Deteção de Erros de Concorrência em Programas de Memória Persistente Utilizando Análise de Locksets

João Oliveira, João Gonçalves, and Miguel Matos

IST U.Lisboa & INESC-ID

Resumo A memória persistente (PM, do inglês *Persistent Memory*), permite o desenvolvimento de aplicações com estado persistente, sem a necessidade de operações de E/S custosas. Para uma aplicação tirar partido desta característica, o programador tem de estar ciente da possibilidade de uma falha causar um estado incoerente, e tem de garantir que tais estados não são alcançados ou que serão posteriormente reparados. Negligenciar esta possibilidade pode levar a erros de PM: perdas ou corrupção de informação, falhas subsequentes, ou comportamento não definido. Para detetar estes erros, várias ferramentas e técnicas têm sido propostas. No entanto, as ferramentas para deteção de erros de PM concorrentes são focadas em domínios específicos como bases de dados chaves-valor, e implicam que o programador desenvolva artefactos adicionais. Neste artigo propomos uma nova abordagem de deteção de erros concorrentes de PM, que utiliza análise de *locksets* para a deteção de acessos concorrentes, e uma heurística que permite reduzir significativamente os falsos positivos e o tempo de análise. Os resultados da nossa avaliação preliminar indicam que a nossa abordagem é bastante escalável, reportando os mesmos erros do estado da arte em menos tempo.

1 Introdução

A memória persistente (PM, do inglês *Persistent Memory*) é endereçável ao byte, tal como a memória volátil, e durável tal como o armazenamento através de HDD/SSD. Além disso, uma aplicação que utilize PM para persistir dados pode atingir um desempenho comparável ao de uma aplicação em memória RAM.

No entanto, a incorreta utilização de PM pode levar a um estado incoerente após uma falha, devido à ausência de cache persistente ou à reordenação e atraso de instruções. Uma aplicação correta tem de prevenir ou remediar estas situações, utilizando instruções de baixo nível que permitem persistir escritas por uma ordem explicita, ou rotinas de recuperação responsáveis por corrigir possíveis estados incoerentes causados por uma falha. Não o fazer pode resultar em perdas de dados, falhas posteriores, ou comportamentos não definidos após a recuperação. Estes requisitos dificultam substancialmente o trabalhado de um programador de PM, que se vê obrigado a raciocinar sobre os possíveis estados incoerentes da aplicação e como corrigi-los. Isto motivou a criação de um grande conjunto de ferramentas que permitem a deteção de erros de PM [1–10].

Podemos traçar um paralelo entre a PM e a concorrência - quando utilizada corretamente permite ganhos de desempenho substanciais, mas a incorreta utilização de primitivas concorrentes pode levar a erros difíceis de detetar, sendo que estes dependem de intercalações de `threads` especificas. Dada a sua relevância, a literatura sobre deteção de erros em programas concorrentes é vasta [11].

A PM e a concorrência podem ser um par ideal para aplicações com altos requisitos de desempenho, no entanto, esta combinação origina num desafio complexo de programação. Para implementar corretamente uma aplicação concorrente de PM, um programador tem de raciocinar sobre o estado de persistência dos dados, a ordem pela qual são persistidos, a sincronização dos acessos a secções critica, e o impacto que diferentes intercalações de `threads` podem ter. Este desafio acrescido levou a propostas recentes para a deteção de erros concorrentes em programas de PM [9,10]. No entanto, estes trabalhos têm limitações como a abrangência, sendo que ambos se focam apenas num tipo específico de aplicação, e a automação, pois ambos necessitam da criação de artefactos de código não triviais por cada aplicação a verificar.

Para responder a estas necessidades apresentamos uma nova ferramenta, que utiliza análise de locksets para detetar acessos concorrentes a memória persistente, suportando qualquer aplicação de PM e não obrigando à criação de artefactos complexos por aplicação a validar. A ideia chave da nossa abordagem assenta na observação que os acessos a PM representam uma pequena fração de todos os acessos à memória, cerca de 4% [12], o que nos permite tirar partido de técnicas mais exaustivas e completas, como a análise de *locksets*. Estas técnicas, inicialmente propostas para programas concorrentes, apresentam alguns problemas de escalabilidade nesse contexto, dado o espaço de procura ser bastante vasto, contudo quando aplicadas a programas de PM concorrentes, que têm um espaço de procura muito menor, apresentam resultados promissores em termos de deteção de erros e de desempenho. É importante notar que, por uma questão de simplicidade, a abordagem atual assume que acessos concorrentes a PM, em que pelo menos um deles é uma escrita, consistem sempre num erro.

A avaliação preliminar demonstra que a nossa ferramenta é escalável e permite a deteção dos erros até agora reportados pelo estado da arte, em menos tempo, de uma forma automática e agnóstica à aplicação alvo.

2 Semântica de Memória Persistente

Os programadores de PM necessitam de ter em consideração a durabilidade das escritas e a ordem pela qual elas ocorrem, pois, não é garantido que estas executem pela ordem em que aparecem no programa [13].

Para controlar a durabilidade das escritas e a ordem pela qual são executadas, o hardware oferece algumas instruções aos programadores de PM. Estas instruções variam entre arquiteturas, no entanto, na sua generalidade incluem as seguintes instruções:

As `escritas` são o método mais comum que os programas utilizam para escrever em memória. Por razões de otimização o CPU pode atrasar a execução

destas instruções, o que implica que o programa pode ter executado uma ou mais escritas sem realmente alterar qualquer valor em cache, ou em memória.

As `escritas não temporais` pertencem a um subconjunto das instruções de escrita, ignoram a cache e escrevem diretamente na memória, no entanto, também podem ser atrasadas tal como as outras escritas.

Os `flushes` despejam uma linha de cache para memória, no entanto, podem ser reordenados em relação a outras instruções, por exemplo, outro *flush*. É importante frisar que esta não é a única maneira em que dados são despejados da cache para memória, a cache pode decidir fazê-lo arbitrariamente dependendo das suas políticas de despejo. As instruções de *flush* apenas permitem fazer este despejo explicitamente em vez de arbitrariamente.

O `fence` atua como uma barreira para o atraso e reordenação de instruções mencionado acima. Uma instrução de *fence* só termina quando todas as instruções em atraso são completadas. Por exemplo, dois *flushes* separados por um *fence* obrigam o CPU a executar o primeiro *flush* antes do segundo.

Para prevenir estados incoerentes o programador tem de despejar linhas de cache explicitamente, e ordenar os despejos com a utilização de *fences*.

Nota sobre tecnologia eADR Os novos processadores da Intel vêm acompanhados de *extended Asynchronous DRAM Refresh*, uma tecnologia que suporta uma cache persistente [14], garantindo que todas as escritas são imediatamente persistidas quando chegam à cache. A tecnologia eADR implica um custo adicional de hardware o que limitará a sua adoção num futuro próximo. Por esse motivo, defendemos que as aplicações de PM não devem depender da sua existência para garantir coerência.

3 Trabalho Relacionado

O estado da arte da deteção de erros concorrentes de PM pode dividir-se em duas categorias: ferramentas que procuram erros de concorrência de forma ativa, e ferramentas que o fazem de forma passiva. Deixamos fora desta discussão o Agamotto [8], pois não suporta execuções concorrentes.

A maioria dos trabalhos deteta esta classe de erros de forma passiva [1–7]. Estes trabalhos analisam uma execução concreta, apenas detetando erros causados por concorrência se a intercalação de threads ocorrida nessa execução tiver exposto esse erro. Não raciocinam sobre possíveis intercalações alternativas, nem determinam se um erro é um efeito secundário da utilização de concorrência.

Por contraste, existem dois trabalhos que atacam este problema de forma ativa. O PMRace [10], que está dividido em duas fases distintas, a primeira é responsável por encontrar escritas baseadas em dados não persistidos, utilizando uma técnica de injeção de atrasos em pontos chaves da execução onde ocorrem escritas frequentes para PM, especialmente em zonas de memória acessíveis a outras threads. Para aumentar a cobertura, é usado um motor de fuzzing para gerar cargas de trabalho. A segunda fase do PMRace utiliza *taint analysis* para garantir que os acessos detetados originam alguma falha. O Durinn [9] testa a

linearizabilidade durável de operações, ou seja, se estas são atómicas mesmo após falhas à granularidade da instrução [4]. Reorganiza as operações de uma carga de trabalho para procurar acessos concorrentes problemáticos, e depois executa essas operações concorrentemente forçando uma falha no sistema. Após a recuperação, aplica uma bateria de testes especializada para validar a coerência do estado. Ambas as abordagens descritas assumem que a aplicação alvo tem a semântica de uma base de dados chave-valor (ou redutível a uma). Este pressuposto implica ainda a criação de artefactos de código não triviais por aplicação a verificar, que apesar de semelhantes na essência são distintos para cada ferramenta, e consistem fundamentalmente num mapeamento entre as operações da base de dados, dos seus argumentos e resultados. Este pressuposto reduz a aplicabilidade das técnicas fora desse contexto, para além de requerer uma quantidade de trabalho manual não negligenciável.

Em suma, o estado da arte na deteção de erros concorrentes em programas de memória persistente fica aquém em relação à automação e à abrangência, obrigando o utilizador a produzir artefactos complexos para cada aplicação, e apenas suportando bases de dados chave-valor ou equivalente. Esta analise motiva o nosso trabalho no desenvolvimento de uma ferramenta automática e abrangente que apresentamos de seguida.

4 Abordagem

Nesta secção descrevemos exemplos de acessos concorrentes a PM, como estes podem originar erros, e qual a nossa abordagem para os detetar de uma forma automática e abrangente.

4.1 Acessos Concorrentes a PM

O nosso trabalho foca-se em detetar erros concorrentes de PM, ou seja, erros de PM originados pela incorreta utilização de concorrência. Mais concretamente, a incorreta sincronização de acessos concorrentes a variáveis em PM.

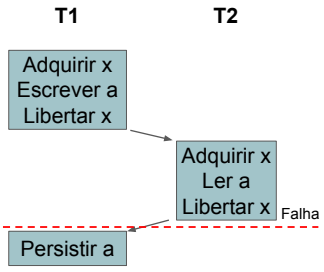


Figura 1: Exemplo de acesso concorrente a uma variável em PM

Por exemplo, na Figura 1 observamos um erro de concorrência na utilização de PM. Neste programa, uma *thread* `T1` escreve para a variável `A` dentro da secção critica `X`. Concorrentemente, a *thread* `T2` lê `A`, também dentro de `X`. No entanto, dado que T1 apenas persiste a escrita para `A` fora da secção critica, é possível que `T2` leia um valor não persistido que pode resultar em efeitos persistentes não desejados. Dependendo da aplicação, esses efeitos poderiam ser uma alteração ao estado da aplicação (ou um comportamento visível para o exterior) visível após a falha, o que constitui um erro.

Para corrigir o erro descrito na Figura 1 é necessário colocar a persistência dentro da secção critica. Essencialmente deve garantir-se que a escrita e a sua respetiva persistência é atómica em relação a outros acessos ao mesmo endereço.

Note-se que, em PM uma escrita só está garantidamente persistida se for executado um `flush` seguido de um `fence`. Logo, apesar das primitivas de sincronização introduzirem `fences` no início e fim das secções criticas, isso não é suficiente para garantir persistência.

4.2 Arquitetura

Para detetar os acessos concorrentes identificados acima, desenvolvemos uma ferramenta que utiliza análise de *locksets* de forma automática e agnóstica da aplicação alvo.

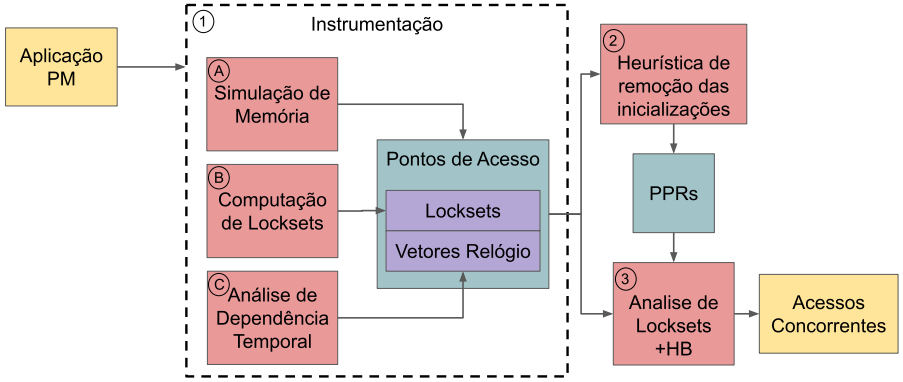


Figura 2: Esquemática da arquitetura da ferramenta

A nossa abordagem é composta por três etapas representadas na Figura 2. A primeira etapa, a instrumentação ①, é responsável por recolher informação sobre a execução e os vários acessos a PM. De seguida, aplicamos uma heurística ② para detetar pontos de acesso propícios a revelar races (PPRs) e por fim confirmamos os pontos detetados utilizando análise de *locksets* ③.

4.3 Etapa 1: Instrumentação

A primeira etapa da ferramenta é a instrumentação. Está dividida em três funções distintas: Ⓐ a simulação de memória, que determina em que pontos da execução a PM foi acedida e persistida, Ⓑ a computação de *locksets*, que mantém o conjunto atual de trincos adquiridos, e Ⓒ a análise de dependência temporal, que calcula os vetores de relógio de cada ponto da execução.

Simulação de Memória Para detetar os pontos da execução em que uma escrita em PM é persistida é necessário manter o estado da cache, simulando o comportamento das operações de escrita, *flush* e *fence* pela ordem que são executadas. Para tal, mantemos um registo de todas as escritas, e o seu estado de persistência, quando uma escrita é persistida, registamo-la para ser posteriormente avaliada. Em particular, estamos interessados em saber qual o ponto, no pior caso, em que uma escrita é persistida. Ou seja, simulamos uma cache que apenas despeja dados após uma instrução de *flush*, e um CPU que atrasa todas as operações de escrita até atingir um *fence*.

Computação de *Locksets* Cada *thread* mantém o atual conjunto de trincos adquiridos, quando um trinco é adquirido é adicionado ao conjunto e quando é libertado é removido. Além disso, é registado uma etiqueta temporal relativa ao momento em que cada trinco é adquirido. A etiqueta temporal é calculada a partir dum contador por *thread* para cada instrução registada (acessos a PM e operações de sincronização). Os *locksets* são utilizados durante a execução quando são registados acessos a PM.

Análise de Dependência Temporal A terceira função da instrumentação é calcular as dependências temporais das operações entre *threads*. Para codificar esta relação temporal, utilizamos vetores relógio [15], em que a criação e junção de *threads* correspondem a eventos com uma dependência temporal. Cada *thread* mantém o seu vetor, e quando uma operação de criação ou junção de *threads* ocorre, as *threads* envolvidas atualizam o vetor. Quando é registado um acesso a PM, também é registado o vetor correspondente ao acesso.

4.4 Etapa 2: Heurística de Remoção de Inicializações

Antes de fazer a análise dos *locksets* é importante selecionar os acessos que realmente serão propícios a resultar num acesso concorrente (PPR), tanto para reduzir o tempo da análise como para reduzir o número de falsos positivos.

A primeira intuição seria comparar todos os acessos. No entanto, por vezes as variáveis são acedidas antes de o seu apontador ser publicado, por exemplo, durante as inicializações. É correto e, na generalidade, mais eficiente não proteger estes acessos, pois nunca poderão ocorrer concorrentemente com acessos noutras *threads*. Para evitar o reporte incorreto destes acessos como concorrentes a nossa ferramenta aplica uma heurística de remoção de inicializações, semelhante ao

Algorithm 1: Analise de locksets

```
input: O conjunto de todas as escritas, writes
       O conjunto de todas os acessos, accesses
foreach (tid, write) in writes do
    // tid representa o identificador da thread
    foreach (tid', access) in accesses where tid ≠ tid' do
        if write.lockset ∩ access.lockset = ∅ then
            report(write, access);
        end
    end
end
```

que é feito noutros trabalhos de análise de *locksets* [16]. Resumidamente esta heurística ignora todos os acessos feitos pela primeira *thread* a uma variável até que uma segunda *thread* lhe aceda, pois, a partir desse ponto, sabe que o apontador é publico. Para estender esta heurística para PM, ignoramos as escritas cuja persistência ocorreu antes de a segunda *thread* ter acedido à variável.

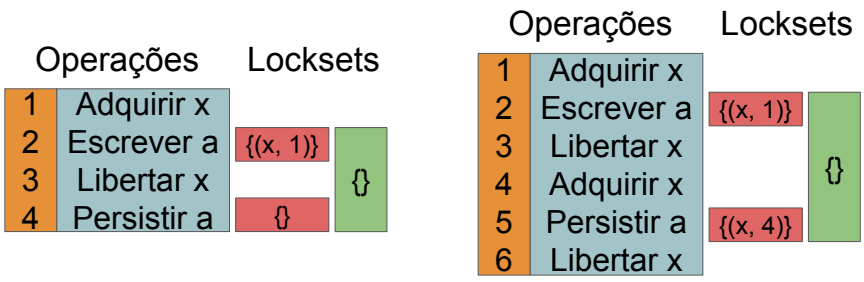
4.5 Etapa 3: Análise de *Locksets*

Nesta secção vamos construir a solução incrementalmente, descrevendo como estendemos um algoritmo de análise de *locksets* tradicional, para detetar possíveis acessos concorrentes em PM. Comece por considerar o Algoritmo 1, que não tem em conta a diferença que existe entre o momento em que uma escrita é feita e aquele em que é persistida. Para cada par de acessos (em que pelo menos um é uma escrita) é feita a intersecção dos *locksets*. Caso a interseção resulte num conjunto vazio então não existe um trinco comum que proteja ambos os acessos, o que significa que aquele par constitui um possível acesso concorrente.

A primeira extensão a ser feita ao algoritmo é a incorporação da persistência. Queremos garantir que a escrita e a persistência são atómicas em relação aos outros acessos ao mesmo endereço. Para tal, computamos a interseção dos *locksets* na escrita e na persistência, ao que chamamos o *lockset* efetivo. No entanto, fazer apenas a interseção dos *locksets* da escrita e da persistência pode ignorar situações em que está presente um acesso concorrente. Ou seja, dois pontos da execução podem estar protegidos pelo mesmo trinco e mesmo assim não serem atómicos, por exemplo, se o trinco for liberto e readquirido entre a escrita e a persistência. Para detetar estas situações, estendemos os *locksets* com a etiqueta temporal em que o trinco é adquirido. A Figura 3 contém dois exemplos da aplicação desta extensão para calcular o *lockset* efetivo de uma escrita em PM. Do lado esquerdo, a persistência está fora da secção critica logo independentemente da etiqueta de temporal, o *lockset* efetivo da escrita será um conjunto vazio. Do lado direito, a escrita e a persistência estão protegidas pelo mesmo trinco. Sem a adição da etiqueta temporal o *lockset* efetivo da escrita não seria vazio, no entanto, é claro que ambas as instruções não pertencem à mesma secção critica.

Algorithm 2: Analise de locksets em PM

```
input: O conjunto de todos os PPRs, PPRs
       O conjunto de todas os acessos, accesses
foreach (tid, write, persist) in PPRs do
    foreach (tid', access) in accesses where isConcurrent(tid, write, tid',
     access) do
        efective_lockset ← write.lockset ∩ persist.lockset;
        if efective_lockset ∩ access.lockset = ∅ then
            report(write, access);
        end
    end
end
```



(a) (b)

Figura 3: Exemplos de *locksets* com base em etiquetas temporais. A primeira coluna a laranja contem a etiqueta temporal de cada instrução na coluna azul. A coluna vermelha contem o *lockset* com etiqueta temporal de cada instrução. A coluna verde contem o *lockset* efetivo da escrita.

Com o *lockset* efetivo, o Algoritmo 2 faz a intercessão com os *locksets* dos outros acessos tal como na análise de *locksets* tradicional.

A segunda extensão visa reduzir o número de falsos positivos devido às dependências temporais das operações. Por exemplo, assuma que a *thread* A faz uma escrita persistida para PM e de seguida cria a *thread* B que faz uma leitura do mesmo endereço. Neste exemplo percebemos facilmente que o segundo acesso apenas pode ocorrer depois do primeiro. Ou seja, independentemente da intercalação das *threads*, os acessos nunca executarão concorrentemente. Nestas situações, é comum um programador não proteger um ou ambos os acessos com trincos, pois tem a certeza que nunca ocorrerão em simultâneo, o que pode originar um falso positivo. É esta observação que codificamos em *isConcurrent*, que nos permite deduzir à partida se dois acessos nunca iram executar concorrentemente através dos vetores relógio calculados anteriormente.

Nota sobre cobertura Ao contrário de outros métodos de análise que só permitam analisar acessos que efetivamente ocorreram concorrentemente, a análise de `locksets` permite analisar quaisquer pares de acessos que sejam possivelmente concorrentes, o que expande consideravelmente o espaço de procura. No entanto, a implementação atual da nossa ferramenta analisa um `trace` concreto da execução, limitado às intercalações observadas e à cobertura da carga de trabalho.

5 Avaliação

Nesta secção apresentamos a avaliação da nossa abordagem. Selecionamos o Fast-Fair [17] (*commit* 0f047e8) como aplicação alvo por já ter sido testado noutros trabalhos, como o PMRace, e portanto servir de base de comparação. Adicionalmente o Fast-Fair faz um uso avançado de PM e várias das operações são *lock-free* o que torna a testagem mais complexa. Mais concretamente, o Fast-Fair é uma implementação de uma B+-Tree em PM tolerante a falhas em que as operações nos nós internos da árvore executam sem recorrer a trincos, e as operações nos nós folha estão protegidas por trincos. Sendo um dos objetivos do trabalho desenvolver uma abordagem automática, que não necessita de artefactos extraordinários, utilizamos a carga de trabalho fornecida pela aplicação. Esta consiste em duas fases: a primeira fase é composta por operações de inserção executadas sincronamente para popular a árvore, e na segunda fase as operações são executadas concorrentemente por 8 *threads*, com uma divisão de 19.17% inserções, 76.67% procuras e 4.17% remoções.

As experiências foram efetuadas numa máquina com um processador Intel(R) Xeon(R) Gold 6338N CPU @ 2.20GHz, com 256 GB de RAM e 128 cores, e um Intel DCPMM de 1 TB em *App Direct mode*.

5.1 Resultados

Para aferir o impacto da heurística de remoção de inicializações efetuamos os testes com e sem essa heurística ativa, e avaliamos quatro cenários em que vamos aumentando progressivamente a carga de trabalho: 3 125, 15 625, 31 250 e 156 250 operações. O tamanho da carga de trabalho é dado pelo código fornecido pelo Fast-Fair, na fase concorrente, para cada *input*, são executadas 5 ou 6 operações do Fast-Fair. O número de *inputs* para cada carga de trabalho é respetivamente: 1 000, 5 000, 10 000 e 50 000. A Figura 4 resume os resultados da nossa avaliação em função do número de operações do Fast-Fair, concretamente os tempos de execução, utilização de memória, PPRs analisados, e erros reportados.

Na Figura 4a reportamos o número de PPRs que, como esperado, é diretamente influenciado pelo tamanho da carga de trabalho, sendo que um aumento do número de escritas implica um crescimento nos pontos a analisar. Também observamos que a heurística de remoção de inicializações é bastante eficaz, pois leva a uma redução de PPRs em cerca de uma ordem de magnitude.

Na Figura 4b observamos que o tempo de instrumentação, que corresponde ao tempo de execução da aplicação instrumentada, domina fortemente o tempo

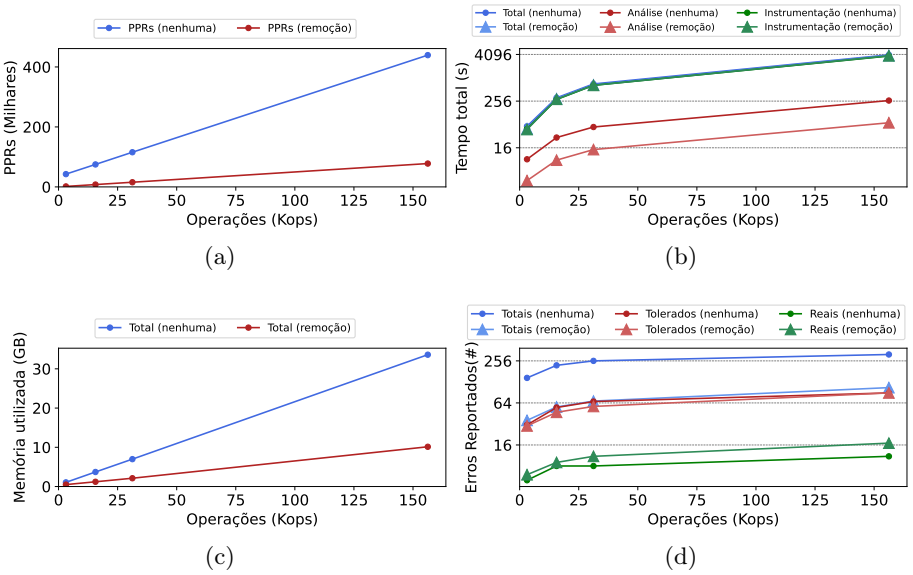


Figura 4: (a) PPRs analisados em função do número de operações do Fast-Fair (b) Tempo de execução em função do número de operações do Fast-Fair (c) Memória no pico em função do número de operações do Fast-Fair (d) Erros reportados em função do número de operações do Fast-Fair

total de execução (concretamente, 93% sem a heurística e 98% com a heurística). Uma grande parte do tempo de instrumentação deve-se à coleção de *backtraces* das operações em PM (não mostrado na figura). A utilização do CPU mantém-se constante independentemente da utilização da heurística, entre os 110% e 115%, o que significa que ocupamos pouco mais de um core do CPU. Apesar das várias funções da instrumentação descritas na Secção 4.3 serem na sua generalidade paralelizáveis ou terem um impacto pequeno no tempo de execução, a coleção de *backtraces* é feita em série, e, em conjunto com os pontos de contenção que naturalmente ocorrem na execução do Fast-Fair, a ferramenta não consegue tomar partido dos vários cores disponíveis.

Na Figura 4c está representado a quantidade de memória utilizada correspondente ao pico de memória reservada. A nossa abordagem implica manter os dados de cada acesso até ao fim da execução logo a utilização de memória cresce com a duração da carga de trabalho.

O tempo total da execução e a utilização de memória aumenta linearmente em função do número de operações, o que demonstra que a ferramenta é escalável para cargas de trabalho maiores, permitindo o reporte de erros menos comuns. Por exemplo, reportamos um acesso concorrente numa função que apenas executa quando a árvore cresce o que não seria reportado com cargas mais pequenas. O tempo da análise de *locksets* e a memória utilizada são reduzidos

pela heurística de remoção das inicializações, pois reduz significativamente o número de pontos a analisar. Concretamente, reduz o tempo de análise em até 73% e a utilização de memória até 69%.

Em suma, mesmo para cargas de trabalhos grandes, a nossa abordagem é eficiente quer em termos de utilização de CPU, quer em termos de consumo de memória, efetuando a análise, mesmo para as maiores cargas de trabalho em cerca de uma hora.

Por fim, na Figura 4d apresentamos os erros reportados pela nossa ferramenta. No contexto desta análise, um erro constitui um par de caminhos que constituem um acesso concorrente. A azul representamos o número total de erros reportados, a vermelho os erros tolerados, que acessos concorrentes são suportados pelo desenho do Fast-Fair, e a verde os erros reais, que correspondem a acessos concorrentes não suportados pelo desenho do Fast-Fair, ou seja, correspondem a erros na implementação.

É de notar que a nossa ferramenta encontra erros expostos ao programador sob a forma de um *backtrace* que o ajudará a encontrar e resolver o erro. Apesar de os caminhos reportados pela ferramenta serem garantidamente únicos, na prática, vários destes caminhos terão uma causa comum (isto é devem-se a um único erro na aplicação).

De seguida discutimos os resultados obtidos. A heurística de remoção das inicializações apenas reduz o número de acessos a analisar, logo, à primeira vista, não é claro como o número de erros verdadeiros reportados pode aumentar quando utilizamos a heurística como representado na Figura 4d pelas linhas verdes. Isto deve-se aos seguintes fatores. O desenho do Fast-Fair implica que existem vários caminhos possíveis de chegar ao mesmo ponto da execução, através de recursão. O desenho da carga de trabalho executa a mesma operação em vários pontos distintos, portanto, apesar de o erro ser na operação, cada uma destas invocações terá um caminho diferente. Concretamente, o número de erros reportados depende do `trace` observado em cada execução da aplicação. Cada uma destas experiências corresponde a uma execução diferente logo, mesmo cargas de trabalhos iguais, podem ter resultados diferentes.

Como mencionámos acima, o desenho do Fast-Fair incluí algumas operações *lock-free* o que leva a que alguns acessos concorrentes não comprometam o correto funcionamento do sistema. Para tal, para cada erro reportado analisámos manualmente se é tolerado pelo desenho do Fast-Fair. A realidade é que o Fast-Fair é um programa fora do comum devido à sua complexidade, boa implementação, e desenho especificamente criado para tolerar estes acessos concorrentes. Este desenho é, portanto, dos mais complexos de analisar para ferramentas de análise de erros como a nossa, sendo que o PMRace [10] encontra este mesmo padrão de comportamento no Fast-Fair.

Os erros verdadeiros que reportamos correspondem ao erro reportado pelo PMRace [10], no entanto, tal como indicamos acima, reportamos os vários caminhos possíveis, o que pode facilitar o programador a diagnosticar a causa.

Por fim, os falsos positivos indicam acessos concorrentes reportados, mas que nunca podem ocorrer. Por exemplo, acessos com uma dependência de ordem

(uma leitura que espere até uma escrita estar persistida através de uma *flag*) ou inicializações. Na nossa abordagem atual, eliminamos automaticamente alguns dos falsos positivos através da análise de dependência temporal e da heurística da remoção das inicializações.

O PMRace utiliza injeção de atrasos para identificar acessos concorrentes no momento em que eles ocorrem. A nossa abordagem, permite detetar estes acessos sem eles ocorrerem em simultâneo, o que sugere que a nossa abordagem é mais rápida. Não efetuamos uma análise experimental concreta entre as duas ferramentas, no entanto, podemos fazer uma comparação de forma lata. Com a carga de trabalho mais pequena, que executa num tempo extremamente reduzido (cerca de 1 minuto), conseguimos detetar o mesmo erro real detetado pelo PMRace. Em contrapartida, a deteção de acessos concorrentes do PMRace é na generalidade medida em horas. Além disso, a nossa abordagem é automática e agnóstica da aplicação alvo, o que na nossa opinião constitui uma vantagem. Na secção seguinte discutimos os próximos passos para resolver ou mitigar as lacunas atuais na nossa abordagem.

6 Conclusão

Neste artigo apresentamos uma abordagem de deteção de erros concorrentes em aplicações de PM de forma automática e agnóstica. A nossa principal observação é que as regiões de PM com acessos concorrentes são uma pequena fração do programa o que nos permite tirar partido de técnicas mais exaustivas e completas, como a análise de *locksets*. Expandimos um algoritmo tradicional de análise de *locksets* para detetar acessos concorrentes a variáveis de PM, e introduzimos uma heurística para escolher os acessos a analisar com o intuito de reduzir o tempo de análise e o número de falsos positivos. Os resultados da nossa avaliação preliminar indicam que a nossa abordagem é bastante escalável, efetuando uma análise correspondente à primeira fase do PMRace em pouco tempo com resultados semelhantes. Além disso, isto é feito de uma forma automática e agnóstica da aplicação, ao contrário do PMRace.

Apesar de detetarmos os mesmos erros do estado da arte a nossa abordagem não contempla uma verificação forte de cada erro. Por exemplo, se um acesso concorrente é tolerado pelo desenho da aplicação, nós reportamos como um erro. Isto implica uma fase de validação manual mais significativa do que o desejado. Como trabalho futuro, queremos investigar possíveis técnicas automáticas e agnóstica à aplicação alvo para a confirmação de erros, para aliviar o esforço manual do utilizador, comparável à segunda fase da análise do PMRace [10].

No âmbito da implementação, planeamos suportar outras bibliotecas de sincronização para além do *pthread* com o auxílio de um ficheiro de configuração simples que só terá que ser produzido uma vez por cada solução de sincronização.

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. P. Lantz, S. Dulloor, S. Kumar, R. Sankaran, and J. Jackson, "Yat: A validation framework for persistent memory software," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 433–438. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz
2. H. Gorjiara, H. Xu, and B. Demsky, "Jaaru: efficiently model checking persistent memory programs," 04 2021, pp. 415–428.
3. H. Gorjiara, G. H. Xu, and B. Demsky, "Yashme: detecting persistency races," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 830–845. [Online]. Available: https://doi.org/10.1145/3503222.3507766
4. X. Fu, W.-H. Kim, A. P. Shreepathi, M. Ismail, S. Wadkar, D. Lee, and C. Min, "Witcher: Systematic crash consistency testing for non-volatile memory key-value stores," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 100–115. [Online]. Available: https://doi.org/10.1145/3477132.3483556
5. S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 411–425. [Online]. Available: https://doi.org/10.1145/3297858.3304015
6. B. Di, J. Liu, H. Chen, and D. Li, "Fast, flexible, and comprehensive bug detection for persistent memory programs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 503–516. [Online]. Available: https://doi.org/10.1145/3445814.3446744
7. S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan, "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*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1187–1202. [Online]. Available: https://doi.org/10.1145/3373376.3378452
8. I. Neal, B. Reeves, B. Stoler, A. Quinn, Y. Kwon, S. Peter, and B. Kasikci, "AGAMOTTO: How persistent is your persistent memory application?" in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1047–1064. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/neal
9. X. Fu, D. Lee, and C. Min, "DURINN: Adversarial memory and thread interleaving for detecting durable linearizability bugs," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 195–211. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/fu

10. Z. Chen, Y. Hua, Y. Zhang, and L. Ding, "Efficiently detecting concurrency bugs in persistent memory programs," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 873–887. [Online]. Available: https://doi.org/10.1145/3503222.3507755
11. H. Fu, Z. Wang, X. Chen, and X. Fan, "A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques," *Software Quality Journal*, vol. 26, pp. 855–889, 2018.
12. S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 135–148. [Online]. Available: https://doi.org/10.1145/3037697.3037730
13. A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis, "Persistency semantics of the intel-x86 architecture," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, dec 2019. [Online]. Available: https://doi.org/10.1145/3371079
14. "eADR: New Opportunities for Persistent Memory Applications." [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html
15. C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.
16. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.
17. D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in {Byte-Addressable} persistent {B+-Tree}," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 187–200.