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

Dissertação

Escalonador de Transações para Arquiteturas NUMA

Michael Alexandre Costa

Pelotas, 2020

Michael Alexandre Costa

Escalonador de Transações para Arquiteturas NUMA

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, 2020

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. **Escalonador de Transações para Arquiteturas NUMA**. Orientador: André Du Bois. 2020. 49 f. Dissertação (Mestrado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2020.

...

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

ABSTRACT

COSTA, Michael Alexandre. **Transaction Scheduler for NUMA Architectures**. Advisor: André Du Bois. 2020. 49 f. Dissertation (Masters in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2020.

...

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

1.1 Motivação

... (**?**).

1.2 Objetivos

... 1.

1.2.1 Objetivo geral

...

1.2.2 Objetivos especificos

- ...; e
- ...

1.3 Estrutura do Texto

... Teste do overleaf. ...

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.

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).

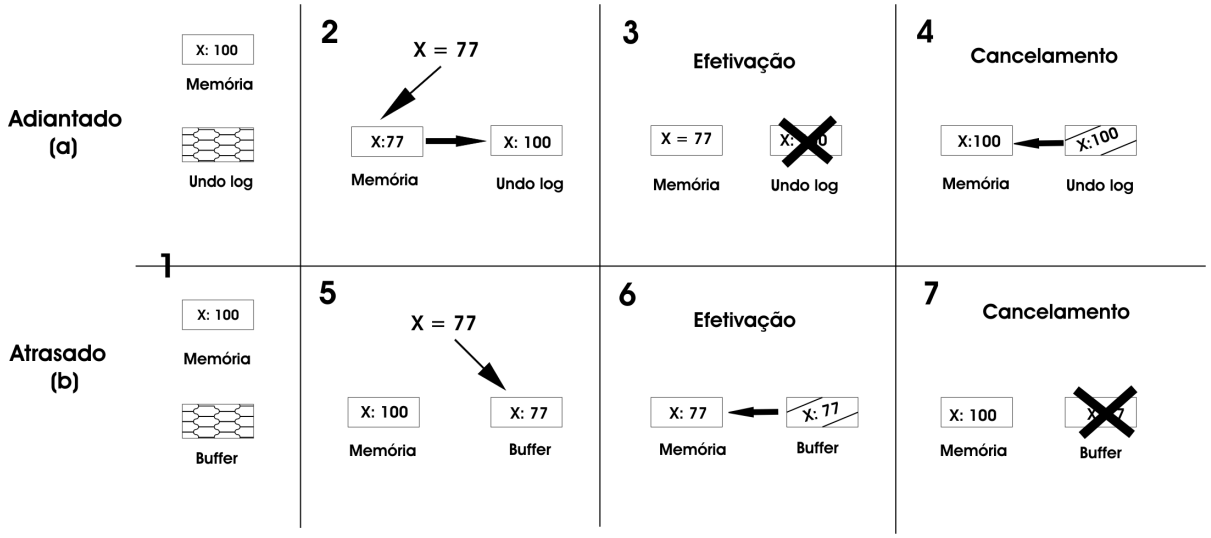


Figura 1 – Exemplo de versionamento adiantado (a) e atrasado (b). Fonte: (BALDASSIN, 2009)

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.

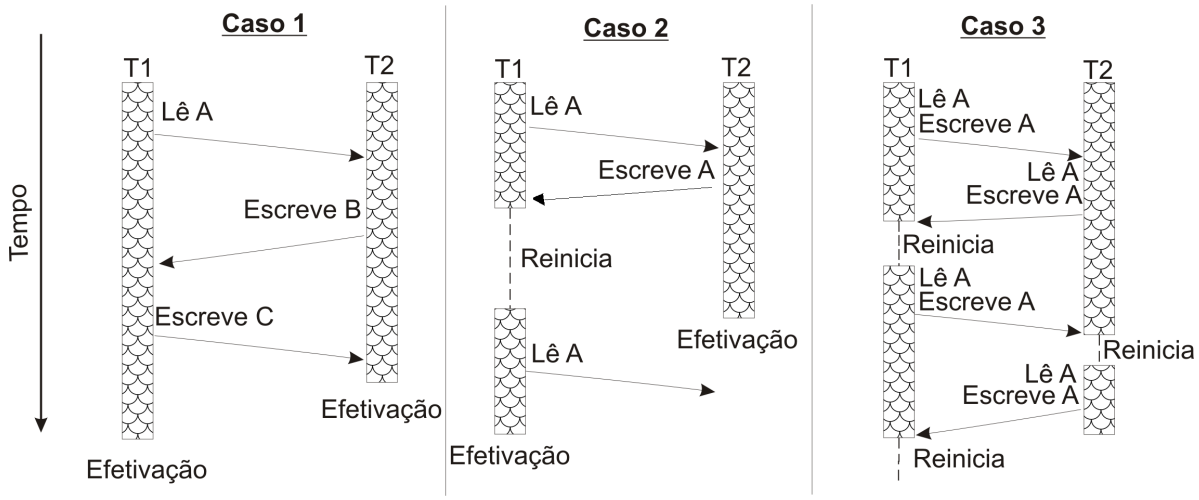


Figura 2 – Detecção de conflitos em modo adiantado. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

- **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.

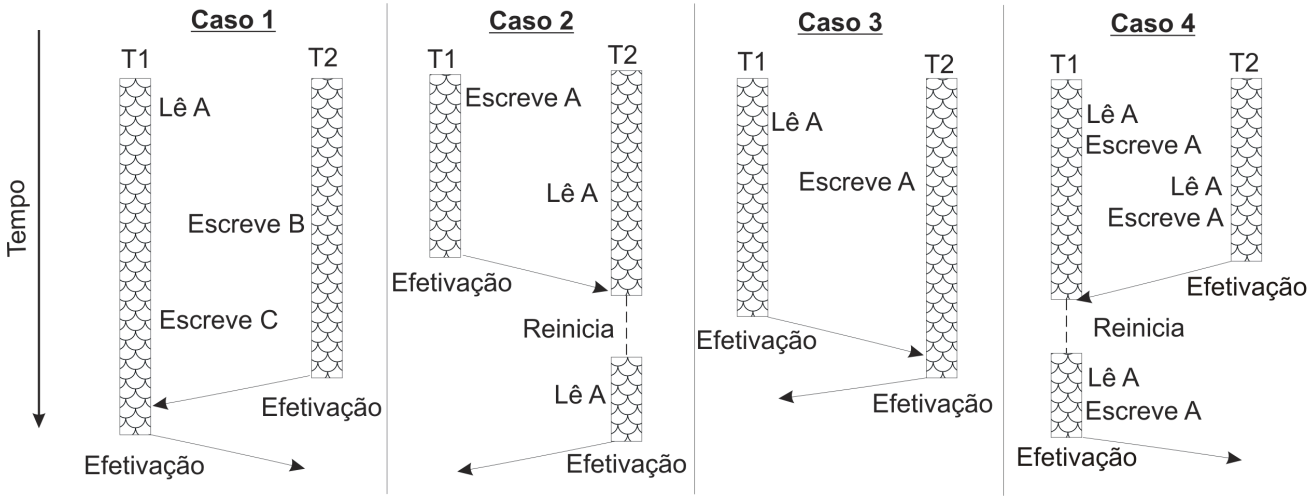


Figura 3 – Detecção de conflitos em modo atrasado. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

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.

3 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*.

3.1 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 endereço 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.

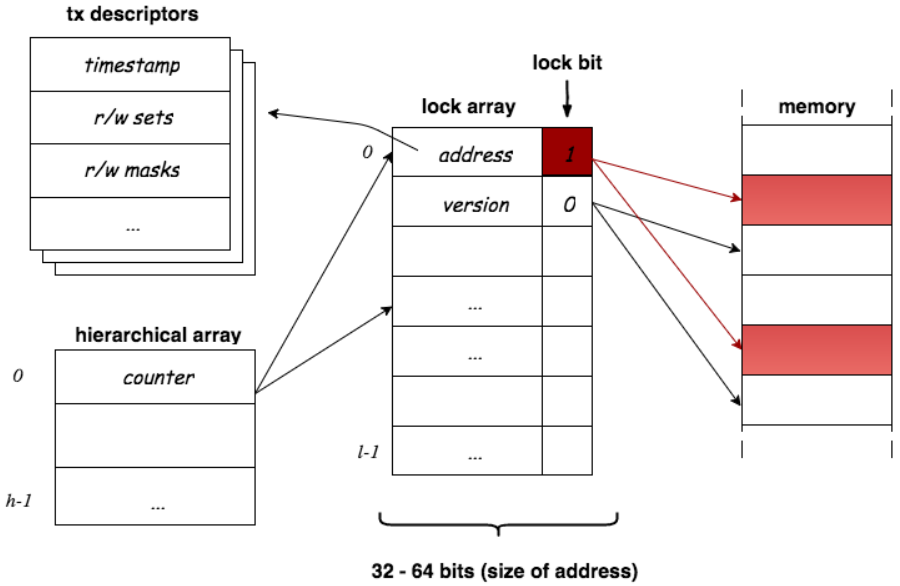


Figura 4 – Estruturas de dados utilizadas na *tinySTM*. Fonte: (FELBER; FETZER; RIEGEL, 2008)

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 *commit 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.

3.2 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.

3.3 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.

3.4 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.

3.5 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 ten-

tativa. 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.

4 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, em (NICÁCIO; BALDASSIN; ARAÚJO, 2012) foi proposto um escalonador de transações dinâmico denominado *LUTS*, este 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. Para este trabalho foram estudados algumas heurísticas que serviram como base para o escalonamento de transações, estes trabalhos são:

4.1 Categorias

Di zando apresenta uma categorização dos escalonadores de STM, onde ele classifica os algoritmos de acordo com as heurísticas apresentadas por eles.

Está classificação é dividica por Baseada em Heurística e Baseado em Modelo.

- Baseado em Heurística:
  - Feedback;
  - Predição;
  - Reativo; e
  - Heurística Mista.
- Baseado em Modelo:
  - Aprendizado de Máquina;

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

- Modelo Analítico; e
- Modelo Misto.

A tabela 2 apresenta a caracterização dos algoritmos revizados na bibliografia

4.1.1 ATS

Adaptive Transaction Scheduling (ATS) (YOO; LEE, 2008) apresenta uma lista global onde é inserida todas transações conflitantes, assim o escalonador garante que será executada apenas uma transação por vez.

4.1.2 Blake

Blake (BLAKE; DRESLINSKI; MUDGE, 2009) apresenta um escalonador proativo com gerenciamento de conflito, esta previsão ocorre através do armazenamento de um valor de confiança que indica a probabilidade de ocorrência do conflito, assim, o escalonador utiliza do valor de confiança para decidir entre executar a transação, esperar ou executar outra transação.

4.1.3 CAR-STM

CAR-STM (DOLEV; HENDLER; SUISSA, 2008) mantém uma fila e uma *thread* para cada núcleo disponível na máquina, o escalonador seleciona uma das filas e insere a transação que esta prestes a iniciar, após isto, passa o controle para a *thread* na qual esta fila pertence. A idéia principal é que durante o tempo de execução o escalonador insira as transações abortadas na fila da transação conflitante, assim, reduzindo o número de *aborts* através da serialização destas transações.

4.1.4 LUTS

LUTS apresenta um escalonador de transações em nível de usuário, este apresenta uma heurística proativa que usa o escalonador para evitar o início de transações

conflitantes, escolhendo na fila de tarefas uma transação menos suscetível ao conflito.

4.1.5 Shirink

Shirink (DRAGOJEVIć et al., 2009) é uma técnica de detecção de conflitos, este evita inicializar transações com maiores chances de *abort*. Esta previsão toma como base os acessos realizados anteriormente pelas transações, porém a técnica é ativada depois que um determinado número de *aborts* ocorram, assim, evitando *overhead* de execução.

5 ARQUITETURAS

Dependendo da localização física da memória em relação aos processadores, o tempo de acesso a uma posição de memória pode ser uniforme ou não. Surge então as arquiteturas denominadas de *UMA (Uniform Memory Access)* e *NUMA (Non Uniform Memory Access)* (CARISSIMI et al., 2007).

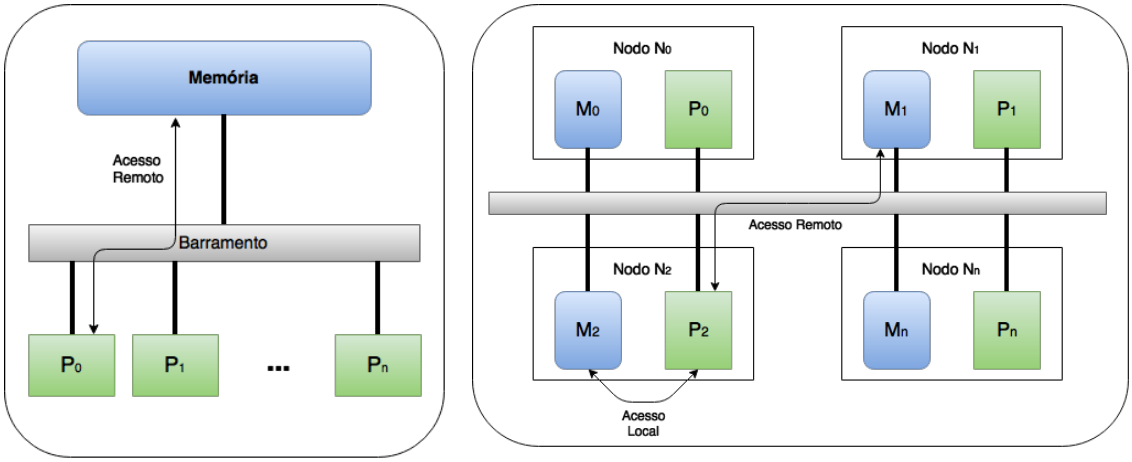
Máquinas *NUMA* tem a vantagem de agregar maior paralelismo ao adicionar mais processadores sem aumentar o número de conflitos e o gargalo de acesso ao barramento. Sua arquitetura é feita para que os processadores não utilizem o mesmo barramento de acesso à memória como é feito em arquiteturas *UMA*.

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). A Figura 5 ilustra de maneira genérica, 5(a) uma máquina de arquitetura *UMA* e 5(b) uma máquina de arquitetura *NUMA*. Pode-se observar que, na arquitetura *UMA* representada na Figura 5(a), as operações relacionadas à memória (leituras e escritas), independente da unidade de processamento *(P0, P1, ..., Pn)* que estão partindo, possuem o mesmo tempo de acesso, pois todos acessam a memória utilizando o mesmo caminho.

Já na Figura 5(b), existem dois possíveis e distintos caminhos para acessar à memória, dependendo de qual unidade de processamento irá partir o acesso e da localização física da memória a ser acessada: (i) Acesso Local (*P2* acessando à memória *M2*) e (ii) Acesso Remoto (*P2* acessando à memória *M1*). Os tempos de acessos à memória local de um processador, tanto para leitura quanto para escrita de dados e instruções, são menores do que os tempos de acessos às memórias remotas. Neste tipo de arquitetura, os acessos a memória ocorrem de maneira não uniforme, essas

assimetrias no acesso caracterizam este tipo de arquitetura.



(a) Arquitetura UMA (b) Arquitetura NUMA

Figura 5 – Configuração das arquiteturas UMA e NUMA. Fonte: (FAVARETTO, 2014)

O custo não uniforme de acesso à memória é importante para o desempenho de uma aplicação, pois este pode oferecer desvantagens quando não forem avaliados. Uma forma bastante comum de medir a não uniformidade de acessos à memória nas arquiteturas assimétricas, é considerar a razão entre o acesso a uma posição de memória remota e um acesso a uma posição de memória local. Esta razão é denominada fator *NUMA*, uma máquina pode apresentar diferentes valores para o fator *NUMA*, dependendo de quais memórias serão acessadas pelos processadores. Portanto, para que uma aplicação execute de forma eficiente é desejável que os processos sejam escalonados próximos a faixa de endereçamento que eles acessam, assim reduzindo o fator *NUMA*.

É importante conhecer as características da arquitetura *NUMA*, principalmente a latência de acesso às memórias remotas. Estas informações servem de subsidio ao desenvolvedor para que este consiga definir estratégias de escalonamento de tarefas mais eficientes para aplicações que executam neste tipo de arquitetura.

5.1 NUMA-Aware

Arquiteturas *NUMA* apresentam boas características para aplicações com alto nível de paralelismo, porém, os programadores precisam ter cuidado ao escrever códigos para estas arquiteturas. Com isto surgiu o conceito de *NUMA-Aware*, que aborda maneiras para que os programas utilizem com total eficiência os recursos da máquina.

O conceito de *NUMA-Aware* esta relacionado a melhorias em programas, como visto em (MOHAMEDIN et al., 2016), pode-se ter melhorias de desempenho a partir gerenciador de contenção que trata conflitos ocorridos entre nodos de forma diferente dos conflitos ocorridos em um único nodo. Também pode-se obter melhor desempe-

nho das arquiteturas com um escalonador de tarefas, no qual o escalonador considera o endereço de memória que a tarefa acessará, assim reduzindo o tempo de leitura e escrita das tarefas executadas pelo programa (BLAGODUROV et al., 2010).

5.2 HwLoc

Para extrair o melhor desempenho de máquinas *NUMA* é necessário conhecer a arquitetura que está sendo utilizada, para obter tais informações de maneira eficiente podemos utilizar o conjuntos de ferramentas *HwLoc (Hardware Locality)*.

HwLoc é um conjunto de ferramentas que possui sua API escrita em linguagem C, esta tem por finalidade obter o mapa hierárquico dos principais elementos presentes na arquitetura de uma máquina, como os níveis de memória *cache*, nodos *NUMA*, soquete, núcleos de processamento e dispositivos de *I/O*.

Esta ferramenta suporta uma variedade de sistemas operacionais e plataformas, além disto, pode agrupar as topologias de várias máquinas em uma única, com a finalidade de permitir que os aplicativos consigam ter uma visão geral da topologia de um conjunto de máquinas, como por exemplo em um *cluster*.

A topologia da máquina compreende todas as informações que podem ser coletadas sobre o *hardware* em que a aplicação está sendo executada, como a localização física de um núcleo de processamento e suas interligações dentro de um determinado processador ou nó de processamento.

As arquiteturas estão tornando-se cada vez mais complexas, com vários níveis de memória *cache*, podendo conter blocos de *cache* específicos (um para cada processador), blocos de *cache* globais (um para todos os processadores) ou ainda alguns blocos de *cache* parcialmente compartilhados (um bloco compartilhado entre alguns processadores).

Estas informações da topologia da máquina precisam ser conhecidas durante a execução do programa para extrair o máximo de desempenho. O *HwLoc* obtém as informações da topologia e as disponibiliza ao programador de diversas maneiras. A topologia da máquina pode ser armazenada em uma representação gráfica ou em outros formatos como texto, xml, pdf, entre outros. Pode ainda ser obtida em tempo de execução com o uso da interface que a ferramenta disponibiliza.

Para obter o máximo possível de informações sobre o *hardware* da máquina, a ferramenta *HwLoc* foi estendida por (PILLA et al., 2014), para que fornecesse uma visão genérica e completa de qualquer arquitetura de computador. O modelo da arquitetura foi melhorado, passando a oferecer estatísticas de memória, como por exemplo os tempo de latência para buscar dados a partir de diferentes níveis de cache e memória compartilhada e também estatísticas das interconexões, como por exemplo a largura de banda para o tráfego dos dados. Em outras palavras, a ferramenta passou

a fornecer informações da distância entre os núcleos de processamento da máquina com base nos tempos de latência nos acessos à memória e largura de banda para a transmissão de dados.

Essas informações são geradas em forma de árvore, a qual é composta pela máquina na raiz e os processadores nas folhas, contendo todos os níveis de cache nos níveis intermediários. As principais vantagens da utilização deste modelo como descrição da topologia da máquina são: (i) modelo genérico, pode ser facilmente calculado por diferentes máquinas e bibliotecas de programação, (ii) agrega as diferentes características de máquinas *multicore*. Além disso, as informações de hierarquia de *cache* e latência de acesso a memória podem ser pré-computadas e armazenadas antes da execução da aplicação, o que reduz o *overhead*, uma vez que elas não precisam ser calculadas novamente toda vez que uma aplicação for executada (PILLA et al., 2014).

6 SHRINK

...

7 STAMP

STAMP (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*. A versão do STAMP utilizada será a 0.9.10. O conjunto de *benchmarks* STAMP foi escolhido devido a ele implementar vários *benchmarks*, assim, atingindo uma maior área de aplicações das STM além de ser o conjunto de *benchmark* mais utilizado na pesquisa de STM.

Os *benchmarks* implementados pelo STAMP são (MINH et al., 2008):

7.1 Bayes

Esta aplicação implementa um algoritmo de aprendizado de redes Bayesianas, que é uma parte importante do aprendizado de máquina. Normalmente, nem as distribuições de probabilidades nem as dependências condicionais entre eles são conhecidas ou podem ser resolvidos por um ser humano, assim redes Bayesianas são frequentemente estudadas com os dados observados. O algoritmo específico implementa uma estratégia de *hill-climbing* ou subida de encosta que usa buscas locais e globais, semelhante à técnica descrita em (CHICKERING; HECKERMAN; MEEK, 1997). Para estimativas eficientes de distribuição de probabilidade, utiliza-se uma *adtree* ou árvore de decisão a partir de (MOORE; LEE, 1997).

7.2 Genome

Este *benchmark* implementa um programa de sequenciamento de genes que reconstrói a sequência de genes a partir de sequências maiores. O algoritmo usado para o sequenciamento de genes têm três fases:

1. Remove os segmentos duplicados utilizando uma *hash*;
2. Combina segmentos utilizando o algoritmo de pesquisa de sequência *Rabin-*

Karp (KARP; RABIN, 1987); e

3. Constrói a sequência.

7.3 Intruder

Este *benchmark* simula o Design 5 dos NIDS (*Network Intrusion Detection System*) descritos por Haagdorens em (HAAGDORENS; VERMEIREN; GOOSSENS, 2005). Pacotes de rede são processados paralelamente e passam por três fases: captação, remontagem e detecção. A estrutura de dados principal na fase de captura é uma simples fila, e a fase de remontagem utiliza um dicionário (implementado por uma árvore auto balanceada), que contém a lista de pacotes que pertencem à mesma seção. Ao avaliar seus cinco designs para um NIDS *multithread*, Haagdorens afirma que a complexidade da fase de remontagem fez com que ele utilize a sincronização de grãos grosso nos designs 4 e 5. Assim, embora estes dois modelos tentam explorar níveis mais elevados de simultaneidade, a sincronização aproximada de grão resulta em um pior desempenho.

7.4 Kmeans

Este *benchmark* foi tirado do *NU-MineBench 2.0* (PISHARATH et al., 2005). *K-means* é um método baseado em partição (BEZDEK, 1981) e é sem dúvida a técnica de agrupamento mais utilizada. Este algoritmo é comumente usado para partição de itens de dados em subconjuntos relacionados. Cada *thread* processa uma partição dos objetos iterativamente. A versão transacional adiciona uma transação para proteger o update do centro do *cluster* que ocorre durante cada iteração.

7.5 Labyrinth

Dado um labirinto, este *benchmark* encontra os caminhos de menor distância entre os pares de pontos inicial e final. O algoritmo de roteamento utilizado é o algoritmo Lee (LEE, 1961).

Nesse algoritmo, o labirinto é representado como uma grade, em que cada ponto de grade pode conter ligações adjacentes, para os pontos da grade que não estão nas diagonais. O algoritmo busca um caminho mais curto entre os pontos de conexão através da realização de uma busca em largura e marca cada ponto da grade com a sua distância para o início. Esta fase de expansão acabará por chegar ao ponto final, se a conexão for possível. A segunda fase de rastreamento, em seguida, estabelece a ligação, seguindo todo o caminho diminuindo a distância. Este algoritmo é garantido

para encontrar o caminho mais curto entre um ponto inicial e final, no entanto, quando vários caminhos são feitos, um caminho pode bloquear outro.

7.6 SSCA2

Scalable Synthetic Compact Applications 2 (SSCA2) (BADER; MADDURI, 2005) é composta por quatro *kernels* que operam em um grande, dirigido e ponderado gráfico. Estes quatro *kernels* gráficos são comumente usados em aplicações que vão desde a biologia computacional até a segurança. STAMP incide sobre um *Kernel*, que constrói uma estrutura de dados eficiente utilizando matrizes de adjacência e matrizes auxiliares.

7.7 Vacation

Este *benchmark* implementa um sistema de reserva de viagens alimentado por um banco de dados não-distribuído. A carga de trabalho é composto por vários segmentos de clientes que interagem com o banco de dados via gerenciador de transações do sistema.

O banco de dados é composto por quatro tabelas: carros, quartos, voos e clientes. Os três primeiros têm relações com os campos que representam um número único de identificação, quantidade reservada, a quantidade total disponível, e preço. A tabela de clientes acompanha as reservas feitas por cada cliente e o preço total das reservas que eles fizeram. As tabelas são implementados como árvores rubro negras.

7.8 Yada

Este *benchmark* implementa o algoritmo de Ruppert para refinamento de malha (RUPPERT, 1995). A versão transacional é similar em design ao apresentado em (KULKARNI; CHEW; PINGALI, 2006).

8 METODOLOGIA

...

9 DESENVOLVIMENTO

Para este trabalho foi desenvolvido um escalonador de memórias transacionais intitulado LStm que identifica as características da arquitetura e do programa em tempo de execução.

Este escalonador foi desenvolvido para biblioteca de STM TinyStm, com base no trabalho intitulado STMap **To do (??)**. O STMap fornece uma matriz de comunicação das threads, esta matriz é montada em tempo de execução com base nas leituras e escritas realizadas.

Essa matriz de comunicação é utilizada nas heurísticas do escalonador LStm junto com outros dados coletados pelo LStm. Estas heurísticas buscam melhorar a execução do programa após a ocorrência dos conflitos entre as transações.

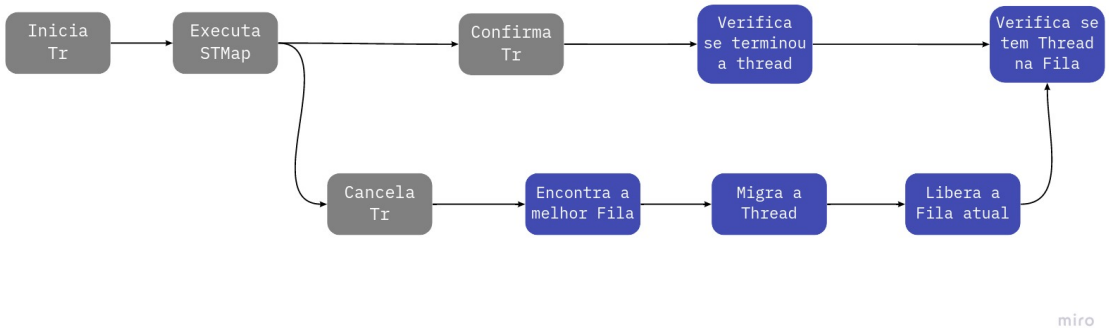


Figura 6 – Fluxo de execução da LStm

9.1 LStm

O escalonador desenvolvido denominado LStm é dividido em três etapas. As etapas são inicialização do sistema, execução e coleta de dados, e migração das threads em execução.

A primeira etapa, inicialização do sistema, como o nome sugere ocorre no inicio da aplicação e é responsável por identificar a arquitetura utilizada, montar as filas de execução e distribuir as threads entre estás filas.

No inicio desta etapa o escalonador lê a arquitetura utilizada para saber a quantidade de cores disponível na arquitetura, se o número de thread solicitado pela aplicação for menor que o número de cores disponível na arquitetura o escalonador cria uma fila de execução para cada thread solicitada. Se o número de threads solicitados for maior que o número de cores da arquitetura, é criada uma fila de execução para cada core da arquitetura.

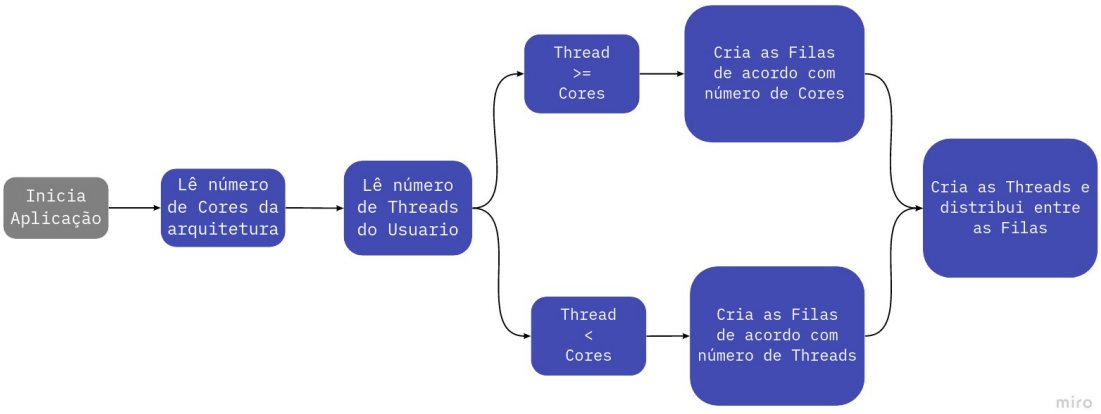


Figura 7 – Inicialização da LStm

Após as filas de execução serem criadas, o escalonador se encarrega de distribuir com base em uma heurística esses threads recém criados entre as filas de execução. Temos duas heurísticas utilizadas para distribuição das threads entre as filas.

A primeira heurística de distribuição resume-se a distribuir de forma ordenada uma thread para cada fila de execução, repetindo a distribuição caso haja mais threads do que filas...

A segunda heurística de distribuição resume-se a distribuir de forma ordenada duas threads para cada fila de execução, repetindo a distribuição caso haja mais threads do que filas...

Na segunda etapa a aplicação é realizada, cada thread pode ter n transações. Nessa etapa o escalonador utiliza os recursos desenvolvidos em STmap para coletar as informações sobre as transações executadas nessas threads.

Aqui é montada a matriz de comunicação que indica o índice de leitura e escrita uma thread possui em relação as demais. Também coletamos as informações sobre o índice de commits e aborts realizados por thread e montamos a matriz com os últimos endereços de memória acessados pelas threads.

A terceira etapa só é executada se uma transação abortar. Nesta etapa avaliamos para qual fila a thread em execução pode ser migrada, e utilizamos uma heurística para tomar a decisão de migrar a thread de sua fila de execução para outra fila.

No momento do abort consultamos a matriz de comunicação previamente montada para descobrir com qual outra thread temos um maior número de leitura e escrita em comum. Após descobrir qual é esta thread utilizamos esse valor para descobrir qual sua fila de execução atual, é para esta fila que o escalonador poderá efetuar a migração.

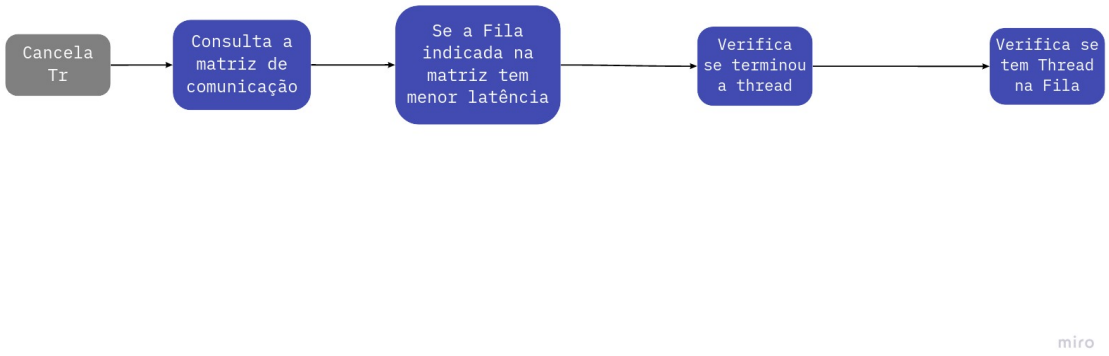


Figura 8 – Migração de threads na LStm

Caso a thread atual não esteja na mesma fila que a outra thread o escalonador avalia a possibilidade de migração desta thread. Para tomar a decisão de executar ou

não a migração foi implementada duas heurísticas distintas. A primeira chamada latency tem como base avaliar a latência de acesso à memória, a segunda denominada threshold avalia a relação entre commits e aborts existentes na thread em execução.

Para realizar a heurística latency primeiro consultamos a matriz com o endereço de memória do ultimo acesso em comum entre as duas threads e então consultamos a qual região de memória esse endereço pertence. Após isto é coletada a informação de qual nodo a fila que a thread esta executando pertence e qual nodo pertence a fila que receberá o thread.

Se a latência entre a fila atual e posição de memória for maior que a latência entre a nova fila e a posição de memória o escalonador se encarrega de adormecer a thread atual e migrar ela para nova fila de execução.

Para executar a heurística threshold o escalonador avalia qual a diferença entre abort e commit existente na thread em execução. Caso o valor da diferença fique acima de um limiar estipulado o escalonador se encarrega de adormecer a thread em execução e migrá-la para nova fila.

Após a heurística ser executada e a migração ter sido realizada, o escalonador avalia se existe mais threads disponíveis para execução na fila de execução. Se a migração não for executada a thread realiza um processo de abort padrão da biblioteca de stm.

10 CONCLUSÃO

...

10.1 Resultados

...

REFERÊNCIAS

BADER, D. A.; MADDURI, K. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In: HIGH PERFORMANCE COMPUTING, 12., 2005, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2005. p.465–476. (HiPC'05).

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.

BEZDEK, J. C. **Pattern Recognition with Fuzzy Objective Function Algorithms**. Norwell, MA, USA: Kluwer Academic Publishers, 1981.

BLAGODUROV, S.; ZHURAVLEV, S.; FEDOROVA, A.; KAMALI, A. A Case for NUMA-aware Contention Management on Multicore Systems. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 19., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.557–558. (PACT '10).

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T. Proactive Transaction Scheduling for Contention Management. In: ND ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 42., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.156–167. (MICRO 42).

CARISSIMI, A.; DUPROS, F.; MÉHAUT, J.-F.; POLANCZYK, R. V. Aspectos de Programação Paralela em Máquinas NUMA. In: MINICURSO DO WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 2007. **Anais...** [S.l.: s.n.], 2007.

CHICKERING, D. M.; HECKERMAN, D.; MEEK, C. A Bayesian approach to learning Bayesian networks with local structure. In: IN PROCEEDINGS OF THIRTEENTH

CONFERENCE ON UNCERTAINTY IN ARTIFICIAL INTELLIGENCE, 1997. **Anais...** Morgan Kaufmann, 1997.

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.

KARP, R. M.; RABIN, M. O. **Efficient randomized pattern-matching algorithms**.

KULKARNI, M.; CHEW, L. P.; PINGALI, K. Using Transactions in Delaunay Mesh Generation. In: WTW'06: PROCEEDINGS OF THE WORKSHOP ON TRANSACTIONAL MEMORY WORKLOADS, 2006, Ottawa, Canada. **Anais...** [S.l.: s.n.], 2006. p.23–31. (Held in conjunction with PLDI 2006).

LEE, C. Y. An Algorithm for Path Connections and Its Applications. **IRE Transactions on Electronic Computers**, [S.l.], v.EC-10, n.3, p.346–365, Sept 1961.

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.

MOHAMEDIN, M.; PALMIERI, R.; PELUSO, S.; RAVINDRAN, B. On Designing NUMA-aware Concurrency Control for Scalable Transactional Memory. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 21., 2016, New York, NY, USA. **Proceedings...** ACM, 2016. p.45:1–45:2. (PPoPP '16).

MOORE, A.; LEE, M. S. Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets. **Journal of Artificial Intelligence Research**, [S.l.], v.8, p.67–91, 1997.

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.

PILLA, L. L. et al. A topology-aware load balancing algorithm for clustered hierarchical multi-core machines. **Future Generation Computer Systems**, [S.l.], v.30, p.191 – 201, 2014. Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, {ICPADS} 2012 Selected Papers.

PISHARATH, J. et al. **NU-MineBench**: Understanding the Performance and Scalability Characteristics of Data Mining Algorithms.

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.

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.

Apêndices

APÊNDICE A – Um Apêndice

Anexos

ANEXO A – Um Anexo

...

ANEXO B – Outro Anexo

...