LTMS: Um escalonador NUMA-Aware para STM

Michael Alexandre Costa[1], **Tiago Perlin**[1,2], **André Rauber Du Bois**[1], **Gerson Cavalheiro**[1]

[1]Programa de Pós-Graduação em Computação – Universidade Federal de Pelotas (UFPel)

[2]Instituto Federal Farroupilha (IFFar)

{macosta,tiago.perlin,dubois,gerson.cavalheiro}@inf.ufpel.edu.br

Resumo. *As transações em sistemas com Memória Transacional são, usualmente, realizadas de forma otimista, de forma a promover a exploração do paralelismo do hardware, mas aumentando a probabilidade de conflitos nos acessos. Neste trabalho é proposta uma estratégia de escalonamento baseada na migração de threads entre núcleos de processamento, apoiada pela observação da localidade de referência de acesso à memória dos threads em execução. Em experimentos realizados com o benchmark STAMP, para a maioria das aplicações, o escalonador proposto produziu menor taxa de aborts e menor tempo de execução em função do agrupamento de threads conforme a possibilidade de conflitos e o custo no acesso à memória.*

1. Introdução

As arquitetura paralelas são comumente encontradas em plataformas computacionais de alto desempenho atuais. Considerando-se a organização da memória, existem dois tipos básicos, arquiteturas UMA (*Uniform Memory Access*) e NUMA (*Non-Uniform Memory Access*). Enquanto nas arquiteturas UMA o espaço de endereçamento da memória é visto como monolítico, no qual um único barramento oferece acesso a ela a todos os *cores* (unidades de processamento) da máquina, nas arquiteturas NUMA, conjuntos de *cores* são organizados em nós, cada nó dispondo de um segmento da memória total da máquina, havendo uma malha de interconexão entre os nós da máquina. O espaço de endereçamento se mantém como contínuo a todos os *cores*, no entanto, devido a interconexão entre os nós, o tempo de acesso à memória depende do *core* e do endereço requisitado. A vantagem das arquiteturas NUMA é sua escalabilidade em termos de *core* em relação às arquiteturas UMA. No entanto, ambientes de execução devem atentar para, na distribuição da carga computacional, considerar também a localidade de acesso dos threads/processos com vistas a explorar positivamente a localidade de dados.

Os recursos de sincronização mais populares na programação concorrente tradicional oferecem estratégias para controle a seções críticas baseados em *locks* para garantir a integridade dos dados quando sujeitos a acessos concorrentes. Entretanto, além de complexos, esses mecanismos podem gerar problemas como impasses (*deadlocks*), condições de corrida, violações de atomicidade e violações de ordem [Adl-Tabatabai et al. 2006], que são difíceis de expor [Owens 2010], detectar [Baek et al. 2007, Baeza-Yates and Ribeiro-Neto 1999, Bienia et al. 2008], depurar [Bienia et al. 2008], e reparar [Cascaval et al. 2008]. Como alternativa, apersenta-se a Memória Transacional (*Transactional Memory* - TM), que é um mecanismo de sincronização que permite a execução atômica de regiões críticas no código que contém posições compartilhadas de memória. Na programação com TM não há a preocupação com aquisições e liberações de *locks*, o que facilita o desenvolvimento e diminui a probabilidade

de erros. Uma TM pode ser implementada em *software*, *hardware* ou híbrido, sendo que, neste trabalho, o foco está na TM em *Software* (*Software Transactional Memory* - STM).

Aplicações multithread com STM costumam apresentar maior contenção devido ao maior número de conflitos e *aborts* em transações causados pelo aumento do paralelismo. Buscando reduzir a possibilidade de conflitos, muitos trabalhos concentraram-se no desenvolvimento de escalonadores que limitam o número de threads ativas ou serializam suas execuções [Yoo and Lee 2008, Dolev et al. 2008, Dragojević et al. 2009, Nicácio et al. 2012, Rito and Cachopo 2015]. Já outros, como [Pasqualin et al. 2020], investigam a distribuição de threads, considerando-se a arquitetura da máquina, e um contexto transacional, buscando-se melhoria de desempenho.

Neste trabalho é apresentado um escalonador de transações que implementa uma heurística para diminuir o número de conflitos enquanto utiliza atributos da arquitetura para alocação de threads sobre os núcleos de processamento. O escalonador proposto – LTMS (*Lups Transactional Memory Schedule* –, opera em em três etapas. Na etapa de inicialização são criadas filas de acordo com a aplicação e a arquitetura, distribuindo inicialmente os threads igualmente entre os *cores* disponíveis. A segunda etapa extrai informações sobre a arquitetura em que o programa encontra-se em execução e, em tempo de execução, registra informações que permitem identificar o histórico de acesso aos dados do programa pelos threads e transações. A terceira etapa é ativada quando da ocorrência de conflitos entre transações: com as informações de acesso aos dados do programa, os threads conflitantes são redistribuídos entre os *cores* com uma heurística para reduzir o número de conflitos futuros enquanto busca uma melhor localidade dos dados.

As principais contribuições deste artigo são: o projeto de um escalonador de STM modular que considera a arquitetura utilizada no escalonamento das transações e a análise de desempenho do LTMS utilizando o conjunto de *benchmarks* STAMP [Minh et al. 2008]. O escalonador foi desenvolvido sobre a biblioteca TinySTM [Felber et al. 2008]. A análise de desempenho conduzida comparou o desempenho de LTMS com o apresentado pelo escalonador padrão da TinySTM. Os resultados mostraram considerável número na redução de *aborts* na execução dos programas STAMP, chegando em alguns casos à 99%, reduzindo o tempo total de execução em consequência.

Este texto está assim organizado: a Seção 2 apresenta uma visão geral sobre Memórias Transacionais e escalonadores de transações e a Seção 3 o escalonador proposto. Na Seção 4 é apresentada uma análise do seu desempenho e a Seção 5 conclui o trabalho.

2. Memórias Transacionais e escalonadores de transações

Na programação com memórias transacionais, todo o acesso a variáveis compartilhadas deve ser realizado dentro de blocos delimitados denominados transações, executadas atomicamente entre si. Assim as propriedades de transações são: atomicidade, consistência e isolamento. A atomicidade garante que cada transação seja vista como um bloco único, que pode ser executado por completo ou não executado, quando a transação é confirmada (*commit*) ou abortada (*abort*). Esta propriedade é conseguida por meio da execução provisória de mudanças na memória compartilhada, sendo as alterações na memória confirmadas quando a transação finaliza em *commit* ou descartada quando ocorre *abort*. A consistência dos dados compartilhados e do sistema é garantida quando o dado confirmado após a transação reflete o cálculo realizado considerando o valor da variável

anotado no início da transação. Por fim, o isolamento indica que uma transação não pode observar ou alterar o estado intermediário de um dado manipulado por outra transação, não interferindo na execução de outra transação. Para garantir essas propriedades implementações de memória transacional utilizam mecanismos de Versionamento de Dados e de Detecção de Conflitos [Guerraoui and Romano 2015].

Um problema atual em para aplicações executando com STM é a perda de desempenho originada quando há quantidade excessiva de *aborts*. Para solucionar este problema, alguns estudos propõem o uso de escalonadores para controlar o número de threads ativas, com o objetivo de reduzir a probabilidade de conflitos, e por consequência, de *aborts*. Oferecendo maior fluidez de execução aos *threads* remanescentes, a expectativa é reduzir o tempo total de execução. A seguir, são apresentados os principais trabalhos da área de escalonadores de transações para Memórias Transacionais, uma visão mais aprofundada da área pode ser encontrada em [Di Sanzo 2017].

Adaptive Transaction Scheduling (ATS) [Yoo and Lee 2008] sendo um dos primeiros trabalhos a propor um escalonador para TM, a abordagem emprega um índice calculado em tempo de execução para determinar a entrada de uma ação de escalonamento. Este índice é aplicado a cada transação e reflete os números de *commits* e *aborts* que estas sofrem. Ao atingir um determinado valor, este índice indica a necessidade de serializar a execução do *thread* responsável por uma determinada transação.

Collision Avoidance and Resolution (CAR-STM) [Dolev et al. 2008] concebido para evitar que conflitos já observados voltem a ocorrer, este escalonador conta com duas heurísticas de gerenciamento. A primeira busca executar de forma serial transações conflitantes sem manter um histórico da execução. Quando detectado um conflito, a transação mais recente é abortada e alocada sobre uma fila de conflitos. Sua execução é serializada em relação à transação mais antiga com a qual o conflito foi observado. A segunda heurística mantém um histórico das transações que conflitaram, forçando uma execução serial em cadeia: 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*.

Light-Weight User-Level Transaction Scheduler (LUTS) [Nicácio et al. 2012] é um escalonador que busca evitar a ociosidade de um núcleo após a serialização de uma transação. A heurística empregada deferencia transações curtas de transações longas. Para transações curtas, é calculada a intensidade de conflito da transação e esta é serializada quando o valor obtido ultrapassa um limiar. Outra transação é selecionada para substituir a atual. Para transações longas, a heurística utiliza 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 início de cada transação o LUTS percorre a *bestTx* e seleciona qual transação deve executar. Quando a *conflictTable* é atualizada o escalonador atualiza a *bestTx*.

Shrink [Dragojević et al. 2009] é um escalonador que busca minimizar a ocorrência de *aborts* com base nos conjuntos de leituras e escritas de cada thread. O escalonador é baseado em predição e explora uma heurística que leva em conta o histórico de acessos à memória das transações executadas anteriormente.

O escalonador *ProVIT* [Rito and Cachopo 2015] fornece uma abordagem otimista ao considerar que uma transação ao abortar, não irá abortar novamente na sequência. É considerado o tamanho das operações atômicas para aplicar sua heurística, podendo mais de uma heurística estar ativa ao mesmo tempo. É considerado que duas transações de leitura e escrita conflitantes podem efetuar o *commit* no caso em que a transação de leitura ocorra primeiro. Operações atômicas longas utilizam uma política baseada em grão fino para melhorar a precisão da predição, observando o conjunto de leitura das transações já executadas, e evitar a reexecução de transações. 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. Quando uma transação tenta efetuar um *commit*, esta lista é verificada para garantir que não haja conflito entre os conjuntos de escrita e leituras. Caso seja verificado conflito entre a transação e alguma VIT, o *commit* é adiado por um tempo pré-determinado, visando que as VITs não abortem novamente.

O *STMap* [Pasqualin et al. 2020] propõe um mecanismo de *sharing-aware thread mapping* para aplicações STMs. Esse mecanismo usa informações sobre quais threads estão acessando os mesmos dados compartilhados e tenta mapear as *threads*| de forma que fiquem próximas na arquitetura onde estão sendo executadas. Em tempo de execução o STMap coleta dados sobre os acessos compartilhados entre os threads, esses dados são usados para calcular um mapeamento que tenta deixar estes threads próximas na arquitetura para compartilhar cache.

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. O LTMS, escalonador proposto na Seção 3, diferente de outros trabalhos, é 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. Esta matriz de comunicação é utilizada para avaliar o custo de acesso à memória e migrar os 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 das tarefas na arquitetura NUMA, reduzindo assim a latência de acesso à memória.

3. LTMS - *Lups Transactional Memory Scheduler*

O LTMS é um escalonador de STM consciente da arquitetura sobre a qual o programa está executando. A arquitetura da máquina obtida conhecida em tempo de execução, juntamente com a informação da latência no acesso à memória nos nós[1]. Estas informações são usadas por heurísticas que promovem a distribuição dos threads em filas de execução vinculadas aos núcleos de execução da máquina buscando-se extrair o maior desempenho em tempo de execução. O escalonador opera em três estágios: *inicialização*, *coleta de dados* e a *migração* de threads.

[1] As informações são coletadas com auxílio do aplicativo **hwloc** [Broquedis et al. 2010].

Durante a *inicialização* da aplicação, o núcleo de escalonamento instancia tantas filas de threads quanto forem o número de *cores* da máquina onde a execução ocorre, cada fila vinculada a um, e somente a um, *core*. Caso o número de threads instanciado na aplicação seja menor que o número de cores. Havendo mais threads do que *cores*, os threads são distribuídos ciclicamente entre as filas, em uma estratégia de distribuição denominada *Round-Robin*. É possível parametrizar a alocação inicial de threads, mantendo a estratégia de distribuição cíclica, mas agrupando os threads em *chunks* de valor configurável maior do que 1 (um) – neste caso, a distribuição inicial é denominada *Chunk*.

A *coleta de dados* é realizada durante a execução do programa, sendo registradas informações sobre o histórico de acessos à memória compartilhada e a quantidade de *aborts* e de *commits* observados nas transações executadas pelos threads. Este registro histórico encontra-se em duas matrizes, de comunicação e de endereços, atualizadas sempre que uma posição de memória compartilhada é acessada. A matriz de comunicação, de tamanho $n \times n$, com n igual ao número de *threads* contabiliza, em cada posição $[i, j]$, as comunicações entre os threads i e j. A matriz de endereços registra, para cada posição da matriz de comunicação, uma tabela *hash* em que a chave corresponde ao endereço de memória acessado e o valor à quantidade de acessos que este endereço recebeu. Para diminuir o *overhead* de acesso às estruturas de registro dos acessos aos dados compartilhados, a coleta acontece por amostragem, sendo realizada a coleta 1 (uma) vez a cada 100 acessos à memória compartilhada pelo thread. Em [Pasqualin et al. 2020] encontra-se a discussão e validação desta amostragem em outro estudo de caso. Por fim, informação mantida diz respeito ao histórico de *commits* e de *aborts* realizados pelos threads, em dois contadores por thread. Esses contadores apoiam a determinação do índice de contenção do thread.

O *estágio de migração* é disparado por um thread quando da ocorrência de um *abort* em uma transação. O objetivo da migração é agrupar os threads conflitantes na mesma fila de threads, para que a execução seja serializada, evitando-se conflitos futuros e ainda oferecendo oportunidades para compartilhamento de cache e, então, obter ganhos de desempenho pela exploração da localidade de referência aos dados. O processo de migração é ocorre em duas fases: a identificação da fila para qual o thread deverá ser migrado; e aplicação da heurística de migração.

```
    template<okToMigrate>
migrateThread(thread){
    if(okToMigrate(thread))
        findBestQueue(thread).push(thread);
}
```

Figura 1. Função de migração

Na Figura 1 está ilustrada a função de migração, denominada *migrateThread*, que é executada na ocorrência de um *abort*. A função *okToMigrate*, apresentada como um gabarito (*template*), implementa uma heurística para determinar se o thread atual deve ser migrada ou não, enquanto que a função *findBestQueue*, em acaso positivo, identifica a fila para qual o thread deve ser migrado. Para isso, a função *findBestQueue* recebe o identificador do thread a ser migrada e consulta na matriz de comunicação qual thread em

execução possui mais acessos em comum à memória e retorna como resposta a fila desse thread.

Diferentes heurísticas de migração podem ser implementadas para *okToMigrate*. A implemtação atual possui duas: *threshold* (Figura 2) e *latency* (Figura 3).

```
bool  thresholdHeuristic (thread) {
   return  thread.aborts / thread.commits >= threshold
}
```

Figura 2. Heurística de migração *threshold*

A heurística *threshold*, conforme afunção *thresholdHeuristic*, avalia o nível de contenção pela razão entre os *aborts* e *commits* realizados pelo thread. Um nível de contenção alto ocorre quando há uma alta taxa de *aborts*. Se o índice de contenção for maior que o limiar informado (o *threshold*), é indicada a migração do thread (retorno *true*), caso contrário, o thread, a orientação é a não migração (retorno *false*). Na atual implementação, o valor do *threshold* é um valor fixo, definido no lançamento do programa. A saber que, um valor baixo para o limiar promove migrações em maior número, enquanto que um valor baixo promove maiores níveis de concorrência e, consequentemente, maior paralelismo e maior probabilidade de *aborts*. Neste trabalho foram executados alguns testes com diferentes valores de *threshold*. A análise empírica, nos casos de estudo, foi identificado o valor de 0,8 como mais adequado.

```
bool  latencyHeuristic (thread) {
      nextQueueId= findBestQueue(thread)
      address  = getAddress( thread , addressMatrix )
      nodeNextQueue = getNUMAnode(nextQueueId)
      nodeCurrentQueue = getNUMAnode(thread.currentQueueId)
      currentLatency  =  latency (nodeCurrentQueue, address )
      nextLatency  =  latency (nodeNextQueue, address)
      return  currentLatency  > nextLatency
}
```

Figura 3. Heurística de migração *latency*

A heurística de migração *latency*, avalia a latência de acesso à memória entre os nós das filas envolvidas na migração e o endereço de memória mais acessado pelo thread. Esta heurística explora as informações contidas na matriz de endereços e busca identificar o(s) endereço(s) de memória mais acessado(s) em comum pelos threads alocados em diferentes filas com vistas a realizar agrupamento destes threads em filas pertencentes aos nós que detêm a memória física que armazena as respectiva variáveis compartilhadas. Estas informações são consideradas observando as propriedades do hardware obtidas no estágio de inicialização do núcleo de escalonamento. Caso a fila atual do thread considerado para migração possua uma latência de acesso ao endereço maior que a fila para a qual pretende-se migrar, então a migração do thread é promovida. Caso contrário, o thread é

mantida na fila atual. Esta abordagem de migração busca reduzir a taxa de *aborts* serializando parte da execução, e busca também aproveitar as características da arquitetura otimizando o acesso à memória.

4. Experimentos

O LTMS foi desenvolvido como um módulo e integrado à biblioteca TinySTM versão 1.0.5 [Felber et al. 2008], uma implementação de STM para as linguagens C e C++. A estratégia padrão, versionamento atrasado com *encounter-time locking*, foi utilizada. Nos experimentos foram utilizados programas do *Stanford Transactional Applications for Multi-Processing* (STAMP) versão 0.9.10 [Minh et al. 2008]. Os experimentos foram executados em uma máquina com 8 nós NUMA, 12 processadores Intel Xeon E5-4650, totalizando 96 núcleos e 192 threads CMT e 468 Gb de memória RAM. Foi utilizando o sistema operacional Linux Debian *kernel* 4.19.0-8-amd64 e o compilador GCC 8.3.0.

Foram executadas quatro baterias de testes com as seguintes configurações: LTMS com distribuição *Round-Robin* e migração *Threshold* (referenciada como Threshold-Round-Robin nos gráficos desta seção); LTMS com distribuição *Round-Robin* e migração *Latency* (Latency-Round-Robin); LTMS com distribuição *Chunks* e migração *Threshold* (Threshold-Chunk); e LTMS com distribuição *Chunks* e migração *Latency* (Latency-Chunk). Para comparação com o LTMS, foi executada uma bateria de testes com a TinySTM sem modificações (Tiny). Quando aplicável, o tamanho do *chunk* corresponde ao número total de threads divido pelo número de listas de threads instanciadas. Cada bateria de teste consiste em 30 execuções de cada benchmark do conjunto STAMP, para os cenários de 1, 2, 4, 8, 16, 32, 64, 128, 256, e 512 threads, que, os testes com 256 e 512 threads ultrapassam o número de threads da máquina usada. O resultado é apresentado em termos do tempo médio para cada caso considerado, sendo informada a amplitude da confiabilidade dos resultados. Nos experimentos o LTMS apresentou na maioria dos casos um melhor tempo de execução quando comparado a TinySTM sem alterações. Importante ressaltar que todos os gráficos de *abort* apresentados estão em escala logarítmica.

A Figura 4 registra os desempenhos obtidos pela execução do programa Intruder, caso em que o LTMS apresentou melhor tempo de execução para todos os cenários a partir de 2 threads. Foi anotado uma redução de até 96% do tempo de execução com a configuração Latency-Round-Robin com 512 threads, sendo que menor ganho de desempenho de 23% foi obtido com a configuração Threshold-Chunks para 4 threads. Todos os resultados do Intruder para um thread apresentaram maior tempo de execução, obtendo no pior cenário, com Threshold-Chunks, um aumento de 24%, o que é esperado, em função da adição de maior quantidade de operações de escalonamento não justificadas pela ausência de paralelismo no hardware. O LTMS também conseguiu reduzir em grande quantidade o número de *aborts*. O melhor resultado pode ser observado para *Latency-Round-Robin* com 512 threads, com 99% de redução nos *aborts* e em Threshold-Round-Robin com 2 threads com redução de 89% dos *aborts*.

A aplicação *Kmeans* apresenta baixo índice de contenção e gasta pouco tempo dentro das transações [Minh et al. 2008]. A maioria dos cenários analisados com este programa apresenta para LTMS tempos de execução inferiores em relação à TinySTM pura, conforme (Figura 5). O pior resultado, entretanto, ocorreu para o teste Threshold-Chunks com 512 threads, que apresentou um aumento de 45% no tempo de execução. O

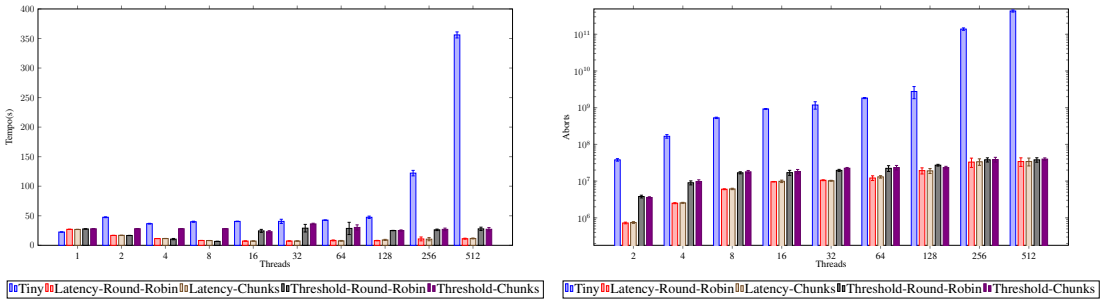

Figura 4. Intruder: tempo de execução em segundos (Esq) e número de aborts (Dir) variando o número de *threads*

melhor resultado, por outro lado, foi observado no teste Threshold-Round-Robin com 8 threads e apresentou um decremento de 80% no tempo de execução. O experimento com *Kmeans* apresentou uma redução do número de *aborts* para todos os cenários e configurações. Pode-se observar uma redução de até 99% para Threshold-Round-Robin com 512 threads. No pior caso observou-se uma redução de 71% dos *aborts* com Latency-Round-Robin com 32 threads.

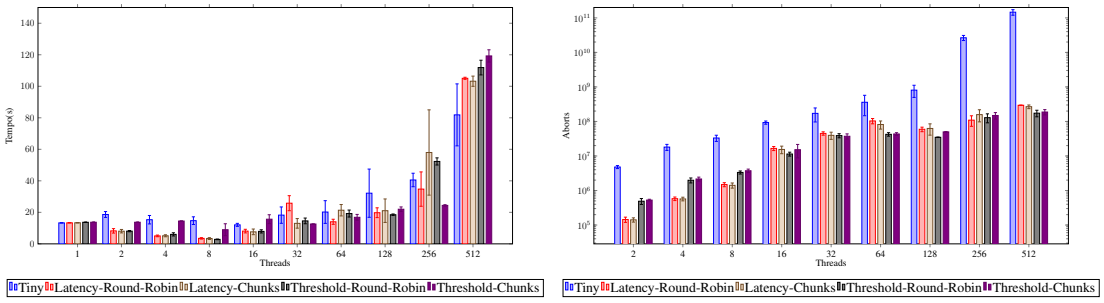

Figura 5. Kmeans: Tempo de execução em segundos (Esq) e número de aborts (Dir) variando o número de *threads*

No experimento com a aplicação *Labyrinth*, conforme a Figura 6, o LTMS apresentou, para maioria dos cenários, um decremento no tempo de execução. O melhor resultado em relação a TinySTM pura foi obtido no teste Latency-Round-Robin com 512 threads, onde é observado uma redução de 54% do tempo de execução, enquanto que o pior resultado foi um aumento de 29% no tempo de execução, observado no teste com Latency-Chunks e 16 threads. Observou-se também, na maioria dos cenários, um decremento na quantidade de *aborts*. O LTMS obteve até 54% de redução dos *aborts* no experimento Latency-Chunks com 512 threads, porém também obteve um aumento de 12% dos *aborts* em relação a TinySTM para o teste Threshold-Chunks com 16 threads.

No experimento *Vacation* (Figura 7) foi observado uma redução de 81% do tempo de execução para o teste Latency-Round-Robin com 512 threads. Para a maioria das configurações de threads com o *Vacation* ocorreu uma redução no tempo de execução com LTMS em relação à TinySTM pura, entretanto, em alguns cenários também ocorreu um aumento no tempo de execução, sendo que no pior cenário, com o teste Latency-Chunks com 128 threads, foi observado um aumento de 64% no tempo de execução. No experimento Vacation foi também observada uma redução do número de *aborts* para todos os testes executados. O melhor valor obtido no LTMS quando comparado a TinySTM

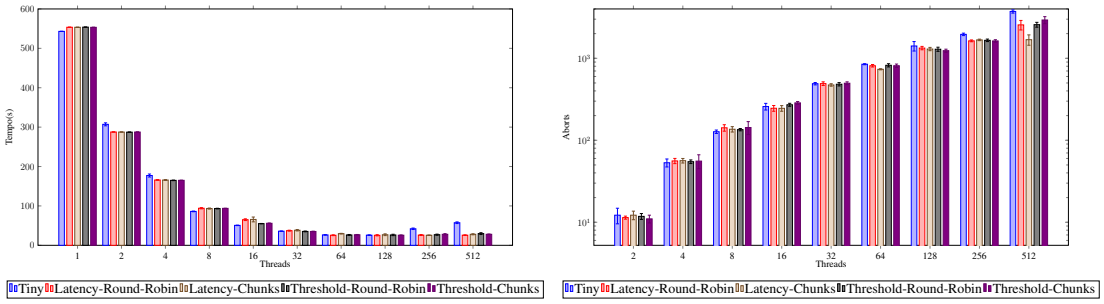

Figura 6. Labyrinth: Tempo de execução em segundos (Esq) e número de aborts (Dir) variando o número de *threads*

foi observado no Latency-Chunks com 512 threads, onde foi obtido 99% de redução nos aborts. Para Threshold-Chunks com 32 threads foi observado a menor redução do número de aborts com 27% de redução em relação a TinySTM.

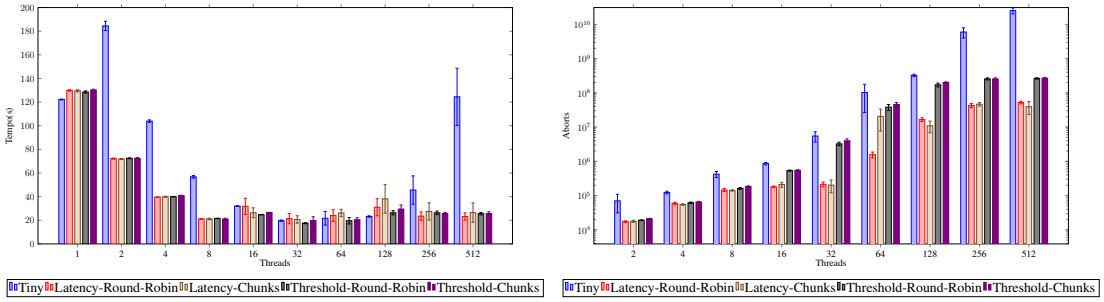

Figura 7. Vacation: tempo de execução em segundos (Esq) e número de aborts (Dir) variando o número de *threads*

O experimento com a aplicação Yada, conforme a Figura 8, apresentou para a maioria dos testes uma redução do tempo de execução do LTMS em comparação a TinySTM pura. O LTMS obteve uma redução de até 92% do tempo de execução para o teste Latency-Chunks com 512 threads e uma redução do tempo de execução de 70% para o teste Threshold-Chunks com 16 threads. No experimento Yada, também observou-se para todos os testes uma redução expressiva no número de *aborts*, sendo uma redução de 99% no teste Latency-Round-Robin com 512 threads. A menor redução de *aborts* foi de 97% com a configuração Latency-Chunks com 2 threads.

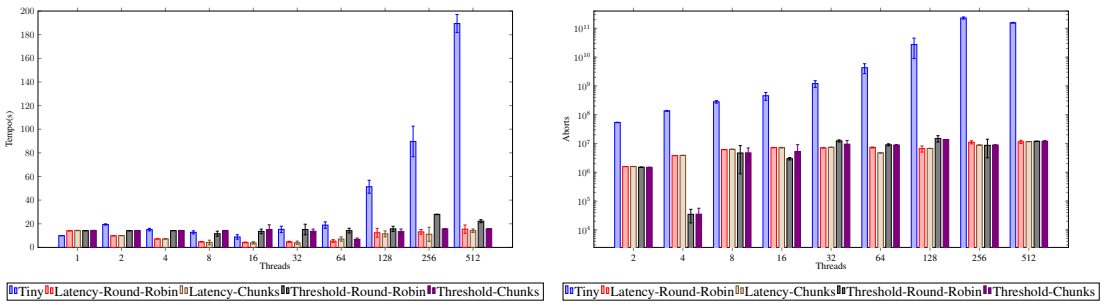

Figura 8. Yada: Tempo de execução em segundos (Esq) e número de aborts (Dir) variando o número de *threads*

4.1. Discussão

Os resultados na maioria dos *benchmarks* foram melhores em desempenho utilizando LTMS. Quando ultrapassado o número de threads CMT disponíveis na arquitetura utilizada, os únicos programa que não permitiram resultados de tempo abaixo da TinySTM foram *Bayes* e *Kmeans*. Para os demais , observou-se uma redução significativa no tempo de execução para todos cenários de threads, sendo mais expressivo para os cenários acima de 256 threads, superior ao número de threads disponíveis na arquitetura. Para o programa *Bayes*, a ordem de *commits* no início da execução afeta o tempo de execução final [Ruan et al. 2014], o que torna seu comportamento não determinístico e dificulta a sua avaliação. Dessa forma, os gráficos do *Bayes* não foram incluídos no artigo.

Os resultados apresentam para maioria dos experimentos uma redução significativa do número de *aborts* quando utilizado o escalonador proposto. Ao ultrapassar o número de threads CMT disponíveis na arquitetura utilizada todos os benchmarks apresentaram uma redução significativa no número de *aborts*, com melhor caso chegando até 99% de redução dos *aborts*. Os *benchmarks Intruder* e Yada utilizando *Latency-Round-Robin* para 512 threads apresentaram ganho de 99,99% na redução de *aborts*. Os *benchmarks Kmeans* utilizando *Threshold-Round-Robin* e *Vacation* utilizando *Latency-Chunks* reduziram respectivamente 99,88% e 99,84% dos *aborts* para o cenário de 512 threads.

Para maioria dos *benchmarks*, considerando-se o tempo de execução e número de *aborts*, obteve-se melhor desempenho utilizando o LTMS, sendo o melhor resultado alcançado com o *benchmark Intruder*. O *Intruder* possui uma alta contenção e suas transações possuem um tempo médio de execução. Para estas características, o escalonador proposto conseguiu prover um melhor mapeamento para a aplicação com a migração dos threads. O *benchmark Kmeans* apresentou um melhor tempo de execução até 128 threads, e teve um aumento no tempo de execução com 256 e 512 threads. Porém, este *benchmark* obteve uma redução no número de *aborts* para todos os cenários de threads. O *Kmeans* possui como característica uma baixa contenção e possui um tempo curto na duração da transação, além de usar uma quantidade pequena de endereços. Esta característica fez com que o LSTM realizasse mais migrações reduzindo o número de *aborts* e aumentando o tempo de execução. Para aplicações com estas características é importante manter o número de threads limitado às características da aplicação. Os experimentos Labyrinth, Vacation e Yada apresentaram resultados melhores com o LTMS para a maioria dos cenários de threads. Estes benchmarks possuem em comum um alto tempo de execução dentro das transações.

As heurísticas desenvolvidas obtiveram resultados parecidos nos testes executados. Sendo que a heurística de migração *Threshold* apresentou um desempenho pior que a heurística *Latency* para aplicações com contenção média e quantidade média de endereços distintos acessados, como foi observado no experimento Yada. É possível identificar que em alguns cenários de threads o LTMS obteve uma redução no número de *aborts* e não apresentou redução no tempo de execução. O LTMS realiza a serialização dos threads conflitantes por meio da migração, sendo que, em aplicações que geram muita migração, foram identificadas duas características. A primeira está na serialização total da aplicação, o que gera um tempo de execução semelhante a execução com 1 thread. A segunda característica identificada foi o aumento de *overhead* originado por migrações repetitivas. As heurísticas de migração buscam entender as aplicações e reduzir estas duas caracterís-

ticas limitando o número de migração. Porém, para aplicações com baixa contenção e que utilizam uma quantidade pequena de endereços, utilizar uma heurística mista que avalie o índice de contenção e o número de threads ativos pode reduzir o tempo de execução, evitando-se o excesso de migração.

5. Conclusões

A principal contribuição deste trabalho está no projeto de um escalonador de STM que considera a arquitetura onde a aplicação está em execução e a afinidade entre os threads com relação às posições compartilhadas de memória. Outra contribuição é a prototipação do escalonador LTMS, utilizando a biblioteca de STM TinySTM. O protótipo construído possui duas estratégia de distribuição de threads e duas heurísticas de migração, que permitem adaptar o comportamento do escalonador a diferentes aplicações.

Por fim, o protótipo do escalonador proposto foi validado utilizando o *benchmark* STAMP. Na maioria dos cenários e configurações observou-se uma redução no tempo de execução e número de *aborts*, quando comparado com a TinySTM pura, sendo que nos melhores casos alcançou-se uma redução de até 96% do tempo de execução e 99% do número de *aborts*.

Desenvolvido de forma modular, o LTMS permite a criação e configuração de diferentes estratégias para a distribuição inicial de threads, criação de filas, e migração dos threads conflitantes entre as filas criadas. Essa característica pode ser usada em estudos futuros. Pode-se explorar o desenvolvimento de heurísticas híbridas de migração de threads, considerando o índice de contenção e a latência de acesso à memória. Outra possibilidade de estudos futuros é avaliação quanto ao impacto energético dos escalonadores de STM, considerando-se o conhecimento quanto à arquitetura da máquina.

Agradecimentos

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

Referências

Adl-Tabatabai, A.-R., Lewis, B. T., Menon, V., Murphy, B. R., Saha, B., and Shpeisman, T. (2006). Compiler and runtime support for efficient software transactional memory. *SIGPLAN Not.*, 41(6):26–37.

Baek, W., Minh, C. C., Trautmann, M., Kozyrakis, C., and Olukotun, K. (2007). The OpenTM transactional application programming interface. In *Proceedings of PACT '07*, pages 376–387, Washington, DC, USA. IEEE Computer Society.

Baeza-Yates, R. A. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., USA.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of PACT'08*, pages 72–81.

Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: A generic framework for managing hardware

affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186.

Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., and Chatterjee, S. (2008). Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58.

Di Sanzo, P. (2017). Analysis, classification and comparison of scheduling techniques for software transactional memories. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3356–3373.

Dolev, S., Hendler, D., and Suissa, A. (2008). CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of PODC '08*, pages 125–134, New York, NY, USA. ACM.

Dragojević, A., Guerraoui, R., Singh, A. V., and Singh, V. (2009). Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of PODC '09*, pages 7–16, New York, NY, USA. ACM.

Felber, P., Fetzer, C., and Riegel, T. (2008). Dynamic performance tuning of word-based software transactional memory. In *Proceedings of PPoPP '08*, pages 237–246, New York, NY, USA. ACM.

Guerraoui, R. and Romano, P., editors (2015). *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913 of *LNCS*. Springer.

Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K. (2008). STAMP: Stanford transactional applications for multi-processing. In *Proceedings of IISWC 2008*.

Nicácio, D., Baldassin, A., and Araújo, G. (2012). Transaction scheduling using dynamic conflict avoidance. *International Journal of Parallel Programming*, 41(1):89–110.

Owens, S. (2010). Reasoning about the implementation of concurrency abstractions on x86-tso. In D'Hondt, T., editor, *Proceedings of ECOOP 2010*, pages 478–503.

Pasqualin, D. P., Diener, M., Du Bois, A. R., and Pilla, M. L. (2020). Online sharing-aware thread mapping in software transactional memory. In *Proceedings of SBAC-PAD'22*.

Rito, H. and Cachopo, J. (2015). Adaptive transaction scheduling for mixed transactional workloads. *Parallel Computing*, 41:31–49.

Ruan, W., Liu, Y., and Spear, M. (2014). STAMP need not be considered harmful. Proceedings of TRANSACT 2014.

Yoo, R. M. and Lee, H.-H. S. (2008). Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 169–178. ACM.