. Como pode ser visto na figura, o algoritmo procura efetuar a transferência de dados em *rajadas*, intercalando os intervalos de transferência intensiva de dados com períodos mais inativos. Desta maneira, é possível alternar entre as atividades de transferência de dados com os intervalos de processamento. Potencialmente, é possível manter o *cache* cheio continuamente sem interferir no comportamento da aplicação. No gráfico, os períodos de transferência de dados são caracterizados pelo crescimento contínuo das posições lidas pelo algoritmo, enquanto os períodos de inatividade são caracterizados pela manutenção dos dados presentes em *cache* apenas. As alterações entre os diferentes modos de operação do algoritmo podem ser vistos nos passos $24 - 50$ e $60 - 85$.

Como foi visto nos testes realizados na seção 6.1, o desempenho e a taxa de utilização do *cache* é compatível com o algoritmo *limited aggressive*, visto anteriormente. Desta maneira, podemos concluir que a utilização deste algoritmo fica a critério da aplicação em questão.

6.4.1.5 Funcionamento conjunto

O funcionamento dos algoritmos de *prefetching* diferentes pode ser visto na figura 19. Na figura, o sistema consiste de um *cache* com a capacidade para 100 blocos. Como pode ser visto, os algoritmos *agressivos* atingem uma taxa de requisição de blocos muito superior aos outros algoritmos, ao custo da possibilidade de diversas requisições dos mesmos blocos. Desta maneira, o funcionamento de algoritmo permanece adequado até o passo em que o algoritmo

passa a retirar os blocos ainda não processados do *cache*. Após este passo, o algoritmo passa a manter o *cache* preenchido com novos blocos retirando os blocos com menor prioridade do *cache*, que normalmente são blocos que ainda não foram lidos pelo sistema.

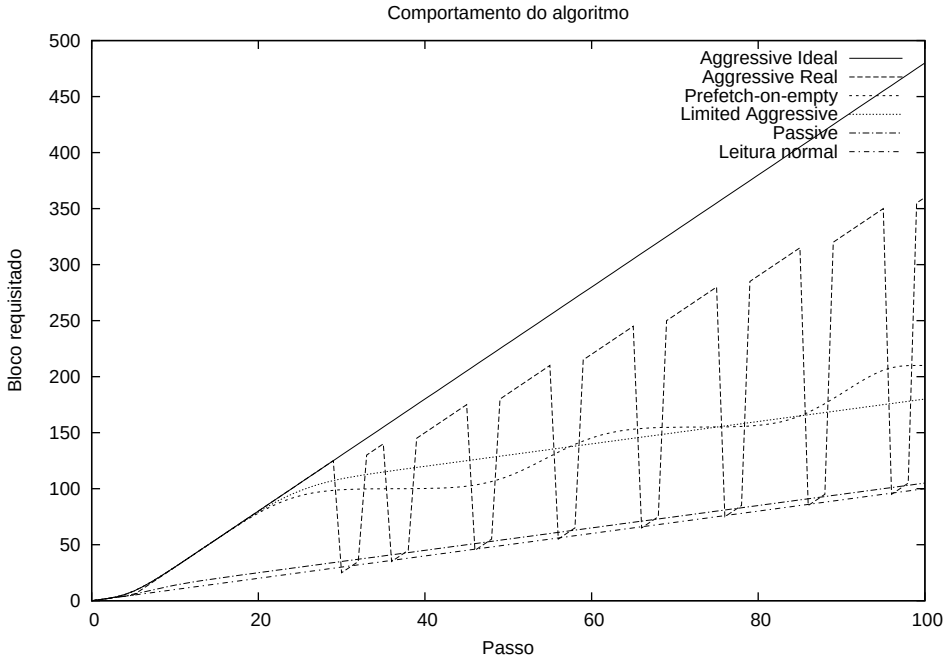

Figura 19: Algoritmos de *prefetching*

O algoritmo *passivo*, por sua vez, procura manter o *cache* preenchido com um número fixo de blocos, evitando acessos muito distantes do ponteiro atual da leitura da aplicação.

Os algoritmos *limited aggressive* e *prefetch-on-empty* apresentam o comportamento similar ao algoritmo agressivo até atingir um certo *threshold* pré-definido por padrão para metade do espaço do *cache* (no caso, 50 blocos). Ao ultrapassar esse limite o algoritmo *limited aggressive* passa a se comportar como o algoritmo *passivo*, e o algoritmo *prefetch-on-empty*, por sua vez, termina as operações de leitura antecipada até o *cache* possuir menos da metade dos blocos requisitados. Depois disto, o algoritmo volta ao comportamento anterior.

6.4.2 Tempos de acesso

Para determinar a variação de tempos de acesso ao longo da execução dos mecanismos de *prefetching*, um teste diferente foi efetuado. Neste teste, um arquivo menor foi transferindo, verificando a variação de tempo de acesso à cada leitura efetuada.

Inicialmente, os testes foram realizados com arquivos de tamanho suficiente para exceder a memória *RAM* em todos os servidores, inibindo o mascaramento dos resultados com o uso de seus *caches* locais do sistema.

Entretanto, a comparação com o uso de arquivos menores mostrou que há pouca influência no resultado final dos tempos de transferência e nos padrões de acesso.

Neste capitulo, para uma melhor visualização do comportamento dos mecanismos de *cache* e *prefetching*, são demonstrados resultados de transferência de arquivos menores. Os dados resultantes da transferência de arquivos grandes são apresentados quando alguma diferença é observada no funcionamento destes mecanismos.

O tamanho do arquivo transferido foi igual a $3 * Tamanho_{cache}$ (tamanho total do *cache*), sendo que o tamanho do *cache* foi fixado em *100 blocos*.

Os tempos de acesso obtidos sem a utilização de mecanismos de *cache* e *prefetching* podem ser vistos na figura 20. Neste caso, os tempos de acesso variam em torno de $1000ms$, que é o tempo necessário para efetuar a leitura convencional de dados neste caso.

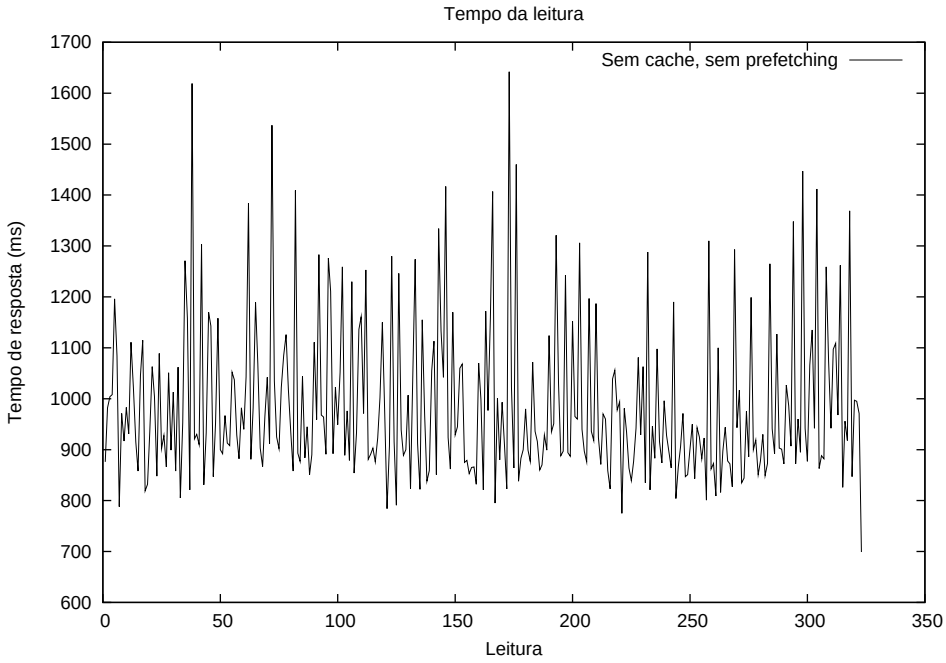

Figura 20: Tempos de acesso sem *cache* e *prefetching*

Ativando os mecanismos de *cache*, os tempos de acesso são alterados de acordo com a figura 21.

Como pode ser observado, os tempos de acesso apresentados na figura 21 são semelhantes aos da figura 20 na maioria das vezes. Entretanto, como foi visto na seção 6.1, em algumas operações de leitura blocos adicionais são requisitados e mantidos em *cache* para futuros acessos, diminuindo o tempo de acesso a eles.

Para avaliar os algoritmos de *prefetching*, um procedimento diferente foi empregado nos

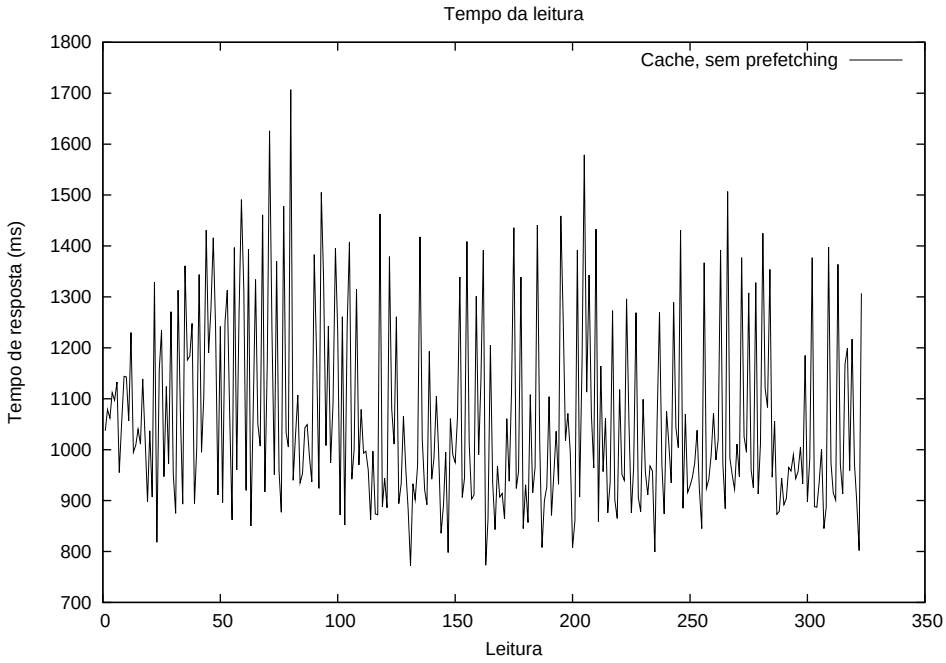

Figura 21: Tempos de acesso com *cache*

testes. Para cada algoritmo de *prefetching* o teste foi realizado com o algoritmo atuando de maneira *síncrona* e *assíncrona*.

Na execução síncrona, as requisições de leitura antecipada foram executadas pela aplicação, visando avaliar o comportamento do algoritmo executado de maneira pré-determinada pela aplicação. Isto é, neste caso foram medidos os tempos de requisição de um bloco com o tempo necessário para executar as operações de *prefetching*. Como é visto nos resultados, quando o mecanismo de leitura antecipada precisa requisitar blocos adicionais, o tempo total de requisição aumenta de acordo com o número de blocos requisitados. Por outro lado, quando todos os blocos a serem lidos já encontram-se em *cache*, o tempo de leitura é mínimo e é igual apenas ao tempo necessário para:

- Determinar se todos os blocos referentes à operação de leitura atual já estão presentes em *cache*.
- Efetuar a cópia dos dados da memória *cache* para a área de memória da aplicação.

Assim, a maneira da requisição dos blocos utilizada no mecanismo síncrono pode ser denominada de *postfetching*, uma vez que os blocos a serem lidos no futuro são requisitados logo depois da última operação da leitura realizada.

O mecanismo assíncrono, por sua vez, utiliza os intervalos de tempo entre as requisições consecutivas para requisitar os blocos antecipadamente. Desta maneira, quando não há interva-

los suficientemente grandes entre as requisições consecutivas, o algoritmo pode não conseguir trazer todos os blocos para a memória *cache* antecipadamente, aumentando o tempo de acesso. Por outro lado, o funcionamento assíncrono permite aproveitar todos os intervalos entre as requisições de dados sem nenhuma coordenação por parte da aplicação.

Tanto a execução síncrona quanto a assíncrona utilizam um mecanismo de acesso exclusivo ao espaço do *cache*, com o objetivo de manter a consistência dos dados. Este mecanismo de *serialização* é utilizado somente durante a atualização dos blocos, evitando possíveis sobreposições de dados. Durante a leitura dos blocos do *cache* o mecanismo não é ativado para não prejudicar o desempenho das operações.

O funcionamento dos algoritmos é apresentado a seguir. Nos gráficos, o tempo necessário para *ler um bloco do NPFS* permanece em torno de *1000ms*, o tempo de leitura de um bloco do *cache* varia em torno de *80ms*, e o tempo necessário para efetuar as operações de *prefetching* é igual a $T_{leiturade1bloco} * N_{blocoslidos}$. Por padrão, número de blocos lidos é igual a *5*, determinando o tempo necessário para realizar as operações de *prefetching* como sendo em torno de *4000-5000ms*. A serialização dos acessos ao espaço do *cache* causa a necessidade de esperar o término da última operação que modifica o conteúdo do *cache* (leitura normal de dados ou leitura antecipada) antes de iniciar a próxima operação, resultando neste aumento de tempo.

Tendo em vista a operação do mecanismo de leitura antecipada, é possível concluir que o tempo mínimo entre as requisições consecutivas deveria ser igual a $T_{leiturade1bloco} * N_{blocoslidos}$, ou $1000ms * 5 = 5000ms$ para evitar conflitos entre as operações de leitura convencional e antecipada.

Os gráficos 22, 23, 24, 25, 26, 27, 28 e 29 a seguir demonstram o comportamento dos mecanismos de *prefetching*, sem nenhum intervalo entre as requisições.

6.4.3 *Prefetching passivo*

O funcionamento do algoritmo de *prefetching* passivo é demonstrado nas figuras 22 (funcionamento síncrono) e 23 (funcionamento assíncrono).

Como pode ser visto nas figuras, o algoritmo entercala constantemente os *cache hits* com *cache misses* nas requisições. O tempo necessário para efetuar a leitura dos blocos já presentes em *cache* varia em torno de *80ms*, enquanto a operação de leitura de blocos ausentes resulta na requisição dos blocos em questão, exigindo um período de tempo muito superior.

Os picos de desempenho ocorrem quando o espaço disponível em *cache* é esgotado, tor-

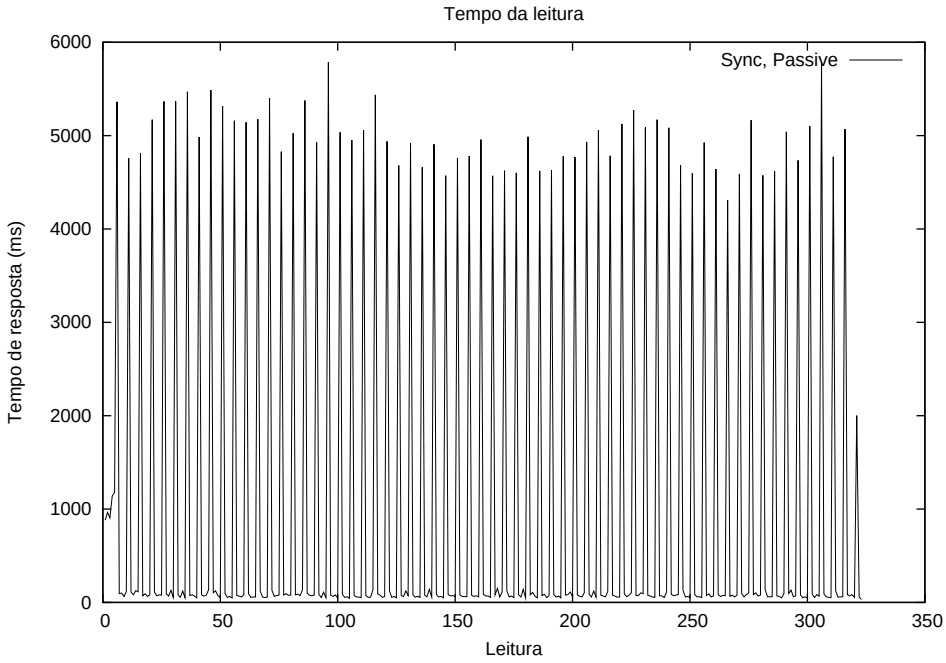

Figura 22: *Prefetching Passivo*, execução síncrona

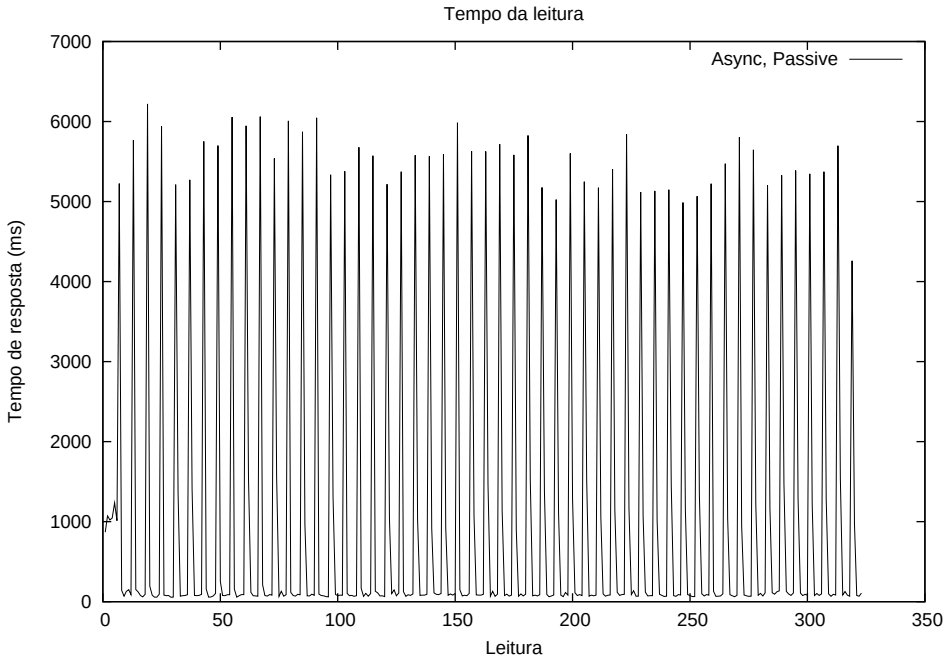

Figura 23: *Prefetching Passivo*, execução assíncrona

nando necessária a remoção de alguns blocos, como pode ser visto nas figuras 22 e 23.

Nos gráficos 22 e 23 podemos ver que o algoritmo de *prefetching passivo* efetua as requisições de leitura antecipada continuamente, até o fim do arquivo. Entretanto, como será visto nos algoritmos apresentados a seguir, o número de blocos requisitados nos processos de leitura antecipada é menor do que nos outros algoritmos, reduzindo o tempo adicional das operações

de entrada e saída.

Este algoritmo em particular não apresenta grandes variações entre os modos de funcionamento síncrono e assíncronos, devido à natureza de leitura antecipada dos blocos. Como o número de blocos requisitados na leitura antecipada é relativamente pequeno (5 blocos por leitura), o comportamento do algoritmo é semelhante tanto nas operações síncronas (controladas pela aplicação) quanto nas assíncronas (controladas pela *prefetching thread*).

6.4.4 *Prefetching agressivo*

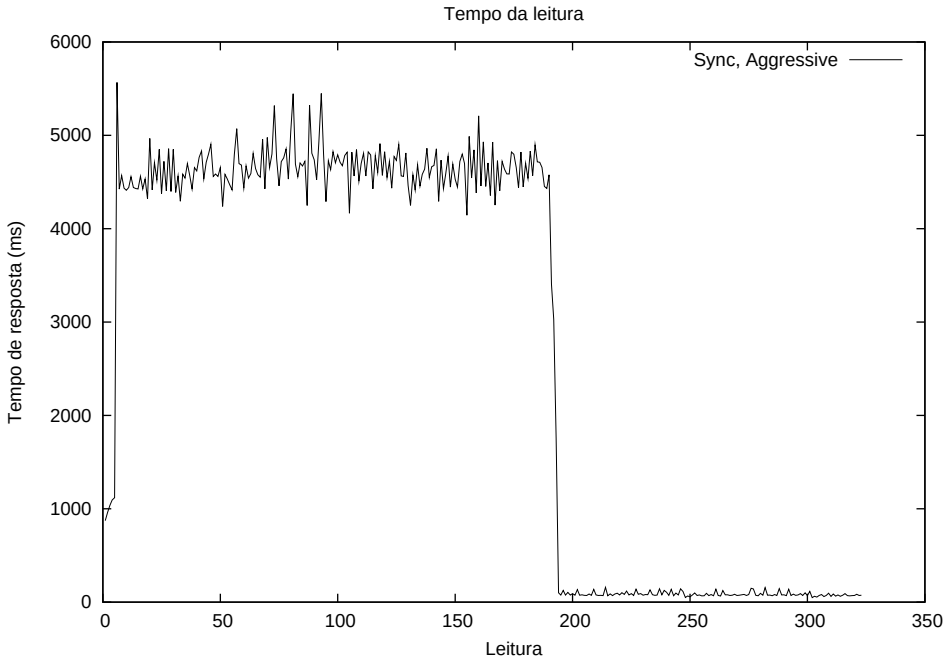

Figura 24: *Prefetching Agressivo*, execução síncrona

Como pode ser visto nas figuras 24 e 25, o comportamento do algoritmo de *prefetching agressivo* apresenta variações visíveis nos modos de funcionamento síncrono e assíncrono.

Entretanto, o comportamento dos dois algoritmos é bastante similar, sendo que a única diferença entre os gráficos são os intervalos de requisição de dados.

No modo de funcionamento síncrono, o algoritmo procura manter o *cache* sempre com os blocos ainda não processados, levando a um aumento de tempo significativo nas operações de leitura no começo das operações. Entretanto, no fim do arquivo, uma vez que todos os blocos já são lidos durante as operações anteriores, o tempo de leitura dos blocos pertencentes ao fim do arquivo é igual apenas ao tempo de leitura dos blocos do *cache*.

Única diferença entre o funcionamento síncrono e assíncrono é a escolha dos intervalos para

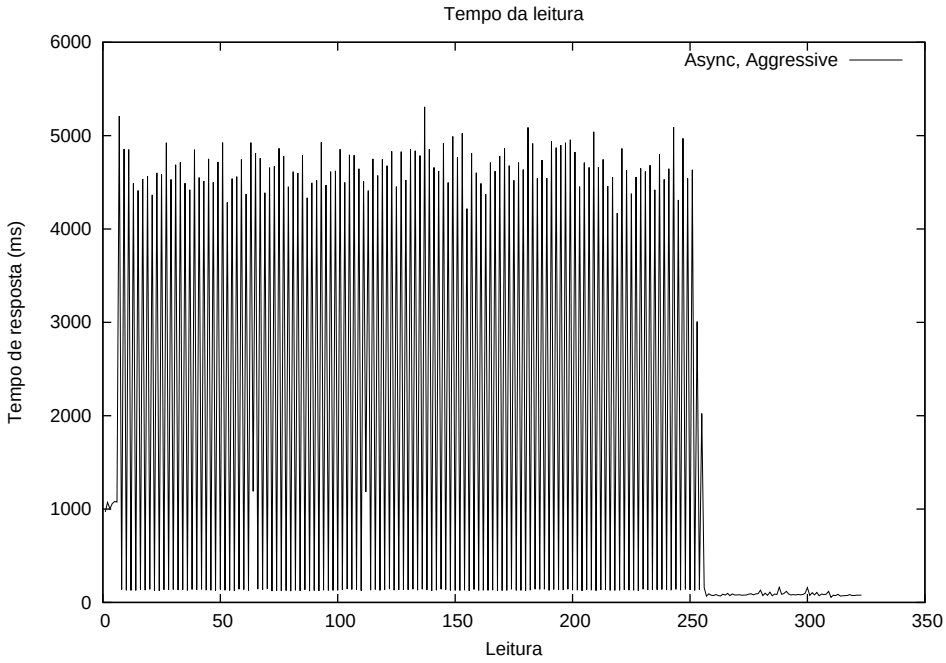

Figura 25: *Prefetching Agressivo*, execução assíncrona

a leitura antecipada dos blocos. Ao contrário do funcionamento síncrono, a *prefetching thread* efetua a leitura antecipada dos blocos nos intervalos de leitura normal, executada pela aplicação. Desta maneira, as operações de leitura da aplicação podem acessar os blocos inseridos em *cache* pelas operações anteriores de *prefetching*, como pode ser visto claramente na figura 24. Na figura, os picos representam a leitura dos dados por parte da *prefetching thread*, enquanto os dados lidos diretamente do *cache* pela aplicação apresentam um tempo de acesso menor.

Como pode ser visto nas figuras 24 e 25, no fim do arquivo tanto o método síncrono quanto o assíncrono apresentam comportamento igual, lendo os blocos apenas do *cache*.

Desta maneira, tanto a operação síncrona quanto a assíncrona apresentam um tempo total de operações compatível, embora o funcionamento esteja diferente.

6.4.5 Algoritmo *Limited Aggressive*

O comportamento deste algoritmo apresenta um balanceamento entre os algoritmos de *prefetching agressivo* e *prefetching passivo*, como pode ser visto nas figuras 26 e 27.

De maneira geral, é possível observar que o algoritmo, tanto na versão síncrona quanto na assíncrona, apresenta traços semelhantes aos do algoritmo de *prefetching agressivo*, principalmente na taxa de requisição de blocos e na utilização efetiva do espaço de *cache* para todas as leituras no fim do arquivo. Entretanto, diferentemente do algoritmo de *prefetching agressivo*,

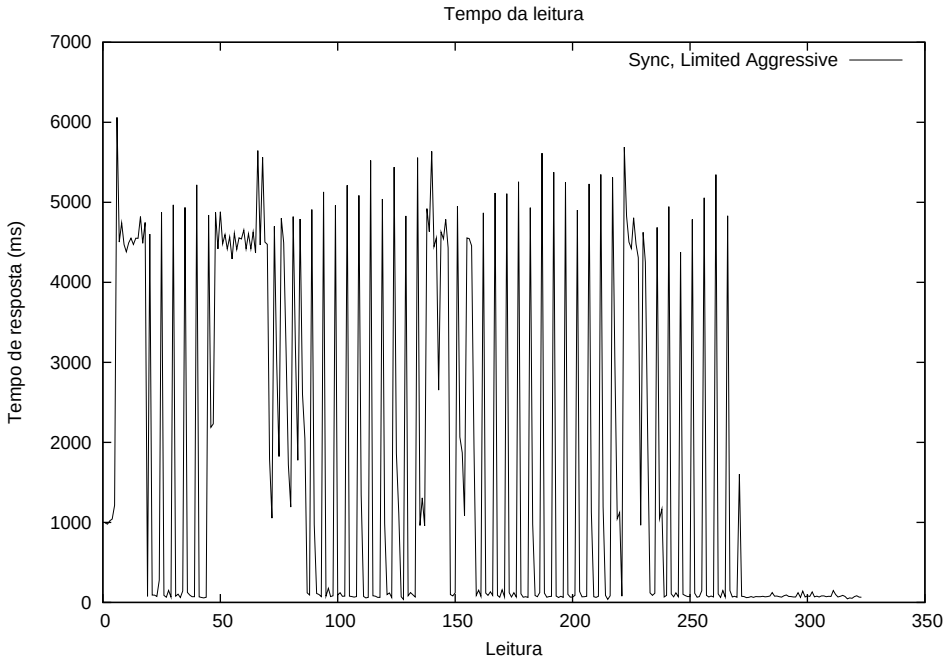

Figura 26: *Prefetching Limited Agressive*, execução síncrona

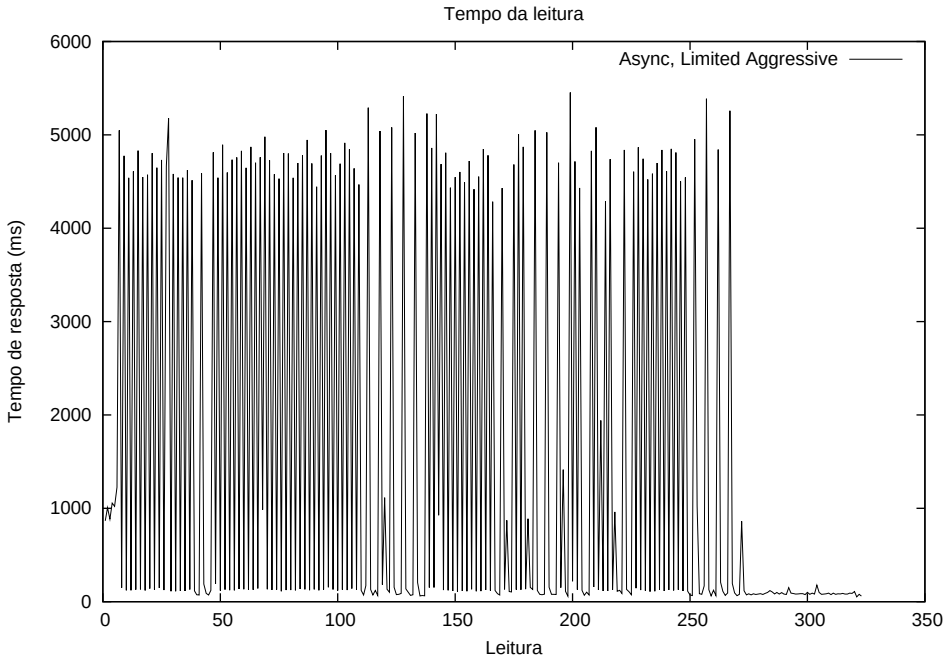

Figura 27: *Prefetching Limited Agressive*, execução assíncrona

o algoritmo *limited aggressive* apresenta características do algoritmo de *prefetching passivo*, principalmente nas leituras de dados buscados antecipadamente intercaladas com as leituras efetuadas do *cache* pela aplicação.

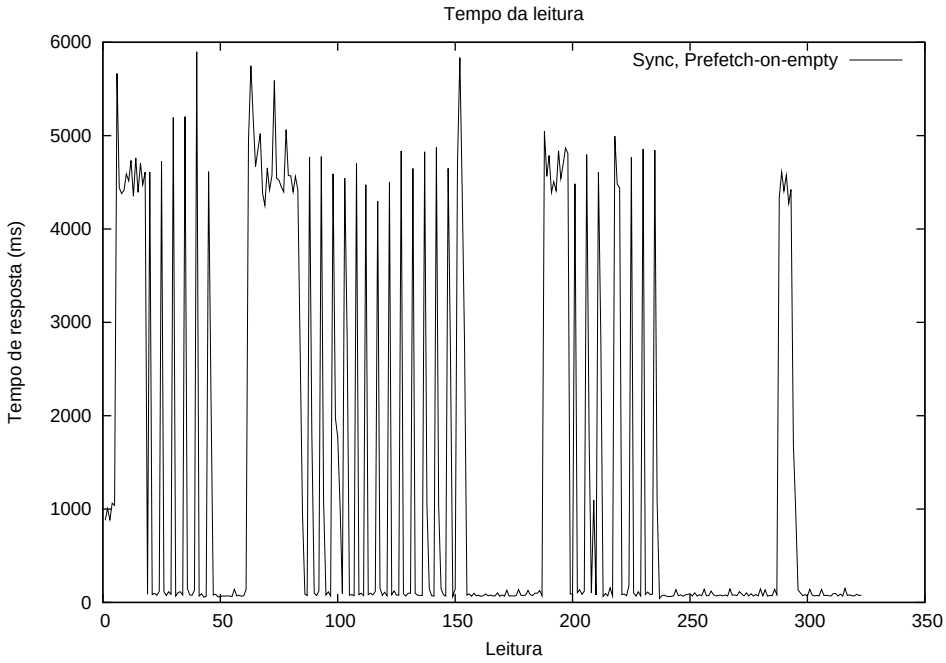

Figura 28: *Prefetch-on-empty*, execução síncrona

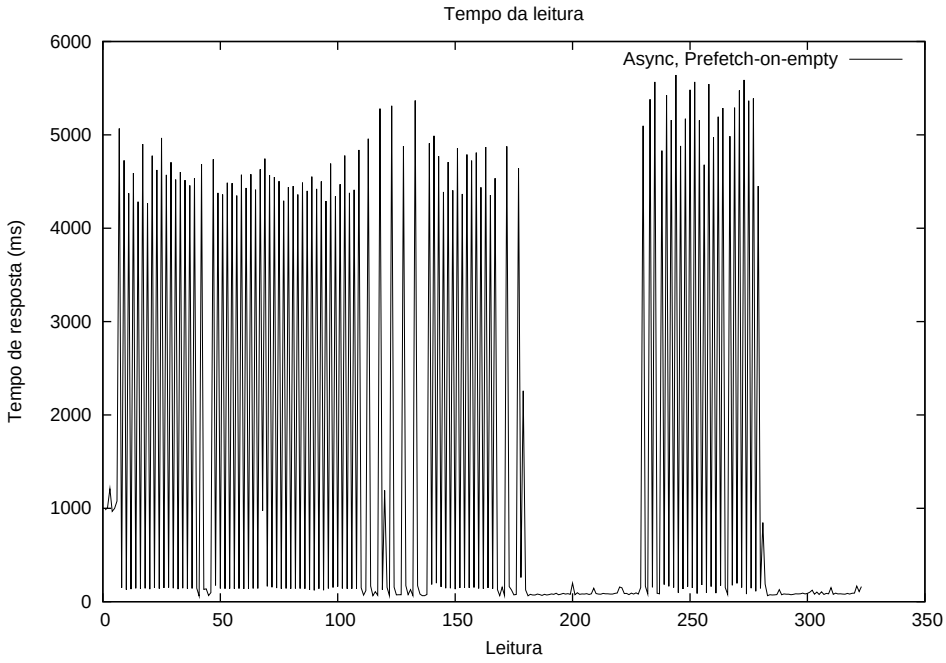

Figura 29: *Prefetch-on-empty*, execução assíncrona

6.4.6 Algoritmo *Prefetch-on-empty*

O funcionamento do algoritmo *prefetch-on-empty* é demonstrado nas figuras 28 e 29. Como pode ser observado, o comportamento dos métodos síncrono e assíncrono é bastante semelhante, com a diferença de que o método assíncrono demora mais para preencher o *cache* do sistema

para alterar o período de requisição de dados por um período de inatividade.

Isso ocorre porque a aplicação continua requisitando os blocos adicionais continuamente no modo de funcionamento assíncrono, forçando o algoritmo a buscar mais blocos e aumentando o tempo necessário para atingir o *threshold* do algoritmo.

Nas figuras 28 e 29 é possível observar a claramente alteração entre os períodos de requisição de blocos e períodos de inatividade, efetuando uma transferência dos dados em “rajadas”.

6.4.7 Execução com atrasos

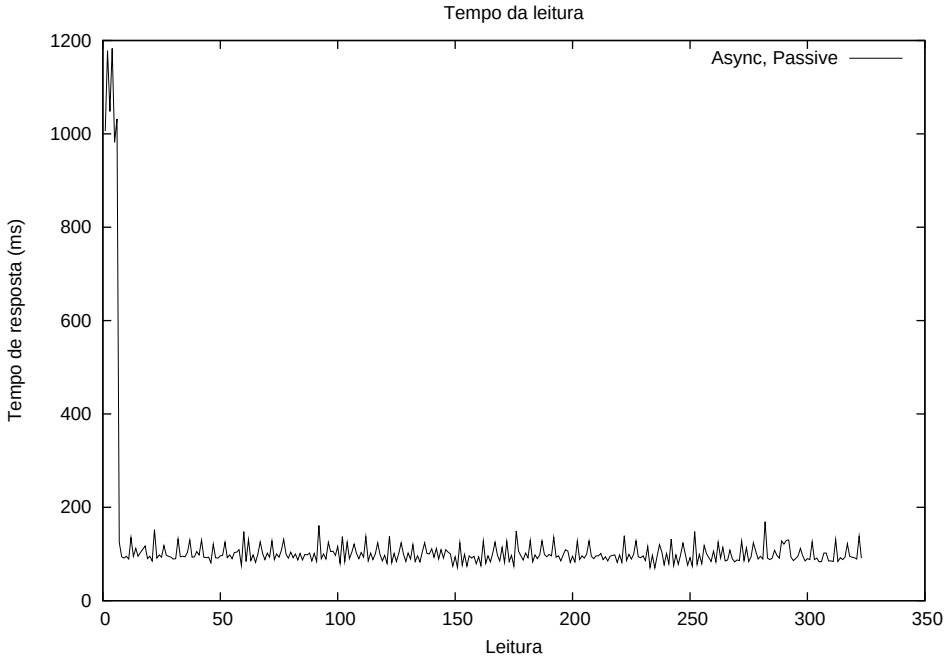

Figura 30: *Prefetching assíncrono*, intervalo de 1 segundo

Para verificar o comportamento dos tempos de acesso oferecidos pelos algoritmos de *prefetching* com intervalos entre as requisições, o mesmo teste foi executado novamente, adicionando intervalos de *1 segundo* entre as requisições consecutivas. Os resultados deste teste são apresentados na figura 30, e todos os algoritmos de *prefetching* (*aggressive*, *passive*, *limited aggressive* e *prefetch-on-empty*) apresentaram comportamentos praticamente iguais neste caso, conforme foi previsto nos capítulos 6.2 e 6.3.

Como pode ser visto na figura 30, depois da ativação do mecanismo de leitura antecipada no passo *5*, o tempo necessário para efetuar as leituras consequentes é igual ao tempo de acesso ao *cache*, uma vez que o mecanismo de *prefetching* consegue ler antecipadamente todos os blocos a serem requisitados.

A maioria dos algoritmos apresentou comportamento constante, independente do tamanho do arquivo utilizado nos testes.

Único algoritmo cujo funcionamento apresentou variações ao longo de tempo da execução foi o algoritmo *aggressive*. Devido à natureza do algoritmo, o tempo de acesso aos blocos aumente a medida que o algoritmo requisita mais blocos, uma vez que ele precisa remover os blocos do *cache* constantemente. Como pode ser visto na figura 31, o algoritmo apresenta um aumento nos tempos de acesso até processar o arquivo completamente e, após isso, os tempos de acesso diminuem para valores menores.

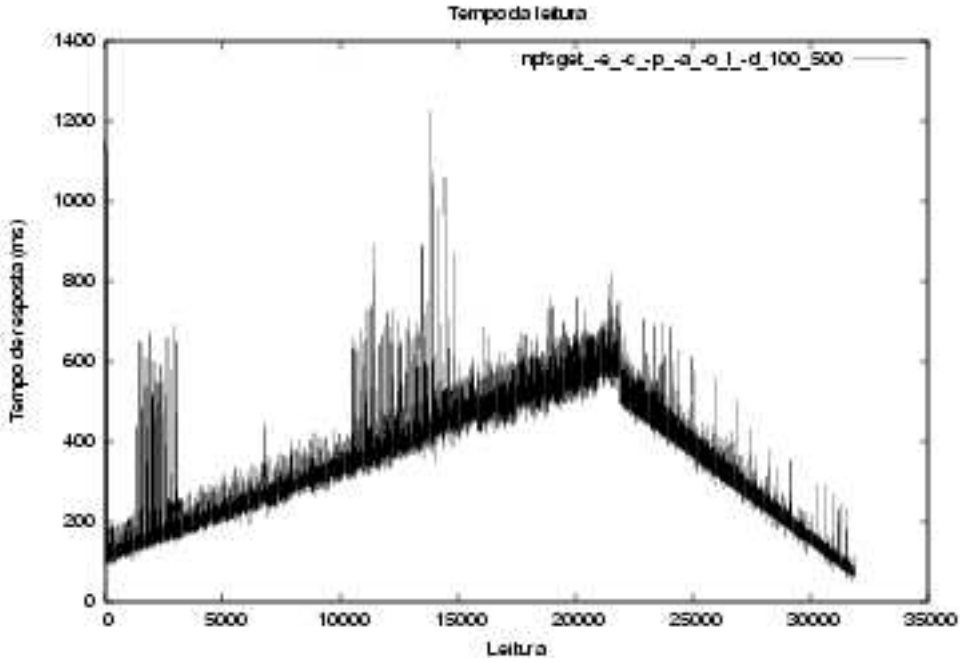

Figura 31: *Prefetching Agressivo*, execução assíncrona 2

Nesta figura, o arquivo lido foi de *500MB*. Como claramente pode ser visto na figura, o tempo de acesso cresceu até o algoritmo de *prefetching agressivo* processar o arquivo completamente e, após isso, os tempos de acesso foram diminuindo.

6.5 Resultados

Como foi visto neste capítulo, o mecanismo de *cache* e *prefetching* proposto ofereceu um ganho de desempenho nas operações de entrada e saída, oferecendo uma latência mais baixa das requisições e uma taxa de transmissão mais alta nos casos estudados.

Os algoritmos de leitura antecipada *prefetch-on-empty* e *limited aggressive*, introduzidos neste trabalho, ofereceram desempenho bom e utilização de *cache* compatível com a do algo-

ritmo de *prefetching agressivo* nos testes realizados. A integração do algoritmo *CPS* ao sistema ofereceu um controle maior sobre os algoritmos de *prefetching* utilizados pela aplicação.

Entretanto, a utilização efetiva dos mecanismos de *cache* e *prefetching* depende da aplicação, uma vez que, como foi visto neste capítulo, em alguns casos os mecanismos de *cache* e *prefetching* prejudicam o desempenho das operações de entrada e saída. O fator principal que limita o funcionamento do mecanismo de *prefetching* é a demora entre as requisições consecutivas de leitura de dados, como foi visto na seção 6.1.

De acordo com os resultados dos testes realizados nesta seção, podemos destacar os seguintes algoritmos, de acordo com a utilização dos mesmos:

- *Aggressive Prefetching* – embora este algoritmo ofereceu a melhor utilização do espaço de *cache*, o tempo de execução dele torna-o inadequado para as classes das aplicações estudadas.

  Uma possível utilização para o algoritmo seria nas aplicações que possuem intervalos entre as requisições superiores ao tempo necessário para efetuar a requisição de dados em si.

- *Passive Prefetching* – este algoritmo ofereceu o melhor tempo de execução, a custo de uma menor utilização do espaço de *cache*.

  Desta forma, este algoritmo é indicado para ser utilizado nas aplicações que precisam de um desempenho maior das operações, mesmo a custo de uma baixa utilização do *cache*.

- *Limited Aggressive* e *Prefetch-on-Empty* – os algoritmos *Limited Aggressive* e *Prefetch-on-Empty* mostraram uma boa utilização de *cache* combinada com um bom tempo de execução.

  Assim, estes algoritmos são indicados para serem utilizados nas aplicações que onde o tempo entre as requisições é suficiente para efetuar as requisições de leitura antecipada.

Como foi visto nos testes feitos, é importante a escolha de um algoritmo de *prefetching* adequado para cada aplicação, levando em conta as particularidades das mesmas. O projeto correto da aplicação, levando em conta a arquitetura da mesma, possibilita um aproveitamento melhor do mecanismo de *cache* e *prefetching*, diminuindo a latência das requisições.

Os principais fatores que influenciam na utilização eficiente do *cache* foram demonstrados nas seções 6.2 e 6.3.

7 Conclusões e trabalhos futuros

Este trabalho apresentou o projeto e a implementação de um mecanismo integrado de *cache* e *prefetching* para sistemas de arquivos paralelos e distribuídos.

Como foi visto nos capítulos iniciais, os sistemas de arquivos paralelos possuem um potencial grande para aumentar o desempenho geral das operações de entrada e saída, aumentando a taxa de transmissão e diminuindo a latência dos acessos. Entretanto, a eficiência de um sistema de arquivos paralelos depende do ambiente utilizado, sendo controlado por algum mecanismo de *hardware* (como, por exemplo, sistemas de *RAID*) ou de *software* (*Software RAID, PFS*). Além disto, é possível a distribuição de dados através de uma *rede de computadores*, oferecendo acesso descentralizado aos dados e uma maior escalabilidade do sistema.

O foco principal deste trabalho foi os sistemas de arquivos paralelos distribuídos em uma rede de computadores, tais como o sistema de arquivos *NPFS*, também apresentado neste trabalho.

A implementação de um sistema de arquivos paralelos numa rede de estações de trabalho demonstrou ter o potencial tanto para aumentar a capacidade de armazenamento de arquivos individuais quanto para incrementar a taxa de transferência nas operações de entrada e saída de dados. Entretanto, o ambiente de rede é, normalmente, compartilhado entre todos os computadores e oferece um desempenho inferior quando comparado aos sistemas locais. Desta forma, a necessidade de uma utilização mais eficiente do meio de comunicação tornou-se evidente.

Visando aumentar a taxa de desempenho nestas operações, diminuindo a latência das requisições, um mecanismo integrado de *cache* e *prefetching* foi proposto. Os mecanismos de *cache* e busca antecipada de dados no disco podem otimizar esses acessos, dependendo dos diferentes padrões de acesso realizados pelas aplicações.

Para determinar o funcionamento mais eficiente dos mecanismos de *caching* e leitura antecipada de dados, a influência do meio de comunicação no funcionamento dos mecanismos propostos foi estudada no trabalho.

No decorrer do trabalho, diversos mecanismos e políticas de *cache* e *prefetching* foram estudados e, baseando-se nos conceitos apresentados, um mecanismo integrado de *cache* e *prefetching* foi implementado. A implementação teve como o foco principal o sistema de arquivos *NPFS*, porém a estrutura dos mecanismos propostos possibilita uma adaptação rápida para outros sistemas de arquivos.

Neste trabalho, foram introduzidos dois algoritmos de *prefetching*, *prefetch-on-empty* e *limited aggressive*, cujo funcionamento foi avaliado.

O algoritmo *prefetch-on-empty* introduzido no trabalho implementa o funcionamento de leitura antecipada em "rajadas", visando alternar seqüências de acesso aos dados com períodos ociosos. Com isso, procura utilizar o meio de comunicação de maneira mais efetiva, possibilitando um funcionamento intercalado de diversos mecanismos de *prefetching*, evitando competições pelo uso do meio de comunicação.

O algoritmo *limited aggressive*, também proveniente do trabalho desenvolvido, procurou juntar a eficiência do algoritmo *agressivo* clássico com o desempenho do algoritmo *passivo*, ambos apresentados no trabalho.

Um diferencial positivo dos algoritmos de *prefetching* desenvolvidos (*prefetch-on-empty* e *limited aggressive*) é o efeito no uso do *cache*, otimizando o desempenho das políticas como *FIFO* e *LRU*, que não é eficiente para padrões de acesso puramente seqüenciais.

Além disto, foi criado um algoritmo de controle dos mecanismos de leitura antecipada, denominado de *CPS*. O algoritmo proposto possibilita uma adaptação rápida às possíveis mudanças nos padrões do acesso dos aplicativos e oferece a possibilidade de adaptação do mecanismo de leitura antecipada à situações diferentes, variando a *agressividade* do mesmo.

Uma análise do comportamento dos mecanismos de *cache* e *prefetching* foi feita no trabalho, procurando determinar a latência mínima necessária para o funcionamento dos mecanismos propostos e a eficiência dos mesmos.

Como foi demonstrado nos testes e estudos de caso, a diminuição da latência nas operações de leitura foi significativa com o uso dos mecanismos propostos. Estudos de casos e testes de desempenho realizados possibilitaram determinar os fatores que limitam a utilização dos mecanismos de *cache* e *prefetching*, tais como o tempo decorrido entre requisições de entrada e saída e padrões de acesso utilizados pelas aplicações.

O trabalho procurou criar um mecanismo de *cache* e *prefetching* adaptativo, visando futura extensão do mesmo com novos algoritmos e técnicas de *cache* e *prefetching*. Para os trabalhos futuros, é prevista a incorporação de novas políticas e algoritmos de *prefetching* e *caching* no

sistema. Além disto, é prevista a extensão dos mecanismos propostos para serem utilizados em ambientes de *GRID*, ou *grades computacionais*. Para isso, é necessário a adequação dos mecanismos criados para um ambiente de *GRID*, composto por servidóres com latência variável.

Novos estudos sobre a utilização de padrões de acesso globais nos clientes e servidores podem auxiliar num possível *prefetching* em cada um deles. Problemas relacionados ao tamanho do *cache* em cada servidor e formas para encontrar possíveis fluxos distrintos devem ser tratados neste caso.

Outros aspectos a serem tratados nos trabalhos subseqüentes são a ativação e desativação automática dos mecanismos de leitura antecipada de dados, baseando-se na taxa de utilização do meio de comunicação e no tempo entre as requisições consecutivas de dados. Por exemplo, a agressividade do *prefetching* poderia ser determinada em função do intervalo entre requisições para a operação assíncrona.

Referências Bibliográficas

1 PATTERSON, D.; GIBSON, G.; KATZ, R. A case for redundant arrays of inexpensive disks (RAID). In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Chicago, IL: ACM Press, 1988. p. 109–116.

2 PATTERSON, D. et al. Introduction to redundant arrays of inexpensive disks (RAID). In: *Proceedings of IEEE Compcon*. [S.l.: s.n.], 1989. p. 112–117.

3 ARUNACHALAM, M.; CHOUDHARY, A.; RULLMAN, B. Implementation and evaluation of prefetching in the Intel Paragon Parallel File System. In: *Proceedings of the Tenth International Parallel Processing Symposium*. [s.n.], 1996. p. 554–559. Disponível em: <http://www.ece.nwu.edu/ meena/papers/ipps.ps>.

4 CORTES, T. Software RAID and parallel filesystems. In: BUYYA, R. (Ed.). *High Peformance Cluster Computing*. [S.l.]: Prentice Hall PTR, 1999. p. 463–496.

5 GUARDIA, H. C.; SATO, L. M. *Soluções Paralelas para Entrada e Saída de Dados com Alto Desempenho*. [S.l.], June 1998.

6 HARTMAN, J. H.; OUSTERHOUT, J. K. Zebra: A striped network file system. In: *Proceedings of the USENIX File Systems Workshop*. [S.l.: s.n.], 1992. p. 71–78.

7 CROCKETT, T. W. File concepts for parallel I/O. In: *Proceedings of Supercomputing '89*. [S.l.: s.n.], 1989. p. 574–579.

8 DEWITT, D.; GRAY, J. Parallel database systems: The future of database processing or a passing fad. *ACM SIGMOD Record, Special Issue on Directions for Future Database Research and Development*, v. 19, n. 4, p. 104, 1990.

9 OLDFIELD, R.; KOTZ, D. *Applications of Parallel I/O*. [S.l.], August 1998. Supplement to PCS-TR96-297. Disponível em: <http://www.cs.dartmouth.edu/reports/abstracts/TR98-337/>.

10 FRENKEL, K. A. The human genome project and informatics. *Commun. ACM*, ACM Press, v. 34, n. 11, p. 40–51, 1991. ISSN 0001-0782.

11 W. T. SULLIVAN, I. et al. A new major seti project based on project serendip data and 100,000 personal computers. *Proc. of the Fifth Intl. Conf. on Bioastronomy*, 1997.

12 FOSTER, I.; KESSELMAN, C.; TUECKE, S. *The Anatomy of the Grid*.

13 CERON, C. et al. Parallel implementation of DNAml program on message-passing architectures. *Parallel Computing*, Elsevier Science, v. 24, n. 5–6, p. 701–716, June 1997. Disponível em: <http://dx.doi.org/10.1016/S0167-8191(98)00002-7>.

14 CHANG, C. et al. Titan: a high-performance remote-sensing database. In: *Proceedings of the Thirteenth International Conference on Data Engineering*. Birmingham, U.K.: [s.n.], 1997. Disponível em: <ftp://hpsl.cs.umd.edu/pub/papers/icde97-final.ps.Z>.

15 LOCKEY, P.; PROCTOR, R.; JAMES, I. D. Characterization of I/O requirements in a massively parallel shelf sea model. *The International Journal of High Performance Computing Applications*, v. 12, n. 3, p. 320–332, Fall 1998.

16 OLDFIELD, R. A.; WOMBLE, D. E.; OBER, C. C. Efficient parallel I/O in seismic imaging. *The International Journal of High Performance Computing Applications*, Sage Science Press, v. 12, n. 3, p. 333–344, Fall 1998. Disponível em: <http://www.cs.dartmouth.edu/ raoldfi/ijsa97>.

17 BREZANY, P.; CHOUDHARY, A.; DANG, M. Parallelization of irregular out-of-core applications for distributed-memory systems. *High-Performance Computing and Networking*, Springer-Verlag, v. 1225, p. 811–820, 1997.

18 FOSTER, I.; NIEPLOCHA, J. *ChemIO: High-Performance I/O for Computational Chemistry Applications*. February 1996. WWW http://www.mcs.anl.gov/chemio/. Disponível em: <http://www.mcs.anl.gov/chemio/>.

19 THAKUR, R.; LUSK, E.; GROPP, W. I/O in parallel applications: The weakest link. *The International Journal of High Performance Computing Applications*, v. 12, n. 4, p. 389–395, Winter 1998. In a Special Issue on I/O in Parallel Applications. Disponível em: <http://www.mcs.anl.gov/ thakur/papers/ijsa-article.ps>.

20 THAKUR, R.; GROPP, W.; LUSK, E. On implementing MPI-IO portably and with high performance. In: *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*. [s.n.], 1999. p. 23–32. Disponível em: <citeseer.nj.nec.com/thakur99implementing.html>.

21 WILLIAM, R. T. *MPI-IO: A Standard, Portable API for High-Performance Parallel IO*. Citeseer.nj.nec.com/414253.html.

22 MOYER, S. A.; SUNDERAM, V. S. PIOUS: A scalable parallel I/O system for distributed computing environments. In: *Proceedings of the Scalable High-Performance Computing Conference*. [s.n.], 1994. p. 71–78. Disponível em: <citeseer.nj.nec.com/article/moyer94pious.html>.

23 GUARDIA, H. C. *Considerações sobre as estratégias de um Sistema de Arquivos Paralelos integrado ao processamento distribuído*. Tese (Doutorado) — EPUSP, 1999.

24 SZEREDI, M. *FUSE - Filesystem in USErspace*. 2002. Http://www.inf.bme.hu/ mszeredi/avfs/.

25 POOL, M. *Newuserfs: a Way to Write Linux Filesystems in Userspace*. 2002. Http://etc.samba.org/newuserfs/.

26 ELECTRONICSTALK. Transceiver provides 10gbit/s connectivity. *http://www.electronicstalk.com/news/iel/iel122.html*, October 2003.

27 Tom's Hardware Guide Russia. Memory timing influence on performance. *http://www.toms-hardware.ru/mainboard/20040119/index.html*, January 2004.

28 VOELKER, G. M. et al. Implementing cooperative prefetching and caching in a globally-managed memory system. In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. ACM Press, 1998. p. 33–43. Disponível em: <http://www.acm.org/pubs/citations/proceedings/metrics/277851/p33-voelker/>.

29 OPERATINGSYSTEMS.NET. *Operating System Technical Comparition*. 2002. Http://www.osdata.com/index.htm.

30 CAO, P. et al. A study of integrated prefetching and caching stategies. In: *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. [S.l.]: ACM Press, 1995. p. 188–197.

31 KOTZ, D.; ELLIS, C. S. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, Academic Press, v. 17, n. 1–2, p. 140–145, January and February 1993. Disponível em: <http://www.cs.dartmouth.edu/ dfk/papers/kotz:jwriteback.ps.Z>.

32 KIMBREL, T.; KARLIN, A. R. Near-optimal parallel prefetching and caching. In: *IEEE Symposium on Foundations of Computer Science*. [s.n.], 1996. p. 540–549. Disponível em: <citeseer.nj.nec.com/12080.html>.

33 KOTZ, D.; ELLIS, C. S. Practical prefetching techniques for parallel file systems. In: *Proceedings of the First International Conference on Parallel and Distributed Information Systems*. IEEE Computer Society Press, 1991. p. 182–189. Disponível em: <http://www.cs.dartmouth.edu/ dfk/papers/kotz:practical.ps.Z>.

34 KIMBREL, T. et al. A trace-driven comparison of algorithms for parallel prefetching and caching. In: *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*. USENIX Association, 1996. p. 19–34. Disponível em: <http://www.usenix.org/publications/library/proceedings/osdi96/kimbrel.html>.

35 FERRARI, R. *Curso de Estruturas de Dados 1999*. [S.l.]: Edição Própria, 1999.

36 REISER, H. Reiserfs 4 whitepaper. *http://www.namesys.com/v4/v4.html*.

37 ROBINSON, J. T.; DEVARAKONDA, M. V. Data cache management using frequency-based replacement. In: *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. [S.l.: s.n.], 1990. p. 134–142.

38 TURNER, R.; LEVY, H. Segmented fifo page replacement. In: *1991 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. [S.l.: s.n.], 1991.

39 NICOLA, V. F.; DAN, A.; DIAS, D. M. Analysis of generalized clock buffer replacement scheme for database transaction processing. In: *SIGMETRICS and PERFORMANCE*. [S.l.: s.n.], 1992.

40 JOHNSON, T.; SHASHA, D. 2q: A low overhead high performance buffer management replacement algorithm. In: *Proceedings of the 20th International Conference on Very Large Databases*. [S.l.: s.n.], 1994. p. 439–450.

41 O'NEIL, E. J.; O'NEIL, P.; WEIKUM, G. The lru-k page replacement algorithm for database disk buffering. In: *Proceedings of the 1993 ACM SIGMOD Conference*. [S.l.: s.n.], 1993. p. 297–306.

42 SMARAGDAKIS, Y.; KAPLAN, S.; WILSON, P. R. EELRU: Simple and effective adaptive page replacement. In: *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. [s.n.], 1999. p. 122–133. Disponível em: <citeseer.nj.nec.com/smaragdakis99eelru.html>.

43 GLASS, G.; CAO, P. Adaptive page replacement based on memory reference behavior. In: *Measurement and Modeling of Computer Systems*. [s.n.], 1997. p. 115–126. Disponível em: <citeseer.nj.nec.com/glass97adaptive.html>.

44 JIANG, S.; ZHUANG, X. *LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance*. 2002. Disponível em: <citeseer.nj.nec.com/jiang02lirs.html>.

45 MEGIDDO, N.; MODHA, D. S. Arc: A self-tuning, low overhead replacement cache. In: *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST 03), San Franciso, CA*. [s.n.], 2003. Disponível em: <citeseer.nj.nec.com/megiddo03arc.html>.

46 REDDY, A. L. N. Evaluation of caching strategies for a multimedia storage server. In: *International Conference on Multimedia Computing and Systems*. [s.n.], 1997. p. 118–125. Disponível em: <citeseer.nj.nec.com/reddy97evaluation.html>.

47 BURNETT, N. C. et al. Exploiting gray-box knowledge of buffer-cache management. In: *Proceedings of the 2002 USENIX Technical Conference*. [S.l.: s.n.], 2002.

48 TATARINOV, I. *Cache Policies for Web Servers*. [S.l.]. 22 p. Disponível em: <citeseer.nj.nec.com/370718.html>.

49 GUSTAFSSON, E.; NILBERT, B. *Cache Coherence in Parallel Multi-processors*. [S.l.], February 1997. Disponível em: <http://www.update.uu.se/~gus/Misc/Compositions/cache_coherence.html>.

50 CORTES, T.; GIRONA, S.; LABARTA, J. PACA: A cooperative file system cache for parallel machines. In: *Proceedings of the 2nd International Euro-Par Conference*. [S.l.: s.n.], 1996. p. I:477–486.

51 CORTES, T.; GIRONA, S.; LABARTA, J. *Avoiding the Cache-Coherence Problem in a Parallel/Distributed File System*. [S.l.], May 1997. Disponível em: <ftp://ftp.ac.upc.es/pub/reports/CEPBA/1996/UPC-CEPBA-1996-13.ps.Z>.

52 CORTES, T.; LABARTA, J. Linear aggressive prefetching: A way to increase the performance of cooperative caches. In: *Proceedings of the Joint International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing*. San Juan, Puerto Rico: [s.n.], 1999. p. 45–54.

53 KOTZ, D.; ELLIS, C. S. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, Kluwer Academic Publishers, v. 1, n. 1, p. 33–51, January 1993.

54 PATTERSON, R. H. et al. Informed prefetching and caching. In: JIN, H.; CORTES, T.; BUYYA, R. (Ed.). *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. New York, NY: IEEE Computer Society Press and Wiley, 2001. cap. 16, p. 224–244. Disponível em: <http://www.buyya.com/superstorage/>.

55 BARVE, R. et al. Competitive parallel disk prefetching and buffer management. In: *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*. San Jose, CA: ACM Press, 1997. p. 47–56. Disponível em: <http://doi.acm.org/10.1145/266220.266225>.

56 KALLAHALLA, M.; VARMAN, P. J. Optimal prefetching and caching for parallel I/O systems. In: *Proceedings of the Thirteenth Symposium on Parallel Algorithms and Architectures*. ACM Press, 2001. p. 219–228. To appear. Disponível em: <http://www.ece.rice.edu/ pjv/spaa.ps>.

57 DRAFT Proposed American National Standard for Information Systems: Small Computer System Interface – 2 (SCSI-2). [S.l.]: Irvine, California: Global Engineering Documents, 1992. (X3T9.2/86-109, revision 10h, October 17, 1991).

58 CORTES, T.; GIRONA, S.; LABARTA, J. *PACA: A Cooperative File System Cache for Parallel Machines*. [S.l.], 1996. Disponível em: <ftp://ftp.ac.upc.es/pub/reports/CEPBA/1996/UPC-CEPBA-1996-7.ps.Z>.

59 BIANCHINI, R.; PINTO, R.; AMORIM, C. L. Data prefetching for software dsms. In: *Proceedings of the 12th international conference on Supercomputing*. [S.l.]: ACM Press, 1998. p. 385–392. ISBN 0-89791-998-X.

60 BIANCHINI, R. et al. Hiding communication latency and coherence overhead in software DSMs. In: *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*. [s.n.], 1996. p. 198–209. Disponível em: <citeseer.nj.nec.com/bianchini96hiding.html>.

61 CORTES, T.; GIRONA, S.; LABARTA, J. Design issues of a cooperative cache with no coherence problems. In: *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*. San Jose, CA: ACM Press, 1997. p. 37–46. Disponível em: <http://doi.acm.org/10.1145/266220.266224>.

62 CORTES, T. *Cooperative Caching and Prefetching in Parallel/Distributed File Systems*. Tese (Doutorado) — UPC: Universitat Politècnica de Catalunya, Barcelona, Spain, 1997. Disponível em: <http://www.ac.upc.es/homes/toni/thesis.html>.

63 SPESSOTO, E. A. *Um Mecanismo de Ajuste Dinâmico para Diminuição de Latência e do Tempo de Resposta em Sistemas de Arquivos Paralelo*. Dissertação (Mestrado) — DC/UFSCar, June 2003.

64 COSTA NETO, J. C. da; GUARDIA, H. C.; SATO, L. M. Integração de um banco de dados e uma data warehouse sobre um sistema de arquivos paralelos. In: *II Workshop de Sistemas Computacionais de Alto Desempenho*. Pirenólopil, Brazil: [s.n.], 2001. p. 111–118.

65 LARA, C. R. F. *Projeto de um Servidor de Vídeo Sob Demanda Paralelo e Distribuído*. Dissertação (Mestrado) — DC/UFSCar, June 2003.

66 MPLAYER: The Movie Player for Linux. *http://www.mplayerhq.hu/*.

67 KOTZ, D.; ELLIS, C. S. Practical prefetching techniques for multiprocessor file systems. In: JIN, H.; CORTES, T.; BUYYA, R. (Ed.). *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. New York, NY: IEEE Computer Society Press and John Wiley & Sons, 2001. cap. 17, p. 245–258.

68 KOTZ, D.; NIEUWEJAAR, N. Flexibility and performance of parallel file systems. *ACM Operating Systems Review*, ACM Press, v. 30, n. 2, p. 63–73, April 1996. Disponível em: <http://www.cs.dartmouth.edu/ dfk/papers/kotz:flexibility.ps.Z>.

69 KOTZ, D. *Bibliography about Parallel I/O*. 1994–2000. Available on the WWW at `http://www.cs.dartmouth.edu/pario/bib/`. Disponível em: <http://www.cs.dartmouth.edu/pario/bib/>.

70 KOTZ, D. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors*. Tese (Doutorado) — Duke University, April 1991. Available as technical report CS-1991-016. Disponível em: <http://www.cs.dartmouth.edu/ dfk/papers/kotz:thesis.ps.Z>.

71 KOTZ, D.; NIEUWEJAAR, N. Dynamic file-access characteristics of a production parallel scientific workload. In: *Proceedings of Supercomputing '94*. Washington, DC: IEEE Computer Society Press, 1994. p. 640–649. Disponível em: <http://www.cs.dartmouth.edu/ dfk/papers/kotz:workload.ps.Z>.

72 NIEUWEJAAR, N. et al. *File-Access Characteristics of Parallel Scientific Workloads*. [S.l.], August 1995. Disponível em: <http://www.cs.dartmouth.edu/reports/abstracts/TR95-263/>.

73 CORTES, T.; GIRONA, S.; LABARTA, J. Avoiding the cache-coherence problem in a parallel/distributed file system. In: *Proceedings of High-Performance Computing and Networking*. [S.l.: s.n.], 1997. p. 860–869.

74 KURATTI, A.; SANDERS, W. H. Performance analysis of the RAID5 disk array. In: *Proceedings of the IEEE International Computer Performance and Dependability Symposium*. Erlangen, Germany: [s.n.], 1995. p. 236–245. Disponível em: <citeseer.nj.nec.com/kuratti95performance.html>.

75 TANNENBAUM, A. S. *Modern Operating Systems, 2nd edition*. [S.l.]: Prentice Hall, 2001.

76 TIERNEY, B. L. et al. A network-aware distributed storage cache for data-intensive environments. In: *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*. Redondo Beach, CA: IEEE Computer Society Press, 1999. p. 185–193. Disponível em: <http://computer.org/conferen/proceed/hpdc/0287/02870033abs.htm>.

77 FITZGERALD, S. et al. A directory service for configuring high-performance distributed computations. In: *Proc. 6th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1997. p. 365–375. Disponível em: <citeseer.nj.nec.com/fitzgerald97directory.html>.

78 FOSTER, I.; KESSELMAN, C. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, v. 11, n. 2, p. 115–128, Summer 1997. Disponível em: <citeseer.nj.nec.com/foster96globu.html>.

79 PAI, V. S.; DRUSCHEL, P.; ZWAENEPOEL, W. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, v. 18, n. 1, p. 37–66, 2000. Disponível em: <citeseer.nj.nec.com/pai00iolite.html>.

80 FREEBSD. The freebsd project. *http://www.freebsd.org/*.

81 SOLOVIEV, V. V. Prefetching in segmented disk cache for multi-disk systems. In: *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*. Philadelphia: ACM Press, 1996. p. 69–82.

82 PARAGON XP/S Product Overview. 1991.

83 THELEN, E. Intel paragon. *http://ed-thelen.org/comp-hist/intel-paragon.html*.

84 KELLER, R. Threads vs processes. *http://www.cs.hmc.edu/courses/2001/spring/cs156/htmlthreads/*, February 2001.

85 FIELDING, R. et al. *RFC2616: Hypertext Transfer Protocol – HTTP/1.1*. jun. 1999.

86 APACHE Foundation. Apache http server project. *http://httpd.apache.org/*, January 2004.

87 ACME Labs Software. thttpd - tiny/turbo/throttling http server. *http://www.acme.com/software/thttpd/*, December 2003.

88 BRANDENBURG, K.; STOLL, G. The iso/mpeg-audio codec: A generic standart for coding of high quality digital audio. In: *Journal of AES, Vol. 42, No. 10*. [S.l.: s.n.], 94.

89 Xiph.org Foundation. *Vorbis 1 specification*. 2003. Http://www.xiph.org/ogg/vorbis/doc/Vorbis_I_spec.html.

90 NULLSOFT shoutcast. *http://www.shoutcast.com/*.

91 ICECAST mpeg-layer 3 streaming server. *http://www.icecast.org/*.

92 XVID DivX codec. *http://www.xvid.org/*.

93 TEAMSOLUTIONS. Streaming multimedia data. *http://www.teamsolutions.co.uk/streaming.html*.

94 STREAMING-LIST.NET. Streaming multimedia industry directory. *http://www.streaming-list.net/*.