UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

Dissertação

LTMS - Lups Transactional Memory Scheduler. Um escalonador NUMA-Aware para STM.

Michael Alexandre Costa

Pelotas, 2021

Michael Alexandre Costa

LTMS - Lups Transactional Memory Scheduler. Um escalonador NUMA-Aware para STM.

Dissertação apresentada ao Programa de Pós-Graduação em Computação do Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. André Du Bois

Pelotas, 2021

Insira AQUI a ficha catalográfica
(solicite em http://sisbi.ufpel.edu.br/?p=reqFicha)

Dedico. . .

AGRADECIMENTOS

Agradeço. . .

Só sei que nada sei.
— SÓCRATES

RESUMO

COSTA, Michael Alexandre. **LTMS - Lups Transactional Memory Scheduler. Um escalonador NUMA-Aware para STM.**. Orientador: André Du Bois. 2021. 42 f. Dissertação (Mestrado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2021.

...

Palavras-chave: Memórias Transacionais - TM. Non-Uniform Memory Access - NUMA. Escalonador.

ABSTRACT

COSTA, Michael Alexandre. **Transaction Scheduler for NUMA Architectures**. Advisor: André Du Bois. 2021. 42 f. Dissertation (Masters in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2021.

...

Keywords: Transactional Memory - TM. Non-Uniform Memory Access - NUMA. Scheduler.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

TM	Memórias Transacionais
STM	Memórias Transacionais em Software
NUMA	Non-Uniform Memory Access
UMA	Uniform Memory Access

SUMÁRIO

1 INTRODUÇÃO

As arquitetura paralelas estão presentes em praticamente todas plataformas computacionais modernas. Processadores com múltiplos núcleos são usados para construção de computadores domésticos e super computadores. O paralelismo desses processadores tendem a crescer, e o aumento de desempenho dos computadores atuais tem se baseado no desenvolvimento de arquiteturas paralelas.

Para os programas extraírem o máximo de desempenho das arquiteturas paralelas, o código deve explorar todo poder computacional oferecido pelas unidades de processamento, porem, a programação paralela está longe de ser uma atividade fácil.

1.1 Motivação

... (**?**).

1.2 Objetivos

... 1.

1.2.1 Objetivo geral

...

1.2.2 Objetivos especificos

- ...; e
- ...

1.3 Estrutura do Texto

...

Tabela 1 – Nome da Tabela

Blabla	Blabla	Blablabla
Bla	Blabla	*Bla blabla blablabla blabla blablabla blabla blablabla.*
Bla	Blabla	*Bla blabla blablabla blabla blablabla blabla blablabla.*
Bla	Blabla	*Bla blabla blablabla blabla blablabla blabla blablabla.*
Bla	Blabla	*Bla blabla blablabla blabla blablabla blabla blablabla.*
Bla	Blabla	*Bla blabla blablabla blabla blablabla blabla blablabla.*
Bla	Blabla	*Bla blabla blablabla blabla blablabla blabla blablabla.*

2 MEMÓRIAS TRANSACIONAIS

Memória Transacional, ou *Transactional Memory* (TM), é uma classe de mecanismos de sincronização que fornece uma execução atômica e isolada de alterações em um conjunto de dados compartilhados. Estas estão sendo desenvolvidas para que no futuro tornem-se o principal meio de fazer a sincronização em um programa concorrente, substituindo a sincronização baseada em *locks* (MORESHET; BAHAR; HERLIHY, 2006). As TMs podem ser implementadas em *software* (STM), em *hardware* (HTM) ou ainda em uma versão híbrida de *hardware* e *software*.

Na programação utilizando STMs, todo o acesso à memória compartilhada é realizado dentro de transações e todas as transações são executadas atomicamente em relação a transações concorrentes.

A principal vantagem na programação usando STM é que o programador apenas delimita as seções criticas e não é necessário preocupar-se com a aquisição e liberação de *locks*. Os *locks*, quando utilizados de forma incorreta, podem levar a problemas como *deadlocks* (BANDEIRA, 2010).

2.1 Propriedades

Transação é uma sequência finita de escritas e leituras na memória executada por uma *thread* (HERLIHY; ELIOT; MOSS, 1993), e deve satisfazer três propriedades:

- **Atomicidade**: cada transação faz uma sequência de mudanças provisórias na memória compartilhada. Quando a transação é concluída, pode ocorrer um *commit*, tornando suas mudanças visíveis a outras *threads* instantaneamente, ou pode ocorrer um *abort*, fazendo com que suas alterações sejam descartadas;

- **Consistência**: as transações devem garantir que um sistema consistente deve ser mantido consistente. Esta propriedade esta relacionada com o conceito de invariância;

- **Isolamento**: as transações não interferem nas execuções de outras transações, assim parecendo que elas são executadas serialmente. Uma transação não

observa o estado intermediário de outra.

Para garantir estas propriedades as TMs utilizam de mecanismos como o de Versionamento de Dados e Detecções de Conflitos. Estes mecanismos são utilizados pelas transações para garantir a execução das TMs.

2.2 Versionamento de Dados

O versionamento de dados faz é responsável pelo gerenciamento das versões dos dados. Ele armazena tanto o valor do dado no início de uma transação como também o valor do dado modificado durante a transação, isso para garantir a propriedade de atomicidade (BALDASSIN, 2009).

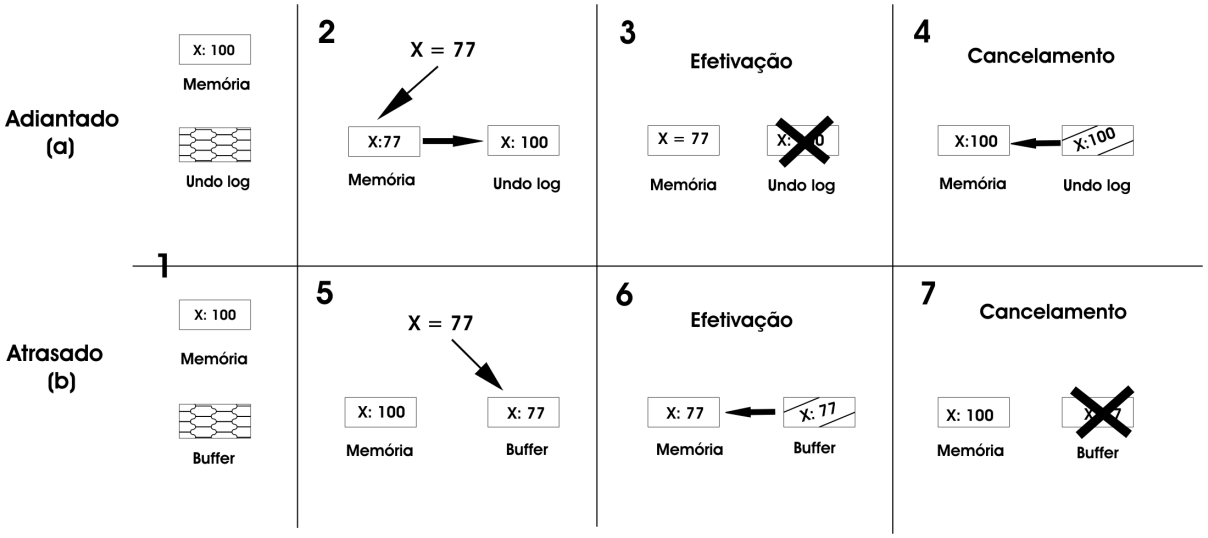

Figura 1 – Exemplo de versionamento adiantado (a) e atrasado (b). Fonte: (BALDASSIN, 2009)

Existem dois tipos de versionamento de dados:

- **Versionamento Adiantado**: como pode ser visto na Figura 1 (a), o valor modificado durante a transação é armazenado direto na memória e o valor inicial é armazenado em um *undo log*, para que no caso de cancelamento na transação o valor inicial seja restaurado na memória.
- **Versionamento Atrasado**: como pode ser visto na Figura 1 (b) neste versionamento o valor modificado durante a transação é armazenado em um *buffer* e o valor inicial é mantido na memória até que aconteça um *commit* na transação, onde o valor armazenado no *buffer* é escrito na memória. Caso aconteça o cancelamento na transação, o valor do *buffer* é descartado.

2.3 Detecção de Conflito

Mecanismos de detecção de conflitos verificam a existência de operações conflitantes durante uma transação. Um conflito ocorre quando duas transações estão acessando um mesmo dado na memória e pelo menos uma das transações está fazendo uma operação de escrita (BALDASSIN, 2009).

Da mesma forma que o versionamento de dados, a detecção de conflito também pode ser de dois tipos:

- **Detecção de Conflitos Adiantado**: ocorrem no momento em que duas transações acessam um mesmo dado e uma delas faz uma operação de escrita. Essa operação de escrita é detectada e então uma transação é abortada. Neste tipo de detecção pode ocorrer um problema chamado de *livelock*, quando duas transações ficam cancelando-se, desta forma, a execução do programa não progride. A Figura 2 mostra como é feita a detecção de conflitos adiantado.

  O Caso 1, mostra a execução sem conflitos, onde as duas transações são executadas sem problemas. Já o Caso 2, mostra o que acontece quando ocorre um conflito, onde T1 lê A e logo depois T2 escreve em A, então o conflito é detectado e T1 é abortada, após ser efetivada T2, a transação T1 consegue ler A sem problema de conflito. Por fim o Caso 3 mostra a situação de *livelock*, onde as duas transações tentam ler e escrever em A, assim as duas acabam sempre se abortando.

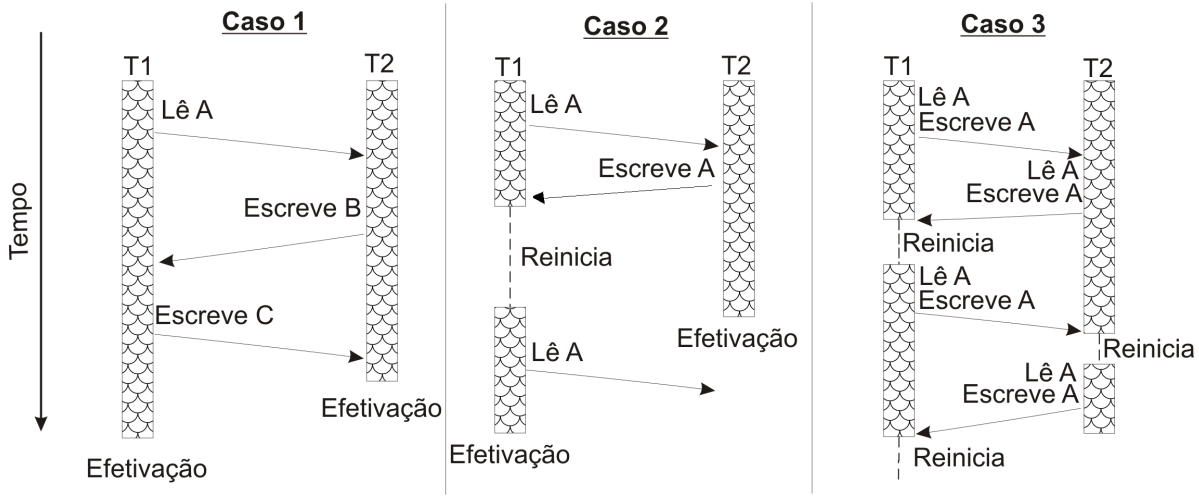

Figura 2 – Detecção de conflitos em modo adiantado. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

- **Detecção de Conflitos Atrasado**: Este tipo de detecção de conflito ocorre no final da transação. Antes da transação ser efetuada, é verificado se ocorreu um conflito. Caso tenha ocorrido, a transação é cancelada, senão é efetivada. Para transações muito grandes não é recomendado este tipo de detecção, pois uma

transação grande pode ser abortada várias vezes por transações pequenas, assim gastando tempo de processamento desnecessário, este problema se chama *starvation*. A Figura 3 mostra como é feita a detecção de conflitos atrasado.

O Caso 1, mostra as transações acessando dados diferentes, não ocasionando conflitos. No Caso 2, T2 lê A que é escrita por T1. A T2 só nota o conflito quando T1 é efetivado. Logo depois de notar o conflito T2 é abortada. No Caso 3 não ocorre nenhum conflito, pois T1 lê A antes de T2 escrever. O Caso 4 mostra a situação em que, após ser cancelada, T1 volta a executar.

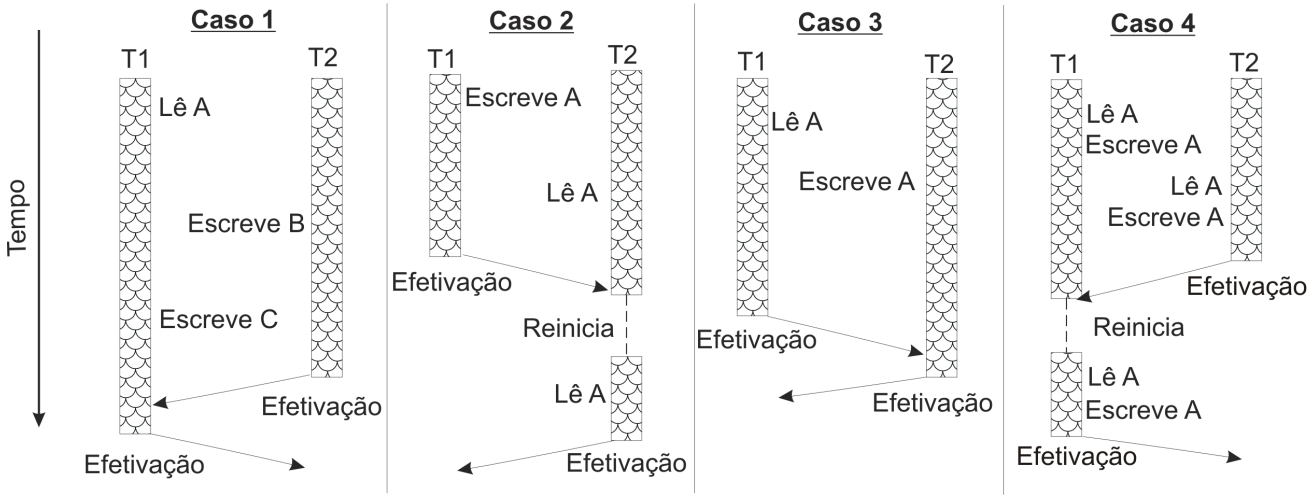

Figura 3 – Detecção de conflitos em modo atrasado. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

Para solucionar o problema de qual transação continuará executando, quando ocorre um conflito, é utilizado um gerenciador de contenção (HARRIS; LARUS; RAJWAR, 2010). O gerenciador de contenção é o responsável por decidir quando e qual transação vai ser abortada, isso para garantir que a execução do programa prossiga sem problemas.

2.4 TinySTM

A *TinySTM* (FELBER; FETZER; RIEGEL, 2008) é uma implementação de STM para as linguagens C e C++. Seu algoritmo é baseado em outros algoritmos de STM como o TL2 (*Transactional Locking* 2) (DICE; SHALEV; SHAVIT, 2006). Ela é uma biblioteca utilizada para escrever aplicativos que usam memórias transacionais para sincronização, em substituição aos tradicionais *locks*.

2.4.1 Sincronização e Versionamento

Na *TinySTM* a sincronização é feita a partir de um *array* de *locks* compartilhado que gerencia o acesso concorrente à memória. Cada *lock* é do tamanho de um ende-

reço da arquitetura (FELBER; FETZER; RIEGEL, 2008), e bloqueia vários endereços de memória. O mapeamento é feito por meio de uma função *hash*. A Figura 4 apresenta as estruturas de dados utilizadas nesta implementação.

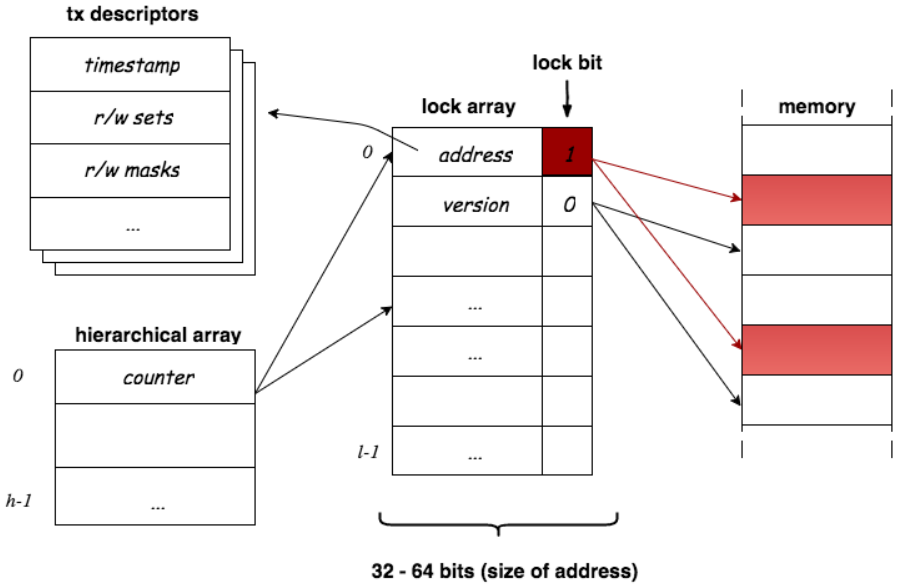

Figura 4 – Estruturas de dados utilizadas na *TinySTM*. Fonte: (FELBER; FETZER; RIEGEL, 2008)

O bit menos significativo é utilizado para indicar se o *lock* está em uso. Se o bit menos significativo indicar que o *lock* não está em uso, nos bits restantes são armazenados um número de versão que corresponde ao *timestamp* da transação que escreveu por último em um dos locais de memória abrangidos pelo *lock*.

Se o bit menos significativo indica que o *lock* está em uso, então nos bits restantes é armazenado um endereço que identifica a transação que está utilizando o dado (isso utilizando o versionamento adiantado), ou uma entrada no *write set* da transação que está utilizando o dado (isso utilizando o versionamento atrasado). Em ambos os casos os endereços apontam para uma estrutura que é *word-aligned* e seu bit menos significativo é sempre zero, por isso, o bit menos significativo pode ser utilizado como bit de bloqueio.

Quando utilizado o versionamento atrasado, o endereço armazenado no *lock* permite uma operação rápida para localizar as posições de memória atualizadas abrangidas pelo *lock*, no caso de serem acessados novamente pela mesma transação. Em contraste, a TL2 deve verificar o acesso à memória se a transação atual ainda não escreveu neste endereço, o que pode ser caro quando *write sets* são grandes. A leitura depois da escrita não é um problema quando é utilizado o versionamento adiantado porque a memória sempre contém o último valor escrito na memória pela transação ativa.

A *tinySTM* apresenta três estratégias de versionamento distintas que podem ser

utilizadas, sendo que duas utilizam versionamento atrasado (write-back) e uma utiliza versionamento adiantado (write-through), estas são:

- **Write_Back_ETL**: esta estratégia implementa o versionamento atrasado com *encounter-time locking*, isso é, o *lock* é adquirido após ocorrer uma operação de escrita e atualiza o *buffer*. O valor é escrito na memória no momento do *commit* da transação;
- **Write_Back_CTL**: esta estratégia implementa o versionamento atrasado com *commit-time locking*,isto é, ele adquire o *lock* antes de ocorrer o um *commit* e atualizar o *buffer*. Assim como no *Write-Back-ETL* o valor é escrito na memória no momento do *commit* da transação;
- **Write_Through**: esta estratégia implementa o versionamento adiantado com *encounter-time locking*, isto é, o valor é escrito direto na memória e mantém um *undo log*, caso ocorra um *abort* na transação é possível restaurar o valor anterior na memória.

A *TinySTM* utiliza *Write_Back_ETL* como sua estratégia de versionamento padrão.

2.4.2 Escritas

Quando ocorre uma escrita em um local da memória, a transação primeiro identifica o *lock* correspondente ao endereço de memória e lê o valor. Se o *lock* está em uso a transação verifica se é a proprietária do *lock* utilizando o endereço armazenado nos restantes bits de entrada. Caso a transação seja a proprietária então ela simplesmente escreve o novo valor e retorna. Caso contrário, a transação pode esperar por algum tempo ou abortar imediatamente. A *TinySTM* utiliza a última opção como padrão em sua implementação.

Se o *lock* não está em uso, a transação tenta adquiri-lo para escrever o novo valor na entrada utilizando uma operação atômica *compare-and-swap*. A falha indica que outra transação adquiriu o *lock* nesse meio tempo, então a transação é reiniciada.

2.4.3 Leituras

Quando ocorre uma leitura na memória, a transação deve verificar se o *lock* está em uso ou se o valor já foi atualizado concorrentemente por outra transação. Para esse fim, a transação lê o *lock* correspondente ao endereço de memória. Se o *lock* não tem proprietário e o valor (número de versão) não foi modificado entre duas leituras, então o valor é consistente.

2.4.4 Gerenciamento de Memória

A *TinySTM* utiliza um gerenciador de memória que possibilita qualquer código transacional utilizar memória dinâmica. As transações mantém o endereço da memória alocada ou liberada. A alocação de memória é automaticamente desfeita quando a transação é abortada, já a liberação não pode ser desfeita antes do *commit*. Contudo uma transação pode somente liberar memória depois de adquirir todos os *locks*, assim, um *free* é semanticamente equivalente a uma atualização.

2.4.5 Gerenciador de Contenção

A *TinySTM* implementa quatro estratégias de gerenciador de contenção, estas são:

- **CM_Suicide**: nesta estratégia a transação que detecta o conflito é abortada imediatamente;
- **CM_Delay**: esta estratégia assemelhasse a *CM_Suicide*, porem, espera até que a transação que gerou o *abort* tenha liberado o *lock*, então reinicia a transação. Isto porque por intuição a transação a transação que foi abortada irá tentar adquirir o mesmo *lock* novamente, provavelmente falhando em mais de uma tentativa. Está estratégia aumenta as chances de que a transação tenha sucesso sem gerar um grande número de *aborts*, melhorando o tempo de execução do processador;
- **CM_Backoff**: também parecida com a *CM_Suicide*, esta estratégia espera um tempo randômico para reiniciar a transação. Este tempo de espera é escolhido ao uniformemente ao acaso em um intervalo cujo tamanho aumenta exponencialmente a cada reinicialização;
- **CM_Modular**: esta estratégia implementa vários gerenciadores de contenção, que são alternados durante a execução. Os gerenciadores utilizados são:
  - **Suicide**: a transação que descobriu o conflito é abortada;
  - **Aggressive**: é o inverso da *Suicide*, a transação abortada é a outra e não a que descobriu o conflito;
  - **Delay**: a mesma que a *Suicide*, mas aguarda pela resolução do conflito para reiniciar a transação;
  - **Timestamp**: a transação mais nova é abortada.

A *TinySTM* utiliza a *CM_Suicide* como sua estratégia padrão de gerenciamento de contenção.

2.5 STAMP

Stanford Transactional Applications for Multi-Processing (MINH et al., 2008) é um conjunto de *benchmarks* criado para pesquisa de memórias transacionais, composto por oito *benchmarks*. Apesar de desenvolvido para a STM TL2, com algumas modificações disponíveis pode ser usado no *TinySTM*.

O conjunto de *benchmarks* STAMP implementa vários *benchmarks*, assim, atingindo uma maior área de aplicações das STM e é o conjunto de *benchmark* mais utilizado na pesquisa de STM. Os oito benchmarks apresentados são:

- **Bayes**: Apresenta uma rede bayesinana de aprendizado;
- **Genome**: Implementa uma aplicação que reconstrói a sequencia de um gene a partir de sequências maiores; S
- **Intruder**: Simula o Design 5 do *Network Intrusion Detection System* (NIDS) (HAAGDORENS; VERMEIREN; GOOSSENS, 2005);
- **Kmeans**: *K-means* é um algoritmo comumente usado para partição de itens de dados em subconjuntos relacionados;
- **Labyrinth**: Implementa uma algoritmo que descobre o menor caminho entre um ponto inicial e um ponto final;
- **SSCA2**: É composto por quatro *kernels* que operam em um grande, dirigido e ponderado gráfico;
- **Vacation**: Implementa um sistema de reserva de viagens alimentado por um banco de dados não-distribuído; e
- **Yada**: Implementa o algoritmo de Ruppert (RUPPERT, 1995) para refinamento de malha.

Neste trabalho vai ser utilizada a versão 0.9.10 do STAMP para avaliar e comparar a execução da biblioteca de STM TinySTM atual e a utilização do escalonador proposto.

3 ESCALONADORES

O uso de escalonadores provem melhorias nas execuções de programas, pode-se utilizar escalonadores de tarefas para melhorar o desempenho de arquiteturas, como visto no trabalho (FAVARETTO, 2014), consegue-se utilizar um escalonador para reduzir a latência de acesso à memória pelo processador em arquiteturas *NUMA*.

Em STM o uso de escalonadores pode reduzir o número de conflitos gerados pelo aumento do paralelismo, como podemos ver em (NICÁCIO; BALDASSIN; ARAÚJO, 2012) o *LUTS* apresenta heurísticas de detecção de conflitos para que o escalonador de transações evite *aborts* no decorrer de sua execução.

Escalonadores fornecem diferentes abordagens para cada problema proposto, estas distintas abordagens permitem aos desenvolvedores explorar heurísticas de escalonamento que se adaptam a arquitetura utilizada propiciando uma solução mais eficiente.

Existem muitas heurísticas diferentes para prever conflitos, estas podem servir como base para o escalonamento de transações. Para este trabalho foram estudados algumas das principais heurísticas e suas classificações.

O trabalho apresentado em (DI SANZO, 2017) fornece uma categorização dos escalonadores de STM, na qual os algoritmos são classificados de acordo com suas heurísticas.

Está categorização é dividida em algoritmos Baseados em Heurística e algoritmos Baseados em Modelo. Cada categorização possui classificações de acordo com o comportamento de sua heurística.

- Baseado em Heurística:
  - Feedback: Utiliza o feedback da execução para realimentar sua heurística;
  - Predição: Utiliza uma predição das informações para tomada de decisão;
  - Reativo: Só executa sua heurística após determinado comportamento da aplicação ocorrer; e
  - Heurística Mista: Mescla as classificações anteriores para otimizar a heurística.

Tabela 2 – Algoritmos e técnicas de escalonamento

Escalonador	Técnica
ATS	Feedback
Probe	Feedback
F2C2	Feedback
Shrink	Predição
SCA	Predição
CAR-STM	Reativo
RelSTM	Reativo
LUTS	Heurística Mista
ProVIT	Heurística Mista
SAC-STM	Aprendizado de Máquina
CSR-STM	Modelo Analítico
MCATS	Modelo Analítico
AML	Modelo Misto

- Baseado em Modelo:
  - Aprendizado de Máquina: Utiliza algoritmos de aprendizado de máquina para tomar decisão;
  - Modelo Analítico: Monta modelos analíticos para tomar decisão; e
  - Modelo Misto: Mistura as classificações acima para otimizar a heurística.

A tabela 2 apresenta as classificações dos principais algoritmos revisados na bibliografia durante o desenvolvimento deste trabalho.

3.1 ATS

Adaptive Transaction Scheduling (ATS) (YOO; LEE, 2008) foi um dos primeiros trabalho a apresentar um escalonador de MT para trabalhar junto com gerenciador de contenção.

O ATS utiliza um valor para tomada de decisão denominado CI (Contention Intensity), cada thread em execução possui seu próprio CI. O CI é calculado cada vez que ocorre um commit ou um abort e é zerado a cada inicio de transação.

O escalonador utiliza o valor do CI em sua tomada de decisão. Quando o valor do CI ultrapassa um limiar pré definido, a thread é colocada em uma única fila para garantir uma execução de forma serial.

3.2 CAR-STM

Collision Avoidance and Resolution (CAR-STM) (DOLEV; HENDLER; SUISSA, 2008) foi desenvolvido para evitar que conflitos já existentes voltem a ocorrer. Para isto é apresentado duas heurísticas de gerenciamento.

A primeira heurística é denominada Básica e busca executar de forma serial as transações conflitantes sem manter um histórico da execução. A segunda denominada

Permanente busca manter um histórico das transações que conflitaram e executa-las de forma serial.

- Básica: Quando detectado um conflito, a transação mais recente é abortada e migrada para fila da transação conflitante, assim sua execução sera serializada.
- Permanente: Quando uma transação Tb aborta em relação a Ta, Tb é migrado para fila de Ta e sua ordem de execução será Ta -> Tb. Caso a transação Ta conflite e aborte em relação a Tc, Ta deverá ser migrada para fila de Tc carregando sua dependência Ta -> Tb.

3.3 LUTS

Light-Weight User-Level Transaction Scheduler (LUTS) (NICÁCIO; BALDASSIN; ARAÚJO, 2012) apresenta um escalonador que busca evitar a ociosidade de um núcleo após a serialização de uma transação.

Para isto cada thread é representado internamente por um Registro de Contexto em Execução (RCE), cada RCE encapsula uma thread. No inicio da execução o escalonador cria uma fila de RCEs para serem executados no futuro.

Assim o LUTS dispara um RCE por núcleo, e utiliza a fila para não disparar mais RCEs que núcleos disponíveis. Cada REC disparado é convertido em uma thread de sistema que executa um conjunto de transações.

Na tentativa de evitar conflitos o LUTS apresenta uma forma dinâmica para soluciona-los, considerando transações curtas e transações longas. Para definir o tamanho da transação é utilizada a contagem de ciclos da mesma, onde a partir de 100 mil ciclos temos uma transação longa.

Para transações curtas, a heurística utilizada é similar a do ATS, o escalonador calcula a intensidade de conflito da transação e serializa esta quando o calculo ultrapassa um limiar. Porém o LUTS escolhe outra transação para substituir a atual.

Para transações longas, a heurística é mais elaborada, utilizando três metadados globais:

- activeTx: Um vetor de tamanho igual ao total de núcleos disponíveis, usado para armazenar o identificador da transação que está sendo executada.
- conflictTable: Uma tabela do histórico de conflitos, cada linha armazena um conjunto de transações dada pelo activeTx, e cada coluna armazena a probabilidade de conflito.
- bestTx: Um vetor que sumariza a melhor transação a ser executada para cada núcleo.

Quando uma transação realiza um commit ou abort o escalonador se encarrega de atualizar a conflictTable na sua respectiva linha, aumentando ou diminuindo sua probabilidade de conflito.

Para evitar percorrer a conflictTable no inicio de cada transação o LUTS percorre a bestTx e seleciona qual transação deve executar. Quando a conflictTable é atualizada o escalonador atualiza a bestTx.

3.4 Shirink

Shirink (DRAGOJEVIć et al., 2009) apresenta um escalonador que busca minimizar a ocorrência de aborts com base nos conjuntos de leituras e escritas ocorridos em cada thread.

O escalonador é baseado em predição e usa como heurística os acessos à memória das transações executadas anteriormente. Para evitar overhead em sua execução o Shirink avalia os acessos apenas se existir uma alta contenção no sistema.

No inicio de cada transação o escalonador avalia se existe a relação entre commit e abort é superior a um limiar predefinido, se esse valor for superior ao limiar o escalonador considera que o sistema possui uma alta contenção e ativa a heurística para serializar as transações em execução.

Cada thread possui um conjunto dos acessos de leitura e escrita realizados pelas transações, quando uma transação vai iniciar com um sistema de alta contenção esse conjunto de leitura e escrita é verificado, se outra thread em execução possuir um conjunto semelhante o escalonador assume que há uma alta chance da transação abortar.

Para que a transação não aborte a thread que iniciaria a transação é bloqueada até o fim da transação na thread em execução, assim o Shrink busca forçar a serialização das execuções.

3.5 ProVit

O escalonador *ProVIT* (RITO; CACHOPO, 2015) fornece uma abordagem otimista da execução, evitando considerar que toda transação que abortou ira abortar novamente na sequencia.

Assim como o LUTS o ProVIT avalia o tamanho das operações atômicas para aplicar sua heurística. Porém no ProVIT mais de uma heurística pode estar ativa ao mesmo tempo.

Também foi apresentada a observação que duas transações conflitantes, de leitura e escrita, podem efetuar o commit dependendo da ordem, se o commit for efetuado primeiro pela transação de leitura, a de escrita não será conflitante.

Operações atômicas longas utilizam uma política é baseada em grão fino para melhorar a precisão da predição e evitar a reexecução de transações. Essa predição utiliza como base o conjunto de leitura das transações já executadas.

Se uma transação efetua um abort o escalonador marca esta transação como Very Important Transaction (VIT) e copia seu conjunto de leitura para uma lista auxiliar global. Quando uma transação tenta efetuar um commit ela verifica a lista global para garantir que não ha conflito entre os conjuntos de escrita e leituras.

Caso haja conflito entre a transação e alguma VIT, o commit é adiado por um tempo pré-determinado. Assim, o escalonador tenta garantir que as VITs não abortem novamente. Caso não haja o conflito o commit é realizado.

Nas operações atômicas curtas a heurística evita a validação com base na intersecção para não adicionar overhead desnecessário. Sendo assim, ela apresenta uma ideia similar a do ATS, onde é utilizada uma métrica de decisão para serializar as transações.

Para definir quando uma transação será serializada, o escalonador utiliza um valor calculado em tempo de execução denominado Tempo Perdido (TP). Cada operação atômica possui seu próprio TP, que é calculado com base em um valor pré-definido e a quantidade de reexecução da transação e seu TP anterior.

Toda operação atômica começa com TP igual a zero e é executada livremente, conforme o TP aumenta o ProVit se encarrega de serializar as transações reduzindo a concorrência até o ponto em que somente uma transação poderá ser executada por vez.

Para definir se uma operação atômica é curta ou longa foi implementado uma politica de definição, onde toda operação atômica inicia sua execução como curta. Quando uma transação finaliza o escalonador atualiza seu conhecimento sobre as operações atômicas com base no TP.

Uma operação atômica é considerada longa quando o tamanho médio do seu conjunto de leitura for maior que um limite pré-definido. Assim, o tamanho da operação atômica é baseado nas operações de leitura e não no tempo de execução.

3.6 STMap

O *STMap* (PASQUALIN et al., 2020a) apresenta um escalonador que utiliza mecanismos de sharing-aware mapping para aplicações STMs. O escalonador utiliza esse mecanismo para executar as transações concorrentes no mesmo núcleo NUMA para otimizar a execução da aplicação.

Em tempo de execução o escalonador coleta dados sobre os comportamentos compartilhados entre as threads, esses dados são usados para calcular um mapeamento otimizado de threads para núcleos e migra as threads em execução.

A coleta de dados é realizada em tempo de execução quando ocorre uma escrita ou leitura dentro de uma transação, esse endereço de leitura ou escrita é comparado com os endereços utilizados pelas outras transações. Se esse endereço é usado por outra transação o STMap incrementa uma matriz de comunicação com essas informações.

A matriz de comunicação possui como tamanho o mesmo número de threads em execução. Supondo que a transação t1 executada pela thread T1 manipule o mesmo endereço que a transação t2 da thread T2, a transação t1 descobre o ID da thread T1 e da thread T2 e incrementa o valor da matriz de comunicação nas posições respectivas aos IDs.

Essa matriz é utilizada para caracterizar as transações e mapear a arquitetura em relação as threads, assim sendo possível agrupar as threads por nodos de processamento para otimizar a execução e aproveitar ao máximo a coerência de cache.

4 LTMS - LUPS TRANSACTIONAL MEMORY SCHEDULER

Memórias transacionais fornecem um nível maior de abstração para o desenvolvimento de programas paralelos, a área caminha para o uso de escalonadores de transações que compreendem a aplicação para extrair o melhor desempenho.

Porém os escalonadores atuais não consideram as diferenças entre as arquiteturas paralelas existentes. O escalonador LTMS se propõem a avaliar a aplicação e a arquitetura em tempo de execução para tirar máximo de proveito do paralelismo existente.

As duas principais arquiteturas paralelas que serão abordadas na proxima seção são, arquiteturas UMA e arquiteturas NUMA. Os escalonadores atuais assim como as bibliotecas de STM são pensados para arquiteturas UMA não considerando as diferenças quando executadas em NUMA.

4.1 Motivação

Máquinas *NUMA* tem a vantagem de agregar maior paralelismo ao adicionar mais processadores sem aumentar o gargalo de acesso ao barramento. Sua arquitetura é feita para que os processadores não utilizem o mesmo barramento de acesso à memória como é feito em arquiteturas *UMA*.

As arquiteturas *NUMA* possuem múltiplos núcleos dispostos em conjuntos de processadores (Nodos), a memória é fisicamente composta por vários bancos de memória, podendo estar cada um deles vinculados a um Nodo e a um espaço de endereçamento compartilhado.

Nesse caso, quando o processador acessa à memória que está vinculada a si, diz-se que houve um acesso local. Se o acesso for à memória de outro processador, diz-se que ocorreu um acesso remoto.

Os acessos remotos são mais lentos que os acessos locais, uma vez que é necessário passar pela rede de interconexão para que se consiga chegar ao dado localizado na memória remota (FAVARETTO, 2014).

Os escalonadores de STM atuais buscam reduzir o número de conflitos para que

ocorra a menor quantidade de reexecução das transações. Para isto estes escalonadores implementam filas de execução e migração de threads que tornam serial a execução das transações conflitantes como nos trabalhos apresentados em (NICÁCIO; BALDASSIN; ARAÚJO, 2012).

Alguns algoritmos NUMA-Aware avaliam os conjuntos de leitura e escrita para decidir qual o melhor nodo de execução para o thread ou quando deve ser migrado o banco de memória. Outros utilizam de uma matriz de comunicação para fazer um mapeamento de threads e assim otimizar a execução, como é feito em (PASQUALIN et al., 2020b).

Os escalonadores de STM atuais não consideram a arquitetura e seu custo de acesso à memória para serializar as execuções. Alguns escalonadores de STM avaliam os conjuntos de leitura e escrita apenas com interesse em reduzir o número de conflitos (DRAGOJEVIć et al., 2009).

Com isto o LTMS implementa um escalonador que avalia as características da arquitetura, e em tempo de execução monta uma matriz de comunicação com base nas leituras e escritas realizadas pelos threads. Está matriz de comunicação é utilizada para avaliar o custo de acesso à memória e migrar as threads em execução entre as filas, buscando diminuir os números de conflitos por meio da serialização das transações e otimizar a execução aproveitando a melhor distribuição de tarefas na arquitetura NUMA, reduzindo assim a latência de acesso à memória.

4.2 Escalonador

O LTMS é um escalonador de STM NUMA-Aware que identifica as características da arquitetura e do programa em tempo de execução para extrair o máximo de desempenho da máquina utilizada e facilitar a implementação de códigos paralelos.

Como podemos ver na figura 5 o escalonador LTMS é inicializado junto com a aplicação, o escalonador é responsável por ler as característica da arquitetura e criar filas de execução com base no número de cores disponíveis e as threads da aplicação.

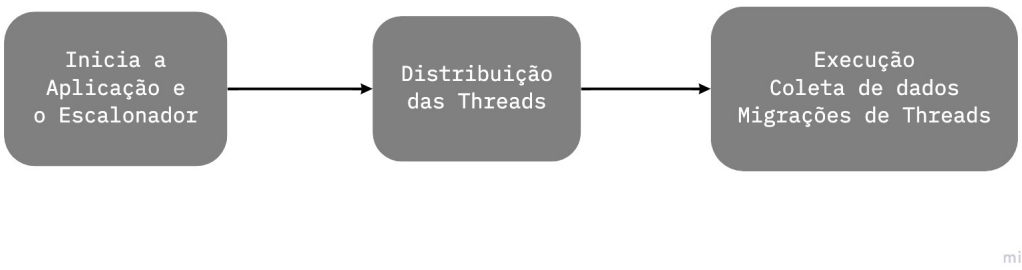

Figura 5 – Fluxo de execução da LTMS

Para reduzir o custo de inicialização o escalonador evita criar mais filas de execução do que o necessário para a aplicação. Assim, o LTMS compara o número de threads da aplicação com a quantidade de cores da máquina, se o número de threads da aplicação for maior que o número de cores o LTMS cria uma fila para cada core, como visto na figura 6.

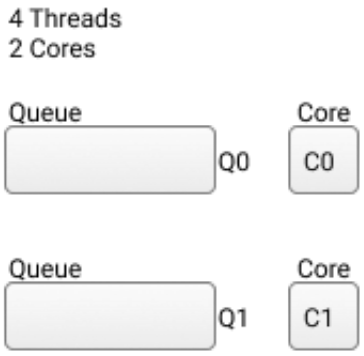

Figura 6 – Criação das filas de execução com base nos cores

Se a quantidade de threads da aplicação for menor que o número de cores disponível na arquitetura, o LTMS cria a mesma quantidade de filas que a quantidade de threads, fixando um core por fila e distribuindo uma thread por fila, como visto na figura 7

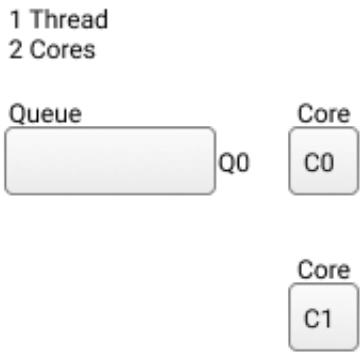

Figura 7 – Criação das filas de execução com base nas threads

O escalonador também se encarrega de distribuir inicialmente as threads entre as filas de execução. Para distribuição de threads entre as filas foram desenvolvidas duas heurísticas que serão apresentadas na subseção 4.2.1.

Em tempo de execução o LTMS coleta as informações como, endereços de leitura e

escrita mais utilizados, quais threads possuem a mesma afinidade de leitura e escrita, e a quantidade de commits e aborts existentes por thread.

Caso ocorra um abort em uma transação, essas informações serão utilizadas para identificar qual fila a thread em execução tem maior afinidade caso sejá necessário executar uma migração.

Para o LTMS decidir se deve migrar ou não uma thread de fila, foram desenvolvidas duas heurísticas que buscam minimizar o overhead do escalonador e aproveitar as características da arquitetura. Estas heurísticas de migrações serão detalhadas na subseção 4.2.2.

4.2.1 Heurísticas de distribuição

Após a criação das filas de execução, conforme apresentado acima, as threads criadas na aplicação são distribuídas com base em uma heurística de distribuição. Para este trabalho foram implementadas duas heurísticas.

A primeira heurística distribui uma thread por fila até a conclusão de todas threads disponíveis. A figura 8 traz como exemplo 4 threads e 2 cores, neste caso serão criadas uma fila para cada core.

O escalonador executará a primeira fase de distribuição, colocando uma thread para cada fila existente. Após a primeira fase o escalonador verifica se ainda possui threads a serem distribuídas, caso hajam threads o LTMS repete a distribuição em uma segunda fase, até acabarem as threads.

Neste cenário a Fila intitulada Q0 fica com as threads t0 e t2, e a fila Q1 fica com as threads t1 e t3. Veja que o LTMS alocou a thread t0 em Q0 e depois alocou t1 em Q1, então voltou a execução para alocar t2 em Q0 e t3 em Q1.

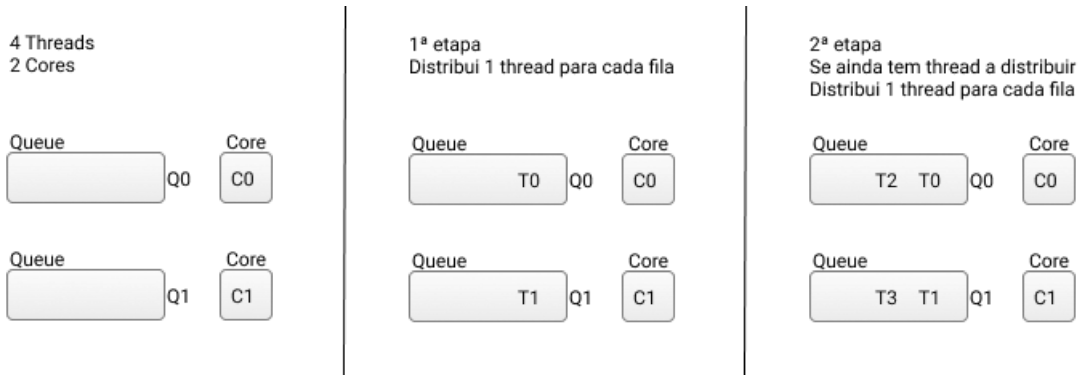

Figura 8 – Heurística um de distribuição de threads

A segunda heurística distribui duas threads por fila, no exemplo apresentado na figura 9 temos o mesmo cenário de filas, cores e threads apresentados anteriormente.

Na primeira fase de distribuição o escalonador aloca duas threads por fila, quando acabam as filas o escalonador verifica se há thread disponível para distribuição, se

não há mais threads o LTMS encerra a distribuição. Caso haja apenas uma ultima thread para distribuir o escalonador aloca esta na fila em que parou a distribuição.

Neste cenário o LTMS aloca na fila Q0 as threads t0 e t1, e na fila Q1 as threads t2 e t3.

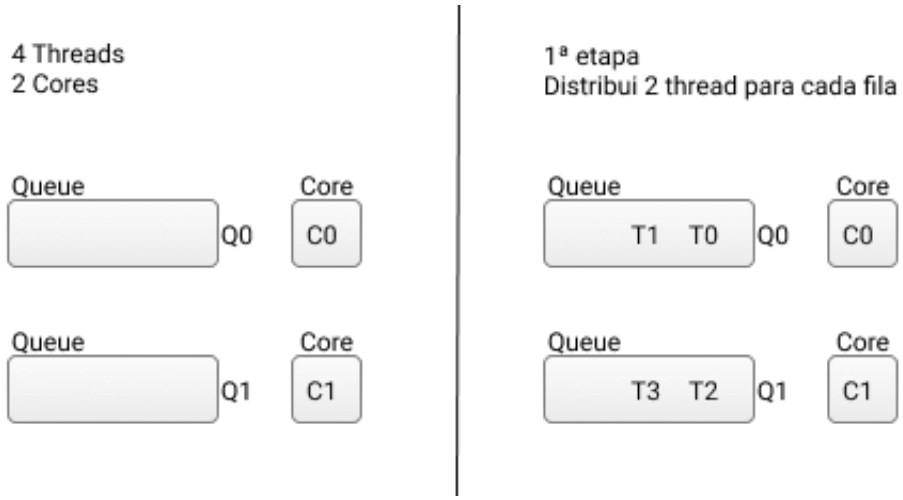

Figura 9 – Heurística dois de distribuição de threads

4.2.2 Heurísticas de migração

Durante a execução das threads são coletados informações sobre sua execução com base em sua fila atual. Entre os dados coletados é montada uma matriz de comunicação e uma matriz de endereços.

A matriz de comunicação consiste em uma matriz que armazena quantas vezes um thread lê e escreve no mesmo endereço de memória que outro thread. A matriz de endereço armazena os endereços de memória no qual um thread realizou operações de leitura e escrita. Outros dados também armazenados por thread em tempo de execução são as quantidade de aborts e commits realizados. Cada thread pode executar n transações, a cada commit e abort de uma transação o thread incrementa seu respectivo contador para manter esse dado atualizado. No momento que a transação efetua um abort, o escalonador avalia a possibilidade de migrar todo thread em execução para uma nova fila, buscando otimizar a execução e evitar futuros aborts.

Para reduzir o impacto no tempo gerados pela migração threads de forma inapropriada foram desenvolvidas duas heurísticas de migração, que avaliam os dados coletados para tomar a decisão de migrar a thread.

Para as duas heurísticas a fila para qual pretendemos migar o thread é escolhida com base na matriz de comunicação. O thread que possui a transação que abortou avalia qual o thread que tem maior afinidade de leitura e escrita. Após identificar o thread com maior afinidade, o LTMS descobre em qual fila o thread está e escolhe esta fila para migrar o thread atual.

A primeira heurística, denominada threshold, apresentada na figura 10 avalia o nivel de contenção apresentado pelo thread em tempo de execução, esse nivel de contenção é medido pela razão entre os aborts e commits realizados pelo thread, onde um resultado alto indica uma maior contenção ocasionada pelos aborts. Para utilizar esta heurística o LTMS primeiro defini para qual fila pretende migrar o thread atual, então consulta com o thread qual a razão entre a quantidade de aborts e commits (nivel de contenção). Se o nivel de contenção do thread atual for superior a um limiar previamente estipulado pelo desenvolvedor, o escalonador decide por migrar o thread atual.

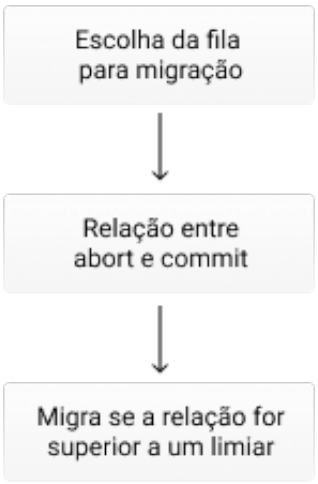

Figura 10 – Heurística de migração threshold

A segunda heurística de migração implementa, denominada latency, é apresentada na figura 11. Nesta heurística após o LTMS escolher a fila para qual a thread pode ser migrada, o escalonador avalia a latência de acesso à memória. Para isto, o LTMS consulta na matriz de endereços, qual o endereço em comum entre o thread atual e o thread que executa na fila indicada para migração. O LTMS armazena qual o node NUMA esse endereço pertence. Com a informação do nodo NUMA do endereço de memória, o escalonador calcula a latência de acesso da fila do thread atual para esse endereço e a latência da futura fila para esse endereço. Se a fila atual possui uma latência de acesso maior que a fila para a qual pretendemos migrar a thread, o escalonador efetua a migração. Caso a latência seja menor ou igual a thread mantem sua execução na fila atual.

Para realizar a migração ambas as heurísticas bloqueiam o thread atual, então o escalonador se encarrega de trocar as informações sobre a execução da thread para nova fila. Após trocar o thread de fila o escalonador verifica se a fila anterior possui mais threads para executar, e assim da inicio a execução de um novo thread.

Se o LTMS decidir não migrar a thread, a transação segue seu fluxo atual utilizando

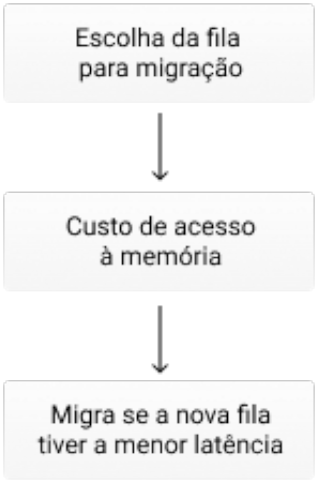

Figura 11 – Heurística de migração latency

os mecanismos de gerenciamento de conflito existentes nas bibliotecas STM.

4.3 Aplicação

O LTMS é um escalonador de STM que inicializa sua execução junto com a aplicação e fornece todo suporte a memórias transacionais. A figura 12 apresenta o fluxo de execução de uma aplicação utilizando o LTMS.

O escalonador foi desenvolvido em C++ para ser utilizado com a biblioteca de STM TinySTM. Para este trabalho foram utilizado os benchmarks do conjunto de benchmark STMAP.

A implementação do LTMS trás um conjunto de funções que utilizam recursos de threads do c++ para executar a aplicação junto com a biblioteca TinySTM. Estes ...

...

...

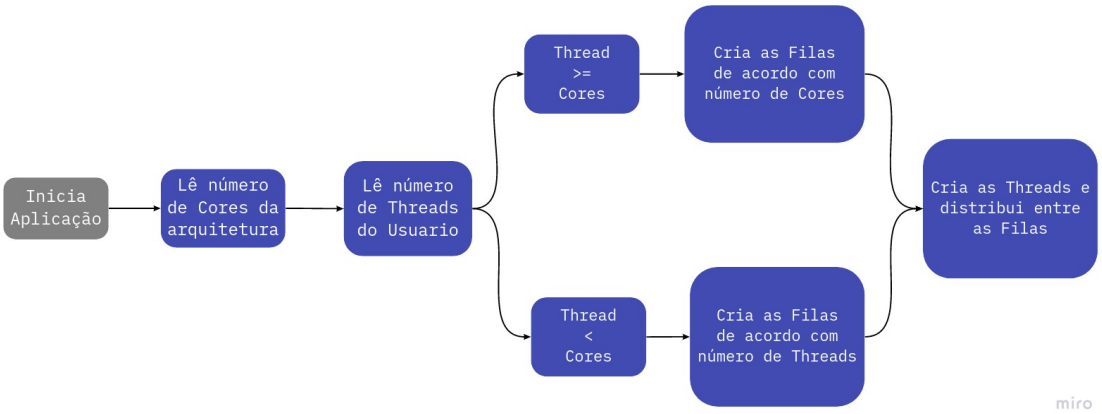

Figura 12 – Inicialização da LTMS

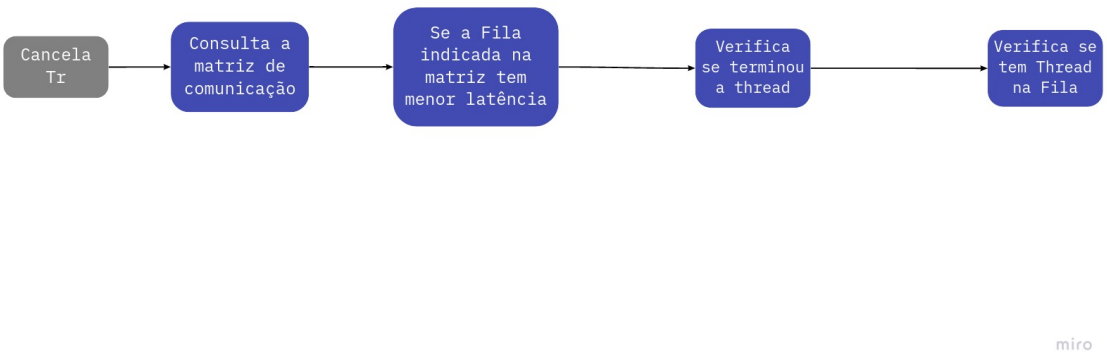

Figura 13 – Migração de threads na LTMS

5 EXPERIMENTOS

...

5.1 Resultados

...

6 CONCLUSÃO

...

REFERÊNCIAS

BALDASSIN, A. J. **Explorando Memória Transacional em Software nos Contextos de Arquiteturas Assimétricas, Jogos Computacionais e Consumo de Energia**. 2009. Dissertação de Doutorado — Universidade Estadual de Campinas.

BANDEIRA, R. de Leão. **Compilador para a linguagem CMTJava**. 2010. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) — Universidade Federal de Pelotas.

DI SANZO, P. Analysis, classification and comparison of scheduling techniques for software transactional memories. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.28, n.12, p.3356–3373, 2017.

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional Locking II. In: DISC 2006, 2006. **Anais...** [S.l.: s.n.], 2006. p.194–208.

DOLEV, S.; HENDLER, D.; SUISSA, A. CAR-STM: Scheduling-based Collision Avoidance and Resolution for Software Transactional Memory. In: TWENTY-SEVENTH ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.125–134. (PODC '08).

DRAGOJEVIć, A.; GUERRAOUI, R.; SINGH, A. V.; SINGH, V. Preventing Versus Curing: Avoiding Conflicts in Transactional Memories. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 28., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.7–16. (PODC '09).

FAVARETTO, R. M. **Escalonamento dinâmico em nível aplicativo sensível à arquitetura e às dependências de dados entre as tarefas**. 2014. Dissertação de Mestrado — PPGC/UFPEL, Pelotas/RS.

FELBER, P.; FETZER, C.; RIEGEL, T. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: PPOPP '08: PROC. OF THE 13TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.237–246.

HAAGDORENS, B.; VERMEIREN, T.; GOOSSENS, M. Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading. In: INFORMATION SECURITY APPLICATIONS, 2005. **Anais...** [S.l.: s.n.], 2005. p.188–203. (Lecture Notes in Computer Science (LNCS), v.3325).

HARRIS, T.; LARUS, J.; RAJWAR, R. Transactional Memory, 2nd edition. **Synthesis Lectures on Computer Architecture**, [S.l.], v.5, n.1, p.1–263, 2010.

HERLIHY, M.; ELIOT, J.; MOSS, B. Transactional Memory: Architectural Support for Lock-Free Data Structures. In: PROC. OF THE 20TH ANNUAL INTL. SYMPOSIUM ON COMPUTER ARCHITECTURE, 1993. **Anais...** [S.l.: s.n.], 1993. p.289–300.

MINH, C. C.; CHUNG, J.; KOZYRAKIS, C.; OLUKOTUN, K. STAMP: Stanford Transactional Applications for Multi-Processing. In: WORKLOAD CHARACTERIZATION, 2008. IISWC 2008. IEEE INTERNATIONAL SYMPOSIUM ON, 2008. **Anais...** [S.l.: s.n.], 2008. p.35–46.

MORESHET, T.; BAHAR, R. I.; HERLIHY, M. Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks. In: WORKSHOP ON MEMORY PERFORMANCE ISSUES, 2006. **Proceedings...** [S.l.: s.n.], 2006.

NICÁCIO, D.; BALDASSIN, A.; ARAÚJO, G. Transaction Scheduling Using Dynamic Conflict Avoidance. **International Journal of Parallel Programming**, [S.l.], v.41, n.1, p.89–110, 2012.

PASQUALIN, D. P.; DIENER, M.; DU BOIS, A. R.; PILLA, M. L. Online Sharing-Aware Thread Mapping in Software Transactional Memory. In: IEEE 32ND INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p.35–42.

PASQUALIN, D. P.; DIENER, M.; DU BOIS, A. R.; PILLA, M. L. Thread affinity in software transactional memory. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED COMPUTING (ISPDC), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p.180–187.

RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. **Memorias Transacionais**: Uma Nova Alternativa para Programação Concorrente. [S.l.]: In Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing, 2007.

RITO, H.; CACHOPO, J. Adaptive transaction scheduling for mixed transactional workloads. **Parallel Computing**, [S.l.], v.41, p.31–49, 2015.

RUPPERT, J. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. **J. Algorithms**, Duluth, MN, USA, v.18, n.3, p.548–585, May 1995.

YOO, R. M.; LEE, H.-H. S. Adaptive transaction scheduling for transactional memory systems. In: PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 2008. **Proceedings...** [S.l.: s.n.], 2008. p.169–178.