Suporte Transaccional para o Sistema de Ficheiros Btrfs

João Eduardo Luís, João M. Lourenço, and Paulo A. Lopes*

CITI — Departamento de Informática,
Universidade Nova de Lisboa, Portugal
jel18287@fct.unl.pt {Joao.Lourenco,pal}@di.fct.unl.pt

Resumo Em caso de falha abrupta de um sistema, é imperativo garantir a consistência do Sistema de Ficheiros (SF). Actualmente existem várias soluções que visam garantir que tanto os dados como os metadados do SF se encontram num estado consistente, mas que não contemplam a garantia de consistência dos dados do ponto de vista das aplicações. Por exemplo, aplicações que pretendam alterar vários ficheiros de configuração terão de encontrar mecanismos para garantir que, ou todos os ficheiros são devidamente alterados, ou nenhum o é, evitando assim que numa situação de falha o conteúdo dos ficheiros fique inconsistente. Do ponto de vista da aplicação, pode não ser simples implementar este comportamento sobre um SF típico; e pode também não ser razoável utilizar um Sistema de Gestão de Bases de Dados (SGBD), que oferece propriedades ACID. Neste artigo propomos, testamos e avaliamos uma integração das propriedades ACID num SF. Partindo do suporte para snapshots do sistema de ficheiros Btrfs, oferece-se uma semântica transaccional às aplicações que operam sobre volumes (sub-árvores) do SF, sem comprometer a semântica POSIX do SF.

Palavras-Chave: Transacções; Sistema de Ficheiros; Linux; Kernel

1 Introdução

Os Sistemas de Ficheiros (SF) modernos incluem vários mecanismos que visam garantir a sua própria consistência em caso de falha abrupta do sistema. Ainda que seja essencial para uma aplicação que o SF se mantenha consistente, os requisitos de consistência de uma aplicação vão para além da consistência do SF. Por exemplo, enquanto que do ponto de vista do SF uma operação se poderá reduzir à escrita ou leitura de um bloco, para a aplicação uma operação poderá ser um vasto conjunto de leituras e escritas, sobre um ou mais ficheiros diferentes.

* Este trabalho foi parcialmente suportado pela Sun Microsystems ao abrigo do *Sun Worldwide Marketing Loaner Agreement #11497*, pelo Centro de Informática e Tecnologias da Informação (CITI), e pela Fundação para a Ciência e Tecnologia (FCT/MCTES) no âmbito dos projectos PTDC/EIA/74325/2006, PTDC/EIA-EIA/108963/2008, e PTDC/EIA-EIA/113613/2009.

Ou seja, os SF nada garantem às aplicações cujo correcto funcionamento depende de uma visão lógica e consistente dos seus dados.

Uma aplicação que depende da correcta alteração de um conjunto de ficheiros de configuração, sem os quais não se poderá garantir o seu correcto funcionamento, que seja alvo de uma falha abrupta a meio da actualização deste conjunto de ficheiros, poderá não ser capaz de ser reiniciada quando o sistema voltar a funcionar. Inclusivamente, poderá ser necessária a reposição de uma cópia de salvaguarda, algo que seria desnecessário se a aplicação tivesse a garantia de atomicidade sobre o conjunto de operações a realizar nos múltiplos ficheiros.

O problema da consistência dos dados produzidos por uma aplicação poderá ir além do exemplo apresentado. Uma aplicação poderá ter necessidade de aceder a ficheiros em concorrência com outras aplicações, ou ser ela própria concorrente. Nestes casos, será imperativo recorrer a técnicas de controlo de concorrência, o que no SF geralmente se resume à utilização de *file locking*, um mecanismo de controlo de baixo nível que não é trivial de codificar correctamente. Além do mais, se aplicações diferentes optarem por diferentes mecanismos de controle de concorrência, eles serão inúteis fora o âmbito de cada aplicação. Por tudo isto, a oferta, por parte de um SF, de mecanismos que permitam a cada aplicação executar como se fosse a única a usar o SF simplificaria em muito a gestão de concorrência. Contudo deverá ser garantido que a visão que uma aplicação tem do SF deverá transitar de um estado inicial consistente, para um estado final também ele consistente, sendo os estados intermédios apenas visíveis pela aplicação que os gera. É também desejável que todas as alterações que a aplicação faça no SF, e que possam ser consideradas como tendo sucesso, sejam efectivamente aplicadas e de forma tão durável quanto possível.

Este artigo propõe o TxBtrfs, um Sistema de Ficheiros Transaccional para o Linux, a ser implementado como um módulo do Kernel, que garante às aplicações as propriedades ACID nas operações sobre o SF através de uma interface explícita disponibilizada ao programador. Relativamente a trabalhos similares desenvolvidos no passado, a nossa proposta, que usa o Btrfs [1] como base para a implementação, apresenta as seguintes contribuições: usa versões standard do núcleo do Linux; não necessita de recorrer a software externo; e apresenta *overheads* pequenos para a grande generalidade das operações.

O restante artigo estará organizado da seguinte forma: na Secção 2 definiremos o contexto em que se insere o nosso trabalho; a Secção 3 abordará a importância do Virtual File System (VFS) e introduziremos conceitos chave sobre o Btrfs; na Secção 4 apresentaremos a nossa implementação, de seguida discutindo os nossos resultados preliminares na Secção 5; por fim, a Secção 6 concluirá o artigo com o nosso trabalho para o futuro mais imediato.

2 Contexto

2.1 ACID e o Sistema de Ficheiros

As necessidades de consistência discutidas anteriormente poderiam ser supridas pela semântica transaccional associada às propriedades ACID dos SGBDs:

Atomicidade, garantindo que todas as operações são aplicadas com sucesso ou nenhuma o é; *Consistência*, dado que o SF transitará sempre entre dois estados consistentes; *Isolamento*, uma vez que a aplicação poderá aceder ao SF como se fosse a única a aceder-lhe; e *Durabilidade*, garantindo que os dados ficam preservados de forma permanente.

As aplicações poderão, certamente, recorrer directamente a um SGBD para guardar neste os dados pretendidos, beneficiando da semântica transaccional, do controlo simplificado de concorrência e das garantias de consistência oferecidas. Contudo, i) um qualquer SGBD terá de estar disponível no sistema; e ii) o seu *overhead* adicional deverá ser tido em conta, ainda que versões "leves", como o SQLite [2] que apenas permite que um processo de cada vez escreva na BD, possam atenuar o referido *overhead*. Por outro lado, é necessário ponderar as vantagens obtidas face ao esforço necessário na alteração da aplicação, que deixará de guardar os seus dados em ficheiros, passando a guardá-los em tabelas, e que terá de ser substancialmente alterada em tudo o que se refere a Entradas/Saídas, o que, de facto, corresponde a uma mudança substancial de paradigma.

Também será possível oferecer um modelo transaccional às aplicações recorrendo a um SO que o suporte. Um exemplo é o TxOS [8], implementado com base no Linux Kernel, que dispõe de uma API de sistema que oferece a semântica transaccional aos processos que desta pretendam tirar partido, sem deixar contudo de suportar processos "tradicionais" não-transaccionais. De acordo com os seus autores, implementar sobre o TxOS uma semântica transaccional num sistema de ficheiros como o *ext3* resume-se a umas poucas centenas de linhas de código adicionais. Contudo, não deixa de ser necessário recorrer a uma versão modificada do núcleo do SO.

Em alternativa, a semântica pretendida pode ser assegurada directamente pelo Sistema de Ficheiros, confinando a este as alterações ao núcleo. Existem variadas implementações com este propósito, ainda que seriamente limitadas: umas apenas suportam um modelo *single-writer/multiple-readers* e com níveis de isolamento mais relaxados [5]; outras recorrem internamente a SGBDs [6, 7]; outras ainda poderão apresentar *overheads* inaceitáveis para as aplicações [10].

3 Virtual File System

Do ponto de vista das aplicações, cedo se tornou evidente que nenhum SF podia, por si só, satisfazer da mesma forma todas as suas necessidades. Assim, alguns SOs passaram a suportar múltiplos SFs, uns oferecendo bom desempenho, outros acesso a dados remotos, outros ainda servindo para promover a interoperabilidade entre diferentes SOs. A necessidade de suportar múltiplos SFs num SO levou ao desenvolvimento de um único "modelo interno" de SF (baseado nas abstrações disponibiliadas pelo Unix: superblocos, inodes, blocos de dados, directórios, etc.), que implementa uma a camada de software "genérico" na qual todas as implementações particulares de SFs se poderiam posteriormente "acoplar", como mostra a Figura 1. Nasce assim o VFS (inicialmente designado Vir-

tual Filesystem Switch [4], hoje apenas Virtual File System). O VFS é, assim, a "porta de entrada" para todos os acessos ao SF, sejam estes solicitados via *system calls* ou via bibliotecas de suporte/APIs "de nível superior", e.g., com suporte de *buffering*, conversões de formatos, etc.

No Linux, a complexidade do módulo VFS resulta não apenas do seu desenho e realização genéricos, mas ainda da forte interacção com o subsistema de memória virtual (VMM), necessária para suportar a "page cache". A realização da nossa proposta de adicionar semântica transaccional ao sistema de ficheiros, traduz-se em alterações a um SF concreto, neste caso o Btrfs, que é acessível através do VFS. A decisão usar o Btrfs como ponto de partida deve-se principalmente ao seu suporte nativo para *snapshots*, o que nos permite concentrar esforços na definição e desenvolvimento dos mecanismos necessários para o suporte de transacções.

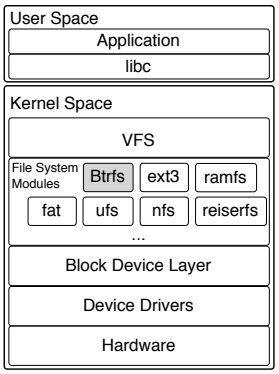


Figura 1: Camadas de Abstracção

3.1 Desde o Utilizador até ao Sistema de Ficheiros

Quando uma aplicação acede a ficheiros, seja usando directamente chamadas ao sistema, como o `read(2)`, seja usando APIs/bibliotecas de "nível superior", como o `fscanf(3)`, a execução atravessa necessariamente o VFS. Aqui, a informação solicitada pela aplicação pode ser imediatamente disponibilizada se se encontrar em *cache* ou, caso contrário, serão chamadas funções específicas do SF "concreto" onde o ficheiro acedido está localizado, sendo que estas poderão acabar por chamar as funções do núcleo para acesso a periféricos orientados ao bloco, e estas os *device drivers* do periférico específico.

Num SF, a introdução de novas funcionalidades a um nível inferior (por exemplo, Btrfs) ao invés de na camada de topo (VFS) acarreta uma complexidade adicional: quando uma função do Btrfs é chamada, nem sempre é possível saber qual foi o serviço que realmente lhe deu origem, uma vez que o caminho inverso não é único. Para a adição do paradigma transaccional ao Btrfs torna-se portanto necessário fazer o registo de muitas informações que, tendo conhecimento de qual o serviço que lhe deu origem, seriam desnecessárias.

3.2 Transacções e Snapshots

O Btrfs suporta nativamente os conceitos de *subvolume* e *snapshot*. O primeiro deve ser visto como uma sub-árvore independente, pertencente ao Sistema de Ficheiros, que apresenta todas as características de um Sistema de Ficheiros singular: tem uma política de atribuição de números de *inodes* própria, é representado por uma árvore (em disco e memória) independente da do Sistema de

Ficheiros original, e que pode inclusivamente ser montada directamente sem necessitar de partilhar o *mountpoint* do Sistema de Ficheiros em que se inclui. Por sua vez, um snapshot é em tudo igual a um subvolume, excepto num detalhe: um subvolume é criado vazio, enquanto que um snapshot é uma cópia lógica de um subvolume. A árvore de um sistema de ficheiros baseado no Btrfs pode facilmente ser representada por subvolumes e snapshots, tomando a raíz da árvore como o *mountpoint* do Btrfs. Como alternativa à criação de múltiplas partições, montadas por exemplo em `/usr` ou `/home`, podemos simplesmente criar subvolumes para cada uma dessas localizações. Também podemos optar por criar apenas um directório `/home` e neste criar subvolumes individuais para cada utilizador. Contudo, se cada utilizador partilhar grande parte da estrutura original do seu *home directory*, podemos então criar somente um subvolume inicial, a ser usado por omissão para a criação dos *home directories*, e fazer um snapshot deste para cada utilizador adicional. Neste caso, `/home/alice` e `/home/bob` poderiam ser snapshots de um subvolume que representaria o estado por omissão de cada *home directory*, permitindo no entanto que os conteúdos de cada utilizador pudessem divergir do estado inicial.

Internamente, um snapshot é apenas uma árvore diferente que inicialmente partilha os dados e metadados do subvolume do qual é originário, não existindo duplicação de informação em disco. Tirando partido de uma técnica de *Copy-on-Write*, o Btrfs é capaz de copiar apenas as informações associadas a múltiplas árvores (tipicamente entre múltiplos snapshots ou entre snapshots e subvolumes) quando estas são alteradas. Os seus mecanismos internos garantem ainda que nenhuma informação é destruída enquanto existir pelo menos uma árvore que dela faça uso.

Estas propriedades do Btrfs tornam-no ideal para a implementação de um Sistema de Ficheiros transaccional: os snapshots podem ser usados para garantir o isolamento das transacções, existindo um snapshot associado a cada transacção iniciada no Sistema de Ficheiros. Ao mesmo tempo, simplificam o cancelamento de transacções, não sendo necessário qualquer operação de compensação, uma vez que cada transacção trabalha sobre a sua cópia lógica dos dados: caso uma transacção seja cancelada, apenas será necessário remover o seu snapshot do Sistema de Ficheiros. Na Secção 4 discutiremos em mais detalhe como tirar partido do Btrfs para a implementação de um Sistema de Ficheiros Transaccional.

3.3 POSIX

Uma vez que a nossa implementação é puramente baseada no Btrfs, não alterando o comportamento das chamadas ao sistema, o seu cumprimento da norma POSIX é exactamente o mesmo que o Btrfs. Tal conclusão é sustentada pela satisfação de todos os testes do pacote *pjd-fstests* [9] satisfeitos pela versão 0.19 do Btrfs, i.e., todos excepto um teste relativo ao `truncate(2)`.

4 *Transactional Btrfs* (TxBtrfs)

O TxBtrfs introduz o conceito de *Subvolume Transaccional* (TxSv), que deriva de um subvolume Btrfs. Como este último não pode abarcar a totalidade de um sistema de ficheiros, i.e., não pode englobar a própria raíz, o mesmo acontece com um subvolume transaccional. Esta particularidade é uma vantagem, pois permite-nos suportar subvolumes não-transaccionais a par com subvolumes transaccionais.

Uma vez que nos comprometemos a manter a compatibilidade com a norma POSIX, decidimos que a forma mais correcta de programar com transacções seria usar *ioctl's* para especificar cada operação, seja esta de inicio, final ou cancelamento de uma transacção. Não desejando, contudo, sobrecarregar o programador com as peculiaridades de cada *ioctl*, criamos uma biblioteca para lidar com estas operações.

4.1 Criação de Subvolumes Transaccionais

Um subvolume transaccional é, na prática, criado exactamente da mesma forma que um subvolume normal, i.e., usando uma ferramenta (aplicação de nível de utilizador) criada para gerir e administrar Sistemas de Ficheiros Btrfs. A única diferença visível ao utilizador/administrador é que um TxSv, após ter sido criado, é explicitamente marcado como sendo um subvolume transaccional. Estas operações, assim como tantas outras suportadas pela ferramenta, interagem com o Sistema de Ficheiros através da execução da chamada de sistema `ioctl(2)`, cujos parâmetros são usados para passar ao SO e módulo TxBtrfs as opções seleccionadas pelo administrador.

4.2 Transparência no Acesso

Um dos nossos objectivos é o de permitir a cada processo aceder, de forma transparente, ao snapshot associado à cada transacção. O processo não deve saber que está a aceder a um snapshot, nem sequer ter forma de o saber. Outros processos, alheios à transacção em curso, não poderão, de forma alguma, ter acesso ao snapshot dessa transacção. Tudo isto é garantido pelo TxBtrfs: quando um processo que acede a ficheiros existentes num subvolume transaccional cria uma transacção, todos os acessos desse processo serão realizados sobre um snapshot (desse volume) associado à referida transacção. Esta correspondência acontece no momento em que se realiza o *lookup* no núcleo do Linux; quando o kernel tenta resolver um caminho (*path*), seja com o objectivo de obter um *file descriptor* ou o de executar uma chamada *path-based*, o VFS percorre cada componente do caminho, nomeadamente ficheiros, directórios, *symbolic links*, etc. Se se conclui que o ficheiro-alvo se encontra num subvolume transaccional, então, i) se não existe transacção em curso, o acesso é negado; ii) se o acesso é consequência de um inicio de transacção (`txbtrfs_start()`), é criado um snapshot que fica associado ao processo; ou iii) se o acesso provém de um processo com transacção iniciada e activa, o acesso é mapeado no snapshot associado à referida transacção.

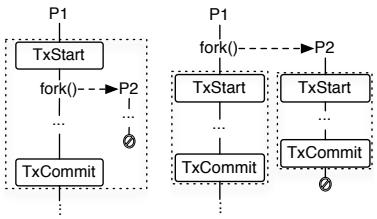
Para aumentar a rapidez dos *lookups*, o VFS mantém uma cache para a resolução de nomes, a *dcache* (contracção de *dentry cache*), que vai sendo povoada com os componentes dos caminhos percorridos, evitando-se assim repetidos acessos ao subsistema de ficheiros "concreto" (neste caso ao Btrfs) cada vez que um componente já está em cache.

A existência da *dcache* surgiu como o nosso primeiro desafio ao implementar acessos transparentes ao subvolume transaccional: após a primeira resolução do nome do subvolume, o VFS deixaria de inquirir o SF. Invalidar a entrada na *dcache* a cada acesso violaria o propósito da *dcache*, pelo que não foi considerado. Inserir múltiplas entradas na *dcache* para cada um dos snapshots existentes também pouco efeito teria, pois seria retornada sempre a mesma entrada. Vimo-nos assim obrigados a tornar cada snapshot numa entidade real do SF, partilhando o mesmo directório pai que o subvolume transaccional, com um formato de nome próprio. Para um subvolume transaccional de nome *Sv*, o snapshot criado pelo processo de PID 31337 teria o nome *Sv#31337*. Assim, aquando do *lookup* ao SF, alteramos simplesmente o nome que está a ser inquirido de *Sv* para *Sv#31337* e deixamos o VFS e o Btrfs prosseguirem o seu caminho. Garante-se, contudo, que a listagem das entradas do directório onde subvolume transaccional está montado e os snapshots se encontram não produz qualquer indício da sua existência.

4.3 Processos e Transacções

Quando é iniciada uma transacção sobre o SF, o processo e transacção ficam associados com um novo snapshot. No entanto admite-se que uma transacção e o seu respectivo snapshot possam estar associados a múltiplos processos. Tal situação é apenas possível em três casos, todos envolvendo um processo e seus filhos, e visa manter as "expectativas habituais" da semântica operacional de partilha de recursos.

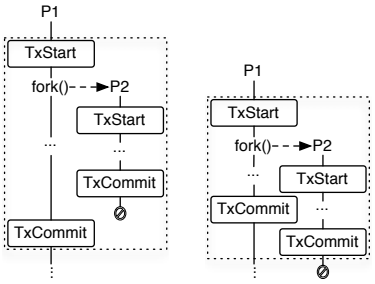
O primeiro caso é aquele no qual um processo inicia uma transacção e posteriormente invoca um `fork(2)`: o processo filho resultante também executará no âmbito da transacção do pai quando aceder ao subvolume transaccional (ver Figura 2a). Desta forma, um processo filho nunca entrará em conflito (na perspectiva transaccional) com o seu progenitor. Se por outro lado o `fork(2)` ocorrer antes do inicio da transacção, o processo filho resultante já não estará abrangido pela transacção do pai, e irá executar no contexto de uma transacção própria (ver Figura 2b).



(a) `fork(2)` invocado no contexto de uma transacção (b) Transacções executando em paralelo

Figura 2: Invocação de um `fork(2)` em diferentes contextos

O segundo caso é aquele em que o processo filho, criado após o pai ter iniciado uma transacção, inicia ele mesmo uma nova transacção. Como o filho partilha a transacção do pai, estabelece-se que qualquer transacção por si iniciada (ou por qualquer um dos seus descendentes) irá ser considerada como *aninhada* segundo o modelo *flat nesting*, i.e., todas as operações feitas na transacção aninhada são parte integrante da transacção do progenitor. Como exemplo veja-se a Figura 3a, na qual *P2* cria uma transacção aninhada dentro do contexto de *P1*, onde o comportamento dos processos será idêntico ao ilustrado na Figura 2a.



(a) Transacções Aninhadas (b) Transacções Intercaladas

Figura 3: Diferentes âmbitos de transacção

Considere-se finalmente o terceiro caso, ilustrado na Figura 3b, no qual o processo filho *P2* também inicia uma transacção no contexto da transacção criada pelo seu pai, *P1*. Contudo, *P1* termina a transacção exterior/envolvente antes de *P2* terminar a sua transacção aninhada. Para eliminar os problemas potencialmente criados por esta situação, consideramos que a transacção apenas será realmente finalizada quando o número de *commits* invocados for igual ao número de *starts*. Na prática, é mantido um contador na transacção que é incrementado a cada início de transacção, e decrementado cada vez que é invocado um final de transacção, sendo a transacção de facto finalizada apenas quando o contador chegar a zero.

Uma vez que a nossa implementação toma como parte integrante da transacção inicial todas as transacções iniciadas pelos filhos, desde que estes tenham sido criados após o inicio da transacção, consideramos como má prática o acesso ao subvolume transaccional em casos como aquele representado na Figura 2a. Sempre que um processo filho deseja aceder ao subvolume transaccional, mesmo que se saiba de antemão que existe uma transacção iniciada pelo pai, o filho deve iniciar uma transacção aninhada. O caso apresentado na Figura 4, na qual o processo *P2* tenta aceder ao subvolume transaccional após a transacção ser finalizada, sustenta esta argumentação. Neste caso, o acesso de *P2* ao subvolume transaccional será negado, uma vez que a transacção herdada do pai foi concluída e o seu snapshot destruído. No entanto, este problema não existiria caso *P2* tivesse iniciado uma transacção aninhada, vindo a concluí-la após *P1* invocar a finalização da sua transacção, uma vez que tal recairia no exemplo ilustrado previamente na Figura 3b. Caso o programador deseje, de facto, não iniciar uma transacção no processo filho (por exemplo, para poder fazer um *fork-exec* de outra aplicação), deverá garantir que o processo que inicia a transacção espera

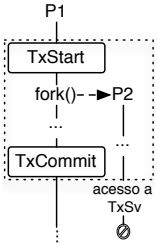


Figura 4: Acesso erróneo a TxSv por parte de *P2*

por qualquer filho que desta possa fazer uso antes de a finalizar a transacção exterior/envolvente.

4.4 Semântica de Partilha

Assuma-se a existência de dois processos independentes, *P1* e *P2*, cada um executando uma transacção num snapshot privado. Assuma-se também que, durante as suas transacções, ambos os processos acedem a um ficheiro *F*. Estando a executar em isolamento, cada transacção no seu snapshot, as suas alterações só serão visíveis fora dos seus snapshots se as suas transacções fizerem *commit* com sucesso. Essas alterações introduzidas pelo commit também só serão visíveis para as transacções que iniciem a partir desse instante, pelo que quaisquer transacções já em execução não terão acesso às alterações feitas.

Contudo, se *P1* fizer commit com sucesso, quando *P2* tentar fazer commit será necessário proceder à detecção de conflitos entre as transacções de ambos os processos e a uma eventual reconciliação. Se tanto *P1* como *P2* tiverem alterado *F*, os processos entrarão em conflito somente se ambos tiverem lido e escrito em zonas sobrepostas, incluindo blocos de dados ou metadados dos ficheiros e descritores de directórios. Nesse caso a reconciliação poderá não ser possível, uma vez que o estado do processo poderá depender de dados previamente lidos. No entanto, se um *P1* tiver lido e escrito e *P2* apenas tiver lido ou escrito, então poderá ser viável tomar *P2* como tendo acontecido no passado, não se considerando a existência de qualquer conflito.

O tópico da reconciliação de snapshots está actualmente em análise pelos autores e ainda não é contemplado no protótipo existente.

4.5 Registo de Escritas e Leituras

Uma vez que cada transacção é executada em completo isolamento relativamente às restantes operações no sistema e sobre uma cópia privada, no momento de finalizar a transacção será necessário aplicar ao subvolume transaccional as alterações entretanto efectuadas. Para que tal seja exequível é necessário que cada transacção mantenha um registo das operações de leitura e escrita efectuadas no seu snapshot.

O TxBtrfs regista portanto qualquer operação submetida ao Sistema de Ficheiros, desde que esta esteja no contexto de uma transacção que acede ao seu snapshot. Este registo é efectuado por transacção, associado a um momento no tempo (mantido por um relógio lógico interno à nossa implementação), sendo mantidos dois registos diferentes: um de leituras, outro de escritas. Acessos a um snapshot transaccional fora de um contexto transaccional serão tacitamente ignorados e irão retornar um código de erro.

Como o grão das operações dos SF concretos é muito mais fino do que aquele em que o VFS trabalha, não é possível fazer corresponder à operação sobre o SF concreto qual a operação desencadeada pelo utilizador que lhe deu origem. Por exemplo, um `chmod(2)` pode chegar ao Btrfs como um conjunto de operações mais elementares. Assim, registamos todas aquelas que têm por alvo ficheiros ou

directórios, por forma a, aquando do *commit*, dispormos de toda a informação necessária para detectar potenciais conflitos entre transacções.

4.6 Commit de Transacções e Reconciliação de snapshots

Embora o nosso objectivo final seja fornecer uma semântica transaccional completa, a versão actual do TxBtrfs ainda não implementa esta fase. Num futuro próximo desejamos ser capazes de validar os registos de cada transacção aquando da sua finalização, com o intuito de verificar potenciais conflitos entre estas e quaisquer alterações que possam ter sido feitas ao subvolume transaccional. Pretendemos ainda garantir que, após verificar que uma transacção entra em conflito, é possível ao TxBtrfs tentar reconciliá-la, sem sacrificar a semântica do Sistema de Ficheiros, de modo a não necessitar de re-executar a transacção.

5 Validação

Como a versão actual do TxBtrfs ainda não valida as transacções, o processo de validação consiste, de momento, em aferir o impacto que toda a transparência no mapeamento dos processos nos snapshots e o registo copmleto das operações têm no desempenho do Sistema de Ficheiros. Os testes foram executados numa máquina com Debian Linux 6.0, com um processador dual-core i5 650 a 3.20 GHz, 4GB de RAM e um disco rígido SATA Samsung P80 SD.

Assim, executou-se o benchmark *IOzone* [3], com os testes automáticos para leitura e escrita, sobre uma versão pura do Btrfs (v0.19) e sobre o TxBtrfs. No caso da execução sobre TxBtrfs, todo o benchmark foi executado no contexto de uma única transacção. Cada teste do benchmark consiste no acesso a ficheiros com tamanhos entre 64KB e 4GB usando registos cuja dimensão varia de 4KB a 16MB. Os resultados obtidos para a largura de banda (LB) vêm expressos em KB/s. O limite superior de 4GB para o ficheiro foi imposto por nós, uma vez que por omissão o *IOzone* só cria ficheiros até 512MB. Este limite superior alternativo, igual à dimensão da RAM da máquina, visa garantir que o teste executa realmente operações de I/O que exercitam o disco físico, não se ficando apenas pela *page cache* do Linux.

De forma a sintetizar os resultados e calcular o *overhead* imposto, fizemos a média dos valores obtidos em execuções sucessivas (forçando a invalidação da *page cache* entre execuções), tanto para o Btrfs como para o TxBtrfs.

Em ambos os testes, verificou-se que tanto o Btrfs como o TxBtrfs obtêm resultados na ordem dos GB/s quando os ficheiros são de dimensão inferior a 2GB. Tal fenómeno é facilmente explicado pela capacidade do Linux Kernel em manter esses ficheiros na *page cache*. Para ficheiros de 4GB, a situação inverte-se completamente, passando os resultados a serem entre os 40 MB/s e os 70 MB/s. Verificámos também que os acessos a um ficheiro de 2GB ora exibem largura de banda na ordem dos GB/s (indiciando que este coube na *page cache*) ora exibem largura de banda em tudo similares à obtida para ficheiros de 4GB. Para

ficheiros de 2GB só foram considerados os "runs" que exibiram largura de banda consentâneas com acessos ao disco, i.e., da ordem dos MB/s.

Na Tabela 1 apresentamos o *overhead* do TxBtrfs face à versão do Btrfs do qual partiu o desenvolvimento, tanto para leituras como para escritas. Seguindo o que foi discutido anteriormente, ao invés de enunciar os tamanhos dos ficheiros testados, optámos por usar a notação *Cache* e *Disco*, uma vez que as diferenças de resultados (para cada uma das categorias) só são relevantes de acordo com o tamanho do registo usado. *Overheads* negativos indicam que o TxBtrfs obteve um melhor desempenho, o que só se pode justificar por variações no estado do sistema, pois o *overhead* do TxBtrfs deveria ser sempre positivo.

Tabela 1: Overhead do TxBtrfs face ao Btrfs, em percentagem.

	Overhead de Leituras (%)						Overhead de Escritas (%)					
Tamanho	Cache			Disco								
do Registo	Mín	Méd	Máx	Mín	Méd	Máx	Mín	Méd	Máx	Mín	Méd	Máx
4K-64K	-12	27	67	N/D	N/D	N/D	-5	10.62	34	N/D	N/D	N/D
128K-2M	-19	4.7	39	0	0.2	1	-14	2.85	12	1	1.6	3
4M-16M	-16	-1	16	0	0.6	1	-17	-1.38	17	1	1.6	3

Poder-se-á verificar que os resultados para as leituras e escritas, ainda que diferentes, são da mesma ordem de grandeza. Quando o acesso faz um maior uso da *cache*, o TxBtrfs apresenta um *overhead* consideravelmente superior para registos de tamanho reduzido (4KB a 64KB), sendo gradualmente reduzido à medida que o tamanho do registo utilizado aumenta. Tal não se verifica, contudo, para acessos que façam efectivamente maior uso do disco, sendo o *overhead* para qualquer tamanho de registo muito próximo de zero.

Este comportamento é consistente com a nossa implementação, sendo toda ela suportada em memória e CPU. Quando o I/O também é muito dependente da memória e do CPU, como aquando do uso de registos de pequena dimensão, a nossa implementação apresenta um comportamento pior. Contudo, à medida que os registos aumentam ou que o acesso aos dados transita da memória para o disco, o *overhead* introduzido pelo uso de CPU e do acesso à memória é atenuado, tornando-se virtualmente negligenciável.

6 Conclusões

O nosso objectivo final é o suporte da semântica transaccional no Btrfs, sendo para isso necessário registar as operações realizadas no contexto de transacções, depois implementar um algoritmo de validação dos registos e detecção de conflitos entre transacções, e posteriormente proceder a reconciliações de snapshots de transacções concorrentes compatíveis. Até ao presente desenvolvemos o módulo de registo das operações, mantendo a conformidade total com a norma POSIX, e

avaliámos a robustez e desempenho do protótipo através de um conjunto de testes baseados no *benchmark* IOZone. Como ficou patente na Secção 5, o uso adicional de CPU e memória pelo TxBtrfs demonstram algum impacto nalguns resultados, face à implementação original do Btrfs, sendo no entanto negligenciável quando o I/O não estiver intrinsecamente dependente do uso de CPU e de memória.

Num futuro próximo, esperamos conseguir reduzir este impacto em ficheiros de pequena dimensão, evitando a introdução de mais *overhead* aquando da finalização das transacções. Iremos também implementar um algoritmo de validação das transacções sobre o Sistema de Ficheiros, e proceder à reconciliação dos snapshots. A validação deste trabalho deverá incluir, para além da avaliação do desempenho, um conjunto de testes visando a verificação da conformidade do TxBtrfs com as propriedades ACID.

Referências

1. Btrfs wiki at kernel.org. `https://btrfs.wiki.kernel.org/`, last modified on 11 January 2011.
2. D. Richard Hipp. SQLite software library. `http://sqlite.org/`, last checked on 25 January 2011.
3. IOzone. IOzone File System Benchmark Home Page. `http://www.iozone.org/`, last checked on 17 June 2011.
4. S. R. Kleiman. Vnodes: An architecture for multiple file system types in sun unix. In *Proceedings of the Summer USENIX Conference*, pages 238–247, Atlanta, 1986.
5. Microsoft. Basic TxF Concepts. `http://msdn.microsoft.com/en-us/library/dd979526%28v=VS.85%29.aspx`, last checked on 14 June 2011.
6. Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system, 2002.
7. Michael A. Olson and Michael A. The design and implementation of the inversion file system, 1993.
8. Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 161–176, New York, NY, USA, 2009. ACM.
9. Tuxera. Posix test suite. `http://www.tuxera.com/community/posix-test-suite/`, last checked on 17 June 2011.
10. Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending acid semantics to the file system. *Trans. Storage*, 3(2):4, 2007.