DONUTS: Um Eficiente Método de Checkpointing em Memórias Não Voláteis

Kleber Kruger, Rodolfo Azevedo, Ricardo Pannain

[1] Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas, SP – Brazil

kleber.kruger@ufms.br, rodolfo@ic.unicamp.br, pannain@ic.unicamp.br

Abstract. *Systems with non-volatile memory (NVM) need to ensure crash consistency. Among main challenges in these systems are creating viable checkpointing mechanisms in terms of performance and usability, for this it is necessary to reduce the amount of writes in NVM, because the excessive increase generates higher bandwidth usage and consequently degrades performance. This work proposes DONUTS, a software-transparent mechanism that generates dynamic epochs through checkpoints integrated to the cache replacement policy. Evaluations show that compared to a previous best performing system, our strategy reduced writes to NVM by 53.8%, providing crash consistency with less than 1% of runtime overhead.*

Resumo. *Os sistemas com memória não volátil (NVM) precisam ser consistentes à falhas. Dentre os principais desafios está criar mecanismos de checkpointing viáveis em termos de desempenho e usabilidade, para isso é necessário reduzir o número de escritas na NVM, pois o aumento excessivo gera maior uso de largura de banda e, consequentemente, degrada o desempenho. Neste trabalho é proposto DONUTS, um mecanismo transparente ao software que gera épocas dinâmicas por meio de checkpoints integrados à política de substituição de cache. Comparado ao sistema anterior de melhor desempenho, o método proposto reduziu a quantidade de escritas na NVM em 53,8%, fornecendo consistência à falhas com menos de 1% de overhead de tempo de execução.*

1. Introdução

A memória não volátil (NVM - *Non-Volatile Memory*)[1] é uma tecnologia que está emergindo como um novo componente da hierarquia de memória e armazenamento [Wu et al. 2019, Wei et al. 2020]. Por fornecer persistência de dados com latência de acesso comparável à DRAM, mas com menor consumo de energia e maior densidade de armazenamento, muitos trabalhos têm utilizado as NVMs como memória principal, seja para aumentar ou substituir completamente as DRAMs [Cai et al. 2020]. Exemplos de memórias não voláteis incluem a *Phase Change Memory* (PCM) [Wu et al. 2013], *Spin-Torque Transfer* RAM (STT-RAM) [Noguchi et al. 2015], *Resistive* RAM (ReRAM) [Fackenthal et al. 2014] e a 3D XPoint [Intel 2015].

[1]Diferentes terminologias são empregadas na literatura para denominar as memórias não voláteis endereçáveis por byte. Além do termo *Non-Volatile Memory* (NVM) usado neste trabalho, outros são: *Persistent MEMory* (PMEM ou PM), *Storage Class Memory* (SCM), *Non-Volatile RAM* (NVRAM), *Byte-addressable Persistent RAM* (BPRAM), *Non-Volatile Main Memory* (NVMM), *Non-Volatile DIMM* (NV-DIMM) and *Non-Volatile Byte-addressable Memory* (NVBM).

Entre as vantagens das arquiteturas NVM está a possibilidade das aplicações acessarem dados persistentes diretamente na memória principal por meio de operações *load*/*store* sem a necessidade de serializar/desserializar os dados e paginá-los dentro/fora dos dispositivos de armazenamento [Wei et al. 2020]. Porém, essa possibilidade impõe novos desafios, sobretudo, a necessidade de garantir a consistência dos dados na presença de falhas do sistema [Liu et al. 2017]. Considerando-se o exemplo de duas linhas de cache sendo modificadas como parte de uma atualização atômica para uma estrutura de dados, se o sistema travar após somente uma das linhas da cache atingir a NVM, a estrutura de dados é deixada em um estado inconsistente por causa da atualização parcial na memória persistente [Joshi et al. 2017]. Portanto, estas novas arquiteturas precisam garantir que os dados na NVM possam ser recuperados para uma versão consistente, chamada de *checkpoint*. Esta propriedade denomina-se consistência de falhas [Ren et al. 2015].

Garantir a consistência de falhas tem sido um desafio para os sistemas de computação com NVM. As alterações dos dados persistentes precisam ser consistentes e efetuadas mediante transações atômicas, isto é, todas as escritas de uma atualização precisam ser realizadas com sucesso ou nenhuma deve ser efetivada. As soluções propostas na literatura costumam apresentar três principais problemas: 1) consistem de estratégias não transparentes ao software que consequentemente restringem o uso de NVM às aplicações baseadas em modelos de programação de memória transacional; 2) geram operações de persistência no caminho crítico de execução; e 3) aumentam substancialmente a quantidade de escritas na memória não volátil, pois, além das escritas convencionais da aplicação, os mecanismos de consistência também precisam gravar os dados de recuperação em regiões específicas da NVM. Isso intensifica o uso de largura de banda no barramento e, como resultado, causa degradação de desempenho em aplicações com alta demanda de acesso à memória.

Neste trabalho é proposto o método DONUTS (*DON't paUse The perSistence*), um mecanismo de hardware cuja contribuição é fornecer consistência a falhas de maneira transparente ao software por meio de *checkpoints* assíncronos e menor uso de largura de banda. Seu principal diferencial é a integração dos *checkpoints* à política de substituição de cache, cujos benefícios são: épocas com intervalos dinâmicos e persistência assíncrona em *group-commit*[2], conceito amplamente conhecido e empregado em sistemas de bancos de dados, mas pouco explorado em arquiteturas NVM. Os resultados mostraram que comparado ao sistema anterior de melhor desempenho, a estratégia proposta reduziu em 53,8% a quantidade de escritas na NVM, e apresentou *overhead* de tempo de execução de 0,75% contra 9,49% do sistema anterior.

Na Seção 2 deste trabalho fundamenta-se o conceito de épocas, nas Seções 3 e 4 o método e a implementação do sistema DONUTS, com os resultados da avaliação na Seção 5. Na Seção 6 tem-se os trabalhos relacionados, e na Seção 7 a conclusão.

2. Fundamentação

2.1. Checkpointing baseado em Épocas

O princípio das técnicas de consistência a falhas dos sistemas NVM baseia-se em salvar periodicamente os dados consistentes da memória (mais o estado da CPU) para ter um

[2] *Group-commit* é uma técnica que agrupa as transações em lotes e as persiste de uma vez a fim de amortizar o custo da persistência.

ponto de restauração (*checkpoint*) caso uma falha ocorra. O *checkpoint* guarda um retrato instantâneo e consistente da memória (*snapshot*), e o intervalo entre um *checkpoint* a outro é comumente chamado de época [Ren et al. 2015].

Uma época é composta por três fases: execução, *commit* e persistência. Na fase de execução, as aplicações podem ler e modificar os dados presentes nas caches voláteis mediante instruções de *load*/*store*. Após o encerramento da época, as atualizações são confirmadas (*commit*) e os dados modificados são persistidos na NVM (persistência). É importante observar que o *commit* garante apenas que a época em execução foi encerrada e que uma nova pode ser iniciada. Todavia, os dados de *checkpoint* só estarão duráveis após a persistência. A duração de uma época depende da estratégia utilizada pelo mecanismo de *checkpointing*, o mais comum é delimitá-las por intervalos periódicos de tempo ou de instruções. A estratégia do DONUTS, no entanto, estabelece épocas dinâmicas que se encerram conforme limiares de lotação das caches.

2.2. Sobreposição de Épocas

Conforme ThyNVM [Ren et al. 2015], a alternância entre as fases de execução e *checkpointing* (`commit` mais `persistência`) sem sobreposição (Figura 1a) pode incorrer em degradação de desempenho significativa, chegando a consumir até 35,4% de todo o tempo de execução com cargas de trabalho intensivas de memória. Outro problema é que as remoções das caches não são escaláveis em computadores com caches de grande capacidade. Quanto maior o seu tamanho, maior é o tempo que o sistema tende a ficar em *stall* para que os dados sujos (modificados) sejam enviados de volta para a memória (*write-back*). Este processo geralmente corresponde a maior parte do *overhead* de um *checkpoint*. Em média, o tempo para persistir uma cache de 2 MB na DRAM é de 1ms [Nguyen and Wentzlaff 2018]. Embora esta métrica (10% de uma época com duração comum de 10ms) seja aceitável para alguns casos de uso, as caches SRAM estão crescendo rapidamente e, inevitavelmente, o *write-back* dos dados em caches maiores podem levar tempos de latência consideráveis, gerando gargalos nos mecanismos de *checkpointing*. CPUs de classe de servidor Intel e AMD, por exemplo, podem ter até 64 MB de L3 e 128 MB de L4, enquanto o *mainframe* IBM z15 tem 256 MB de L3 e 960 MB de cache L4. Nesta última configuração, o *write-back* de toda a cache poderia levar até 480ms.

As técnicas de consistência de falhas propostas na literatura fazem sobreposição das fases a fim de retirar a persistência do caminho crítico e evitar que o sistema bloqueie, mitigando a degradação de desempenho [Ni et al. 2019]. Enquanto uma época atual está em execução, épocas anteriores podem ser confirmadas (*commit*) e estarem na fila para serem persistidas na NVM. Cabe observar que para garantir consistência, ao menos uma época deve estar completamente persistida. A persistência das épocas podem ocorrer de forma síncrona ou assíncrona [Wei et al. 2020]. Neste segundo caso, os *checkpoints* são persistidos em segundo plano fora do caminho crítico.

Um exemplo de sobreposição de épocas é ilustrado na Figura 1, que demonstra o esquema de ThyNVM [Ren et al. 2015]. No primeiro caso (Figura 1a), as épocas ocorrem de forma alternada, resultando em degradação significativa de desempenho. No segundo (Figura 1b), as fases de execução e de *checkpointing* são sobrepostas. A cópia de trabalho ativa $\mathtt{W}_{\mathtt{active}}$ corresponde à região onde os dados em execução são atualizados. Enquanto executa a época atual $\mathtt{W}_{\mathtt{active}}$, a anterior $\mathtt{C}_{\mathtt{last}}$ é persistida na NVM. Contudo, uma falha durante a persistência de $\mathtt{C}_{\mathtt{last}}$ poderia deixar os dados de *checkpoint* inconsistentes,

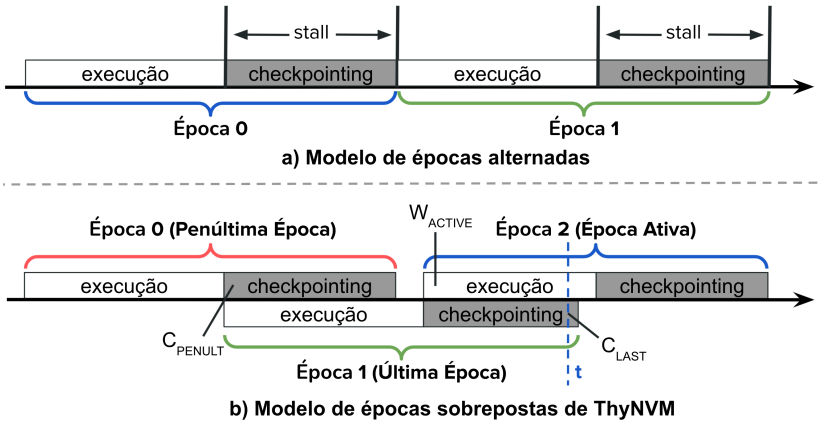

Figura 1. Exemplo de esquema de *checkpointing* com fases alternadas (a) e outro com sobreposição de fases (b).

por isso, o penúltimo *checkpoint* C_{penult} é mantido até que C_{last} seja finalizado. Portanto, este esquema faz a sobreposição de três épocas: enquanto a atual (época 2) está em execução, a anterior (época 1) faz *checkpointing* e a penúltima (época 0) corresponde à época segura, isto é, a que pode ser restaurada caso uma falha ocorra.

2.3. Técnicas de Consistência de Falhas

Os trabalhos propostos na literatura para garantir consistência de falhas em sistemas NVM têm utilizado diferentes abordagens. A mais comum é *write-ahead logging* (WAL), que se subdivide em técnicas de *undo-logging* ou *redo-logging*. Em resumo, o *redo-logging* permite a recuperação para um estado consistente mediante um processo que refaz as operações pendentes, enquanto o *undo-logging* recupera-se voltando as alterações incompletas para os valores de um estado anterior consistente.

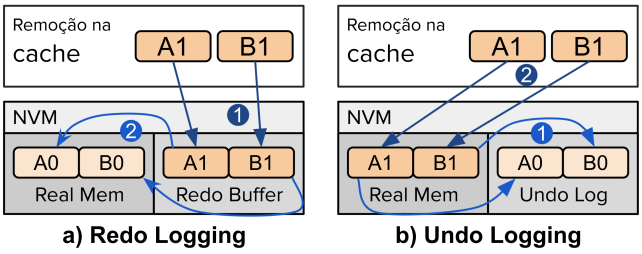

Figura 2. *Redo-logging* e *undo-logging*,

No *redo-logging* (Figura 2a), as remoções da cache são mantidas temporariamente em um *redo-buffer* localizado na NVM para preservar a consistência da memória principal até que as atualizações sejam escritas na região padrão de endereçamento de memória. Semelhante a um *write-buffer* de CPU, esse *redo-buffer* é espionado em todos os acessos à memória para evitar o retorno de dados desatualizados. Nesta técnica, a escrita é feita de maneira indireta, pois os dados atualizados são armazenados primeiramente na região de *redo-buffer*, e posteriormente atualizados em seus locais padrões. Isso exige mapeamento adicional para que seja possível localizar as últimas versões consistentes dos dados. No *undo-logging* (Figura 2b), em uma remoção da cache, os dados originais (isto é, os dados antes das modificações) são primeiro lidos do seu endereço de memória padrão.

Então, esses dados são persistidos em um *undo-buffer* na NVM e, finalmente, a remoção é gravada localmente na memória. Essa sequência é chamada de sequência de acesso de leitura-log-modificação [Nguyen and Wentzlaff 2018]. Se o sistema travar ou perder energia, ele reverte essas gravações aplicando as entradas no *undo-buffer* para restaurar a consistência do último *checkpoint*.

3. DONUTS

Assim como em trabalhos anteriores [Ren et al. 2015, Nguyen and Wentzlaff 2018, Wei et al. 2019], o método DONUTS baseia-se em sobreposição de épocas para gerar *checkpoints* do sistema. O objetivo da sobreposição é permitir que uma época em fase de persistência não bloqueie outra em execução. Para isso, as épocas são independentes e não há necessidade de sincronização entre as etapas. Os dados modificados em cada época podem ser rastreados por um campo `EpochID` (`EID`) adicionado em cada bloco da cache. Este campo armazena a época em que o dado foi modificado pela última vez. Um evento de *commit* ocorre sob três condições:

A) Ao atingir um limiar do conjunto associativo;
B) Ao atingir um limiar de blocos sujos na cache;
C) Conforme um limite especificado de tempo (*timeout*).

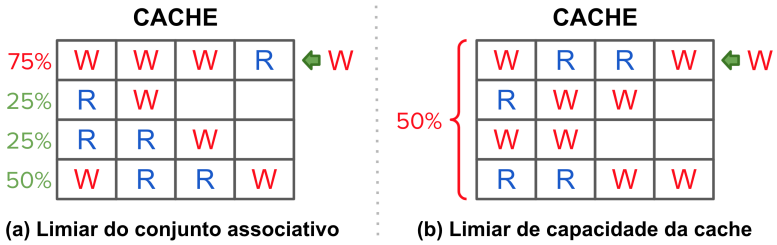

Figura 3. Limiares do conjunto associativo (a) e da capacidade da cache (b).

Na Figura 3 são ilustrados exemplos das condições A e B. A letra R denota os blocos limpos e W os blocos sujos (isto é, marcados com *dirty bit*). Nesta configuração, o limiar do conjunto associativo foi definido em 75% e o limiar da cache em 50%. Como resultado, um *commit* ocorrerá sempre que a) algum dos conjuntos associativos extrapolar a margem de 75% de blocos sujos, ou b) caso o sistema suje mais da metade dos blocos da cache. As porcentagens ao lado dos conjuntos associativos na Figura 3a representam a quantidade de blocos sujos em cada conjunto. O valor de 50% na Figura 3b significa que metade dos blocos da cache estão sujos. Portanto, cada uma das figuras está exemplificando uma situação limite, em que a alocação de um novo bloco na cache (representado pela seta verde) causará o encerramento da época em execução. Ao ocorrer este evento (*commit*), todos os blocos sujos da cache são marcados para serem enviados em segundo plano para a NVM, e uma nova época é iniciada (`SystemID++`).

3.1. Política de Substituição de Cache

O DONUTS utiliza uma nova política de substituição de cache nomeada LRU-R (*Least Recently Used Read*) como principal estratégia para definir dinamicamente a duração das épocas. Esta política é baseada em uma tradicional LRU (*Least Recently Used*), mas com uma leve modificação para que as remoções vitimizem apenas os blocos limpos. Isso

minimiza obter curtos intervalos de *checkpointing*, uma vez que os blocos sujos são mantidos por maior tempo, alongando a duração da época em execução. Apesar da escolha da LRU como base, outras políticas podem ser futuramente implementadas, derivando-se, por exemplo, de políticas tradicionais como a LFU (*Least Frequently Used*) ou a aleatória, desde que utilizada a mesma modificação de vitimização de blocos limpos. Na Figura 4 é mostrado um exemplo da LRU-R com limiar definido em 100%. As setas verdes sinalizam o bloco alocado em cada etapa. Se existir bloco disponível, este é selecionado pela política de substituição; caso contrário, escolhe-se o bloco limpo menos recentemente usado. As porcentagens ao lado dos blocos estão relacionadas à quantidade de blocos sujos em cada conjunto. No passo 10, é possível observar que após todos os blocos estarem sujos, o pedido para armazenamento de um novo bloco (passo 11a), seja para operação de leitura ou de escrita, resultará em um *commit* (cm) da época em andamento. O *commit* marca todos os blocos sujos da cache para serem persistidos em segundo plano e altera os seus estados na cache para limpo, ficando disponíveis a serem substituídos (passo 11b).

Figura 4. Política de Substituição de Cache de DONUTS.

3.2. Execução das épocas sobrepostas no sistema DONUTS

Na Figura 5 exemplifica-se um fluxo de execução de DONUTS. Novamente, R denota os blocos limpos e W os blocos sujos. R^{W} representa os blocos confirmados que voltam para o estado limpo após o *commit* da respectiva linha da cache, pois foram adicionados ao *buffer* de persistência. No passo 1, o sistema está em uma época denotada em azul e um dos conjuntos associativos da cache excedeu o limiar do conjunto associativo, atingindo a condição A. Isso causa o *commit* da época azul, e uma nova época denotada em verde é iniciada. Ao efetuar o *commit*, os blocos sujos da época confirmada devem ser enviados à NVM, mas ao contrário de uma operação convencional de *flush*, os blocos não são invalidados após o envio, apenas redefinidos para o estado limpo (W → R). Essa estratégia de *write-back* evita *misses* em prováveis novos acessos, mas permite que o bloco seja posteriormente substituído pela política de substituição de cache. O passo 2 mostra esse processo. Na representação, os blocos sujos foram enviados ao *buffer* (passo a) e o menos recentemente usado foi substituído por um novo bloco já pertencente à época verde (passo b). Neste momento, é importante observar que enquanto a época verde está iniciando a execução, a azul está fazendo *commit* dos dados, gerando um efeito de sobreposição de épocas. No decorrer da execução, ambas as épocas caminham simultaneamente em suas respectivas fases. No passo 3, grande parte da cache está composta por dados da época

verde, e apenas uma parte dos dados da época azul não foi substituída. Durante o passo 3, os dados da época azul estão sendo persistidos em segundo plano para a NVM com uso de um *write-buffer* convencional. No passo 4, a época azul já foi persistida e apenas blocos da época verde estão presentes na cache, que novamente atingiu o limiar de um dos conjuntos associativos e, como resultado, no passo 5, uma nova época inicia-se, agora na cor laranja. O passo 6 é análogo ao passo 3.

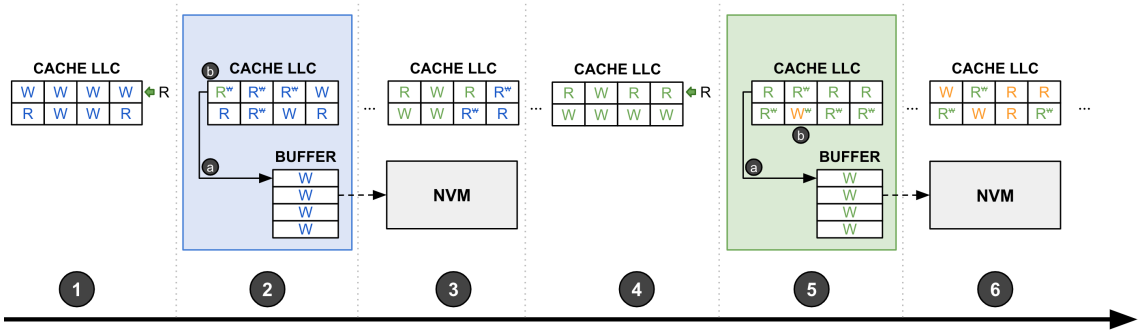

Figura 5. Fluxo de execução das épocas de DONUTS.

4. Implementação

O sistema DONUTS foi implementado no simulador Sniper 7.4. As caches foram levemente modificadas com a adição do campo `EpochID` em cada bloco, a LLC configurada para usar a política de substituição LRU-R descrita na Seção 3.1, e os eventos de *commit* foram adicionados ao controlador da cache de último nível. Com exceção a estes, nenhum outro componente precisou sofrer modificação. Os resultados foram comparados contra o PiCL [Nguyen and Wentzlaff 2018], criteriosamente implementado no mesmo simulador conforme as suas especificações. Assim, a política de coerência de cache MESI foi modificada para adicionar os eventos de *commit* e de *checkpoint* do respectivo mecanismo, e o sistema de época do DONUTS foi expandido para incrementar as épocas do sistema em intervalos fixos de instruções. Ambas as implementações encontram-se disponíveis em: `https://github.com/kleberkruger/donuts`.

5. Avaliação

A fim de avaliar o desempenho do DONUTS, foram realizados três conjuntos de experimentos com o SPEC CPU2006 (*benchmark* também utilizado nas avaliações dos trabalhos anteriores). Os experimentos executaram cada aplicação do SPEC com um bilhão de instruções, e os parâmetros de configurações do sistema foi semelhante aos definidos no trabalho do PiCL [Nguyen and Wentzlaff 2018], mostrados na Tabela 1.

Processador	2 GHz, in-order
Cache L1 D/I	Private 32KB, 4-way, 64B block; 1 cycles hit
Cache L2	Private 256KB, 8-way, 64B block; 4 cycles hit
Cache L3	Shared 2MB, 8-way, 64B block; 30 cycles hit

Tabela 1. Configurações do sistema.

O primeiro conjunto de testes utilizou oito aplicações de uso intensivo de memória. O objetivo foi identificar o melhor ajuste de limiar do conjunto associativo.

Assim, o respectivo parâmetro foi testado com os valores de 50%, 75% e 100%. Os limiares de capacidade da cache e de *timeout* não foram utilizados, mantendo os eventos de *commit* restritos ao preenchimento dos conjuntos associativos. Os resultados descritos na Tabela 2 identificaram que normalizados ao tempo de execução de um sistema *baseline*[3], os limiares de 50% e 75% resultaram em um *slowdown* menor que 2%, e o limiar de 100%, um *speedup* de 2,98%. Este ganho é justificado pela alteração da política de substituição de cache que nas aplicações `leslie3d` e `lbm` reduziu a taxa de *misses* na LLC entre 1,5% a 2,6%.

Projeto	Política LLC	Checkpoint	Limiar	Tempo de Exec
Baseline	LRU			
DONUTS	LRU-R	✓	50%	+1,40%
DONUTS	LRU-R	✓	75%	+1,97%
DONUTS	LRU-R	✓	100%	-2,98%

Tabela 2. Avaliação de diferentes valores de limiares para o conjunto associativo.

O segundo conjunto de testes avaliou o desempenho do DONUTS comparando-o ao PiCL, projeto com melhor desempenho de tempo de execução entre os trabalhos encontrados na literatura[4]. Neste, os limiares do conjunto associativo e de capacidade da cache foram definidos em 100% e 75%, respectivamente, e o valor de *timeout* para os *checkpoints* foi fixado em 50ms. No PiCL, o tamanho do intervalo entre as épocas foi definido em 30 milhões de instruções, e ao parâmetro `acs-gap` (janela para persistência de épocas pendentes) foi atribuído o valor três, conforme as configurações do artigo original. Os resultados exibidos na Figura 6a estão ordenados pelas aplicações com maior uso de largura de banda de memória. Portanto, `libquantum`, `bwaves` e `lbm` são as que apresentam maiores taxas. No sistema *baseline*, por exemplo, consomem respectivamente uma média de 74%, 50% e 42% de uso de largura de banda (Figura 6b). Embora ambos os mecanismos não gerem *overhead* significativo de desempenho nas aplicações com baixa intensidade de acesso à memória, no grupo com alta taxa de acesso o desempenho do PiCL foi afetado em até 106%, com aumento de largura de banda chegando a 17% na aplicação `libquantum`. Em compensação, a maior sobrecarga no tempo de execução do DONUTS foi de 12,98%, sendo importante observar que a política de substituição LRU-R afetou positivamente a aplicação `libquantum`. No DONUTS, em vez de *slowdown*, a aplicação apresentou *speedup* de 14,79%. Em termos gerais, o *overhead* médio de tempo de execução foi de 9,47% no PiCL contra 0,75% no DONUTS.

Na Figura 6c são exibidas as quantidades de gravações na NVM de ambos os projetos normalizados a um sistema ideal sem *checkpoints*. O DONUTS manteve a taxa de escrita abaixo de 1,3x em 79% das aplicações avaliadas. A aplicação de maior *overhead* foi `gamess`, cujo aumento foi de 3,83x. Esta aplicação consiste em uma ampla gama de cálculos, e o fator de aumento nas gravações NVM deve-se ao comportamento denso de acesso aos *arrays* na memória. Entretanto, no PiCL o *overhead* gerado foi de 13,4x.

[3]Baseline corresponde à arquitetura padrão do simulador, sem *checkpoints* e com política de substituição LRU em todos os níveis de cache.

[4]Outros trabalhos avaliados foram: ThyNVM [Ren et al. 2015], NICO [Wei et al. 2019] e Dual-Page Checkpointing [Wu et al. 2019]. Foram descartados projetos não completamente transparentes ao software.

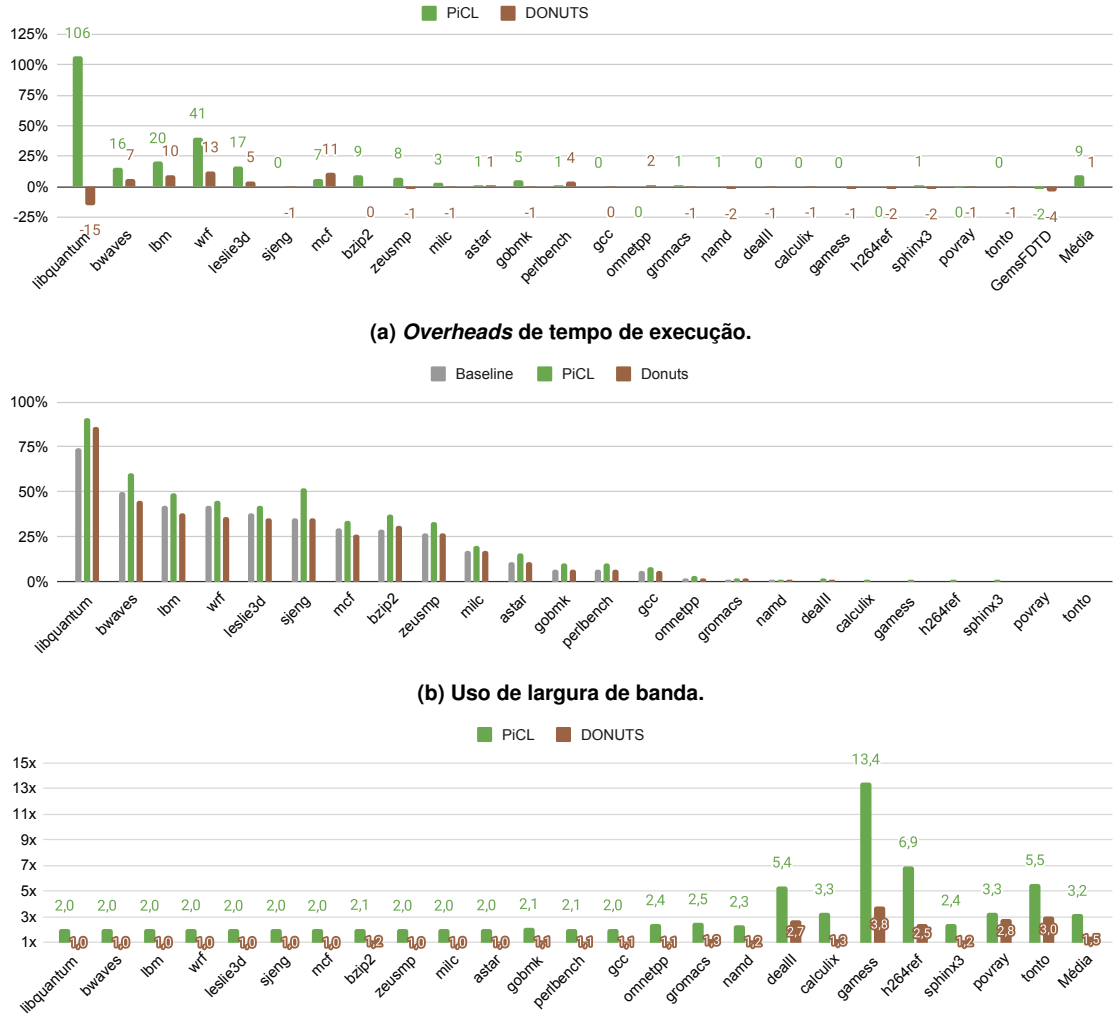

Figura 6. Desempenho do PiCL e do DONUTS comparado a um sistema *Baseline*.

Considerando-se todas as aplicações, o DONUTS apresentou uma média de 1,46x contra 3,16x do sistema PiCL. Também é importante observar o impacto da Lei de Amdahl no desempenho. Apesar da aplicação `gamess` apresentar a maior taxa de *overhead* de escritas na NVM, esta realizou cerca de 185x menos acessos à memória do que a aplicação de maior intensidade (`libquantum`), ocasionando um impacto menos expressivo ao uso de largura de largura de banda (de 0% no sistema *baseline* foi para 1% com o PiCL).

Na Figura 7 estão exibidos os histogramas dos intervalos de tempo entre *checkpoints* de algumas aplicações. Neste terceiro conjunto de testes, o limiar do conjunto associativo foi definido em 50% com o intuito de avaliar a frequência dos *checkpoints*. Para melhor visualização, aplicou-se um filtro para retirar 0,5% de *outliers*. As aplicações `bwaves` e `leslie3d` fazem uso mais intensivo de acesso à memória e, portanto, obtiveram *checkpoints* em intervalos menores de tempo. Este fato evidencia o maior *overhead* mostrado na Figura 6a. Em contraponto, a aplicação `omnetpp`, cujo comportamento é menos intenso à memória, realizou *checkpointings* em intervalos maiores de tempo (entre 8ms a 10ms), e `gcc` apresentou comportamento mais esparso. Os dois últimos cenários,

consequentemente, geraram menores sobrecargas ao tempo de execução.

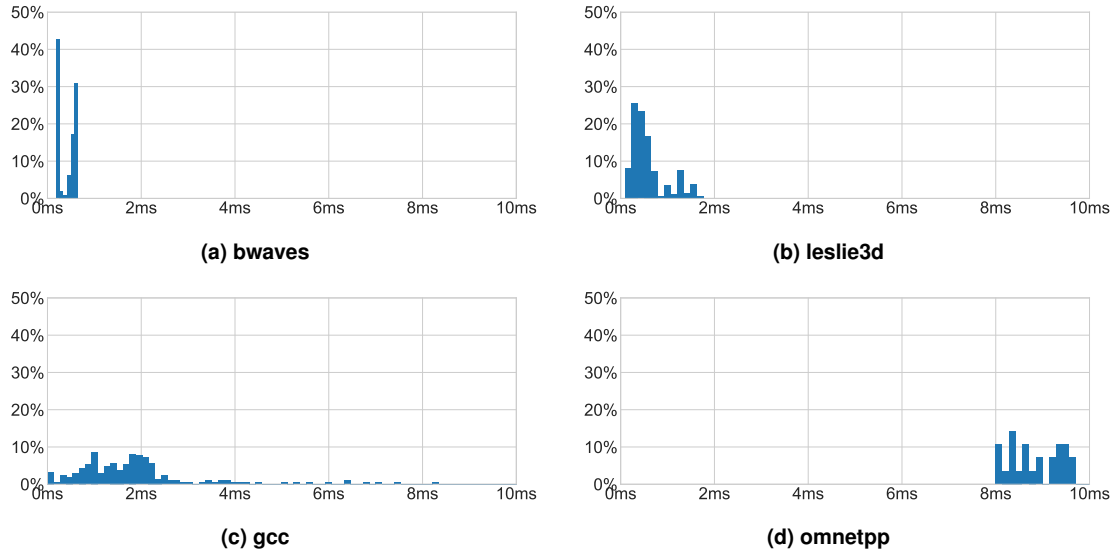

Figura 7. Histogramas de aplicações com diferentes intervalos de *checkpointing*.

6. Trabalhos Relacionados

Os trabalhos propostos na literatura dividem-se em estratégias baseadas em software e em hardware. As soluções via software [Volos et al. 2011, Coburn et al. 2011, Chakrabarti et al. 2014, Giles et al. 2015, Kolli et al. 2016, Hsu et al. 2017, Liu et al. 2017, Gogte et al. 2018] fornecem consistência de falhas por meio de instruções para sincronizar as ordens das operações de leitura/escrita e de instruções para forçar *write-backs* nas linhas de cache. Porém, isso impõem restrições significativas de uso e de desempenho. As estratégias de hardware, por sua vez, mitigam o *overhead* de tempo de execução, mas apesar do uso de hardware, a maior parte [Doshi et al. 2016, Kolli et al. 2016, Joshi et al. 2017, Shin et al. 2017, Jeong et al. 2018, Ogleari et al. 2018, Ni et al. 2019, Wei et al. 2020, Cai et al. 2020] depende de modelos de programação baseados em software de memória transacional (STM - *Software Transacional Memory*) para indicar as transações, ficando o hardware restrito a criar e manipular automaticamente os dados e metadados de logs e *checkpoints*. Alguns projetos [Ren et al. 2015, Nguyen and Wentzlaff 2018, Wei et al. 2019, Wu et al. 2019] resolveram este problema, fornecendo soluções completamente transparentes ao software, mas ainda assim, apresentaram outros. ThyNVM [Ren et al. 2015], por exemplo, adota um modelo sincronizado de épocas sobrepostas, mas a necessidade de sincronismo afeta o desempenho do sistema, principalmente em cenários *multi-core*. PiCL [Nguyen and Wentzlaff 2018] estabelece um conceito de *multi-undo-logging*, que desacopla as épocas do sistema e permite persistência assíncrona fora do caminho crítico de execução, mas seu modelo gera escritas excessivas na NVM.

7. Conclusão

Embora os trabalhos recentes tenham proposto alternativas para diminuir o *overhead* de tempo de execução, a maior parte deles apresenta soluções não transparentes ao software

que restringem o uso de NVM às aplicações baseadas em memória transacional. Além disso, devido às operações de *logging*, as soluções atuais tendem a multiplicar a quantidade de escritas na NVM, ocasionado gargalos em aplicações com alta demanda de acesso à memória. Este trabalho apresenta uma solução transparente ao software com *checkpointings* assíncronos integrados à política de substituição de cache, que ao contrário dos demais encontrados na literatura, gera épocas com intervalos dinâmicos. Os resultados mostram que o modelo sugerido apresenta menor *overhead* de tempo de execução e reduz o uso de largura de banda no barramento de comunicação com a memória principal. Cabe observar que futuras otimizações, tais como as descritas em CCHL [Wei et al. 2020] podem reduzir o tráfego de dados e, consequentemente, diminuir a latência das operações de *checkpointing*.

Referências

Cai, M., Coats, C. C., and Huang, J. (2020). Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 584–596.

Chakrabarti, D. R., Boehm, H.-J., and Bhandari, K. (2014). Atlas: Leveraging locks for non-volatile memory consistency. *SIGPLAN Not.*, 49(10):433–452.

Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. (2011). Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGARCH Comput. Archit. News*, 39(1):105–118.

Doshi, K., Giles, E., and Varman, P. (2016). Atomic persistence for scm with a non-intrusive backend controller. volume 2016-April, pages 77–89.

Fackenthal, R., Kitagawa, M., Otsuka, W., Prall, K., Mills, D., Tsutsui, K., Javanifard, J., Tedrow, K., Tsushima, T., Shibahara, Y., and Hush, G. (2014). 19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 338–339.

Giles, E., Doshi, K., and Varman, P. (2015). Softwrap: A lightweight framework for transactional support of storage class memory. volume 2015-August.

Gogte, V., Diestelhorst, S., Wang, W., Narayanasamy, S., Chen, P. M., and Wenisch, T. F. (2018). Persistency for synchronization-free regions. *SIGPLAN Not.*, 53(4):46–61.

Hsu, T. C.-H., Brügner, H., Roy, I., Keeton, K., and Eugster, P. (2017). Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 468–482, New York, NY, USA. Association for Computing Machinery.

Intel (2015). Intel and micron produce breakthrough memory technology.

Jeong, J., Park, C., Huh, J., and Maeng, S. (2018). Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. volume 2018-October, pages 520–532.

Joshi, A., Nagarajan, V., Viglas, S., and Cintra, M. (2017). Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372.

Kolli, A., Pelley, S., Saidi, A., Chen, P., and Wenisch, T. (2016). High-performance transactions for persistent memories. volume 02-06-April-2016, pages 399–411.

Liu, M., Zhang, M., Chen, K., Qian, X., Wu, Y., Zheng, W., and Ren, J. (2017). Dudetm: Building durable transactions with decoupling for persistent memory. volume Part F127193, pages 329–343.

Nguyen, T. M. and Wentzlaff, D. (2018). Picl: A software-transparent, persistent cache log for nonvolatile main memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 507–519.

Ni, Y., Zhao, J., Litz, H., Bittman, D., and Miller, E. (2019). Ssp: Eliminating redundant-writes in failure-atomic nvrams via shadow sub-paging. pages 836–848.

Noguchi, H., Ikegami, K., Kushida, K., Abe, K., Itai, S., Takaya, S., Shimomura, N., Ito, J., Kawasumi, A., Hara, H., and Fujita, S. (2015). 7.5 a 3.3ns-access-time 71.2μw/mhz 1mb embedded stt-mram using physically eliminated read-disturb scheme and normally-off memory architecture. In *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*, pages 1–3.

Ogleari, M., Miller, E., and Zhao, J. (2018). Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. volume 2018-February, pages 336–349.

Ren, J., Zhao, J., Khan, S., Choi, J., Wu, Y., and Mutiu, O. (2015). Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 672–685.

Shin, S., Tirukkovalluri, S. K., Tuck, J., and Solihin, Y. (2017). Proteus: A flexible and fast software supported hardware logging approach for nvm. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 178–190.

Volos, H., Tack, A., and Swift, M. (2011). Mnemosyne: Lightweight persistent memory. pages 91–103.

Wei, X., Feng, D., Tong, W., Liu, J., Wang, C., and Ye, L. (2020). Cchl: Compression-consolidation hardware logging for efficient failure-atomic persistent memory updates. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA. Association for Computing Machinery.

Wei, X., Feng, D., Tong, W., LIU, J., and Ye, L. (2019). Nico: Reducing software-transparent crash consistency cost for persistent memory. *IEEE Transactions on Computers*, 68(9):1313–1324.

Wu, Q., Sun, F., Xu, W., and Zhang, T. (2013). Using multilevel phase change memory to build data storage: A time-aware system design perspective. *IEEE Transactions on Computers*, 62(10):2083–2095.

Wu, S., Zhou, F., Gao, X., Jin, H., and Ren, J. (2019). Dual-page checkpointing: An architectural approach to efficient data persistence for in-memory applications. *ACM Trans. Archit. Code Optim.*, 15(4).