UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de Rio Claro

Rafael Pizzirani Murari

Otimização de Desempenho em Ambientes com Memória Persistente via Transações em Fase

Rio Claro - SP

2019

Rafael Pizzirani Murari

Otimização de Desempenho em Ambientes com Memória Persistente via Transações em Fase

Orientador: Prof. Dr. Alexandro José Baldassin

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Geociências e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de Rio Claro.

Financiadora: CAPES

Rio Claro - SP

2019

M972o

Murari, Rafael Pizzirani
Otimização de Desempenho em Ambientes com Memória Persistente via Transações em Fase / Rafael Pizzirani Murari. -- Rio Claro, 2019
78 f. : il., tabs.

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Instituto de Geociências e Ciências Exatas, Rio Claro
Orientador: Alexandro José Baldassin

1. Ciência da computação. 2. Programação (Computadores). 3. Memória Persistente. 4. Transações Duráveis. 5. Memória Transacional em Fases. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de Geociências e Ciências Exatas, Rio Claro. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

Rafael Pizzirani Murari

Otimização de Desempenho em Ambientes com Memória Persistente via Transações em Fase

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Geociências e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de Rio Claro.

Financiadora: CAPES

Banca Examinadora

- Prof. Dr. Alexandro José Baldassin (Orientador)
  Departamento de Estatística, Matemática Aplicada e Computação
  Universidade Estadual Paulista - UNESP
- Prof. Dr. Emilio de Camargo Francesquini
  Centro de Matemática, Computação e Cognição
  Universidade Federal do ABC - UFABC
- Prof. Dr. Orlando de Andrade Figueiredo
  Departamento de Estatística, Matemática Aplicada e Computação
  Universidade Estadual Paulista - UNESP

Rio Claro - SP

28 de fevereiro de 2019

Agradecimentos

Em primeiro lugar gostaria de agradecer a Deus por ter me guiado e auxiliado em todos os momentos de minha vida. “Tu és o meu Deus, e eu te louvarei; Tu és o meu Deus, e eu te exaltarei” - Salmos 118:28.

Agradeço a meus pais por todo suporte, pelo carinho e pela educação que me deram até este momento. Não há palavras suficientes para expressar a minha gratidão a Deus por todos os momentos que passamos juntos.

Meus sinceros agradecimentos ao meu professor e orientador, Dr. Alexandro José Baldassin pela imensa contribuição em minha vida acadêmica e profissional. Seus ensinamentos e ajuda foram de profunda valia para realização desta dissertação. Agradeço também a João Paulo Labegalini de Carvalho pela grande contribuição no desenvolvimento deste trabalho.

Agradeço a todos os professores e amigos que contribuíram em minha formação acadêmica, em especial os membros do Projeto Radiar e os integrantes da equipe do desafio de programação paralela.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001, à qual agradeço.

Resumo

As emergentes tecnologias de memória persistente (PM) visam eliminar a lacuna existente entre a memória principal e a secundária. No entanto, os sistemas atuais não são capazes de usufruir totalmente dos benefícios proporcionados por estas, essencialmente devido a possíveis falhas de sistema que podem resultar em um estado inconsistente e irrecuperável. Além disto, o uso simplista da PM resulta em uma degradação de desempenho, advinda do alto custo associado às operações de escrita. Neste contexto, o uso de transações duráveis é uma das abordagens mais investigadas para facilitar a adoção da PM. Em particular, implementações de memória transacional em hardware (HTM) possibilitam a execução de transações com uma sobrecarga mínima, porém apresentam limitações de recursos. Embora as transações em software (STM) sejam flexíveis e não possuam tais limitações, estas não apresentam um bom desempenho na execução de transações curtas. Esta dissertação apresenta a solução NV-PhTM, um sistema transacional baseado em fases capaz de alterar dinamicamente o modo de execução, software ou hardware, mediante as características apresentadas pelas aplicações. A implementação do NV-PhTM foi embasada pelo sistema PhTM*, um arcabouço que provê um conjunto de heurísticas para guiar a transição e seleção do melhor modo de execução (HW/SW). O PhTM*, no entanto, foi concebido para ambientes de memória volátil. Neste contexto, o NV-PhTM propõe novas heurísticas visando contemplar as estratégias de garantia de durabilidade e as características da PM. Visando manter a corretude do sistema, estratégias foram elaboradas a fim de garantir a persistência durante a transição entre as fases. O NV-PhTM é o primeiro sistema transacional baseado em fases a prover transações duráveis. Os resultados experimentais obtidos, na execução do *benchmark* STAMP, comprovam a eficácia das novas heurísticas em guiar a transição das fases. Quando comparado ao NV-HTM (solução exclusivamente em hardware) e ao PSTM (solução exclusivamente em software), o NV-PhTM obteve os melhores resultados devido a sua natureza de seguir o sistema com melhor desempenho.

Palavras-chave: memória persistente, transações duráveis, memória transacional.

Abstract

The emerging persistent memory technologies (PM) are aimed to eliminate the gap between main memory and storage. Nevertheless, today's programs will not readily benefit from them essentially because crash failures might render the program in an unrecoverable and inconsistent state. In addition, naive utilization of PM leads to performance degradation, mainly due to expensive writes. In this context, the usage of durable transactions is one of the main investigated approaches to ease the adoption of non-volatile memory. In particular, hardware transactional memories (HTM) provide low-overhead but are resource-constrained. Although software transactions (STM) are flexible and unbounded, they may significantly hurt the performance of short-lived transactions. This dissertation presents NV-PhTM, a transactional system for PM that delivers the best out of both HW and SW transactions by dynamically changing the execution according to the characteristics of the application. NV-PhTM is built upon the PhTM* system, a framework that provides a set of heuristics to guide phase transition and selects the best execution mode (HW/SW). PhTM*, however, did not take into account PM systems. Designing NV-PhTM required new heuristics that consider the nature of durability strategies and the characteristics of PM. In order to keep the accuracy of the system, new strategies were elaborated to guarantee persistency between phase transitions. NV-PhTM is the first phase-based system to provide durable transactions. Experimental results with the STAMP benchmark show that the proposed heuristics are efficient in guiding phase transitions with low overhead. When compared to NV-HTM (hardware-only durable transactions) and PSTM (software-only durable transactions), NV-PhTM provided the best overall results due to its nature of following the best performing system.

Keywords: persistent memory, durable transactions, transactional memory.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API	*Application Programming Interface*
BFC	*Backward Filtering Checkpointing*
CAS	*Compare-And-Swap*
CLWB	*Cache Line Write Back*
CP	*Checkpoint Process*
DAX	*Direct Access*
DRAM	*Dynamic Random Access Memory*
HDD	*Hard Disk Drive*
HTM	*Hardware Transactional Memory*
HyTM	*Hybrid Transactional Memory*
IMDB	*In-Memory Database*
ISA	*Instruction Set Architecture*
LLC	*Last Level Cache*
MMU	*Memory Management Unit*
NV-PhTM	*Non-Volatile Phased Transactional Memory*
NVM	*Non-Volatile Memory*
PCM	*Phase-Change Memory*
PhTM	*Phased Transactional Memory*
PM	*Persistent Memory*
PS	*Persistent Snapshot*
RRAM	*Resistive Random Access Memory*
RTM	*Restricted Transactional Memory*
SGL	*Single Global Lock*
SPMD	*Single Program Multiple Data*
SRAM	*Static Random Access Memory*
SSD	*Solid-State Drive*
STAMP	*Stanford Transactional Applications for Multi-Proccessing*
STM	*Software Transactional Memory*
STT-RAM	*Spin-Transfer Torque Random Access Memory*
TLB	*Translation Lookaside Buffer*
TM	*Transactional Memory*
TSX	*Transactional Synchronization Extension*
WP	*Working Process*
WS	*Working Snapshot*

Sumário

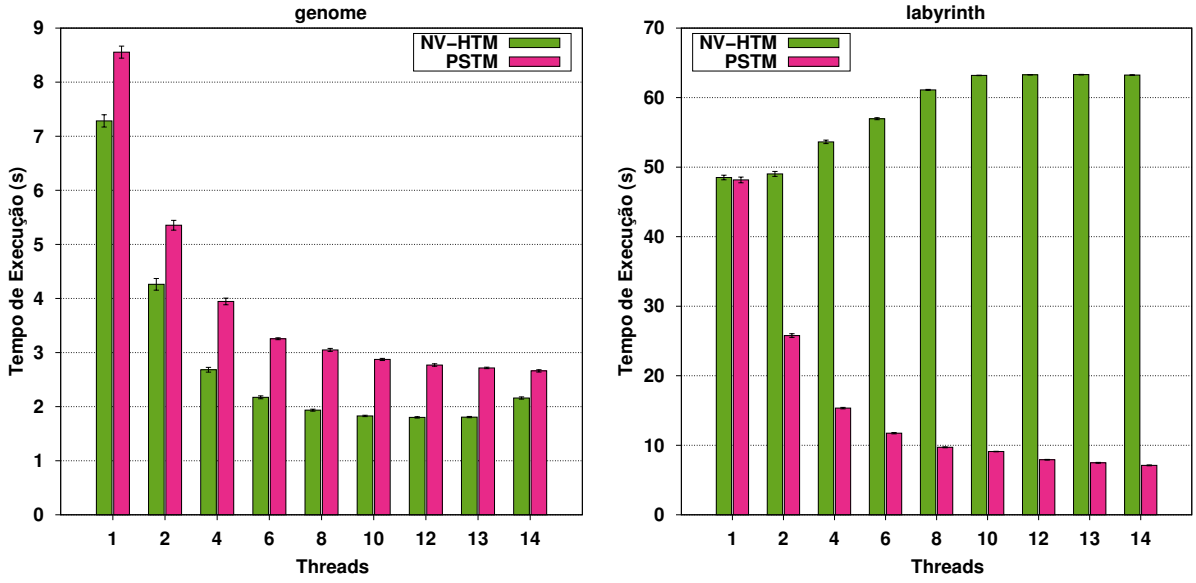
1 Introdução

A grande capacidade computacional, disponibilizada pelos processadores contemporâneos, em conjunto com a crescente demanda de dados pelas aplicações, tem exposto enfaticamente o gargalo de desempenho do subsistema de memória, resultante da alta latência de acesso aos componentes persistentes deste. Neste contexto, as emergentes tecnologias de memória não-voláteis (NVM) visam equacionar os *trade-offs* de persistência e desempenho mediante as características disruptivas apresentadas, tais como alta densidade, endereçamento por byte, acesso via instruções de memória (*load* e *store*) e latência de acesso diminuta [1].

A adoção desta tecnologia prenuncia um elevado desempenho das aplicações, no entanto certas precauções devem ser tomadas para garantia de consistência dos dados, visto que eventuais quedas de energia ou falhas de sistema podem corromper permanentemente regiões inteiras de memória. Diversas abordagens para garantia de consistência podem ser encontradas na literatura, entre estas destaca-se o uso de transações duráveis, conceito inicialmente proposto no contexto de banco de dados [2]. Esta abordagem visa delimitar as operações persistentes em blocos atômicos de execução, ou seja, as instruções presentes nestes são executadas de maneira indivisível e atômica, garantindo a transição de um estado previamente consistente para outro estado consistente ao término de sua execução.

Grande parte das soluções transacionais propostas são em software (STM) [3, 4, 5, 6, 7], possibilitando seu uso de maneira irrestrita, isto é, sem a requisição de recursos em hardware para sua execução. Entretanto, as instrumentações inseridas pelas implementações em software ocasionam custo extra, sendo este crítico em transações com *workload* reduzido. Neste âmbito, o suporte transacional em hardware (HTM), disponível em processadores da Intel e IBM, pode prover um relevante ganho de desempenho em ambientes com memória não-volátil [8]. Porém, a conjuntura NVM e HTM não é trivial, principalmente pelas especificidades apresentadas pelas implementações de HTM contemporâneas.

Diversos trabalhos propuseram modificações arquiteturais para facilitar o uso conjunto destas tecnologias [9, 10, 11, 12], todavia não é possível realizar uma avaliação real destes sistemas, visto que estes foram validados através de simulações. A solução NV-HTM, desenvolvida por Castro e colegas [8], é uma das pioneiras a viabilizar o uso de HTM sem extensões, sendo



(a) Genome (b) Labyrinth

Figura 1 – Tempo de execução das aplicações: (a) Genome e (b) Labyrinth.

capaz de usufruir do alto desempenho proporcionado por este. O único limitante desta solução é a baixa disponibilidade de recursos, advinda do pouco espaço em cache para manutenção do trabalho especulativo gerado pelas transações. Logo, a implementação transacional em hardware não é capaz de efetivar transações com amplo *workload*, recorrendo a estratégia de aquisição de um *lock* global, serializando a execução das transações, e concluindo estas em software.

As implementações atuais, em hardware e software, não contemplam o amplo espectro de características apresentadas pelas aplicações, culminando em perda de desempenho. Para ilustrar tal cenário, a Figura 1 apresenta o tempo de execução obtido pelas aplicações Genome e Labyrinth, integrantes do *benchmark* STAMP [13], em dois sistemas distintos: NV-HTM (implementação em hardware) e PSTM [10] (implementação em software). As baixas taxas de cancelamentos e de contenção das transações, apresentadas pela aplicação Genome, possibilitam um melhor desempenho em implementações em hardware, visto que o software introduz um elevado *overhead* para garantia de consistência. No entanto, aplicações com transações longas, isto é, grande quantidade de instruções presentes no bloco atômico, e com amplo *workload*, conforme apresentado pela aplicação Labyrinth, não obtêm êxito em hardware, forçando constantes serializações para garantia de progresso. Neste cenário, a implementação em software proporciona desempenho amplamente superior.

1.1 Objetivos e Contribuições

Este trabalho apresenta a solução NV-PhTM, sistema transacional baseado em fases capaz de alterar dinamicamente o modo de execução, software ou hardware, mediante as características apresentadas pelas aplicações. A implementação do NV-PhTM foi embasada pelo PhTM* [14], sistema em fases, concebido para ambientes de memória volátil, coordenado por um conjunto de heurísticas responsável pela detecção do modo mais eficiente de execução (hardware/software) e pela respectiva transição para tal modo.

As principais contribuições do NV-PhTM são: (i) a adição do conceito de durabilidade em sistemas baseados em fase; (ii) a elaboração de novas heurísticas, contemplando a natureza das estratégias de durabilidade empregadas pelo NV-HTM e as características da NVM; (iii) o desenvolvimento de estratégias para garantia de consistência durante o processo de transição entre os distintos modos do sistema.

A avaliação experimental realizada comprova a eficácia das novas heurísticas em guiar a transição das fases, bem como o baixo *overhead* adicionado neste processo. A comparação realizada com sistemas exclusivamente em software ou hardware, demonstra o êxito do NV-PhTM em direcionar o sistema a acompanhar a melhor abordagem nos mais distintos cenários.

1.2 Organização do Texto

Para melhor compreensão deste trabalho, o texto foi organizado da seguinte forma. O Capítulo 2 apresenta a fundamentação teórica dos diversos conceitos envolvidos no desenvolvimento desta dissertação. O Capítulo 3 apresenta os principais trabalhos propostos na literatura para uso conjunto de HTM e NVM. Em seguida, o Capítulo 4 apresenta o sistema NV-PhTM e seus respectivos detalhes de implementação. Como motivação da proposta, apresenta-se uma análise que contrasta o desempenho das implementações em software e hardware. Por fim, os Capítulos 5 e 6 apresentam a avaliação experimental realizada e as conclusões obtidas, respectivamente.

2 Fundamentação Teórica

O cerne da Computação baseia-se na premissa de solucionar problemas cada vez mais complexos de maneira eficiente, avaliando-se os *trade-offs* associados a cada situação específica e fornecendo soluções capazes de extrair o máximo desempenho tanto em software quanto em hardware. Durante décadas, boa parte do desempenho obtido foi proveniente do aumento da frequência de *clock*, verificado em cada novo modelo de microprocessador produzido, possibilitando a execução de uma maior quantidade de instruções em um mesmo período de tempo. A Figura 2 ilustra o histórico de crescimento da frequência nos microprocessadores.

Além do aumento da frequência, outro fator importante a ser destacado é a redução do tamanho dos transistores utilizados na produção dos microprocessadores. Esta asserção foi previamente definida pela Lei de Moore [16], a qual especifica que a cada 2 anos a quantidade de transistores presentes em uma mesma área de chip dobra, possibilitando o acréscimo de mais componentes e circuitos digitais.

O aumento da frequência de *clock*, associado à disposição de uma maior área de chip para inserção de novos circuitos, possibilitou o desenvolvimento de otimizações, algumas previamente concebidas em software, em hardware, reduzindo grande parte do consumo de ciclos de execução e consequentemente proporcionando uma melhoria no desempenho das aplicações.

Ao se observar atentamente a Figura 2, constata-se um interessante marco na trajetória da evolução dos microprocessadores: em meados de 2003 houve uma estagnação no crescimento da frequência de operação dos processadores. Tal fato é decorrente dos obstáculos atribuídos ao limite da dissipação de energia de processadores refrigerados a ar e a falta de um maior paralelismo a nível de instrução [15]. A solução abordada para tais adversidades foi a redução da frequência de operação do processador, aliada à adição de mais núcleos de processamento em um mesmo chip, originando os multiprocessadores. Desta forma, a potência cresce linearmente, possibilitando um maior controle da dissipação de energia e evitando os altos custos energéticos atrelados às demais tecnologias de refrigeração.

A disposição de vários núcleos de processamento expõem enfaticamente a latência de acesso a memória, visto que múltiplas unidades de processamento realizam requisições, tanto de dados quanto de instruções, à memória, e consequentemente evidenciando a necessidade

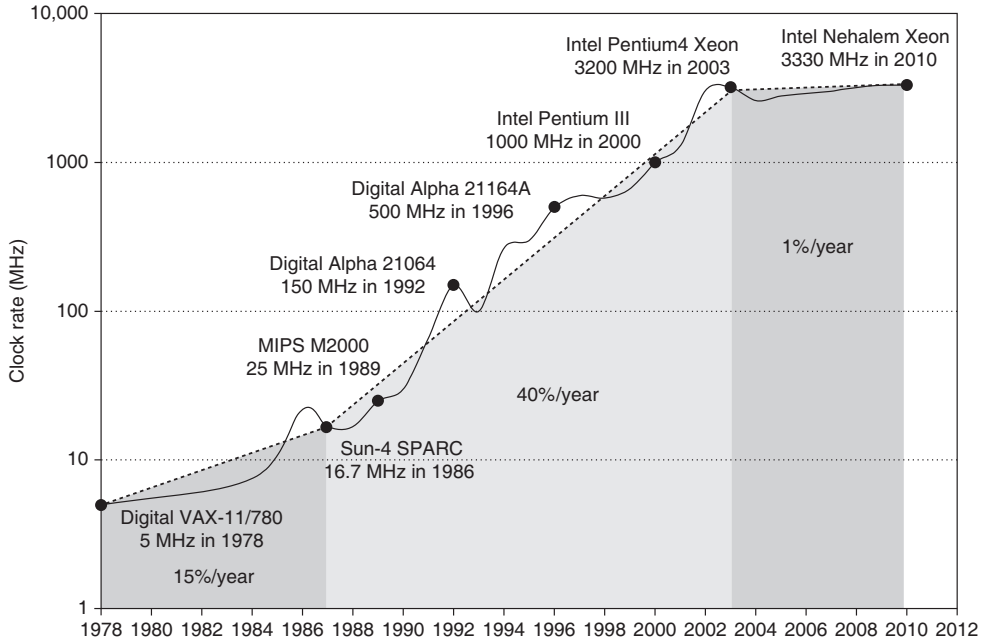


Figura 2 – Aumento da frequência de *clock* nos microprocessadores. Extraído de [15].

de tecnologias de memória e avanços arquiteturais, capazes de reduzir o impacto negativo no desempenho do sistema.

2.1 Tipos de Memória

As decisões arquiteturais, relacionadas aos tipos de memória empregados no sistema, são de vital importância na correlação existente entre memória e desempenho. Conforme descrito anteriormente, a expansão na quantidade de requisições feitas à memória, pelo processador, demanda a utilização de uma memória, suficientemente grande, capaz de proporcionar um tempo de acesso diminuto, idealmente imediato [17]. Tal tecnologia, conhecida no estado da arte como Memória Universal, seria capaz de realizar as funções de memória de trabalho e de armazenamento, eliminando a enorme quantidade de transferência de dados entre os níveis da hierarquia de memória e, consequentemente, simplificando a arquitetura do sistema [18]. Entretanto, tal tecnologia ainda não se encontra disponível.

Neste contexto, concomitantemente com a evolução dos microprocessadores, vários tipos de memória foram desenvolvidos, buscando oferecer novas alternativas aos *trade-offs* impostos até então. Entre estes tipos destacam-se os seguintes: *Hard Disk Drive* (HDD), *Solid-State Drive* (SSD), *Dynamic Random Access Memory* (DRAM) e *Static Random Access Memory* (SRAM); os quais apresentam diferentes especificidades conforme os seguintes aspectos: volatilidade,

Tabela 1 – Características dos Diferentes Tipos de Memória. Adaptado de [19, 20].

	HDD	**SSD**	**DRAM**	**SRAM**
Latência de Leitura	3 ms	25 μs	15 ns	2-3 ns
Latência de Escrita	3 ms	300 μs	15 ns	2-3 ns
Endereçamento	Bloco	Bloco	Byte	Byte
Volatilidade	Não	Não	Sim	Sim
Densidade	Alta	Alta	Média	Baixa
Custo por bit	Baixo	Baixo	Médio	Alto
Duração (# de escritas/bit)	$> 10^{18}$	10^5	10^{18}	$> 10^{18}$

endereçamento, densidade, velocidade e custo.

A volatilidade da memória refere-se a necessidade de alimentação contínua de energia elétrica para manutenção de seu conteúdo, ou seja, a ausência desta implica a exclusão de todo conteúdo presente na memória. Memórias não-voláteis são comumentemente utilizadas para armazenamento, enquanto as voláteis são utilizadas como memória de trabalho, devido principalmente ao tempo de acesso inferior.

O endereçamento especifica a forma de acesso à memória, podendo ser realizada de duas maneiras distintas: por byte ou por bloco. Endereçamento por byte permite o acesso a regiões de memória com uma granularidade mais fina, capaz de selecionar apenas o byte requisitado, enquanto o endereçamento por bloco carrega as informações em blocos de tamanho fixo, e.g. 512B do HDD, sendo necessário certo processamento via software para extração de pequenas frações necessárias pelas aplicações.

A densidade de memória é mensurada através da quantidade de bits capazes de serem armazenados em uma determinada área de chip. Usualmente deseja-se uma alta densidade, permitindo que um grande volume de dados seja armazenado em um mesmo espaço físico, entretanto as limitações impostas pelas tecnologias utilizadas na fabricação de determinados tipos de memórias, além do elevado custo associado, podem inviabilizar a escalabilidade destes.

A Tabela 1 apresenta os tipos de memória e suas respectivas especificidades. O HDD e o SSD, memórias não-voláteis, destacam-se principalmente pela alta densidade e pelo baixo custo por bit, sendo utilizados como memória secundária, *storage*. A grande capacidade destes contrasta com sua velocidade de acesso, as piores entre os tipos de memória previamente citados. Um fator preponderante no elevado tempo de acesso é o *overhead* associado ao endereçamento por blocos, requisitando passos intermediários para leitura/escrita de seu conteúdo, tais como: utilização do barramento de I/O, serialização dos dados e a utilização de *drivers* específicos para

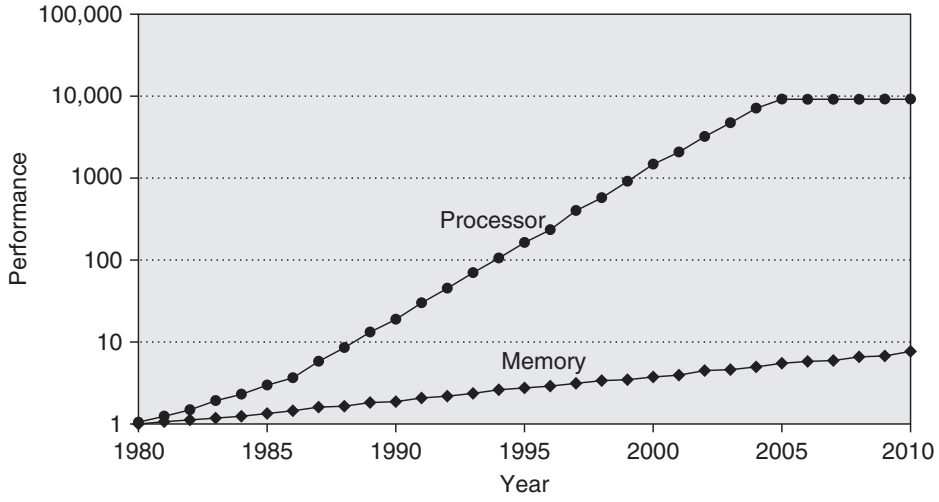


Figura 3 – Lacuna de desempenho entre o Processador e a Memória. Diferença de tempo entre as requisição de memória por um único processador e a latência de acesso à DRAM. Extraído de [15].

comunicação.

Os tipos de memória volátil, DRAM e a SRAM, destacam-se por proporcionarem altas velocidades de acesso aliado ao endereçamento por byte, o qual permite a inserção destas diretamente no barramento de memória, e consequentemente eliminando procedimentos intermediários que acarretam perda de desempenho.

A DRAM, constituída de um transistor e um capacitor por célula, proporciona uma densidade maior e um menor custo por bit que a SRAM, porém apresenta velocidades de acesso inferiores e um custo energético maior, proveniente da necessidade de constantes *refreshes*, em intervalos de milissegundos, para evitar a perda dos dados armazenados em seus capacitores. A SRAM, por sua vez, é constituída por transistores, reduzindo o consumo de energia para retenção de seu conteúdo. Porém, o tamanho da célula, o qual requer seis transistores para armazenar um bit, impossibilita a fabricação de memórias de grande densidade, compactas e a um custo acessível.

Neste contexto, a utilização da DRAM como memória principal, ou memória de trabalho, e o disco ou SSD como *storage*, tornou-se uma solução mais viável. No entanto, há uma enorme discrepância de desempenho existente entre a velocidade de processamento de um microprocessador e a latência de acesso à DRAM. Este gargalo é conhecido na literatura como *Memory Wall* [21], sendo alvo de constantes pesquisas. Este fenômeno pode ser observado na Figura 3, a qual apresenta a projeção da evolução do desempenho de um processador, com apenas um núcleo, em comparação ao aumento de desempenho no tempo de acesso à memória

principal. A curva correspondente ao processador mostra o aumento das requisições de memória por segundo em média, enquanto a curva de memória apresenta o aumento da quantidade de acessos por segundo à memória principal [15].

2.2 Memória Cache

As constantes requisições de dados e instruções resultariam em um gigantesco *slowdown*, degradando consideravelmente os esforços empreendidos no aumento da velocidade de processamento dos microprocessadores. Neste contexto, foi proposto a inserção de uma memória de tamanho suficientemente pequena, com tempo de acesso diminuto, mais próxima às unidades de processamento, originando a memória cache.

A cache procura armazenar o *working set*, conjunto de dados e instruções utilizados por determinada aplicação, em pequenas regiões de memória, reduzindo seu custo e consumo de energia, possibilitando acessos extremamente rápidos. Para tal, faz uso da tecnologia SRAM, capaz de proporcionar latência de acesso diminuta, conforme pode ser observado na Tabela 1, e consequentemente possibilitando que o processador fique minimamente ocioso.

A cache explora os princípios de localidade temporal e espacial. A localidade temporal indica a tendência de uma aplicação referenciar uma certa quantidade de dados múltiplas vezes em determinado período de tempo, tais como a constante atualização de uma variável durante a execução de um laço. A localidade espacial destaca a probabilidade da utilização de dados relativamente próximos, aos previamente utilizados, serem referenciados futuramente pela aplicação, tais como o somatório dos valores presentes em um vetor.

A correta manutenção de dados e instruções na cache é essencial para esconder a latência de memória. Os princípios de localidade permitem um elevado índice de *hit* na cache, ou seja, os dados estarem presentes na cache no momento que são necessários. Porém para obter tal índice de acertos é necessário a elaboração de estratégias que evite os principais causadores de *misses*, faltas, na cache: a falta compulsória, a falta por capacidade e a falta por conflito.

A falta compulsória caracteriza-se pela ausência do primeiro bloco a ser acessado na cache, obrigando a busca de tais informações em níveis mais elevados da hierarquia de memória, por consequência expondo a latência de acesso a memória e reduzindo o desempenho das aplicações. A falta por capacidade caracteriza-se pela falta de espaço na cache para manutenção do *working set* das aplicações, sendo necessário aumentar o tamanho da cache ou alterar as

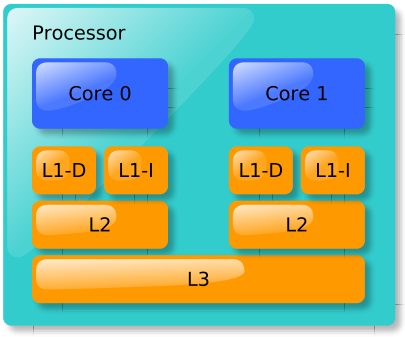


Figura 4 – Organização dos níveis de cache presentes em um microprocessador.

aplicações de forma que o *working set* de cada fase de execução caiba na cache. Por fim, a falta por conflito caracteriza-se pela colisão entre duas regiões de memória, ambas necessárias pelas aplicações, mapeadas para o mesmo bloco da cache, resultando em uma constante troca. O aumento da associatividade da cache minimiza a incidência de conflitos, ou seja, cada bloco de memória pode ser mapeado para dois ou mais blocos na cache.

Outro fator preponderante no projeto da cache é o *trade-off* associado à correlação entre velocidade e taxa de *hit*. Caches grandes possibilitam uma elevada incidência de *hits*, entretanto a latência associada à adição de circuitos digitais nesta aumentam o tempo de acerto. Para solucionar tal problema, os processadores multicore utilizam múltiplos níveis de cache, cada nível com um respectivo tamanho, visando otimizar o acesso a estas.

A Figura 4 ilustra uma possível organização dos níveis de cache presente nos processadores contemporâneos, na qual destaca-se a divisão do primeiro nível, L1, em cache de dados e de instruções, evitando constantes conflitos de ocupação entre ambos. Conforme pode ser observado, os níveis 1 e 2 da cache, L1 e L2 respectivamente, estão privados para cada núcleo, enquanto o nível 3, L3, é compartilhado entre os núcleos, visando equacionar o *trade-off* associado entre a velocidade de acesso e a correta manutenção dos dados, também conhecida como coerência de cache.

Grande parte das operações realizadas em memória cache são de leitura, entretanto a forma em que são tratadas as escritas são de vital importância no projeto e organização da arquitetura do sistema. A cache possui duas políticas de escrita: *write-through* e *write-back*. A política *write-through* caracteriza-se pela atualização do dado tanto na memória cache quanto nos demais níveis de memória, tais como a memória principal e o disco. Tal configuração requer menor complexidade de gerenciamento, porém acarreta uma possível perda em desempenho. A

Tabela 2 – Definição dos estados das linhas de cache pelo Protocolo MESI.

Estado	**Descrição**
Modificado	A linha se encontra somente na cache corrente e seu valor foi alterado
Exclusivo	A linha se encontra somente na cache corrente e não sofreu alterações
Compartilhado	A linha é compartilhada com as demais caches
Inválido	A linha é inválida, seu valor foi alterado em alguma outra cache

política *write-back* caracteriza-se pela atualização do dado somente na cache, sendo necessário sua escrita nos demais níveis de memória somente quando o bloco da cache precisar ser substituído. Grande parte dos sistemas contemporâneos adotam a política *write-back*.

Postergar a transmissão de atualizações implica certas precauções a serem tomadas para garantia de coerência de cache. Ao considerarmos a organização da cache apresentada na Figura 4, as modificações realizadas pelos *cores* permanecem nos níveis privados de cache, L1 e L2, por um período arbitrário de tempo. Tal proceder acarreta em inconsistências caso os valores modificados sejam compartilhados entre as caches dos demais *cores*, visto que os níveis privados de cache dos demais *cores* não refletem os valores mais recentes.

A manutenção da coerência entre múltiplas caches é obtida através dos protocolos de coerência de cache. Estes protocolos operam através de duas principais características: o estado associado a cada linha de cache e a comunicação realizada entre os controladores de cache. De forma a simplificar a explanação dos estados das linhas de cache, será considerado o protocolo MESI [22], utilizado como base pelos protocolos presentes nos microprocessadores atuais. Cada linha de cache utiliza dois bits adicionais indicando um dos seguintes estados: Modificado, Exclusivo, Compartilhado (*Shared*) e Inválido. A Tabela 2 apresenta as descrições dos estados citados.

O design e implementação da comunicação dos estados das linhas de cache entre os controladores de cache dependem estritamente da viabilidade da utilização de *broadcasts* ou não, ou seja, a capacidade de envio de uma mensagem para muitos receptores ao mesmo tempo. Neste contexto, duas abordagens distintas de protocolos de coerência de cache foram propostos: *snoopy* e *directory*.

O protocolo *snoopy* é implementado utilizando um barramento compartilhado, onde as mensagens enviadas via *broadcast* são interceptadas pelos controladores de cache, os quais respondem apropriadamente se possuem uma cópia da linha que esteja sendo processada [23]. O protocolo *directory* mantém um diretório responsável por indicar o status de cada linha de cache

e suas respectivas localizações. No momento em que um *core* atualiza uma cópia de um valor armazenado em sua cache, este verifica a entrada no diretório correspondente a linha de cache modificada. Caso esta linha seja compartilhada com as demais caches, um sinal de invalidação é enviado a cada controlador de cache presente na lista de localizações.

Os protocolos de coerência de cache exercem um papel fundamental na manutenção de corretude dos valores armazenados, possibilitando extrair o máximo desempenho advindo dos diminutos tempos de acesso proporcionados pela memória cache.

2.3 Hierarquia de Memória e Otimizações

A Figura 5 apresenta as diferentes tecnologias de memória descritas organizadas em uma hierarquia, comumentemente referenciada por hierarquia de memória. Conforme pode ser observado, os tipos de memória mais próximos ao processador apresentam menor latência de acesso, tais como os registradores, pequenas regiões de memória de alta velocidade que permitem o armazenamento de valores intermediários ou informações de controle, e a memória cache. Os dois últimos níveis, DRAM e disco, por apresentarem maior densidade são utilizados como memória de trabalho e armazenamento, respectivamente.

A implementação de uma hierarquia de memória, apesar de adicionar certa complexidade, permite amenizar a latência de acesso, possibilitando a extração das melhores características de cada componente da hierarquia, tais como: a velocidade de acesso de componente mais rápido, baixo consumo de energia e um custo acessível [24].

O surgimento de novos microprocessadores e da hierarquia de memória fomentou a pesquisa e o desenvolvimento de técnicas para esconder ao máximo a latência de acesso à memória, permitindo que o processador fique minimamente ocioso a espera de dados ou instruções para processamento. Entre as técnicas desenvolvidas se destacam: *Prefetching*, Memória Virtual e Troca de Contexto.

Prefetching é uma técnica de predição que busca solucionar a falta compulsória na cache, ou seja, traz os dados ou instruções para cache antes dos mesmos serem utilizados [23]. Tal técnica pode ser implementada tanto em software, no qual o programador ou o compilador insere as instruções de *prefetching* providas pelo conjunto de instruções da arquitetura, *Instruction Set Architecture* (ISA), quanto em hardware, onde este monitora os acessos e encontra padrões, gerando requisições de *prefetch* automaticamente.

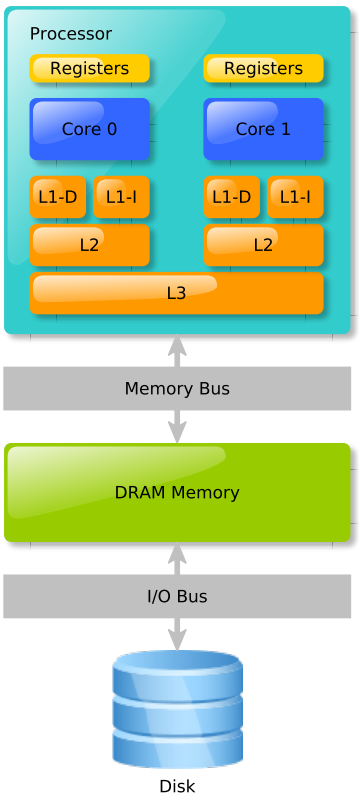


Figura 5 – Hierarquia de Memória presente nos Microprocessadores atuais.

Prefetching adiciona um novo critério a ser observado para verificação de desempenho, *timeliness*, o qual envolve a constatação do tempo em que o *prefetching* foi iniciado e seus respectivos impactos de desempenho. Caso o processo tenha iniciado muito cedo, é possível que os dados buscados sejam removidos da cache antes mesmo de sua utilização. Caso o processo seja tardio, não permitirá esconder totalmente a latência de memória, apenas parte dela.

O adição da cache aliada à utilização do *prefetching* possibilitam esconder a latência de acesso à DRAM. No entanto, as latências associadas às tecnologias utilizadas como *storage* constituem grande parte da degradação de desempenho do sistema, sendo necessário a integração de outras técnicas capazes de mitigar tal degradação. A primeira das técnicas empregadas foi a Memória Virtual.

A Memória Virtual foi inicialmente desenvolvida com o intuito de expandir o espaço de endereçamento dos processos, criando a ilusão de uma memória extremamente grande, disco, com velocidade de uma DRAM. Para tal, a memória virtual e a memória física são divididas em páginas de tamanho fixo, 4KB na maioria dos sistemas, sendo referenciadas através de endereços virtuais e endereços físicos respectivamente. Desta forma, a memória virtual utiliza a DRAM

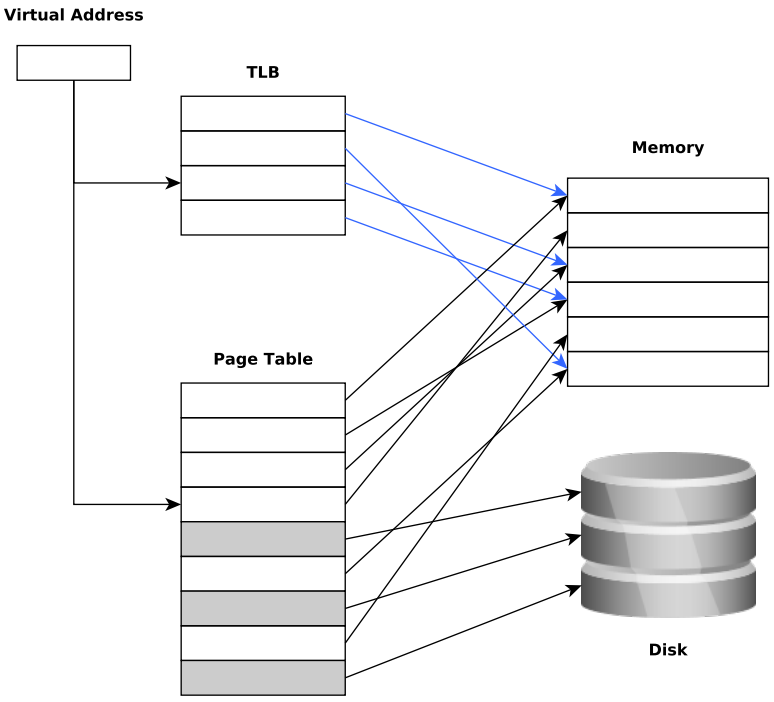


Figura 6 – Processo de tradução de endereços virtuais e verificação das páginas.

como uma cache para acesso a disco, armazenando as páginas que são constantemente acessadas na memória, buscando reduzir a quantidade de acessos ao disco.

A tradução de endereços virtuais em físicos é realizada pela tabela de páginas, sendo esta também responsável pela verificação da presença das páginas na memória. Para acelerar o processo de verificação da tabela de páginas, armazenada na DRAM, foi inserido uma pequena cache para a tradução dos endereços virtuais, chamada de *Translation Lookaside Buffer* (TLB). Em caso de acerto na TLB o endereço físico da página é retornado rapidamente, entretanto se ocorrer uma falta, será necessário acessar a tabela de páginas na memória física. A Figura 6 ilustra o esquema de verificação das páginas.

Considerando as técnicas para esconder a latência de memória descritas até o momento, o processo de busca dos dados e instruções na hierarquia de memória pode ser explicitado. O acesso aos dados é feito verificando se estes estão presentes na cache, concomitantemente ocorre a tradução dos endereços virtuais para endereços físicos para verificação da presença da página na DRAM. Se os dados não forem encontrados na cache mas a página estiver na DRAM, os dados são trazidos da memória para a cache, entretanto se ocorrer uma falta de página o processador gera uma exceção que é capturada pelo sistema operacional, este ficando encarregado de localizar a página no disco e trazê-la para a memória.

O último método para esconder a latência de memória visa resolver o problema causado pela falta na tabela de páginas, a troca de contexto. A interrupção gerada pelo processador é atendida pelo sistema operacional, que inicia o processo de I/O para busca da página no disco. Simultaneamente, o sistema operacional salva o estado do processo, que gerou tal interrupção, e retorna o controle da CPU para a execução de algum outro processo. Após o término da busca da página no disco o sistema operacional restaura o estado do processo previamente suspenso.

Como pode ser observado, as técnicas desenvolvidas foram otimizadas para esconder a latência de acesso a memória em todos os níveis da hierarquia de memória, possibilitando grandes ganhos em desempenho.

2.4 Memória Persistente

O crescente volume de dados demandado pelas aplicações, aliado a disposição de múltiplos *cores* pelos processadores, tem tornado mais evidente o gargalo de desempenho associado ao subsistema de memória. Apesar das várias otimizações arquiteturais propostas para esconder a latência de acesso à memória, a grande demanda de dados pelas aplicações ainda esbarra em eventuais faltas de páginas e outros possíveis problemas que forçam a constante busca de dados no disco.

Neste contexto, a ampliação da quantidade de memória principal disponível no sistema mostrou-se bastante eficaz, viabilizando a expansão do *working set* das aplicações e consequentemente mitigando o elevado número de acessos ao disco. Várias soluções em software usufruem desta expansão, entre elas destacam-se os bancos de dados em memória, *In-Memory Database* (IMDB), os quais usufruem da velocidade de acesso da DRAM e dispõem de sub-rotinas para garantia de consistência em eventuais quedas de energia ou falhas de sistema, visto que seu conteúdo seria totalmente perdido nestas ocasiões.

A expansão da quantidade de memória principal, creditada a grande capacidade de escalabilidade da DRAM, estimulou uma mudança no projeto de software, antes baseado na tradicional abordagem centrada no disco, para aplicações centradas na massiva utilização da memória principal[25].

No entanto, as recentes limitações tecnológicas demonstradas pela DRAM [26] têm inviabilizado a continuidade de sua expansão. Entre os obstáculos encontrados, destaca-se o elevado custo energético associado à manutenção de seu conteúdo, diretamente pelos constantes

Tabela 3 – Características das tecnologias de NVM. Adaptado de [20].

Tecnologia	**Latência de Leitura (ns)**	**Latência de Escrita (ns)**	**Duração (# de escritas por bit)**
PCM	50	150	10^{10}
STT-RAM	20	20	10^{15}
RRAM	100	100	10^{8}

refreshes e indiretamente através do sistema de resfriamento, além da dificuldade na manutenção de confiabilidade dos dados devido a interferências causadas pelo circuitos, tal como a corrupção de dados causada pelas ativações subsequentes de uma mesma linha de memória em um intervalo de *refresh*, conhecido na literatura como *row hammer* [26].

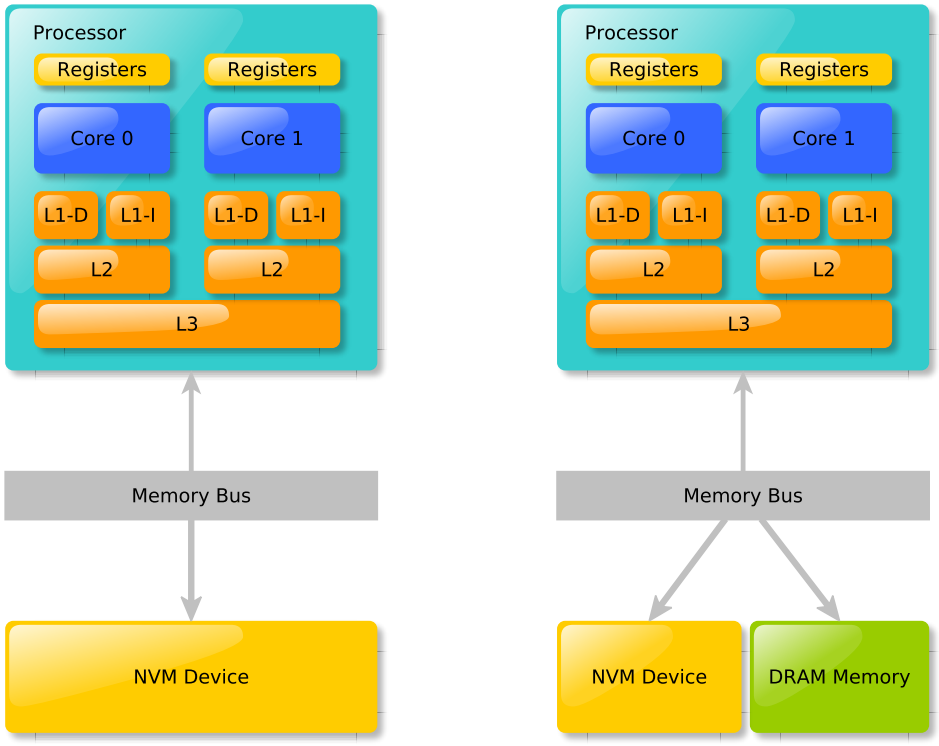
Estas limitações em conjunto com a elevada demanda de dados pelas aplicações evidenciam a necessidade por melhores tecnologias de memória, capazes de solucionar os *trade-offs* impostos pelos exemplares até então disponíveis. Neste âmbito, as recentes tecnologias de memória persistente, comumentemente referenciadas por *Non-Volatile Memory* (NVM) [1], prenunciam uma drástica aceleração de desempenho em aplicações com uso intensivo de dados.

A NVM apresenta as seguintes características: (i) persistência; (ii) endereçamento por byte; (iii) alta densidade; (iii) acesso via instruções de *load* e *store*; (iv) tempos de acesso semelhantes à DRAM. Tais características rompem com a tradicional distinção de memórias até então estabelecida, memórias não-voláteis de alta densidade, acessadas por bloco e com elevado tempo de resposta, utilizadas como *storage*, e memórias voláteis com menor densidade, acessadas por byte e com tempo de acesso diminuto, utilizadas como memória de trabalho.

Há uma grande expectativa que a NVM substitua a DRAM [27, 28] ou complemente esta, proporcionando uma memória principal híbrida [29, 30]. A completa substituição da DRAM, conforme a Figura 7a, resultaria em uma degradação de desempenho proveniente das latências de acesso ainda serem maiores comparadas à DRAM. Tal fato pode ser observado na Tabela 3, onde são apresentadas características de três principais tecnologias de NVM em desenvolvimento: *Phase-Change Memory* (PCM) [31], *Spin-Transfer Torque Random Access Memory* (STT-RAM) [32] e *Resistive Random Access Memory* (RRAM) [33].

Demais fatores tecnológicos também impactam negativamente no desempenho obtido, tais como o custo energético das escritas na PCM ser maior que das operações de leitura, além

[1] Demais nomenclaturas encontradas na literatura: *Storage-Class Memory* (SCM), *Byte-Addressable Persistent Random Access Memory* (BPRAM), *Nonvolatile Random Access Memory* (NVRAM), *Non-Volatile Byte-addressable Memory* (NVBM) e *Persistent Random Access Memory* (PRAM).



(a) (b)

Figura 7 – Novas Hierarquias de Memória com uso de NVM.

da menor durabilidade apresentada por estas tecnologias, requisitando técnicas para redução da quantidade de ativações dos bits nas operações de escrita.

A hierarquia com memória principal híbrida, ilustrada na Figura 7b, se apresenta mais factível em questões de desempenho até o presente momento. A inserção da NVM diretamente no barramento de memória permite o mapeamento de regiões persistentes de memória diretamente no espaço de endereçamento dos processos, eliminando grande parte do *overhead* proveniente das camadas intermediárias de software, e.g., sistema operacional, *drivers*, *Kernel*, entre outros.

A Figura 8 apresenta a sequência de procedimentos realizados para exposição da memória persistente às aplicações. Inicialmente a aplicação solicita determinado arquivo ou região de memória ao sistema de arquivos, o qual interage com o *Driver* para efetuar possíveis ações necessárias para descobrimento ou configuração da NVM. Em seguida, o sistema operacional mapeia o arquivo da memória para o espaço de endereçamento do processo, através da utilização do `mmap`, possibilitando a leitura e escrita do arquivo através das instruções *load* e *store*, sem as constantes chamadas de sistema para acesso ao arquivo. Após o mapeamento a aplicação tem livre acesso à memória persistente através da *Memory Management Unit* (MMU), responsável pela tradução dos endereços virtuais em endereços físicos.

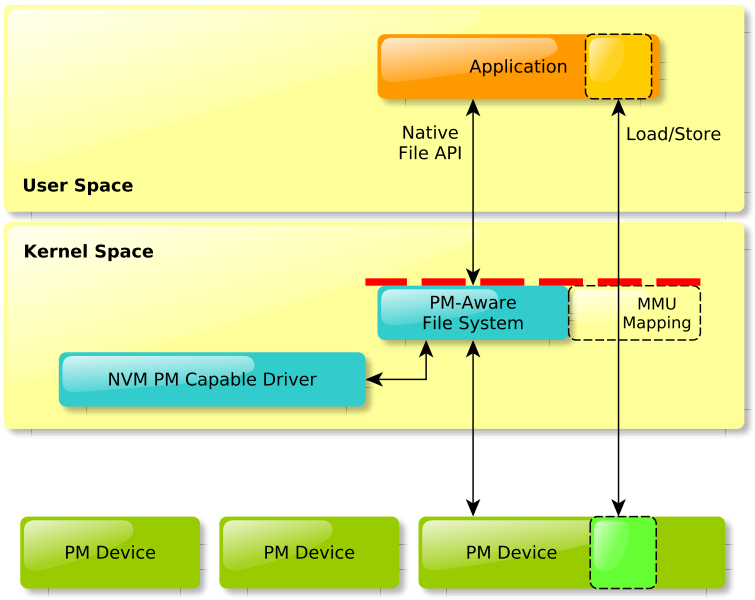


Figura 8 – Representação gráfica do processo de mapeamento de uma região persistente de memória. Extraído de [34].

O mapeamento de regiões não-voláteis de memória no espaço de endereçamento dos processos simplifica a persistência dos dados, porém tal procedimento não garante a consistência destes em eventuais falhas de sistema ou quedas de energia. As inconsistências são originadas por dois principais motivos: as escritas realizadas em regiões persistentes não são atômicas, podendo residir arbitrariamente na cache antes de sua persistência, e a modificação da ordem das escritas definida pelas aplicações, realizada pelos microprocessadores para extração de paralelismo a nível de instrução.

A primeira adversidade ocasiona perda de parte do estado da aplicação, previamente armazenada na cache, possibilitando que a região persistente de memória mapeada não reflita o último estado consistente da aplicação. A segunda adversidade possibilita o vazamento de memória, e.g., a inversão de atribuição de endereços de ponteiros em uma estrutura de dados, causando uma corrupção permanente de memória.

Diversas estratégias foram propostas visando equacionar a garantia de consistência em ambientes com memória persistente. Entre estas estratégias destacam-se a modificação de estruturas de dados convencionais [4, 19, 35], a adição de persistência em tradicionais mecanismos de controle de concorrência [36, 37, 38] e a utilização de Memória Transacional [6, 3, 36, 7].

Este trabalho baseia-se na utilização de Memória Transacional, responsável pelo gerenciamento de aplicações concorrentes e pela garantia de consistência em ambientes com memória

persistente, reduzindo os encargos do programador e facilitando sua respectiva adoção.

2.5 Memória Transacional

Memória Transacional (TM) é uma abstração conceitual de computação paralela inspirada no modelo de programação de banco de dados [2]. O conceito chave desta abordagem é a utilização de transações, responsáveis por delimitar um conjunto de operações a serem realizadas de maneira atômica e indivisível.

As transações, definidas inicialmente por Eswaran [39], obedecem as seguintes propriedades que definem o acrônimo ACID (Atomicidade, Consistência, Isolamento e Durabilidade). O último atributo, Durabilidade, usualmente não era considerado no contexto de TM, visto que as operações eram realizadas exclusivamente em memória volátil. Entretanto, o emprego de TM para provisão de suporte à memória persistente traz novamente um enfoque a esta propriedade.

As definições das propriedades das transações são elencadas a seguir:

- Atomicidade: uma transação é dita atômica se todas as operações presentes no bloco atômico, conjunto de instruções executadas de maneira indivisível, são efetivadas, ou então nenhuma o é;
- Consistência: a efetivação de uma transação deve garantir a transição de um estado previamente consistente para um novo estado consistente de memória;
- Isolamento: os passos intermediários das transações não são visíveis as demais transações, garantindo assim o isolamento entre estas;
- Durabilidade: as alterações efetivadas pelas transações são duráveis, possibilitando sua recuperação em uma eventual falha no sistema.

As transações em memória aumentam o nível de abstração para o controle de concorrência em programas concorrentes, possibilitando ao programador não se preocupar em como garantir a sincronização, e sim especificar o que deve ser executado atomicamente, além de explorar o paralelismo de forma mais agressiva, viabilizando a execução de métodos concorrentemente em situações em que os acessos são a dados disjuntos [40].

As implementações de TM utilizam dois conceitos chaves: versionamento de dados e gerenciamento de conflitos. O mecanismo de versionamento é encarregado de gerenciar a versão corrente e a especulativa dos dados acessados durante a execução da transação. A versão

corrente se refere ao valor do dado previamente a execução da transação, enquanto a versão especulativa reflete o valor intermediário produzido pela transação. Ao término da transação, o valor especulativo torna-se corrente caso a transação seja efetivada ou, caso contrário, o valor especulativo é descartado mantendo-se o valor corrente.

Há duas estratégias de versionamento: versionamento diferido, também conhecido na literatura como *lazy versioning*, e versionamento direto, *eager versioning*. No versionamento diferido, os valores especulativos são mantidos internos a transação, em um *redo log*, sendo transferidos para memória somente durante a fase de efetivação da transação (*commit*). Esta estratégia é vantajosa quando as transações são frequentemente abortadas, bastando descartar os valores especulativos.

No versionamento direto, os valores especulativos são armazenados diretamente na memória compartilhada, enquanto os valores correntes são registrados em um *undo log*. Ao término da transação, os valores já estão efetivados na memória, removendo o procedimento de transferência dos valores para memória. No entanto, caso uma transação seja abortada, é necessário uma estratégia de *rollback*, responsável por desfazer as alterações realizadas através da transferência dos valores correntes mantidos no *undo log* para a memória.

O gerenciamento de conflitos abrange as fases de detecção dos conflitos e resolução destes. Para realização da detecção dos conflitos, as transações mantém um conjunto de leitura e um conjunto de escrita, contendo os endereços lidos e os modificados respectivamente. O conflito é caracterizado pela presença de um mesmo endereço em múltiplos conjuntos, no qual ao menos um destes seja de escrita, ou seja, duas ou mais transações acessam um mesmo elemento e um dos acessos é de escrita.

Assim como no mecanismo de versionamento, há duas estratégias distintas para detecção de conflitos: pessimista, *eager conflict detection*, e otimista, *lazy conflict detection*. A detecção pessimista presume uma elevada frequência de conflitos, desta forma a verificação é realizada no momento de acesso às posições de memória, constatando se estas já foram previamente alteradas por alguma transação em andamento. Na detecção otimista, a verificação é postergada até a fase de efetivação da transação.

Em cenários onde os conflitos são frequentes a estratégia pessimista reduz o desperdício de trabalho ao cancelar uma transação antecipadamente. No entanto, em cenários onde os conflitos são esporádicos, a estratégia otimista permite que transações conflitantes prossigam na

esperança do conflito não se efetivar de fato [40].

A resolução de conflitos fica a cargo do gerenciador de contenção, componente responsável pela garantia de progresso do sistema e por reduzir a probabilidade de ocorrência de futuros conflitos. Diversas políticas de gerenciamento de contenção foram propostas, tais como *suicide* e *backoff* que especificam o cancelamento e imediato reinício da transação que detectou o conflito ou a espera arbitrária de tempo antes de seu reinício, respectivamente. A escolha da melhor política é guiada, assim como o mecanismo de detecção, pelas características das aplicações e da carga de trabalho das mesmas [41].

As implementações de sistemas transacionais são comumentemente dividas em software, hardware e híbridas. As subseções a seguir apresentam aspectos referentes as implementações em software e hardware, destacando os *trade-offs* associados a cada uma destas.

2.5.1 Implementação em Software (STM)

A concepção de Memória Transacional, realizada por Herlihy [42], vislumbrava o suporte a esta em hardware, no entanto as limitações físicas e arquiteturais enfrentadas inviabilizaram a disponibilidade de uma versão comercial até a última década. Neste intuito, a *Software Transactional Memory* (STM), originalmente desenvolvida por Shavit e Touitou [43], se consolidou como uma importante alternativa às tradicionais diretivas de sincronização baseadas em travas. Entre as vantagens oferecidas pela STM estão: (i) flexibilidade na implementação dos algoritmos; (ii) não obrigatoriedade de suporte transacional em hardware para sua execução; (iii) possibilidade de execução de transações com amplos conjuntos de leitura e escrita.

Além das características apresentadas na Seção 2.5, a STM instrumenta os acessos realizados pelas transações através de barreiras de leitura e escrita, incorporando os endereços de memória nos respectivos conjuntos de leitura e escrita de cada transação. As barreiras podem ser observadas na Tabela 4, onde é apresentada uma *Application Programming Interface* (API) típica de STM.

As implementações em software são categorizadas em bloqueantes, isto é, fazem uso de travas em sua implementação, ou não-bloqueantes, os quais utilizam técnicas como indireção para garantia de modificações atômicas pelo sistema.

Para melhor exemplificar uma implementação em STM, será descrita a solução NOrec [44], utilizada na elaboração deste trabalho. O NOrec é uma STM bloqueante que apresenta as

Tabela 4 – Descrição de uma API típica de STM. Extraído de [40].

Interface	**Descrição**
`TxStart`	Marca o início de uma nova transação.
`TxCommit`	Realiza uma tentativa de término da transação. Se a transação for efetivada com sucesso, os dados especulativos ficam visíveis para todo o sistema de forma atômica. Caso contrário, a transação é cancelada e o fluxo de execução retorna para a instrução seguinte ao comando `TxStart`.
`TxAbort`	Cancela a transação atual. Essa primitiva é chamada implicitamente pela STM quando a operação `TxCommit` falha, mas pode também ser invocada explicitamente.
`TxLoad`(l)	A palavra contida na posição de memória l é lida e retornada.
`TxStore`(l, v)	O valor v é armazenado na posição de memória l.

seguintes características: (i) o uso de uma trava sequencial global para proteção da efetivação das transações; (ii) o conjunto de escrita é indexado através de uma função *hash*, reduzindo o *overhead* de verificação da presença de um endereço de memória neste; (iii) a detecção de conflitos fica a cargo do processo de validação baseado em valores (*value-based validation*), eliminando a tradicional tabela de registro de posse, tabela *hash* contendo o registro de posse de cada posição de memória lida e/ou escrita pelas transações.

A trava sequencial, utilizada durante a fase *commit* das transações, apresenta a grande vantagem de ser escalável em cenários de baixa taxa de atualizações, visto que a aquisição desta é restrita aos acessos de escrita. A aquisição e liberação desta trava são realizadas através de incrementos de um contador, no qual valores ímpares do contador indicam a aquisição do *lock*, enquanto os pares indicam que o *lock* está liberado.

O ato de postergar seu uso para a fase de efetivação proporciona benefícios, tais como a redução do período em que uma escrita mantém a trava adquirida, aumentando a probabilidade de transações concorrentes, contendo somente operações de leitura, serem efetivadas, além das operações especulativas de leitura e escrita poderem proceder concorrentemente, devido também à adoção de uma estratégia otimista de detecção de conflitos.

As operações especulativas, instrumentadas pelas barreiras transacionais, são armazenadas em dois conjuntos distintos: um *redo log* implementado através de uma tabela *hash* para manutenção das escritas, versionamento diferido, e uma lista contendo as entradas lidas e os respectivos endereços.

O Algoritmo 1 apresenta os metadados utilizados pelo NOrec, onde o *lock* atua basica-

mente como um contador global indicando a efetivação de escritas para as demais transações. Cada transação mantém localmente conjuntos de leitura e escrita e um *snapshot* do valor do *lock*, utilizado para validação das leituras especulativas mediante comparação ao valor global.

```
1:  volatile unsigned global_lock
2:
3:  local unsigned snapshot
4:  local List<Address, Value> reads
5:  local Hash<Address, Value> writes
```

Algoritmo 1 – Metadados utilizados pelo NOrec. Extraído de [44].

A estratégia de contenção, realizada pelo processo de validação, consiste na releitura dos endereços, presentes no conjunto de leitura, e na verificação de um ponto no tempo em que as leituras especulativas foram realizadas de maneira atômica [44], isto é, os valores do *snapshot* e do *lock* sejam idênticos de forma que o conjunto de leitura da transação reflita os valores correntes do sistema.

```
1:  unsigned Validate()
2:    while (true)
3:      time = global_lock
4:      if ((time & 1) != 0)
5:        continue
6:
7:      for each (addr, val) in reads
8:        if (*addr != val)
9:          TxAbort()
10:
11:     if (time == global_lock)
12:       return time
```

Algoritmo 2 – Processo de validação do conjunto de leitura. Extraído de [44].

Os detalhes do procedimento de validação são apresentados no Algoritmo 2. O primeiro passo do algoritmo é a leitura do contador associado a trava global e a subsequente verificação da disponibilidade do *lock*, linhas 3 e 4. Caso o *lock* esteja adquirido por alguma transação, o processo de validação é reiniciado. Caso contrário, é feita uma varredura das entradas presentes no conjunto de leitura, verificando se estas não sofreram modificações, linhas 7 a 9. Qualquer diferença encontrada durante a varredura força o cancelamento da transação. Por fim, há uma checagem de possível interferência oriunda da efetivação de alguma transação de escrita, ocasionando o reinício do processo de validação.

A construção do conjunto de leitura de uma transação é de total responsabilidade da barreira de leitura. Ao se executar uma leitura especulativa, primeiro é verificado a existência desta

entrada no conjunto de escrita. Tal procedimento é adotado para evitar conflitos *read-after-write*, em outras palavras, leituras subsequentes às escritas de entradas transacionais devem refletir o valor especulativo e não o corrente. No caso de a posição de memória não se encontrar no conjunto de escrita, o valor é lido da memória compartilhada. O Algoritmo 3 apresenta a sequência de operações que compõem a barreira de leitura.

```
1:  Value TxLoad(Address addr)
2:    if (writes.contains(addr))
3:      return writes[addr]
4:
5:    val = *addr
6:    while (snapshot != global_lock)
7:      snapshot = Validate()
8:      val = *addr
9:
10:   reads.append(addr, val)
11:   return val
```

Algoritmo 3 – Barreira de leitura. Extraído de [44].

As linhas 6 a 8 apresentam uma pós-validação da entrada lida, de forma a garantir que a execução especulativa da transação não possua um estado inconsistente [44]. Por fim, o valor e seu respectivo endereço são inseridos no conjunto de leitura, linha 10.

Em contraste a barreira de leitura, a implementação da barreira de escrita é bem simplificada. Conforme pode ser observado no Algoritmo 4, o único procedimento realizado é a inserção da nova entrada e seu respectivo endereço no conjunto de escrita da transação.

```
1:  void TxStore(Address addr, Value val)
2:    writes[addr] = val
```

Algoritmo 4 – Barreira de escrita. Extraído de [44].

Outra operação simplificada oferecida pelo NOrec é o procedimento de início de uma transação, demonstrado no Algoritmo 5, restrito à inicialização do *snapshot* com um valor que reflita o estado consistente mais recente do sistema, ou seja, nenhuma transação está com a trava global adquirida.

```
1:  void TxStart()
2:    do
3:      snapshot = global_lock
4:    while ((snapshot & 1) != 0)
```

Algoritmo 5 – Início de uma transação. Extraído de [44].

Para finalizar a API disponibilizada pelo NOrec, o procedimento de efetivação é apresentado pelo Algoritmo 6. Todas as transações que iniciam a fase de efetivação possuem um *snapshot* e a garantia de consistência devido a pós-validação da barreira de leitura [44]. Desta forma, transações exclusivas de leitura podem ser efetivadas sem demais verificações, conforme as linhas 2 e 3.

Transações de escrita buscam adquirir o *lock* via instrução *Compare-And-Swap* (CAS), utilizando o *snapshot* como valor esperado do estado global do sistema. A falha na execução do CAS indica a existência de uma transação de escrita concorrente a esta, requisitando a validação novamente da transação para garantia de consistência, linhas 5 e 6. Após a aquisição do *lock*, as modificações especulativas tornam-se os valores correntes na memória compartilhada, linhas 8 e 9. Por fim, o *lock* é liberado através do incremento de seu contador, linha 11.

```
 1:  void TxCommit()
 2:    if (read-only transaction)
 3:      return
 4:
 5:    while (!CAS(&global_lock, snapshot, snapshot+1))
 6:      snapshot = Validate()
 7:
 8:    for each (addr, val) in writes
 9:      *addr = val
10:
11:    global_lock = snapshot + 2
```

Algoritmo 6 – Efetivação de uma transação. Extraído de [44].

Apesar das vantagens proporcionadas pelo NOrec, duas adversidades impactam o desempenho proporcionado por esta abordagem: o gargalo oriundo do limite de efetivação de apenas uma transação de escrita por vez, e o elevado *overhead* de execução das transações, comparado a implementação em hardware.

2.5.2 Implementação em Hardware (HTM)

As implementações em hardware, usualmente referenciadas por *Hardware Transactional Memory* (HTM), permitem a execução especulativa de transações conferindo a tarefa do gerenciamento de contenções ao protocolo de coerência de cache. Para tal, as linhas de cache são estendidas com bits transacionais, responsáveis pela diferenciação das linhas pertencentes as execuções especulativas das demais presentes na cache.

As principais diretrizes adotadas para detecção/resolução de conflitos [45] são elencadas

a seguir:

- A invalidação de uma entrada especulativa resulta no *abort* da transação, visto que a invalidação caracteriza um conflito de sincronização;
- Todas as modificações especulativas são descartadas em caso de *abort* da transação;
- O despejo, *eviction*, de uma entrada especulativa da cache resulta no *abort* da transação, uma vez que a detecção de conflitos é restrita à cache;
- A efetivação de uma transação desativa os bits transacionais das entradas especulativas, externalizando os valores para as demais transações.

Entre as principais vantagens desta implementação, destacam-se o baixo *overhead* de inicialização e *rollback* das transações e o melhor desempenho na execução de transações pequenas e médias. O suporte transacional em hardware, disponibilizado pelo microprocessadores contemporâneos [46, 47, 48], proveem uma implementação baseada na política de melhor esforço (*best-effort*), no qual não há garantia da efetivação das transações mesmo na ausência de conflitos, dependendo de um mecanismo de *fallback* para garantia de progresso do sistema.

A principal dificuldade enfrentada pela HTM é a limitação de recursos, advinda do pouco espaço em cache para manutenção dos conjuntos de leitura e escritas das transações. Neste âmbito, as soluções comumentemente adotadas são, após a realização de algumas tentativas via hardware, o redirecionamento da execução para software (STM), abordagem híbrida, ou a aquisição de um *lock* global, serializando a execução das transações.

A implementação HTM adotada neste trabalho é a *Restricted Transactional Memory* (RTM), disponibilizada pela Intel através da extensão denominada *Transactional Synchronization Extension* (TSX). A RTM disponibiliza uma interface contendo as seguintes instruções: `xbegin`, `xend` e `xabort`. A primeira salva o estado atual da arquitetura e inicia a execução de uma nova transação. A instrução `xbegin` recebe como parâmetro o endereço do caminho de *fallback* para redirecionamento caso não seja possível a efetivação da transação. A segunda instrução realiza o *commit* da transação caso nenhum conflito seja detectado. Por fim, a última instrução permite o *abort* de uma transação de maneira explícita pelo programador.

O Algoritmo 7 apresenta um exemplo de implementação das operações `TxStart` e `TxCommit` utilizando as instruções disponibilizadas pela interface da TSX. O mecanismo de *fallback* utilizado foi a serialização das transações via aquisição de um *lock* global.

```
1:  bool TxStart()
2:    while (true)
3:      if (xbegin() == STARTED)
4:        if isLocked(global_lock)
5:          xabort()
6:      else
7:        retries = retries + 1
8:        // Caminho de fallback
9:        if retries > MAX_RETRIES
10:         lock(global_lock)
11:     return
```

```
1:  bool TxCommit()
2:    if retries > MAX_RETRIES
3:      // O lock foi travado pois
          não foi possível
          concluir a transação em
          hardware
4:    else
5:      // Tentativa de efetivação
          da transação
6:      xend()
7:    retries = 0
```

Algoritmo 7 – Exemplo de implementação das operações `TxStart` e `TxCommit` via instruções de hardware. Extraído de [41].

2.5.3 Implementação em Fases (PhTM)

Memória Transacional em fases, conhecida na literatura por *Phased Transactional Memory* (PhTM), foi originalmente proposta por Lev e colegas [49] visando uma alternativa eficiente às tradicionais implementações híbridas de memória transacional, denominadas *Hybrid Transactional Memory* (HyTM). A HyTM possibilita a execução concorrente de transações via STM e HTM, no entanto requer o uso de mecanismos complexos para resolução de conflitos entre as transações em modos distintos, impactando o desempenho do sistema.

Neste contexto, a PhTM propõe a coordenação da execução de TMs em software e hardware através de fases, isto é, todas as transações em uma mesma fase são executadas em um mesmo modo. O bom desempenho advindo desta abordagem é totalmente atrelado à resolução dos seguintes desafios:

- A identificação do modo mais eficiente de execução para as distintas especificidades apresentadas pelas transações, tais como tempo de execução da transação, tamanho dos conjuntos de leitura e escrita, elevado nível de contenção, entre outras;
- A realização da transição entre os modos de maneira precisa e eficaz, reduzindo possíveis *overheads* de sincronização e impactando minimamente o desempenho da solução;
- A decisão do momento apropriado para a transição entre os modos, possibilitando usufruir as vantagens de cada implementação.

Visando atender aos requisitos definidos acima, a solução PhTM*, desenvolvida por Carvalho e colegas [14], propôs o uso do *feedback* retornado pela execução em HTM para guiar a transição entre os três modos disponibilizados pelo sistema: (i) modo em hardware (HW),

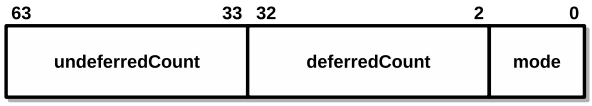


Figura 9 – Indicador de modo. Adaptado de [14].

intitulado *fast-path* em virtude do baixo *overhead* de execução; (ii) modo em software (SW), denominado *slow-path*; e (iii) modo serial (GLOCK).

O PhTM* utiliza uma estrutura global denominada indicador de modo, apresentada na Figura 9, para auxiliar no controle de transição entre os modos. O indicador é composto por três variáveis: (i) `mode`, aponta em qual modo as transações estão operando; (ii) `deferredCount`, enumera a quantidade de transações que solicitaram a transição entre os modos; (iii) `undeferredCount`, enumera a quantidade de transações interrompidas pelo processo de transição, além das demais transações, exceto as deferidas, iniciadas em software posteriormente à transição.

Estas informações foram estruturadas em uma variável de 64 bits, no qual 2 bits são utilizados para indicação do modo em operação, enquanto os demais são divididos igualmente para manutenção dos contadores de transações. Tal organização é justificada pelo fato de que os microprocessadores contemporâneos garantem atomicidade em operações de acesso a no máximo 64 bits, possibilitando que as atualizações realizadas nestas variáveis sejam visíveis uniformemente pelo sistema.

O PhTM* faz uso do indicador de modo, em conjunto das informações retornadas pelo cancelamento das transações, para escolha do modo a ser executado, conforme o autômato apresentado na Figura 10. As transações iniciam sua execução em hardware, *fast-path*, buscando extrair o máximo desempenho. Ao detectar a ocorrência de subsequentes *aborts* por capacidade e a taxa de *aborts* tiver atingido o limiar fixado (`ABORT_THRSD`), caracterizando a existência de transações longas, o sistema realiza a transição para o modo de software, vide Ⓐ. O emprego desta estratégia é resultante da constatação de que transações que obtém subsequentes *aborts* por capacidade nunca serão efetivadas via hardware, visto que as implementações de HTM tipicamente possuem capacidade transacional limitada [50].

A taxa de *aborts* é calculada mediante a fórmula apresentada a seguir, onde são consideradas todas as causas de *aborts* em hardware exceto por capacidade.

$$abort\_rate_t = \alpha \cdot abort\_rate_{t-1} + (1 - \alpha) \cdot abort$$

Esta fórmula foi inspirada no cálculo de intensidade de contenção proposto por Yoo e

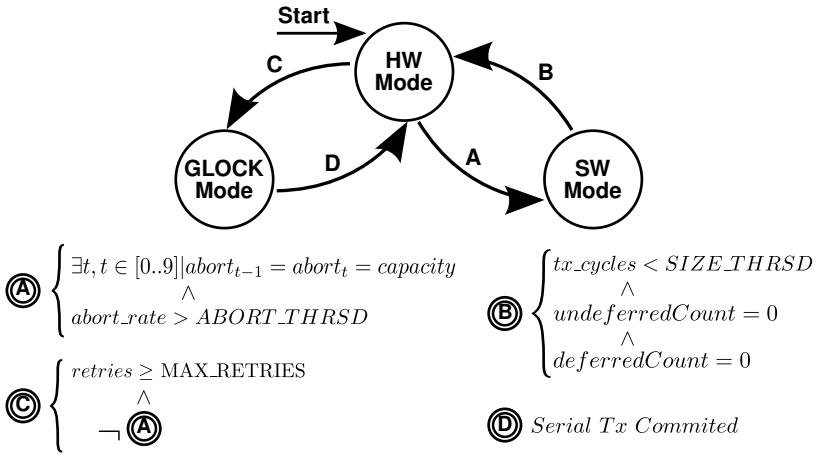


Figura 10 – Autômato de transição de fases do PhTM*. Extraído de [14].

colegas [51], no qual o parâmetro α é responsável por atribuir um peso maior ao histórico de *aborts* ($abort\_rate_{t-1}$) ou ao *abort* corrente ($abort$).

As transações deferidas, solicitantes da transição para SW, atualizam o valor presente em `deferredCount`, consequentemente resultando no *abort* das demais transações em hardware, dado que o indicador de modo é mantido no conjunto de leitura destas. Em seguida, o valor de `mode` é atualizado, refletindo a efetivação da transição HW para SW, possibilitando as transações deferidas iniciarem sua execução imediatamente. Ao término destas, o contador é decrementado até restar somente o valor um.

As transações indeferidas a princípio checam se o sistema se encontra na fase de software e `deferredCount` é diferente de zero, indicando que não houve uma transição de software para hardware. Logo após, a transação incrementa o contador `undeferredCount` e inicia sua execução em software. Por fim, a efetivação das transações indeferidas resulta no decremento de seu respectivo contador.

Caso a checagem retorne que o modo corrente é hardware ou que ambos contadores estão zerados, a transação aguarda transição de software para hardware. Após a execução de N transações, valor definido empiricamente, é calculada uma média de ciclos gastos na execução das transações. Se a média for maior que o limiar definido `SIZE_THRSD`, a fase é mantida em software e o valor de N é dobrado. Caso contrário, o contador `deferredCount` é decrementado, zerando o valor deste, e a transição para hardware é realizada se `undeferredCount` também estiver zerado, vide Ⓑ.

O PhTM* serializa a execução somente quando: (i) as transações ultrapassam o número

de tentativas de execução em hardware (`MAX_RETRIES`); (ii) a causa dos cancelamentos não seja por subsequentes *aborts* por capacidade, vide ©. Como o PhTM* serializa somente transações pequenas, o período de tempo em que o sistema permanece neste modo é relativamente curto, garantindo assim a efetivação da transação e um rápido retorno ao modo de hardware [14], conforme ®. A transição para o modo GLOCK pode falhar somente no caso de outra *thread* ter realizado a transição do sistema para software.

3 Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados mais relevantes no contexto do uso conjunto de TM com NVM. A apresentação destes foi dividida em três seções. As seções 3.1 e 3.2 descrevem as soluções NV-HTM e PSTM, utilizadas na elaboração deste trabalho. Por fim, a Seção 3.3 apresenta distintas soluções, encontradas na literatura, para garantia de consistência em ambientes com memória persistente via uso de transações.

3.1 NV-HTM

A solução NV-HTM [8] provê suporte à execução de transações em memória persistente utilizando a implementação transacional em hardware, disponível em microprocessadores da Intel e IBM, sem modificações arquiteturais. Para tal, o sistema pressupõe que as transações operam exclusivamente em regiões persistentes de memória, mantendo uma semântica simples e intuitiva ao programador [8].

Os principais aspectos empregados pela solução são elencados a seguir:

- As modificações efetivadas pelas transações em hardware são persistidas via software;
- A efetivação durável das transações, realizada em software, é postergada para garantia da ordem de execução definida pela HTM;
- O gerenciamento dos *logs* persistentes é feito por uma rotina de *checkpoint*, responsável por limitar o espaço de memória utilizado.

O uso conjunto de HTM e NVM apresenta um prognóstico de elevado desempenho, no entanto algumas particularidades das implementações de HTM acrescentam certa complexidade na comprovação desta hipótese. A primeira particularidade é a ausência de transmissão automática dos valores efetivados pelas transações para os demais níveis de memória (DRAM e NVM). A HTM apenas garante que as escritas sejam visíveis às demais transações via protocolo de coerência, podendo estas permanecerem arbitrariamente na cache antes de serem persistidas.

Tal comportamento afeta diretamente a propriedade de durabilidade do sistema de memória transacional, visto que não há garantias de que os estados consistentes produzidos pelas efetivações das transações atinjam a persistência, consequentemente inviabilizando a recuperação

da totalidade dos valores correntes em eventuais falhas de sistema ou quedas de energia.

A segunda particularidade é a inibição do uso de *flushes*, para externalização do trabalho especulativo aos demais níveis de memória, durante a execução da transação. A retirada de uma entrada especulativa da cache resulta no *abort* da transação, visto que o mecanismo de detecção de conflitos é restrito à cache, além da necessidade de inserção de uma lógica mais robusta em caso de *rollback* de transações contendo dados externalizados para a memória persistente.

Visando solucionar tais empecilhos, o NV-HTM instrumenta as escritas especulativas realizadas via HTM, adicionando-as em um *redo log*. Ao realizar o *commit*, o trabalho especulativo é exposto às demais transações via protocolo de coerência de cache, caracterizando o *commit* não durável. Após o *commit*, os valores presentes no *redo log* são persistidos via software, mediante a um processo de verificação de dependências entre transações, garantindo a ordem de efetivação definida pela HTM. Ao término da persistência, a transação é concluída, caracterizando o *commit* durável.

Um fato importante a ser observado é que após o término de uma transação apenas seu *log* se encontra persistido, consequentemente o estado da aplicação presente na memória não assegura consistência. O diferencial da solução NV-HTM é o descarte do estado da aplicação durante o processo de *recovery*, sendo este reconstruído através dos *logs* previamente armazenados [8]. Neste cenário, a correta manutenção do *log* e o gerenciamento de seu respectivo tamanho é de primordial valia para a solução. Por conseguinte, o NV-HTM emprega um processo de *checkpoint*, denominado *Backward Filtering Checkpointing* (BFC), responsável por criar um *snapshot* persistente do estado da aplicação, de forma independente das execuções das transações, removendo a persistência das alterações do caminho crítico, e consequentemente melhorando o desempenho da aplicação.

Outro fator que impacta positivamente no desempenho do sistema é o uso de uma técnica de filtragem para construção do *snapshot* persistente. O uso desta técnica é corroborado pela observação de que múltiplos *benchmarks* e aplicações reais tendem a concentrar uma elevada taxa de atualizações, executada por diferentes transações, em uma pequena região de memória (*hot-spot*) [8].

O BFC percorre os *logs* armazenados de maneira inversa a ordem dos *commits*, armazenando em uma tabela *hash* os endereços (*key*) e os respectivos valores (*value*) presentes em cada transação. Ao verificar que uma determinada posição de memória já se encontra na tabela,

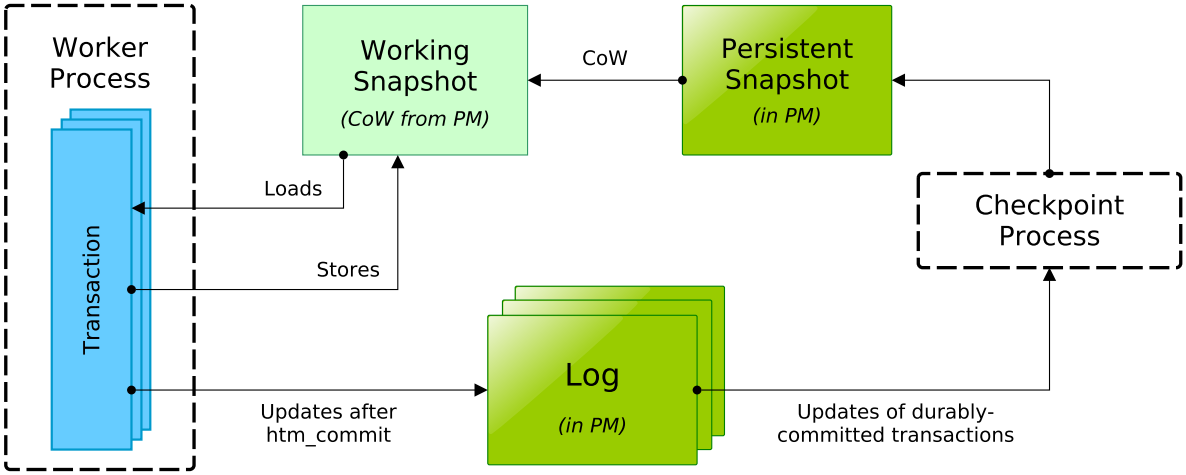


Figura 11 – Arquitetura do Sistema NV-HTM. Extraído de [8].

seu valor é desconsiderado, mantendo assim o valor mais recente produzido pelas transações. Ao término da execução, o *snapshot* persistente reflete um estado consistente e atualizado do sistema, enquanto o *log* apresenta uma baixa taxa de ocupação.

Para melhor organização e fluência do texto, as seguintes seções apresentam a arquitetura do NV-HTM (Seção 3.1.1) e os respectivos detalhes de implementação (Seção 3.1.2).

3.1.1 Arquitetura do Sistema

O NV-HTM é organizado mediante a arquitetura apresentada na Figura 11, sendo constituída basicamente pelos seguintes componentes:

- *Working Snapshot* (WS): estado temporário da aplicação oriundo do mapeamento privado, via *Copy-on-Write* (CoW), do estado persistente. Este tipo de mapeamento cria uma cópia volátil da página e evita que as modificações sejam propagadas para a NVM;
- *Log* durável: estrutura descentralizada composta por vários pequenos *logs*, cada um associado a uma *thread* em execução no sistema. Esta organização permite maximizar a localidade e minimizar os problemas de sincronização [8];
- *Persistent Snapshot* (PS): estado persistente e consistente da aplicação, acessado somente pelo processo de *checkpoint*;
- *Working Process* (WP): processo responsável pela inicialização das múltiplas *threads* de execução e pelo mapeamento das regiões persistentes de memória no espaço de endereçamento do processo;
- *Checkpoint Process* (CP): processo responsável pela contenção do crescimento do *log* e

pela manutenção do estado persistente da aplicação.

As regiões persistentes de memória (*Log* e PS) são expostas às aplicações através de *heaps* persistentes, mapeadas no espaço de endereçamento dos processos através do *Direct Access* (DAX) [52], mecanismo provido pelo *Kernel* do Linux que possibilita o acesso direto a arquivos presentes na NVM eliminando a necessidade de cópia dos mesmos na cache de páginas, realizada pelo sistema operacional e mantida na DRAM.

3.1.2 Implementação

Seguindo o mesmo padrão adotado para explanação do NOrec, a seguir serão apresentados o conjunto de metadados utilizados e a respectiva implementação da API transacional do sistema. Os metadados utilizados pelo NV-HTM, apresentados no Algoritmo 8, podem ser categorizados em dois conjuntos: (i) o conjunto global de metadados, linhas 2 e 3, composto pelos *logs* associados a cada *thread* e por um vetor contendo o *timestamp*, tempo físico medido pelo processador, de cada transação em execução no sistema; (ii) o conjunto local a cada *thread*, constituído do *timestamp* da fase de efetivação da transação e de uma *flag* indicando se a transação é exclusivamente de leitura (*read-only*), linhas 6 e 7.

```
1:  Shared variables:
2:    log[N]                    // Um log por thread, armazenado na NVM
3:    ts[N]                          // Timestamp das transações ativas
4:
5:  Thread local variables:
6:    locTS                     // Timestamp da efetivação da transação
7:    isRO       // Flag usada para identificação de transação read-only
```

Algoritmo 8 – Metadados utilizados pelo NV-HTM. Extraído de [8].

Antes do início de uma transação em hardware, esta é identificada como *read-only* e sua respectiva posição no vetor de transações ativas é inicializada através da leitura do *timestamp*, conforme as linhas 2 e 3 do Algoritmo 9. Para garantia de que as demais *threads* identifiquem o início da nova transação, uma barreira de memória é utilizada, linha 4.

Conforme a Seção 2.5.2, as implementações de HTM necessitam de um mecanismo de *fallback* para garantia de progresso do sistema. Neste contexto, o NV-HTM realiza um número predeterminado de tentativas de execução via hardware antes de adquirir o *lock* global, *Single Global Lock* (SGL), para efetivação da transação.

```
1:  function TxStart()
2:    isRO = true
3:    ts[t] = readTS()
4:    mem_fence()      // Garante que a tx está ativa às demais threads
5:    htm_begin()                // Inicia a execução da tx em hardware
```

Algoritmo 9 – Início de uma transação em hardware. Extraído de [8].

Iniciada a transação, as escritas especulativas realizadas por esta são instrumentadas pela barreira de escrita, apresentada no Algoritmo 10. A primeira etapa realizada é a remoção da identificação de exclusividade de leitura da transação, linha 2, modificando a lógica de *commit*, vide Algoritmo 11, a ser usada para efetivação da transação.

```
1:  function TxWrite(addr, value)
2:    isRO = false
3:    if logCheckSpace(log[t]) == FULL
4:      TxAbort(LOG_FULL)
5:
6:    *addr = value            // Escrita realizada no Working Snapshot
7:    log[t].append(addr, value)
8:
9:  function TxAbort(abort_code)
10:   htm_abort(abort_code)
11:   ts[t] = INACTIVE_MARKER
```

Algoritmo 10 – Barreira de escrita e o procedimento de *abort*. Extraído de [8].

A segunda etapa consiste da checagem da disponibilidade de espaço no *log*, linha 3, tanto para a nova entrada e seu respectivo endereço quanto para a marcação do *commit* e seu respectivo *timestamp*. Caso haja espaço disponível no *log*, a escrita especulativa é realizada no WS e seus respectivos metadados são armazenados no *log* persistente, linhas 6 e 7. Caso contrário, a transação é abortada, resetando seu *timestamp* na lista de transações ativas (vide linhas 9 a 11). A *thread* aguarda o término do CP para reiniciar a execução da transação. O CP inicia sua execução ao ser notificado por uma das *threads* do sistema. Previamente a execução de `TxStart`, a *thread* verifica a taxa de ocupação de seu respectivo *log*. Caso esta seja maior que um limiar fixado, e.g., 50% de ocupação, uma notificação é enviada.

Para minimizar concorrência entre o *checkpoint* e as demais *threads* em execução, os *logs* foram estruturados através de listas circulares, contendo ponteiros que indicam seu início e término. Ao iniciar a execução do *checkpoint*, os ponteiros de término dos *logs* são lidos atomicamente, delimitando a região de análise para criação do PS. Esta abordagem permite o acesso concorrente ao *log* pelo *checkpoint* e pelas demais *threads*, visto que a efetivação das alterações acessa regiões posteriores ao conjunto delimitado para análise.

```
1:  function TxCommit()
2:    if isRO                    // Lógica de commit para txs read-only
3:      htm_commit()
4:      ts[t] = INACTIVE_MARKER
5:      waitCommit()
6:    else                    // Lógica de commit para txs com escritas
7:      locTS = readTS()
8:      htm_commit()
9:      ts[t] = locTS
10:     logFlush(log[t])                          // Persistência do log
11:     waitCommit()
12:     log[t].append(COMMIT_MARKER, locTS)
13:     log[t].endP = locEndP
14:     logFlush(log[t])          // Commit marker e endP persistentes
15:     ts[t] = INACTIVE_MARKER
16:
17: function waitCommit()
18:   for all t* in [1, N] and t* != t
19:     wait until ts[t*] > ts[t]
```

Algoritmo 11 – Efetivação durável e não durável de uma transação. Extraído de [8].

Ao término do trabalho especulativo, a transação entra na fases de efetivação durável e não durável. Neste primeiro momento será descrita a lógica utilizada para o *commit* de transações com escritas, representada pelas linhas 7 a 15 do Algoritmo 11. O primeiro passo a ser executado é a leitura do *timestamp*, indicando o instante de tempo em que a transação finalizou o trabalho realizado dentro da seção crítica.

O segundo passo é a realização do *commit* não durável, efetuado pelo protocolo de coerência de cache, expondo as modificações especulativas para as demais transações em execução no sistema, linha 8. Ao concluir o *commit*, o *timestamp* lido é inserido no vetor de transações ativas, atualizando o valor previamente armazenado durante o início da transação.

Os próximos passos a serem executados são a persistência do *log* e fase de espera, linhas 10 e 11. A fase de espera visa garantir a propriedade de que ao efetivar uma transação T de maneira durável, isto é, ao persistir a marcação de *commit*, as demais transações T* as quais foram serializadas antes da transação T pela HTM já devem estar persistidas. O procedimento de espera, invocado na linha 11, checa o vetor de transações ativas, aguardando que os *timestamps* das demais transações sejam maiores que o *timestamp* lido ao fim da transação, garantindo assim a ordem de execução, vide linhas 17 a 19 do Algoritmo 11.

Concluída a fase de espera, o *commit* durável é realizado através da inserção da marcação de *commit* e do *timestamp*, lido ao fim da transação, no *log* e a respectiva persistência desses valores, linhas 12 a 14. Por fim, o *timestamp* presente na lista de transações ativas é resetado,

indicando às demais transações na fase de espera o fim desta, linha 15.

A lógica utilizada para o *commit* de transações *read-only* é mais simplificada, linhas 2 a 5, visto que as demais transações de escrita não necessitam aguardar o término desta para efetivação. Desta forma, após o *commit* via hardware o *timestamp* é resetado, conforme a linha 4. No entanto, as transações *read-only* ainda necessitam passar pela fase de espera para garantia de que as entradas lidas, possivelmente produzidas por alguma transação de escrita, já se encontrem persistidas, linha 5.

3.2 PSTM

A solução *Persistent Software Transactional Memory* (PSTM) [10, 8], provê suporte à execução de transações em memória persistente utilizando a implementação transacional em software. Para tal, aplica estratégias baseadas no sistema Mnemosyne [3], primeiro sistema de memória transacional a prover transações operando em ambientes com memória não-volátil.

Semelhante a demais implementações [3, 8, 4], o PSTM expõe as regiões de memória persistente às aplicações através de *heaps* persistentes. Desta forma, as aplicações mantêm duas *heaps* mapeadas em seus respectivos espaços de endereçamento: (i) uma para manutenção dos *logs* associados a cada *thread* em execução no sistema, utilizados pelo processo de recuperação para garantia de consistência; (ii) outra para o estado persistente da aplicação.

O trabalho especulativo realizado pelas transações é armazenado em um *redo log*, caracterizando o uso do versionamento diferido. As entradas no *log* são tornadas persistentes previamente à fase de efetivação da transação, reduzindo o tempo empregado na execução desta. Ao iniciar o *commit*, uma marcação de efetivação é inserida no *log*, indicando a completude das informações produzidas presentes no mesmo.

Esta estratégia permite a recuperação do último estado consistente produzido pelo sistema, bastando verificar a presença desta marcação e tornar suas respectivas entradas persistentes. Caso a marcação não seja encontrada, os últimos valores presentes no *log* são descartados. Ao término da persistência do novo estado da aplicação, os ponteiros do *log* são atualizados de forma a reaproveitar o espaço previamente utilizado, adotando assim uma política simplista de gerenciamento do crescimento do *log*. Após esta atualização, a transação é concluída.

3.3 Outros Trabalhos

O uso de Memória Transacional para garantia de consistência, em ambientes com memória persistente, é um tema recorrente e de grande enfoque por parte da academia. Diversos trabalhos têm investigado o uso de diferentes estratégias e implementações de TM, visando reduzir possíveis impactos de desempenho e assegurar a manutenção de corretude em operações persistentes.

Os primeiros trabalhos com este enfoque foram o Mnemosyne [3] e o NV-Heaps [4]. Ambos propuseram expor a NVM às aplicações através do mapeamento de *heaps* persistentes, possibilitando o acesso direto via operações de memória. No entanto, o uso destas operações foram delimitadas a blocos atômicos, incorporando assim a propriedade de *failure atomicity* ao sistema, isto é, a capacidade de assegurar a consistência de dados persistentes após algum evento de falha. O Mnemosyne e o NV-Heaps foram implementados a partir de extensões feitas em sistemas de memória transacional em software. Entre as extensões destacam-se o emprego de um esquema de *logging* e de rotinas de recuperação, para correta manutenção do estado persistente das aplicações.

Abordagens utilizando HTM também foram propostas, porém a grande maioria apresenta modificações arquiteturais a serem feitas para viabilizar seus respectivos usos. O PTM [9] estende as linhas de cache em 8 bits, utilizados para identificação de qual transação estas pertencem, além da adição de uma pequena tabela, no processador, para checagem de dependências entre as transações.

O PHTM [10] propõe a adição de uma nova primitiva em hardware denominada *flush* transparente, possibilitando a execução de *flushes* durante a execução de uma transação sem que esta seja abortada. O PHyTM [11], primeiro sistema híbrido de memória transacional projetado para ambientes com memória persistente, também faz uso das extensões propostas pelo PHTM. O DHTM [12] expande o conjunto de escrita das transações, antes restrita ao nível L1 da cache, permitindo o *overflow* até o último nível de cache, também conhecido como *Last Level Cache* (LLC).

A solução NV-HTM [8] provê suporte à execução de transações, em memória persistente, utilizando HTM sem modificações arquiteturais. Conforme explanado na Seção 3.1, o NV-HTM propõe a manutenção de dois estados da aplicação, um temporário e o outro persistente. Tal abordagem também é utilizada em Kamino-Tx [5] e em Romulus [53], no entanto estes não

Tabela 5 – Características das implementações transacionais em memória persistente.

	Características			
Soluções	**Software**	**Hardware**	**Sem Modificações Arquiteturais**	**Sem Limitações de Capacidade**
Mnemosyne/PSTM	✓		✓	✓
NV-Heaps	✓		✓	✓
PTM		✓		
PHTM		✓		
PHyTM	✓	✓		✓
DHTM		✓		
DudeTM	✓	✓		✓
cc-HTM		✓	✓	
Kamino-Tx	✓		✓	✓
Romulus	✓		✓	✓
NV-HTM		✓	✓	
NV-PhTM	✓	✓	✓	✓

oferecem suporte a HTM.

O NV-HTM também faz uso de um esquema de *checkpoint* para construção do estado persistente da aplicação. Demais soluções também utilizam esta estratégia, tais como o DudeTM [54] e o cc-HTM [55]. No entanto, o primeiro apresenta problemas de escalabilidade em *workloads* com alta taxa de atualização, devido ao uso de um relógio global lógico para serialização das transações em HTM. O segundo requer a instrumentação das leituras, adicionando um *overhead* significativo às execuções [8].

A Tabela 5 classifica as aplicações mediante quatro principais características: (i) suporte a execução em software; (ii) suporte a execução em hardware; (iii) ausência de modificações arquiteturais; (iv) ausência de limitações de capacidade da implementação. Um aspecto a ser destacado é a dependência de modificações arquiteturais para uso de HTM em grande parte das aplicações, inviabilizando uma avaliação real da efetividade destas soluções. As soluções cc-HTM e NV-HTM foram as primeiras a fazer uso do suporte de HTM disponível no processadores sem extensões, no entanto as limitações de capacidade, impostas pelo tamanho da cache, são resolvidas através da serialização do sistema, impactando consideravelmente o desempenho. As soluções em software não apresentam tais limitações, porém os *overheads* introduzidos podem ser um grande limitador de eficiência destas soluções.

A grande desvantagem do NV-HTM é observada na execução de transações com amplos

conjuntos de leitura e escrita. Tal cenário resulta em constantes *aborts* por capacidade, ocasionando a serialização através da obtenção do *lock* global, consequentemente reduzindo o desempenho do sistema. Neste âmbito, o NV-PhTM propõe uma abordagem baseada em fases, visando identificar o modo mais eficiente de execução (SW ou HW) para as distintas especificidades apresentadas pelas transações. Visando equacionar os *trade-offs* entre as distintas implementações, o NV-PhTM provê a execução de ambos os modos (software e hardware) sem a requisição de modificações arquiteturais.

4 Solução Proposta

Este capítulo apresenta a implementação do sistema em fases proposto por este trabalho. Para tal, a Seção 4.1 descreve a motivação e a avaliação experimental preliminar que embasam a concepção do mesmo. Por fim, a Seção 4.2 apresenta a arquitetura, as heurísticas e os detalhes de implementação.

4.1 Problemas com Abordagens Atuais

Grande parte dos sistemas transacionais contemporâneos baseiam sua solução de persistência em abordagens exclusivamente em software ou hardware, como apresentado anteriormente na Tabela 5. Apesar de certa redução na complexidade de gerenciamento das transações, comparados a sistemas em fase e híbridos, estas soluções podem não apresentar um bom desempenho, em virtude de suas limitações, nas mais distintas especificidades transacionais apresentadas pelas aplicações. Para comprovação de tal fato, uma avaliação experimental foi realizada nos sistemas NV-HTM (HW) e PSTM (SW) através da execução da aplicação Vacation, integrante do consolidado *benchmark* STAMP [13]. O ambiente experimental e a respectiva metodologia abordada são detalhados na Seção 5.3.

O primeiro aspecto a ser destacado, ao observar a Figura 12, é a discrepância no tempo de execução entre o PSTM e o NV-HTM com uma e duas *threads*. Esta discrepância é resultante dos seguintes fatores: (i) o grande desempenho oferecido pela HTM, visto que o NV-HTM permanece grande parte do tempo em modo HW; (ii) o *overhead* ocasionado pelas escritas em memória persistente, uma vez que no PSTM cada escrita resulta em uma inserção no *log*, durante a barreira de escrita, e uma inserção no estado persistente durante a fase de *commit*. A vantagem do NV-HTM reside no processo de *checkpoint*, o qual executa a etapa de persistência do estado da aplicação de maneira simultânea às transações, consequentemente reduzindo o tempo empregado na efetivação das mesmas.

A medida que o número de *threads* aumenta, no PSTM, há uma redução no impacto negativo causado pelas escritas persistentes. Este comportamento é justificado pela distribuição das operações de escrita entre as diversas *threads*, possibilitando a sobreposição destas e consequentemente reduzindo consideravelmente o tempo empregado por *thread* para as respectivas

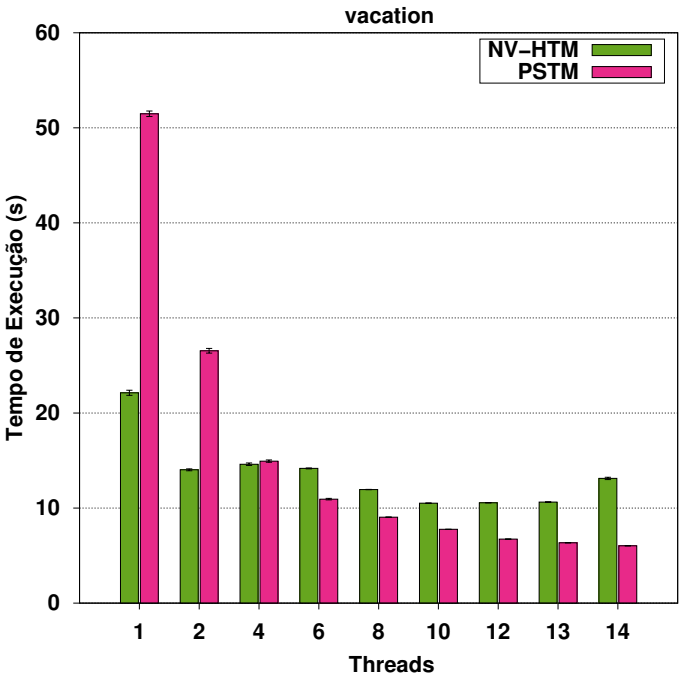


Figura 12 – Tempo de execução da aplicação Vacation nos sistemas PSTM e NV-HTM.

efetivações na memória.

O segundo aspecto a ser notado é a elevada taxa de *aborts* por capacidade apresentada pelo NV-HTM. Este tipo de cancelamento é decorrente da incapacidade do hardware concluir a execução das transações, devido a limitações de recursos disponíveis para manutenção dos conjuntos de leitura e escrita. Logo, o sistema recorre constantemente a aquisição do *lock* global para garantia de progresso, impactando significativamente o desempenho do mesmo.

4.2 NV-PhTM

Objetivando identificar o melhor modo de execução para as distintas especificidades transacionais, a solução *Non-Volatile Phased Transactional Memory* (NV-PhTM), proposta nesta dissertação, utiliza o *feedback* retornado pelas transações em conjunto com heurísticas para guiar a transição entre as distintas implementações transacionais (HW e SW).

A implementação do NV-PhTM foi fundamentada nos conceitos apresentados pelo PhTM*, no entanto certas modificações foram propostas: (i) a adição do conceito de durabilidade, visto que este foi concebido em ambientes de memória volátil; (ii) a criação de estratégias para garantia de consistência durante a transição entre os modos do sistema; (iii) a extensão do conjunto inicial de heurísticas, visando solucionar potenciais problemas encontrados nas

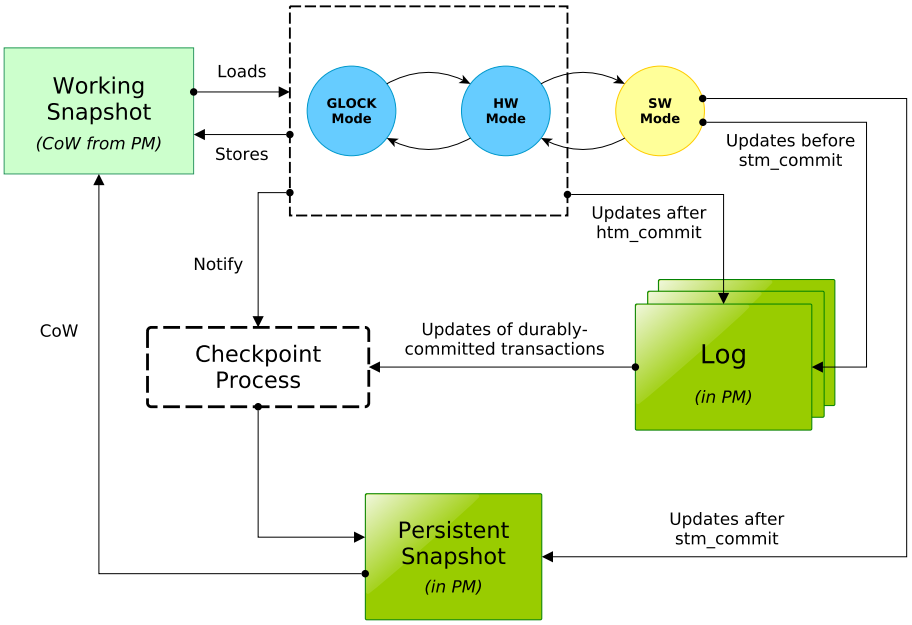


Figura 13 – Arquitetura do Sistema NV-PhTM.

soluções em HTM disponíveis para NVM.

Para melhor organização e fluência do texto, as seguintes seções apresentam a arquitetura do NV-PhTM (Seção 4.2.1), as estratégias empregadas para consolidação das transições (Seção 4.2.2) e o conjunto de heurísticas propostas para regular a transição entre os modos (Seção 4.2.3).

4.2.1 Arquitetura do Sistema

A primeira mudança realizada pelo NV-PhTM foi a substituição das implementações em HW e SW, RTM e NOrec previamente, por soluções que ofereçam suporte a persistência. Para o modo em HW foi selecionado o NV-HTM em virtude de não requisitar modificações arquiteturais para sua execução. A provisão do modo em SW é realizada através do PSTM.

A integração entre estas implementações distintas, e a respectiva arquitetura do sistema, podem ser observadas na Figura 13. O NV-PhTM apresenta os seguintes componentes:

- *Working Snapshot* (WS): estado temporário da aplicação, mantido em memória volátil, com a finalidade de evitar a propagação das escritas especulativas para a NVM nos modos HW e serial;
- *Log* durável: conjunto de *logs* associados às *threads* em execução. Os *logs* são partilhados entre os diferentes modos do sistema;
- *Persistent Snapshot* (PS): estado persistente e consistente da aplicação, atualizado pelo

processo de *checkpoint*, nos modos HW e serial, e pela fase de efetivação das transações em SW;

- *Working Process* (WP): processo responsável pela inicialização do sistema e pela execução das transações;
- *Checkpoint Process* (CP): processo responsável pela contenção do crescimento do *log* e pela manutenção do estado persistente da aplicação nos modos HW e serial;

A arquitetura projetada para o NV-PhTM visa manter a modularidade proposta pelo PhTM*, o qual possibilita a substituição dos módulos em SW e HW por outras implementações capazes de prover o mesmo conjunto de operações definidas em cada API transacional. Entretanto, as limitações apresentadas pelas soluções de HTM, em ambientes de memória persistente, inviabilizaram a modularização da implementação em hardware, ficando a encargo do NV-HTM a provisão deste modo. A modularidade do NV-PhTM se restringe ao modo SW, possibilitando a avaliação de distintas implementações de forma a obter melhor desempenho.

A efetividade da modularização reside na independência de execução entre os modos. Neste âmbito, a arquitetura foi projetada considerando os seguintes aspectos: (i) o compartilhamento das regiões persistentes entre os modos HW e SW; (ii) a consolidação do estado persistente da aplicação antes da conclusão da transição; (iii) a efetividade das heurísticas propostas, visando reduzir a quantidade de transições realizadas pelo sistema.

As regiões persistentes de memória destinadas ao estado da aplicação e aos *logs* são partilhadas entre as duas implementações, isto é, ambas atualizam estas regiões porém em fases distintas do sistema. Tal organização possibilita o reúso eficiente de memória, reduzindo o *footprint* utilizado pelo sistema para garantia de persistência. Entretanto, certas precauções devem ser tomadas durante a transição de HW para SW, e vice-versa, a fim de evitar possíveis condições de corrida, advindas das distintas estratégias utilizadas para modificação do estado persistente da aplicação. Enquanto o NV-HTM delega esta tarefa ao *checkpoint*, o PSTM aplica as atualizações durante a fase de efetivação das transações. Deste modo, antes da realização da transição, é necessário que o estado persistente da aplicação, produzido pelo modo corrente, reflita o último estado consistente, garantindo a unicidade de execução de cada modo.

4.2.2 Estratégias de Consolidação

Uma etapa de consolidação do estado persistente foi incorporada às transições entre as distintas implementações (NV-HTM e PSTM). Na transição de HW para SW, a *thread* solicitante de tal ação incrementa o contador de transações deferidas, presente no indicador de modo, resultando no cancelamento das demais transações em execução no sistema. Em seguida, a etapa de consolidação é iniciada através da invocação do processo de *checkpoint*, o qual efetua o procedimento de drenagem dos *logs*, aplicando a estratégia de filtragem em todas as entradas presentes nos mesmos e persistindo o último estado produzido pelo modo corrente. Durante a execução do *checkpoint*, as *threads* permanecem aguardando em uma barreira, evitando condições de corrida entre as transações e o *checkpoint*. Ao término da drenagem, o espaço em *log* é liberado e o *snapshot* persistente da aplicação se encontra consistente, possibilitando o prosseguimento das execuções em SW. Eventuais interrupções ocorridas na fase de drenagem não afetam a corretude dos dados, uma vez que a liberação de espaço nos *logs* é o último procedimento a ser realizado pela etapa de consolidação. Desta forma, o sistema é capaz de prover resiliência, retornando ao estado prévio à falha através de procedimentos de *recovery*.

A etapa de consolidação de SW para HW é efetuada através da inserção de uma barreira, responsável por aguardar o término das transações correntes em SW. Uma vez que todas se encontram efetivadas, o *snapshot* persistente reflete o último estado consistente produzido pelo sistema e o espaço em *log* se encontra liberado. Finalizando esta etapa, o sistema realiza um novo mapeamento do estado temporário da aplicação, retratando as alterações realizadas em SW.

Para melhor elucidar os conceitos envolvidos na solução proposta, o seu respectivo funcionamento é detalhado a seguir. Inicialmente o NV-PhTM mapeia os *logs* e o estado persistente da aplicação (PS) no espaço de endereçamento do processo. Em seguida, o sistema verifica a existência de entradas nos *logs*, fruto de uma possível execução prévia interrompida, e aplica um processo de recuperação (*recovery*) destas informações, possibilitando a retomada do último estado consistente produzido pela aplicação. Ao término da fase de *recovery*, cria-se uma cópia volátil do estado da aplicação (WS), para evitar a propagação do trabalho transacional, realizado pelos modos HW e GLOCK, para a NVM. As transações então iniciam suas execuções em modo HW, utilizando os mesmos conceitos de *commits* duráveis e não duráveis definidos na Seção 3.1.

Durante a fase de efetivação das transações, as respectivas entradas de seus *logs* são

persistidas, possibilitando a recuperação do sistema em eventuais falhas ou quedas de energia. Previamente ao início de novas transações, há uma verificação da taxa de ocupação do *log*. Caso este tenha atingido o limiar de 50%, o sistema envia uma notificação ao processo de *checkpoint* requisitando a liberação de espaço nos *logs*, e consequentemente a atualização do estado persistente da aplicação.

Quando o sistema detecta um baixo desempenho no modo HW, o *feedback* retornado pelos *aborts* das transações é analisado para determinar a transição para o modo mais eficiente. Caso o modo GLOCK seja identificado como o mais promissor, o sistema transita para este mantendo as mesmas estratégias empregadas pelo modo HW, tais como as escritas no estado temporário e o uso do *checkpoint*. Ao efetivar a transação em modo serial, o sistema retorna para o modo HW.

Caso o modo em SW seja o mais atrativo, a etapa de consolidação é iniciada através do envio de uma notificação ao processo de *checkpoint* para realização da drenagem dos *logs*. As *threads* permanecem em modo de espera até que o último estado consistente, produzido em HW, se encontre persistente e que o *logs* se encontrem vazios. O esvaziamento do *log* é justificado pelas distintas estratégias adotadas, pelo NV-HTM e pelo PSTM, para atualização do mesmo. Enquanto o NV-HTM utiliza o *checkpoint* para controle da taxa de ocupação do *log*, o PSTM reseta o ponteiro que indica o final do mesmo ao término de cada transação, possibilitando o reaproveitamento do espaço previamente utilizado.

Concluída a etapa de consolidação, as *threads* podem prosseguir suas execuções em modo SW. As transações efetuadas neste modo armazenam o trabalho especulativo em seus respectivos *logs* persistentes previamente a fase de efetivação. No início do *commit*, uma marcação de efetivação é inserida no *log*, indicando a completude das informações produzidas pelo mesmo. Em seguida, as modificações transacionais são persistidas, gerando o novo estado consistente da aplicação. Por fim, os ponteiros do *log* associado a *thread* em execução são atualizados, liberando o espaço no mesmo.

Quando o sistema detecta um baixo desempenho no modo SW, e etapa de consolidação ativa a barreira para garantia do término das transações correntes. Concluídas as transações, o sistema realiza o mapeamento do novo estado temporário da aplicação (WS) refletindo o último estado consistente produzido em software. Em seguida, as *threads* iniciam suas execuções em hardware.

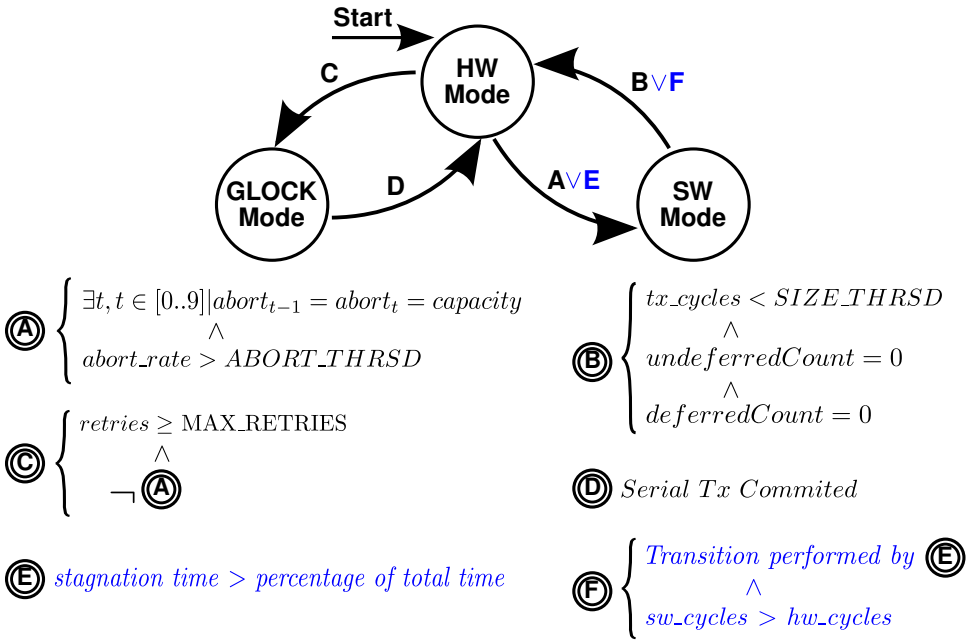


Figura 14 – Autômato de transição de fases do NV-PhTM.

4.2.3 Heurísticas de Transição

De maneira análoga ao PhTM*, a escolha da implementação a ser executada é determinada pelo uso conjunto do indicador de modo e de uma coleção de heurísticas, propostas através da análise do *feedback* retornado pela execução das transações. O curso do sistema pode ser observado através do autômato apresentado na Figura 14.

O autômato de transição de fases do NV-PhTM estende a versão proposta pelo PhTM*, detalhada na Seção 2.5.3, através da adição de duas novas transições, Ⓔ e Ⓕ, para resolução da estagnação do sistema. A heurística definida por Ⓔ contempla o cenário no qual as transações em execução no sistema apresentam as seguintes características: (i) poucas instruções, caracterizando transações curtas; (ii) baixo nível de contenção; (iii) alta frequência de execução. Esta conjuntura culmina em um rápido crescimento dos *logs*, resultando em frequentes chamadas ao processo de *checkpoint*. A elevada taxa de ocupação dos *logs* amplia a quantidade de trabalho desempenhado pelo processo de filtragem do *checkpoint*, consequentemente impactando negativamente o tempo de execução do mesmo.

O alto índice de efetivação das transações, aliado à lentidão proveniente do *checkpoint*, aumentam a quantidade de *aborts* explícitos oriundos da falta de espaço em *log*, conforme o Algoritmo 10 da Seção 3.1.2. Uma vez executado o cancelamento explícito, a *thread* permanece aguardando o término do processo de *checkpoint* para liberação de espaço no *log*, e por conseguinte

retomar sua execução. Este cenário se torna crítico a medida que várias *threads* permanecem estagnadas simultaneamente por um período arbitrário de tempo, resultando em uma degradação de desempenho do sistema.

Objetivando solucionar tal adversidade, a heurística definida em Ⓔ propõe a transição para software quando o sistema apresenta um tempo de estagnação superior a um percentual do tempo total de execução, definido empiricamente. O cálculo do tempo é realizado após a execução de um número N de transações no modo HW, permitindo averiguar a frequência das estagnações e a respectiva influência no desempenho do sistema.

A heurística definida por Ⓕ realiza o retorno para o modo em hardware quando as seguintes condições são atendidas: (i) a transição prévia do sistema para SW foi realizada através de Ⓔ; (ii) a média de ciclos gastos na execução das transações em software é maior que a média de ciclos empregue na execução em hardware, contabilizando a quantidade de ciclos desperdiçadas pelas estagnações.

5 Avaliação Experimental

Este capítulo apresenta a avaliação experimental do sistema proposto. Para tal, a Seção 5.1 descreve o *benchmark Stanford Transactional Applications for Multi-Proccessing* (STAMP) [13], conjunto de aplicações transacionais utilizado para validação do sistema. Na Seção 5.2 são apresentados os sistemais transacionais utilizados. A Seção 5.3 detalha as configurações do ambiente experimental e a metodologia empregada na condução dos experimentos. Por fim, a Seção 5.4 apresenta os resultados obtidos.

5.1 STAMP

O *benchmark* STAMP é um conjunto de aplicações científicas desenvolvidas para avaliação das distintas implementações de TM, buscando abranger as mais diferentes especificidades que influenciam o desempenho de sistemas transacionais. A Tabela 6 apresenta as aplicações e suas respectivas características: (i) tamanho das transações dado pelo número de instruções; (ii) tamanho dos conjuntos de leitura e escrita; (iii) tempo gasto na execução das transações; (iv) nível de contenção da aplicação.

Tabela 6 – Caracterização qualitativa das aplicações do STAMP. Adaptado de [13].

Aplicação	**Tamanho**	**Conjunto de Leitura/Escrita**	**Tempo**	**Contenção**
Genome	Médio	Médio	Elevado	Baixo
Intruder	Pequeno	Médio	Médio	Alto
Kmeans	Pequeno	Pequeno	Baixo	Baixo
Labyrinth	Grande	Grande	Elevado	Alto
SSCA2	Pequeno	Pequeno	Baixo	Baixo
Vacation	Médio	Médio	Elevado	Baixo/Médio
Yada	Grande	Grande	Elevado	Médio

O STAMP foi paralelizado através do estilo *Single Program Multiple Data* (SPMD), onde as *threads* executam a mesma sequência de instruções em dados distintos, tornando o código mais regular e homogêneo. Neste contexto, a aplicação Bayes não foi considerada na avaliação

deste trabalho devido a natureza estocástica do algoritmo, resultando em uma grande variação de tempo de execução [14].

5.2 Sistemas Transacionais

O trabalho objetiva validar a hipótese de que sistemas em fase proveem desempenho superior a sistemas atuais que empregam NVM. Para tal, os seguintes sistemas transacionais foram avaliados:

- **PSTM:** descrito na Seção 3.2, foi originalmente implementado através de extensões adicionadas a biblioteca TinySTM [56, 57], sistema transacional que utiliza a tabela de registro de posse para detecção/resolução de conflitos. No entanto, visando extrair o máximo desempenho do sistema, o PSTM foi reimplementado utilizando o NOrec como base. A rotina de persistência das atualizações, previamente realizada na etapa final do *commit*, foi postergada para uma fase pós efetivação, reduzindo o impacto causado pela demora na liberação do *lock* global.
- **NV-HTM:** descrito na Seção 3.1, compartilha parte do código do *fast-path* da implementação PhTM* em conjunto das rotinas de operações no *log* persistente e no *checkpoint*, desenvolvidas por Castro et al. [8]. Tal fato visa manter uma comparação mais justa, eliminando possíveis discrepâncias nas medições de tempo ocasionadas pela diferença de implementação do sistema transacional.
- **NV-PhTM:** descrito na Seção 4.2, foi implementado utilizando o PhTM* como base, estendendo este através da substituição do NOrec pelo PSTM no modo SW, além da adição dos *logs* persistentes e do processo de *checkpoint*, conforme a implementação do NV-HTM. Desta forma, é possível validar a hipótese de que o sistema em fases provê desempenho superior a sistemas atuais em HW ou SW.

5.3 Ambiente Experimental e Metodologia

Os experimentos foram conduzidos em uma máquina com as seguintes especificações: (i) Sistema Operacional Linux CentOS 7, *Kernel* versão 3.10; (ii) 64GB de memória RAM; (iii) Processador Intel Xeon E5-2660 v4 2.0GHz com 14 *cores* físicos e 28 *threads* em hardware

Tabela 7 – Configurações dos sistemas NV-HTM e NV-PhTM.

Sistema	Tamanho de cada Log	Tentativas em HW	Taxa de Aborts	Limiar de Ciclos	Limiar de Estagnação
PSTM	10KB	-	-	-	-
NV-HTM	10KB	9	-	-	-
NV-PhTM	10KB	9	80%	30K	45%

através de *hyper-threading*. A capacidade do conjunto de escrita é limitada pelo tamanho da cache L1 de dados (32KB) e a granularidade da detecção de conflitos é no nível de linha de cache (64 bytes). O recurso de *hyper-threading* não foi empregado na condução dos experimentos em razão da queda de desempenho oriunda do elevado número de *aborts* por capacidade [14].

As aplicações e bibliotecas utilizadas foram compiladas usando o GCC 7.2.1 (Agosto/2017), utilizando a API transacional disponibilizada pela RTM. Objetivando evitar problemas de desempenho induzidos pelo alocador de memória [58, 59], a versão modificada do alocador TCMalloc [14] foi utilizada.

Devido a ausência de uma versão consolidada das tecnologias de NVM, as escritas em regiões de memória persistente são simuladas através da inclusão de uma latência em cada operação de *flush*. Tal comportamento também permite simular a instrução *Cache Line Write Back* (CLWB), recentemente adicionada no ISA da Intel, responsável por escrever os valores presentes em uma linha de cache na DRAM sem retirar esta da cache. Desta forma, as leituras de valores persistentes, previamente produzidos pelas transações, usufruem de uma latência de acesso diminuta.

As configurações dos sistemas transacionais são apresentadas na Tabela 7. Quando o NV-PhTM se encontra no modo software, a média do tamanho das transações, em ciclos, é calculada a cada 100 execuções. Caso o sistema permaneça em modo SW, este valor é dobrado podendo atingir o total de 1000 execuções. Estas configurações foram escolhidas com base na análise de desempenho conduzida no *benchmark* STAMP [14]. A taxa de *aborts* explícitos é calculada a cada 1000 execuções em hardware.

Os parâmetros utilizados na execução das aplicações do STAMP, apresentados na Tabela 8, contemplam os *workloads* com amplos conjuntos de leitura/escrita e elevados níveis de contenção. Os resultados obtidos representam a média de 30 execuções com intervalo de confiança de 95%.

Tabela 8 – Aplicações do STAMP e seus respectivos parâmetros e descrições. Adaptado de [41].

Aplicação	Parâmetros	Descrição
Genome	`-g16384 -s64 -n16777216`	sequenciamento de genes
Intruder	`-a10 -l128 -n262144 -s1`	detecção de ataques em redes
Kmeans	`-m15 -n15 -t0.00001 -irandom-n65536-d32-c16`	algoritmo de clusterização
Labyrinth	`-i random-x512-y512-z7-n512`	algoritmo de roteamento
SSCA2	`-s20 -i1.0 -u1.0 -l3 -p3`	conjunto de operações em grafos
Vacation	`-n4 -q60 -u90 -r1048576 -t4194304`	sistema de reserva de passagens
Yada	`-a15 -ittimeu1000000.2`	algoritmo para refinamento de malhas pelo critério de Delaunay

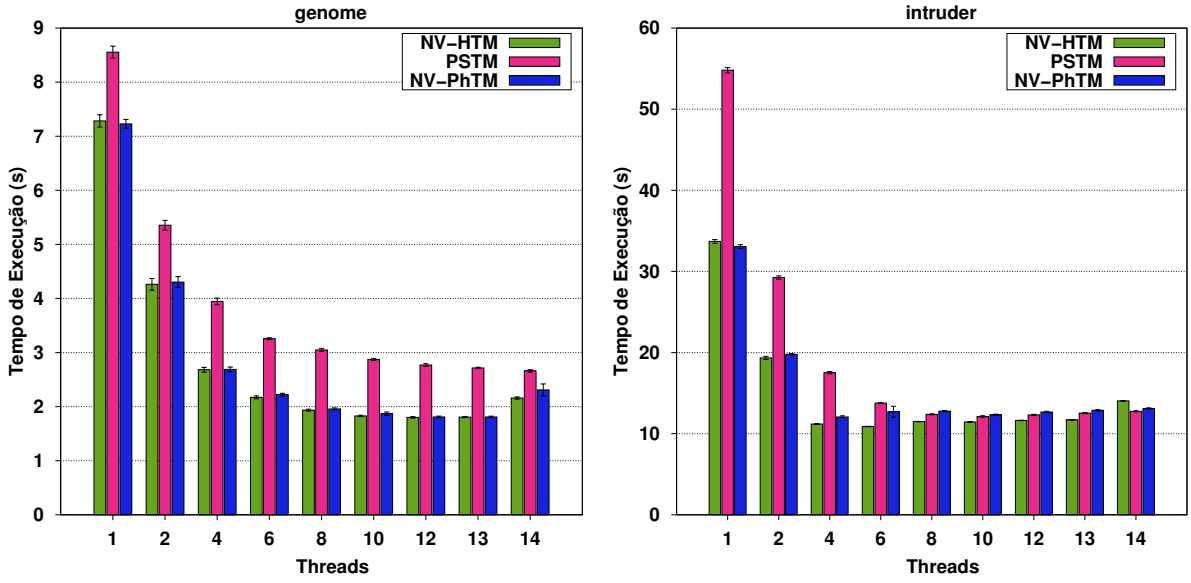
5.4 Resultados

Os resultados experimentais obtidos foram examinados seguindo três distintas análises. A primeira análise, apresentada na Seção 5.4.1, avalia o desempenho dos sistemas através da medição de seus respectivos tempos de execução. Em seguida, na Seção 5.4.2, as novas heurísticas, propostas por este trabalho, são analisadas mediante a sua efetividade em detectar o melhor modo de execução. Por fim, a Seção 5.4.3 apresenta uma análise do impacto de desempenho oriundo das técnicas de garantia de consistência, adicionadas no processo de transição entre os modos.

5.4.1 Análise de Desempenho

A análise de desempenho realizada nesta seção é subdividida em três vertentes: (i) aplicações com proeminência de execução em hardware, isto é, apresentam características que propiciam um desempenho superior ao serem executadas em uma implementação em hardware; (ii) aplicações com proeminência em software; (iii) aplicações que apresentam um comportamento em fases, ou seja, fazem uso efetivo das heurísticas de transição entre os modos para obter melhor desempenho.

Para melhor organização do texto, a Seção 5.4.1.1 apresenta a análise de proeminência em hardware, a Seção 5.4.1.2 destaca a superioridade em software e, finalizando, a Seção 5.4.1.3 apresenta o ganho proporcionado pelo sistema de fases.



(a) Genome (b) Intruder

Figura 15 – Tempo de execução das aplicações com melhor desempenho em hardware: (a) Genome e (b) Intruder.

5.4.1.1 Proeminência em Hardware

A Figura 15 exibe o tempo de execução (eixo y) obtido pelas aplicações Genome e Intruder nos três sistemas previamente citados. A primeira constatação a ser feita é a enorme discrepância apresentada pelo PSTM com um número reduzido de *threads*, vide Figura 15b. Tal fato é decorrente do *overhead* ocasionado pelas operações de escrita em memória persistente, realizadas durante a barreira de escrita, para manutenção do *log*, e na etapa de efetivação da transação, para consolidar o estado da aplicação de maneira durável. A medida que o número de *threads* aumenta, a carga de trabalho é distribuída, possibilitando a sobreposição das operações de escrita, e consequentemente uma redução no tempo de espera de cada *thread* para efetivação destas na memória. Os elevados custos de escrita são mitigados no NV-HTM através do processo de *checkpoint*, responsável por aplicar a filtragem das entradas, reduzindo a quantidade de operações de escrita, e por persistir o estado da aplicação de maneira simultânea a execução das transações.

A aplicação Genome não manifesta tamanha discrepância de tempo de execução entre as implementações em virtude da elevada taxa de *aborts* por capacidade, conforme pode ser constatado na Tabela 9, a qual força constantes serializações do NV-HTM para garantia de progresso. Todavia, a execução serializada ainda é mais rápida que o modo em software. O sistema NV-PhTM, ao constatar a alta incidência de cancelamentos por capacidade, transita para o modo software visando evitar subsequentes serializações, vide Ⓐ na Figura 14. Entretanto,

Tabela 9 – Estatísticas de execução do Genome no sistema NV-PhTM.

Threads	**% de tempo**			**Taxa de aborts**	**Aborts por conflito**	**Aborts por capacidade**	**Aborts explícitos**
	HW	**SW**	**GL**				
1	92.66	0.82	6.52	8.87	0.26	96.98	0.08
2	90.73	1.20	08.08	8.24	0.69	94.41	0.29
4	88.21	0.82	10.98	7.23	1.46	89.76	1.22
6	86.28	0.77	12.95	7.38	1.85	86.18	1.54
8	84.68	0.77	14.55	7.36	2.27	83.14	1.98
10	83.56	1.45	15.00	7.43	2.41	77.98	2.32
12	83.15	0.68	16.17	7.89	2.52	76.5	2.36
13	82.93	0.82	16.25	8.03	2.71	73.17	2.40
14	59.58	27.72	12.69	6.25	11.56	65.2	2.75

ao verificar que a média de ciclos, despendida na execução das transações em SW, é superior à média obtida nos modos HW e GLOCK, transita novamente para hardware, permanecendo neste uma quantia arbitrária de tempo. Desta forma, o NV-PhTM detecta o modo HW como mais promissor e permanece executando neste modo, conforme a porcentagem de tempo apresentada na Tabela 9.

Na aplicação Intruder, o NV-PhTM se mantém em hardware nas execuções com 1, 2 e 4 *threads*. Contudo, a partir de 6 *threads* há um aumento na taxa de *aborts* explícitos, proveniente da máxima ocupação do *log*, suscitando a transição para o modo SW, vide Ⓔ na Figura 14. Tal comportamento advém de uma especificidade apresentada pela aplicação, inicialmente as transações no Intruder são maiores e frequentes, possibilitando o sistema transitar para o modo SW utilizando a nova heurística proposta. Porém, a medida que a aplicação avança sua execução, as transações reduzem a quantidade de instruções presentes nas regiões críticas, consequentemente sendo efetivadas de maneira mais ágil. A nova métrica mantém a última média de ciclos de execução lida em HW antes da transição, por conseguinte ao realizar a comparação das médias em SW e HW, conforme a heurística Ⓕ na Figura 14, o sistema se mantém em modo SW, não obtendo o melhor desempenho.

A execução com 14 *threads* apresenta uma particularidade a qual possibilita o software ser mais efetivo. A implementação em hardware executa o processo de *checkpoint* simultaneamente às demais *threads* da aplicação, empregando o recurso de *hyper-threading* do processador. Logo, há uma redução de recursos disponíveis para execução das transações, resultando em constantes *aborts* por capacidade e frequentes serializações do sistema, culminando no *slowdown* da aplicação.

5.4.1.2 Proeminência em Software

Entre as aplicações presentes no *benchmark* STAMP, apenas o Labyrinth apresenta uma proeminência completa de execução em software. Conforme a Tabela 6 presente na Seção 5.1, este comportamento é justificado pela presença de transações longas e com amplos conjuntos de leitura e escrita, os maiores encontrados no *benchmark*. Esta conjuntura impossibilita a conclusão das transações em hardware no NV-HTM, ocasionando uma taxa de 90% de aborts, grande parte por capacidade. Logo, o sistema frequentemente requisita o mecanismo de *fallback* para garantia de progresso.

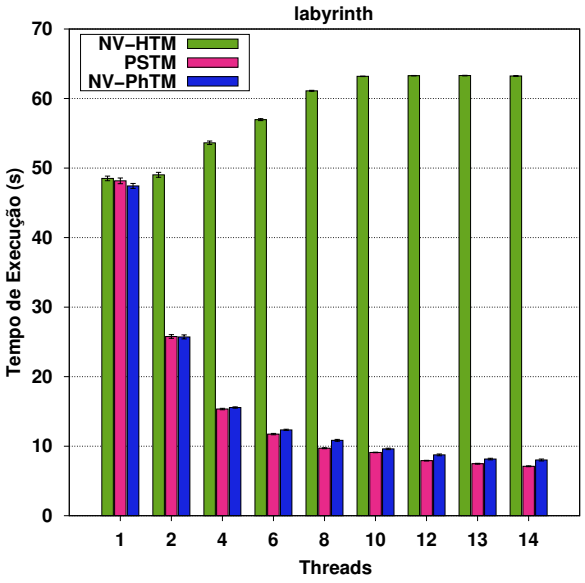
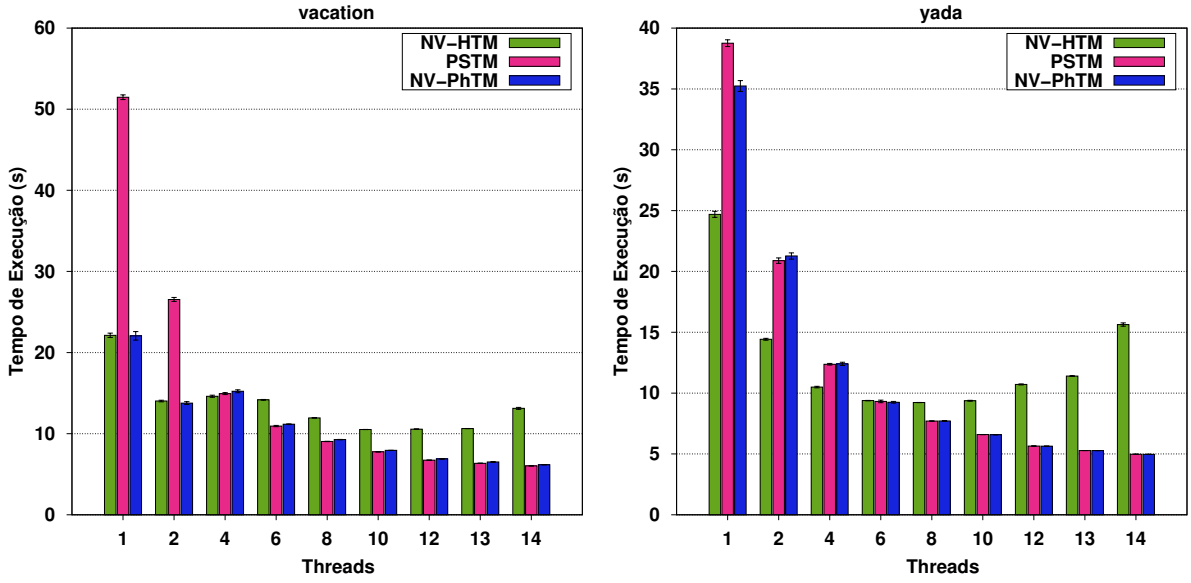


Figura 16 – Tempo de execução da aplicação Labyrinth.

O NV-PhTM, via heurística definida em Ⓐ na Figura 14, é capaz de detectar a incidência de transações longas na aplicação, guiando o sistema para execução em modo SW. A Figura 16 demonstra o êxito do NV-PhTM em direcionar o sistema de forma a acompanhar a melhor abordagem de execução no presente cenário.

5.4.1.3 Proeminência em Fases

Grande parte das aplicações do *benchmark* STAMP se beneficiam da execução em fases por dois principais aspectos: (i) elevado número de cancelamentos por capacidade, à medida que o número de *threads* aumenta; (ii) estagnação gerada pela máxima ocupação dos *logs*. A Figura 17 ilustra o primeiro cenário citado.



(a) Vacation (b) Yada

Figura 17 – Tempo de execução das aplicações: (a) Vacation e (b) Yada.

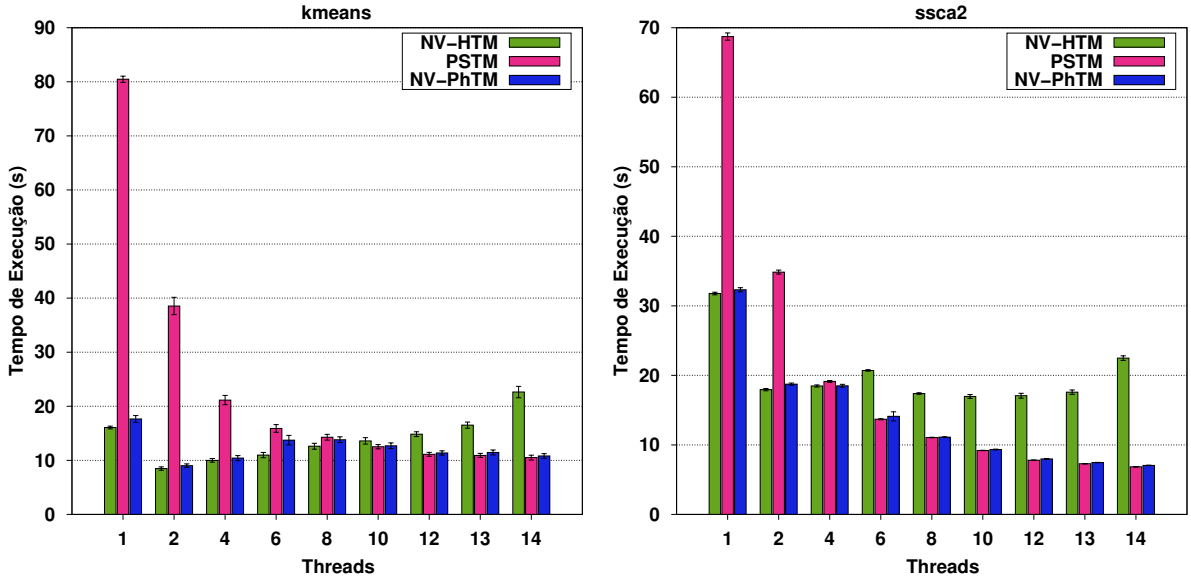
As aplicações Vacation e Yada apresentam um bom desempenho em hardware nas execuções com 1, 2 e 4 *threads*, em virtude do alto *overhead* das escritas persistentes presenciado pelo PSTM. No entanto, a partir de 6 *threads* as constantes serializações da implementação em hardware sobrepujam o tempo de execução em SW, conduzindo o NV-HTM a um fraco desempenho. O NV-PhTM detecta tal mudança e passa a acompanhar a execução em SW a partir de 6 *threads*. Uma particularidade apresentada pela aplicação Yada impede a execução do NV-PhTM em hardware em um número reduzido de *threads*. A alta taxa de *aborts* da aplicação, em torno de 65%, originados em grande parte por capacidade, forçam o sistema a transitar constantemente para o modo SW.

As aplicações Kmeans e SSCA2 se encaixam no cenário descrito pela transição Ⓔ na Seção 4.2.3, no qual as transações executadas possuem as seguintes características: poucas instruções, elevada frequência de execução e baixa contenção. Esta conjuntura culmina na rápida ocupação dos *logs* persistentes, requisitando constantes execuções do *checkpoint*. Devido a alta demanda de análise do processo de filtragem, o *checkpoint* acaba estendendo seu período de atuação, por conseguinte retardando a liberação de espaço nos *logs*. Logo, as transações que atingem a máxima ocupações destes, permanecem estagnadas no aguardo do término do *checkpoint*. Este problema se torna crítico à medida que múltiplas *threads* permanecem em espera simultaneamente, reduzindo a efetividade operacional da solução.

A aplicação Kmeans, cujas estatísticas de execução no sistema NV-HTM são detalhadas

Tabela 10 – Estatísticas de execução do Kmeans no sistema NV-HTM.

Threads	**% de tempo**		**Taxa de aborts**	**Aborts por conflito**	**Aborts por capacidade**	**Aborts explícitos**
	HW	**GL**				
1	99.97	0.03	7.37	97.98	1.46	0.29
2	99.93	0.07	10.48	97.99	0.70	1.09
4	99.75	0.26	16.5	72.81	0.49	26.61
6	98.76	1.24	19.68	77.36	0.59	21.96
8	90.68	9.32	21.72	79.80	0.69	19.44
10	98.04	1.96	15.40	69.96	0.34	29.57
12	97.97	4.03	16.40	71.63	0.66	27.58
13	82.08	17.92	21.19	79.19	0.46	20.28
14	63.41	36.59	13.39	62.85	1.94	35.02



(a) Kmeans (b) SSCA2

Figura 18 – Tempo de execução das aplicações: (a) Kmeans e (b) SSCA2.

na Tabela 10, permanece majoritariamente no modo HW, apesar de uma considerável taxa de cancelamentos, em sua maioria por conflito. No entanto, há um fato a ser destacado, o aumento da taxa de *aborts* explícitos, resultante exclusivamente do cancelamento provocado pela ausência de espaço nos *logs*. A partir de 4 *threads*, o sistema sofre uma degradação de eficiência proveniente das frequentes estagnações, conforme pode ser constatado na Figura 18a, resultando em subsequentes *slowdowns* à medida que o número de *threads* em execução aumenta.

De maneira análoga ao Kmeans, a aplicação SSCA2 permanece a totalidade de sua execução em modo HW, no entanto os efeitos causados pela estagnação são mais nítidos, conforme pode ser constatado na Tabela 11. A taxa de *aborts* apresentada pelo sistema é ínfima, atingindo um valor máximo de 0.78% com 8 *threads*, contudo a grande maioria dos cancelamentos é

Tabela 11 – Estatísticas de execução do SSCA2 no sistema NV-HTM.

Threads	**% de tempo**		**Taxa de aborts**	**Aborts por conflito**	**Aborts por capacidade**	**Aborts explícitos**
	HW	**GL**				
1	100.0	0.0	0.28	83.0	12.61	0.03
2	100.0	0.0	0.15	81.13	2.96	2.52
4	100.0	0.0	0.50	19.91	1.23	76.88
6	100.0	0.0	0.65	19.92	0.58	78.05
8	100.0	0.0	0.78	20.37	1.79	76.41
10	100.0	0.0	0.77	20.07	0.32	78.13
12	100.0	0.0	0.76	19.67	0.30	78.50
13	100.0	0.0	0.76	19.49	0.26	78.63
14	100.0	0.0	0.72	15.11	0.72	82.43

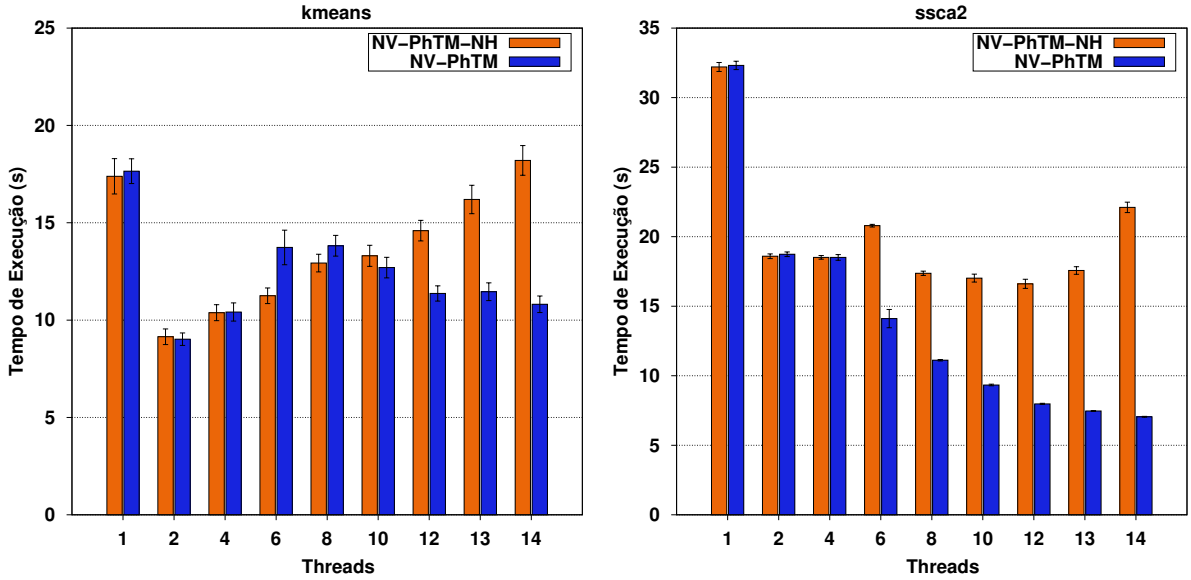
oriunda de *aborts* explícitos.

A aplicação SSCA2, de maneira divergente ao Kmeans, não possui código não transacional entre as execuções dos blocos atômicos, consequentemente há uma maior quantidade de efetivações proporcionadas pelo sistema. Devido ao baixo nível de contenção da aplicação, os *logs* são preenchidos rapidamente e de maneira simultânea, ocasionando a estagnação de múltiplas *threads* de forma sincrônica. Na Figura 18b é possível observar o impacto negativo deste fenômeno. A partir de 4 *threads*, o NV-HTM apresenta uma grande perda de desempenho, mantendo uma constância de tempo de execução de 8 a 13 *threads*. A nova heurística Ⓔ é capaz de detectar a elevada incidência de cancelamentos explícitos e transitar para o modo SW, possibilitando usufruir do melhor desempenho proporcionado pelo PSTM.

5.4.2 Análise de Efetividade das Novas Heurísticas

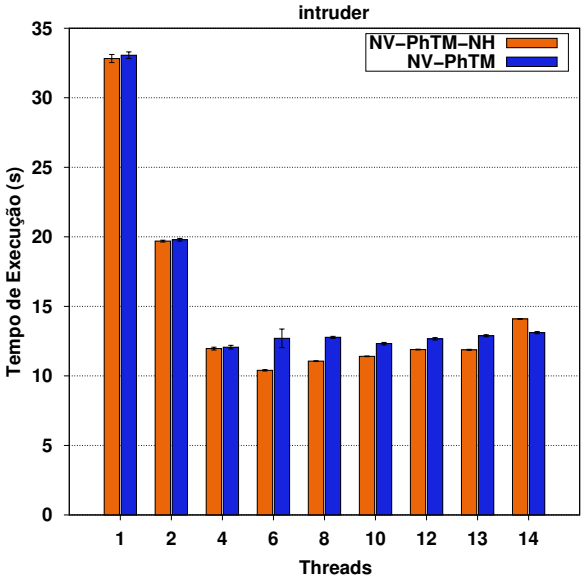
A análise de efetividade foi conduzida através da comparação dos tempos de execução obtidos pelo NV-PhTM utilizando dois conjuntos de heurísticas: (i) a versão inicial definida pelo PhTM*, apresentado na Figura 10 da Seção 2.5.3; (ii) a versão estendida, contemplando as novas heurísticas propostas por este trabalho. As Figuras 19a e 19b ilustram o ganho de desempenho obtido, pelas aplicações Kmeans e SSCA2, através do uso das novas heurísticas. As métricas simplificadas, representadas no gráfico pela versão NV-PhTM-NH, não são capazes de detectar as estagnações oriundas da máxima ocupação dos *logs*, consequentemente acabam mantendo o sistema executando no modo HW, resultando em um baixo desempenho.

Entre as aplicações do *benchmark* STAMP, somente o Intruder obteve um desempenho inferior ao utilizar as novas heurísticas, conforme pode ser constatado na Figura 19c. A variação



(a) Kmeans

(b) SSCA2



(c) Intruder

Figura 19 – Comparação de efetividade das novas heurísticas nas seguintes aplicações: (a) Kmeans, (b) SSCA2 e (c) Intruder.

de tamanho das transações, inicialmente longas porém com uma redução gradativa a medida que a aplicação avança sua execução, não é contemplada pelas novas heurísticas. O sistema não é capaz de retornar para HW pois a média de ciclos, calculada antes da transição para SW, reflete os valores obtidos pelas transações longas. Portanto, ao realizar a verificação proposta pela métrica Ⓕ, o sistema acaba permanecendo em SW mesmo obtendo desempenho inferior.

Tabela 12 – Porcentagem do tempo de execução utilizado pelas rotinas de consolidação durante as transições entre os modos HW e SW.

Aplicações	**Transição**	**Threads**								
		1	2	4	6	8	10	12	13	14
Genome	hw->sw	0.0	0.0	0.02	0.04	0.02	0.02	0.02	0.02	0.03
	sw->hw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01
Intruder	hw->sw	0.0	0.03	0.12	0.10	0.01	0.0	0.0	0.0	0.0
	sw->hw	0.0	0.01	0.01	0.01	0.0	0.0	0.0	0.0	0.0
Kmeans	hw->sw	0.0	0.0	0.0	0.01	0.02	0.03	0.03	0.03	0.02
	sw->hw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.01
SSCA2	hw->sw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	sw->hw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Labyrinth	hw->sw	0.0	0.0	0.16	0.81	0.79	1.05	1.60	2.07	1.73
	sw->hw	0.0	0.0	0.0	0.01	0.08	0.08	0.19	0.14	0.22
Vacation	hw->sw	0.0	0.01	0.01	0.03	0.01	0.01	0.02	0.02	0.01
	sw->hw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Yada	hw->sw	0.01	0.07	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	sw->hw	0.0	0.03	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5.4.3 Análise do Impacto das Transições

A Tabela 12 apresenta a porcentagem de tempo de execução gasto pelo processo de consolidação. A primeira observação a ser feita é o baixo *overhead* adicionado por estas rotinas, chegando a um percentual máximo de 2.07% do tempo de execução na aplicação Labyrinth. Apesar do reduzido número de transições apresentado por esta aplicação, visto que o sistema detecta rapidamente o modo SW como mais promissor e se mantém neste até a conclusão de sua execução, esta apresenta os maiores percentuais de tempo demandado pelas etapas de consolidação. Tal comportamento é proveniente de suas longas transações, as quais preenchem a quase totalidade dos *logs*, aumentando a carga de trabalho efetuada no processo de drenagem.

As aplicações Yada e Intruder apresentam uma maior quantidade de transições quando executadas com poucas *threads*, porém à medida que este número aumenta, o sistema tende a realizar apenas uma transição para software e permanecer neste modo até a conclusão das aplicações. O Kmeans apresenta um comportamento oposto: a medida que o número de *threads* aumenta, o sistema detecta o *overhead* introduzido pelas estagnações e transita de forma mais frequente para o modo SW.

O Genome e o Vacation também apresentam um percentual maior de tempo a medida que as *threads* aumentam, no entanto tal comportamento é oriundo da incidência de cancelamentos por capacidade. No Genome, o sistema transita para o modo SW, visando obter melhor desempenho.

Entretanto, a implementação em software é inferior ao modo serial do sistema, por consequência o sistema retorna para o modo HW para prosseguir a execução. No Vacation, a partir de seis *threads* o NV-PhTM passa a executar em modo SW, devido ao baixo desempenho em HW, mantendo uma certa estabilidade na quantidade de transições realizadas.

Por fim, a aplicação SSCA2 é a que apresenta os menores percentuais de tempo empreendidos nas consolidações. A elevada taxa de *aborts* explícitos, originada pelo máxima ocupação dos *logs*, faz com que o sistema seja guiado a executar em modo SW, através das novas heurísticas propostas, permanecendo neste até a conclusão de sua execução.

Os resultados experimentais utilizando o STAMP mostram que o custo extra causado pela adição do processo de consolidação do estado persistente durante as transições é baixo, não impactando no desempenho das aplicações.

6 Conclusão

Neste trabalho foi apresentado o potencial desempenho proporcionado pelas emergentes tecnologias de memória persistente. A análise feita revelou que a maior exposição da memória para as aplicações permite eliminar vários *overheads* e maximizar o desempenho das aplicações. Porém, é necessário a utilização de estratégias para garantia de consistência dos dados em eventuais falhas de sistemas ou quedas de energia. Entre as estratégias foi destacado o uso de Memória Transacional em suas distintas implementações, software e hardware. Mais especificamente, foram analisadas as principais virtudes e limitações apresentadas pelas diversas soluções presentes na literatura, culminando na constatação de que os sistemas atuais não contemplam o amplo espectro de características das aplicações, por consequência obtendo um desempenho subótimo.

Como alternativa às soluções existentes, este trabalho apresenta uma implementação transacional baseada em fases (NV-PhTM), a qual utiliza o *feedback* retornado pelas transações em conjunto com heurísticas para guiar a transição entre as distintas implementações transacionais. As principais contribuições deste trabalho são: (i) a adição do conceito de durabilidade em sistemas baseados em fase; (ii) a concepção de novas heurísticas que contemplam as especificidades apresentadas pela NVM e pelas soluções integrantes do sistema; (iii) a elaboração de estratégias para garantia de consistência durante a transição entre os distintos modos.

A avaliação experimental realizada comprova o êxito do NV-PhTM em detectar e acompanhar a melhor abordagem de execução nos mais distintos cenários apresentados. As novas heurísticas foram determinantes para evitar o processo de estagnação, originado pela ausência de espaço nos *logs* e pela demora na execução do processo de *checkpoint*. Por fim, as estratégias empregadas para consolidação das transições introduziram um *overhead* diminuto, impactando minimamente o desempenho do sistema.

6.1 Trabalhos Futuros

Apesar dos bons resultados obtidos, ainda assim pode-se destacar uma possível melhoria a ser feita: a elaboração de novas métricas que contemplem os cenários adversos apresentados pelas aplicações Intruder e Yada. Em ambos os casos o sistema transita para software, permanecendo neste modo independentemente do melhor desempenho proporcionado pelo hardware. Neste

contexto, a inserção de transições esporádicas para o modo HW poderia contribuir na atualização de estatísticas, tais como a média de ciclos empregadas pelas transações, utilizadas para detecção da melhor implementação a ser executada.

Considerando a modularidade oferecida pelo NV-PhTM, diversas propostas de trabalhos futuros podem ser elencadas:

- A avaliação de distintas implementações transacionais em software, encontradas na literatura, visando encontrar a solução que melhor se adequa às especificidades apresentadas pela implementação em hardware (NV-HTM), e também pela etapa de consolidação, realizada durante as transições para garantia de consistência;
- A implementação de uma abordagem de STM baseada nos mesmos conceitos apresentados pelo NV-HTM (*checkpoint*, *snapshot* persistente, *snapshot* temporário, ...). A inserção deste novo módulo em SW no sistema possibilita a exclusão do processo de drenagem dos *logs*, visto que ambas implementações (HW e SW) utilizam as mesmas estratégias para garantia de consistência do estado persistente da aplicação. Esta nova conjuntura apresenta duas possíveis configurações: (i) o uso compartilhado do *log* pelos distintos modos, possibilitando o reuso de memória; (ii) a disponibilização de uma região persistente distinta de memória para manutenção do *log* da implementação em SW, proporcionando um certo isolamento dos metadados das soluções;
- A ampliação do conjunto de aplicações utilizadas para avaliação do sistema, possibilitando uma melhor validação e uma possível expansão do conjunto de heurísticas previamente definido.

Referências

1 BADAM, A. How persistent memory will change software systems. *Computer*, IEEE, v. 46, n. 8, p. 45–51, 2013.

2 HARRIS, T.; LARUS, J.; RAJWAR, R. *Transactional Memory*. 2. ed. [S.l.]: Morgan & Claypool Publishers, 2010.

3 VOLOS, H.; TACK, A. J.; SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2011. (ASPLOS XVI), p. 91–104.

4 COBURN, J.; CAULFIELD, A. M.; AKEL, A.; GRUPP, L. M.; GUPTA, R. K.; JHALA, R.; SWANSON, S. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2011. (ASPLOS XVI), p. 105–118.

5 MEMARIPOUR, A.; BADAM, A.; PHANISHAYEE, A.; ZHOU, Y.; ALAGAPPAN, R.; STRAUSS, K.; SWANSON, S. Atomic in-place updates for non-volatile main memories with kamino-tx. In: *Proceedings of the Twelfth European Conference on Computer Systems*. New York, NY, USA: ACM, 2017. (EuroSys '17), p. 499–512.

6 TEAM, T. N. L. *Pmem.io: Persistent Memory Programming*. 2014. Disponível em: <http://pmem.io/>. Acessado em: 09 de jan. 2019.

7 DENNY, J. E.; LEE, S.; VETTER, J. S. Nvl-c: Static analysis techniques for efficient, correct programming of non-volatile main memory systems. In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. New York, NY, USA: ACM, 2016. (HPDC '16), p. 125–136.

8 CASTRO, D.; ROMANO, P.; BARRETO, J. Hardware transactional memory meets memory persistency. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Vancouver, BC, Canada: IEEE, 2018. p. 368–377.

9 WANG, Z.; YI, H.; LIU, R.; DONG, M.; CHEN, H. Persistent transactional memory. *IEEE Computer Architecture Letters*, IEEE, v. 14, n. 1, p. 58–61, Jan 2015.

10 AVNI, H.; LEVY, E.; MENDELSON, A. Hardware transactions in nonvolatile memory. In: *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363*. Berlin, Heidelberg: Springer-Verlag, 2015. (DISC 2015), p. 617–630.

11 AVNI, H.; BROWN, T. Persistent hybrid transactional memory for databases. *Proc. VLDB Endow.*, VLDB Endowment, v. 10, n. 4, p. 409–420, nov. 2016.

12 JOSHI, A.; NAGARAJAN, V.; CINTRA, M.; VIGLAS, S. Dhtm: Durable hardware transactional memory. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. Los Angeles, CA, USA: IEEE, 2018. p. 452–465.

13 MINH, C. C.; CHUNG, J.; KOZYRAKIS, C.; OLUKOTUN, K. Stamp: Stanford transactional applications for multi-processing. In: *2008 IEEE International Symposium on Workload Characterization*. Seattle, WA, USA: IEEE, 2008. p. 35–46.

14 CARVALHO, J. P. D.; ARAUJO, G.; BALDASSIN, A. The case for phase-based transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, 2018.

15 HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

16 MOORE, G. E. Progress in digital integrated electronics. In: *Electron Devices Meeting*. Santa Clara, California: IEEE, 1975. v. 21, p. 11–13.

17 BURKS, A. W.; GOLDSTINE, H. H.; NEUMANN, J. V. Preliminary discussion of the logical design of an electronic computing instrument. In: *The Origins of Digital Computers*. Berlin Heidelberg: Springer, 1982. p. 399–413.

18 ÅKERMAN, J. Toward a universal memory. *Science*, American Association for the Advancement of Science, v. 308, n. 5721, p. 508–510, 2005.

19 YANG, J.; WEI, Q.; CHEN, C.; WANG, C.; YONG, K. L.; HE, B. Nv-tree: Reducing consistency cost for nvm-based single level systems. In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2015. (FAST'15), p. 167–181.

20 ZHANG, Z.; FENG, D.; CHEN, J.; YU, Y.; ZENG, J. The design and implementation of a lightweight management framework for non-volatile memory. In: *2016 IEEE Trustcom/BigDataSE/ISPA*. Tianjin, China: IEEE, 2016. p. 1551–1558.

21 WULF, W. A.; MCKEE, S. A. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 23, n. 1, p. 20–24, mar. 1995.

22 PAPAMARCOS, M. S.; PATEL, J. H. A low-overhead coherence solution for multiprocessors with private cache memories. In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 1984. (ISCA '84), p. 348–354.

23 BAER, J.-L. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. 1. ed. New York, NY, USA: Cambridge University Press, 2009.

24 JACOB, B.; NG, S.; WANG, D. *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

25 OUKID, I.; LEHNER, W. Data structure engineering for byte-addressable non-volatile memory. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. New York, NY, USA: ACM, 2017. (SIGMOD '17), p. 1759–1764.

26 MUTLU, O.; SUBRAMANIAN, L. Research problems and opportunities in memory systems. *Supercomputing frontiers and innovations*, South Ural State University, v. 1, n. 3, p. 19, 2014.

27 KüLTüRSAY, E.; KANDEMIR, M.; SIVASUBRAMANIAM, A.; MUTLU, O. Evaluating stt-ram as an energy-efficient main memory alternative. In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Austin, TX, USA: IEEE, 2013. p. 256–267.

28 LEE, B. C.; IPEK, E.; MUTLU, O.; BURGER, D. Architecting phase change memory as a scalable dram alternative. In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2009. (ISCA '09), p. 2–13.

29 DHIMAN, G.; AYOUB, R.; ROSING, T. Pdram: A hybrid pram and dram main memory system. In: *2009 46th ACM/IEEE Design Automation Conference*. San Francisco, CA, USA: IEEE, 2009. p. 664–669.

30 YOON, H. Row buffer locality aware caching policies for hybrid memories. In: *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD 2012)*. Washington, DC, USA: IEEE Computer Society, 2012. (ICCD '12), p. 337–344.

31 WONG, H.-S. P.; RAOUX, S.; KIM, S.; LIANG, J.; REIFENBERG, J. P.; RAJENDRAN, B.; ASHEGHI, M.; GOODSON, K. E. Phase change memory. *Proceedings of the IEEE*, IEEE, v. 98, n. 12, p. 2201–2227, 2010.

32 CHEN, E.; APALKOV, D.; DIAO, Z.; DRISKILL-SMITH, A.; DRUIST, D.; LOTTIS, D.; NIKITIN, V.; TANG, X.; WATTS, S.; WANG, S.; WOLF, S. A.; GHOSH, A. W.; LU, J. W.; POON, S. J.; STAN, M.; BUTLER, W. H.; GUPTA, S.; MEWES, C. K. A.; MEWES, T.; VISSCHER, P. B. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics*, IEEE, v. 46, n. 6, p. 1873–1878, 2010.

33 CHANG, T.-C.; CHANG, K.-C.; TSAI, T.-M.; CHU, T.-J.; SZE, S. M. Resistance random access memory. *Materials Today*, v. 19, n. 5, p. 254 – 264, 2016.

34 POSITION, S. T. *NVM Programming Model (NPM) Version 1.2*. 2017. Disponível em: <https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf>. Acessado em: 09 de jan. 2019.

35 VENKATARAMAN, S.; TOLIA, N.; RANGANATHAN, P.; CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In: *Proceedings of the 9th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2011. (FAST'11), p. 61–75.

36 CHAKRABARTI, D. R.; BOEHM, H.-J.; BHANDARI, K. Atlas: Leveraging locks for non-volatile memory consistency. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. New York, NY, USA: ACM, 2014. (OOPSLA '14), p. 433–452.

37 IZRAELEVITZ, J.; KELLY, T.; KOLLI, A. Failure-atomic persistent memory updates via justdo logging. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2016. (ASPLOS '16), p. 427–442.

38 HSU, T. C.-H.; BRüGNER, H.; ROY, I.; KEETON, K.; EUGSTER, P. Nvthreads: Practical persistence for multi-threaded applications. In: *Proceedings of the Twelfth European Conference on Computer Systems*. New York, NY, USA: ACM, 2017. (EuroSys '17), p. 468–482.

39 ESWARAN, K. P.; GRAY, J. N.; LORIE, R. A.; TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM*, ACM, New York, NY, USA, v. 19, n. 11, p. 624–633, nov. 1976.

40 BALDASSIN, A. Tudo o que você sempre quis saber sobre memória transacional mas tinha medo de perguntar. In: MARTINS, C. A. P. da S. (Ed.). *Sistemas Computacionais*. Porto Alegre: Sociedade Brasileira de Computação, 2014.

41 CARVALHO, J. P. L. de. *PhTM*: uma implementação eficiente de transações em fases*. Tese (Mestrado em Ciência da Computação) - Instituto de Biociências, Letras e Ciências Exatas, Universidade Estadual Paulista. Rio Claro, São Paulo. 2016.

42 HERLIHY, M.; MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 1993. (ISCA '93), p. 289–300.

43 SHAVIT, N.; TOUITOU, D. Software transactional memory. In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 1995. (PODC '95), p. 204–213.

44 DALESSANDRO, L.; SPEAR, M. F.; SCOTT, M. L. Norec: Streamlining stm by abolishing ownership records. In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2010. (PPoPP '10), p. 67–78.

45 HERLIHY, M.; SHAVIT, N. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

46 INTEL CORPORATION. *Intel® Architecture Instruction Set Extensions Programming Reference*. [S.l.], 2012.

47 JACOBI, C.; SLEGEL, T.; GREINER, D. Transactional memory architecture and implementation for IBM system z. In: *MICRO12*. Vancouver, BC, Canada: IEEE, 2012. p. 25–36.

48 JIANG, J.; PHILIPPEN, P.; KNOBLOCH, M.; MOHR, B. Performance measurement and analysis of transactional memory and speculative execution on IBM blue Gene/Q. In: *EUROPAR14*. Porto, Portugal: Springer, 2014. p. 26–37.

49 LEV, Y.; MOIR, M.; NUSSBAUM, D. PhTM: Phased Transactional Memory. In: *Workshop on Transactional Computing (Transact)*. Portland, Oregon, USA: [s.n.], 2007.

50 GOEL, B.; TITOS-GIL, R.; NEGI, A.; MCKEE, S. A.; STENSTROM, P. Performance and energy analysis of the restricted transactional memory implementation on haswell. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. Phoenix, AZ, USA: IEEE, 2014. p. 615–624.

51 YOO, R. M.; LEE, H.-H. S. Adaptive transaction scheduling for transactional memory systems. In: *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2008. (SPAA '08), p. 169–178.

52 KERNEL LINUX DOCUMENTATION. *DAX - Direct Access for Files*. Disponível em: <https://www.kernel.org/doc/Documentation/filesystems/dax.txt>. Acessado em: 09 de jan. 2019.

53 CORREIA, A.; FELBER, P.; RAMALHETE, P. Romulus: Efficient algorithms for persistent transactional memory. In: *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2018. (SPAA '18), p. 271–282.

54 LIU, M.; ZHANG, M.; CHEN, K.; QIAN, X.; WU, Y.; ZHENG, W.; REN, J. Dudetm: Building durable transactions with decoupling for persistent memory. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2017. (ASPLOS '17), p. 329–343.

55 GILES, E.; DOSHI, K.; VARMAN, P. Continuous checkpointing of htm transactions in nvm. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. New York, NY, USA: ACM, 2017. (ISMM 2017), p. 70–81.

56 FELBER, P.; FETZER, C.; RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In: *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2008. p. 237–246.

57 FELBER, P.; FETZER, C.; MARLIER, P.; RIEGEL, T. Time-based software transactional memory. *IEEE Transactions on Parallel & Distributed Systems*, IEEE Computer Society, Los Alamitos, CA, USA, v. 21, p. 1793–1807, 2010.

58 BALDASSIN, A.; BORIN, E.; ARAUJO, G. Performance implications of dynamic memory allocators on transactional memory systems. In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2015. (PPoPP 2015), p. 87–96.

59 DICE, D.; HARRIS, T.; KOGAN, A.; LEV, Y. The influence of malloc placement on TSX hardware transactional memory. *CoRR*, abs/1504.04640, 2015.

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

TERMO DE REPRODUÇÃO XEROGRÁFICA

Autorizo a reprodução xerográfica do presente Trabalho de Conclusão, na íntegra ou em partes, para fins de pesquisa.

São José do Rio Preto, 11/03/2019

Rafael Pizzirani Murari

Assinatura do autor