Kaiyo: Testagem Automática de Aplicações de Memória Persistente

Henrique Fernandes, João Gonçalves, and Miguel Matos

IST U. Lisboa & INESC-ID

Resumo A memória persistente (PM) é uma nova tecnologia que tem performance equivalente a DRAM e oferece não-volatilidade. PM é endereçável ao byte e comunica diretamente com o processador, aumentando o desempenho quando comparado com discos.
Ao escrever em PM, a escrita é primeiro guardada na cache do CPU e só depois é persistida na PM. Na eventualidade de uma falha, qualquer escrita ainda na cache perde-se. Além disto, o hardware pode reordenar a ordem pela qual as instruções são executadas para aumentar o desempenho. Isto resulta na perda de garantias do estado da PM depois de um crash, podendo existir estados pós-falha erróneos ou irrecuperáveis.
Com o objetivo de evitar estados indesejados, os programadores necessitam de ferramentas que lhes permitam assegurar *crash-consistency* nas suas aplicações, eliminando-as de *bugs* que possam levar a estes estados. Existem ferramentas no estado da arte com este propósito, mas nenhuma oferece simultaneamente eficiência, automação e cobertura. Algumas são ineficientes temporalmente. Outras requerem que o programador recompile código ou testam apenas um conjunto limitado de aplicações.
Este artigo apresenta Kaiyo, uma ferramenta que oferece simultaneamente estas três características. Conseguimos isto pois: i) aplicamos um conjunto de heurísticas que reduzem o espaço de estados a testar, ii) tiramos proveito de copy-on-write para eficiência temporal e espacial na criação deste espaço e iii) paralelizamos o processo de recuperação, que permite reduzir substancialmente o tempo de teste.

Keywords: Memória persistente · crash-consistency · testagem · automação · paralelização.

1 Introdução

Memória persistente (em inglês *Persistent Memory* ou PM) é uma tecnologia que combina o desempenho próximo de DRAM com a não-volatilidade dos discos e memória flash. Esta memória endereçável ao byte é acessível pelos programas através de instruções de `store` e `load` normais. A PM interage diretamente com o processador através do barramento de memória, curto-circuitando o kernel. Isto traduz-se num aumento de desempenho significativo quando comparado com HDD ou SSD.

Para obter este desempenho, as escritas não são persistidas imediatamente, mas são inicialmente guardadas na cache do processador. As escritas só são

persistidas quando a respetiva linha de cache é despejada. Os processadores modernos têm instruções especiais para os programadores terem controlo sobre este comportamento, nomeadamente as instruções de `flush` e `fence`. Um `flush` instrui que uma linha da cache seja despejada. Um `fence` atua como uma barreira, garantindo que todas as escritas que foram *flushed* são persistidas. Isto significa que só existe a garantia de persistência de uma escrita caso esta seja seguida por um `flush` e um `fence`. Consequentemente, duas ou mais escritas para linhas de cache distintas entre dois `fences` não têm qualquer garantia de ordem de persistência[1]. Fortalecer estas garantias teria um impacto negativo no desempenho, derrotando uma das principais vantagens de PM.

Definimos coerência-sob-falhas ou CSF como a propriedade que, na eventualidade de uma falha, garante que o sistema é restaurado para um estado consistente. Isto significa que não observamos informação incorreta ou incoerente em memória. CSF é fulcral no caso de aplicações de PM pois o comportamento na eventualidade de uma falha não é igual ao de aplicações que usam DRAM. Numa falha, todo o conteúdo da memória volátil é perdido. Visto de outra forma, é como se observássemos um *rollback* para um estado correto, pois a memória está vazia. No caso de PM, as escritas vão sendo persistidas à medida que a aplicação executa. Isto significa que, quando experenciamos uma falha, o estado da memória é incerto, e depende de quais escritas persistiram até àquele ponto da execução. O problema torna-se então óbvio, pois certos estados de memória podem levar a uma execução errónea da aplicação. Vamos considerar o caso em que tentamos persistir um par chave-valor antes de uma falha. Se, após a falha, apenas a chave tiver persistido e tentarmos aceder ao valor, a aplicação vai exibir um comportamento inesperado. Este comportamento pode traduzir-se num *crash* por acesso indevido a memória (uma vez que o valor não foi persistido) ou numa execução indesejada (um valor aleatório naquela posição de memória). Para evitar este caso, o programador tem de persistir o valor antes da chave.

Este exemplo ilustra que existem mais estados possíveis a que um programa pode chegar para além daqueles da ordem do programa. Devido às reordenações que o hardware pode efetuar, existem inúmeros estados pós-falha que não são óbvios [13] e que se podem traduzir em *bugs* nas aplicações. Para detetar estes *bugs* têm surgido no estado da arte várias ferramentas com diferentes compromissos entre cobertura, eficiência e automação. Algumas ferramentas exploram demasiados estados e não o conseguem fazer em tempo útil [1,2]. Outras recorrem ao uso de anotações para diminuirem o espaço de estados a explorar [4,5], mas sacrificam automação e adicionam uma variável de incerteza à eficácia da própria ferramenta: o uso correto destas anotações. Assim, apresentamos Kaiyo, que preenche um espaço existente no estado da arte, ao oferecer simultaneamente automação, eficiência e cobertura.

A nossa abordagem utiliza o processo de recuperação como oráculo de correção. Deste modo evitamos impor trabalho extra aos programadores pois, à semelhança de outras abordagens [8], consideramos que o processo de recuperação é

[1] Excluindo o caso em que as linhas de cache de cada escrita são explicitamente despejadas com recurso a `flushes` não-otimizados [13].

um componente fundamental da própria aplicação de PM. Assim, usar o processo de recuperação como oráculo (ainda que imperfeito) confere automação à nossa ferramenta. Isto significa que, se uma aplicação recupera corretamente então não reportamos *bugs*. Por outro lado, um erro na recuperação indica a existência de um ou mais *bugs*. Note-se que não é garantida a ausência de *bugs* num estado recuperável.

Para alcançar eficiência, executamos a aplicação apenas uma vez e obtemos o nosso espaço de estados. Explorar todo o espaço é impossível em tempo útil [1,2]. Assim, para mitigar este problema, implementamos um algoritmo que exclui estados impossíveis[2] e um conjunto de heurísticas que excluem estados menos prováveis de conter *bugs* na sua origem. Utilizamos *copy-on-write* na criação do espaço do estados e paralelizamos o processo de recuperação, de modo a aumentar a eficiência da ferramenta. Conseguimos então explorar uma quantidade elevada de estados num intervalo de tempo aceitável, aumentando a cobertura. Em particular, conseguimos alcançar um speedup de 3x ao paralelizarmos a recuperação enquanto reduzimos o espaço utilizado em disco em 99%.

2 Trabalho Relacionado

Automação Começamos a apresentação de algumas ferramentas do estado da arte com foco na automação. Esta propriedade oferece aos programadores um método de testarem as suas aplicações com uma configuração geralmente mais direta. No entanto, podem geram um espaço de estados demasiado grande, o que limita a sua escalabilidade. O Yat [1] testa todos os espaços possíveis através das reordenações das instruções que o hardware pode efetuar. Os autores desta ferramenta demonstram como esta abordagem não escala para qualquer aplicação minimamente complexa. Ademais, o Yat é limitado a aplicações que utilizem sistema de ficheiros PMFS. O Jaaru [2] usa a lógica base do Yat, e contabiliza quais as linhas de cache garantidas de terem persistido antes de uma falha. Consegue assim testar menos estados do que o Yat, pois elimina certos estados impossíveis. No entanto, a eficiência desta técnica depende da frequência de um padrão de programação - *commit store*. Isto leva a que a ferramenta seja melhor usada para testar código mais simples e pequeno, uma vez que, para aplicações mais complexas, o Jaaru ultrapassa rapidamente o limite temporal aceitável.

O Witcher [7] cria um espaço de estados mais pequeno que o Yat e o Jaaru. Alcança isto através da deteção de padrões nas dependências entre as escritas. Os estados gerados pelo Witcher correspondem a imagens de PM que violam as dependências dos padrões que detetam. Para detetar *bugs*, o Witcher compara o *output* da execução entre o estado que testa e o estado equivalente onde não tenha occurido uma falha. No entanto, só consegue aplicar estas técnicas à custa de restringir a testagem a aplicações de bases de dados chave-valor e da criação manual de um *driver* específico para a base de dados. O Agamotto [6] utiliza execução simbólica em conjunto com heurísticas para encontrar estados mais

[2] Que poderiam ser obtidos através de uma reordenação simplista das instruções e que não está ciente das garantias do *hardware*.

propícios a conterem *bugs*, sem executar a aplicação em si. No entanto, execução simbólica tem grande complexidade espacial, o que não é viável para aplicações mais longas ou complexas. O Agamotto recorre a oráculos customizados para detetar *bugs*. Os autores apenas apresentam dois oráculos universais que se focam em detetar *bugs* de desempenho. A eficácia do Agamotto depende da criação correta destes oráculos para a semântica de cada aplicação.

O Mumak [8] oferece eficiência e funciona de forma *black-box*, sendo agnóstico à semântica das aplicações. Realiza instrumentação binária com heurísticas para criar um espaço de estados reduzido ao evitar injetar faltas repetidamente em pontos que teriam o mesmo caminho de execução. A isto junta análise do *trace* da execução de forma a detetar *bugs* não abrangidos pela instrumentação, como por exemplo *bugs* de desempenho, que não originam estados irrecuperáveis. No entanto, o Mumak apenas testa estados que sejam observáveis na ordem do programa, logo, não tem em conta outras reordenações possíveis que o hardware pode realizar e que podem resultar em *bugs*.

Anotações do programador O PMDebugger [3] foca-se em eficiência e cobertura ao investir na forma como guarda informação sobre o estado da memória e em vários padrões que levam a *bugs*. Encontra *bugs* ao analisar as estruturas que mantém sobre a memória que a aplicação utiliza. Esta técnica é prática para encontrar *bugs* de desempenho. No entanto, para muitos dos *bugs* de correção, necessita de recorrer a anotações que descrevem a ordem pela qual as escritas devem persistir em cada função.

Existem dois sistemas cujo funcionamento assenta nas mesmas bases. O PMTest e o XFDetector recorrem a anotações para a exploração do espaço de estados. O PMTest utiliza anotações também para saber quais as variáveis a rastrear e qual a ordem pela qual as variáveis devem ser persistidas [4]. Tendo esta informação, o PMTest verifica se a execução da aplicação garante a intenção do programador. Já o XFDetector simula falhas em certos pontos e verifica a coerência da memória entre as execuções pré- e pós-falha [5].

A eficácia destas ferramentas depende em grande parte da utilização correta das anotações que cada uma requer. Além disto, existe um tempo extra necessário para se aprender a utilizar cada uma das ferramentas, o que é indesejado.

3 Abordagem

O nosso objetivo é oferecer uma ferramenta que permita detetar o máximo de *bugs* possível numa janela de tempo aceitável (consideramos 12 horas para que seja integrado num *pipeline* de desenvolvimento contínuo), sem impor trabalho extra aos programadores e que seja agnóstica à aplicação a testar. O ponto de partida é uma solução ingénua que testa todos os estados exaustivamente.

O problema crítico desta abordagem é a explosão do espaço de estados. Mitigamos isto através das ideias seguintes: i) aplicamos um conjunto de heurísticas que reduzem o espaço de estados; ii) executamos a aplicação apenas uma vez para

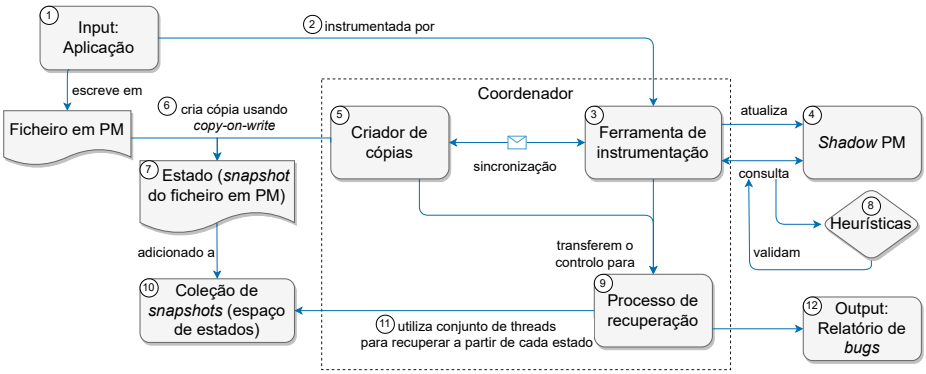


Figura 1: *Pipeline* do Kaiyo.

gerarmos todo o espaço de estados com recurso a instrumentação binária e *copy-on-write*; iii) utilizamos a recuperação da aplicação como oráculo de coerência, tornando a nossa solução agnóstica à aplicação; e iv) paralelizamos o processo de recuperação, uma vez que pré-geramos todo o espaço de estados, tornando a nossa solução altamente escalável.

De forma a explorarmos mais estados, nomeadamente os estados pós-falha provenientes das possíveis reordenações das instruções que o CPU pode realizar, implementamos ainda um algoritmo que simula estas reordenações sem criar estados impossíveis, respeitando a semântica das instruções de PM [13]. Aumentamos assim a cobertura da nossa solução comparativamente a sistemas que não têm em conta estas reordenações [5,6,8].

Fluxo do Kaiyo Seguindo a Figura 1, começamos por instrumentar a aplicação a testar ① - ③. Ao executar a aplicação vamos mantendo o seu estado numa *Shadow PM* ④. Em pontos chave (detalhados abaixo), paramos a execução da aplicação e criamos uma cópia do estado, que é representado por um ficheiro em disco ⑦. Existe uma ferramenta externa ⑤ que sincroniza com a instrumentação nos pontos chave e cria as cópias ⑥. De seguida, consultamos a *Shadow* PM e recorremos a heurísticas para obter os estados provenientes das reordenações das instruções ⑧. Para cada um destes estados, criamos uma cópia do estado da memória. No final da instrumentação obtemos um conjunto de *snapshots* que corresponde ao espaço de estados a testar ⑩. Passamos então à fase de recuperação ⑨, em que utilizamos um conjunto de *threads* para recuperar cada estado em paralelo ⑪. Cada processo individual de recuperação gera depois um relatório de *bug* caso a recuperação não termine com sucesso. Este conjunto de relatórios é o *output* do Kaiyo, com a descrição dos *bugs* e a sua origem ⑫.

Para além da ordem do programa Com o objetivo de aumentarmos a cobertura da nossa solução, exploramos não só os estados correspondentes à ordem

do programa, como também estados provenientes das possíveis reordenações que o CPU pode efetuar sobre as instruções para melhorar o desempenho. Note-se que não consideramos reordenações provenientes de intercalações num programa que recorra a vários fios de execução. Apesar de podermos aplicar a nossa abordagem a este caso, a implementação foca-se num único fio de execução. De forma a respeitar a semântica das instruções e evitar gerar estados impossíveis, consideramos as seguintes restrições, que correspondem a garantias dadas pelo *hardware* [13]: i) dentro de uma linha de cache, as escritas persistem pela ordem em que são instruídas; ii) na PM globalmente, as escritas persistem conforme os despejos que o CPU realiza na cache, estando dependentes das possíveis reordenações que este pode efetuar; e iii) uma escrita só é garantidamente persistida após um `flush` e `fence`.

Com estas considerações e tirando proveito da informação que guardamos na *Shadow* PM, implementamos um algoritmo que nos permite criar o espaço de estados resultante da aplicação destas possíveis reordenações. Detalhamos de seguida o funcionamento do mesmo, apresentado no Algoritmo 1.

Definimos um segmento como uma parte da execução da aplicação entre dois `fences`. A função *Test Reorderings* (linhas 4 - 9) é invocada quando processamos um `fence`, sendo o final de cada segmento e um ponto da execução em que existem mudanças permanentes no estado de persistência da PM. Começamos por obter as reordenações dentro de cada linha de cache (linhas 6 e 7) através da função *Reorderings Cacheline* (linhas 11 - 21). A *Shadow* PM contém uma entrada para cada endereço da PM mapeada pela aplicação. Para cada entrada, temos uma lista dos valores possíveis que se podem observar naquele endereço, onde cada valor tem um contador universal (*timestamp*) do momento em que se observou a escrita desse valor. Assim, percorremos os *timestamps* em que se observaram escritas na linha de cache e definimos um intervalo deslizante (linhas 13 - 15). Para cada intervalo temporal, criamos um conjunto de pares endereço-valor (*reordering*). Estes pares representam o valor que cada endereço pode ter no intervalo definido, obtido na linha 18. Explicamos mais à frente com o exemplo da Figura 2 como obtemos estes valores. Este processo é repetido para cada intervalo de tempo até que sejam determinadas todas as ordens observáveis naquela linha de cache no segmento em causa.

Para obtermos todas as reordenações possíveis na PM, aplicamos depois o produto cartesiano entre todas as combinações de cada linha (linha 8). No final, aplicamos cada reordenação em memória para gerar o respetivo *snapshot*. As linhas 20 - 24 ilustram como geramos os *snapshots* de cada reordenação. Para cada reordenação, iteramos sobre o conjunto de pares endereço-valor, e escrevemos o valor na PM (linhas 22 e 23). Tendo a reordenação aplicada na PM, criamos então o *snapshot* em disco deste estado. Visto que a diferença entre dois *snapshots* pode ser bastante pequena (por exemplo uma escrita de 4 *bytes*), recorremos a uma técnica de gestão de sistemas de ficheiros que nos permite fazer cópias utilizando drasticamente menos espaço quando as cópias têm conteúdo comum: *copy-on-write* ou CoW [10]. Para tal, utilizamos um sistema de ficheiros que permite, de forma eficiente, realizar cópias de ficheiros usando CoW. Assim,

Algoritmo 1 Criação dos estados a partir das reordenações das instruções.

```
 1: Shadow_PM ← {}
 2: stores_per_cacheline ← {}
 3:
 4: function Test_Reorderings(Shadow_PM, stores_per_cacheline)
 5:     reorderings_per_cacheline ← {}
 6:     for all cl, ts_set ∈ stores_per_cacheline // cl = cacheline
 7:         reorderings_per_cacheline[cl] ← Reorderings_Cacheline(Shadow_PM, cl, ts_set)
 8:     reorderings ← ∏_{i=1}^{n} reorderings_per_cacheline_i
 9:     Apply_Reorderings(reorderings)
10:
11: function Reorderings_Cacheline(Shadow_PM, cacheline, timestamp_set)
12:     reorderings ← ∅
13:     for index ∈ timestamp_set
14:         lower_bound ← timestamp_set[index]
15:         upper_bound ← timestamp_set[index + 1]
16:         reordering ← ∅
17:         for all addr ∈ cacheline
18:             value ← Shadow_PM[addr].getBoundedValue(lower_bound, upper_bound)
19:             reordering ← reordering ∪ ⟨addr, value⟩
20:         reorderings ← reorderings ∪ reordering
21:     return reorderings
22:
23: function Apply_Reorderings(reorderings)
24:     for all reodering ∈ reorderings
25:         for all ⟨addr, value⟩ ∈ reordering
26:             Write_Value_To_Memory(addr, value)
27:         Create_Snapshot() // ⑤ - ⑦ na Figura 1
```

conseguimos ter simultaneamente em disco centenas de milhares de estados de uma aplicação, ocupando apenas uma fração do espaço que seria necessário caso não usássemos esta técnica. No entanto, a utilização de CoW é incompatível com DAX (em inglês *direct access*, ou acesso direto) [16,17], que permite uma comunicação direta entre a aplicação e RAM, aumentando a eficiência das aplicações. A incompatibilidade surge da necessidade do *kernel* intercetar as escritas para implementar CoW, eliminando o acesso direto e o ganho de desempenho oferecido por DAX. Mostramos na avaliação o compromisso experimental entre utilizar uma ou outra técnica.

A Figura 2 ilustra um exemplo da aplicação deste algoritmo. Temos 3 endereços (a, b e c) em 2 linhas de cache distintas (X e Y). As escritas têm um *timestamp* único representado por *ts* em ①. Quando a execução chega ao `fence` no final do segmento, a *Shadow* PM encontra-se como em ②. Cada entrada corresponde a um endereço, e cada conjunto de valores possíveis está na forma [*timestamp*, valor]. Todos os endereços têm uma entrada com *timestamp* 0, apesar de este não estar representado no segmento de exemplo. Este *timestamp* serve para representar o valor inicial em memória para cada um dos endereços,

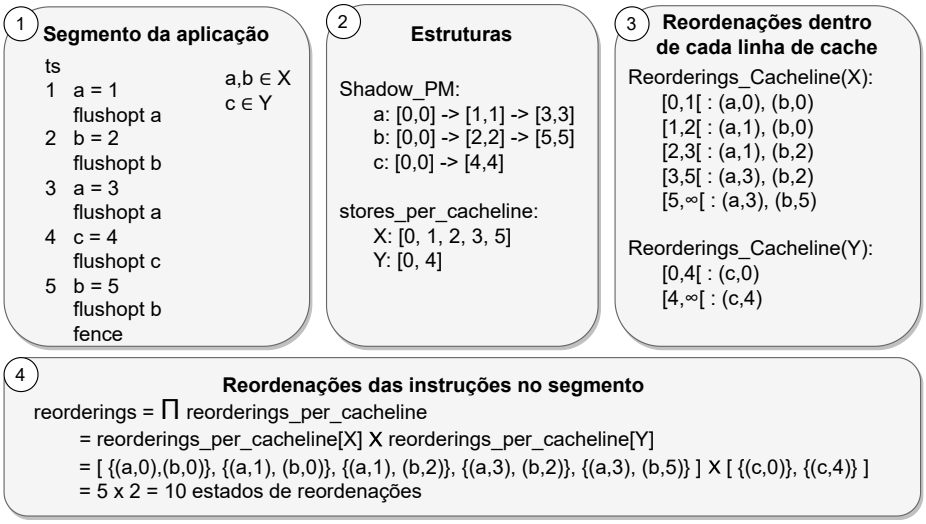


Figura 2: Exemplo da criação dos estados possíveis provenientes das reordenações das instruções.

e deve ser sempre inferior a qualquer *timestamp* dentro do segmento, sendo 0 neste exemplo. A estrutura *stores_per_cacheline* representa os *timestamps* em que cada linha de cache observou uma escrita. Calculamos primeiro as reordenações dentro de cada linha de cache. A caixa ③ representa o *output* de aplicarmos a função *Reorderings Cacheline* (linhas 11 - 21 do Algoritmo 1) às linhas de cache X e Y. A linha Y tem dois *timestamps*, logo tem dois intervalos. Para cada intervalo, obtemos o valor cujo *timestamp* se encontra dentro do mesmo (linha 18 do Algoritmo 1). Após gerarmos as reordenações dentro de cada linha, calculamos o produto cartesiano destes conjuntos e obtemos as reordenações possíveis para o segmento em questão (④). Para este exemplo em que temos 5 escritas, uma reordenação simplista das escritas, sem ter conhecimento das garantias do *hardware*, resultaria em $2^5 = 32$ estados possíveis, sendo a maior parte deles inatingíveis numa execução. Aplicando este algoritmo obtemos apenas 10 estados.

Paralelização do processo de recuperação O processo individual de recuperar a aplicação a partir de um certo estado é completamente independente e, portanto, a recuperação do espaço de estados é inteiramente paralelizável. Uma vez que pré-geramos todo este espaço, conseguimos aumentar a eficiência da ferramenta ao paralelizar o processo de recuperação. O Kaiyo utiliza a recuperação da aplicação como oráculo de coerência para a deteção de *bugs*. Isto significa que, ao recuperarmos a aplicação a partir de um certo estado, detetamos a existência de um *bug* caso a aplicação não consiga recuperar a partir daquele estado, e assumimos a ausência de *bugs* em caso contrário. A maior vantagem que retiramos

da paralelização deste processo é a escalabilidade relativamente ao número de estados que conseguimos recuperar em tempo útil.

Heurísticas De forma a criar um espaço de estados significativo e de dimensão gerível, implementamos um conjunto de heurísticas que excluem estados com menos probabilidade de estarem associados a *bugs*. Para cada heurística, apresentamos a inspiração e o impacto da mesma.

Máximo de escritas por reordenação Existem aplicações que podem ter dezenas, centenas ou até mais escritas entre cada segmento. Nestes casos, o número de estados possíveis provenientes das reordenações destas escritas todas é demasiado alto. De forma a controlar isto, implementamos uma heurística, que limita o número máximo de escritas que reordenamos de cada vez. Ao reordenarmos apenas um conjunto de escritas mais recentes, tiramos proveito do facto de probabilisticamente as escritas mais recentes terem menos chance de terem sido persistidas, e consequentemente poderem levar à existência de *bugs*. Esta heurística é inspirada no Chipmunk [18], cujos autores observam que não há ganho substancial em cobertura ao testar todas as reordenações em segmentos com muitas escritas. Isto significa que é vantajoso testar apenas as reordenações de um número limitado de escritas a cada segmento pois aumentamos a eficiência sem comprometermos a cobertura significativamente.

Idade máxima para uma escrita persistir Por decisão intencional (ou não) do programador, certas escritas podem não ser explicitamente persistidas. Nestes casos, a persistência da escrita fica ao critério do hardware, não havendo qualquer garantia de persistência. Considerar todas estas escritas nas reordenações futuras resulta numa explosão das combinações possíveis. Ademais, o *hardware* invalida as linhas de cache eventualmente, tornando pouco provável que estas escritas não sejam persistidas durante muito tempo. De modo a evitar esta explosão, impomos um limite na idade (em número de segmentos) que uma escrita leva a ser persistida pelo hardware. Segundo um estudo realizado pelos autores do PMDebugger [3], 84.5% das escritas são persistidas nos primeiros dois segmentos que lhes sucedem. Isto significa que um limite de 2 a 4 para a idade máxima de uma escrita já é suficiente para evitar o problema mencionado, sem que se diminua significativamente a cobertura.

eADR Esta nova tecnologia [20] extende o domínio persistente até à cache do CPU, eliminando assim a necessidade de fazer `flush` explicitamente. No entanto, não só esta técnica não resolve todos os *bugs* associados a PM, nomeadamente *bugs* de ordem e atomicidade, dado que as escritas ainda podem ser ordenadas pelo CPU, como requer também *hardware* específico não presente em todos os sistemas. Deste modo, continua a ser necessário que as aplicações para PM suportem o domínio persistente mínimo.

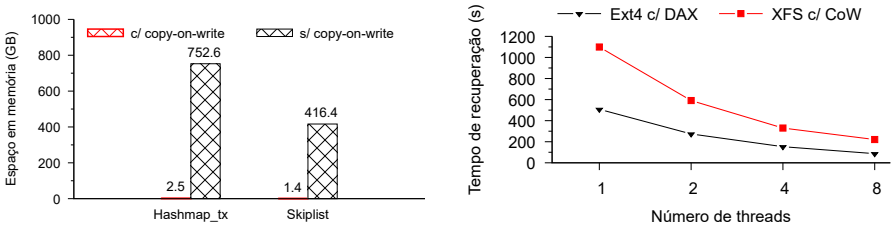
Implementação Realizamos a instrumentação com base no Intel PIN [11]. Implementamos a lógica associada à instrumentação, gestão da informação da PM e criação das cópias em C++. Utilizamos um *script* em *bash* para orquestrar a execução dos vários componentes da nossa ferramenta (Coordenador na Figura 1). Recorremos ao XFS [12] como sistema de ficheiros que suporta *copy-on-write* para a gestão do espaço de estados em disco.

4 Avaliação

Avaliamos o Kaiyo num processador Intel(R) Xeon(R) Gold 6338N com 128 cores a 2.20GHz, com 256GB de RAM e 1TB Intel DCPMM em modo *App Direct*. Em termos de software usamos Ubuntu 22.04, Linux kernel 5.15, Intel Pin 3.24 e Xfs 5.19. Usamos ainda `dstat` para a obtenção de métricas da utilização do CPU. De modo a abrir a comparação futura com outros sistemas no estado da arte, utilizamos um conjunto de aplicações exemplo da `libpmemobj` do PMDK [19] que são amplamente utilizadas e implementam diferentes estruturas de dados em PM. Nomeadamente utilizamos Hashmap Tx e Skiplist. Para cada teste, executamos 3 iterações de uma carga de trabalho com 1000 operações de inserção e 1000 operações de remoção.

Copy-on-write Começamos por avaliar o impacto da utilização de *copy-on-write* (CoW) na geração dos *snapshots*. A Figura 3a ilustra a diferença do espaço ocupado, em GB, entre usar e não usar CoW. Para este efeito consideramos um cenário simplificado do Skiplist em que apenas consideramos a ordem do program pois testar todas as reordenações excederia o espaço disponível na variante sem CoW. Concluímos que existe um ganho considerável ao utilizar CoW, permitindo reduzir em 99.6% o espaço utilizado. Avaliámos também a recuperação em paralelo - os resultados para diferentes números de *threads* são apresentados na Figura 3b. Apesar de no geral o tempo de recuperação com DAX ser inferior ao de CoW, com 8 *threads* a diferença é negligenciável. Tendo em conta os ganhos substanciais de espaço que o CoW oferece, o que permite testar mais estados, no resto da avaliação usamos a configuração com CoW.

Recuperação em paralelo De seguida avaliamos a escalabilidade da recuperação em paralelo. A Figura 4 ilustra o tempo de recuperação de várias aplicações para diferentes números de threads. Para cada teste, apresentamos 6 cenários que correspondem ao uso das heurísticas com diferentes limites. Para cada cenário, H_e representa o limite de escritas que reordenamos e H_i o limite da idade para uma escrita persistir. *NA* significa que não impomos limite. No geral, obtemos um *speedup* significativo até 8 *threads*. A partir deste ponto observámos uma redução drástica do *speedup*. Apesar de uma explicação definitiva para este comportamento carecer de mais experiências, acreditamos tratar-se de contenção no XFS à medida que o número de *threads* aumentam, à semelhança do que foi reportado por outros trabalhos [14,15]. Ainda assim, é possível diminuir em até 5 vezes o tempo de recuperação face a uma abordagem sequencial.



(a) Eficiência espacial com e sem CoW. (b) Eficiência temporal de CoW vs DAX.

Figura 3: Comparação da eficiência espacial e temporal de XFS com CoW versus não utilizar CoW ou utilizar DAX.

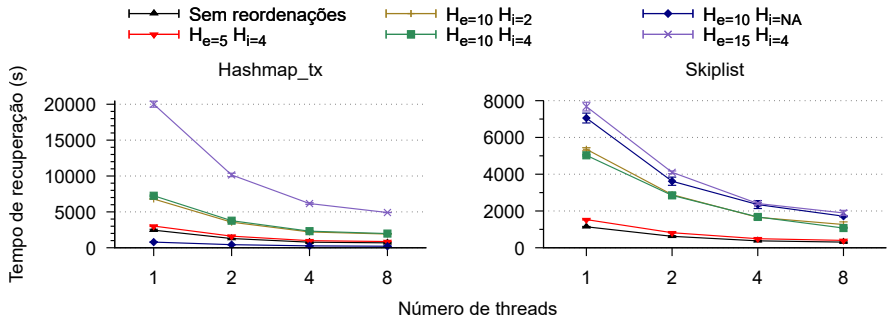


Figura 4: Tempo necessário para recuperar todo o espaço de estados para diferentes números de *threads*, aplicando heurísticas com diferentes limites.

Heurísticas Apresentamos agora o impacto, em várias dimensões, de utilizar as heurísticas para reduzir o espaço de estados proveniente das reordenações das instruções. Os resultados encontram-se compilados na Figura 5. Para cada teste, apresentamos o tempo de criação do espaço de estados, o espaço em disco do mesmo, o número de estados gerados, e quantos destes estados são irrecuperáveis (isto é o programa de recuperação não consegue recuperar a partir desses estados). Alertamos para o facto de cada barra significar o aumento, na respetiva métrica, que uma certa heurística proporciona. Por exemplo, para o tempo de criação dos estados no Skiplist, sem reordenações demora cerca de 400s e com $H_{e=15}$ e $H_{i=4}$ demora cerca de 1800s. Retiramos as seguintes conclusões relativamente às heurísticas: i) o seu impacto e os limites escolhidos dependem muito de cada aplicação; ii) permitem-nos obter um espaço de estados consideravelmente maior do que o da ordem do programa e ainda assim gerível; e iii) levaram-nos a gerar novos estados irrecuperáveis, que podem estar associados a novos *bugs*. Note que a recuperação do Hashmap_tx gera uma percentagem muito elevada de estados irrecuperáveis. Isto deve-se a um *bug* no próprio programa de recuperação que a nossa abordagem ajuda a identificar. No caso do Skiplist, cerca de 20% dos estados gerados são irrecuperáveis, o que é um valor bastante elevado.

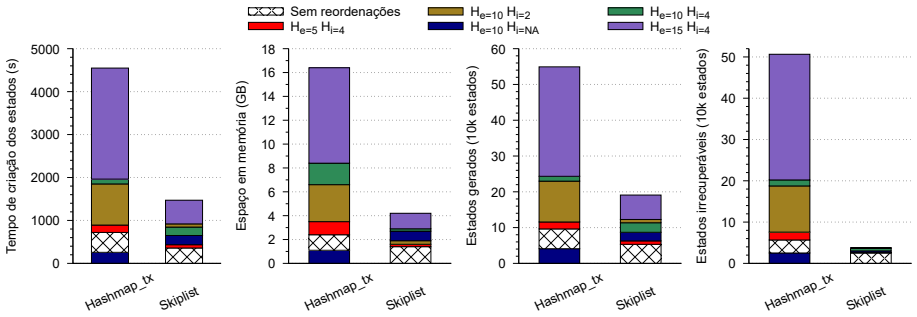


Figura 5: Impacto das heurísticas no espaço de estados. O desvio padrão médio do tempo de criação dos estados é de 21.27s

É de notar que vários estados irrecuperáveis podem ter como causa o mesmo *bug*. Para guiar o programador, quando observamos um estado irrecuperável, reportamos ao utilizador esse estado juntamente com o *backtrace* associado.

5 Conclusão

Este artigo apresenta o Kaiyo, uma ferramenta para testagem de aplicações de PM que é automática, eficiente, e maximiza a cobertura possível em tempo útil. Alcançamos isto ao: pré-gerar o espaço de estados, aproveitar as garantias do *hardware*, aplicar um conjunto de heurísticas que reduz o estado de espaços, e ainda recuperar cada estado em paralelo. A avaliação experimental mostra que as heurísticas nos reduzem o espaço de procura em várias ordens de magnitude, o uso de CoW reduz o espaço em disco em 99.6% e a recuperação em paralelo permite um speedup de 3x no tempo de teste. Desta forma demonstramos que não é necessário comprometer cobertura para ter uma testagem de erros em aplicações de PM escalável.

A nossa abordagem promete alta eficiência na deteção de *bugs*. No entanto, o trabalho apresenta alguns pontos que podem ser explorados em trabalho futuro: i) o Kaiyo foca-se em detetar *bugs* de correção, no entanto, podemos adicionar lógica à *Shadow* PM de forma a detetar *bugs* de desempenho, sem um custo adicional significativo, como fazem outras ferramentas [3]; ii) a escalabilidade é limitada pela contenção do XFS. Seria interessante perceber a origem desta contenção e aumentar o limite prático da recuperação em paralelo; e iii) há uma probabilidade considerável de vários estados serem iguais. Eliminar estados que representem a mesma informação em memória pode diminuir o tempo total de recuperação e, consequentemente, de teste.

Agradecimentos Este trabalho é parcialmente financiado por Fundos Nacionais através da FCT - Fundação para a Ciência e a Tecnologia no âmbito do projeto PTDC/CCI-COM/4485/2021 (Ainur) e UIDB/50021/2020.

Referências

1. Philip, L., Subramanya, D., Sanjay, K., Rajesh, S., Jeff, J.: Yat: A Validation Framework for Persistent Memory Software. In: Procceedings of 2014 USENIX Annual Technical Conference
2. Hamed, G., Guoqing, H., Brian, D.: Jaaru: Efficiently Model Checking Persistent Memory Programs. In: ACM Transactions on Programming Languages and Systems (2021) https://doi.org/10.1145/3445814.3446735
3. Bang, D., Jiawen, L., Hao, C., Dong, L.: Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs. In: ACM Transactions on Programming Languages and Systems (2021) https://doi.org/10.1145/3445814.3446744
4. Sihang, L., Yizhou, W., Jishen, Z., Aasheesh, K., Samira, K.: PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In: ACM Transactions on Programming Languages and Systems (2019) https://doi.org/10.1145/3297858.3304015
5. Sihang, L., Korakit, S., Yizhou, W., Thomas, W., Aasheesh, K., Samira, K.: Cross-Failure Bug Detection in Persistent Memory Programs. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020 https://doi.org/10.1145/3373376.3378452
6. Ian, Ne., Ben, R., Ben, S., Andrew, Q.: AGAMOTTO: How persistent is your persistent memory application? In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (2020)
7. Xinwei, F., Wook-Hee, K., Ajay, P., Mohannad, I., Sunny, W., Dongyoon, L., Changwoo, M.: Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In: ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021 https://doi.org/10.1145/3477132.3483556
8. João, G., Miguel, M., Rodrigo, R.: Mumak: Efficient and Black-Box Bug Detection for Persistent Memory. In: Eighteenth European Conference on Computer Systems (EuroSys'23), May 8–12, 2023 https://doi.org/10.1145/3552326.3587447
9. `https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html\#SafeCopy`
10. Daniel, B., Marco, C.: Understanding the Linux Kernel, 2nd edn. O'Reilly and Associates 2003
11. Intel: Pin - A Dynamic Binary Instrumentation Tool, 2012 `https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html`
12. Anthony, I.: The XFS Linux wiki `https://xfs.wiki.kernel.org/`
13. Azalea, R., John, W., Gil, N., Viktor, V.: Persistency Semantics of the Intel-x86 Architecture. In: Proc. ACM Program. Lang. 4, POPL, Article 11 (January 2020) https://doi.org/10.1145/3371079
14. Dohyun, K., Kwangwon, M., Joontaek, O., Youjip, W.: ScaleXFS: Getting scalability of XFS back on the ring. In: Proceedings of the 20th USENIX Conference on File and Storage Technologies, 2022
15. Rohan, K., Saurabh, K., Soujanya, P., Harshad, S., Gregory, G., Aasheesh, K., Vijay C.: WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In: ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21) https://doi.org/10.1145/3477132.3483567

16. mkfs.xfs man page, https://manpages.ubuntu.com/manpages/focal/man8/mkfs.xfs.8.html, 2019
17. DAX, https://www.kernel.org/doc/Documentation/filesystems/dax.txt
18. Hayley, L., Shankara, P., Om, S., Isil, D., James, B., Vijay, C.: Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In: Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023 https://doi.org/10.1145/3552326.3567498
19. Intel: Persistent Memory Development Kit (PMDK) https://pmem.io/pmdk/
20. Intel: eADR: New Opportunities for Persistent Memory Applications https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html