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

Dissertação

Escalonamento de Transações em Memória a Nível de Usuário em Haskell

Rodrigo Medeiros Duarte

Pelotas, 2020

Rodrigo Medeiros Duarte

Escalonamento de Transações em Memória a Nível de Usuário em Haskell

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 Rauber Du Bois
Coorientador: Prof. Dr. Gerson Geraldo Homrich Cavalheiro

Pelotas, 2020

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

D812e Duarte, Rodrigo Medeiros

Escalonamento de transações em memória a nível de usuário em Haskell / Rodrigo Medeiros Duarte ; André Rauber Du Bois, orientador ; Gerson Geraldo Homrich Cavalheiro, coorientador. — Pelotas, 2020.

95 f. : il.

Dissertação (Mestrado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2020.

1. Memorias transacionais. 2. Escalonamento de transações. 3. Multicore. I. Du Bois, André Rauber, orient. II. Cavalheiro, Gerson Geraldo Homrich, coorient. III. Título.

CDD : 005

Elaborada por Maria Inez Figueiredo Figas Machado CRB: 10/1612

Rodrigo Medeiros Duarte

Escalonamento de Transações em Memória a Nível de Usuário em Haskell

Dissertação aprovada, como requisito parcial, para obtenção do grau de Mestre em Ciência da Computação, Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas.

Data da Defesa: 23 de abril de 2020

Banca Examinadora:

Prof. Dr. André Rauber Du Bois (orientador)
Doutor em Computação pela Heriot-Watt University.

Prof. Dr. Gerson Geraldo H. Cavalheiro (coorientador)
Doutor em Computação pelo Institut National Polytechnique de Grenoble.

Prof. Dr. Dalvan Jair Griebler
Doutor em Computação pela University of Pisa.

Prof. Dr. Adenauer Correa Yamin
Doutor em Computação pelo Institut National Polytechnique de Grenoble.

À Natali, por sempre confiar em mim e me dar o incentivo e apoio necessário para continuar. Inspiração para minha vida.

AGRADECIMENTOS

Meu primeiro agradecimento é a Deus, por me permitir viver e explorar os caminhos da ciência e da engenharia.

Agradeço a meus pais Eni e Fernando, por terem me dado carinho, respeito e por todo o sacrífico que tiveram para que eu chega-se até aqui. Palavras não podem descrever o enorme carinho que tenho por vocês, tenham minha eterna gratidão. Aos meus irmãos Fernando, George e Érica, pelo esforço e apoio incondicional. Agradeço por cada momento de apoio, pelos descontraídos em situações boas e ajuda e compreensão nos momentos ruins. Nossa união sempre venceu os maiores obstáculos na vida, não foi diferente desta vez. Obrigado por tudo.

Faço meu agradecimento a meu orientador e professor André Rauber Du Bois e ao professor Gerson Cavalheiro, pelo apoio, ensino e ajuda durante toda o caminho.

Agradeço também a todos os amigos do LUPS (Laboratory of Ubiquitous and Parallel Systems), que sempre me ajudaram com revisões e códigos.

A todos os que aqui não citei aqui, mas que também fizeram parte deste caminho, não se julguem excluídos, cada mínima parte de suas presenças em minha vida de alguma forma contribuíram para chegar até aqui.

Beyond the horizon of the place we lived when we were young
In a world of magnets and miracles
Our thoughts strayed constantly and without boundary
The ringing of the division bell had begun
DAVID GILMOUR E POLLY SAMSON

RESUMO

DUARTE, Rodrigo Medeiros. **Escalonamento de Transações em Memória a Nível de Usuário em Haskell**. Orientador: André Rauber Du Bois. 2020. 95 f. Dissertação (Mestrado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2020.

Memória Transacional é um modelo de sincronização entre *threads* que utiliza um conceito de transações parecido com o de bancos de dados. Este modelo de sincronização apresenta uma elevada abstração, pois nele o programador somente define os dados que deve ser sincronizados, ficando a tarefa da correta sincronização a cargo do sistema transacional.

Apesar de sua elevada abstração, memórias transacionais tendem a perder desempenho em sistemas de alta contenção de memória devido à alta taxa de conflitos entre as transações. Essa alta taxa de conflitos conduz a uma degradação do desempenho por não explorar de forma eficiente os recursos disponíveis (*cores*).

Neste contexto, o escalonamento transacional emerge como uma alternativa para reduzir a alta taxa de cancelamentos. Nesta estratégia, as transações são gerenciadas com o objetivo de evitar que novos conflitos venham a ocorrer, distribuindo as transações na tentativa de explorar de forma mais eficiente os recursos computacionais disponíveis (*cores*).

Este trabalho apresenta o desenvolvimento de três diferentes escalonadores transacionais em Haskell, explorando a praticidade de implementar escalonadores, os quais são estruturas complexas, dentro de uma linguagem de alto nível de abstração. O objetivo é a comparação do comportamento de diferentes técnicas de escalonamento sobre diferentes algoritmos transacionais, como o TL2, TINY e o SWISS, que apresentam versionamento de dados tardio, adiantado e misto, respectivamente.

Os resultados obtidos demonstraram que, o uso dos escalonadores reduziu a quantidade de conflitos na maioria das aplicações, impactando em uma redução no tempo de execução. Também observou-se que, quando os escalonadores foram utilizados em aplicações de contenção máxima de memória, esses evitaram que o sistema tivesse maior degradação, apresentando um tempo de execução quase constante independente da quantidade de *threads* usadas. Por fim, observou-se que a eficiência em um escalonador tem em reduzir os conflitos também está associado ao tipo de algoritmo de memória transacional que este usa e não somente em sua capacidade de lidar com conflitos.

Palavras-chave: Memorias Transacionais. Escalonamento de Transações. Multicore.

ABSTRACT

DUARTE, Rodrigo Medeiros. **User-level Scheduling of Memory Transactions in Haskell**. Advisor: André Rauber Du Bois. 2020. 95 f. Dissertation (Masters in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2020.

Transactional memory is a thread synchronization model that uses a transaction concept which is similar to the one seen in databases. This model is highly abstract, since the programmer only needs to define data that must be synchronized, whereas the synchronization task is performed by the transactional system.

Although transactional memories are highly abstract, they tend to lose performance in high contention systems due to the large conflict rate between transactions. This leads to performance deterioration, as available resources (cores) are not explored in an efficient way. In this context, the transactional scheduling appears as an alternative to mitigate the high rate of cancellations. In this strategy, transactions are managed so that new conflicts are avoided, by distributing transactions in a way that available computational resources (cores) can be explored more efficiently.

This work presents three transactional schedulers which are developed in Haskell, exploring its practicality of implementing complex structures in a programming language which has a high level of abstraction. The goal of this work is to show the behaviour comparison of different scheduling techniques over different transactional algorithms, such as TL2, TINY and SWISS – where they have, respectively, late, forward and mixed data versioning.

The results of this work show that by using the schedulers, the number of conflicts has been reduced in most of applications, which reduced overall runtime. In addition, when schedulers were used in applications where maximum memory contention occurred, system deterioration was lower – runtime was near constant, independently of number of threads used. Finally, it has been noted that scheduler efficiency (reducing conflicts) is also associated with the type of transactional memory algorithm that is used, and not just with its capacity to deal with conflicts.

Keywords: Transactional Memory. Transactional Scheduler. Multicore.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SMP	Symmetric Multi-Processor
NUMA	Non-Uniform Memory Access
UMA	Uniforme Memory Access
GC	Contention Manager
MT	Memoria Transacional
STM	Software Transactional Memory
CPU	Central Processing Unit
RGE	Registro de Contexto de Execução
STM	Software Transactional Memory
HTM	Hardware Transactional Memory
NUMA	Non Uniform Memory Access
FIFO	First In First Out

SUMÁRIO

1 INTRODUÇÃO

Com a popularização dos processadores *multicore*, a demanda por aplicações que explorem de forma eficiente este hardware paralelo disponível têm se intensificado. Dessa forma, a programação paralela e concorrente se apresenta cada vez mais popular e necessária. Contudo, a programação para essas arquiteturas não é trivial (CZARNUL, 2018). Quando há mais de um fluxo de execução em um programa (*thread*) acessando uma mesma região de memória (ou seção crítica), este acesso necessita ser sincronizado para evitar erros de condição de corrida.

Tradicionalmente, o controle de acessos das *threads* as regiões críticas são realizados por bloqueios (*mutex*). Porém este modelo de sincronização é complexo e propenso a erros (JONES, 2007; MCKENNEY et al., 2010), pois fica a cargo do programador garantir a correta aquisição e liberação dos bloqueios no código para evitar condições de corrida e *deadlocks*. Aliado a isso, um programa moderno tem muitas linhas de código e muitos *threads*, delegar a função somente ao programador de garantir a sincronização se torna uma tarefa complexa. Assim, torna-se necessária a utilização de uma abstração que facilite a programação da sincronização de processos.

Uma alternativa para substituição dos bloqueios é a utilização da abstração chamada de "Memória Transacional"(MT), no qual a seção crítica de um código é acessada na forma de uma transação, similar a utilizada em banco de dados. Dessa forma, o programador somente deve delimitar o trecho de código que precisa ser sincronizado. O sistema de MT é que deve ser responsável pela execução correta e consistente do código, ou seja, sem *deadlocks* ou condições de corrida (HARRIS; LARUS; RAJWAR, 2010). Uma transação que foi executada sem conflitos pode ser confirmada/efetivada (*commit*), ou seja, deve atualizar em memória os novos valores. Se um conflito foi detectado, um cancelamento (*abort*) é executado e a transação é reiniciada, até que a efetivação seja possível.

Entretanto, apesar da elevada abstração e facilidade que MTs proporcionam, em alguns cenários elas podem apresentar um baixo desempenho, sobretudo em aplicações que apresentam uma alta contenção (YOO; LEE, 2008), ou seja, várias *threads*

tentando acessar uma mesma seção crítica ao mesmo tempo. Esta característica leva as transações a conflitarem e usarem de forma ineficiente os recursos computacionais, pois um conflito faz com que as transações sejam canceladas e reiniciadas novamente, podendo levar a problemas conhecidos como *starvation* e o *live lock*. Para resolver o problema dos conflitos, MTs utilizam gerenciadores de contenção (GC), que definem, no momento do conflito, qual transação deve ser cancelada e qual deve prosseguir. Contudo, uma desvantagem do GC é que ele atua de forma reativa, ou seja, somente após o conflito ser detectado e sua única ação é reiniciar a transação conflitante (NICÁCIO; BALDASSIN; ARAÚJO, 2013).

O escalonamento de transações ou escalonador transacional (MALDONADO et al., 2010) surgiu como uma alternativa para reduzir a taxa de conflitos entre transações, consequentemente aumentando o desempenho das MTs. Em uma transação que foi cancelada, todas as operações computacionais que foram executadas até o cancelamento podem ser consideradas como um desperdício de trabalho e recursos. Dessa forma, reduzindo os conflitos, faz-se com que o sistema transacional utilize de forma mais eficiente os recursos computacionais disponíveis.

1.1 Motivação e Objetivos

Haskell é uma linguagem funcional pura e de tipagem estática que esta em crescente evolução. Seu elevado nível de abstração em conjunto com uma variedade de ferramentas para a programação paralela, permite que se desenvolva programas complexos e robustos. Como exemplo disso, existem várias implementações dos algoritmos complexos de MT, desenvolvidos inteiramente em Haskell, como por exemplo, TL2 (DU BOIS, 2011), a TINY (DUARTE et al., 2015) e a SWISS (DU BOIS; PILLA; DUARTE, 2012), entre outros (SABEL, 2014; HUCH; KUPKE, 2006).

STM Haskell estende a linguagem funcional Haskell com um conjunto de primitivas para MT que, aliada ao forte tratamento de tipos de Haskell, fornece uma ferramenta de sincronização intuitiva, evitando que o programador cometa erros de sincronização (MARLOW, 2013). Essa verificação de erros é feita durante a compilação do programa, graças a tipagem forte e estática da linguagem. Esta biblioteca, além de fornecer primitivas básicas para transações, fornece também primitivas de alto nível onde é possível compor transações alternativas ou criar condições que devem ser satisfeitas antes que uma transação possa executar.

Os trabalhos sobre escalonamento de transações de memória presentes na literatura, focam em linguagens imperativas/procedurais além de apresentar os resultados de escalonamento para apenas um ou no máximo dois algoritmos de MT. Neste trabalho buscou-se desenvolver três modelos de escalonadores de transações, a nível de usuário na própria linguagem Haskell, para verificar quais seriam os impactos na

execução de aplicações usando MTs quando estas estão sobre a supervisão de um escalonador de transações. Estes escalonadores foram testados sobre três diferentes algoritmos de STM clássicos (TL2 (DICE; SHALEV; SHAVIT, 2006), TINY (FELBER; FETZER; RIEGEL, 2008) e SWISS (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009)), com o proposito de verificar qual seria a relação entre o comportamento dos escalonadores sobre diferentes modos de operação de MTs.

O objetivo principal deste trabalho é demostrar como diferentes modelos de escalonamento de transações se comportam quando combinados com diferentes algoritmos de MT, analisando aplicações com diferentes níveis de contenção de memória.

Como objetivo secundário, é fornecida uma plataforma para testes de escalonadores, visto que modificações pode ser facilmente feitas nos códigos Haskell já prontos, para a adaptação dos escalonadores para futuras bibliotecas de STM e modelos de escalonamento.

1.2 Metodologia de Desenvolvimento

O desenvolvimento do trabalho iniciou-se com a revisão dos principais conceitos teóricos relacionados ao escalonamento de transações em nível de usuário disponíveis na literatura. O estudo englobou os primeiros trabalhos na tentativa de mitigar o problema dos conflitos, até trabalhos mais novos onde técnicas mais aprimoradas são aplicadas. A partir destes estudos, três estrategias de escalonamento de transações foram tomadas como base para o desenvolvimento de três escalonadores, dois usando técnicas base como a serialização e a migração de transações e outro, envolvendo uma analise instrumental mais completa do sistema transacional para a tomada de decisões no escalonamento.

As ferramentas usada para o desenvolvimento e testes dos escalonadores foram o compilador GHC, que é a ferramenta mais completa para a programação concorrente em Haskell (MARLOW, 2013), e o STM Haskell Benchmark (PERFUMO et al., 2008), que é um *benchmark* especifico para memórias transacionais, onde se explora diferentes tipos de aplicações sobre diferentes tipos de contenção de memória.

Para fins de avaliação dos desempenhos dos escalonadores, foi comparado seus tempos de execução e suas taxas de cancelamentos com as das bibliotecas sem a utilização de escalonamento.

1.3 Contribuição e Resultados

Os trabalhos na área de escalonamento transacional, a nível de usuário, apenas apresentam comparação do uso de escalonadores em no máximo dois tipos diferentes de algoritmos transacionais e sobre um único modelo de escalonamento. Este trabalho

contribui com um dos primeiros estudos do desenvolvimento, implementação e testes, em uma linguagem funcional, de três modelos de escalonamento transacional sobre três diferentes algoritmos de memória transacional.

Os resultados obtidos demostraram que o uso dos escalonadores reduziram a quantidade de conflitos levando a uma redução dos tempo de execução das aplicações. Demostraram também que a quantidade de cancelamentos não é a única métrica que deve ser levada em conta pelo escalonador para a tomada de decisões, é necessário avaliar a contenção da aplicação e o tipo de algoritmo de STM sendo utilizado.

1.4 Estrutura do Texto

Este trabalho esta organizado em 7 capítulos. O presente capitulo trouxe uma introdução ao trabalho desenvolvido, apresentando o problema abordado e a motivação para a realização do trabalho. No capitulo 2 é apresentado os conceitos de memórias transacionais, fazendo uma revisão sobre versionamento de dados, detecção de conflitos e sobre o gerenciador de contenção. Nele também são apresentados os principais algoritmos de memórias transacionais. No Capitulo 3 são apresentados os trabalhos relacionada na área de escalonamento, fazendo uma apresentação cronológica dos escalonadores a nível de usuário que foram estudados para o desenvolvimento deste trabalho. Esta seção demostra a evolução dos escalonadores para transações, desde os de eurística mais simples ate os de instrumentação mais elaborada. Nos Capitulo 4 é apresentado como foram desenvolvidos os escalonadores deste trabalho, detalhando o funcionamento dos algoritmos bem como descrevendo as estruturas de dados e as função implementadas em Haskell. Esta seção demostra também como os escalonadores são associados as bibliotecas de memórias transacionais.

O Capitulo 5 descreve todas as ferramentas, e suas configurações, usadas para a realização dos testes dos escalonadores. Este capitulo detalha como aplicação é executada, seus algoritmos e quais os níveis de contenção. Dando continuidade, o Capitulo 6 apresenta os resultados obtidos sobre duas seções diferentes. A primeira apresenta os resultados obtidos com as aplicações sem controle da quantidade de *threads* e que somente fazem uso das primitivas mais simples de memórias transacionais. Já a segunda apresenta os resultados das aplicações que fazem uso de primitivas que estende as ações transacionais.

Por final, o Capitulo 7 apresenta as conclusões obtidas com o trabalho, fazendo uma observação geral dos resultados e apresentando uma conclusão geral.

2 MEMÓRIAS TRANSACIONAIS

Memória transacional é um modelo de sincronização entre *threads* que aumenta o nível de abstração para a sincronização no acesso a dados em memória. Este modelo de sincronização para programação concorrente usa como base o conceito de transações (semelhate às transações de bancos de dados) para garantir o sincronismo entre *threads* (HARRIS; LARUS; RAJWAR, 2010).

Em memórias transacionais, uma transação é uma sequência de operações que modificam a memória de forma atômica, ou seja, são executadas por completo (*commit*) ou podem ser canceladas (*abort*) caso ocorra um conflito. Outro item importante é a garantia de isolamento. Transações não podem ver os resultados intermediários uma das outras. Dessa forma a execução de transações concorrentes equivale ao resultado da execução dessas mesmas transações em alguma ordem serial. Essas características (atomicidade e isolamento) são garantidas por duas propriedades que são o versionamento de dados e detecção de conflitos.

Memórias Transacionais podem ser implementadas em três abordagens distintas, em software também conhecida como STM (*Software Transactional Memory*), em Hardware HTM (*Hardware Transactional Memory*), ou ainda de forma híbrida HyTM (*Hybrid Transactional Memory*), que envolve tanto sotware como em hardware.

2.1 Versionamento de Dados

Para garantir atomicidade, uma transação deve guardar tanto o dado corrente quanto o especulativo até o momento do *commit*, quando então, o dado especulativo passa a ser a nova informação armazenada. Caso o *commit* ocorra, o dado corrente deve ser mantido como informação válida. Para isso, memórias transacionais podem usar dois tipos de versionamento, o adiantado e o tardio.

2.1.1 Versionamento Adiantado

No versionamento adiantado os dados especulativos da transação são guardados diretamente na memória compartilhada entre as *threads* sendo preservado o dado

corrente em um registro temporário para uma possível recuperação (*undo log*). Caso não ocorra nenhum conflito (*commit*), a única operação realizada deve ser descartar o valor do *undo log*, mas se caso ocorra um conflito (*abort*), o valor do *undo log* deve ser restaurado na memória e a transação reiniciada. Um esquema da sequência de passos envolvidos no versionamento adiantado é apresentado na Figura 1.

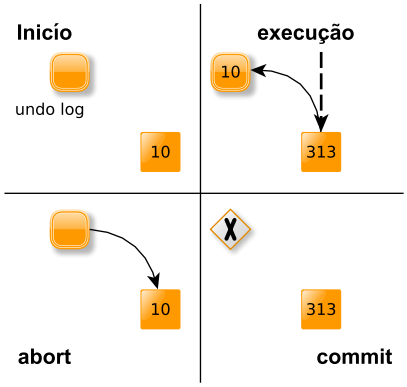

Figura 1 – Esquema de funcionamento do *undo log* no versionamento adiantado.

2.1.2 Versionamento Tardio

Neste tipo de versionamento, os dados especulativos para escrita são preservadas em um *buffer* local. Se a transação conseguir realizar um *commit*, os valores do *buffer* devem ser escritos na memória. No caso de ocorrer um conflito, o valor do *buffer* local deve ser descartado e a transação reiniciada. Um esquema de como este versionamento funciona pode ser visto na Figura 2.

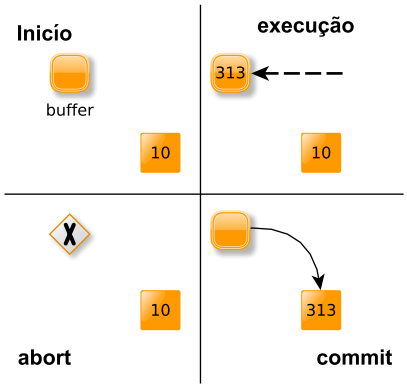

Figura 2 – Esquema de funcionamento do *buffer de escrita* no versionamento tardio.

Versionamento adiantado apresenta melhor desempenho na efetivação em casos

de baixa contenção (pucos conflitos), pois os valores especulativos já estão na memória, porém em casos de alta contenção este modelo de versionamento não é tão eficiente, pois neste caso a transação tem o custo extra de desfazer as alterações na memória. No entanto uma abordagem de versionamento tardio, em casos de alta contenção, é mais eficiente na efetivação (*commit*). Transações devem transferir seus dados locais para a memória, porém em cancelamentos não é necessário nenhuma intervenção extra (RIGO; CENTODUCATTE; BALDASSIN, 2007).

2.2 Detecção de conflitos

Um conflito ocorre quando pelo menos duas transações estão acessando um mesmo dado na memória e pelo menos um destes acessos é de escrita. A detecção de conflitos também pode ser realizada de forma adiantada (pessimista) ou tardia (otimista).

2.2.1 Detecção adiantada

Este tipo de detecção de conflito ocorre no momento em que uma transação realiza acesso a memória (leitura ou escrita). Se o mesmo dado em memória está sendo acessado por outra transação, o conflito é detectado e a transação é abortada. Dependendo da forma como um conflito é tratado neste tipo de detecção, problemas como *livelock* podem ocorrer. Este problema ocorre quando transações ficam abortando umas as outras e não conseguem progredir. O esquema de funcionamento da detecção de conflitos pode ser visto na Figura 3, em que são demonstrados quatro casos distintos possíveis.

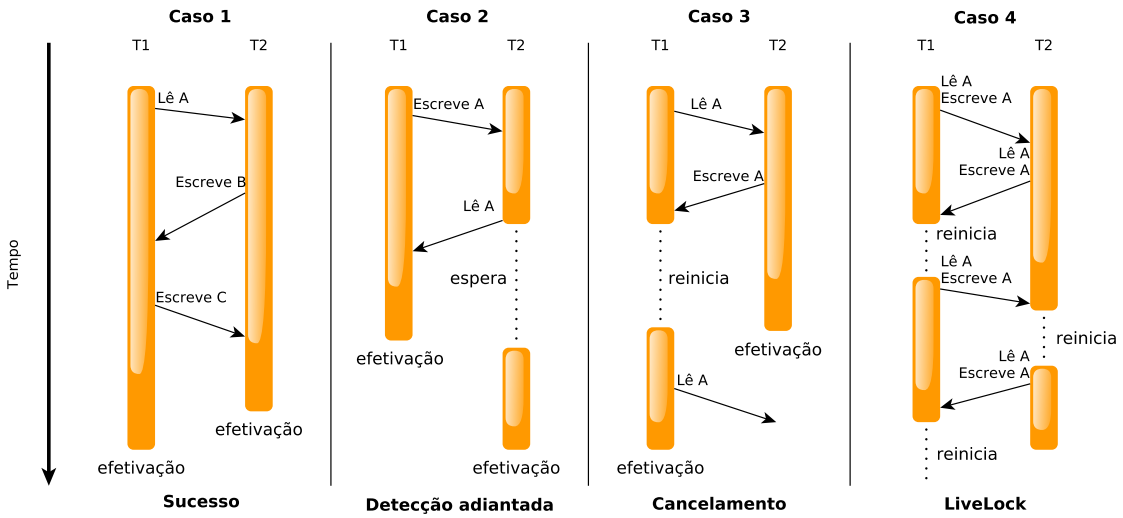

Figura 3 – Esquema de funcionamento da detecção de conflitos adiantada.

No Caso 1, as transações T1 e T2 acessam dados disjuntos, logo ambas as tran-

sações podem ser efetivadas. Já no Caso 2, a transação T2 espera até que T1 seja efetiva, garantindo que os dados não serão alterados podendo assim ser efetivada. O Caso 3, T1 é cancelada porque a visão da memória se torna inconsistente devido a escrita realizada pela T2. E finalmente, no Caso 4, é apresentado o problema de *Live Lock*. onde duas transações pequenas se cancelam mutuamente por tempo indeterminado.

2.2.2 Detecção tardia

Neste método, os conflitos são detectados somente no momento do *commit*, ou seja, no final da execução da transação. Aqui o problema que pode vir a ocorrer é o de *starvation*, onde uma transação que possui uma grande carga de trabalho pode ser abortada por transações menores, levando a uma postergação da conclusão desta transação maior. Pode-se ver um esquema de funcionamento deste método na Figura 4, onde novamente são demonstrados quatro casos distintos.

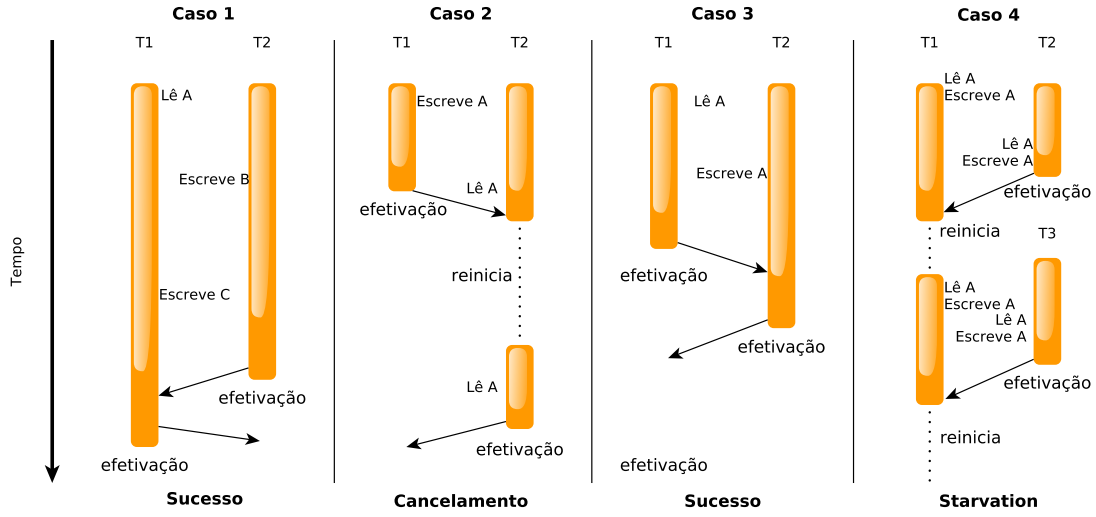

Figura 4 – Esquema de funcionamento da detecção de conflitos tardia.

No Caso 1, novamente ambas as transações podem ser efetivadas pois acessam dados disjuntos na memória. Já no Caso 2, T2 é abortada porque T1 efetiva seus dados da memória e T2 no momento do *commit* verifica que o estado da memória não é consistente e aborta. No Caso 3, ambas as transações podem ser efetivadas pois a leitura realizada por T1 se dá antes da efetivação da escrita por parte de T2, logo T1 no momento do *commit*, tem uma visão consistente da memória. E por último o Caso 4, onde uma transação T1 que possui uma computação onerosa, é abortada por transações menores (T2,T3,...) que modificam os mesmos dados acessados por T1 e, por possuírem menor tempo de computação, conseguem efetivar antes de T1, levando a postergação da efetivação de T1 (*starvation*).

É importante perceber que versionamento de dados adiantado necessita que a

detecção de conflitos também seja desta forma, para evitar inconsistência nos dados computados pelas transações. Porém o versionamento de dados tardio não obedece a esta regra, o mesmo pode usufruir de ambos os tipos de detecção de conflitos.

2.2.3 Gerenciador de Contenção

Para resolver um conflito, as implementações de memórias transacionais utilizam um Gerenciador de Contenção (*Contention Manager*) (GUERRAOUI; HERLIHY; POCHON, 2005a). O Gerenciador de Contenção é uma unidade fundamental para o correto funcionamento do sistema transacional, é ele quem decide qual transação deve abortar e qual deve prosseguir sua computação. Sem o Gerenciador de Contenção as transações podem demorar muito para serem efetivadas ou entrar em *loop*, como os problemas apresentados na seção anterior. A decisão de qual transação deve ser cancelada é realizada aplicando regras específicas onde, dependendo do tipo de regra adotada, pode-se conseguir melhor desempenho por diminuír a ocorrência de cancelamentos. Existem diferentes tipos de Gerenciadores de Contenção, cada um com suas características, entre eles estão os seguintes (DEMSKY; DASH, 2010):

- **Timid:** neste modelo de gerenciador de contenção, a transação que detectou o conflito é imediatamente reiniciada, não prevendo que possivelmente a transação possa conflitar novamente pelos mesmos motivos, tornando este modelo sujeito a problemas de *livelock*.
- **Backoff:** a transação conflitante é cancelada e espera por um período (geralmente exponencial) para ser re-executada, o valor deste tempo é de acordo com a quantidade de cancelamentos que a transação sofreu.
- **Delay:** a transação cancelada será reexecutada somente após a transação com quem conflitou seja efetivada, funciona somente em versionamento adiantado.
- **Greedy:** neste modelo é atribuído um marcador de tempo *timestamp* a cada transação no início de sua primeira execução. Caso um conflito ocorra, a transação que possuí o *timestamp* mais antigo prevalece.
- **Aggressive:** frente a um conflito, a transação que detectou o conflitou aborta a transação conflitante porém, este modelo é propenso a problemas de *livelock*.
- **Karma:** fornece prioridades dinâmicas as transações (definido pelo número de objetos lidos pela transação), em um conflito a transação que têm a prioridade mais baixa é cancelada.
- **Eruption:** tem funcionamento similar ao Karma, porém aqui, uma transação de baixa prioridade que conflitou com uma de alta, adiciona a sua prioridade a tran-

sação de alta, aumentando a probabilidade da transação de mais alta prioridade efetivar.

- **Kindergarten:** cancela a transação quando detectado o conflito, porém, se um conflito for detectado na reexecução, a transação que está em reexecução tem prioridade para continuar.
- **Priority:** é atribuída a transação uma prioridade estática no início de sua execução.
- **Polka:** é uma combinação dos gerenciadores Karma e Backoff, fornecendo a transação conflitante aumento exponencial de tempo para a efetivação, aumentando exponencialmente a diferença de tempo entre as transações conflitantes.

2.3 Algoritmos de STM

2.3.1 TL2

O algoritmo TL2 (DICE; SHALEV; SHAVIT, 2006), utiliza versionamento de dados atrasado, granularidade em nível de palavra e detecção de conflitos tardia, a sincronização é realizada usando *locks* versionados (DICE; SHALEV; SHAVIT, 2006). Além disso, esta implementação faz uso de um relógio global, denominado GV4, que funciona como um mecanismo de versão para os dados compartilhados.

O relógio global é incrementado de forma atômica por todas as transações efetivadas que escrevem na memória e lido por todas transações. No início de cada transação, o relógio global é lido e copiado para um variável local da *thread* chamada *read-version*. Em cada leitura ou escrita dos dados da transação é verificado a consistência dos dados, comparando-se o *read version* da *thread* com a versão do dado correspondente a localização da memória. Se o *read version* é menor do que o número da versão do dado, a transação é abortada pois a localização de memória foi modificada após a transação ter sido iniciada. Em tempo de efetivação da transação, o relógio global do sistema é incrementado e esse valor é usado como o novo número de versão das localizações de memória modificadas.

Apesar da redução do custo de validação utilizando-se o relógio global, este mecanismo pode se tornar um gargalo conforme o número de *threads* aumenta, pois maior será a contenção no acesso ao relógio. A TL2 utiliza o gerenciador de contenção tímido que aborta prontamente a transação que detectou o conflito.

2.3.2 TinySTM

TinySTM (FELBER; FETZER; RIEGEL, 2008; LIU et al., 2018; POUDEL; SHARMA, 2019) é uma biblioteca STM ainda considerada estado da arte na implementação de

memórias transacionais. É baseada no algoritmo LSA (Lazy Snapshot Algorithm), o qual possui características similares às do TL2 (FELBER; FETZER; RIEGEL, 2008). Assim como a TL2, utiliza-se um relógio global, a sincronização é baseada em *locks* versionados, a granularidade de detecção de conflitos é em nível de palavra. A principal diferença é o uso de versionamento de dados adiantado, dessa forma evitando que uma transação condenada a abortar continue executando, assim desperdiçando recursos computacionais.

2.3.3 SwissTM

O algoritmo SwissTM busca obter bom desempenho tanto para transações curtas e estruturas de dados pequenas, como para transações grandes e cargas de trabalhos complexas (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009). O algoritmo têm características parecidas com a TL2: versionamento de dados atrasado, granularidade em nível de palavra e sincronização baseada em *locks* versionados, além de usar um relógio global para o processo de validação dos dados.

A diferença é que a detecção de conflitos de escrita/escrita é feita de modo adiantado a fim de prevenir que transações condenadas a abortar continuem executando. Em transações de leitura/escrita o sistema transacional detecta conflitos de forma atrasada, na esperança do conflito não se efetivar de fato e assim permitir de forma otimista que mais paralelismo seja obtido. Além disso, quando detecta uma localização de memória com versão maior do que o *read version* de uma transação, o algoritmo revalida o conjunto de leitura para tentar adiantar o *read version* da transação para que este seja o mesmo da localização de memória que gerou o conflito. Se todos os elementos do *read set* da transação continuam com valores abaixo do *read version* da transação, este pode ser adianto, sendo atribuído o mesmo valor de versão da localização de memória que gerou o conflito.

2.4 STM Haskell

STM Haskell (LE; YATES; FLUET, 2016) é uma extensão da linguagem funcional Haskell que fornece abstrações de alto nível para a programação concorrente usando a abstração de memórias transacionais. Uma linguagem funcional como Haskell é ideal para implementar memórias transacionais por dois motivos (HARRIS; MARLOW; JONES, 2005):

- O tratamento de tipos consegue separar as ações que possuem efeito colateral das que não possuem. Nem todo o tipo de ação pode ser executada dentro de uma transação, por exemplo operações de entrada e saída. O sistema de tratamento de tipos de Haskell consegue garantir que somente as ações corretas serão executadas dentro de uma transação, evitando assim que o programador

cometa erros durante a programação.

- Em Haskell, a maioria das computações são puras, ou seja, não possuem nenhum tipo de efeito colateral. Esse tipo de ação não modifica a memória e não precisa ser acompanhada pelo sistema transacional. Essas ações nunca precisam ser desfeitas em caso de uma transação abortar.

Dentro de transações, apenas operações que modificam a memória podem ser executadas. Para isso STM Haskell provê a abstração de variáveis transacionais (`TVars`). O tipo `TVar a` representa uma variável transacional que contém um valor genérico de tipo `a`. Este tipo de variável pode ser modificada por duas primitivas especificas:

```
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

A primitiva `readTVar` recebe como argumento uma variável do tipo `TVar a` e retorna como resposta uma ação transacional do tipo STM que quando executada, retorna o valor de tipo `a` contido na variável transacional. A primitiva `writeTVar` serve para escrever um valor de tipo `a` em uma variável de tipo `TVar a`.

A única maneira de executar ações transacionais de tipo `STM` é utilizando a primitiva `atomically`.

```
atomically :: STM a -> IO a
```

A primitiva `atomically` faz com que uma ação transacional seja executada atomicamente em relação a outras transações sendo executadas concorrentemente, garantindo atomicidade e isolamento (JONES, 2007).

Como exemplo o código apresentado na Figura 5, demostra a aplicação de STM-Haskell para o tratamento de depósito e saque em uma conta bancária.

```
1  saque :: TVar Int -> Int -> STM ()
2  saque conta val = do
3      saldo <- readTVar conta
4      writeTVar conta (saldo - val)
5
6  deposita :: TVar Int -> Int -> STM ()
7  deposita conta val = do
8      saldo <- readTVar conta
9      writeTVar conta (saldo + val)
10
11 (...)
12
13 conta <- atomically $ newTVar (313::Int)
14 forkIO $ atomically $ saque conta 24
15 forkIO $ atomically $ deposita conta 100
```

Figura 5 – Depósito e saque concorrente em uma conta bancária.

Nas linhas 1 e 6 são declaradas duas funções para saque e depósito respectivamente em uma conta bancária. Nas linhas 14 e 15, são criadas duas *threads*, uma onde se realiza um saque e outra que realiza um depósito. A obrigatoriedade da execução das operações dentro de uma função `atomically` protege as transações (saque e depósito) de condições de corrida. O sistema de tipos protege o programador de cometer erros, ou seja, se o programador tentar acessar uma das operações fora de uma operação atômica, o compilador detecta o erro, informando ao programador que as funções devem ser protegidas por um bloco atômico (MARLOW, 2013).

O STM Haskell, diferentemente de outras bibliotecas, apresenta também primitivas de alto nível para a composição de transações, que são apresentadas das próximas duas seções.

2.4.1 Retry

A primitiva `retry` permite implementar transações que somente prossigam quando um certo recurso está disponível, o que é importante para a composição sequencial de transações:

```
retry :: STM a
```

A operação realizada por `retry` é simples, abortar a transação e executar ela novamente no momento que pelo menos uma das variáveis lidas pela transação tenha sido modificada. Isso evita que a transação fique abortando consecutivamente devido a mesma condição que levou a chamada de `retry`.

```
1  saque :: TVar Int -> Int -> STM ()
2  saque conta val = do
3      saldo <- readTVar conta
4      if (saldo >= val) then (writeTVar conta (saldo - val)) else retry
```

Figura 6 – Função de operação de saque com a operação de *retry*

No código exemplo da Figura 6, se não houver saldo suficiente para o saque, a transação é abortada através de uma chamada a `retry`, ficando em espera. Quando uma outra transação escrever na variável `conta` e houver saldo suficiente, a transação é reiniciada, a condição do `if` é satisfeita e a transação é efetivada, caso nenhum outro conflito ocorra.

2.4.2 OrElse

Outra primitiva importante é a `orElse`. Esta primitiva recebe como argumento duas transações e devolve uma outra ação transacional que faz uma escolha entre as ações que recebeu como argumento, dessa forma permitindo compor transações como alternativas:

```
orElse :: STM a -> STM a
```

Nem sempre desejamos bloquear uma transação devido a um `retry`. Assim a primitiva `orElse` nos dá a opção de executar uma ação alternativa caso a primeira chame `retry`. O código da Figura 7 demonstra um exemplo de uso do `orElse`.

```
1  saque :: TVar Int -> Int -> STM ()
2  saque conta val = do
3      saldo <- readTVar conta
4      if (saldo > val) then (writeTVar ptr (saldo-val)) else retry
5
6  (...)
7
8  conta1 <- atomically $ newTVar (313::Int)
9  conta2 <- atomically $ newTVar (412::Int)
10 forkIO $ atomically $ saque conta1 350 `orElse` saque conta2 350
```

Figura 7 – Operação de saque alternativa entre contas bancárias.

Primeiramente a transação que realiza o saque na `conta1` é executada, se esta chamar `retry`, ela então é descartada e a segunda transação (saque na `conta2`) passa a ser executada. Se a segunda transação também chamar `retry`, então toda a transação é reexecutada.

2.5 Considerações Finais

Memórias Transacionais é um modelo de sincronização que apresenta elevada abstração para a programação concorrente. Devido ao sistema transacional garantir o correto acesso à memória pelas regiões críticas das *threads*, o programador fica isento desse cuidado na hora da programação, facilitando seu uso. Sua diversidade de configurações e características, como o versionamento de dados, a detecção de conflitos e gerenciador de contenção, permite que se explore diversas configurações para se otimizar o desempenho sobre diferentes aplicações.

Contudo, apesar de existir diferentes modelos de gerenciadores de contenção, este somente tem ação reativa ou seja, somente ira atuar frente a um conflito e sua única ação e somente reiniciar a transação conflitante. Assim devido a essa deficiência apresentada pelos gerenciadores de contenção, uma nova abordagem para o tratamento de conflitos é o escalonamento transacional. No escalonamento transacional, transações tem sua ordem de execução reordenada para que se evite novos conflitos ou ate mesmo evite os.

3 ESCALONAMENTO DE TRANSAÇÕES

Como apresentado no Capitulo 2, quando duas transações acessam um mesmo dado na memória, sendo que pelo menos um destes acessos seja de escrita, há um conflito. Conflitos são resolvidos por uma unidade das implementações de memórias transacionais chamado Gerenciador de Contenção (*contention manager*). Este determina qual transação deve continuar e qual transação deve ser cancelada frente ao conflito. Muitos estudos já foram realizados sobre a forma como os gerenciadores de contenção tratam conflitos (SPEAR et al., 2009; SCHERER III; SCOTT, 2005; GUERRAOUI; HERLIHY; POCHON, 2005b; DEMSKY; DASH, 2010). Porém gerenciadores de contenção somente tratam conflitos após estes ocorrerem, e pior, eles não fazem nenhum tipo de tratamento para evitar que uma transação seja reiniciada novamente por conflitos consecutivos (NICÁCIO; BALDASSIN; ARAÚJO, 2013). Como alternativa aos gerenciadores de contenção o foco das pesquisas mudaram para escalonadores. Nos escalonadores é possível conseguir um maior controle sobre como tratar conflitos e em alguns casos até mesmo a tentativa de evitar que eles ocorram (DRAGOJEVIć et al., 2009).

3.1 ACC

Um dos primeiros trabalhos a apresentar um estudo sobre a variação dos níveis de contenção em sistemas transacionais foi ACC (Adaptative Concurrent Control) (ANSARI et al., 2008a). Neste trabalho os autores se preocuparam em entender as oscilações que um sistema transacional apresenta no decorrer de sua execução. É apresentado que delegar ao programador a responsabilidade por definir a quantidade de *threads* concorrentes não é a melhor opção, pois alcançar desempenho com este tipo de ação se torna uma forma de tentativa e erro. No trabalho é apresentado TCR (*transaction commit ratio*), que controla o nível de concorrência dinamicamente, ou seja, quando o nível de efetivações aumenta após um limiar definido pelo TCR, o sistema aumenta a quantidade de *threads* concorrentes, se a quantidade de efetivações começa a diminuir, então o sistema começa a reduzir a quantidade de concorrência.

Manter um nível de concorrência constante (número de *threads*) pode gerar perda de desempenho pelo alto nível de contenção gerado ou pelo uso ineficiente dos recursos disponíveis.

Como solução o trabalho apresenta a análise de quatro diferentes modelos de controle de contenção operando juntamente com os gerenciadores de contenção apresentados na Seção 2.2.3, com exceção do modelo *Timid*. Estes modelos são descritos como a seguir:

- **SimpleAdjust**: é um esquema simples onde, se a quantidade de efetivações ultrapassa o limiar definido pelo TCR, o mesmo incrementa em um, a quantidade de *threads* que podem ser executadas concorrentemente. Ao contrário, se o valor decai abaixo do limiar definido, o mesmo reduz em um o número de *threads* que podem ser executadas. A quantidade de efetivações é mensurada através de amostras realizadas em períodos de tempo.

- **ExponentialInterval:** neste modelo o tempo de amostragem da taxa de efetivações é reduzido pela metade quando há alterações na quantidade de *threads* sendo executadas, fazendo com que o sistema se torne mais responsivo. Caso o número de *threads* não mude, então o sistema dobra o valor do tempo de amostragem. O esquema de aumento e redução do número de *threads* funciona igual ao modelo anterior.

- **ExponentialAdjust:** é uma extensão do *Simple Adjust* em que ajusta a quantidade de *threads* pela diferença entre o TCR amostrado e o TCR limiar, que é um threshold. A fórmula inicialmente decide colocar ou retirar uma *thread* a mais no sistema transacional, logo após dobra o valor para cada 10% que o valor do TCR medido esteja fora do *range* limiar. Assim, se um sistema está em um *range* de 30% a 60% e tem uma amostra de 80%, o sistema transacional pode adicionar ou remover até quatro *threads*.

- **ExponentialCombined:** Uma mistura entre ExponentialInterval e ExponentialAdjust, formando um sistema mais responsivo.

O trabalho é implementado sobre a biblioteca DSTM2 (HERLIHY; LUCHANGCO; MOIR, 2006). Para os testes os autores utilizam um algoritmo de roteamento de circuítos que estressa o sistema transacional, onde se conseguiu aumentos de performance utilizando os sistemas TCR em 38% no melhor e 2% no pior caso.

3.2 PoCC

O segundo trabalho também faz um controle do nível de concorrência do sistema transacional, controlando a quantidade de *threads* executando concorrentemente (AN-

SARI et al., 2008b). O controlador apresentado neste trabalho, PoCC (*P-only Concurrency Controller*) é baseado em um controlador PID modelo P-only (ÅSTRÖM; HÄGGLUND, 1995). Neste novo algoritmo são inseridos filtros para evitar ruídos nas amostras de tempo, gerados por transações com grandes blocos de código. O controlador deste trabalho necessita que seja ajustado o intervalo da amostra de tempo para cada diferente aplicação, o código do controlador pode ser visto no Algoritmo 3.1.

Algoritmo 3.1 – Algoritmo do PoCC

```
1
2  if currenttime − lastSmapleTime < samplePeriod, goto Step 1;
3  if numTransactions < minTransactions, goto Step 1;
4  TCR <− numCommits / numTransactions × 100;
5  ΔTCR ← TCR − setPoint;
6  if (numCurrentThreads=1) & (TCR > setPoint);
7      (a) then Δthreads ← 1;
8      (b) else Δthreads ← ΔTCR × numCurrentThreads / 100
9        (rounded to the closest integer);
10 newThreads ← numCurrentThreads + delta_threads;
11 Adjust minThreads ≤ newThreads ≤ maxThreads;
12 numCurrentThreads ← newThreads;
13 Set lastSampleTime ← currentTime, goto Step 1;
```

Onde:

- samplePeriod: é o período de amostra (ajustável);
- minTransactions: é o mínimo de transações requeridas em uma amostra (ajustável);
- setPoint: é o ponto de ajuste;
- minThreads: é o mínimo de *threads* que são executadas de forma concorrente, no caso uma única.
- maxThreads: máximo de *threads* que podem ser executadas concorrentemente, geralmente o número de *cores* físicos do processador.

O valor em $setPoint$ determina o quão conservativo será o PoCC para usar os recursos eficientemente. Um valor de $setPoint$ alto (e.g. 90%) é rápido para reduzir o número de *threads*, porém lento para adaptar o sistema para mudanças súbitas na taxa do TCR e vice e versa. Porém os autores demostram que ajustar um $setPoint$ para um valor de até 70% não apresentou degradação de desempenho.

Este trabalho, assim como o anterior, é implementado sobre a DSTM2. Os testes realizados neste tipo de controladora foram sobre dois benchmarks, o STAMP (MINH et al., 2007) e o Lee (ANSARI et al., 2008) comparando o PoCC com os controladores do trabalho anterior (ACC). Como resultado os autores chegam a uma melhora de 31% no tempo de execução em relação aos modelos apresentados em 3.1.

3.3 ATS

Um dos primeiros trabalhos a inserir a ideia de escalonamento em transações foi o ATS (*Adaptive Transaction Scheduling*) (YOO; LEE, 2008). O escalonador proposto opera com o mesmo objetivo dos trabalhos anteriores, que é limitar o nível de concorrência do sistema para reduzir conflitos. Para isso o escalonador obtém informações oriundas do gerenciador de contenção, sendo que o escalonador somente consulta o GC quando o sistema se encontra em uma situação de alta contenção. Isso reduz o *overhead* imposto pelo escalonador para o sistema transacional. Para determinar quando uma transação deve ser escalonada, é inserida uma métrica dinâmica chamada CI(*contention instensity*). Cada *thread* possui seu próprio CI, calculado pela Equação 1.

$$CI_n = \alpha \times CI_{n-1} + (1 - \alpha) \times CC \tag{1}$$

Onde, CI indica o nível de contenção por *thread*, CC é a contenção corrente, que é re-setada quando a transação consegue efetivar e ou setado quando a transação é cancelada. O valor de α é um valor usado para ajustar o balanceamento da equação, se o valor de α for baixo a situação atual da transação é mais relevante para o cálculo da intensidade de contenção. Se caso o valor de α seja alto, então o histórico da transação é levado em consideração. Quando uma transação inicia seu CI é zerado, conforme a transação aborta o valor de CI vai aumentando. Conforme o CI vai aumentando o valor, as transações vão sendo colocadas em uma fila que serializa a execução das transações conflitantes. Assim que as taxas de transações conflitantes diminui de valor, as transações deixam de ser enfileiradas para voltar a explorar paralelismo.

O esquema de funcionamento do escalonador pode ser visto na Figura 8. Quando uma *thread* detecta que seu nível de contenção (CI) se tornou elevado, a mesma informa ao escalonador que deve ser inserida em uma fila de espera (Fila de Transações). O escalonador verifica o nível de contenção e determina a quantidade de *threads* que o mesmo deve liberar para execução. As *threads* somente consultam o escalonador quando a contenção é alta. Se o sistema transacional esta em baixa contenção, então as *threads* não fazem consultas ao escalonador, reduzindo o custo de operação do escalonador.

O escalonador foi implementado sobre a biblioteca de memórias transacionais RSTM (MARATHE et al., 2006), utilizando o gerenciador de contenção modelo Polka. Os testes foram realizado sobre cinco diferentes microbenchmarks que são: RBTree, HashTable, LinkedList, RandomGraph e LFU-Cache, microbenchmarks estes conhecidos na literatura (MARATHE et al., 2006; SCHERER III; SCOTT, 2005). Os testes demostraram que o escalonador melhorou o desempenho do sistema em 1,3x e 1,5x,

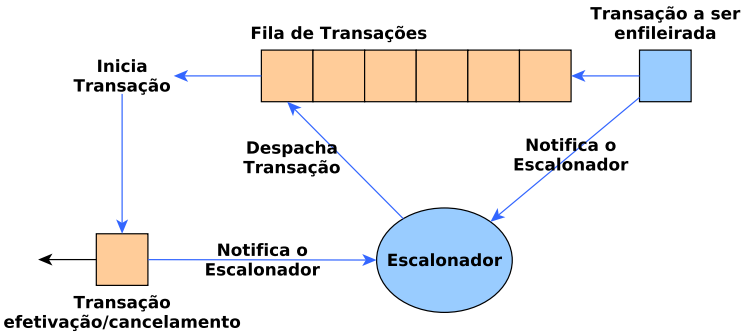

Figura 8 – Esquema de funcionamento do escalonador da ATS.

alcançando o pico máximo de 5.9x.

3.4 CAR-STM

Em (DOLEV; HENDLER; SUISSA, 2008) é apresentado CAR-STM, que é um escalonador que opera de duas formas distintas. A primeira, usando apoio do gerenciador de contenção, serializando em uma única fila de execução todas as transações conflitantes, com o objetivo de reduzir a contenção e a probabilidade de novos conflitos. Na segunda abordagem é usando o *proactive collision avoidance*, que tem o objetivo de colocar *threads* com transações conflitantes no mesmo *core* do processador, evitando assim que as mesmas conflitem entre si. Um esquema da arquitetura do escalonador pode ser visto na Figura 9. Para cada *core* existente no processador, CAR-STM fixa um *thread* (*TQ thread*), que executa as transações de uma fila. Essa fila é preenchida pelo módulo *dispacher*, que recebe transações conflitantes de outras threads. Toda a *thread* que teve sua transação escalonada, é colocada em *idle* pelo escalonador, esta somente é colocada novamente em execução quando sua transação estiver concluída. O número de *TQ threads* é igual a quantidade de *cores* que o processador possui.

CAR-STM garante que cada transação que se encontra na fila de transações, vai ser executada na ordem de chegada, a menos que a mesma seja movida para o gerenciador de contenção. Um gerenciador de contenção implementado especificamente para este escalonador, chamado *serializing contention manager*, resolve conflitos migrando uma transação conflitante para outra fila de transações de outra *TQ threads*.

Dois modelos de gerenciadores de contenção são implementados:

- *Basic Serializing Contention Manager* (BSCM): move a transação conflitante para a fila de transações do *core* onde está sendo executada a transação com quem conflitou.

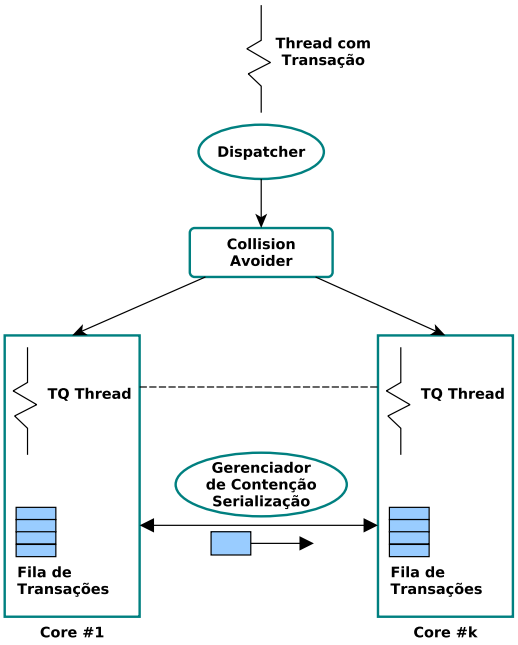

Figura 9 – Esquema de funcionamento do escalonador da CARSTM.

- *Permanent Serializing Contention Manager* PSCM: realiza a mesma ação que o anterior, porém aqui, a transação conflitante é marcada como "dependente" da transação com quem detectou o conflito, ficando esta liberada somente quando sua opositora for efetivada.

Este processo de marcar a transação como dependente, está relacionado ao fato de que uma transação conflitante pode conflitar novamente, se uma terceira transação também estiver envolvida. Por exemplo, no BSCM digamos que uma transação T_b conflita com uma transação T_a, logo o gerenciador resolve serializar T_b após T_a. Se um conflito entre T_a e uma terceira transação, digamos T_c ocorrer, T_a pode ser serializada após T_c, logo T_b pode conflitar com T_a novamente. Ao marcar a transação como subordinada no PSCM, quando uma transação é migrada de uma fila de transações para outra, a transação marcada como subordinada é levada junto, ou seja, se T_a conflita com T_c e T_b é subordinada de T_b, a serialização será T_c,T_a e T_b.

Sempre que uma nova transação é iniciada o módulo *dispatcher* consulta o *collision avoider* para identificar a probabilidade que esta nova transação tem de conflitar com qualquer outra que está sendo executada nos *cores* disponíveis, o que tiver menor probabilidade de conflito a transação é disparada. Para isso, cada transação possui uma estrutura (*T-info*), que contém informações sobre quais os dados que a transação ira acessar durante sua execução.

O escalonador foi incorporado a implementação da RSTM (MARATHE et al., 2006) e os testes foram realizados usando o STMBench7 (GUERRAOUI; KAPALKA; VITEK,

2007). Os resultados obtidos mostram que o uso da CAR-STM fornece menores tempos de execução, maior estabilidade do sistema transacional (redução de conflitos) e apresenta melhor desempenho se comparada a implementação pura da RSTM. Quanto ao uso do *collision avoider*, o mesmo fornece melhores resultados se comparado a implementação usando somente os gerenciadores de contenção BSCM e PSCM.

3.5 SHRINK

Em SHRINK (DRAGOJEVIć et al., 2009) os autores apresentam um escalonador que prediz o futuro acesso de cada *thread* que está executando uma transação, baseado no histórico de acessos passados e dinamicamente serializa as transações baseado na predição dos conflitos. SHRINK usa os conceitos de localidade de referência e "afinidade de serialização"para escalonar as transações.

Para a localidade de referência é usado dois métodos, o primeiro usando a noção de localidade temporal, que analisa as variáveis mais recentemente acessadas através da análise do conjuntos de leituras das transações que conseguiram efetivar recentemente. E a segunda analisando o conjunto de escrita das transações que repetiram mais vezes. Assim SHRINK usa as informações recolhidas destas duas análises para decidir onde escalonar as transações.

SHRINK serializa as transações somente se a contenção for alta, Para detectar se o sistema transacional esta operando em alta ou baixa contenção, o escalonador mantém um parâmetro chamado *success rate* para cada *thread*. Assim o escalonador começa a serialização somente se o valor de *success rate* cair abaixo de um limite.

Para a localidade temporal, o escalonador mantém o *readset* de algumas transações efetivadas de cada *thread* em um conjunto do tipo *bloon filter*. Então, quando o escalonador dispara uma nova transação, ele verifica os endereços acessados no *bloon filter* e através de uma análise de um intervalo de confiança, ele verifica a região de memória que será acessada no futuro. Para a análise do conjunto de escritas (*write sets*), o escalonador somente adquire os dados em transações que reiniciaram pelo menos uma vez. Resumindo, SHRINK sempre verifica se um endereço que será acessado pela transação já se encontra em um dos conjuntos de predição, se algum endereço se encontra, a transação é serializada, se não, executa normalmente.

SHIRINK serializa as transações usando um *lock* global. Sempre que uma transação inicia, ela adquire um *lock* global e sempre que uma transação termina ela libera o *lock*. Também mantém por *thread* um conjunto de *Bloom Filters*, para representar os conjuntos de leitura das *threads* da última transação executada. *Bloon Filters* fornecem um meio rápido de verificar os endereços de acesso de leitura das transações. O parâmetro `locality_window` especifica o número de transações que são usadas para

a predição.

O *Flowchart* do SHRINK pode ser visto na Figura 10 e as modificações necessárias para integrar SHRINK a uma biblioteca de STM é visto no Algoritmo 3.2.

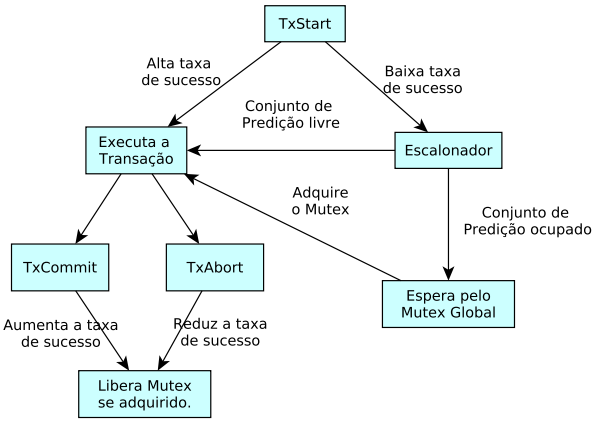

Figura 10 – Flowchart do Escalonador Shrink.

Quando uma transação lê um endereço, SHRINK verifica se este está presente no *BloonFilter* correspondente aos *read-sets* das transações antigas da *thread*. Se o endereço é frequentemente acessado, então os endereços são adicionados ao *predict_read_set* da próxima transação pela *thread*. Quando uma transação efetiva, o *success rate* é incrementado e o *lock* global é liberado caso a transação estivesse serializada. Em um cancelamento, o escalonador copia o *write set* para o *prediction_write_set*, para avaliar a próxima transação da *thread*, também o *success rate* é decrementado e o *lock* global liberado, se adquirido.

Ao iniciar uma transação, o escalonador somente é ativado se o *success rate* estiver abaixo de um determinado valor. Primeiro, SHRINK aplica a heurística de afinidade de serialização. Em caso de um baixo valor de afinidade, a transação inicia normalmente sem interferência do escalonador. No caso da afinidade de serialização ser alta, SHRINK verifica se algum endereço acessado pela transação está no *prediction_read_set* ou no *predicted_write_set* de outra *thread*. Se existe de fato tal endereço, SHRINK serializa a transação que iniciou.

Algoritmo 3.2 – Algoritmo do Shrink

```
1
2  On Transactional Start:
3  if succ_rate < succ_threshold then
4    generate a random number r between 1 and 32
5    if r < wait_count then
6      for each address read_pred in pred_read_set
7        if some other thread is writing read_pred then
8          lock global_lock
9          atomically increment wait_count
10         break
11     if not owner of global_lock then
```

```
12        for each address write_pred in pred_write_set
13           if some other thread is writing write_pred
14              lock global_lock
15              atomically increment wait_count
16              break
17  if last transaction was committed then
18    remove all addresses from pred_read_set
19  remove all addresses from pred_write_set
20
21  On Transactional Read Of addr:
22  if (addr ∉ bf_0) then
23    add addr to bf_0
24    for (i = 1; i < locality_window; i := i + 1) do
25      if (addr ∈ bf_i) then
26        confidence = confidence + ci
27      if confidence ≥ confidence_threshold then
28        add addr into pred_read_set
29
30  On Transactional Commit:
31  succ_rate := (succ_rate + success)/2
32  if own global_lock then
33    unlock global_lock
34    atomically decrement wait_count
35
36  On Transactional Abort:
37  copy write_set of transaction into pred_write_set
38  succ rate := succ rate/2
39  if own global_lock then
40    unlock global_lock
41    atomically decrement wait_count
```

As variáveis do algoritmo são:

- **global_lock**: é um **lock** global compartilhado entre as *threads*;
- **success, succ_threshold** e **confidence_threshold**: são constantes;
- **succ_rate**: é uma variável inteira por *thread*;
- **read_pred**, **write_pred** e $addr$: são endereços;
- **pread_read_set** e **pred_write_set** são os conjuntos de predição por *thread*;
- **wait_count** e **confidence** são inteiros (inicializados com 0);
- bf_i é o *Bloon Filter* das ultimas transações;
- ci é uma constante representando o valor de confiança das ultimas transações;
- **locality_windows** é o tamanho dos conjuntos de *Bloon Filters* por *thread*

Os testes foram realizados sobre duas bibliotecas de STM, Swiss-TM (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009) e TinySTM (FELBER; FETZER; RIEGEL, 2008), usando variados valores para os itens do Algoritmo 3.2. Por análise de desempenho, os seguintes valores foram escolhidos: `succ_threshold = 0.5`, `success = 1`,

`locality_window = 4`, `confidence_threshold = 3`, `c1 = 3`,`c2 = 2`, e `c3 = 1`. SHRINK é implementado em C++, para facilitar a integração com as bibliotecas de STM. Também foi utilizados como benchmarks o STAMP (MINH et al., 2007) e o STMBench7 (GUERRAOUI; KAPALKA; VITEK, 2007).

Os resultados demostram que o uso do escalonador com serialização, para transações dominadas por leituras no STMBench7, prove ganhos de até 55% se comparado a implementação sem escalonamento da SwissTM. Porém, com o uso de predição de acessos o escalonador acaba apresentando perda de desempenho de até 4%. Já no STAMP, a implementação apresentou ganho de desempenho para ambientes de alta contenção, devido a serialização das transações. O que acabou reduzindo a quantidade de conflitos e melhorou o desempenho geral do sistema.

3.6 THROTTLE and PROBE

No trabalho de (CHAN; LAM; WANG, 2011) os autores demostram que, ao aumentar a quantidade de *threads* em um sistema transacional, o desempenho tende a cair devido ao aumento de conflitos que podem ocorrer. Para provar essa análise os autores utilizam o *benchmark* STAMP, aumentando o número de *threads* de 2 até 32 em um sistema de 8 *cores*. Ao aumentar o número de *threads*, os autores demostram que o sistema transacional acaba apresentando uma gradativa perda de desempenho.

Para resolver o problema apresentado anteriormente os autores apresentam duas técnicas de escalonamento que são Throttle e Probe. A ideia principal dos algoritmos é evitar que transações conflitantes entrem na seção crítica do código ao mesmo tempo, com o intuito de evitar conflitos. Para isso o escalonador transacional utiliza as informações de efetivação e cancelamentos como um *feedback* para tomar as decisões.

Alguns parâmetros são introduzidos pelos autores para o controle de concorrência, são eles:

1. *quota*: número de *threads* que podem executar transações, também chamado de *concurrency quota*

2. *active*: número de *threads* ativas que entraram em transações, porém ainda não saíram.

3. *peak*: número máximo de *threads* que iniciaram transações, incluindo as que já deixaram suas transações.

4. *commit*: número de transações efetivadas.

5. *aborts*: número de transações canceladas.

6. *stallen*: uma *flag* booleana indicando que algumas *threads* estão presas em suas transações.

quota é um valor ajustável; *commit* e *aborts* são valores fornecidos pelo sistema STM; *active*, *peak* e *stalled* são variáveis derivadas que facilitam o algoritmo de escalonamento.

Algoritmo 3.3 – Mecanismo de controle de quota

```
1  function onBegin
2  retry:
3    if active >= quota then
4      set stalled to true
5      yield
6      goto retry
7    end if
8    active := active + 1
9    if peak  < active then
10     peak := active
11   end if
12 end
13
14 function onCommit
15   active := active - 1
16   commit := commits + 1
17 end
18
19 function onAbort
20   active := active - 1
21   aborts := aborts + 1
22 end
```

O Algoritmo 3.3 apresenta o código básico que opera com o conceito de *quota*. Quando uma *thread* tenta iniciar uma nova transação, esta deve verificar se a quantidade de *threads* ativas não foi alcançada (*quota*). Se este for o caso, a *thread* deve esperar até que alguma das *threads* ativas termine sua transação (em outras palavras, até que uma transação efetive ou cancele) ou o valor corrente de *quota* seja aumentado pelo escalonador. O valor de *quota* é ajustado dinamicamente conforme a política de escalonamento empregada.

O método *Throttle* tem seu pseudo código visto no Algoritmo 3.4. Quando a taxa de cancelamentos aumenta, a taxa de efetivação começa a cair (definido como *ratio* no Algoritmo). Se a taxa cair abaixo de um valor (*threshold*) definido, o sistema reduz a *quota*, como consequência reduzindo o número de *threads* do sistema. Assim que a taxa *quota* subir novamente as *threads* passam a explorar a concorrência novamente.

Algoritmo 3.4 – Politica Throttle

```
1  while true do
2    sleep for a constant time (e.g. 5ms)
3    if commits + aborts < warmup then
4      continue
5    end if
```

```
6     ratio = commits / (commits + aborts)
7     if peak < quota then
8       quota := peak
9     else if ratio < threshold then
10      quota := quota - 1
11    else if stalled is true and ratio > threshold then
12      quota := quota + 1
13    end if
14    reset peak, commits, aborts to zero and
15    stalled to false
16  end do
```

peak e outros contadores são re-setados em um intervalo regular de 5ms. A heurística de ajustar *quota* igual a *peak* se *quota* for maior que *peak* é de re-setar a *quota* corrente, fazendo com que todas as *threads* parem. Para evitar um congelamento do sistema, uma medida é tomada: atualizar *quota* somente quando `commits + aborts >= warmup` (e.g. warmup = 10). No momento que `commits + aborts` alcançarem um valor maior que 10, ao menos 10 *threads* estarão aptas a rodar no sistema, evitando assim que *quota* seja nula, evitando o travamento do sistema.

O método *Probe* pode ser visto no Algoritmo 3.5. Este método é similar ao anterior, porém aqui é utilizada uma técnica de tentativa e erro para tentar aumentar a taxa de efetivações. Nesta política os autores usam *commit rate* como uma medida do número de transações efetivadas por unidades de tempo, ao invés de *commit ratio*. Esta unidade de tempo é quantificada por *laps*. O *commit rate* é calculado como a quantidade de efetivações divida pela quantidade de *laps*.

Algoritmo 3.5 – Politica Probe

```
1   set direction to down
2   while true do
3     sleep for a constant time (e.g. 5ms)
4     if peak = 0 and active = 0 then
5       continue
6     else if commits + aborts < warmup then
7       laps := laps + 1
8       continue
9     else
10      laps := laps + 1
11    end if if peak < quota then
12      quota := peak + 1
13      set direction to down
14    else if quota = 1 then
15      set direction to up
16    else if commits / laps < last_commits / last_laps then
17      set direction to reverse(direction)
18    end if
19    if direction is down then
20      quota := quota - 1
21    else
22      quota := quota + 1
23    end if
24    last_commits := commits
25    last_laps := laps
```

```
26     reset peak, commits, aborts, laps to zero
27  end do
```

O sistema completo é implementado sobre a TinySTM e também é feita uma comparação com outros dois modelos de escalonadores vistos aqui, ATS e o SHRINK usando o *benchmark* STAMP. Os resultados demonstram que o uso das técnicas de escalonamento alcançaram melhoras de 11.1% e 12.4% em média no desempenho, se comparada a execução das aplicações usando a Tiny sem escalonamento. Se comparado aos outros dois escalonadores, as técnicas aqui apresentaram perdas médias de 2.1% se comparada a ATS e ganhos de até 14.5% se comparado a SHRINK.

3.7 LUTS

Em (NICÁCIO; BALDASSIN; ARAÚJO, 2011) é apresentado o escalonador LUTS. Nesta implementação são utilizados dois tipos de heurísticas para a detecção do nível de contenção. A primeira para transações pequenas, que acrescenta baixo custo para o cálculo da contenção, é implementada com a mesma ideia apresentada no algoritmo da ATS (Figura 8). O segundo, já para transações grandes, insere um pouco mais de instrumentação para a coleta dos dados, esta instrumentação extra não acrescenta muito custo a execução da transação pois o *overhead* inserido não é relevante se comparado ao tempo de execução da transação.

LUTS fornece uma interface simples para o desenvolvimento de programas com múltiplos dados (SPMD), isso inclui funções para criação e gerência de *threads*. Nenhum método de sincronização é fornecido, já que é adotado STM como modelo de sincronização provido pela biblioteca de STM. A única exceção é uma barreira, que bloqueia as *threads* até que todas alcancem o mesmo ponto de execução.

Neste modelo de escalonador, também não são criadas mais *threads* que a quantidade de recursos da CPU(*cores*), por exemplo, se tivermos uma máquina de 8 *cores*, apesar de ser possível iniciar um programa com 16 *threads*, de forma transparente o escalonador somente executa 8 *threads* por vez. Para realizar esta tarefa, LUTS utiliza o conceito de `RCEs` ou (Registros de Contexto de Execução). Cada `RCE` encapsula o estado de uma das *threads* e o escalonador fica responsável por despachar as `RCEs` para uma *thread* de sistema que irá executa-lo. Quando o escalonador é iniciado, LUTS cria o número necessário de `RCEs` para atender as *threads* de usuário e aloca estas `RCEs` em *threads* de sistema, conforme o número de *cores* da CPU. A principal razão pela qual LUTS aborda este método é de evitar falso paralelismo.

Quando o número de `RCEs` é menor que o número de *cores* disponível, o sistema é executado de forma convencional (sem nenhuma tarefa de escalonamento). Quando um `RCE` é mapeado para uma *thread* de sistema, só existe duas formas de sua correspondente *thread* de sistema ficar livre que é, ou a *thread* concluindo sua tarefa, ou

explicitamente chamando uma uma ação de *yeld*. Os métodos de gerenciamento de *threads* e interface com o escalonador fornecidos por LUTS são:

- **luts_init(num threads)**: Inicia o sistema com o número de *threads* especificado.
- **luts_start(func, args)**: Após a chamada desse método, cada *thread* irá começar a executar a função *func* com os argumentos *arg*.
- **luts_barrier wait()**: Execução só progride depois que todas as *threads* atingem a barreira.
- **luts_shutdown()**: Limpa as estruturas utilizadas pelo LUTS e finaliza o processo.
- **luts_yield()**: O contexto corrente é inserido ao final da fila e o contexto do início da fila é colocado em execução.
- **luts_getid(ctxId)**: Retorna um ID único do contexto de execução.
- **luts_switch_from(ctxId)**: O escalonador troca o contexto corrente por outro contexto na fila que contenha um ID diferente do passado como argumento.
- **luts_switch_to(ctxId)**: O escalonador troca o contexto corrente por outro contexto na fila que contenha o ID passado.

Como apresentado anteriormente, LUTS utiliza duas heurísticas para o escalonamento de transações. A primeira CILUTS, para transações curtas, utiliza um modelo parecido com o visto na ATS 8. Nesta técnica, cada transação mantém internamente uma variável que descreve a probabilidade de conflito. Para quantificar esta probabilidade é utilizado o conceito de Intensidade de Contenção (IC), Equação 2.

$$IC_n = \alpha \times IC_{n-1} + (1 - \alpha) \times CA \tag{2}$$

Inicialmente IC tem valor zero e a cada término de uma transação (cancelamento ou efetivação) a equação é avaliada. O valor de CA (Contenção Atual) é definida como 0 em uma efetivação e 1 em um cancelamento. Assim como na ATS, o valor de α define se o histórico ou a situação atual das transações devem ser considerado para o cálculo. Após alguns testes, os autores concluíram que o valor de 0.75 é o ideal. Quando uma transação esta prestes a iniciar, esta verifica se o valor de IC está acima de um valor determinado. Se este for o caso uma ação é tomada para evitar um possível conflito, enquanto que na ATS a transação seria serializada, em LUTS a transação é simplesmente substituída por outra na fila de RECs, usando o método `luts_switch_from`.

A segunda técnica HASHLUTS, é utilizada para transações longas. Esta técnica utiliza instrumentação inserida na transação para prover uma ação mais efetiva do escalonador. Basicamente a ação de HASHLUTS é prever qual a melhor transação a ser escalonada, dado um conjunto de transações ativas. O pseudo código de HASHLUTS pode ser visto no Algoritmo 3.6. Para auxiliar a explicação do algoritmo o autores usam a Figura 11.

Algoritmo 3.6 – Algoritmo da LUTS

```
1  double conflictTable[][];
2  int bestTx[];
3
4  upon stm_init
5    resetBestTx();
6    resetCT();
7    for each core i
8      activeTx[i] = INVALID;
9
10 upon start
11   int line_index = hash(activeTx);
12   int tx_id = bestTx[line_index];
13   luts_switch_to_id(tx_id);
14   updateActiveTx(thisCore, tx_id);
15
16 upon abort
17   updateActiveTx(thisCore, INVALID);
18   int line_index = hash(activeTx);
19   increaseIntCT(line_index, tx_id);
20   if(bestTx[line_index] == tx_id){
21     for each transaction tx {
22       if(conflictTable[line_index][tx] <
23         conflictTable[line_index][tx_id])
24       {
25         bestTx[line_index] = tx;
26       }
27     }
28   }
29
30 upon commit
31   updateActiveTx(thisCore, INVALID);
32   int line_index = hash(activeTx);
33   decreaseIntCT(line_index, tx_id);
34   if(conflictTable[line_index][tx_id] <
35     conflictTable[line_index][bestTx[line_index]])
36   {
37     bestTx[line_index] = tx_id;
38   }
```

HASHLUTS utiliza três estruturas globais para a previsão de conflitos que são: `activeTx` que é um vetor que contém as transações ativas, este com o tamanho igual a quantidade de *cores* da CPU; `conflictTable` que é uma tabela que armazena a probabilidade de conflitos e um vetor que serve como atalho para escolher a melhor transação (`bestTx`). Estas estruturas são descritas nas linhas 1 e 2 do pseudo código. Cada posição de `activeTx` armazena um ID único para cada transação que está

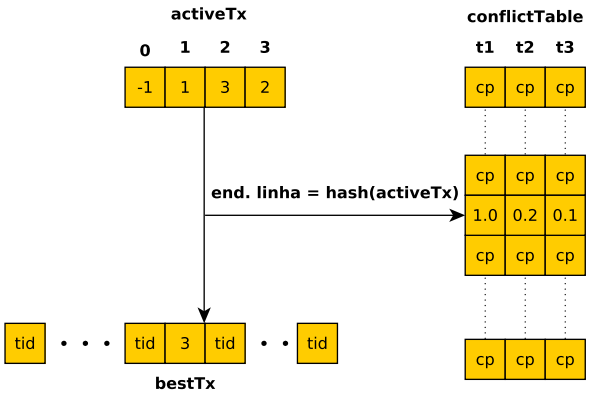

Figura 11 – Estrutura de funcionamento do segundo tipo de heurística da LUTS.

rodando em um *core* especifico. A figura assume uma máquina com 4 *cores* e três diferentes transações, representadas pelos IDs 1, 2 e 3. Se o *core* não estiver executando uma transação, o mesmo é anotado com o ID -1. O conjunto de transações ativas na figura é dado pelo vetor -1,1,3,2, portanto o *core* 0, não esta executando nenhuma transação. Quando o *core* 0 vai iniciar uma transação, para selecionar a melhor transação ele consulta `conflictTable`. Cada linha de `conflictTable` identifica um conjunto de transações, onde cada coluna identifica a intensidade de conflito (IC) para cada transação dado o conjunto de transações que esta sendo executado. Uma função *hash* é usada para mapear cada conjunto de transações ativas para um índice especifico em `conflictTable`. A configuração da imagem mostra que a *hash* dos conjuntos de transações executando é mapeado para a linha onde $t1$, $t2$ e $t3$ possuem IC 1.0, 0.2 e 0.1, logo a transação que possuiu a menor IC, e que deve ser escalonada, é $t3$. Consultar a tabela toda vez que uma transação nova é inciada pode se tornar custoso. Para reduzir este custo o vetor `bestTx` contém a melhor transação que se encaixa na configuração atual da CPU.

Para a implementação de LUTS, foi usada como base a biblioteca TinySTM e SwissTM, avaliada com os *benchmarks* STAMP e STMBench7. Também são realizadas comparações entre os escalonadores ATS e SHRINK. Os resultados mostram que, quando comparado a implementação da biblioteca sem escalonamento, LUTS somente começa a apresentar ganho de desempenho a partir de uma certa quantidade de *threads*, isso devido ao fato de que ao aumentar a quantidade de *threads*, sistemas de STM começam a perder desempenho devido a elevada taxa de cancelamentos, o que não ocorre em LUTS, pois a mesma controla o nível de contenção. Já comparada aos outros escalonadores, LUTS apresentou comportamento similar aos demais, apresentando somente em alguns casos melhor desempenho. A aplicação da LUTS com a SwissTM apresenta ganho de desempenho se comparado a utilização com TinySTM, principalmente em ambientes que possuem elevada contenção.

Porém, nestes testes o principal objetivo dos autores era demostram a fácil integração entre o escalonador e uma biblioteca de STM diferente.

3.8 RELSTM

O escalonador RELSTM, apresentado em (SAINZ; ATTIYA, 2013), mostra uma técnica onde se tenta prever futuros conflitos através da análise de conflitos indiretos com outras transações, também conhecidos como *second-hop conflicts*. A ideia principal é verificar a relação entre as transações que estão sendo executadas. Duas transações estão relacionadas se estas conflitam uma com a outra. Esta ideia pode ser estendida para *second-hop conflicts*: por exemplo, um conjunto de transações T_{1h} que conflita diretamente com uma transação T_x e, por sua vez, o conjunto T_{2h} (*second-hop*) que possui transações que conflitam com alguma transação de T_{1h}. Assim recursivamente pode-se estender a ideia de relações para *n-hop*.

Quando um conflito ocorre, cada transação registra sua oponente, bem como todas as transações em que a oponente entrou em conflito. Após ser cancelada, uma transação é colocada em *backoff* se sua oponente direta e ou uma porcentagem de transações de *second-hop* ainda estiverem em execução. O algoritmo em si é uma espécie de escalonador pro-ativo, isto porque ele ativa a política de escalonamento antes que um conflito ocorra, por exemplo: se uma transação T_a é uma *second-hop transaction* de T_b e T_a ainda está em execução, o escalonador pode bloquear a execução de T_b até T_a ser concluída, evitando um conflito.

O Flowchart de execução do escalonador de RELSTM pode ser visto na Figura 12. E seu pseudo código para uma transação que está iniciando, pode ser visto em Algoritmo 3.7.

Algoritmo 3.7 – Pseudo Código do RELSTM antes da transação começar

```
1  Upon transaction Tx start:
2  if (Tx.conflicted is not NULL)
3    if Tx.conflicted is registered in executing array
4      lock on serial lock
5  else if (Tx.twohop is not empty)
6    tx executing: Number
7    for each ID in Tx.twohop
8      if (ID is registered in executing array)
9        increment tx executing
10   if ((executing / size of Tx.twohop) ≥ TWOHOP_RATE)
11     backoff during Tx.backoff time
12     increment Tx.backoff for exponential backoff
13 end if
```

Quando uma transação inicia, primeiro ela se registra como executando e então verifica se possui conflitos do tipo *one-hop*: se existe alguma colisão anterior com alguma transação que ainda está em execução, a nova transação é serializada. Caso contrário, a transação verifica se não há conflitos do tipo *second-hop*, e se sim, quais

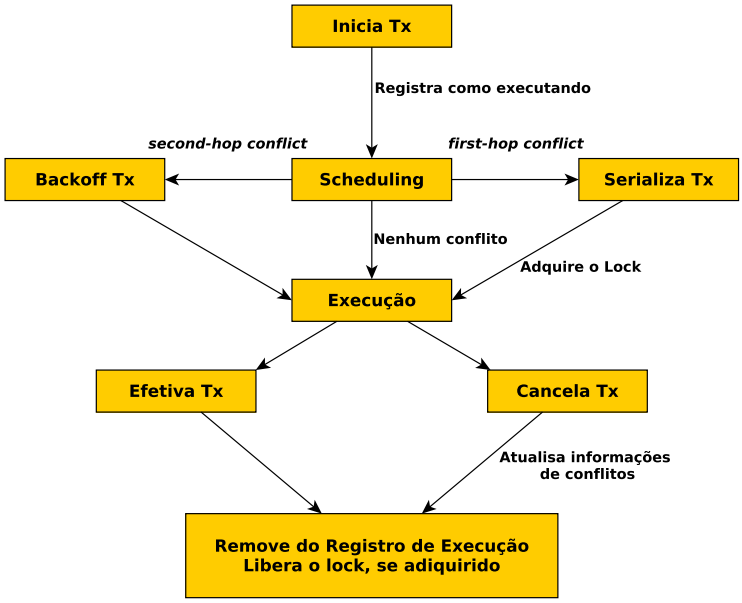

Figura 12 – Flowchar do Escalonador RELSTM

das transações conflitantes ainda estão em execução. Se a quantidade de transações do tipo *second-hop* forem maiores que uma taxa predefinida em `TWOHOP_RATE`, a transação nova é colocada em *back off*. A razão pela qual os dois níveis de conflitos (*one* e *second-hop*) não serem analisados ao mesmo tempo, é de evitar uma excessiva espera de transações que possuem múltiplas relações.

O Algoritmo 3.8 mostra as ações para *commit*, *abort* e *conflict* de RELSTM. Quando duas transações conflitam, a colisão é rastreada e usada como informação para ajuste do escalonador. Primeiro a transação insere seus oponentes em *second-hope*, então a transação que vence o conflito adiciona suas oponentes ao campo *killed*, enquanto que as transações que perderam preenchem seus campos *conflicted*. Após uma efetivação ou um cancelamento, as transações se removem do array de execução e liberam o *lock* global se estas o possuírem.

Algoritmo 3.8 – Pseudo Código do RELSTM para commit, abort e conflitos

```
1  Transaction data:
2  emph{Other transaction information...}
3     ID: Number
4     Twohop: Array          \\list of transaction IDs that had a two-hop conflict
5     Conflicted: Number     \\Transaction ID of the last one-hop lost conflict
6     Killed: Number         \\Transaction ID of the last one-hop won conflict
7     Backoff: Number        \\Current backoff time to wait
8
9  Upon conflict between T1 and T2:
10    if (T2.conflicted is not NULL
11    and T2.conflicted ≠ T1.ID)
12       Add T2.conflicted to T1.twohop
13    if (T2.killed is not NULL
14    and T2.killed ≠ T1.ID
15    and T2.killed ≠ T2.conflicted)
```

```
16      Add T2.killed to T1.twohop
17    if (T1.conflicted is not NULL
18    and T1.conflicted ≠ T2.ID)
19      Add T1.conflicted to T2.twohop
20    if (T1.killed is not NULL
21    and T1.killed ≠ T2.ID
22    and T2.killed ≠ T2.conflicted)
23      Add T1.killed to T2.twohop
24    if (T1 won) T2.conflicted = T1.ID T1.killed = T2.ID
25    else
26      T1.conflicted = T2.ID
27      T2.killed = T1.ID
28    end if
29
30  Upon abort:
31    Unegister transaction in executing array
32    if (queue lock.owned) Unlock queue lock
33
34  Upon commit:
35    Unregister transaction in executing array
36    Empty twohop
37    if (queue lock.owned)
38      Unlock queue lock
```

Para a implementação deste trabalho, foi utilizada a biblioteca TinySTM e as avaliações de desempenho usaram o *benchmark* STAMP. Os resultados obtidos com RELSTM mostram que na maioria dos casos RELSTM apresenta melhor desempenho que a implementação original da TinySTM. Porém, quando o número de *threads* é pequeno, o *overhead* imposto pelo escalonamento acabada causando perda de desempenho.

3.9 PROPS

Em (RITO; CACHOPO, 2014) é apresentado o ProPS (*Progressively Pessimistic Scheduling*) que progressivamente reduz a concorrência entre as *threads* quando o nível de contenção do sistema transacional aumenta, sendo este escalonador uma simplificação do modelo visto na Subseção 3.3.

O modelo proposto coleta informações em tempo de execução para evitar que transações sejam serializadas desnecessariamente. Para permitir este escalonamento mais fino, ProPS mantém uma matriz de nível de concorrência CL (*Concurrent Level Matrix*), que contém informações do nível de concorrência entre um par de transações, isto é, para cada operação atômica do tipo i e cada operação atômica do tipo j, o valor de C_ij descreve quantas transações com operações do tipo i podem ser executadas concorrentemente com transações que contenham operações do tipo j.

No incio, todos os valores da matriz são iguais a `MAX_THREADS`, que corresponde a quantidade de processadores que a CPU possui. ProPS usa o valor de CL para decidir o nível de concorrência do sistema.

Quando uma transação vai realizar uma operação atômica do tipo i, o escalonador calcula o mínimo valor de CL_ij entre a transação que se iniciam e as que estão em execução. Operações atômicas com um valor de CL igual a `MAX_THREADS` prosseguem normalmente. No entanto, conforme o valor de CL diminuí, ProPS reduz a quantidade de transações que podem executar operações to tipo i. Quando a transação do tipo i cancela devido a um conflito com outra transação do tipo j, ProPS reduz o nível de concorrência entre este par de operações usando a Equação 3, onde k é um valor entre 0 e 1:

$$CL_{ij} = CL_{ij} \times k \tag{3}$$

Quando uma transação do tipo i consegue efetivar, para cada operação do tipo j associada na matriz, ProPS atualiza os valores usando a Equação 4, onde α é um valor entre 0 e 1, $\texttt{numRestarts} \geq 0$ corresponde ao número de vezes que a transação se reiniciou antes de conseguir efetivar.

$$CL_{ij} = min(MAX\_THREADS, CL_{ij} + MAX\_THREADS \times \alpha \div (1 + numRestarts)) \tag{4}$$

Assim, ProPS reduz exponencialmente o nível de concorrência quando o sistema se encontra em um alto nível de contenção, porém aumenta linearmente a concorrência quando a quantidade de efetivações aumenta, isto permite que o escalonador atue de forma rápida quando em alta contenção e consiga analisar de forma constante o nível de concorrência quando as transações começam a efetivar. O pseudo código do escalonador pode ser visto no Algoritmo 3.9. O escalonador é completamente descentralizado e permitindo que cada *thread* tome decisões sobre prosseguir ou esperar.

Algoritmo 3.9 – Implementação de ProPS

```
1  static double[][] CL; static TxInfo[] txs; TxInfo myInfo
2
3  upon tx.begin:
4    myInfo.id = tx.id; myInfo.numRestarts = 0
5    do
6      cl = MAX_THREADS; enemies = 1; worstEnemy = nil
7      for each inFlightTx in txs do
8        if (CL[tx.id][inFlightTx.id] < cl)
9      cl = CL[tx.id][inFlightTx.id]; enemies = 1; worstEnemy = inFlightTx
10       else if (inFlightTx == worstEnemy)
11     ++enemies
12   while (cl ÷ enemies < 1)
13   limitConcurrency(cl ÷ enemies)
14
15 upon tx.abort caused by enemyTx:
16   myInfo.numRestarts++
17   CL[enemyTx.id][myInfo.id] = CL[enemyTx.id][myInfo.id] × k
```

```
18
19 upon tx.commit:
20   txs[myInfo.pos] = nil
21   for each opId in atomicOperations do
22     CL[myInfo.id][opId] = min(MAX_THREADS,
23       CL[myInfo.id][opId] + MAX_THREADS × α ÷ (1 + myInfo.numRestarts))
```

Cada *thread* armazena localmente em $TxInfo$ informações sobre a transação que está sendo executada por ela, bem como o ID da operação atômica desta transação e o número de vezes que esta transação cancelou. Em CL armazena o nível de concorrência entre os pares de transações, como descrito anteriormente, e txs armazena todas as transações que estão sendo executadas concorrentemente no sistema.

Quando o sistema se inicia, o escalonador atualiza as informações em $TxInfo$ com os dados da nova transação (Linha 4) e usa CL para calcular o cl (*concurrent level*), dependendo do ID da operação e da configuração atual do sistema (Linhas 5 a 12). A função `limitConcurrency` tenta inserir a nova transação em txs, esperando caso não consiga uma posição livre. Quando a transação consegue ser inserida em txs, esta pode iniciar sua execução. Assim que a transação termina, esta libera sua posição em txs (Linha 20). Os acessos a matriz CL não são sincronizados porque os autores do trabalho assumem que é melhor existir um pequeno nível de imprecisão do que onerar o sistema com a sincronização, evidenciando que esta imprecisão não traz problemas semânticos a execução do escalonador.

ProPS é implementado sobre a biblioteca FlashBackSTM (RITO; CACHOPO, 2013) e executado sobre os *benchmarks* STAMP e STMBench7. Também é realizada uma comparação entre os escalonadores ATS (Seção 3.3), CAR-STM (Seção 3.4) e SHRINK (Seção 3.5), onde os valores de $k = 0.5$ e $\alpha = 0.05$ são usados. Os resultados demostram que o uso de ProPS com o STMBench7, se apresenta mais escalável que qualquer uma das outras implementações. Os resultados em comparação ao *benchmark* STAMP, também demostram melhor desempenho, sendo que no geral os autores demostram que o ProPS alcança ganhos de até 40% se comparado as demais implementação de escalonadores e também com a biblioteca sem escalonamento.

3.10 PROVIT

Dando segmento ao trabalho da seção anterior, os autores apresentam um novo modelo de escalonamento intitulado ProVIT (*Progressive Very Important Transactions*) (RITO; CACHOPO, 2015). Neste trabalho é apresentado um escalonador que trabalha de duas formas distintas, trata transações curtas em um modelo de serialização e transações longas com uma relação de prioridades (VIT).

O pseudo código do ProVIT pode ser visto no Algoritmo 3.10. Novamente como o escalonador anterior, este é completamente descentralizado, assim cada *thread* de-

cide seu próprio escalonamento.

Algoritmo 3.10 – Implementação de ProVIT

```
1  static TxStats TX_STATS, VITList VIT_LIST, int VIT_THRESHOLD;
2  TxInfo myInfo;
3
4  upon tx.begin:
5    myInfo.id = tx.id;
6    myInfo.expectedRS = φ;
7    myInfo.numRestarts = 0;
8    limitConcurrency(TX_STATS.WW[myInfo.id]);
9
10 upon tx.abort caused by enemyTx:
11   myInfo.numRestarts++;
12   TX_STATS.registerConflict(myInfo.id, enemyTx.id);
13   if (!myInfo.isVIT && #tx.readSet < VIT_THRESHOLD) return;
14   if (!myInfo.isVIT) VIT_LIST.add(myInfo);
15   myInfo.expectedRS = myInfo.expectedRS ∪ tx.readSet;
16
17 upon tx.preCommit:
18   Iterable < TxInfo > olderVITs = myInfo.isVIT
19     ? VIT_LIST.olderVITsView(myInfo)
20     : VIT_LIST;
21   for (TxInfo VIT: olderVITs) {
22     if (TX_STATS.conflicted(myInfo.id, VIT.id)
23       && #VIT.expectedRS > #tx.readSet
24       && tx.writeSet ∩ VIT.expectedRS ≠ φ;) {
25         waitForCommit(VIT);
26     }
27   }
28 upon tx.commit:
29   releaseExecutionToken();
30   VIT_LIST.remove(myInfo);
31   updateWastedWork(myInfo.numRestarts);
```

ProVIT guarda informações globais do sistema no objeto `TX_STATS`, e informações locais por *thread* no objeto `myInfo`. `TX_STATS` armazena informações sobre a quantidade de trabalho perdido (WW) de cada transação i e também uma lista de quais transações já conflitaram ao menos uma vez. No objeto `myInfo`, armazena o `id` da transação em execução pela *thread*, o *readset* esperado pela transação, a quantidade de reinícios dessa transação e se esta é uma VIT ou não. Também, o objeto `VIT_LIST`, armazena informação de todas as VITs que estão em execução no sistema.

O cálulo da quantidade de trabalho perdido (WW) é feito pela Equação 5, onde WW é a quantidade de trabalho perdido, k é uma constante pré definida e $numRestarts$ retorna a quantidade de vezes que a transação foi reiniciada. A constante k define o que é mais relevante para o cálculo da quantidade de trabalho perdido, se o histórico de cancelamentos e ou a quantidade de cancelamentos atual.

$$WW_i = WW_i \times k + numRestarts \times (1 - k) \tag{5}$$

Quando uma transação inicia (`tx.begin`, linha 4). esta alimenta as informações do

objeto `myInfo` com os dados da transação e logo apos, usa o valor de WW na função `limitConcurrency`. Esta função pode atrasar o inicio de uma transação porque esta à força a adquirir um *token* de execução em um *array* de tamanho fixo. Este *array* geralmente tem como tamanho a quantidade de *cores* que o processador possui.

Para transações curtas, `limitConcurrency` usa a Equaçao 6 para definir a quantidade de *tokens* que essas podem adquirir.

$$LimitsTxs(WW) = \begin{cases} \frac{S-1}{RL^2} \times (WW - RL)^2 + 1 & \text{se} \quad 0 \leq WW < RL \\ 1 & \text{se} \quad WW \geq RL \end{cases} \tag{6}$$

$LimitsTxs$ retorna um valor entre 1 e S, indicando quantas posições no *array* podem ser usadas pela *thread* para iniciar uma transação. Com um valor de RL igual a 1, o escalonador se comporta similar ao ProPS. Valores altos de RL podem fazer o ProVIT se tornar menos pessimista. Independente do valor de RL, se o valor do WW de uma transação for 0, então esta pode adquirir qualquer posição do *array*. Quando o valor de WW aumenta, o escalonador gradativamente reduz a quantidade de posições livres no *array* que podem ser adquiridas, chegando ao ponto de uma serialização total do sistema. Já para transações longas não existe nenhuma restrição, elas sempre podem adquirir uma posição livre do *array* (mesmo havendo limitação para as transações curtas) e sempre que o sistema não esteja sobrecarregado ou seja, se o número de transações de leitura e escrita é menor que o número de *cores* da CPU.

Quando uma transação é cancelada devido a um conflito (`tx.abort`, linha 10), o escalonador incrementa o número de re-inícios e registra o conflito entre as operações atômicas em `TX_STATS` (Linha 12), antes de decidir se deve reiniciar a transação novamente (Linha 13) ou promove-lá a uma VIT (linha 14). O *readset* esperado sempre é atualizado a cada reinicio da transação (linha 15). O escalonador define uma transação como VIT quando elas apresentam um *readset* grande, ou seja, acima do valor definido em `VIT_THREAHOLD`. Para tornar-se uma VIT, a transação adiciona as informações de `myInfo` a `VIT_LIST`.

No momento da efetivação (`tx.preCommit`, linha17), a transação interage com `VIT_LIST` para verificar se esta não esta gerando um conflito com outra VIT em execução. Para isto o escalonador verifica se existe uma intersecção do *write-set* da transação com o *read-set* das outras VITs. Transações normais (que não são VIT) executam este teste com todas as outras VITs em execução, enquanto que VITs somente verificam as VITs mais antigas (Linhas 18 a 20). Se a intersecção entre os conjuntos não for vazia, as transação oponentes esperam até que a transação VIT seja efetivada (Linha 25). Após a efetivação (`tx.commit`, linha 28), a transação perde seu status de VIT, removendo seu `myInfo` de `VIT_LIST` e o escalonador atualiza os valores em WW usando o valor encontrado em `numRestarts`. Para evitar o *overhead* da verificação da intersecção dos conjuntos de leitura e escrita, ProVIT implementa uma

otimização baseada na observação passada. Uma transação que está no processo de efetivação somente realiza o processo com as VITs com quem esta conflitou no passado (Linha 22).

ProVIT também é implementado sobre a biblioteca FlashBackSTM e é testado com os *benchmarks* STAMP e STMBench7. Novamente, assim como o escalonador da seção anterior, ProVIT é comparada a outros escalonadores que são ATS, CAR-STM, SHRINK e ProPS. Os resultados obtidos mostram que o escalonador apresenta melhor escalabilidade que as outros escalonadores (ATS, CAR-STM e SHRINK) e tem desempenho similar ou melhor que ProPS, dependendo do domínio de operações das transações (se o sistema é dominado por leituras e ou escritas).

3.11 Outros Trabalhos

Trabalhos mais recentes na área de escalonamento de transações têm abordado técnicas mais elaboradas. Estes trabalhos envolvem, para a tomada de decisão do escalonador, técnicas baseadas em aprendizado de máquina (RUGHETTI et al., 2015), modelos híbridos de memórias transacionais em software e hardware (Zhou et al., 2016; CHEN et al., 2020) ou ainda baseada em modelos, como cadeias de markov (Di Sanzo et al., 2020). Outros trabalhos ainda exploram o escalonamento sobre outras arquiteturas como a NUMA (*Non Uniforme Memory Access*) (MOHAMEDIN et al., 2016)

Como estes trabalhos fogem do escopo da proposta desta dissertação, que é o uso de algoritmos de escalonamento a nível de usuário e em software, os mesmos não foram cogitados para serem usados como base para este trabalho.

3.12 Considerações Finais

Os trabalhos estudados aqui apresentaram diferentes heurísticas de escalonamento para memórias transacionais, algumas reativas e outras preditivas, porém o objetivo final é serializar transações conflitantes para evitar novos conflitos. Serializar transações que estão fadadas a serem canceladas evita que estas ocupem recursos computacionais de forma ineficiente, melhorando o desempenho geral do sistema.

Outro item observado nos trabalhos é o fato de que os escalonadores apresentados limitam o número de *threads* executando código transacional ao número de recursos (*cores*) disponíveis na CPU, evitando conflitos entre *threads* instanciadas no mesmo *core* do processador. O escalonador do próprio sistema operacional pode fazer com que duas *threads* ou mais acabem conflitando devido a troca de contexto. Por exemplo, digamos que um *thread* T_1 está dentro de um bloco transacional quando o escalonador do *kernel preempta* este *thread* e escalona outro, digamos T_2 que tam-

Tabela 1 – Comparativo entre os modelos de escalonadores.

Escalonador	Modelo	Biblioteca	Versionamento	Linguagem	Benchmark
ACC	reativo	DSTM2	Adiantado	Java	Apli. própria
PoCC	reativo	DSTM2	Adiantado	Java	Lee-TM/STAMP
ATS	reativo	RSTM	Adiantado e Atrasado	C/C++	Apli. própria
CAR-STM	preditivo	RSTM	Adiantado e Atrasado	C/C++	STMBench7
SHRINK	preditivo	Tiny/Swiss	Adiantado/ Misto	C/C++	STAMP/STMBench7
TRHOTTLE	reativo	Tiny	Adiantado	C/C++	STAMP
LUTS	reativo/preditivo	Tiny/Swiss	Adiantado/ Misto	C/C++	STAMP/STMBench7
RELSTM	preditivo	Tiny	Adiantado	C/C++	STAMP
PROPS	preditivo	FlashBack	Atrasado	Java	STAMP/STMBench7
PROVIT	preditivo	FlashBack	Atrasado	Java	STAMP/STMBench7

bém executa código transacional. Se ambas T_1 e T_2 acessam dados comuns, ambas poderão conflitar devido a troca de contexto. Trabalhos como ATS, CAR-STM e LUTS demostram preocupação com estas ações.

Alguns trabalhos utilizam análise reativa, ou seja, somente tomam uma ação de escalonamento frente a um conflito. Outros trabalhos já abordam uma ação preditiva, tentando prever possíveis conflitos futuros, porém inserindo um custo de instrumentação ou análise de regiões de memória para prever os conflitos. Geralmente tais tarefas acabam sendo realizadas em estruturas compartilhadas entre os *threads*, gerando gargalos nos sistemas.

Apesar do uso de heurísticas simples, no geral os escalonadores acabam melhorando o desempenho das aplicações, de forma mais notável quando o número de *threads* é grande. Isso provavelmente está ligado ao fato de que o uso mais racional dos recursos computacionais acaba suprimindo os custos inseridos pelos escalonadores ao sistema transacional, ou seja, o custo do escalonamento é suprimido pela quantidade de trabalho útil gerado pelas transações.

A Tabela 1 mostra os diferentes modelos de escalonamento, suas abordagens (preditiva/reativa), quais *benchmarks* foram usados e quais as bibliotecas em que estes foram integrados.

Os primeiros dois trabalhos (ACC e PoCC) na verdade não propõem, de fato, escalonadores. No entanto eles são discutidos neste texto por apresentarem os primeiros esforços em reduzir a quantidade de conflitos para prover ganhos de desempenho. Em ATS é demostrado a primeira heurística prática para o cálculo da intensidade de conflito, esta heurística é retomada de forma adaptada em outros trabalhos como LUTS e ProPS. Também nota-se que, conforme a evolução dos métodos de escalonamento, o modelo passou de reativo para preditivo.

Nos escalonadores LUTS, ProPS e ProVIT, são apresentados modelos adaptáveis de escalonamento ou seja, o escalonador muda de abordagem conforme o tamanho das transações. Para transações curtas, o escalonador assume uma abordagem reativa e para transações longas uma abordagem preditiva. Nestes trabalhos os autores se preocupam em não onerar as transações curtas com instrumentação, o que causaria acréscimo excessivo ao tempo de execução.

Finalizando esta discussão, observa-se uma evolução clara nos modelos de escalonadores, partindo de heurísticas simples, como o caso da ATS, até sistemas híbridos como por exemplo a ProVIT. Entretanto, nota-se na Tabela 1, que a ideia de escalonar transações é geralmente investigada somente em linguagens imperativas, usando no máximo dois algoritmos de STM.

4 IMPLEMENTAÇAO DOS ESCALONADORES DE TRANSAÇÕES EM HASKELL

Este capitulo descreve a implementação de três algoritmos de escalonamento selecionados depois da revisão de literatura apresentada no capítulo anterior. Os algoritmos escolhidos foram o ATS, CAR-STM e ProVIT. Os algoritmos ATS e CAR-STM foram escolhidos por servirem de base para vários outros trabalhos, como por exemplo, SHIRINK, TRHOTTLE, LUTS, ProPS e ProVIT.

O algoritmo ProVIT foi escolhido pois utiliza uma abordagem híbrida, usando diferentes algoritmos de escalonamento apresentados na literatura, dependendo do comportamento do sistema transacional.

Os escalonadores implementados foram integrados a três bibliotecas de STM para Haskell. Essas bibliotecas contemplam as abordagens mais usadas no *design* de sistemas de STM, e também foram desenvolvidas completamente em Haskell, i.e., TL2 (DU BOIS, 2011) que usa versionamento de dados tardio, TINY (DUARTE et al., 2015) que usa versionamento adiantado, e SWISS (DU BOIS; PILLA; DUARTE, 2012) que usa uma técnica de versionamento misto. Todas as bibliotecas usam um gerenciador de contenção tímido.

Os escalonadores foram implementados em Haskell, usando as extensões de programação concorrente disponíveis no compilador GHC (PEYTON-JONES; MARLOW et al., 2020), que é o compilador estado da arte para Haskell. Como os escalonadores de transações foram implementados a nível de usuário, esses coexistem com o escalonador do sistema de tempo de execução do GHC. Esse escalonador mantém uma fila de *threads* para cada *core*, que são escalonados usando o modelo *round robin*. Entre os *cores* os *threads* são distribuídos por *work stealing*.

4.1 Escalonador Usando o Modelo de Migração - CAR-STM

Este modelo de escalonador usa uma técnica baseada no algoritmo de CAR-STM 3.4, que impõe baixo *overhead* sobre as bibliotecas de STM. Neste modelo de escalonador para cada *core* existente no processador, um *thread* é fixado. Es-

ses *threads* possuem a função específica de executar transações oriundas de *threads* que tiveram suas transações abortadas, ou seja, as mesmas recebem transações e retornam o resultados destas para as *threads* de origem. O diagrama básico do escalonador pode ser visto na Figura 13.

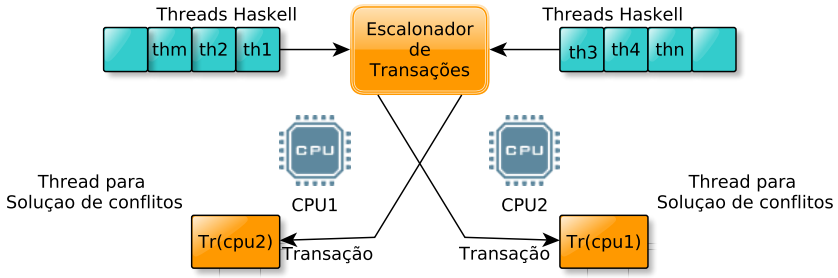

Figura 13 – Estrutura do escalonador desenvolvido.

Neste exemplo, é apresentada uma máquina com dois processadores. Cada processador possuí uma fila de *threads* Haskell, que são gerenciadas pelo escalonador do sistema de tempo de execução (filas azuis). O escalonador de transações implementado, mantém para cada *core* um *thread* fixo, que executa as transações conflitantes, na Figura, são as *threads Tr(cpu(n))*. Cada *Tr* possui uma fila de tarefas que recebe as transações conflitantes oriundas do escalonador de transações. As transações presentes nesta fila são resolvidas por uma regra do tipo FIFO, ou seja, a primeira transação a chegar é a primeira a ser resolvida.

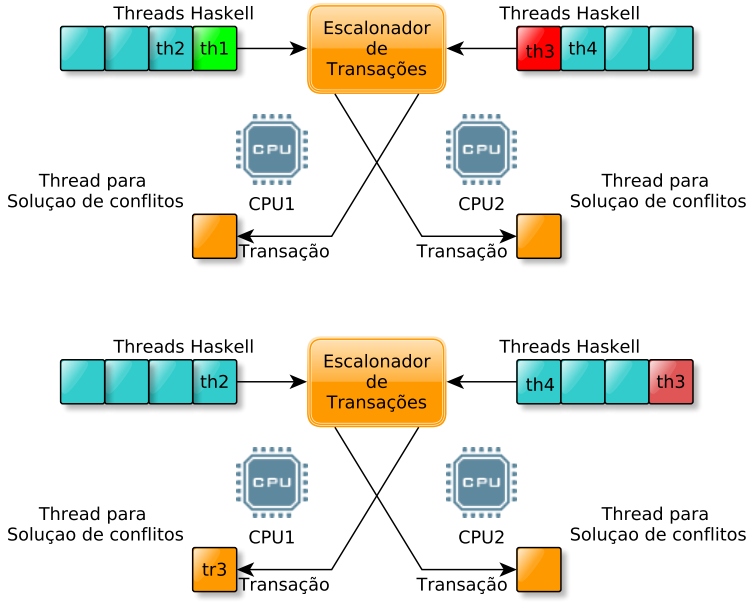

Figura 14 – Passos de escalonamento de transações.

Na Figura 14 são apresentados os passos de ação do escalonador, em um cenário onde temos dois *threads* sendo executadas de forma concorrente (*th1* e *th3*).

Neste cenário, quando o escalonador de transações verifica que a transação de *th3* conflitou com uma transação que estava presente no processador 1 (*th1*), este migra a transação de *th3* (*tr3*) para a *Tr* do processador 1 e coloca *th3* para o final da fila de execução de processador 2. Dessa forma, como as duas transações estão sendo executadas no mesmo processador, a chance delas conflitarem diminui. Assim que a transação for concluída no processador 1, a mesma retorna seu resultado para a *th3* desbloqueando a mesma. As *threads* que tiveram suas transações migradas, são colocadas no fim da fila de execução do escalonador do GHC. Isso se deve ao fato de que a *thread* que teve a sua transação migrada, fica em uma espera por bloqueio até que a sua transação tenha sido resolvida. Assim, estas são colocadas para o fim da fila de execução para que outros *threads* tenham a chance de executar suas operações.

A Figura 15 apresenta o código Haskell para a inicialização do escalonador. São definidos dois tipos de dados: `ListOfTasks`, que representa a fila dos `Trs` de cada core, e `CPUs`, que guarda em um `array` a fila de cada *core*. As filas são implementadas usando o tipo `Chan` do Haskell, que é uma fila *thread safe*.

```
1  type ListOfTasks = Chan (IO ())
2  type CPUS = Array Int ListOfTasks
3
4  schedulerInit :: IO ()
5  schedulerInit = do
6      nCpus <- getNumProcessors
7      setNumCapabilities nCpus
8      slots <- replicateM nCpus (newChan)
9      let cpus = listArray (0,nCpus-1) slots
10     writeIORef listOfCpus cpus
11     startThreads (nCpus-1) cpus
12
13     startThreads :: Int -> CPUS -> IO ()
14     startThreads (-1) _ = return ()
15     startThreads cap listOfCPU = forkOn cap (getWork $ listOfCPU ! cap)  >>
           startThreads (cap-1) listOfCPU
```

Figura 15 – Inicializando o escalonador

Quando o escalonador é inicializado, primeiramente é lido o número de *cores* presentes na máquina (`getNumProcessors` - Linha 6). `setNumCapabilities` é usado para forçar o sistema de tempo de execução do Haskell a usar todos os cores da máquina. O *array* de filas é criado (Linha 9) e depois são inicializadas os *threads* que executam as transações migradas (*Trs*) (Linha 11). A função `startThreads` (Linha 13), inicializa todos os *threads Trs* fixando estes a uma posição do *array* de filas e a seu respectivo *core* por meio da função `forkOn`.

Cada *thread Tr* executa as transações recebidas com a função `getWork`, Figura 16. Esta retira uma transação da fila (Linha 3), executa a mesma e passa para a próxima transação da fila. Caso não exista transações na fila, a função `readChan` bloqueia esperando um trabalho ser inserido.

```
1       getWork :: ListOfTasks -> IO ()
2       getWork listOfTasks = do
3       work <- readChan listOfTasks
4       work
5       getWork listOfTasks
```

Figura 16 – Inicializando o escalonador

Quando uma transação aborta devido a um conflito, o escalonador de transações precisa saber para onde esta transação deve ser migrada. Para isso, a estrutura interna das variáveis transacionais (Seção 2.4) do STM Haskell foi modificada para conter a informação do processador onde está o *thread* que fez a última modificação nessa variável (Figura 17). Dessa forma, quando uma transação detecta um conflito em uma varável transacional, o escalonador sabe para qual processador essa transação deve ser migrada.

```
1  data TVar a = TVar
2  {
3       (...)
4       lastCpuCommit :: IORef Int
5  }
```

Figura 17 – Estrutura de dados da TVar.

A outra modificação está na forma como as bibliotecas de STM tratam um cancelamento (*abort*). Quando a transação cancela, ela não realiza o *roll back* e reinicializa a sua execução. Na modificação, ela passa a sua transação para o escalonador que migra esta para o processador que continha a transação com quem esta conflitou, por meio da função `sendWork` (Figura 18). Para realizar isso, a transação verifica se conflitou, pelos métodos já existentes nos algoritmos originais das bibliotecas (Linha 14), porém agora, a transação lê o campo `lastCpuCommit` para verificar qual o último processador que conseguiu efetivar a transação, realiza o *roll back* desfazendo as modificações (Linha 15), encapsula a transação em uma nova operação de IO e informa ao escalonador para qual processador a transação deve ser migrada, ou seja, no processador com `id` igual ao lido no campo `lastCpuCommit`.

O escalonador agora coloca a *thread* que teve sua transação migrada para o final da fila de execução e busca outra *thread* para execução. Quando a transação migrada é executada e consegue realizar *commit*, o resultado de sua execução é devolvido para a *thread* proprietária da mesma, liberando-a para voltar à execução.

4.2 Escalonador usando o Modelo de Serialização - ATS

O segundo escalonador desenvolvido usa como base o algoritmo do ATS 8, que serializa todas as transações que conflitaram em uma única fila de execução. Este

```
1  sendWork :: STM a -> Int -> IO a
2  sendWork stmAct cpuId = do
3      cpus <- readIORef listOfCpus
4      newPtr <- newEmptyMVar
5      writeChan (cpus!cpuId) (atomically stmAct >>= (\x -> putMVar newPtr x))
6      yield
7      takeMVar newPtr
8
9  atomically :: STM a -> IO a
10 atomically stmac@(STM ac) = do
11     ts <- newTState
12     r <- ac ts
13     case r of
14         Invalid nts hec -> do
15             clean nts
16             cpu <- getCurrentCPU
17         result <- sendWork stmac cpuId
18             return result
```

Figura 18 – Modificações na biblioteca de STM

modelo também apresenta baixa complexidade e assim não insere muito *overhead* às bibliotecas de STM.

O escalonador foi implementado modificando-se o escalonador da Seção anterior. O escalonador utiliza apenas um *thread*, fixado a um dos *cores*, e que possuí uma fila que recebe todas as transações que foram cancelas.

Toda a transação cancelada, independente do *core* de origem, é colocada na fila de transações conflitantes. Assim como no modelo anterior, o *thread* que teve sua transação migrada é colocado no final da fila do escalonador do sistema de tempo de execução.

A execução das transações conflitantes segue a ordem imposta pela fila, a primeira transação a entrar na fila é a primeira a sair, dessa forma impondo a serialização dessas transações.

4.3 Escalonador usando o modelo Híbrido em Haskell - ProVIT

Para o desenvolvimento deste escalonador usou-se como base o algoritmo ProVIT (Seção 3.10). Esse algoritmo usa uma abordagem híbrida, dando um tratamento diferente às transações dependendo do tamanho. Transações pequenas, são tratadas de maneira parecida com o ATS, ou seja, o paralelismo é restringido. Transações grandes, que devem reexecutar muito trabalho quando são canceladas, podem se beneficiar com uma heurística mais cara, essas transações são chamadas de VIT (*Very Important Transactions*).

Para as transações curtas, o algoritmo trabalha com uma heurística baseada na ideia de trabalho perdido. Cada vez que uma transação reinicia, ela incrementa a quantidade global de trabalho perdido do sistema. Conforme a quantidade de trabalho perdido aumenta, a quantidade de paralelismo que pode ser explorada pelo sistema

transacional é reduzida. A intuição aqui explorada é a de que, reduzindo o paralelismo no sistema, as transações conflitam menos, consequentemente gerando menos cancelamentos.

A redução de paralelismo é realizada por um processo de aquisição de *tokens*. Cada *thread*, para poder prosseguir a execução de sua transação, deve adquirir um *token*, e a cada momento o escalonador libera mais ou menos *tokens* dependendo da quantidade global de trabalho perdido. A quantidade máxima de *tokens* é limitada pela quantidade de *cores* do sistema.

A forma com que os *tokens* são liberados é definida pela Equação 7. Esta equação limita a quantidade de *tokens* que o sistema transacional pode disponibilizar, de acordo com o trabalho perdido. Quanto maior for o valor da quantidade de trabalho perdido, menos *tokens* são liberados, levando a uma serialização do sistema.

$$LimitsTxs(WW) = \begin{cases} \frac{S-1}{RL^2} \times (WW - RL)^2 + 1 & \text{se} \quad 0 \leq WW < RL \\ 1 & \text{se} \quad WW \geq RL \end{cases} \tag{7}$$

Nesta equação, WW é a quantidade de trabalho perdido pelas transações, S é a quantidade de cores que o processador possui e RL é um fator do escalonador que o torna mais ou menos pessimista. Para valores de RL grandes, o escalonador se torna menos pessimista, ou seja, considera que mesmo que o número de cancelamentos cresça, o sistema não ira reduzir o nível de paralelismo. Já para um valor de RL pequeno, o escalonador se torna mais pessimista, limitando a concorrência ao se obter valores menores de cancelamentos.

Para transações longas, é utilizado o conceito de VIT. Transações longas são propensas a abortar devido a conflito com transações menores, levando a *starvation*. Assim, no conceito de VIT, transações longas tem prioridade sobre transações curtas e transações de somente leitura tem prioridade sobre transações de escrita/leitura. Também, para evitar problemas de *livelock*, toda a transação eleita como VIT têm uma ordem cronológica de prioridade, ou seja, transações eleitas como VIT mais antigas têm prioridades de efetivação sobre as mais novas. Uma transação é eleita como VIT quando seu `readset` alcança um valor pré definido.

A estrutura `TXStats`, que armazena os dados globais do escalonador, pode ser vista na Listagem 19.

```
1  type Enemys = IORef (Set (Integer,Integer))
2
3  data TXStats = TX{
4      ww :: IORef Integer,
5      enemys :: Enemys
6  }
```

Figura 19 – Estrutura Global TXStats

Nesta estrutura são armazenados a quantidade de trabalho perdido (ww) e um conjunto que armazena quais transações já conflitaram ao menos uma vez. Toda vez que uma transação conflita com outra, os respectivos `ids` destas transações (a que conflitou e a conflitante) são armazenados em uma tupla, registrando que houve o conflito. Este conjunto serve para a verificação de conflitos entre VITs onde, para reduzir a quantidade de verificação entre os conjuntos de leitura e escrita durante a fase de sincronização entre VITs, somente é verificado as VITs que tem relações sobre este conjunto. Essa operação é explicado mais adiante na Subseção 4.3.3.

A próxima estrutura é a `MYInfo`, onde são armazenados dados locais da *thread* e da transação que a *thread* está executando. Esta estrutura pode ser vista na Figura 20.

```
1  type RS = IORef (Set Integer)
2
3  data MYInfo = MI{
4       id :: Integer,
5       numRestart :: IORef Integer,
6       expectedRS :: RS,
7       isVit :: IORef Bool,
8       lock :: IORef Bool
9  }
```

Figura 20 – Estrutura Local `MYinfo`

Na estrutra são mantidos o `id` da *thread*, que será único para cada *thread* do sistema, a quantidade de reinícios que a transação executada pela *thread* sofreu (`numRestarts`). É também armazenado o *readset* esperado para essa transação, sendo este campo atualizado a cada reinício da transação, pois se a transação leu determinados tipos de dados a probabilidade de ela acessar esses mesmos dados na próxima execução é grande. Também possui dois campos, um com uma *flag*, que indica se a transação é uma VIT e outro que é um espécie de *lock*, usado para realizar a sincronização de execução das VIT. O *lock* faz com que determinada VIT espere que a outra mais antiga se efetive para que ela possa continuar.

Por último, a estrutura global `Vitlist`, armazena a lista de todas as VITs que estão ativas no sistema. Esta estrutura pode ser vista na Figura 21.

```
1  type Vitlist = [(Integer,MYInfo)]
2
3  vitlist :: IORef Vitlist
4  vitlist = unsafePerformIO (newIORef [])
```

Figura 21 – Estrutura Global Vitlist

O escalonador opera basicamente em três fazes diferentes do algoritmo de STM. A primeira no inicio da transação, a segunda durante um cancelamento e a última na fase de efetivação da transação. Essas ações são descritas nas próximas Seções.

4.3.1 Inicio de uma Transação

Na fase de início (Figura 22), os dados da estrutura `MyInfo` são lidos pela transação sendo inciada e então é chamada a função `limitConcurrency`, que usa a Equação 7 para liberar um *token* para a transação. Dessa forma, a transação fica bloqueada na chamada ao *limitConcurrency* até que um *token* fique disponível.

```
1  -- transaction begin
2  txstats <- readIORef txStats
3  wk <- readIORef (ww txstats)
4  v <- readIORef (isVit myinfo)
5  token <- limitConcurrency v wk
```

Figura 22 – Incio da Transação

4.3.2 Fase de Cancelamento

Se uma transação é cancelada devido a um conflito, (Figura 23), além das operações específicas de cancelamento do algoritmo de STM sendo usado, a transação deve liberar o *token* adquirido (Linha 2), incrementar o seu número de reinícios e inserir na lista de transações conflitantes seu `id` e o `id` da transação com quem conflitou. Após isso, se a transação não é uma VIT e também não possui um *readSet* com o tamanho suficiente para se tornar uma VIT (Linha 6), o *readset* esperado da transação é atualizado e esta é reiniciada. Se o *readset* é grande o suficiente para que a transação se torne uma VIT, esta é adicionada na lista de VITs, seu *readSet* esperado é atualizado e a transação reiniciada. Se a mesma já é uma VIT, somente é atualizado o *readset* esperado e a transação é reiniciada.

```
1  -- transaction abort
2      putMVar token ()
3      readIORef numRestart >>= (\x -> writeIORef numRestart (x+1))
4      readIORef (enemys txtstats) >>= (\x-> writeIORef (enemys txstats) (Set.insert
           (tid,enemy) x))
5      rs <- readIORef expectedRS
6      if(not v && toInteger(Set.size rs) < vitThreshold) then do
7          incrementReadSet expectedRS rset
8          --restart
9          else do
10         if (not v) then do
11             addVitList (tid,(MI tid numRestart expectedRS isVit lock))
12             incrementReadSet expectedRS rset
13             writeIORef isVIt True
14             --restart
15             else do
16                 incrementReadSet expectedRS rset
17                 --restart
```

Figura 23 – Cancelamento da Transação

4.3.3 Fase de Efetivação

Já na fase de efetivação (Figura 24), inicialmente o *token* do processador é liberado (Linha 2). É lido o *writeset* da transação e a lista de VITs mais antigas que a VIT atual, estes dados são passados a função de Pré Efetivaçào (`preCommit`). A função (`precommit`) verifica se a transação é uma VIT e em caso positivo, verifica se o conjunto de escrita da transação se intersecciona com o conjunto de leitura de outras VITs. Se este for o caso, a transação espera o termino da execução das VITs mais antigas ao qual se interseccionou para poder prosseguir. Após a fase de Pré Efetivação, o *lock* de espera da VIT é liberado (Linha 6), a VIT é removida da lista de VITs, a quantidade de trabalho perdido é atualizada e a transação é finalizada.

```
1  -- transaction commit
2  putMVar token ()
3  wr <- readIORef wset
4  oldvits <- getOldVITS (tid,myinfo)
5  preCommit (tid,myinfo) wr oldvits
6  writeIORef lock False
7  deleteVitList (tid,myinfo)
8  nr <- readIORef numRestart
9  writeIORef (ww txstats) (wk + nr)
10 -- end of transaction
```

Figura 24 – Efetivação da Transação

5 ESPECIFICAÇÃO DO AMBIENTE DE TESTES

Os escalonadores desenvolvidos foram avaliados sobre aplicações com diferentes características, desde aplicações que melhor exploram os aspectos de concorrência usando memórias transacionais, até aquelas em que resultam em baixo desempenho devido ao excesso de conflitos (elevada contenção de memória). O principal objetivo foi averiguar em quais aplicações e sobre quais abordagens de versionamento de dados de MTs a utilização dos escalonadores impactariam no tempo de execução e na quantidade de cancelamentos.

Para a execução dos testes foi utilizado o STM Haskell Benchmark (PERFUMO et al., 2007). Este *benchmark* consta de aplicações que testam diferentes condições de execução de memórias transacionais, entre elas, alta e baixa taxa de conflitos. Os testes realizados se concentraram em nove aplicações deste *benchmark* que são:

- **SI**: Programa que consiste em n *threads* incrementando uma única variável inteira compartilhada. O programa SI testa o comportamento das memórias transacionais sobre ambientes de alta contenção. O uso de uma única variável compartilhada leva a uma serialização no acesso a seção crítica, fazendo com que a taxa de conflitos aumente quando se incrementa o número de *threads*. Os testes realizados usando esta aplicação constaram em 200.000 iterações de incremento da variável compartilhada sobre diferentes quantidades de *threads*.

- **LL**: Em LL são realizadas inserções e remoções em uma lista encadeada usando números aleatórios. Cada *thread* possui duas transações consecutivas, sendo uma para a inserção de um número e outra para a remoção. A lista é testada com 10000 iterações e os números aleatórios podem variar de 0 a 600.

- **BT**: O programa BT segue a mesma ideia do LL, neste também são inseridos e deletados números aleatórios usando duas transações consecutivas nos *thread*. São realizadas 12000 operações de inserção e remoção de números aleatórios na faixa de 0 a 600.

- **HT**: No programa HT, são realizadas inserções, deleções e consultas sobre uma tabela *hash* concorrente. Essas operações são distribuídas da seguinte forma:

10% de deleções, 10% de inserções e 80% de consultas. A cada interação é sorteado um número aleatório para ser inserido na tabela e depois é sorteada a operação que será realizada. Neste programa foram realizadas 1000000 iterações sobre uma faixa de números aleatórios entre 0 e 600.

- **Sud**: Programa que procura a solução de um jogo de Sudoku. A aplicação Sud testa três níveis de complexidade (fácil, médio e difícil). Este programa possui a particularidade de possuir transações que operam sobre linhas, colunas e blocos do jogo, assim a taxa de conflitos tende a aumentar conforme a complexidade do jogo aumenta. Este programa utiliza três *threads*, uma para resolver as linhas, outra para resolver colunas e uma para verificar os blocos.

- **CCHR-UnionFind**: Utiliza um sistema concorrente de manipulação de regras de restrição (CCHR - Concurrent Constraint Handling Rules) para realizar a união de conjuntos distintos, onde os conjuntos são representados por árvores binárias. O CCHR aplica um conjunto de regras até que se obtenha um solução simplificada para um problema. Com o objetivo de alcançar essa simplificação, as aplicações usam uma lista de regras sobre estruturas de dados compartilhadas com sincronização realizada por STM. Como essas regras possuem soluções alternativas, a primitiva orElse do STM Haskell é usada para as simplificações.

- **CCHR-Sudoku**: Utiliza o sistema CCHR para solucionar um jogo de Sudoku.

- **CCHR-Prime**: Utiliza o sistema CCHR para buscar os 4000 primeiros números primos.

- **CCHR-BlockWorld**: Utiliza o CCHR para simular dois agentes autônomos, cada um movendo 100 blocos entre locais sem sobreposição.

Todos os programas foram testados em uma máquina com processador i7 2600 de 4 *cores* físicos + 4 lógicos, com 8Gb de memória RAM e sistema operacional Debian 10 64bits. O compilador Haskell utilizado foi o GHC 8.2.2. Os testes realizados usam 1, 2, 4, 8, 16 e 32 *threads* testando os diferentes escalonadores implementados (migração, serialização e híbrido), combinados com os diferentes algoritmos de STM (versionamento adiantado, tardio e misto).

Foram utilizadas as seguintes *flags* para a compilação de todos os programas:

- -threaded: habilita o uso de *threads* no RTS.

- -rtsopts: habilita a utilização de *flags* do RTS em tempo de execução.

Já para a execução dos programas, foram usadas as seguintes *flags* do RTS:

- -N[cores]: define o número de *cores* que serão usados durante a execução.

Tabela 2 – Valores utilizados para a execução das aplicações do STM Benchmark

Aplicação	Operações	Valores	Heap (H)	Pilha (K)
SI	200000	N/A	50M	50M
LL	10000	600	20M	10M
BT	12000	600	20M	5M
HT	1000000	600	20M	10M
SUD	N/A	N/A	400M	400M
UnionFind	N/A	N/A	200M	200M
BlockWorld	N/A	N/A	200M	200M
Prime	N/A	N/A	400M	200M
Sudoku	N/A	N/A	400M	100M

- -H: define o tamanho da *heap* para ser utilizada pelo RTS do GHC para a aplicação.
- -K: define o tamanho da pilha de execução a ser utilizada pelo RTS do GHC para a aplicação.

A Tabela 2 mostra a lista de valores usados para cada aplicação. Esses valores foram utilizados pois apresentaram os melhores desempenhos nas aplicações sem o uso dos escalonadores. As aplicações que não recebem valores de entrada estão marcadas com (N/A) na tabela.

Os resultados obtidos estão divididos em duas seções. A primeira abordando os resultados das aplicações que não fazem uso das primitivas `retry` e `orElse` e o segundo onde as aplicações usam essas primitivas.

Os três escalonadores desenvolvido tem seus resultados apresentados como segue abaixo:

Nos gráficos apresentados, são usadas as seguintes legendas:

- **TINY**: execução usando a biblioteca de STM com o algoritmo da TINY (versionamento adiantado)
- **TL2**: execução usando somente o algoritmo da TL2 (versionamento tardio)
- **SWISS**: execução usando somente o algoritmo da SWISS (versionamento misto)
- **TINY Migração**: execução usando o escalonador por migração na TINY
- **TL2 Migração**: execução usando o escalonador por migração na TL2
- **SWISS Migração**: execução usando o escalonador por migração na SWISS
- **TINY Ser.**: execução usando o escalonador por serialização na TINY
- **TL2 Ser.**: execução usando o escalonador por serialização na TL2

- **SWISS Ser.**: execução usando o escalonador por serialização na SWISS
- **TINY Hibrid**:execução usando o escalonador híbrido na TINY
- **TL2 Hibrid**: execução usando o escalonador híbrido na TL2
- **SWISS Hibrid**: execução usando o escalonador híbrido na SWISS

Para todas as aplicações foram coletados os tempos de execução e cancelamentos de trinta amostras, calculando a média e o desvio padrão. Também, os cálculos de *Speeup* são relativos a execução das bibliotecas sem escalonamento.

Os valores de RL, k e $vitThreshold$ usados foram respectivamente, 100, 0.3 e 4000. Estes valores foram usados porque demostraram a melhor resposta no algoritmo original da ProVIT.

6 AVALIAÇÃO DE DESEMPENHO DOS ESCALONADORES

Este capítulo divide os resultados obtidos com a aplicação dos escalonadores desenvolvidos em duas partes. A primeira apresentando os resultados das aplicação que não fazem uso de `retry` e `orElse` e não possuem limite para a quantidade de *threads*. São estas aplicações a SI, BT, LL e HT.

A segunda parte apresenta os resultados das aplicações que fazem uso do `retry` e `orElse` e que possuem limite de *threads*. São elas a Sudoku, CCHR-Sudoku, CCHR-UnionFind, CCHR-Prime e CCHR-BlockWorld, que são aplicações limitadas a no máximo de três *threads*.

6.1 Aplicações sem retry/orElse

6.1.1 SI

Os resultados de tempo obtidos com a execução da aplicação SI podem ser vistos no gráfico da Figura 25. Nela podemos observar que, devido a natureza de alta contenção da aplicação, a mesma não apresenta redução nos tempos de execução para quase nenhum dos escalonadores. A única exceção foi na utilização do TINY Hibrid, que até o número físico de *cores* da máquina (quatro *cores*), conseguiu obter redução no tempo. O uso dos escalonadores SWISS Ser. e SWISS Hibrid reduziram o tempo de execução da biblioteca original SWISS, que acabou apresentando os valores mais altos de tempo de execução. Os demais escalonadores após alcançarem a máxima quantidade de *cores* físicos permaneceram com seus tempos de execução com mínimas alterações.

Apesar de apresentar uma redução no tempo de execução, o escalonador TINY Hibrid não apresentou uma redução significativa nos canelamentos, como pode ser observado na Figura 26, onde percebe-se que além do número de cancelamentos aumentar o mesmo oscila bastante, como pode ser observado por seu elevado desvio padrão.

O uso do escalonador SWISS Hibrid evitou o aumento do tempo de execução da SWISS como também manteve uma taxa de cancelamentos quase constante, sendo

que as menores taxas de cancelamentos foram obtidas com a SWISS Ser.

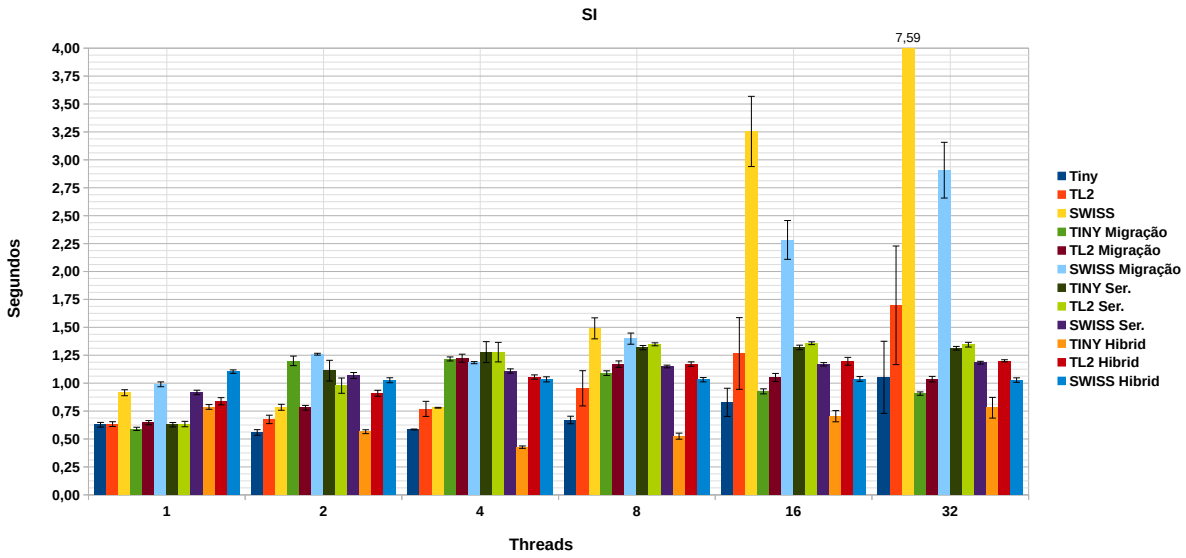

Figura 25 – Tempos de execução da aplicação SI

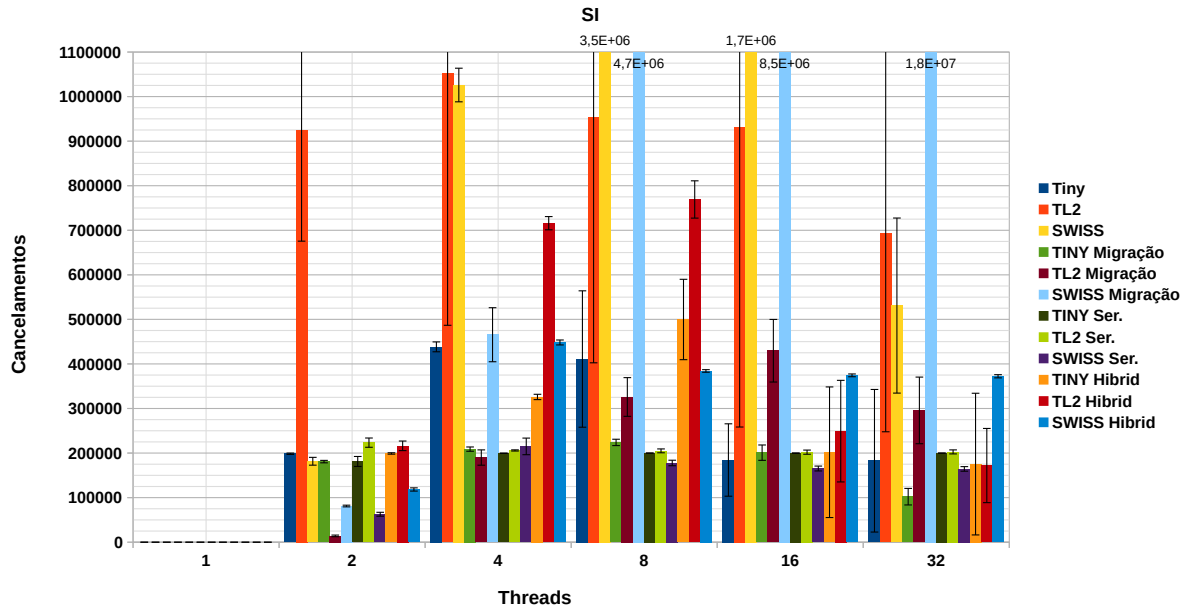

Figura 26 – Quantidade de cancelamentos da aplicação SI

Os escalonadores apesar de não apresentarem ganho no desempenho, o que era esperado devido a natureza de alta contenção da aplicação, acabaram apresentando redução das taxas de cancelamento, o que impactou no tempo de execução da aplicação. Isso pode ser observando quando foi utilizado mais *threads* que cores físicos, onde os escalonadores limitaram o tempo de execução, deixando-os quase constantes. Essa constatação pode ser observada no gráfico da Figura 27, onde observa-se

um SpeedUp quase constante a partir de quatro *threads*, sendo o melhor resultado o da SWISS Híbrido, onde a curva de escalabilidade é quase constante.

O único escalonador que se apresentou escalável até a quantidade de *cores* físicos foi o TINY Hibrid que manteve o aumento de SpeedUp além de sua versão sem escalonamento.

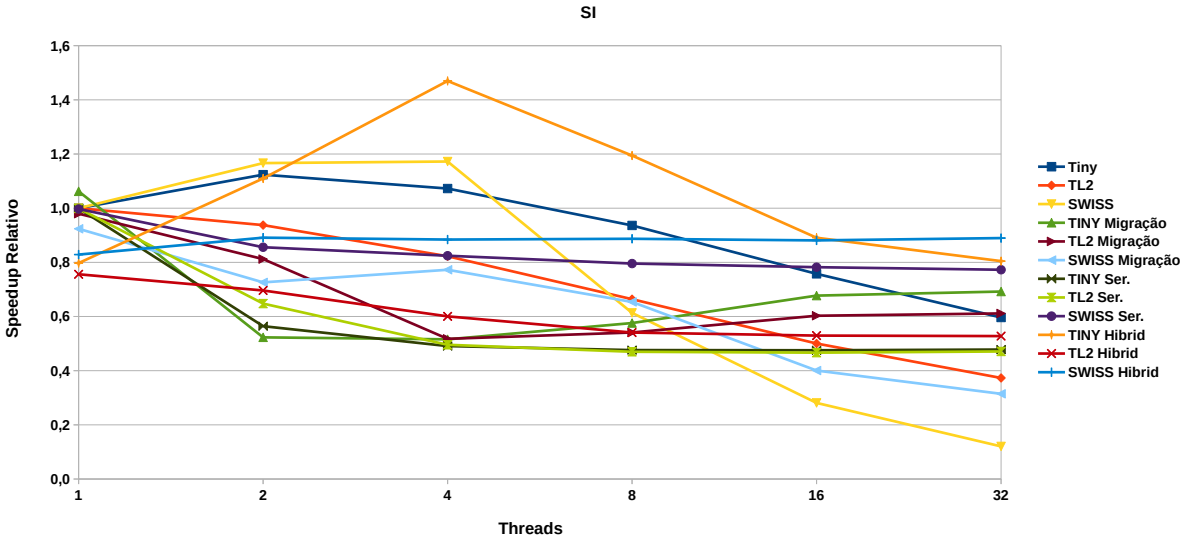

Figura 27 – Curva de escalabilidade relativa da aplicação SI

6.1.2 LL

A aplicação LL apresenta um cenário de média contenção, onde transações inserem e removem valores gerados aleatoriamente em uma lista encadeada concorrente. Nesta aplicação averiguamos o comportamento dos escalonadores sobre este novo cenário de contenção.

Na Figura 28, pode-se observar que os escalonadores em geral, até a quantidade de *cores* físicos, conseguiram reduzir o tempo de execução, com exceção dos escalonadores Híbridos, que devido a sua complexidade de operação acabaram perdendo desempenho. Os escalonadores que apresentaram os menores valores de tempo foram o SWISS Migração e SWISS Ser. onde, com quatro threads, o SWISS Ser. apresentou o melhor resultado.

Os outros escalonadores mantém tempos quase constantes após a quantidade de *cores* físicos ter sido alcançada, o que demostra que os escalonadores não permitiram um aumento no tempo de execução, como ocorreu com as bibliotecas puras e o uso dos escalonadores Híbridos.

Este comportamento por parte dos escalonadores híbridos se da pelo fato de que o elevado tempo de execução do escalonador para concluir as operações de eleição das VITs, levou as transações mais curtas a conflitarem mais seguidamente. Em LL

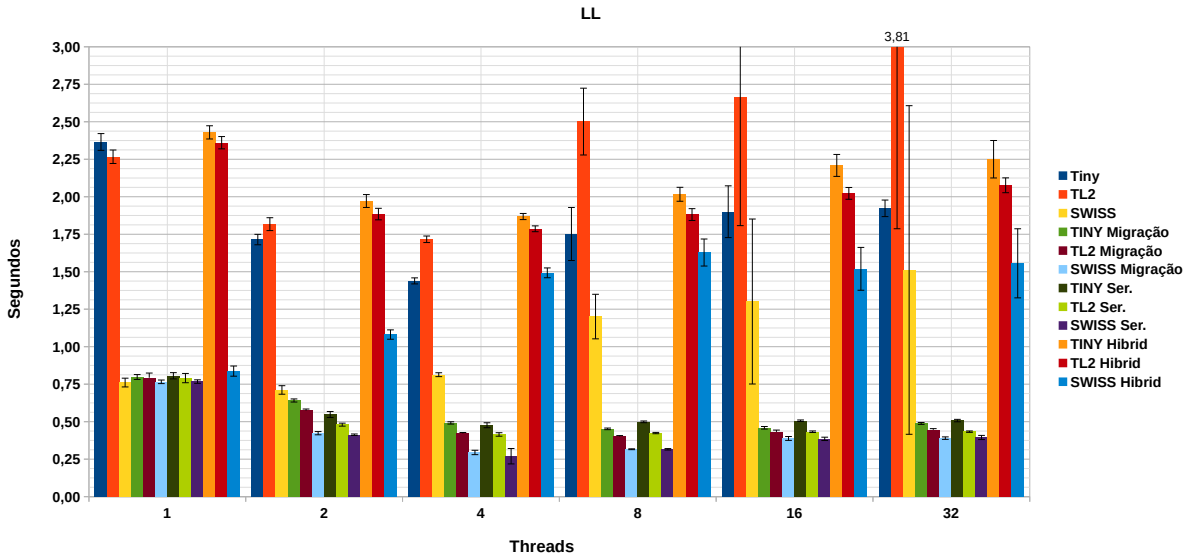

Figura 28 – Tempos de execução da aplicação LL

temos uma mistura de transações curtas e longas, devido a natureza aleatória das inserções e remoções.

Um resultado interessante nesta aplicação foi o da SWISS Ser. e SWISS Migração, onde a aplicação apresentou os menores tempos de execução e também as menores taxas de conflito. As mesmas foram tão baixas que tiveram que ser apresentadas como valores sobre as barras do gráfico, como pode ser observado na Figura 29.

Observa-se também o escalonador com serialização apresentou os melhores tempos de execução e as menores taxas de cancelamento com os diferentes algoritmos de STM.

Conforme o gráfico de escalabilidade, observado na Figura 30, o único escalonador que não escalou foi o SWISS Hibrid. Porém, acima de *quatro threads*, o escalonador manteve sua curva quase constante. Os demais escalonadores apresentaram-se escaláveis até a quantidade de *cores* físicos da máquina, depois permaneceram com suas curvas de escalabilidade quase constante.

Um item interessante a se observar no gráfico da Figura 28, é o baixo tempo de execução com um *thread* quando usando os escalonadores de migração e serialização. Este resultado é estranho pois os algoritmos de escalonamento inserem um certo *overhead* na execução com um *thread*, e os resultados mostram que os algoritmos de STM usando esses escalonadores foram mais rápidos do que sem eles. Para tentar encontrar o motivo desse resultado, usou-se um *profiler* do Haskell que mostrou que, com o uso de algumas funções de concorrência de baixo nível no desenvolvimento dos escalonadores, o GHC realizou otimizações na gerência da *heap*, como também habilitou o *garbage colector* paralelo do GHC, o que não acontece quando não se usa esses escalonadores.

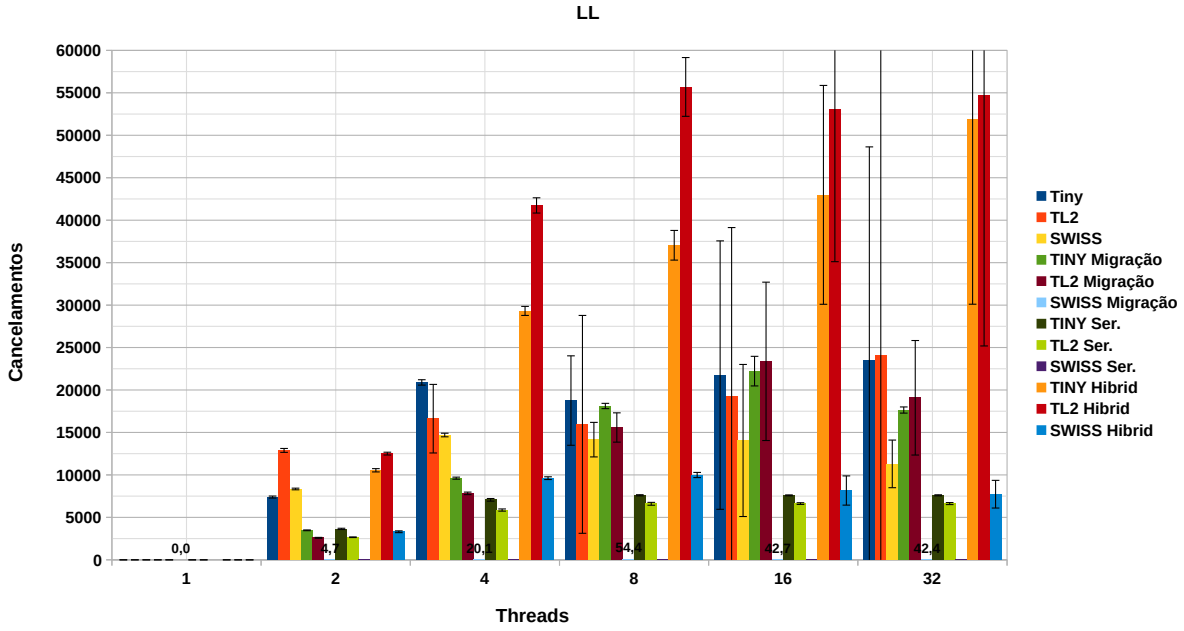

Figura 29 – Quantidade de cancelamentos da aplicação LL

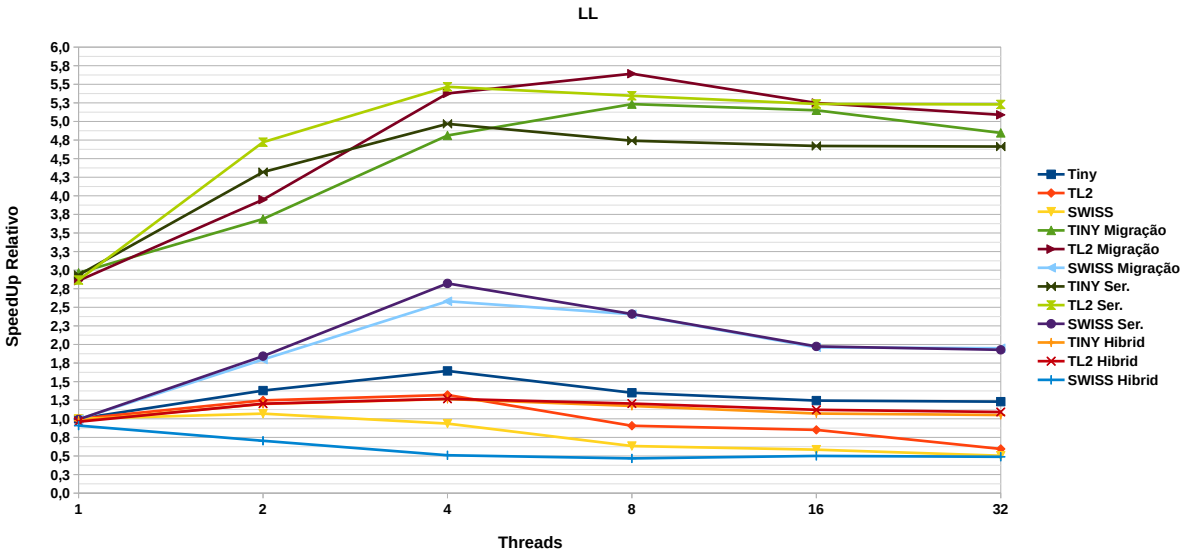

Figura 30 – Curva de escalabilidade relativa da aplicação LL

6.1.3 BT

A aplicação BT insere e remove valores aleatórios em uma arvore binária. Esta aplicação apresenta contenção média para baixa. Os tempos de execução com e sem o uso dos escalonadores pode ser visto no gráfico da Figura 31.

Pode-se observar pelos tempos de execução que os escalonadores conseguiram reduzir o tempo de execução conforme se aumentou o número de *threads* e que, após o número máximo de *cores* físicos, alguns escalonadores conseguiram manter um tempo constante de execução, como os casos de TL2 Migração, TL2 Ser., SWISS

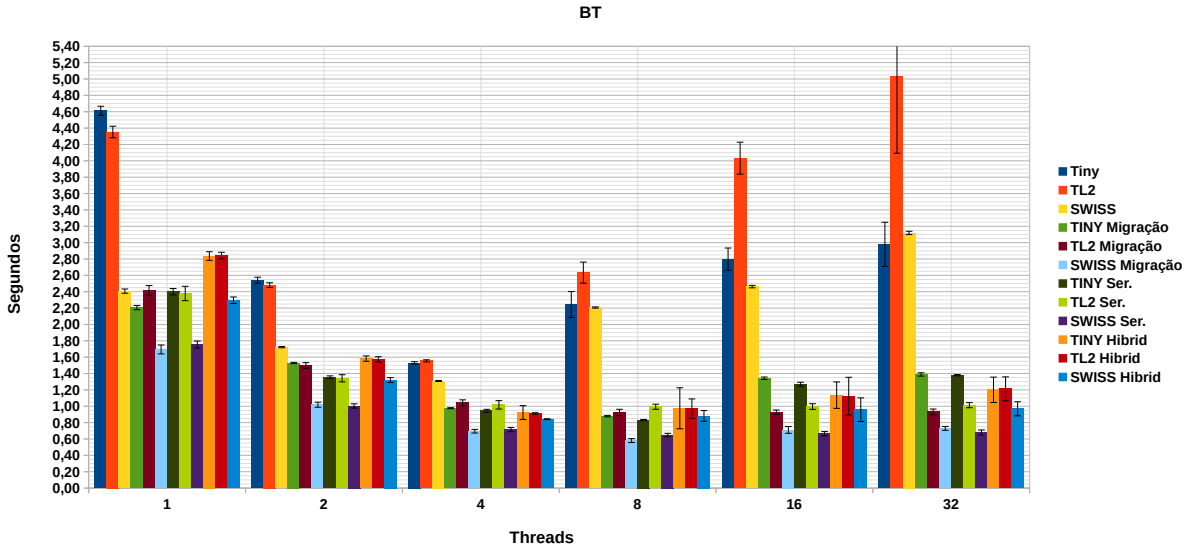

Figura 31 – Tempos de execução da aplicação BT

Migração, SWISS Ser. e todos os híbridos. Sendo que os melhores resultados obtidos foram com o uso do escalonador SWISS Ser.

Na execução sem o uso de escalonamento, ao se aumentar a quantidade de *threads* até a quantidade de *cores* físicos, as aplicações ganham desempenho, logo após isso, a taxa de cancelamentos sobe rapidamente, como pode ser visto na Figura 32, isso acaba levando a aplicação a perder desempenho. Porém quando usa-se escalonadores além da redução do tempo de execução, consegue-se uma redução da taxa de conflitos, com exceção dos escalonadores Tiny Migração e Tiny Ser, que ao chegarem a oito *threads*, aumentam a taxa de cancelamentos e acabam elevando o tempo de execução. No entanto casos como a TINY Hibrid, que aumentam a quantidade de canelamentos mas mantém o tempo de execução quase constante demostram que mesmo a quantidade de cancelamentos aumentando o escalonador ainda sim consegue controlar o tempo de execução.

Já no gráfico de escalabilidade, Figura 33, observa-se que os escalonadores e as bibliotecas conseguiram escalar até quatro *threads*. Após isso, somente as versões com o uso de escalonadores conseguiram escalar até oito *threads* ou mantiveram seus níveis de escalabilidade constantes, como nos casos da TL2 Migr., TL2 Ser. e SWISS Ser..

Em BT, assim como na LL, o tempo de execução com os escalonadores com o modelo de migração e serialização foram mais baixos, levando também a um SpeedUp de dois com um *thread*. Isso novamente se deu pelo fato do GHC realizar otimizações no acesso a *heap* e ativar o GC paralelo. Como o ocorrido na aplicação LL.

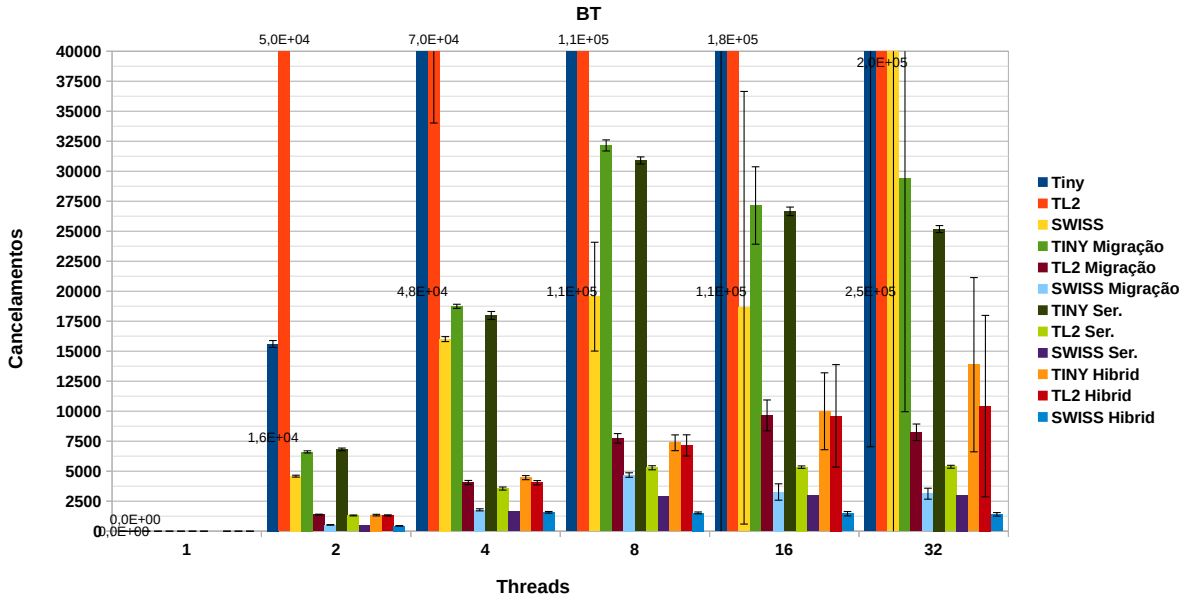

Figura 32 – Quantidade de cancelamentos da aplicação BT

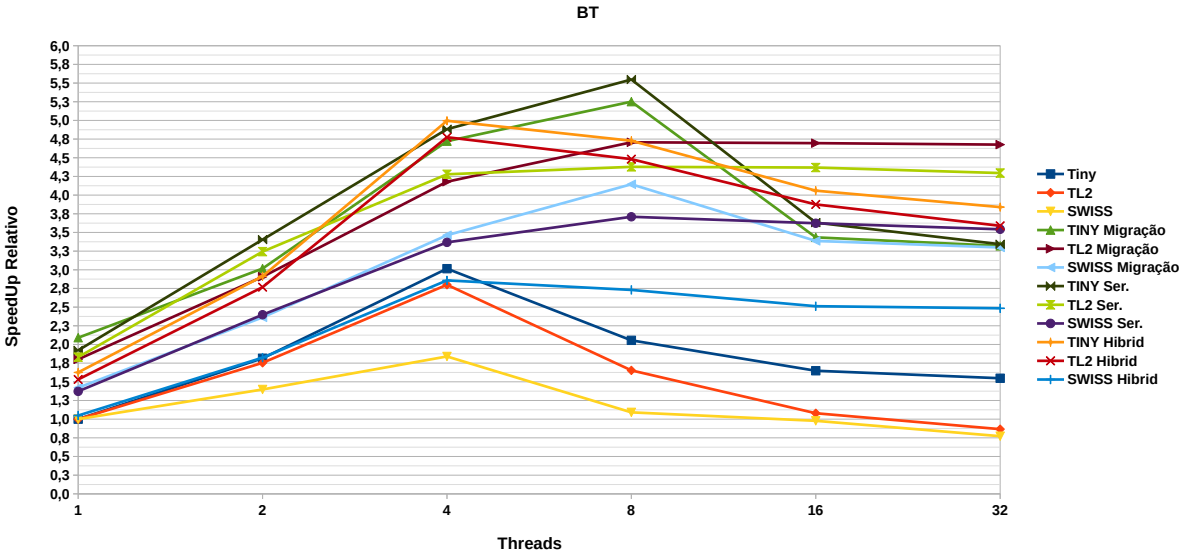

Figura 33 – Curva de escalabilidade relativa da aplicação BT

6.1.4 HT

HT é uma aplicação que realiza inserções, remoções e consultas em uma tabela *hash*. Devido ao acesso aos dados na estrutura serem na maioria disjuntos, essa aplicação oferece baixa contenção de memória. Como pode-se observar no gráfico de tempos de execução da Figura 34, após a quantidade de *threads* ter ultrapassado a quantidade de *cores* físicos todos os escalonadores e bibliotecas perderam desempenho e esse comportamento está relacionado a baixa contenção da aplicação. Quando aumentamos a quantidade de *threads*, temos mais acessos à tabela hash, aumentando a taxa de conflitos e exigindo mais operações dos escalonadores para evitar

que os conflitos ocorram. As versões que apresentaram os menores tempos foram a SWISS e SWISS Hibird.

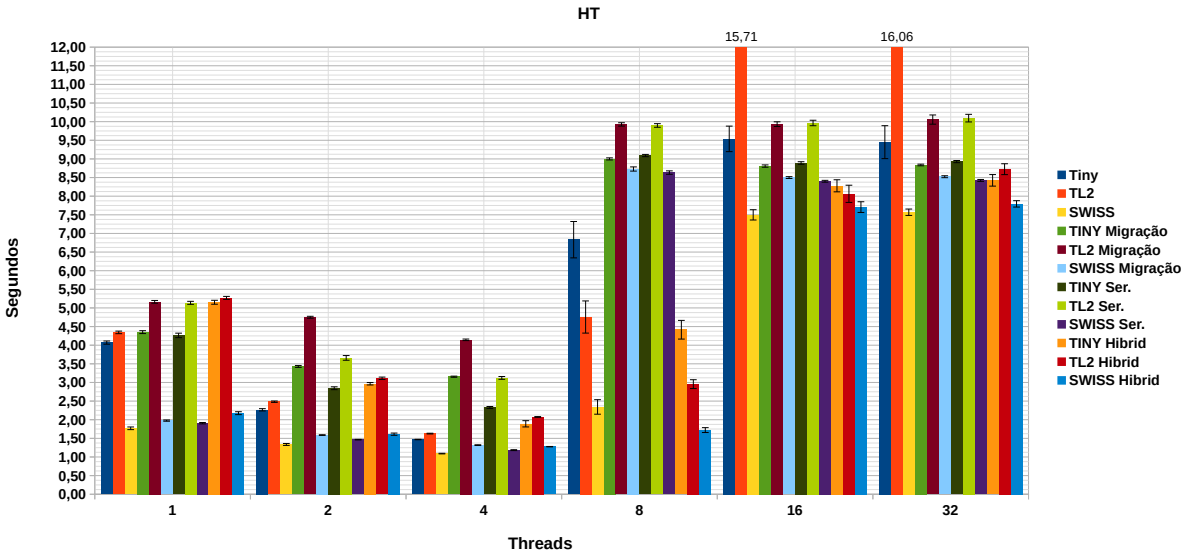

Figura 34 – Tempos de execução da aplicação HT

Pode-se observar também pelos gráfico de cancelamentos, Figura 35, que os escalonadores conseguiram reduzir as taxas de conflito se comparados as suas versões originais (sem escalonamento), mas que ainda sim, devido ao custo de operação dos escalonadores, o tempo de execução ainda sim acabou sendo elevado. As menores taxas de cancelamento foram obtidas com SWISS Migração, SWISS Ser. e SWISS Hibrid.

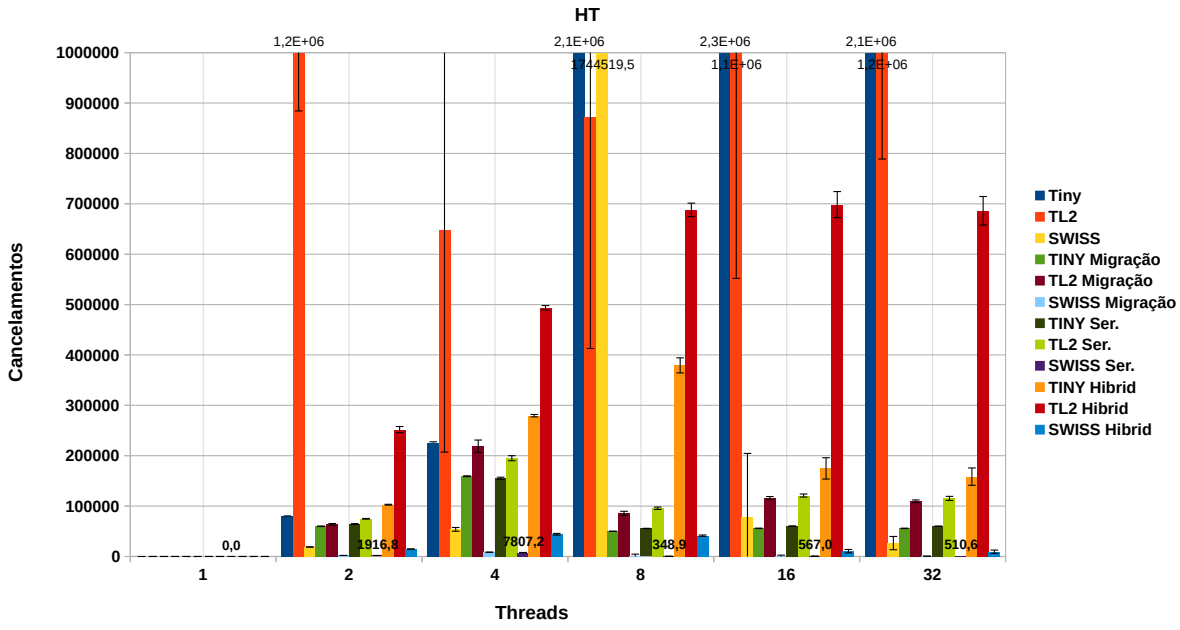

Figura 35 – Quantidade de cancelamentos da aplicação HT

O gráfico de escalabilidade da aplicação pode ser visto na Figura 36 que, como esperado, a aplicação perde escalabilidade acima de quatro *threads*. Porém neste gráfico observa-se que após alcançar o valor de dezesseis *threads* a aplicação mantém seus valores de escalabilidade constante, ou seja, o uso do escalonador evita que a aplicação perca ainda mais desempenho.

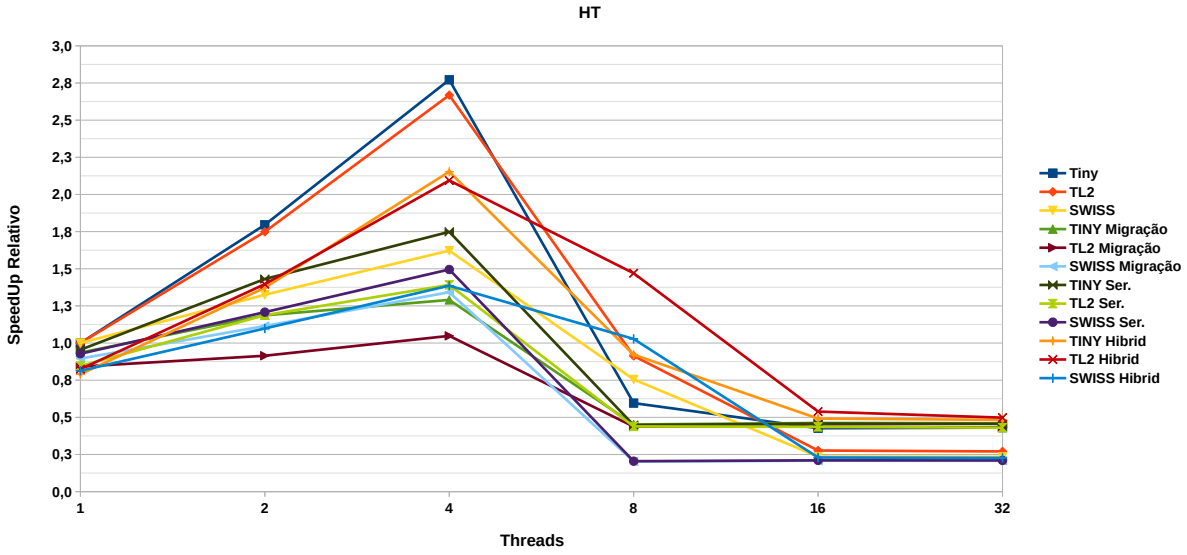

Figura 36 – Curva de escalabilidade relativa da aplicação HT

6.2 Aplicações com retry/orElse

Nesta seção são explorados os resultados obtidos com o uso dos escalonadores sobre aplicações que usam `retry` e `orElse`. Essas aplicações foram separadas pelo motivo de que as mesmas usam no máximo três *threads* durante a execução.

Novamente os gráficos de *SpeedUp* são relativos a execução da versão paralela sem uso o de escalonadores, sendo cada combinação de algoritmo de STM/Escalonador, comparado com o mesmo algoritmo de STM sem escalonamento.

Como estas aplicações possuem um número fixo de *threads*, em todos os gráficos, o que é variado é o número de cores sendo utilizados. Isso é possível por causa da *flag N* do sistema de tempo de execução do GHC.

6.2.1 Sud

Sud é um programa que resolve um jogo de sudoku. Esta aplicação dispara três *threads*, uma para a solução das colunas, outra para a solução das linhas e mais uma para a solução dos blocos. A contenção nesta aplicação é média para alta, porque as transações presentes em um *thread* dependem da solução das demais.

Os resultados dos tempos de execução desta aplicação podem ser visto no grá-

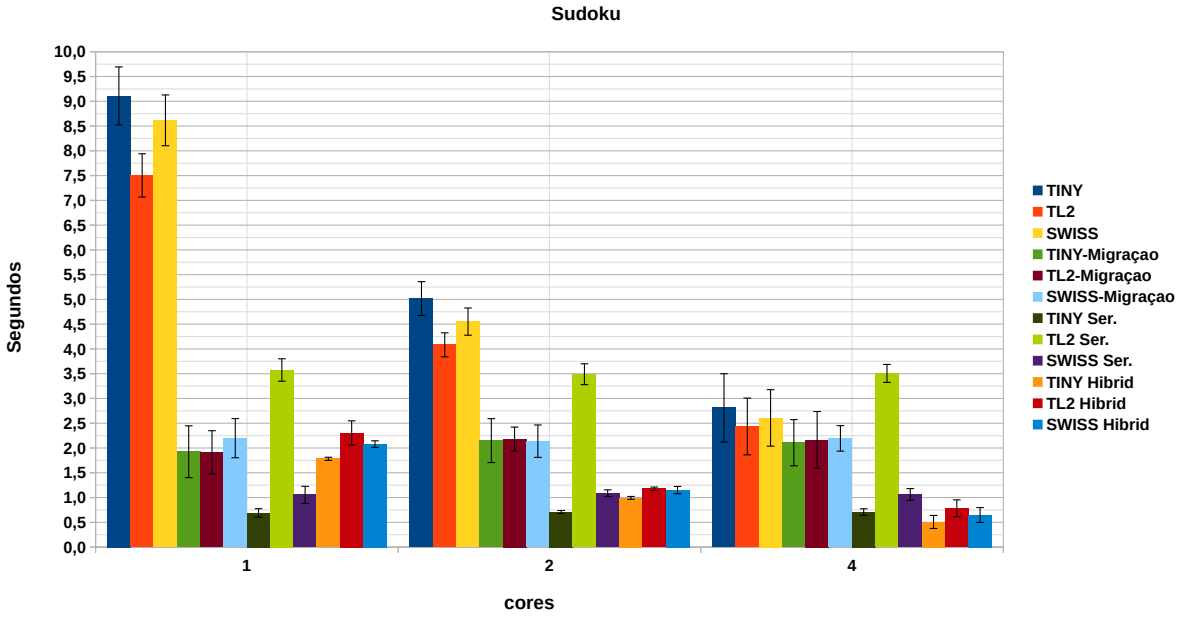

Figura 37 – Tempos de execução da aplicação Sud

fico da Figura 37. Neste gráfico podemos observar que todas as bibliotecas de STM obtiveram redução nos tempos com o aumento do número de cores, com exceção do escalonador TL2.Ser. que, apesar de não obter redução no tempo, o manteve constante para todas as quantidades *cores* usados. Os menores tempos de execução foram obtidos com quatro *cores* usando as versões TINY Ser., SWISS Ser. e o escalonador híbrido com todos os algoritmos de STM.

A quantidade de cancelamentos da aplicação usando os diferentes escalonadores pode ser vista na Figura 38, onde as menores taxas de cancelamentos foram alcançadas com TINY Migração e Tiny Ser, seguidos de SWISS Ser. TINY Hibrid e TL2 Ser., sendo que as piores taxas de canelamento ficaram por parte das bibliotecas TINY e TL2 sem escalonamento. As versões com serialização obtiveram as menores taxas de cancelamento devido ao fato de que, serializar transações, faz com que as mesmas não entrem em conflito tão facilmente, reduzindo o tempo de execução

Já no gráfico de escalabilidade (Figura 39), as aplicações TINY Hibrid, SWISS migração, TL2 Hibrid, TL2 Ser., SWISS e TINY se mostraram escaláveis, porém os escalonadores TINY Migração, SWISS Hibrid, TL2 Migração, TINY Ser. e SWISS Ser. perdem escalabilidade, porém em uma curva pouco acentuada.

6.2.2 CCHR-Sudoku

A aplicação CCHR-Sudoku é uma aplicação parecida com a anterior, porém utiliza o sistema CCHR para resolver o problema e apenas dois *threads*.

Podemos observar pelo gráfico de tempo da Figura 40 que tanto as bibliotecas puras quanto a utilização dos escalonadores resultaram em redução do tempo de

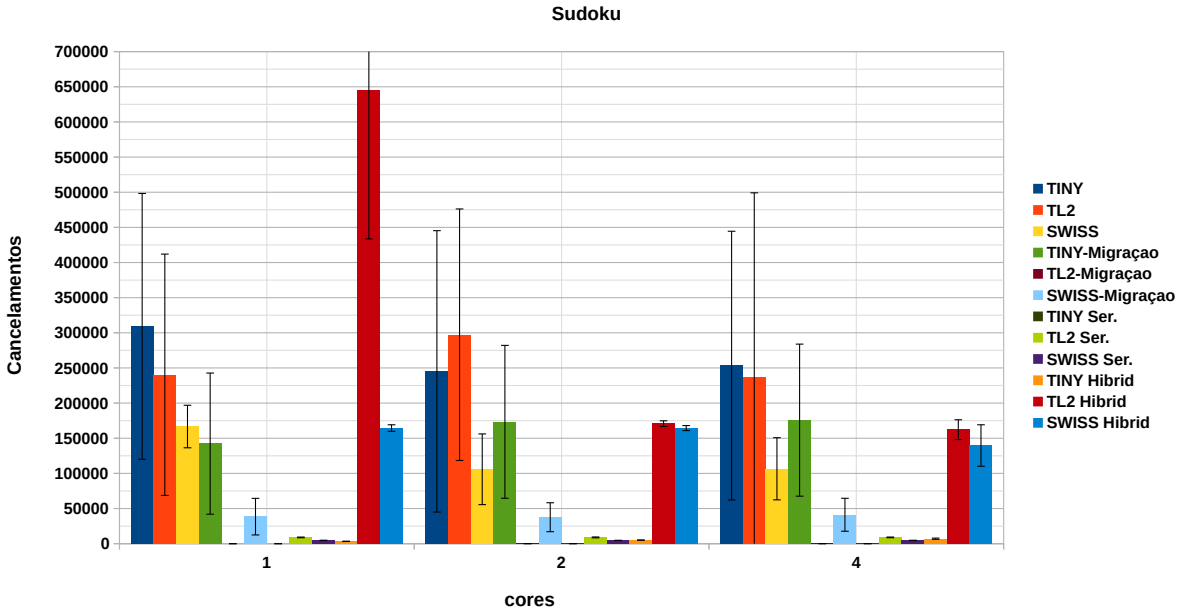

Figura 38 – Quantidade de cancelamentos da aplicação Sud

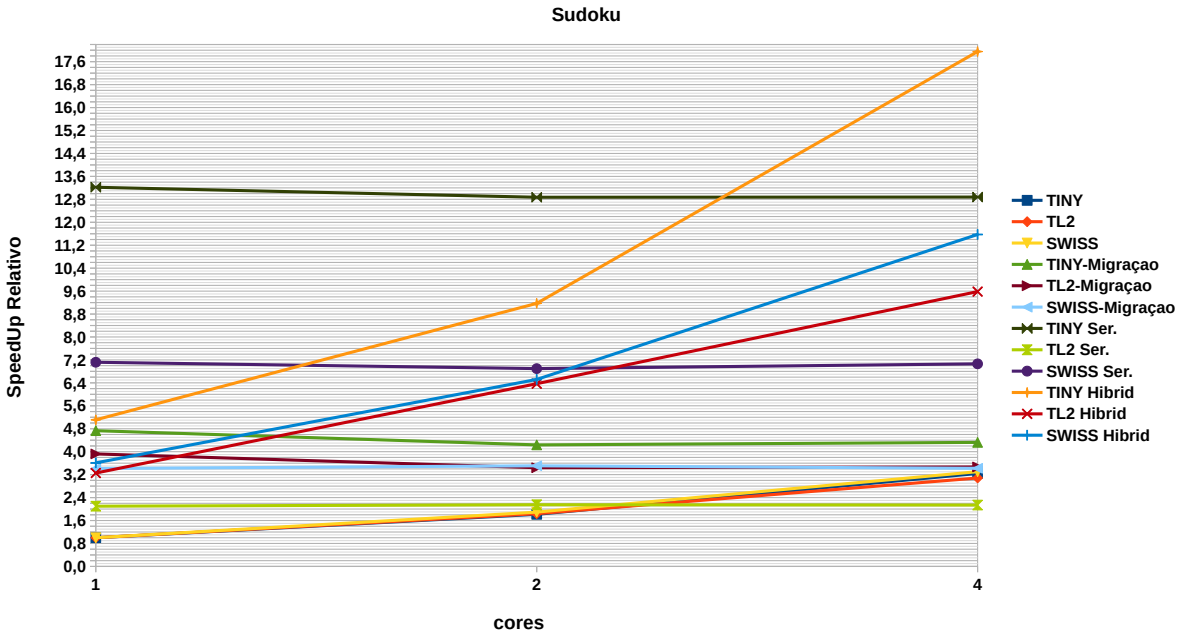

Figura 39 – Curva de escalabilidade relativa da aplicação Sud

execução, ficando as únicas exceções por parte da SWISS Ser. que acabou elevando o tempo de execução e SWIS Hibrid, que com o uso de um *thread* apresentou um tempo bastante elevado em relação as demais versões.

Na questão dos cancelamentos, Figura 41, apesar dos escalonadores em geral apresentarem redução dos tempos de execução, o mesmo não procedeu sobre a quantidade de cancelamentos. Alguns escalonadores como o TINY Migração, TL2-Migração, TINY Ser., TL2 Ser., Tiny Hibird, TL2 Hibrid e SWISS Hibrid, apesar da

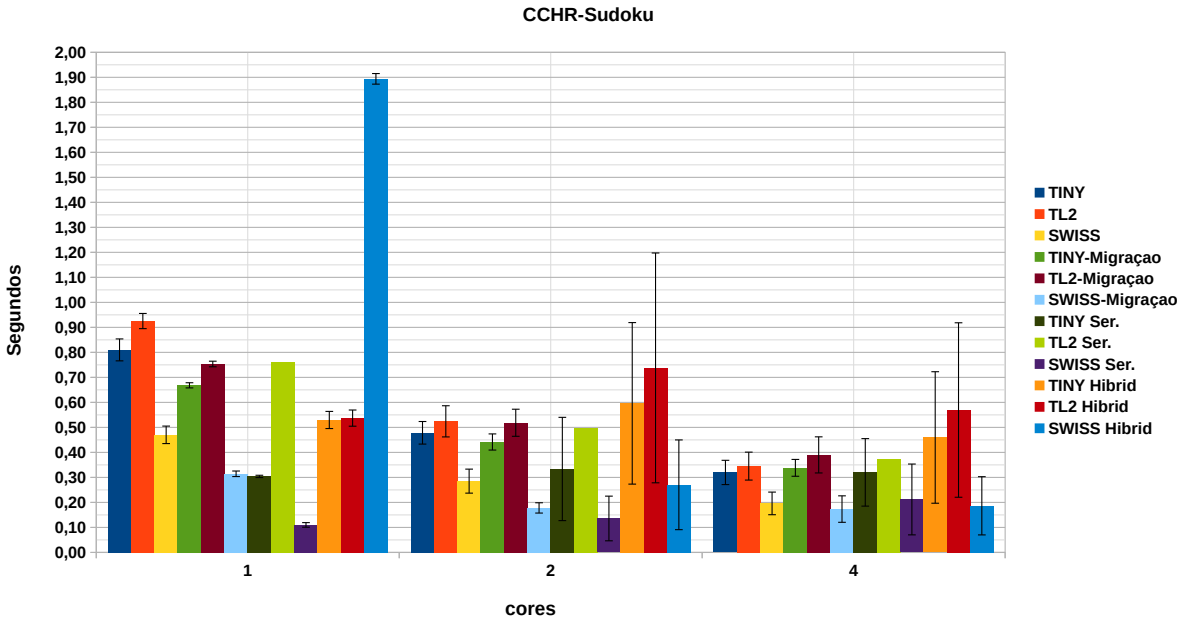

Figura 40 – Tempos de execução da aplicação CCHR-Sudoku

redução do tempo de execução, acabaram aumentando a quantidade de cancelamentos. Isso demostra que se mesmo havendo cancelamentos, os escalonadores conseguirem resolve-los rapidamente, assim obtendo uma redução no tempo de execução.

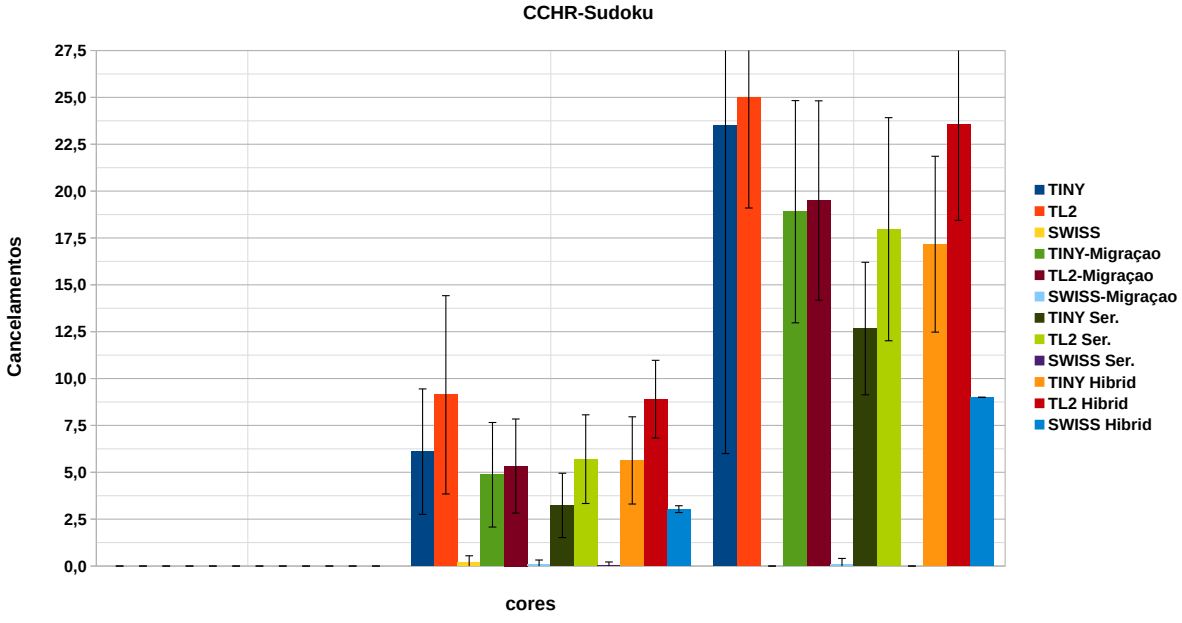

Figura 41 – Quantidade de cancelamentos da aplicação CCHR-Sudoku

No gráfico de escalabilidade, Figura 42, observamos que no geral todaos os algoritmos escalam, com exceção do SWISS.Ser. que perde escalabilidade quando aumenta-se o número de *cores*. Esse comportamento está relacionado a eficiência

desta biblioteca sobre esta aplicação. A mesma se mostrou tão eficiente que ao se aumentar a quantidade de *cores* a mesma não conseguiu explorar o paralelismo.

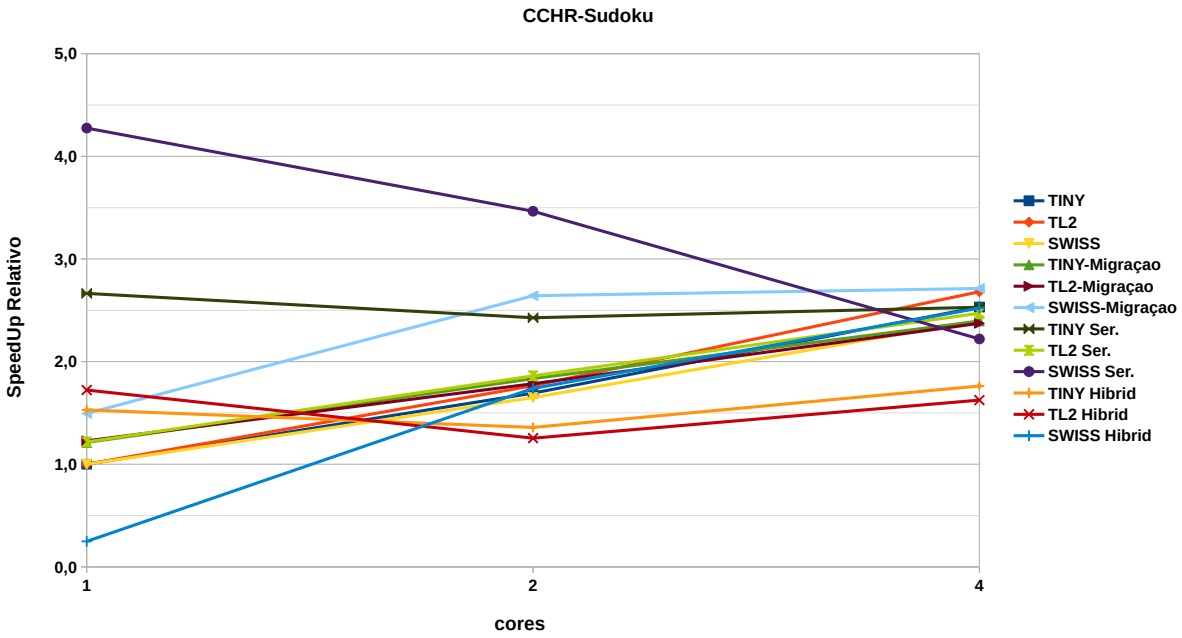

Figura 42 – Curva de escalabilidade relativa da aplicação CCHR-Sudoku

6.2.3 CCHR-UnionFind

A aplicação CCHR-UnionFind é uma aplicação que calcula a união de conjunto de dados, onde os conjuntos são tratados como árvores binárias e duas threads são usadas para cálculo. Seus tempos de execução são apresentados no gráfico da Figura 43 onde observa-se que todos os algoritmos obtiveram redução no tempo de execução, com exceção da SWISS Ser., que manteve seu tempo de execução quase constante e obteve os melhores resultados. Outro resultado interessante é o fato de que usar migração ou serialização sobre a TL2 não demostrou diferença significante, o que não corresponde com as relações da SWISS Migração com SWISS Ser. e TINY Migração com a TINY Ser., em que os tempos de execução foram bem diferentes para as diferentes quantidades de **cores**.

Quando observado o gráfico de cancelamentos (Figura 44), novamente temos outro caso curioso, As implementações usando o modelo Hibrid acabaram aumentando o número de cancelamentos, porém isso não interferiu no tempo de execução que se mantiveram baixos, demostrando que os escalonadores conseguem lidar bem com os conflitos.

No gráfico de escalabilidade, Figura 45, podemos observar que todas as implementações escalam bem, ficando o melhor resultado com SWISS Ser.

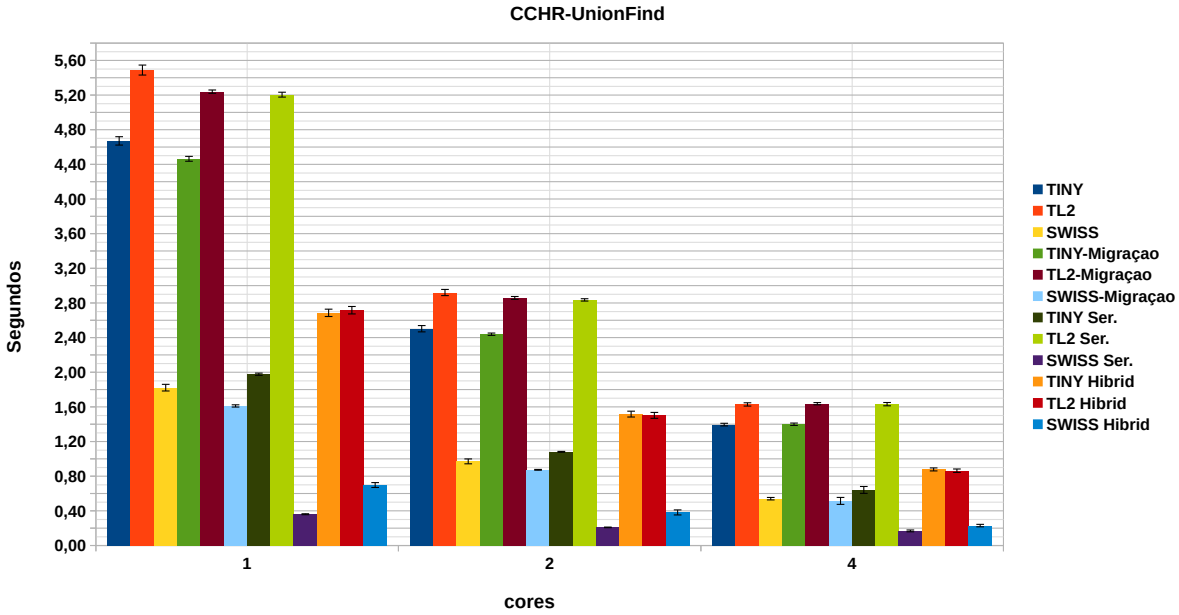

Figura 43 – Tempos de execução da aplicação CCHR-Union

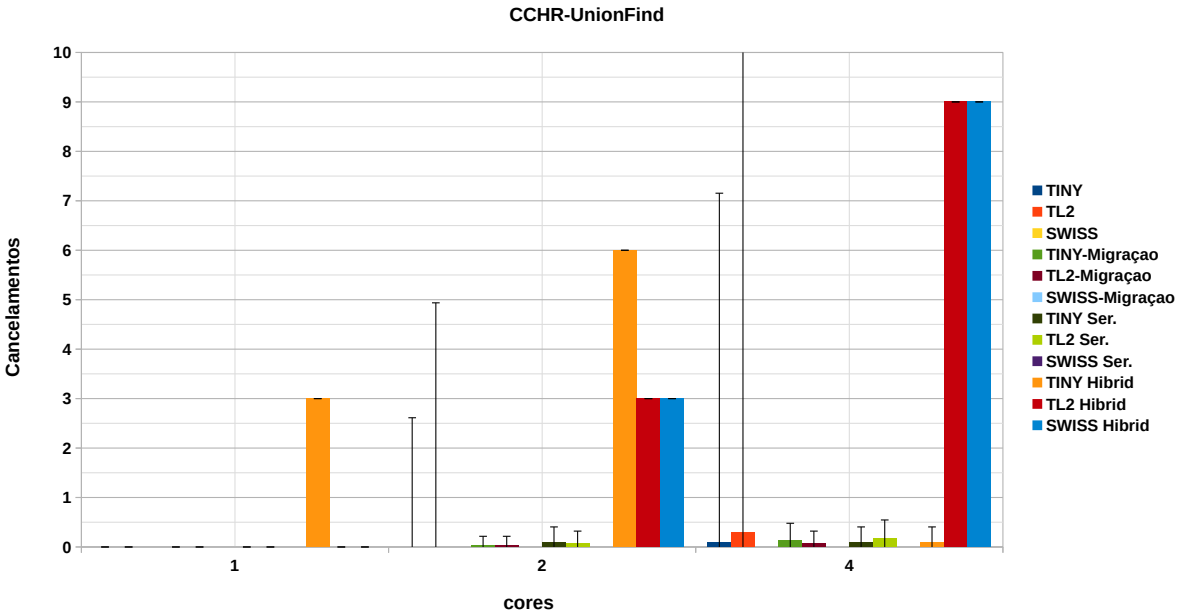

Figura 44 – Quantidade de cancelamentos da aplicação CCHR-Union

6.2.4 CCHR-Prime

A aplicação CCHR-Prime é uma aplicação que realiza a busca pelos 4000 primeiros números primos usando dois *threads*. Nesta aplicação todos os escalonadores também apresentaram redução no tempo de execução, obtendo os melhores resultados com SWISS Ser. e a SWISS Hibrid, como se pode observar pela Figura 46.

Já na quantidade de cancelamentos, alguns algoritmos acabaram apresentando um aumento nos valores, como por exemplo os escalonadores Tiny Hibrid e Tl2 Hibrid,

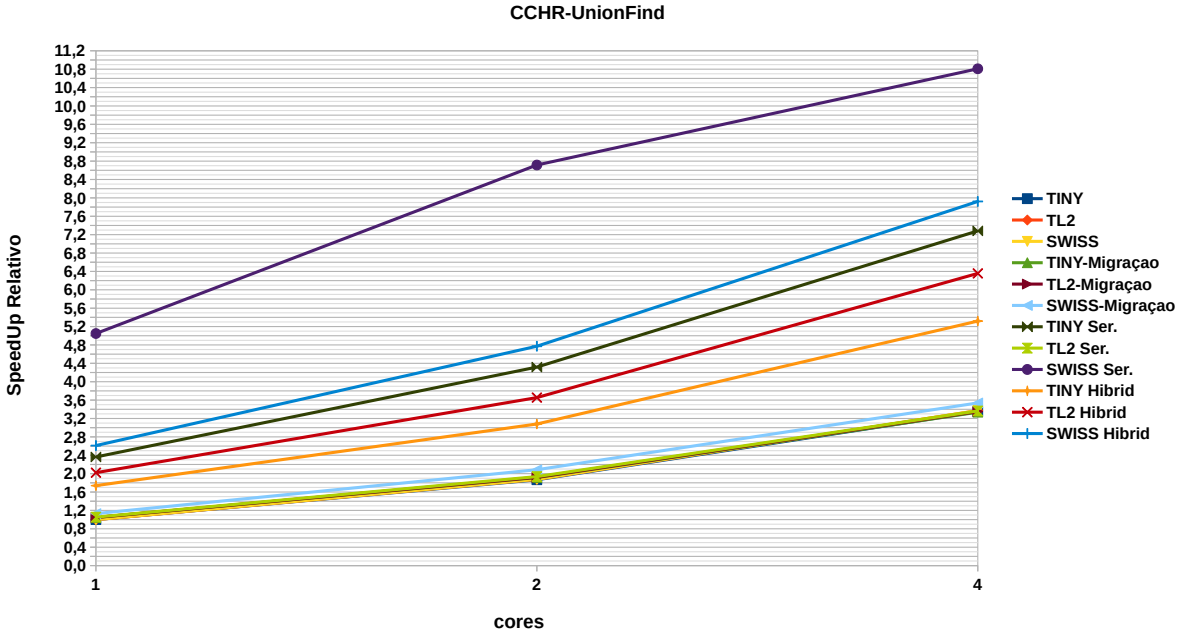

Figura 45 – Curva de escalabilidade relativa da aplicação CCHR-Union

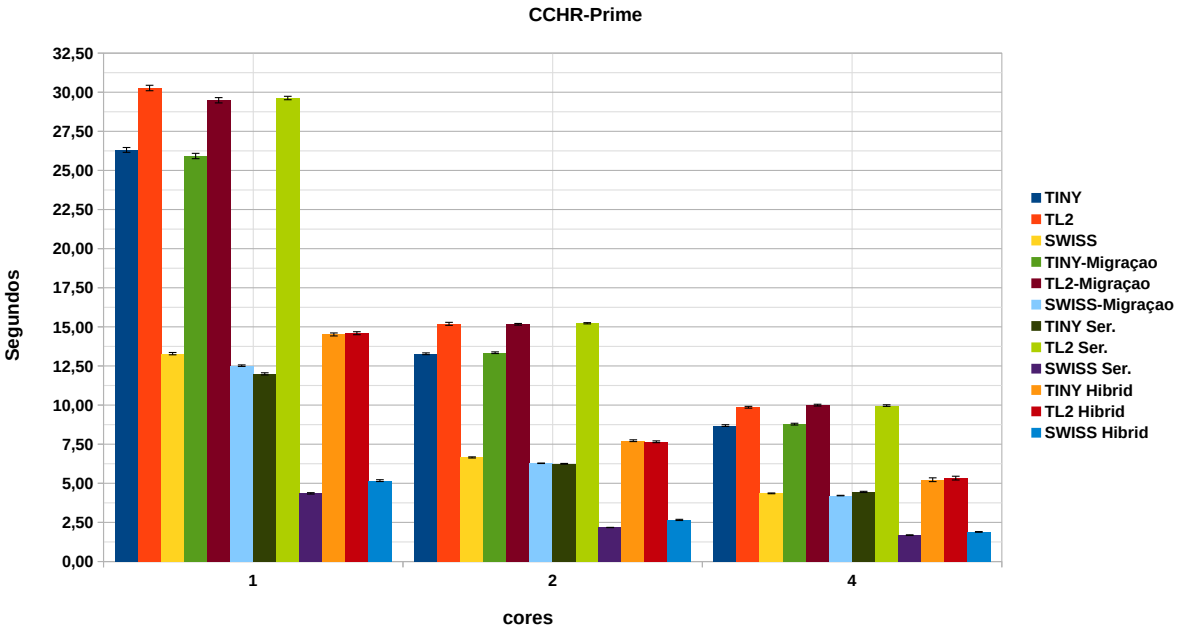

Figura 46 – Tempos de execução da aplicação CCHR-Prime

quando se aumenta a quantidade de *cores* de dois para quatro. Porém os valores de cancelamento são baixos (não ultrapassando de 40 na média), não interferindo no tempo de execução da aplicação.

Quando observada a curva de escalabilidade no gráfico da Figura 48, todos os algoritmos escalaram bem, sendo o melhor resultado obtido novamente pela SWISS Ser. e SWISS Hibrid.

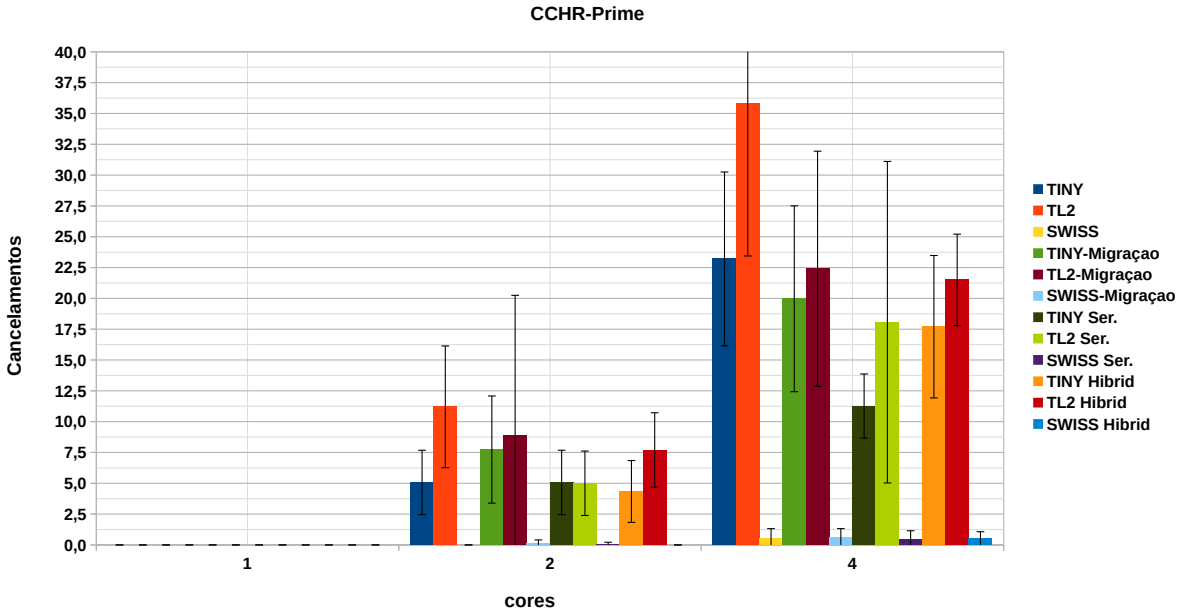

Figura 47 – Quantidade de cancelamentos da aplicação CCHR-Prime

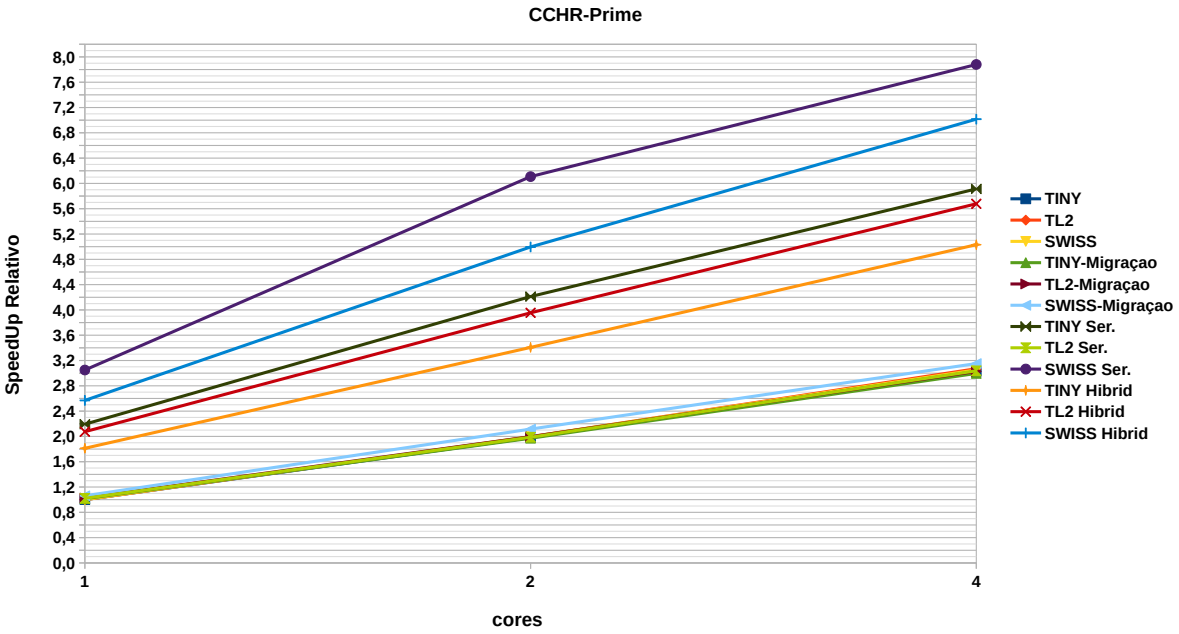

Figura 48 – Curva de escalabilidade relativa da aplicação CCHR-Prime

6.2.5 CCHR-BlockWorld

A aplicação CCHR-Blockword é um programa que simula a interação entre dois agentes movendo 100 blocos entre localizações sem sobreposição. O programa usa dois *threads*.

Os tempos de execução da aplicação são apresentados na Figura 49. Novamente, nesta aplicação, todos os escalonadores apresentaram redução nos tempos de exe-

cução, sendo novamente os melhores resultados obtidos com o uso da SWISS Ser. e SWISS Hibrid.

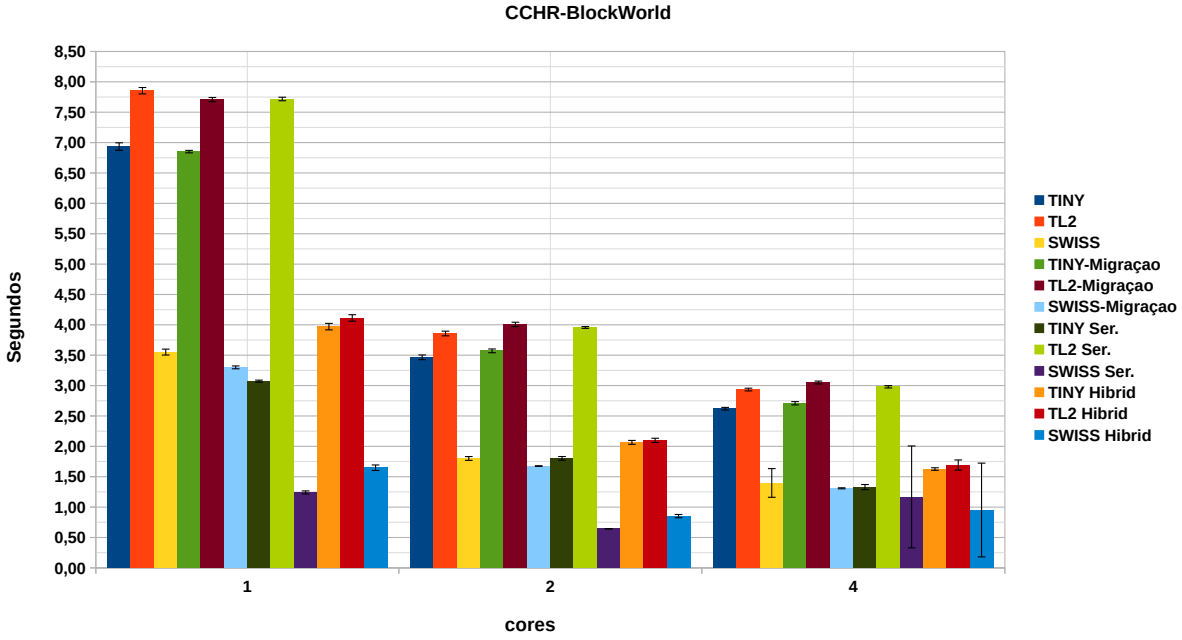

Figura 49 – Tempos de execução da aplicação CCHR-Blockworld

Já nos gráficos de cancelamentos (Figura 50), os únicos escalonadores e bibliotecas que apresentaram aumento no número de cancelamentos foi TINY e TL2, tanto nas Migrações, Serialização e Híbridos, sendo que os resultados dos demais foram zero ou no máximo dois cancelamentos.

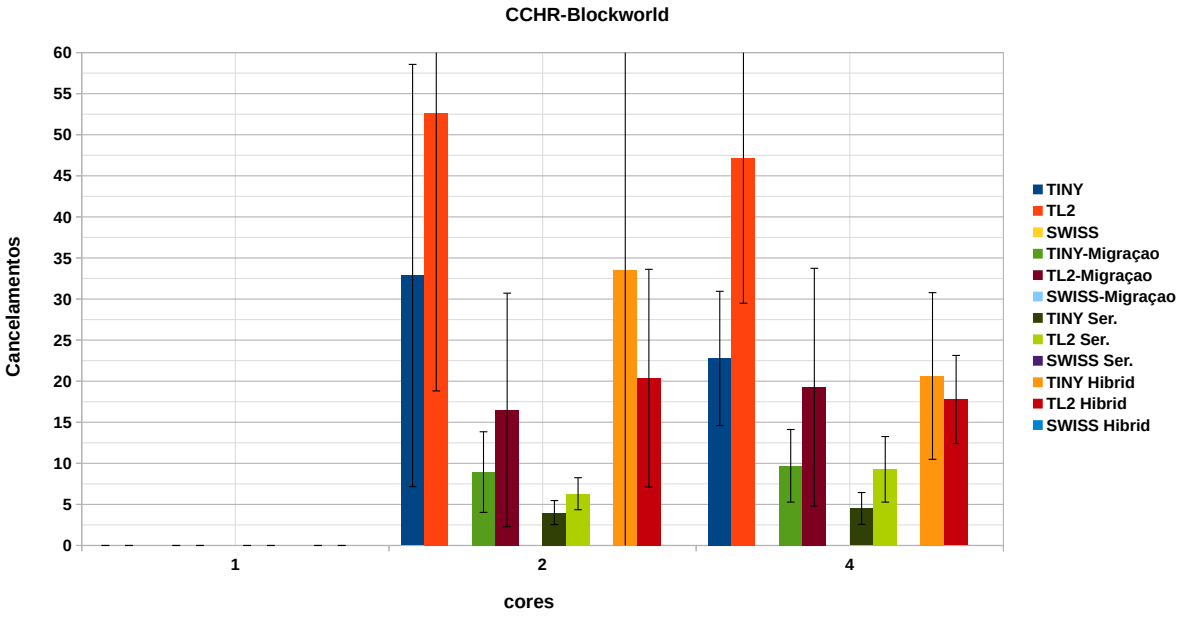

Figura 50 – Quantidade de cancelamentos da aplicação CCHR-Blockworld

Quanto a curva de escalabilidade (Figura 51), os algoritmos mais escaláveis foram TINY Ser. e Tl2 Ser. Tanto a SWISS Ser. quanto a Hibrida escalaram, mas escalabilidade com quatro *cores*.

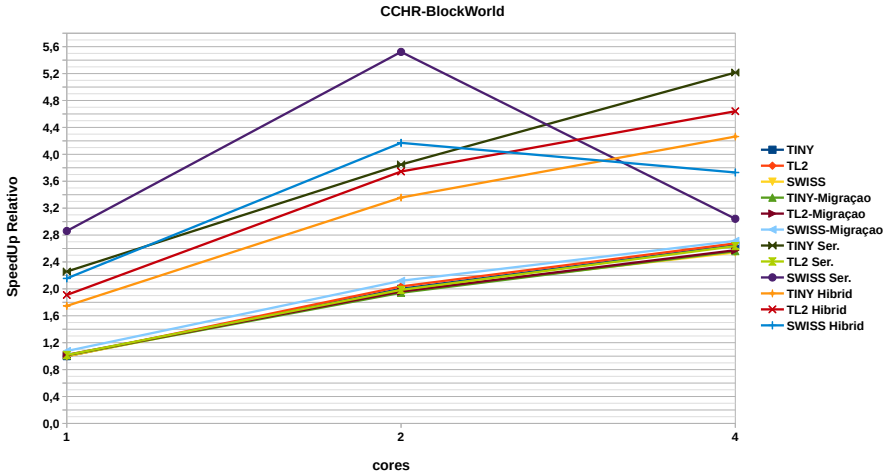

Figura 51 – Curva de escalabilidade relativa da aplicação CCHR-Blockworld

6.3 Observações finais

Neste capitulo apresentamos os resultados da utilização dos escalonadores desenvolvidos para as diferentes bibliotecas de STM Haskell que foram a TINY, TL2 e a SWISS, observando que o comportamento dos escalonadores variou bastante em relação as diferentes aplicações. Em aplicações de alta contenção, como no caso da SI, os escalonadores acabaram em geral evitando um aumento no tempo de execução por controlarem a quantidade de cancelamentos. Porém em aplicações de baixa contenção, como no caso da HT, o uso dos escalonadores não surtiu efeito sobre o tempo de execução apesar destes controlarem a quantidade de cancelamentos. Isso ocorre pois, devido a baixa contenção, as aplicações já exploram bem o paralelismo disponível na arquitetura.

Observamos também que o uso de algumas primitivas de concorrência em Haskell levam, por parte do compilador, a uma otimização da gerência da *heap*, ocasionando uma melhora significativa na execução das aplicações, como foi o caso da LL e da BT com uma única *thread*.

Nos resultados obtidos das aplicações que usam as primitivas `retry` e `orElse`, observamos como essas primitivas podem afetar no tempo de execução. Por exemplo a aplicação CCHR-Prime, em que os tempos de execução com o uso dos escalonadores foi relativamente melhor que as versões das bibliotecas puras, mesmo apresentando valores maiores de cancelamento. Com isso algumas observações foram feitas e serão apresentadas no próximo Capítulo.

7 CONCLUSÃO

O escalonamento transacional tem por objetivo melhorar o desempenho dos sistemas de MTs através da redução de cancelamentos. Esta tentativa de redução de conflitos é feita pelo escalonador transacional através do controle da sequência de execução das transações e ou pelo limite de transações sendo executadas de forma concorrente.

Os trabalhos sobre escalonamento de transações apresentados na revisão de literatura, geralmente investigam os algoritmos de escalonamento no contexto de linguagens imperativas, usando no máximo dois algoritmos de STM. Neste trabalho exploramos o uso de três modelos de escalonamento (migração, serialização e uma abordagem híbrida), sobre três diferentes tipos de algoritmos de STM (TL2, TINY e SWISS). Os algoritmos usados neste trabalho, contemplam as abordagens mais usadas no *design* de sistemas de STM. Também verificamos a viabilidade da implementação de escalonadores transacionais em uma linguagem funcional pura e de tipagem forte como Haskell.

Os resultados mostraram que na maioria das aplicações (HT, CCHR-Sudoku, CCHR-Unionfind, CCHR-Prime e CCHR-Blockworld), todos os escalonadores integrados com a SWISS (versionamento misto), foram os que obtiveram os melhores resultados, tanto em tempo de execução quanto em número de cancelamentos.

Em cenários de alta contenção de memória (aplicação SI), os melhores resultados, em relação ao tempo de execução, foram obtidos com o uso do escalonador híbrido integrado a TINY (versionamento adiantado), mesmo esta configuração não apresentando as menores taxas de cancelamento. Também no mesmo cenário, os escalonadores que utilizam serialização e migração com a TINY, apresentaram tanto redução no tempo de execução quanto na quantidade de cancelamentos. Esses resultados também foram corroborados com a aplicação de alta contenção (Sud), onde os escalonadores de serialização e híbrido integrados a TINY, obtiveram os melhores resultados, evidenciando que em ambientes de alta contenção, uma estratégia híbrida de escalonamento usando o versionamento adiantado na STM é a melhor opção.

Já em cenários de alta/média (LL); média/baixa (BT) e baixa contenção (HT), o uso

dos escalonadores por migração e serialização com a SWISS apresentaram os melhores resultados, reduzindo tanto o tempo de execução quanto a quantidade de cancelamentos. Como resultado observa-se então que para esses cenários, um algorítimo de versionamento misto combinado com os algoritmos mais simples de escalonamento, i.e., migração e serialização, é a melhor opção.

A analise dos resultados demostrou também que nem sempre a taxa de conflitos está relacionada ao tempo de execução. Exemplos como o uso do escalonador por migração com a TINY (TINY Migração) na aplicação BT, demostram que apesar da elevada quantidade de conflitos e seu grande desvio padrão, a execução da aplicação com o uso deste escalonador obteve um resultado de baixo tempo de execução. Outro exemplo é a aplicação HT, que apesar das baixas taxas de conflitos conseguidas com os escalonadores sobre os diferentes algoritmos de STM, apresentou um aumento considerável no tempo de execução.

Assim, conclui-se que os escalonadores não devem levar somente a taxa de cancelamentos como métrica para o controle do nível de concorrência entre as transações. A decisão de escalonamento também deve levar em conta o tipo de algoritmo de STM e o modelo de escalonamento sendo usado.

7.1 Trabalhos Futuros

A pesquisa em escalonamento de transações de memória possui ainda várias áreas a serem exploradas, sendo assim, aqui citamos algumas ideias de continuidade para este trabalho.

Uma analise mais aprofundada do comportamento do sistema de tempo de execução do GHC se faz necessária, devido a observação de que este se comporta de maneira diferente no gerenciamento da *heap* e do *garbage collector*, dependendo das primitivas de concorrência usadas. Esse conhecimento pode ajudar a otimizar aplicações concorrentes em geral, não somente as que usam STM e escalonadores transacionais.

Uma outra proposta seria o desenvolvimento de um escalonador dinâmico, que além de usar a análise de conflitos, também considere as características de contenção das aplicações para decidir qual modelo de escalonamento e qual algoritmo de STM devem ser utilizados.

A eficiência energética dos escalonadores é outro item importante a ser investigado. Instrumentar os escalonadores apresentados no trabalho para mensurar o consumo de energia, usando recursos dos processadores modernos conhecidos como RAPL (LIMA et al., 2019), é uma outra proposta de trabalho.

Por fim, como um trabalho maior, os algoritmos apresentados neste trabalho podem ser implementados no *core* do sistema de tempo de execução do GHC, dessa

forma reduzindo os *overheads* do uso de uma linguagem de mais alto nível, fornecendo um escalonador transacional mais robusto para o GHC.

REFERÊNCIAS

ANSARI, M. et al. Adaptive concurrency control for transactional memory. In: FIRST WORKSHOP ON PROGRAMMABILITY ISSUES FOR MULTI-CORE COMPUTERS, 2008, Goteborg, Sweden. **Anais...** HIPEAC, 2008. p.64–71.

ANSARI, M. et al. Advanced Concurrency Control for Transactional Memory Using Transaction Commit Rate. In: EURO-PAR 2008 – PARALLEL PROCESSING, 2008, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2008. p.719–728.

ANSARI, M. et al. Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory. In: ALGORITHMS AND ARCHITECTURES FOR PARALLEL PROCESSING, 2008, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2008. p.196–207.

ÅSTRÖM, K. J.; HÄGGLUND, T. **PID controllers**: theory, design, and tuning. Research Triangle Park, NY, USA: Isa Research Triangle Park, NC, 1995. v.2.

CHAN, K.; LAM, K. T.; WANG, C.-L. Adaptive thread scheduling techniques for improving scalability of software transactional memory. In: IASTED INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND NETWORKS, PDCN 2011, 10., 2011, New York, NY, USA. **Anais...** ACM, 2011.

CHEN, Z. et al. HaTS: Hardware-Assisted Transaction Scheduler. In: INTERNATIONAL CONFERENCE ON PRINCIPLES OF DISTRIBUTED SYSTEMS (OPODIS 2019), 23., 2020, Dagstuhl, Germany. **Anais...** Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020. p.10:1–10:16. (Leibniz International Proceedings in Informatics (LIPIcs), v.153).

CZARNUL, P. **Parallel programming for modern high performance computing systems**. Boca Raton, NY, USA: CRC Press, 2018.

DEMSKY, B.; DASH, A. Evaluating contention management using discrete event simulation. In: FIFTH ACM SIGPLAN WORKSHOP ON TRANSACTIONAL COMPUTING (APRIL 2010), 2010. **Anais...** ACM, 2010.

Di Sanzo, P. et al. Adaptive Model-Based Scheduling in Software Transactional Memory. **IEEE Transactions on Computers**, New York, NY, USA, v.69, n.5, p.621–632, 2020.

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional Locking II. In: DISTRIBUTED COMPUTING, 2006, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 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. **Anais...** ACM, 2008. p.125–134. (PODC '08).

DRAGOJEVIć, A.; GUERRAOUI, R.; KAPALKA, M. Stretching Transactional Memory. **SIGPLAN Not.**, New York, NY, USA, v.44, n.6, p.155–165, June 2009.

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. **Anais...** ACM, 2009. p.7–16. (PODC '09).

DU BOIS, A. R. An Implementation of Composable Memory Transactions in Haskell. In: SOFTWARE COMPOSITION, 2011, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2011. p.34–50.

DU BOIS, A. R.; PILLA, M. L.; DUARTE, R. M. A High-Level Implementation of STM Haskell with Write/Write Conflict Detection. In: APPLICATIONS FOR MULTI-CORE ARCHITECTURES (WAMCA), 2012 THIRD WORKSHOP ON, 2012. **Anais...** IEEE, 2012. p.24–29.

DUARTE, R. M.; DU BOIS, A. R.; PILLA, M. L.; CAVALHEIRO, G. G. H. Composable Memory Transactions with Eager Version Management. In: ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 30., 2015, New York, NY, USA. **Anais...** ACM, 2015. p.2093–2098. (SAC '15).

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

GUERRAOUI, R.; HERLIHY, M.; POCHON, B. Toward a Theory of Transactional Contention Managers. In: TWENTY-FOURTH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2005, New York, NY, USA. **Anais...** ACM, 2005. p.258–264. (PODC '05).

FRAIGNIAUD, P. (Ed.). **Distributed Computing**: 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005. Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p.303–323.

GUERRAOUI, R.; KAPALKA, M.; VITEK, J. STMBench7: A Benchmark for Software Transactional Memory. In: ND ACM SIGOPS/EUROSYS EUROPEAN CONFERENCE ON COMPUTER SYSTEMS 2007, 2., 2007, New York, NY, USA. **Anais...** ACM, 2007. p.315–324. (EuroSys '07).

HARRIS, T.; LARUS, J.; RAJWAR, R. Transactional memory. **Synthesis Lectures on Computer Architecture**, USA, v.5, n.1, p.1–263, 2010.

HARRIS, T.; MARLOW, S.; JONES, S. P. Haskell on a Shared-memory Multiprocessor. In: ACM SIGPLAN WORKSHOP ON HASKELL, 2005., 2005, New York, NY, USA. **Anais...** ACM, 2005. p.49–61. (Haskell '05).

HERLIHY, M.; LUCHANGCO, V.; MOIR, M. A Flexible Framework for Implementing Software Transactional Memory. In: ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, 21., 2006, New York, NY, USA. **Anais...** ACM, 2006. p.253–262. (OOPSLA '06).

HUCH, F.; KUPKE, F. A High-Level Implementation of Composable Memory Transactions in Concurrent Haskell. In: IMPLEMENTATION AND APPLICATION OF FUNCTIONAL LANGUAGES, 2006, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2006. p.124–141.

JONES, S. **Beautiful concurrency**. CA, USA: O'Reilly, 2007. p.385–406.

LE, M.; YATES, R.; FLUET, M. Revisiting Software Transactional Memory in Haskell. In: INTERNATIONAL SYMPOSIUM ON HASKELL, 9., 2016, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2016. p.105–113. (Haskell 2016).

LIMA, L. G. et al. On Haskell and energy efficiency. **Journal of Systems and Software**, Radarweg 29, 1043 NX Amsterdam, The Netherlands, v.149, p.554–580, 2019.

LIU, M. et al. DudeTx: Durable Transactions Made Decoupled. **ACM Trans. Storage**, New York, NY, USA, v.14, n.1, Apr. 2018.

MALDONADO, W. et al. Scheduling Support for Transactional Memory Contention Management. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 15., 2010, New York, NY, USA. **Anais...** ACM, 2010. p.79–90. (PPoPP '10).

MARATHE, V. J. et al. Lowering the overhead of nonblocking software transactional memory. In: WORKSHOP ON LANGUAGES, COMPILERS, AND HARDWARE SUPPORT FOR TRANSACTIONAL COMPUTING (TRANSACT), 2006, Rochester, NY, USA. **Anais...** ACM, 2006.

MARLOW, S. **Parallel and Concurrent Programming in Haskell**: Techniques for Multicore and Multithreaded Programming. CA, USA: "O'Reilly Media, Inc.", 2013.

MCKENNEY, P. E.; MICHAEL, M. M.; TRIPLETT, J.; WALPOLE, J. Why the Grass May Not Be Greener on the Other Side: A Comparison of Locking vs. Transactional Memory. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.44, n.3, p.93–101, Aug. 2010.

MINH, C. C. et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 34., 2007, New York, NY, USA. **Anais...** ACM, 2007. p.69–80. (ISCA '07).

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...** Association for Computing Machinery, 2016. (PPoPP '16).

NICÁCIO, D.; BALDASSIN, A.; ARAÚJO, G. LUTS: A Lightweight User-Level Transaction Scheduler. In: ALGORITHMS AND ARCHITECTURES FOR PARALLEL PROCESSING, 2011, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2011. p.144–157.

NICÁCIO, D.; BALDASSIN, A.; ARAÚJO, G. Transaction Scheduling Using Dynamic Conflict Avoidance. **International Journal of Parallel Programming**, Berlin, Heidelberg, v.41, n.1, p.89–110, 2013.

PERFUMO, C. et al. Dissecting transactional executions in Haskell. In: TRANSACT 07: SECOND ACM SIGPLAN WORKSHOP ON TRANSACTIONAL COMPUTING, 2007, Portland, Oregon, USA. **Anais...** ACM, 2007.

PERFUMO, C. et al. The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-Core Environment. In: CONFERENCE ON COMPUTING FRONTIERS, 5., 2008, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2008. p.67–78. (CF '08).

PEYTON-JONES, S.; MARLOW, S. et al. **Glasgow Haskell Compiler**. Disponivel em https://www.haskell.org/ghc/. Acesso em: Março de 2020.

POUDEL, P.; SHARMA, G. Adaptive Versioning in Transactional Memories. In: STABILIZATION, SAFETY, AND SECURITY OF DISTRIBUTED SYSTEMS, 2019, Cham. **Anais...** Springer International Publishing, 2019. p.277–295.

RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. Memórias Transacionais: Uma Nova Alternativa para Programação Concorrente. In: MINICURSOS DO VIII WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, WSCAD 2007, 2007, Gramado,RS,Brasil. **Anais...** SBC, 2007.

RITO, H.; CACHOPO, J. FlashbackSTM: Improving STM Performance by Remembering the Past. In: LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING, 2013, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2013. p.266–267.

RITO, H.; CACHOPO, J. ProPS: A Progressively Pessimistic Scheduler for Software Transactional Memory. In: EURO-PAR 2014 PARALLEL PROCESSING, 2014, Cham. **Anais...** Springer International Publishing, 2014. p.150–161.

RITO, H.; CACHOPO, J. Adaptive transaction scheduling for mixed transactional workloads. **Parallel Computing**, College Park, MD, USA, v.41, p.31 – 49, 2015.

RUGHETTI, D. et al. Tuning the level of concurrency in software transactional memory: An overview of recent analytical, machine learning and mixed approaches. In: **Transactional Memory. Foundations, Algorithms, Tools, and Applications**. Chan: Springer, 2015. p.395–417.

SABEL, D. A Haskell-Implementation of STM Haskell with Early Conflict Detection. In: GEMEINSAMER TAGUNGSBAND DER WORKSHOPS DER TAGUNG SOFTWARE ENGINEERING 2014, 25.-26. FEBRUAR 2014 IN KIEL, DEUTSCHLAND., 2014. **Anais...** CEUR, 2014. p.171–190.

SAINZ, D.; ATTIYA, H. Relstm: A proactive transactional memory scheduler. In: ACM SIGPLAN WORKSHOP ON TRANSACTIONAL COMPUTING, 8., 2013, Houston, TX, USA. **Anais...** ACM, 2013. p.1–8.

SCHERER III, W. N.; SCOTT, M. L. Advanced Contention Management for Dynamic Software Transactional Memory. In: TWENTY-FOURTH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2005, New York, NY, USA. **Anais...** ACM, 2005. p.240–248. (PODC '05).

SPEAR, M. F.; DALESSANDRO, L.; MARATHE, V. J.; SCOTT, M. L. A Comprehensive Strategy for Contention Management in Software Transactional Memory. **SIGPLAN Not.**, New York, NY, USA, v.44, n.4, p.141–150, Feb. 2009.

YOO, R. M.; LEE, H.-H. S. Adaptive Transaction Scheduling for Transactional Memory Systems. In: TWENTIETH ANNUAL SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.169–178. (SPAA '08).

Zhou, N. et al. Autonomic Parallelism and Thread Mapping Control on Software Transactional Memory. In: IEEE INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING (ICAC), 2016., 2016, New York, NY, USA. **Anais...** IEEE, 2016. v.1, n.1, p.189–198.