UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUSTAVO DANIEL FARIAS ALVES

ANÁLISE DE DESEMPENHO DA API IO_URING EM UMA APLICAÇÃO DE USO INTENSIVO DE DADOS

CAMPINA GRANDE - PB

2021

GUSTAVO DANIEL FARIAS ALVES

ANÁLISE DE DESEMPENHO DA API IO_URING EM UMA APLICAÇÃO DE USO INTENSIVO DE DADOS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Thiago Emmanuel Pereira da Cunha Silva.

CAMPINA GRANDE - PB

2021

A474a Alves, Gustavo Daniel Farias.
Análise de desempenho da Api io_uring em uma aplicação de uso intenso de dados. / Gustavo Daniel Farias Alves. – 2021.

9 f.

Orientador: Prof. Dr. Thiago Emmanuel Pereira da Cunha Silva.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. API io_uring. 2. Interface io_uring. 3. Operações de I/O. 4. I/O síncrono. I. Silva, Thiago Emmanuel Pereira da Cunha. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

GUSTAVO DANIEL FARIAS ALVES

ANÁLISE DE DESEMPENHO DA API IO_URING EM UMA APLICAÇÃO DE USO INTENSIVO DE DADOS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Thiago Emmanuel Pereira da Cunha Silva
Orientador – UASC/CEEI/UFCG

Professor Dr. Carlos Wilson Dantas Almeida
Examinador – UASC/CEEI/UFCG

Professor Tiago Lima Massoni
Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 20 de outubro de 2021.

CAMPINA GRANDE - PB

ABSTRACT

In May 2019, at the release of version 5.1 of the Linux kernel, a new API called *io_uring* was introduced. The interface came as a new option to perform asynchronous I/O operations, with proposals for simplicity, better performance and coverage of use cases than its predecessors, like the *aio* interface, did not support. Since the introduction of the API, some software projects have been trying to introduce *io_uring* into their codebases. Of these, there have been some reports of significant performance gains, reaching in some cases double the speed before the interface was implemented. This article provides a brief introduction to the interface, the context in which it is inserted and a comparative analysis between the performance in I/O operations of a real data-intensive application before and after the use of *io_uring*. The results after the modification indicate that the simple introduction of *io_uring* brought a considerable performance drop to the application, which after analysis was shown to have a naturally synchronous and blocking behaviour, nullifying the possible benefits of using asynchronous I/O while maintaining the overhead arising from the management of the *io_uring* interface.

Análise de Desempenho da API *io_uring* em uma Aplicação de Uso Intensivo de Dados

Gustavo Daniel Farias Alves
gustavo.daniel.alves@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

RESUMO

Em maio de 2019, no lançamento da versão 5.1 do kernel Linux, foi introduzida uma nova API chamada *io_uring*. A interface veio como uma nova opção de realizar operações de I/O assíncrono, com propostas de simplicidade, melhor desempenho e cobertura de casos de uso em que seus predecessores, a exemplo da interface *aio*, não davam suporte. Desde a introdução da API, alguns projetos de software vêm tentando introduzir *io_uring* nas suas bases de código. Desses, houveram alguns relatos de ganhos significativos de desempenho, chegando em alguns casos ao dobro da velocidade antes da implementação da interface. Este artigo traz uma breve introdução à interface, o contexto em que ela está inserida e uma análise comparativa entre o desempenho em operações de IO de uma aplicação real de uso intensivo de dados antes e após a utilização de *io_uring*. Os resultados após a modificação indicam que a simples introdução de *io_uring* trouxe uma queda de desempenho considerável à aplicação, que após análise foi demonstrado que ela possui comportamento naturalmente síncrono e bloqueante, nulificando os possíveis benefícios do uso de I/O assíncrono porém mantendo o *overhead* decorrente do gerenciamento da interface *io_uring*.

[1]

1 INTRODUÇÃO

Operações de I/O são parte básica do funcionamento de um computador. Por I/O entendemos como a comunicação de um sistema com o usuário ou com outro sistema de comunicação. Podemos tomar como exemplo a escrita e escrita de leitura de um arquivo em um disco, o envio e recebimento de pacotes pela interface de rede e até mesmo receber entrada de um usuário pelo teclado ou imprimir um conteúdo na tela do computador.

No Linux, por muito tempo só foi possível realizar I/O de forma síncrona e bloqueante, ou seja, no momento em que uma aplicação inicia uma operação de I/O ela não consegue fazer outra coisa até que a operação tenha sido concluída. Na maioria das aplicações este comportamento é aceitável e especialmente com a velocidade do hardware atual é rápido o suficiente para não causar inconveniência aos usuários. Por outro lado, certas aplicações não podem se dar o luxo de esperar que operações de I/O terminem para que continuem a processar outras tarefas. Também há o caso de aplicações que trabalhavam normalmente com I/O síncrono mas que por motivos diversos (a exemplo do impacto causado na mitigação das vulnerabilidades Spectre e Meltdown[11]) já não podiam mais continuar desse jeito.

Em casos em que o I/O síncrono não é ideal, podemos utilizar o I/O assíncrono, introduzindo no Linux com a interface *aio* em 2003. Com I/O assíncrono uma aplicação consegue trabalhar em outras tarefas enquanto uma operação de I/O ainda não foi finalizada. A solução *aio* apesar de tudo, não foi tão bem recebida. Muitos usuários reclamavam que a interface era complicada de se usar, ruim de se manter e que não cobria vários casos de uso de I/O assíncrono [4].

Com o passar dos anos, surgiram nas listas de discussão do kernel várias propostas de alternativas a *aio*, todas sem sucesso[8]. Em 2019 entretanto, a proposta de interface *io_uring* desenvolvida pelo contribuidor Jens Axboe foi integrada a *branch* principal do kernel. A nova interface trabalha com o conceito de anéis de buffer compartilhados entre usuário e kernel e trata dos principais problemas existentes em *aio*. Alguns benchmarks realizados pelo autor indicam um ganho expressivo de desempenho comparado ao *aio* [4].

Este trabalho tem como objetivo a introdução de *io_uring* em uma aplicação com intensivo de I/O para verificar se há um ganho de desempenho comparado a sua implementação original. Antes de tudo, apresentaremos uma fundamentação teórica ao conceito de I/O aos leitores para que se familiarizem como também as interfaces de I/O disponíveis no Linux para uma maior familiarização do leitor com o contexto do trabalho. Também será dada uma breve introdução sobre o design e funcionamento da interface *io_uring*. Ao fim da fundamentação teórica, explicaremos a metodologia utilizada para aferir o desempenho da interface e falaremos brevemente sobre o libtiff, aplicação que será utilizada como estudo de caso. Após isto, teremos uma apresentação dos resultados encontrados e concluiremos com uma discussão sobre os resultados e propostas para trabalhos futuros.

Numa situação em que o libtiff obtenha um ganho considerável de desempenho, poderemos ter um impacto bastante positivo na comunidade. A biblioteca está presente nas mais diversas aplicações, que vão desde softwares de editoração[3] a sistemas de monitoramento do processo de desertificação no semiárido brasileiro[1].

2 FUNDAMENTAÇÃO TEÓRICA

2.1 I/O Síncrono

Para demonstrar o que é I/O síncrono, podemos utilizar como exemplo a chamada de sistema read() disponível em sistemas POSIX. Ela permite, como o próprio nome indica, a leitura dos conteúdos de um arquivo. Para ler um arquivo inteiro, realizamos sucessivas

[1] "Os autores retêm os direitos, ao abrigo de uma licença Creative Commons Atribuição CC BY, sobre todo o conteúdo deste artigo (incluindo todos os elementos que possam conter, tais como figuras, desenhos, tabelas), bem como sobre todos os materiais produzidos pelos autores que estejam relacionados ao trabalho relatado e que estejam referenciados no artigo (tais como códigos fonte e bases de dados). Essa licença permite que outros distribuam, adaptem e evoluam seu trabalho, mesmo comercialmente, desde que os autores sejam creditados pela criação original."

chamadas que lerão um bloco de dados do arquivo a partir de um offset especificado. Ao realizar cada chamada a aplicação deve esperar a leitura completa do bloco para poder continuar, liberando a CPU e sendo posta para o kernel para dormir enquanto a operação não é finalizada (Figura 1). A este tipo de chamada, dizemos que é bloqueante, pois o programa não consegue realizar outros tipos de processamento enquanto a leitura não é finalizada[2].

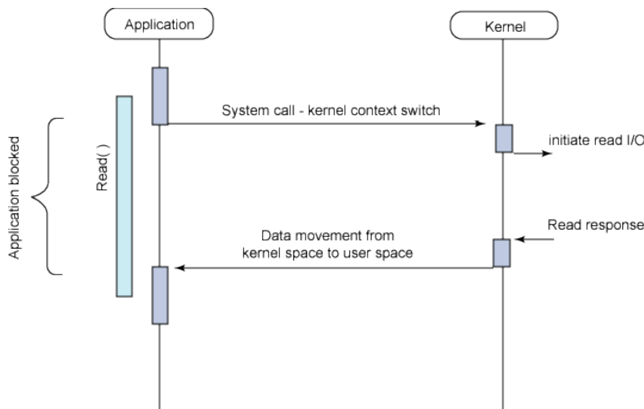

Figura 1: Operação de I/O síncrona bloqueante [10]

Além de chamadas como read() que fazem que um processo espere até a operação de I/O seja finalizada, temos também operações de polling (Figura 2). Nas operações de polling, como poll() e epoll() temos que o processo ativamente consulta o kernel para verificar se os dados resultantes da operação de I/O já estão disponíveis ou já foram processados [2]. Este tipo de operação é bastante comum em sistemas como servidores web, que constantemente verificam a chegada de requisições HTTP pela rede.

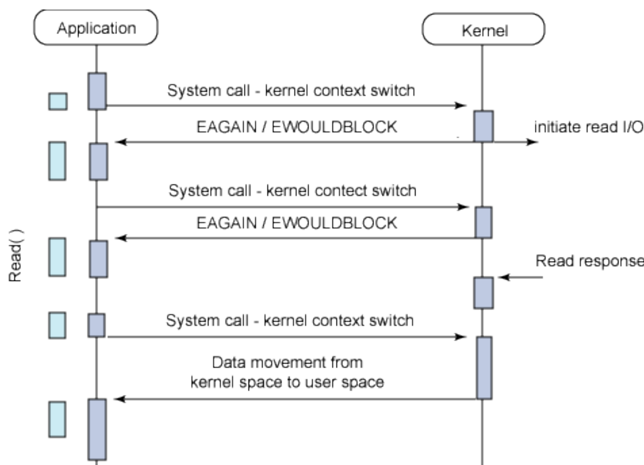

Figura 2: Operação de I/O síncrona (polling) [10]

Para a maioria das aplicações, esperar que uma operação de escrita ou leitura seja finalizada antes de continuar o processamento e execução do código não é um problema ou até mesmo é necessário para o contexto da aplicação. Entretanto, em algumas situações seria interessante que enquanto uma operação de I/O é realizada, a aplicação pudesse continuar trabalhando fazendo outra coisa. A este tipo de I/O chamamos de I/O assíncrono.

2.2 I/O assíncrono

No I/O assíncrono (Figura 3), uma aplicação realiza uma chamada ao sistema indicando que quer que algo seja feito. O kernel recebe esse pedido e começa a trabalhar. Enquanto a operação não é terminada, a aplicação consegue continuar realizando outros tipos de processamento, recebendo uma notificação do kernel indicando que o I/O foi finalizado e que ela pode terminar de realizar a tarefa que envolveu o I/O [10].

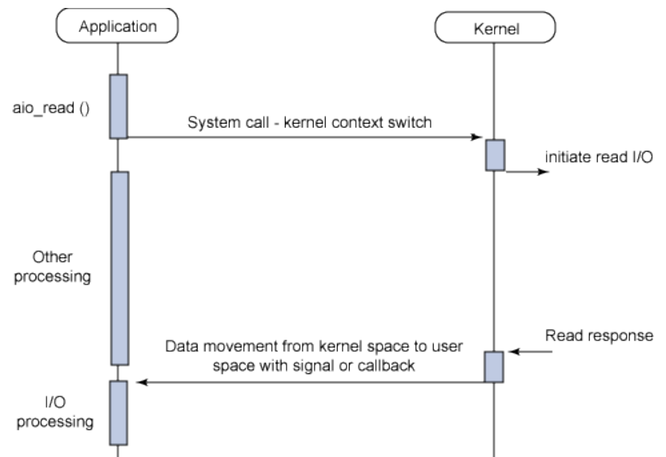

Figura 3: Operação de I/O assíncrona [10]

2.3 Async I/O (aio)

Em sistemas Linux, a interface utilizada para tratar de I/O assíncrono se chama *aio*. Introduzida no final de 2003 por Bhattacharya, Pratt, Pulavarty, e Morgan [7], veio com a pretensão de resolver o problema do I/O assíncrono com o Linux.

Ao utilizar *aio*, uma aplicação poderia submeter diversas requisições e continuar processando outras tarefas enquanto a requisição não era dada por completa. A interface apesar de promissora demonstrou com o tempo que não era uma solução satisfatória por causa de diversas limitações na sua implementação. Para requisições com buffer a interface funcionava de forma assíncrona, o oposto do que ela queria fazer. Ela também trazia um *overhead* causado pelas diversas chamadas de sistema necessárias e com o custo associado às cópias necessárias nas filas de submissão e conclusão de I/O entre o contexto do usuário e do kernel. A interface também era considerada complicada de se lidar, o que prejudicava tanto o usuário, que poderia realizar a operação de forma errada como os mantenedores que tinham dificuldades em estender e melhorar suas funcionalidades [4].

Após algum tempo, surgiram diversas propostas da comunidade para o desenvolvimento de novas interfaces de I/O assíncrono que substituiriam a interface *aio* [8]. Infelizmente a maioria delas era também considerada insatisfatória e acabavam não sendo integradas à branch principal do kernel.

Em 2019, o desenvolvedor dinamarquês Jens Axboe propôs uma nova interface para as operações assíncronas chamada *io_uring*. Felizmente esta interface foi considerada boa o suficiente para que no fim do ano fosse integrada ao kernel linux na versão 5.1 [13].

2.4 A interface *io_uring*

Com a intenção de substituir o uso da biblioteca *aio*, Jens Axboe teve como diretrizes a criação de uma interface que fosse simples de usar, tivesse o mínimo de *overhead* e que desse suporte aos casos em que *aio* tinha desempenho insatisfatório. [4]

Para lidar com I/O assíncrono, a interface *io_uring* utiliza a ideia de anéis de buffer compartilhados que gerenciam filas de submissão e conclusão de requisições de I/O (Figura 4). Quando uma aplicação inicia uma requisição de I/O, ela insere uma entrada numa fila de submissão que é compartilhada entre o usuário e o kernel. Após a operação ser processada, o kernel adiciona um evento na fila de conclusão, que também é compartilhada com o usuário. Uma entrada de submissão é uma estrutura que contém informações sobre o tipo de operação a ser realizada, o descritor de arquivo a ser utilizado, o offset absoluto do arquivo, dados a serem utilizados na operação e flags de configuração. Um evento na lista de conclusão por outro lado só contém os dados utilizados na operação finalizada, a resposta resultante (número de bytes escritos no caso de uma operação do tipo write() ou um erro) e um conjunto de flags que trazem metadados sobre a operação feita.

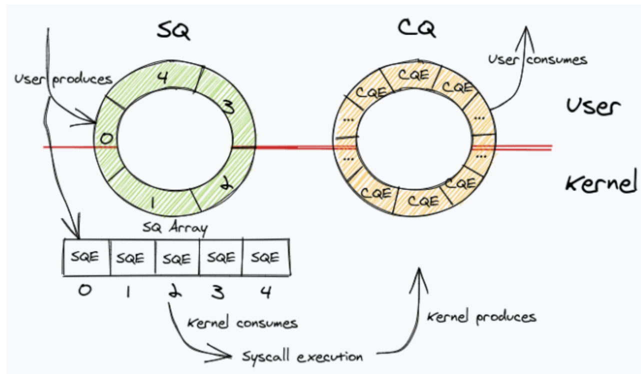

Figura 4: Fluxo de operação do *io_uring* [14]

O fato do kernel compartilhar os anéis de buffer com o usuário evita um dos problemas existentes com *aio* que era o custo com cópias necessárias para cada entrada de submissão e evento de conclusão. Com *aio* as duas juntas tem um custo intrínseco de 104 bytes[4], o que apesar de aparentar ser pouco pode se tornar um *overhead* enorme quando estamos a falar de milhões de operações por segundo, o que é um caso comum em determinados tipos de sistemas.

Quando um usuário cria entradas de submissão suficientes, ele as submete ao kernel uma única vez, evitando o uso de várias chamadas ao sistema. Esta solução também é eficiente por diminuir a quantidade de chamadas ao sistemas necessárias no fluxo do I/O. Após processar cada evento de conclusão, o usuário indica isto ao anel de buffer, que libera o espaço nas filas para realização de novas operações de I/O. Por base temos que a API utiliza três funções diferentes: uma função para inicializar os anéis de buffer, o descritor de arquivo referente à estrutura *io_uring* e configurações adicionais, outra para registrar buffers ou arquivos a serem utilizados nas operações e outra para iniciar e completar o I/O.

A implementação de *io_uring* em uma aplicação é relativamente simples, entretanto nota-se que a maior parte do código escrito é composto por *boilerplate*. Visando simplificar ainda mais o uso da interface, foi desenvolvida uma biblioteca de alto nível chamada liburing que diminui a quantidade de código necessário para utilização pelo usuário na sua base de código. Por exemplo, um clone do utilitário cat tem 360 linhas com o *io_uring* puro e apenas 160 com o liburing, ou seja, menos da metade das linhas de código [9].

Desde a sua introdução, alguns desenvolvedores realizaram testes com a interface e obtiveram ganhos de desempenho consideráveis. Guilherme Bernal, que trabalha com a linguagem Crystal, criou alguns benchmarks em que constatou uma melhora de até 102,56% ao utilizar o *io_uring* em detrimento de interfaces síncronas.[6] Jens Axboe, em um apresentação no Kernel Recipes 2019, afirmou que testes em aplicações como RocksDB e num projeto interno de bigcache do Facebook houve ganhos de 73 e 35% respectivamente.[5]

3 METODOLOGIA

Para verificar o desempenho do *io_uring*, torna-se necessário comparar o desempenho de uma aplicação com uso intensivo de operações de IO com e sem o uso da interface. A aplicação escolhida como estudo de caso é a biblioteca de manipulação de imagens no formato TIFF conhecida por libtiff.

Criada ainda no fim dos anos 80, libtiff é uma biblioteca livre e de código aberto que provê ferramentas para a manipulação de imagens TIFF[12]. TIFF é um formato de arquivo utilizado para descrever e armazenar imagens raster. Desenvolvida pela Aldus Corporation e posteriormente pela Adobe[3], o formato é popular em diferentes meios, como fotografia, editoração eletrônica, design gráfico e até imagens de satélite. Seus arquivos podem chegar a ocupar gigabytes de memória.

A biblioteca libtiff por si só não é útil para a análise de desempenho, já que ela é apenas um conjunto de ferramentas para lidar com imagens TIFF. Portanto este trabalho utilizará para avaliação do desempenho um pequeno utilitário que gera imagens TIFF quadradas com conteúdo aleatório utilizando a libtiff por baixo [15]. O utilitário tem código aberto e foi desenvolvido pelo SADes, equipe parte do Laboratório de Sistemas Distribuídos da Universidade Federal de Campina Grande.

Para a análise, verificaremos o tempo necessário para que a aplicação finalize a execução e geração da imagem TIFF. Nesta situação, um tempo menor de execução indica que *io_uring* traz um ganho de desempenho à aplicação.

Para cada caso de teste, será considerada a média de 25 execuções da aplicação. A verificação do tempo ficará a cargo do utilitário Linux perf disponível, parte do pacote linuxtools. Os testes rodarão numa instância Ubuntu 21.04 com versão do kernel 5.11.0-37-generic e com o pacote liburing 0.7-3ubuntu1. A máquina de testes possui um processador AMD Ryzen 3700u, 8GB de RAM DDR4-2400 e um SSD NVMe de 256 GB.

3.1 Modificando a biblioteca

A biblioteca libtiff possui um tamanho considerável mas a modificação terá foco em apenas uma função disponível no arquivo tif_unix.c: a função static tmsize_t _tiffWriteProc(thandle_t fd, void* buf, tmsize_t size). A assinatura da função nos mostra que ela tem como parâmetros fd, que representa um descritor de arquivo, buf que representa o buffer que contém o conteúdo a ser escrito no arquivo, e size que indica o tamanho do buffer em bytes. A função retorna um valor do tipo tmsize_t, que representa o número de bytes escritos com a operação.

_tiffWriteProc() é consideravelmente simples. Sua implementação divide o buffer em blocos de tamanho máximo TIFF_IO_MAX (definido como 2147483647U) e serialmente executa a system call write(). Após o fim da execução, são retornados a soma dos retornos da função write, ou seja, o número de bytes escrito.

A modificação proposta adiciona uma pequena camada de complexidade à função. Em resumo, podemos dividir a modificação nas seguintes partes: inicializar estrutura *io_uring*, preparar entradas de submissão, submetê-las, esperar por entradas na fila de conclusão, notificar ao kernel que as entradas da fila de conclusão foram processadas e retornar número de bytes escritos.

Iniciamos uma estrutura do tipo *io_uring* com a função *io_uring* _queue_init(), que inicializa as filas de submissão para uso. Após isto, entramos em um loop que pega entradas na lista de submissão com a função *io_uring*_get_sqe() e prepara nelas operações do tipo write com *io_uring*_prep_write(). A assinatura desta função difere da função clássica write() por dois motivos: além de indicar a entrada da fila de submissão, é necessário indicar o offset absoluto do arquivo a ser escrito. Após gerar a quantidade necessária de entradas de submissão, executamos a função *io_uring*_submit(), que submete as entradas para o kernel. Para verificarmos se as escritas ocorreram, esperamos em um loop com a função *io_uring*_wait_cqe() espera retornando uma entrada da fila de conclusão. Após o retorno de cada entrada da fila de conclusão, verificamos a resposta disponível na entrada, indicando a quantidade bytes escritos e indicamos que ela foi processada com a função *io_uring*_cqe_seen(). Após todas as operações serem concluídas, destruímos a estrutura *io_uring* com a função *io_uring*_queue_exit(), que desmapeia e desaloca os anéis de buffer compartilhados entre o usuário e o kernel além de realizar outras operações necessárias. Terminamos enfim retornando a quantidade total de bytes escritos.

4 RESULTADOS

Os resultados da suíte de testes nos mostram que a versão da aplicação com escrita sequencial foi mais rápida do que a versão com *io_uring*. Após aplicação do teste de Student para todos os tamanhos, foi verificado com 95% de confiança que em todos os casos ($p_valor < 2,2 * 10^{-16}$) a diferença é estatisticamente significante, exceto para imagens de 64x64px ($p_valor = 0,19$) (Figuras 5 e 6).

Uma rápida análise de como o gerador de arquivos TIFF e a biblioteca libtiff funcionam pode nos ajudar a entender melhor o resultado: Apesar de estarmos utilizando uma interface de I/O assíncrona, a geração da imagem TIFF ocorre de forma sequencial, ou seja, linha por linha. Com isto o fluxo de execução se torna similar ao da versão original que utiliza a chamada ao sistema write(). Tendo isto em mente, podemos supor que a perda de desempenho é significante pois o custo da inicialização da estrutura *io_uring* tem um impacto considerável na versão modificada.

Com este resultado em mãos, pode-se imaginar que este trabalho vai de encontro com os trabalhos citados anteriormente, o que seria errado. As aplicações que obtiveram um ganho de desempenho com o uso de *io_uring* conseguiram tal feito por estarem num contexto que o I/O assíncrono faz mais sentido (servidores e banco de dados). Como discutido aqui, a aplicação testada neste trabalho acaba por estar num contexto oposto.

5 CONCLUSÕES

Os resultados descritos anteriormente demonstram que apenas a simples substituição dos métodos de IO síncronos bloqueantes por assíncronos não bloqueantes em uma aplicação não necessariamente trará um aumento de desempenho, já que às vezes a aplicação tem natureza síncrona e bloqueante, como o caso do gerador de imagens TIFF/libtiff.

Apesar do resultado diferente do esperado, este estudo de caso ainda assim tem um impacto positivo: a demonstração de uma situação em que o *io_uring* não traz benefícios aparentes a uma aplicação. A análise gerada pode ajudar a comunidade a ter uma melhor ideia de quando deve-se tentar ou não utilizar o *io_uring* em uma aplicação.

Como sugestão de trabalhos futuros, temos diferentes casos de estudo: uma análise em outra parte de libtiff, como a que lida com a leitura de arquivos; uma análise de desempenho com um outro tipo de aplicação qualquer de uso intensivo de I/O, e/ou a comparação de uma aplicação que utiliza I/O assíncrono usando *aio* com a mesma aplicação utilizando *io_uring*.

AGRADECIMENTOS

A realização deste trabalho não seria possível sem a contribuição de diversas pessoas durante a graduação. Em especial, gostaria de agradecer ao professor Thiago Emmanuel, que não só deu a ideia de tema para a realização deste TCC como também foi meu professor na disciplina de laboratório de sistemas operacionais; aos diversos colegas e amigos que me acompanharam durante a graduação, nos momentos fáceis e difíceis; a todos aqueles que contribuem desde sempre para a existência de um ensino superior gratuito e de qualidade, como o que recebi na Universidade Federal de Campina Grande, e, por último, gostaria de agradecer também à comunidade de software livre, já que este trabalho tem como base a dedicação contínua de milhares de diferentes pessoas na execução de projetos abertos que impactam bastante a sociedade, a exemplo de todos os softwares utilizados na construção desse artigo, como Linux, libtiff, ferramentas GNU, Ubuntu, vim, LaTeX entre outros.

REFERÊNCIAS

[1] [n. d.]. Sistema de monitoramento do processo de desertificação. https://www.lsd.ufcg.edu.br/#/projeto/1951
[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces* (1.00 ed.). Arpaci-Dusseau Books.
[3] Adobe Developers Association. 1992. TIFF - Revision 6.0. https://www.adobe.io/content/dam/udp/en/open/standards/tiff/TIFF6.pdf
[4] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf
[5] Jens Axboe. 2019. Faster IO through io_uring. https://www.youtube.com/watch?v=-5T4Cjw46ys
[6] Guilherme Bernal. 2021. Reaching 200k req/s on a single core with io_uring. https://www.youtube.com/watch?v=TYq_ohhYZ9A

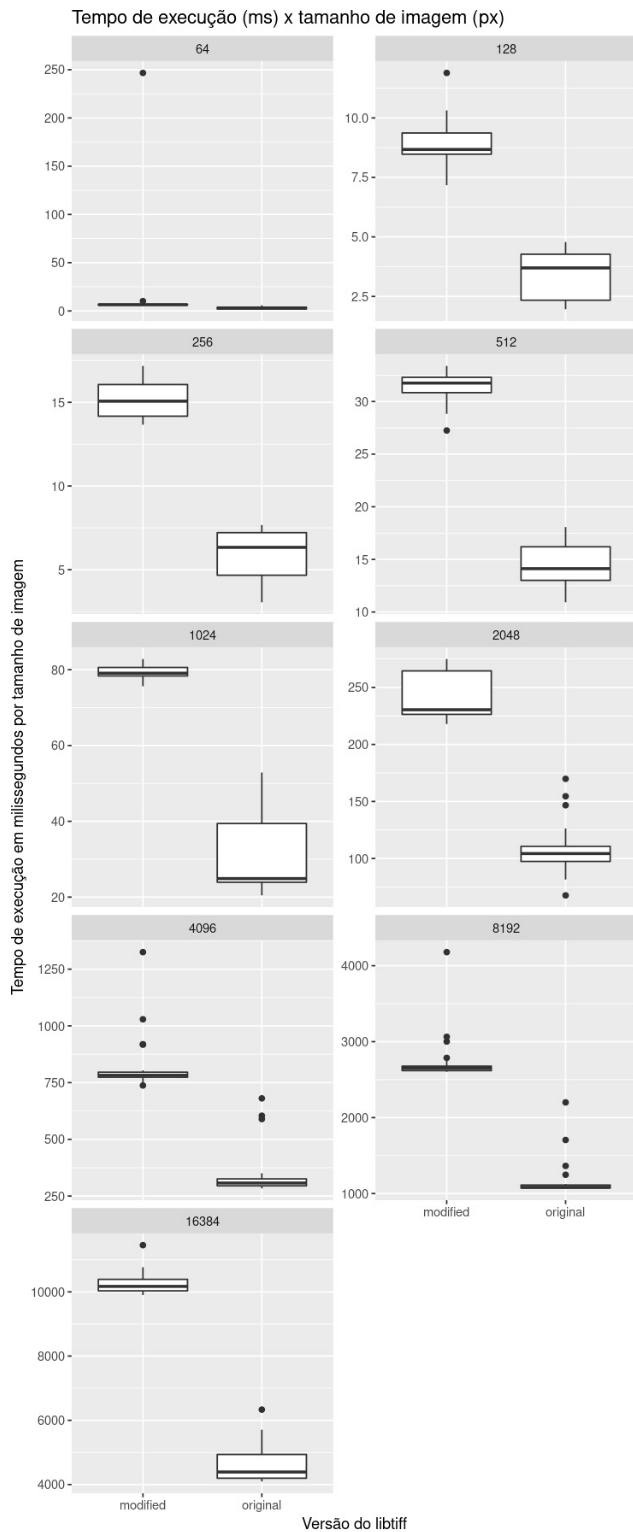

Figura 5: Distribuição dos dados de tempo de execução da aplicação por tamanho de imagem e versão do libtiff

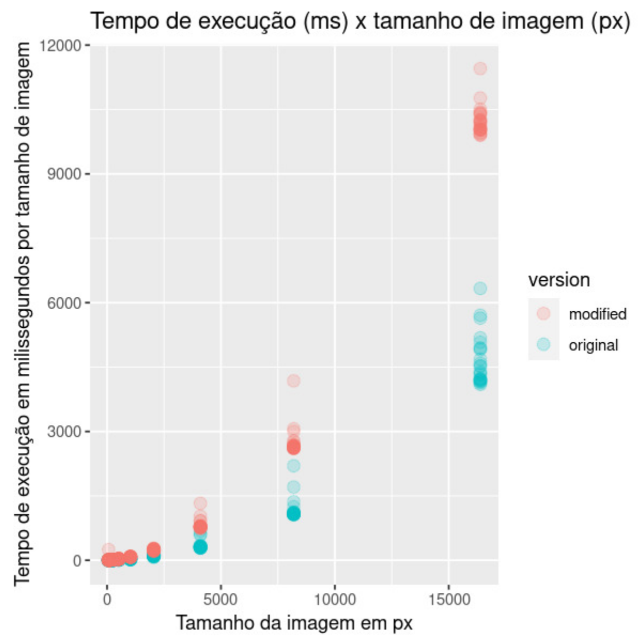

Figura 6: Tempo de execução da aplicação por tamanho de imagem e versão do libtiff

[7] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. 2003. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*. 371–386.

[8] Jonathan Corbet. 2020. The rapid growth of io_uring. https://lwn.net/Articles/810414/

[9] Shuveb Hussain. 2020. cat with liburing. https://unixism.net/loti/tutorial/cat_liburing.html

[10] M Jones. [n. d.]. Boost application performance using asynchronous I/O. https://developer.ibm.com/articles/l-async/

[11] Michael Larabel. 2020. Looking At The Linux Performance Two Years After Spectre / Meltdown Mitigations. https://www.phoronix.com/scan.php?page=article&item=spectre-meltdown-2

[12] Sam Leffler. 1986. libtiff. http://www.libtiff.org/

[13] Linux Kernel Newbies. 2019. Linux 5.1. https://kernelnewbies.org/Linux_5.1#High-performance_asynchronous_I.2FO_with_io_uring

[14] Agniva De Sarker. 2021. Getting hands-on with io_uring using Go. https://mattermost.com/blog/iouring-and-go/

[15] SADes UFCG. 2018. TiffUtils. https://github.com/simsab-ufcg/TiffUtils

A REPOSITÓRIOS

Os softwares utilizados na execução deste trabalho estão disponíveis de forma aberta na internet. Os repositórios estão listados logo abaixo:

- libtiff: https://gitlab.com/libtiff/libtiff
- liburing: https://github.com/axboe/liburing/
- Gerador de TIFFs: https://github.com/simsab-ufcg/TiffUtils
- libtiff modificado: https://github.com/gustavodfa/libtiff