ADEMIR MARQUES JUNIOR

Consumo de Energia em Escalonadores de Transações em Sistemas de Memória Transacional em Software

São José do Rio Preto
28 de Abril de 2016

ADEMIR MARQUES JUNIOR

Consumo de Energia em Escalonadores de Transações em Sistemas de Memória Transacional em Software

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

Orientador: Prof°. Dr. Alexandro José Baldassin

São José do Rio Preto
28 de Abril de 2016

Marques Junior, Ademir.
Consumo de energia em escalonadores de transações em sistemas de memória transacional em software / Ademir Marques Junior. -- São José do Rio Preto, 2016
89 f. : il., tabs.

Orientador: Alexandro José Baldassin
Dissertação (mestrado) – Universidade Estadual Paulista “Júlio de Mesquita Filho”, Instituto de Biociências, Letras e Ciências Exatas

1. Computação. 2. Programação paralela (Computação) 3. Sistema de transação (Sistemas de computação) 4. Processamento eletrônico de dados em tempo real. 5. Computação - Consumo de energia. I. Baldassin, Alexandro José. II. Universidade Estadual Paulista "Júlio de Mesquita Filho". Instituto de Biociências, Letras e Ciências Exatas. III. Título.

CDU – 518.721

Ficha catalográfica elaborada pela Biblioteca do IBILCE
UNESP - Câmpus de São José do Rio Preto

ADEMIR MARQUES JUNIOR

Consumo de Energia em Escalonadores de Transações em Sistemas de Memória Transacional em Software

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

BANCA EXAMINADORA

Prof° Dr Alexandro José Baldassin
UNESP/DEMAC - Rio Claro
Orientador

Prof° Dr Rodolfo Jardim de Azevedo
IC/UNICAMP - Campinas

Prof° Dr Daniel Carlos Guimarães Pedronette
DEMAC/UNESP - Rio Claro

São José do Rio Preto
28 de Abril de 2016

Mistakes are the portals of discovery

James Joyce

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, e irmã, e em especial ao meu pai que não está mais entre nós, a quem eu agradeço a minha formação como pessoa e dedico este trabalho.

Agradeço o suporte dos meus amigos e colegas de Fatec, Tupinikings MG e afins: Bruna Romero, Fernando Salvador e Natália Pedro, Rezende e Suely, Valdeci e Ana Maria, Cristiane Mangueira, Luis Paulo, e Nathália Gerotti.

Gostaria de agradecer aos professores da Fatec Jahu que me apoiaram em especial aos professores e amigos Dra. Valéria Validório, Me Aparecida Zem Lopes, Dra. Magaly Romão, Me Flávio Ventura, Me Marcos Bonifácio e Me Dalva Vitti parceira de projetos.

Agradeço ainda ao meu orientador Dr Alexandro Baldassin por todo esse período de aprendizado, estendendo esse agradecimento aos outros professores do Departamento de Matemática e Computação de Rio Claro com quem tive contato, os professores Dr. Fabrício Breve e Dr. Ivan Rizzo Guilherme coordenador do PPGCC.

RESUMO

O conceito de Memória Transacional foi criado para simplificar a sincronização de dados em memória, necessária para evitar a computação de dados inconsistentes por processadores multinúcleos, que se tornaram padrão devido às limitações encontradas em processadores de um núcleo. Em evolução constante pela busca de desempenho, os escalonadores de transação foram criados como alternativa aos gerenciadores de contenção presentes nos Sistemas de Memória Transacional. O consumo de energia é preocupação crescente, desde os grandes *data centers* até os dispositívos móveis que dependem de tempo de bateria, sendo também explorado no contexto de sistemas com Memória Transacional. Trabalhos anteriores consideraram, em sua maioria, somente o uso de gerenciadores de contenção, sendo o objetivo deste trabalho uma análise sobre o uso de escalonadores de transação. Desta forma, são exploradas nesta dissertação as técnicas de escalonamento dinâmico de tensão e frequência (DVFS) para a criação de uma heurística para a redução do consumo de energia utilizando o escalonador LUTS como base. Com o uso de aplicações do *benchmark* STAMP e biblioteca de memória transacional TinySTM, este trabalho faz uma análise sobre a eficiência energética dos escalonadores de referência ATS e LUTS, enquanto propõe uma nova heurística com o objetivo de reduzir o consumo de energia, denominada LUTS-Dynamic-Serializer, que alterna entre o uso de *spinlock* e de trava *mutex* de forma dinâmica. O uso desta heurística reduziu o EDP em até 17% e 61% em valores de EDP (*Eenergy-Delay Product*), e 4,95% e 15,8% na média geométrica das aplicações estudadas, em comparação aos escalonadores LUTS e ATS respectivamente, quando se utilizou a configuração de 8 *threads*, que é a limitação física de *threads* do processador utilizado no ambiente de experimento.

Palavras chave: Memória Transacional, Escalonadores, Consumo de Energia

ABSTRACT

The Transactional Memory concept was created to simplify the synchronization of data in memory, needed to avoid computation of inconsistent data in multicore processors, which became standard due to limitations in single core processors. In constant search for performance, transactional schedulers were created as a alternative to contention managers present in transactional memory systems. The energy consumption is a crescent worry, ranging from big data centers to mobile devices which are dependent on battery life, and also being studied in Transactional Memory systems. Past works only considered the use of contention managers, and therefore this work seeks to analyse the impact of transactional schedulers. The techniques involving Dynamic Frequency-Voltage Scaling (DVFS) were explored with the motivation to create a heuristic to reduce energy consumption using the LUTS scheduler as a start. Utilizing the STAMP benchmark and the TinySTM transactional memory library, this work does an analysis about the energy efficiency of the reference scheduler ATS and LUTS, while proposing a new heuristic, named LUTS-Dynamic-Serializer, with the aim to reduce energy consumption by making a choice between spin-lock and mutex lock in a dynamic manner. We achieve with our heuristic up to 17% and 61% in EDP (Energy-Delay Product), and 4,95% and 15,8% considering the geometric mean among the applications studied, compared against the schedulers LUTS and ATS respectively, when we used the configuration with 8 threads, which is the physical limit of threads in the processor used in the experiments.

Keywords: *Transactional Memoy, Schedulers, Energy Consumption*

Lista de Figuras

2.1 Escalonador baseado em filas 29

2.2 Expedição de transações no CAR-STM 31

2.3 Fluxograma de funcionamento do escalonador Shrink . . 32

2.4 Mapeamento de *threads* no LUTS 35

2.5 Heurística para transações longas do LUTS - HASHLUTS 37

3.1 Testes preliminares: Consumo de energia e Tempo da aplicação STAMP-Intruder 51

3.2 Fluxograma de funcionamento simplificado do algoritmo 1 (stm_prepare) 57

3.3 Fluxograma de funcionamento simplificado do algoritmo 2 (stm_commit) 59

3.4 Exemplo de funcionamento com 4 *threads* 59

3.5 Testes de sensibilidade com valores máximos de spin fixos para aplicações STAMP com 8 *threads* 62

3.6 Projeção de curva de melhores valores máximos de spinlock 64

4.1 Gráficos de execução STAMP - Genome 71

4.2 Gráficos de execução STAMP - Intruder 72

4.3 Gráficos de execução STAMP - Kmeans 72
4.4 Gráficos de execução STAMP - Labyrinth 73
4.5 Gráficos de execução STAMP - Ssca2 73
4.6 Gráficos de execução STAMP - Vacation 74
4.7 Gráficos de execução STAMP - Yada 74
4.8 Gráficos normalizados com LUTS-Dynamic-Serializer . . 77

Lista de Tabelas

2.1 Interface de funcionamento do escalonador LUTS 34

2.2 Comparativo entre escalonadores de transação 38

2.3 Domínios que integram a interface RAPL da Intel . . . 39

2.4 Funções presentes na API para a leitura de consumo energético 40

2.5 C-States disponíveis em um processador Intel de 4ª geração 43

4.1 Aplicações presentes no *benchmark* STAMP 67

4.2 Características qualitativas das aplicações do *benchmark* STAMP Aplicação Comprimento da transação 68

4.3 Argumentos utilizados na execução das aplicações do *benchmark* STAMP 68

4.4 Configuração de *Hardware* da plataforma de testes . . . 69

4.5 Comparativo EDP para as aplicações STAMP 75

4.6 Características das aplicações STAMP e melhores escalonadores encontrados para execução com 8 *threads* 78

Lista de siglas

ACPI Advanced Configuration and Power Interface

AMD Advanced Micro Devices

API Application Program Interface

APU Accelerated Processing Unit

ATS Adaptive Transaction Scheduler

BCM Baseboard Management Controller

BSCM Basic Serializing Contention Manager

CAR-STM Collision Avoidance and Resolution for Software Transaction Memory

CC Current Contention

CI Contention Intensity

CPU Central Processing Unit

CPUFREQ CPU frequency scaling

DDR3 Double Data Rate 3

DRAM Dynamic Random Access Memory

DVFS Dynamic Voltage-Frequency Scalling

ECR Execution Context Record

EDP Energy-Delay Product

ETL Encounter-Time Locking

FIVR Fully Integrated Voltage Regulator

GCC GNU Compiler Collection

HTM Hardware Transactional Memory

ISA Instruction Set Architecture

LogTM Log-Based Transactional Memory

LUTS Lightweight User-Level Transactional Scheduler

MSR Model Specific Register

MUTEX Mutual Exclusion

PKG Package

PP Power Plane

PPGCC Programa de Pós Graduação em Ciência da Computação

PSCM Permanent Serializing Contention Manager

R-STM Robust Adaptive STM

RAPL Running Average Power Limit

RSTM Rochester Software Transactional Memory

SLT System Level Thread

STAMP Stanford Transactional Applications for MultiProcessing

STM Software Transactional Memory

TCC Transactional Coherence and Consistency

TDP Thermal Design Power

TL2 Transactional Locking II

TM Transactional Memory

TQ Transactional Queue

UNESP Universidade Estadual Paulista

Conteúdo

1 Introdução **17**

1.1 Objetivo e Contribuição 21

1.2 Apresentação deste trabalho 21

2 Fundamentação teórica **23**

2.1 Memória Transacional 23

2.1.1 Versionamento de dados 25

2.1.2 Detecção e resolução de conflitos 26

2.1.3 Gerenciadores de contenção 27

2.2 Escalonadores de transação 27

2.2.1 ATS 28

2.2.2 CAR-STM 29

2.2.3 Shrink 31

2.2.4 Suporte de Escalonamento no *kernel* 33

2.2.5 LUTS 34

2.2.6 Comparativo entre os escalonadores 36

2.3 Medição de energia 37

2.3.1 A interface RAPL 38

2.3.2 Métricas para avaliação do consumo de energia . 41

2.4 Dynamic Voltage and Frequency Scaling 41

2.4.1 P-States 42

2.4.2 C-States 42

2.4.3 Turbo Boost e Turbo Core 43
2.4.4 Turbo Boost 2.0 43
2.4.5 Controle de frequência e *governors* 44
2.5 Avaliação de consumo de energia em Memória Transacional 46

3 A Heurística LUTS-Dynamic-Serializer 50
3.1 Abordagem 50
3.2 Heurística 52
3.2.1 Algoritmo 53
3.2.2 Definição da fórmula de spin máximo 60
3.3 Discussão 64

4 Resultados experimentais 66
4.1 TinySTM 66
4.2 *Benchmark* STAMP 67
4.3 Configuração do ambiente de experimento 69
4.4 Análise dos resultados 70
4.5 Discussão 77

5 Conclusões 80
5.1 Trabalhos futuros 81

Capítulo 1

Introdução

Desde os primórdios da computação o desempenho dos processadores tem crescido de forma consistente guiado ou inspirado pela lei de Moore, a qual prevê que a quantidade de transistores em uma mesma área dobra a cada dois anos devido à miniaturização desses componentes [44]. Porém, atualmente, o acréscimo de desempenho já não é mais possível somente aumentando-se a quantidade de componentes e a frequência de operação dos processadores, pois há uma dificuldade de dissipar a potência gerada, esbarrando-se no chamado *power wall* [35][50].

A saída encontrada pela indústria de processadores foi a adoção de processadores com múltiplos núcleos de execução, chamados de *multicore*. Em vez de somente aumentar a frequência, mais núcleos de processamento são adicionados ao processador. Isso permite a paralelização do código, no qual as tarefas são divididas entre os núcleos do processador para se aumentar o desempenho sem necessariamente aumentar a frequência de operação.

O problema com os processadores de múltiplos núcleos é que os programadores agora precisam enfrentar novos paradigmas. É preciso criar códigos que possam usufruir dos novos processadores, o que nem sempre pode ser uma tarefa fácil.

A programação paralela traz a necessidade de o programador estar atento aos acessos de memória realizados pelas *threads* para que não haja o processamento de dados inconsistentes e falhas. Tradicionalmente usa-se as travas ou *locks mutex* (*mutual exclusion*) para proteger os acessos de memória em um programa com múltiplas *threads*. A sessão que deve ser protegida é chamada de sessão crítica e só deve ser alterada por uma *thread* de cada vez. A *thread* que chegar na trava *mutex* primeiro adquire tal trava e executa a sessão crítica, liberando-a ao final dessa sessão. As outras *threads* devem aguardar a liberação do *mutex* para prosseguir.

O tipo de trava descrito anteriormente é conhecido como trava global. Ela na prática serializa uma parte do código que poderia estar executando em paralelo. Por exemplo, a inserção e remoção em uma lista ligada poderiam ser feitas simultaneamente se os elementos a serem modificados não estivessem próximos. Em vez de se travar todo o acesso a lista com uma trava global é possível criar travas para cada posição de memória, aumentando o desempenho. Esse tipo de trava é conhecida como trava de grão fino.

Embora seja uma solução mais eficiente, o uso de travas finas aumenta a complexidade do código, tornando a tarefa de programação paralela menos trivial aos programadores menos experientes. Uma solução inspirada nas transações encontradas nos sistemas de bancos de dados foi criada para tornar mais fácil a escrita de código paralelo, tirando do programador a responsabilidade da criação de protocolos de travamentos e transferindo-a ao sistema de tempo de execução, que pode ser implementado em *hardware* ou software. Essa solução é conhecida como Memória Transacional (*Transactional Memory* – TM) [25] [28] [26].

Os sistemas de Memória Transacional emprestam algumas pro-

priedades de transações encontradas nos sistemas de banco de dados que são: isolamento, atomicidade e consistência. O uso de sistemas de Memória Transacional é simplificado pois o programador só precisa identificar o trecho do código que é uma sessão crítica marcando o começo e o fim da transação. O sistema de Memória Transacional se encarregará de evitar que os dados sejam modificados simultaneamente por mais de uma *thread*.

Se uma *thread* estiver executando uma transação e verificar que outra transação já está a modificar a mesma porção de memória, somente uma delas prosseguirá enquanto a outra descartará as modificações feitas anteriormente e começará novamente. Caso não ocorra conflito, as transações estão livres para executarem paralelamente [57].

Detectado um conflito entre transações, o sistema de Memória Transacional deve agir e decidir o que fazer. Essa decisão é associada aos gerenciadores de contenção, que aplicam determinadas políticas para a resolução do conflito. Os gerenciadores de contenção são somente reativos, ou seja, quando um conflito ocorre ele entra em ação. Uma alternativa mais recente do que os gerenciadores de contenção veio com a criação dos escalonadores de transação.

As pesquisas sobre implementações em *software* de Memória Transacional sempre seguiram em busca de melhor desempenho, que fizessem com que esta técnica pudesse ser aplicada em aplicações reais, uma vez que seu desempenho em cenários de alta contenção sempre foi um ponto de preocupação, que agora também inclui a preocupação com o consumo de energia [8][36].

Com o intuito de aumentar o desempenho das implementações de memória transacional em software, diferentes técnicas de gerenciadores de contenção foram criadas, dando lugar posteriormente aos escalonadores de transação [26]. As implementações de escalonadores de transação

têm como um dos principais objetivos diminuir a quantidade de *aborts*, devido a conflitos entre duas ou mais transações que estão a executar simultaneamente, evitando o reinício de uma transação e posterior conflito. A ideia é que, ao escolher qual transação será executada, conflitos podem ser evitados e, consequentemente, maior desempenho pode ser conseguido.

Os escalonadores evitam, na medida do possível, que sejam desperdiçados ciclos do processador em transações condenadas, com o objetivo de aumentar o *throughput* geral da aplicação e potencialmente diminuir seu consumo energético.

Os escalonadores de transação, assim como os gerenciadores de contenção, podem gerar diferentes *overheads* adicionais devido à sua implementação que varia em complexidade e que, de acordo com a heurística utilizada, pode ser mais ou menos eficiente em evitar que conflitos entre transações concorrentes ocorram indiscriminadamente. Temos então um *tradeoff* entre a complexidade e eficiência do escalonador e o *overhead* que ele pode adicionar à aplicação.

Somando-se à busca por mais desempenho, a preocupação com a eficiência energética tem aumentado nos últimos anos influenciada por questões ambientais como o aquecimento global, sendo um ponto importante a se considerar na gestão energética e refrigeração dos grandes mainframes e servidores, além de ser igualmente importante em dispositivos móveis, que precisam economizar energia para aumentar a sua autonomia [46][9][53]. Este cenário traz também a necessidade de se reduzir o consumo de energia, além da tradicional busca por cada vez mais desempenho.

1.1 Objetivo e Contribuição

Este trabalho propõe uma nova heurística para escalonadores de transação, chamada de LUTS-Dynamic-Serializer, com o objetivo de se obter uma melhor relação entre consumo de energia e desempenho. É importante ressaltar que não há na literatura nenhum trabalho abordando a questão energética em escalonadores de transação. Desta forma, a grande contribuição deste trabalho é de apresentar, pela primeira vez na literatura, uma abordagem para escalonamento de transações que leva em consideração o consumo de energia.

As publicações geradas como parte deste trabalho até o momento da escrita deste texto são:

- MARQUES JUNIOR, A., BALDASSIN, A.
  Primeiros estudos sobre consumo de energia em escalonadores de transações In: VI Escola Regional de Alto Desempenho de São Paulo - ERAD-SP, 2015, São José do Rio Preto, SP.

- MARQUES JUNIOR, A., BALDASSIN, A.
  Consumo de Energia em Escalonadores de Transações em Sistemas de Memória Transacional In: V Workshop do Programa de Pós-Graduação em Ciência da Computação da Unesp, 2015, Bauru, SP.

1.2 Apresentação deste trabalho

Neste trabalho de dissertação são apresentados os conceitos de Memória Transacional, trabalhos correlatos sobre Escalonadores de Transação e avaliação de energia (Capítulo 2), a metodologia e informações técnicas para a criação da heurística (Capítulo 3), além dos resultados experimentais e sua análise (Capítulo 4). Por último, também

é apresentada a conclusão do trabalho, considerações finais e trabalhos futuros (Capítulo 5).

Capítulo 2

Fundamentação teórica

Esta capítulo traz o referencial teórico relacionado a sistemas de Memória Transacional, escalonadores de transação, além de trabalhos relacionados à análise de consumo de energia em sistemas de Memória Transacional. Neste capítulo também são abordadas as técnicas de escalonamento dinâmico de frequência (DFVS), a tecnologia de gerenciamento de energia presente nos processadores atuais e a métrica utilizada neste trabalho para avaliar a eficiência energética.

2.1 Memória Transacional

Memória Transacional [26] é um mecanismo para sincronização de código paralelo que usa o conceito de *transação*. Uma transação, nesse contexto, pode ser entendida como um conjunto de instruções que: (1) é executado de forma atômica (ou todas as instruções são executadas ou nenhuma é); (2) mantém o estado do sistema consistente; (3) é isolado da execução de outras transações concorrentes. A grande vantagem do modelo transacional é prover ao programador um modelo de programação de mais alto nível, deixando para o sistema de tempo de execução (*runtime system*) os detalhes de implementação das propriedades transacionais (atomicidade, consistência e isolamento).

A primeira proposta de um Sistema de Memória Transacional foi criada para funcionar inteiramente em *hardware*, conhecida como *Hardware Transactional Memory* (HTM). Em geral esse tipo de memória transacional possui bom desempenho [27]. Os primeiros sistemas de memória transacional em *hardware* traziam modificações no protocolo de consistência de memória *cache* e adição de algumas instruções ao conjunto de instruções da arquitetura (*Instruction Set Architecture* – ISA), mantendo o estado especulativo em *buffer* ou *cache* para uso em caso de *commit* ou *abort*, sendo essa a configuração básica mais comum [25].

Sistemas de Memória Transacional em *Software* (*Software Transactional Memory – STM*) foram idealizados primeiramente pelo trabalho de Shavit e Touitou em 1995 [54], e desde então têm sido um importante tema de pesquisa. A implementação de Shavit e Touitou necessitava que se informasse para a transação quais posições de memória seriam acessadas, diferindo um pouco das implementações atuais que fazem o levantamento dos conjuntos de leitura e escrita de forma dinâmica. Embora tenha desempenho reduzido, os sistemas de STM possuem algumas vantagens em comparação aos sistemas de HTM, como maior flexibilidade e integração com as linguagens existentes, facilidade de implementação e modificação do código, além de possuir menos limitações se comparado a HTM que tem estrutura fixa [26].

Devido às limitações de HTM, uma transação com muitos acessos à memória pode ser impossível de ser executada, por falta de espaço para armazenar os dados especulativos. Uma alternativa é utilizar também o suporte em *software*, criando assim sistemas híbridos. Esse trabalho dará atenção principalmente à Memória Transacional em *Software*.

Em Memória Transacional, algumas propriedades devem ser garantidas por sua estrutura para garantir que em uma situação de conflito

entre transações concorrentes as mesmas não alterem os dados de forma inconsistente. Na implementação de sistemas de TM algumas estruturas básicas devem garantir essas propriedades, como o *versionamento de dados* e a *detecção e resolução de conflitos* descritos a seguir.

2.1.1 Versionamento de dados

Durante a execução de uma transação a alteração dos valores de memória é feita de maneira especulativa, necessitando que os valores originais sejam armazenados em algum lugar, uma vez que podem precisar ser restaurados caso modificados. Desta forma o sistema de TM deve armazenar diferentes versões para cada variável acessada, processo chamado de versionamento de dados. O versionamento de dados pode ser feito de duas formas dependendo de sua implementação: de maneira ansiosa ou *eager*, ou de maneira diferida ou *lazy*. A ansiosa armazena a versão especulativa do dado na memória principal enquanto a versão original é salva internamente. Já na diferida o oposto ocorre: a versão especulativa é armazenada internamente, enquanto o dado na memória principal não é alterado.

O versionamento de dados *eager* se dá com a implementação de um *undo log*. A transação irá armazenar um espaço dedicado na memória onde os valores antigos serão utilizados para restauração em caso de falha. A transação irá alterar imediatamente as variáveis de memória durante a sua execução e caso haja sucesso na execução da transação, o *undo log* é descartado. Caso a transação falhe, os valores do *undo log* são transferidos para a memória.

O versionamento de dados *lazy* funciona de maneira inversa. A execução da transação mantém um *buffer* em memória onde os valores de variáveis que serão alterados serão armazenados. Caso a transação tenha sucesso, os valores do *buffer* são transferidos para a memória e o

buffer é descartado. Caso a transação falhe, o *buffer* é descartado e as posições de memória se mantêm.

2.1.2 Detecção e resolução de conflitos

As implementações de TM em software ainda diferem entre si quanto a granularidade de detecção de conflitos, podendo esta ser por objeto ou palavra [57]. Um conflito entre transações ocorre quando duas ou mais transações acessam o mesmo dado de memória e pelo menos um desses acessos é de escrita.

Cada transação deve ser associada a um conjunto de endereços de memória que deverão ser modificados ou somente lidos. O conjunto de leitura é o conjunto de variáveis que serão lidas pela transação, enquanto o conjunto de escrita é o conjunto de variáveis que serão modificadas pela transação. Quando uma transação está para modificar as posições de memória, por exemplo, em um *commit*, ela checa se as posições de memória estão consistentes com seu conjunto de leitura, identificando se o dado não foi alterado por outra transação (verificando a versão corrente do dado), ou se o dado pertence ao conjunto de escrita de outra transação [25].

De maneira análoga ao versionamento de dados, a detecção de conflitos pode ser também *eager* ou *lazy*. A detecção de conflitos *eager*, ou pessimista, necessita que os conjuntos de leitura e escrita das outras transações estejam visíveis para verificar previamente a possibilidade de haver conflito entre as operações das transações. Já a detecção de conflitos *lazy*, ou otimista, faz essa verificação somente quando a transação está para fazer o *commit*.

2.1.3 Gerenciadores de contenção

Uma vez detectado o conflito, o Sistema de Memória Transacional necessita decidir o que deve ser feito. Por exemplo, deve decidir qual transação pode continuar e qual deve ser reiniciada. Outra alternativa seria fazer com que a transação conflitante espere até que o conflito seja sanado e então retomar a transação parada. A resolução de conflitos é originalmente tarefa do gerenciador de contenção (*contention manager*) que, de acordo com a sua política, deve tratar a contenção. Muitas políticas para gerenciadores de contenção foram desenvolvidas, podendo variar em sua forma de implementação [26][28].

Os gerenciadores de contenção reagem a conflitos entre transações decidindo qual transação deve ser interrompida, ou se uma delas deve esperar por um tempo determinado. Por sua natureza reativa, os gerenciadores de contenção somente agem depois que um conflito ocorreu [28][26]. Em contrapartida, os escalonadores de transação têm o objetivo de evitar que haja desperdício de ciclos em transações com tendência a entrar em conflito, forçando a serialização ou escolhendo transações com menores probabilidades de conflitarem entre si [26]. Desta forma, os escalonadores agem proativamente ao permitir ou não que uma transação seja iniciada.

2.2 Escalonadores de transação

Esta subseção traz uma revisão sobre escalonadores de transação em diferentes implementações, como apresentado a seguir.

2.2.1 ATS

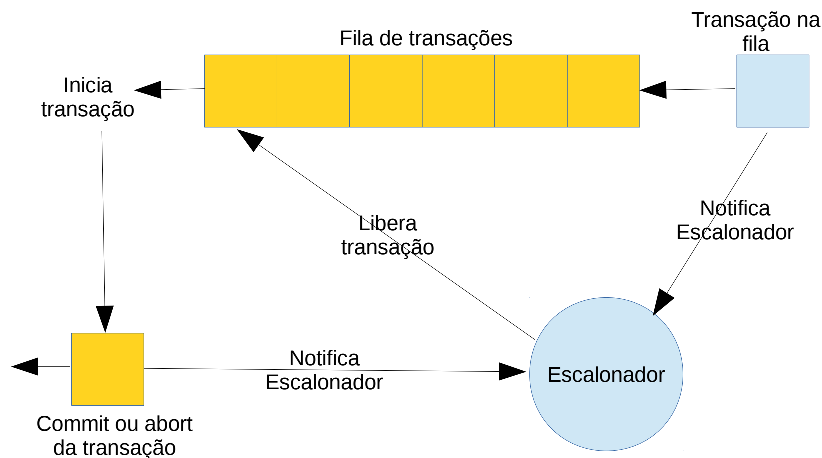
O *Adaptive Transaction Scheduler*(ATS), criado por Yoo e Lee [58], implementa o primeiro escalonador conhecido para Sistemas de Memória Transacional em *software* e *hardware*. O ATS é um escalonador de baixo custo operacional que introduz o conceito de Intensidade de Contenção (*Contention Intensity* – CI), que é mantido para cada *thread*, e um escalonador global de transações. A intensidade de contenção é dada pela equação 2.1, onde CI é a intensidade de contenção para uma determinada transação e CC é a contenção atual (*Current Contention*). Se uma transação tem sucesso, CC é definido como tendo valor zero, e se a transação aborta então CC é igual a 1; α é uma variável de peso que determina uma maior influência dos eventos anteriores (CI_{n-1}) ou da contenção atual.

$$CI_n = \alpha \times CI_{n-1} + (1 - \alpha) \times CC \tag{2.1}$$

Se transações concorrentes encontram alta contenção, assim causando um maior número de cancelamentos e reinícios, pode ser conveniente serializar essas transações para que o desempenho não seja degradado demasiadamente. Para evitar que as transações sejam serializadas indiscriminadamente diminuindo o paralelismo, o ATS somente recorre ao seu escalonador quando a intensidade de contenção está acima de um limiar definido. Se a intensidade de contenção se mantiver baixa, o ATS tem somente o custo de verificar qual é a contenção atual deixando as *threads* livres para executarem as transações concorrentemente. Caso uma *thread* inicie uma transação em alta contenção, a *thread* recorre ao escalonador esperando o que deve ser feito.

Conforme pode ser visto na Figura 2.1, o escalonador implementa uma fila de transações semelhante à fila de processos presente nos

Figura 2.1: Escalonador baseado em filas



Fonte: [58]

sistemas Unix, que escalona uma transação por vez. Se uma transação estiver na cabeça da fila e nenhuma outra estiver executando, ela é colocada para executar. Caso contrário, ela espera na fila até chegar a sua vez de ser executada. O escalonador é notificado toda vez que uma nova transação entra na fila e toda vez que uma transação efetiva ou aborta, para escalonar a execução de outra transação na cabeça da fila.

2.2.2 CAR-STM

O escalonador CAR-STM (*Collision Avoidance and Resolution for Software Transaction Memory*) desenvolvido por Dolev, Hendler e Suissa [15] introduz um gerenciador de contenção serializador e um redutor de colisão proativo (*proactive collision reduction*).

CAR-STM mantém para cada núcleo do sistema uma fila de transações, evitando o não determinismo na execução de transações quando se executa mais de uma transação na mesma *thread*, o que pode causar um falso paralelismo. O gerenciador de contenção serializador pode atuar no modo básico (*Basic Serializing Contention Manager*

– BSCM) e no modo permanente (*Permanent Serializing Contention Manager* – PSCM).

No modo básico, quando há um conflito entre duas transações o gerenciador de contenção serializador interrompe a transação mais nova e a coloca na fila da transação mais antiga, evitando assim que elas entrem em conflito novamente. Porém, caso uma terceira transação force que uma transação troque novamente de fila, o conflito pode ocorrer novamente. Uma solução é aplicar o modo permanente que marca transações como subordinadas ao trocar transações de fila.

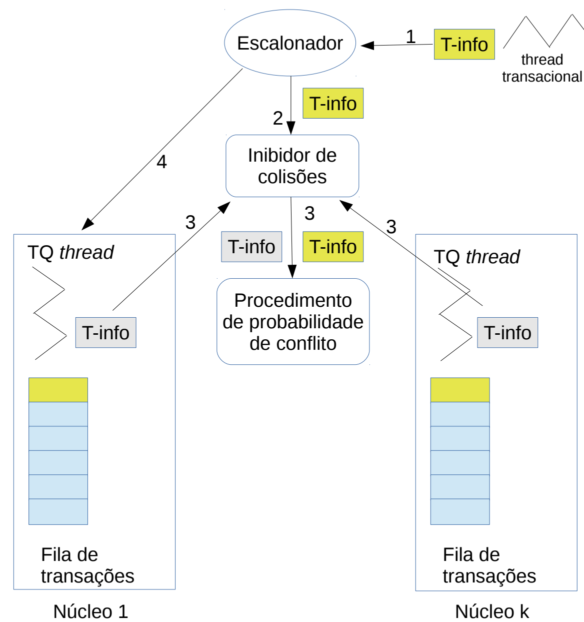
Uma transação ao entrar em conflito com outra, se ela for a mais nova, muda para a fila onde havia a outra transação, sendo marcada como subordinada a esta. Caso uma terceira transação entre em conflito, ela pode forçar que a primeira transação seja mudada de fila, nesse caso transferindo também todas as transações subordinadas.

O redutor de colisão proativo funciona como exemplificado na Figura 2.2. Uma aplicação invoca um método que inicializa as estruturas de dados do CAR-STM, que inicia uma fila de transações (*TQ thread*) para cada núcleo. Cada *TQ thread* aguarda em uma variável de condição por novas transações.

Opcionalmente, a aplicação pode fornecer um ponteiro ao método de probabilidade de conflito, que será utilizado pelo escalonador de transações. Este método é utilizado para decidir se uma determinada transação pode ser executada.

O escalonador de transações é chamado por uma nova transação (1), passando como parâmetros informações como a probabilidade de conflito fornecida pela aplicação para aquela transação (estrutura *T-info*). O escalonador então chama o procedimento Inibidor de Colisões (*Collision Avoider*) (2) que utiliza o procedimento de probabilidade de conflito, comparando duas transações a partir de suas estruturas *T-info*

Figura 2.2: Expedição de transações no CAR-STM



Fonte: [15]

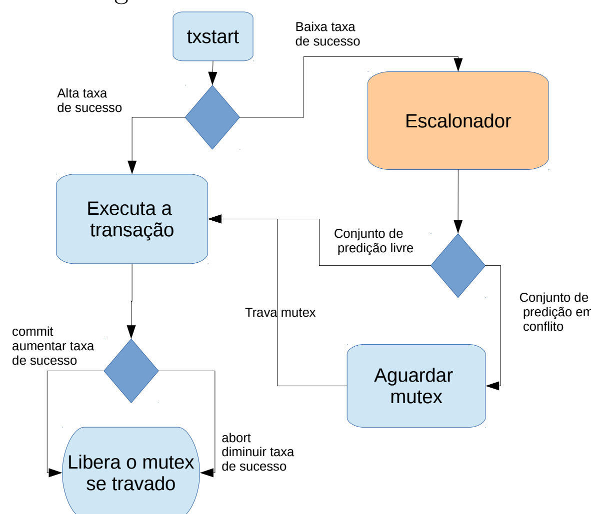
respectivas (3). O retorno do Inibidor de Colisões é utilizado então pelo escalonador de transações (4) para decidir em qual fila a transação será alocada. Após a inserção de uma transação na *thread TQ*, o expedidor sinaliza para a *thread TQ* correspondente que há trabalho disponível.

2.2.3 Shrink

A fim de evitar que as transações sejam arbitrariamente serializadas quando elas poderiam normalmente executar concorrentemente, Dragojevic *et al.* [16] criaram o escalonador Shrink, que traz técnicas mais refinadas para escolher quais transações serializar.

O escalonador Shrink utiliza técnicas de predição baseadas na localidade de referência entre transações para predizer o conjunto de leitura e escrita das transações. A localidade de referência é baseada em duas formas de localidade: a localidade temporal para predizer os conjuntos de leitura; e a localidade entre conjuntos de escrita das

Figura 2.3: Fluxograma de funcionamento do escalonador Shrink



Fonte: [16]

transações abortadas para predizer conjuntos de escrita.

Após observar os acessos de memória consecutivos realizados em transações efetivadas nos *benchmark* STMBench7 [21] e STAMP [43], os autores identificaram que as transações acessam endereços de memória próximos, e que esses acessos podem ser utilizados para prever os futuros conjuntos de leitura. Shrink baseia-se também na predição do conjunto de escrita para evitar o início de uma transação que potencialmente escreveria na mesma posição de memória de alguma outra transação em execução (evitando conflitos de escrita após escrita).

O Shrink introduz ainda a heurística de afinidade de serialização (*serialization affinity*) que indica quando o escalonador deve ser usado baseando-se em uma taxa de sucesso das transações, de modo semelhante à intensidade de contenção do ATS. Se uma *thread*, ao iniciar uma transação, encontra alta taxa de sucesso, a transação inicia imediatamente; caso contrário o escalonador é chamado. A transação pode ser serializada ou não com base na predição dos conjuntos de leitura e escrita. A taxa de sucesso é incrementada ou decrementada em caso de sucesso ou falha

da transação, respectivamente. A Figura 2.3 ilustra esse processo.

2.2.4 Suporte de Escalonamento no *kernel*

Para tratar o não determinismo e falta de precisão do sistema operacional em tratar *threads* que estão executando transações, Maldonado *et al.* [40] propõem diversas soluções no nível do *kernel*. Maldonado *et al.* também propõe a serialização de uma transação caso ela encontre um conflito, fazendo com que ela fique em espera até que a outra transação termine.

Essa espera poderia inicialmente ser implementada em nível de usuário, utilizando-se um *spinlock*, mesmo que essa solução desperdice ciclos do processador. Uma solução no nível de *kernel* bloquearia a *thread* usando chamadas do sistema com uma variável de condição. No entanto, utilizar essas chamadas do sistema pode causar um *overhead* dispendioso se as transações forem curtas ou se houver pouca contenção. Desta forma, uma das soluções desenvolvidas é a utilização de uma área de memória compartilhada entre o *kernel* e a biblioteca de STM, para a troca de informações sobre o estado das *threads* transacionais, em oposição ao uso de chamadas ao sistema para bloquear e acordar *threads*.

Além disso, uma *thread* pode ter sua prioridade diminuída se ela conflitar com outras *threads*, em vez de bloquear em uma condição de espera e diminuir o determinismo se ela for iniciada em um novo endereço.

Como alternativa ao bloqueio de transações é proposta a serialização através da extensão da fatia de tempo. Uma *thread* que estiver executando uma transação pode continuar até o término de seu trabalho ou até um limite máximo de tempo, evitando assim que outra transação escalonada pelo sistema entre em execução e conflite com a transação que entrou em espera.

Tabela 2.1: Interface de funcionamento do escalonador LUTS

Rotina	**Descrição**
Interface de *threads*	
$luts_init(num_threads)$	Inicializa o sistema com um número especificado de *threads*
$luts_start(func, args)$	Após a chamada, cada *thread* começa a executar a função $func$ com o argumento $args$
$luts_barrier_wait()$	A execução só continua se todas as *threads* chegarem à barreira
$luts_shutdown()$	Limpa o estado do LUTS e sai
Interface de escalonamento	
$luts_yield$	O contexto atual é inserido no final da fila de execução, e outra contexto na cabeça da fila é tomado.
$luts_getid(ctxId)$	Retorna um número identificador do contexto de execução atual
$luts_switch(ctxId)$	O escalonador irá trocar o contexto atual por outro na fila de execução que tenha um Id diferente do especificado
$luts_switch_to_id(ctxId)$	O escalonador irá trocar o contexto atual por outro que tenha o Id especificado

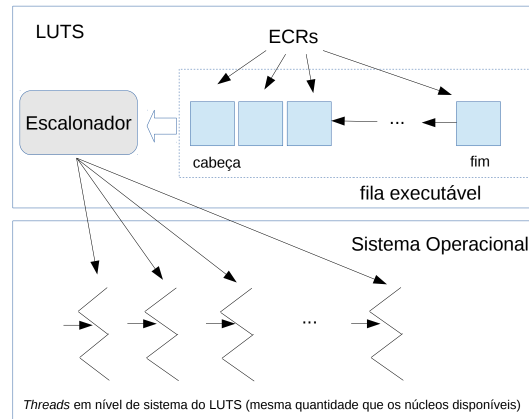
Fonte: [49]

2.2.5 LUTS

O LUTS (*Lightweight User-level Transactional Scheduler*) foi criado com o intuito de oferecer uma solução alternativa além da serialização [49]. O LUTS oferece duas interfaces para o gerenciamento de transações: interface de *threads* e interface de escalonamento. A interface de *threads* do LUTS fornece, em nível de usuário, rotinas para inicialização, criação, utilização de barreiras e encerramento mostradas na Tabela 2.1.

O funcionamento da interface de *threads* do LUTS é exemplificado na Figura 2.4. O escalonador LUTS faz o controle das *threads* transacionais na estrutura de *threads* em nível de sistema chamada aqui de *System Level Threads* ou SLT. Cada SLT é vinculada a um núcleo do

Figura 2.4: Mapeamento de *threads* no LUTS



Fonte: [49]

sistema, sendo o total de SLTs lançadas igual a quantidade de núcleos do processador utilizado. Ao iniciar o sistema de STM com $stm_init()$, as *threads* criadas são contidas em uma estrutura que salva o seus contextos de execução aqui chamados de ECR (Execution Context Record), e postas em fila. O escalonador então mapeia os ECRs para os SLTs disponíveis. Essa técnica evita o pseudo-paralelismo ao não permitir que mais *threads* do que a quantidade de núcleos disponíveis estejam executando ao mesmo tempo. A interface de escalonamento oferece no nível de usuário ferramentas para o controle das *threads* como a troca de contexto e retirada da fila de execução. Esse interface permite um maior controle sobre quais *threads* devem executar e ainda permite heurísticas para escolher as transações com a menor probabilidade de conflito.

No trabalho original do LUTS [49] (referenciado aqui como LUTS-Dynamic) foram propostos dois tipos de heurísticas se baseando no tempo médio de execução das transações. Se as transações forem curtas uma heurística mais simples é usada. Se as transações forem longas uma heurística mais refinada é usada, uma vez que o *overhead* gerado por essa heurística pode ser amortizado.

Na heurística mais simples, a equação de intensidade de conten-

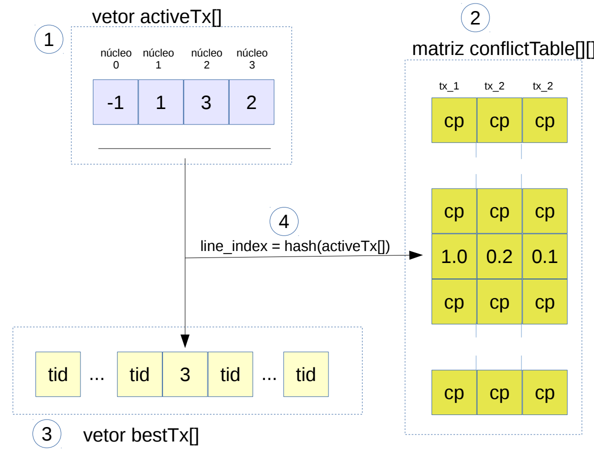
ção (CI) criada por Yoo e Lee [58] é utilizada para definir a probabilidade de conflito de uma transação. Se esse valor for maior que um limite estabelecido outra transação é escolhida para executar. Essa heurística é referenciada como CILUTS [48].

Para transações longas uma heurística que utiliza estruturas de tabelas hash é utilizada, conhecida como HASHLUTS. Esta heurística utiliza três estruturas principais, ilustradas na Figura 2.5: um vetor com as transações ativas (activeTx) em 1, uma tabela de conflito (conflictTable) em 2, e um vetor com o resumo das melhores transações candidatas (bestTx) em 3. Cada posição no vetor activeTx representa um núcleo e o valor armazenado representa a identificação (ID) de uma transação atualmente executando. Caso um núcleo não esteja executando nenhuma transação, a posição no vetor terá o valor -1. Caso um núcleo não esteja executando nenhuma transação, neste exemplo o núcleo 0, ele deve verificar a tabela conflictTable, para escolher a transação que tem a menor probabilidade de conflito dada por seu valor cp. A tabela conflictTable armazena em cada linha os valores de probabilidade de conflito para um determinado conjunto de transações ativas em activeTx. A definição de qual linha será utilizada é obtida a partir de uma função hash aplicada sobre o conjunto de transações ativas em 4.

2.2.6 Comparativo entre os escalonadores

Nas sub-seções anteriores foram vistos os principais escalonadores estudados e suas características. Os mesmos diferem entre si em implementação e complexidade mas tem entre si algumas similaridades. A Tabela 2.2 traz um resumo das principais características entre os escalonadores estudados, onde podemos destacar o escalonador LUTS, devido a sua heurística dinâmica, e a serialização de transações que é

Figura 2.5: Heurística para transações longas do LUTS - HASHLUTS



Fonte: [49]

comum aos outros escalonadores apresentados.

2.3 Medição de energia

O crescente interesse da comunidade científica em aspectos ambientais como o aquecimento global, consumo de energia, geração de energia de fontes renováveis e mais limpas, refletiu também na indústria [53]. O interesse em avaliar o consumo de energia de aplicações teve um aumento significativo recentemente, principalmente porque a busca constante por desempenho deve acompanhar o uso dos recursos energéticos quanto a alimentação e refrigeração dos grandes servidores e *data centers*, adiciona-se a isso a questão recorrente sobre economia de energia em dispositivos móveis que dependem de bateria [46] [9] [20] [53].

Como ficou evidente na seção anterior, os escalonadores de transação tem como foco principal o aumento do desempenho e não levam em conta aspectos de consumo de energia. Visando o desenvolvimento de heurísticas que visem a redução desse consumo, essa seção apresenta a metodologia adotada para a medição do consumo.

Tabela 2.2: Comparativo entre escalonadores de transação

	Ativação em alta contenção	Controle de *threads*	Heurística dinâmica	Serialização
ATS	X			X
CAR-STM		X		X
Shrink	X			X
Escalonador em Kernel		X		X
LUTS		X	X	

Fonte: Autor

2.3.1 A interface RAPL

Um crescente interesse dos pesquisadores na redução do consumo de energia de *hardware* e *software* levou fabricantes como Intel e AMD a fornecerem meios de se obter essas medições de maneira mais prática [24].

Com o lançamento da microarquitetura Sandy-Bridge na família Core pela Intel, um novo método de avaliação do consumo de energia foi implementado diretamente no processador através dos Registradores Específicos de Modelo (MSR's) [13] [6]. Novos registradores não arquiteturais foram adicionados aos já existentes na arquitetura Intel trazendo novas funcionalidades de gestão de energia. Esses registradores fazem parte da interface RAPL (*Running Average Power Limit*) para gerenciamento de energia [22].

A interface RAPL fornece meios para o gerenciamento de energia através da escrita e leitura dos seus registradores de desempenho. A interface RAPL permite o controle de energia dinâmico entre outras possibilidades descritas abaixo:

Tabela 2.3: Domínios que integram a interface RAPL da Intel

Domínio	**Descrição**	**Desktop**	**Servidor**
Package (PKG)	todo o *socket*	X	X
Power Plane 0	núcleos do processador	X	X
Power Plane 1	dispositivos não ligados aos núcleos (vga e outros *in socket*)	X	
DRAM	soma de energia na interface de memória do *socket*		X

Fonte: [22]

- *Power Limit*: Registradores de interface para especificar limitação de energia;
- *Energy Status*: Interface de medição de energia para prover informação sobre a energia consumida;
- *Perf Status*: Interface para informação sobre desempenho afetado pela limitação de energia;
- *Power Info*: Interface que provê informações sobre parâmetros para um dado domínio;
- *Policy*: Informação sobre prioridade que serve de dica para o *hardware* gerenciar a energia entre os domínios principais.

Essas características estão disponíveis em grande parte, dependendo da versão do processador, para os seus quatro domínios principais descritos na Tabela 2.3.

Os domínios da interface RAPL disponíveis dependem do tipo da plataforma (desktop/servidor) analisada. PKG integra PP0 (que são os núcleos do processador), PP1 (descritos como uncore, são outros componentes no mesmo chip do processador como por exemplo componentes gráficos), e DRAM avalia a energia na comunicação com a memória.

Tabela 2.4: Funções presentes na API para a leitura de consumo energético

Função	Descrição
msrInit()	Inicialização da API
msrGetCounter()	Retorna o valor de energia atual
msrDiffCounter(valor0, valor1)	Retorna a diferença de energia consumida entre dois estados diferentes

Fonte: [6]

Em uma plataforma desktop PKG é a soma de PP0 e PP1. E em uma plataforma servidor PKG é a soma de PP0 e DRAM.

API para a leitura de energia consumida

No Linux, os registradores específicos ficam disponíveis para leitura e escrita em nível de kernel, o que pode tornar o seu uso menos trivial. Para facilitar o acesso e a medição de energia, uma API em linguagem C foi criada por Carvalho, Baldassin e Azevedo [6], facilitando a sua integração nas aplicações em que se deseja medir o consumo de energia. Suas funções estão descritas na Tabela 2.4.

Após a inicialização da API com $msrInit()$, os valores retornados por $msrGetCounter()$ são armazenados em variáveis do tipo inteiro para serem posteriormente utilizados na função $msrDiffCounter$, a qual retorna um valor do tipo double com a energia consumida em Joules. O uso da API facilita a medição do consumo, pois pode ser facilmente integrada nas aplicações. Essa API foi integrada aos *benchmark* STAMP[43] para avaliar o consumo de energia entre os escalonadores que serão avaliados neste trabalho.

2.3.2 Métricas para avaliação do consumo de energia

A interface RAPL fornece informações sobre o consumo de energia, realizado através da observação da energia consumida em joules (J) por seus domínios. Embora esse seja um importante dado, a redução de seu valor não significa necessariamente que aumentamos a eficiência. Por exemplo, podemos reduzir a potência consumida simplesmente reduzindo a frequência de operação, ou diminuindo a alimentação. Em ambos os casos, o tempo necessário para se terminar o trabalho aumentará, sem necessariamente ter consumido menos, pois haverá consumo de potência por um período maior de tempo. Uma solução a esse problema é a utilização da métrica *Energy-Delay Product* ou EDP [30]. Com o EDP analisamos em conjunto os valores de energia e tempo, fazendo com que as variações de desempenho e eficiência energética tenham o mesmo peso quando comparamos diferentes cenários. O EDP é dado aqui pelo produto consumo(*energy*) e tempo (*delay*) conforme a Equação 2.2.

$$EDP = \triangle t \times \triangle J \tag{2.2}$$

Uma variante dessa métrica é a métrica *Energy-Delay Square product* ou ED^2P [42], porém essa métrica é mais utilizada quando damos mais prioridade ao desempenho [5].

2.4 Dynamic Voltage and Frequency Scaling

Este trabalho tem como proposta tirar proveito das tecnologias de escalonamento de frequência e tensão dinâmicas (*Dynamic Voltage and Frequency Scaling* – DVFS). Com a ideia de reduzir o consumo de energia e melhorar o desempenho quando possível, AMD e Intel desen-

volveram para seus processadores diferentes técnicas de DVFS [56][39].

2.4.1 P-States

Os P-States são os estados de desempenho (*Performance States*). Cada nível define um valor de tensão e frequência de trabalho diferente com incrementos fixos de um para o outro. Os P-States disponíveis são dependentes do modelo do processador [29]. Por exemplo, processador Intel Core I7 de segunda geração utilizado nos testes possui 10 P-States que vão de 1,6GHz até 3,4GHz em incrementos de 200MHz.

2.4.2 C-States

Os C-States são os estados de baixa energia utilizados para se reduzir o consumo de energia quando há recursos ociosos no processador. O estado C0 é o estado em que o processador ou núcleo está em pleno funcionamento, enquanto os estados de C1 em diante são os estados em que se opera em baixo nível de energia, desligando em parte ou totalmente o núcleo e o *clock*. Quanto mais profundo o estado, mais demorada a entrada e saída desse estado. Os P-States somente têm efeito quando se está em C0 [29].

A Tabela 2.5 detalha os C-States principais em um processador Intel Core de 4ª geração [31]. AMD e Intel possuem para cada arquitetura diferentes configurações de C-States disponíveis, para o conjunto do procesador e para os núcleos individualmente e suas combinações. Outras configurações de C-States são omitidas por não estarem disponíveis na arquitetura utilizada como exemplo.

Tabela 2.5: C-States disponíveis em um processador Intel de 4ª geração

Estado	Descrição
C0	Modo ativo
C1	Estado de parada
C1E	Estado de parada com a frequência mais baixa e menor tensão de operação
C3	Núcleos ativos descarregam sua memória *cache* de baixo nível (L1 e L2) para a memória *cache* de último nível. O *clock* é desligado para os núcleos nesse estado ou processador.
C6	Núcleos nesse estado salvam seu estado arquitetural antes da alimentação para cada núcleo ser removida.
C7	Comportamento semelhante ao C6. Se todos os núcleos requisitarem esse estado, a memória *cache* de último nível também é esvaziada, e tem sua alimentação desligada.

Fonte: [31]

2.4.3 Turbo Boost e Turbo Core

As tecnologias "turbo" nos processadores Intel e AMD necessariamente precisam de certas combinações de P-States e C-States para serem ativados. O "turbo" permite que, em certas condições, a frequência de operação se eleve acima da frequência nominal respeitando-se o seu projeto de energia térmica (*Thermal Design Power* – TDP), geralmente quando alguns núcleos estão em estado de dormência (C1, C2, ...). O processador utilizado nos testes (Core I7 2600K) tem frequência nominal de 3,4GHz com turbo boost máximo de 3,8GHz. Para se alcançar essa frequência somente um dos quatro núcleos deve estar ativo. A frequência máxima diminui de acordo com o número de núcleos no estado C0. Com 2 núcleos ativos a frequência máxima é 3,7GHz, com 3 é 3,6GHz e com 4 núcleos ativos a frequência é 3,5GHz [2].

2.4.4 Turbo Boost 2.0

O Turbo Boost 2.0 permite ao processador operar acima do TDP por um curto período de tempo se obedecidas as condições para

operação em turbo. Presente nas arquiteturas Intel Sandy Bridge e mais novas, essa tecnologia conta com diversos sensores para acompanhar o funcionamento do processador [1].

Diferente do que acontecia com sua versão inicial, o aumento de temperatura é visto pelo processador de forma gradativa, permitindo ao processador operar acima do TDP enquanto a temperatura ainda não chegou a seu máximo [47].

2.4.5 Controle de frequência e *governors*

O sistema operacional ou usuário pode requisitar P-States ao processador com a escrita de registradores específicos de acordo com a arquitetura do processador. Isso é facilitado com o uso do *driver* $cpufreq$ [29] ou do *driver* Intel P-State [51], utilizados como padrão a partir do *kernel* 3.9 do Linux para processadores Intel mais recentes [37]. O controle dinâmico de frequência é feito pelo sistema operacional através de suas políticas de gerenciamento de frequência ou *governors* [29]. Para o módulo ACPI-cpufreq temos 5 governors principais, que são:

- *Ondemand* - quando a carga de trabalho está acima de um determinado limiar, a frequência irá imediatamente para a máxima disponível;
- *Userspace* - o usuário pode definir a frequência de operação entre os P-States disponíveis. Os modos de turbo não ficam disponíveis diretamente [39];
- *Conservative* - quando a carga de trabalho aumenta acima de um limiar a frequência é configurada para trabalhar em uma frequência imediatamente acima da atual. Se a carga de trabalho diminui, o inverso ocorre;

- *Powersave* - a frequência de trabalho é a mínima disponível;
- *Performance* - a frequência de trabalho é a máxima disponível.

Os processadores da Intel e AMD têm comportamento diferentes quanto à configuração manual de seus P-States. Enquanto a arquitetura AMD permite um maior controle manual e configuração de frequências distintas entre seus núcleos ou par de núcleos (caso da APU FX) [56], a arquitetura Intel mantém um maior controle com o *hardware*. Deve-se considerar ainda que somente as políticas *powersave* e *performance* estão disponíveis quando se utiliza o *driver* Intel P-State, no qual o *hardware* controla os P-states [4].

Os núcleos ativos nos processadores Intel Sandy Bridge e posteriores possuem a mesma frequência de operação entre eles, mesmo se alguns núcleos (ou *threads*) requisitarem uma frequência diferente, apesar deles poderem estar em diferentes C-States. Nessas arquiteturas o processador sempre terá a maior frequência requisitada entre os núcleos ativos [56]. Por exemplo, dois núcleos 0 e 1 podem estar configurados a trabalharem em 1,6GHz, o núcleo 2 em 2,0GHz e outro núcleo 3 em 3,4GHz. Enquanto o núcleo 3 estiver ativo, ele será a base para a operação do processador.

Uma melhoria da arquitetura Haswell (4ª geração da arquitertura Intel Core) é a integração de reguladores de tensão internos (*Fully Integrated Voltage Regulator* – FIVR) dentro do pacote do processador, permitindo a regulagem de tensão entre os diversos núcleos e dispositivos dentro do processador [23][31]. Isso permite melhor otimização entre desempenho e economia de energia [32]. Embora permita frequências distintas entre os núcleos (o *clock* e a tensão são reduzidos ao mesmo tempo), o controle dinâmico de frequência e o Turbo boost se mantém sob controle do processador. O FIVR foi removido na 6ª geração da

arquitetura Core (Skylake) para dar mais espaço para overclock [11].

2.5 Avaliação de consumo de energia em Memória Transacional

Esta seção fará um levantamento sobre trabalhos anteriores que avaliaram o consumo energético em Sistemas de Memória Transacional em *software* e *hardware*. Moreshet, Bahar e Herlihy [45] avaliam o consumo de energia em Memória Transacional em *Hardware* comparando com travas tradicionais. Esse trabalho foi complementado mais tarde em [46], no qual os autores encontraram menor consumo de energia nas aplicações utilizando memória transacional quando a contenção em geral é baixa, e alto consumo quando a contenção é alta. Os autores propõem a serialização de transações para a redução de consumo. Esse trabalho foi um dos primeiros a comparar o uso de Memória Transacional em *Hardware* e travas, conduzido em um ambiente simulado.

Também utilizando ambientes simulados, Baldassin *et al.* [8] investigam o consumo de energia utilizando Memória Transacional em *Software*. Utilizando o *benchmark* STAMP [43] e a biblioteca de Memória Transacional em *Software* TL2 [14], os autores avaliam o consumo de Memória Transacional nas configurações *eager* e *lazy*, propondo uma política baseada em DVFS para os casos em que o gerenciador de contenção coloca transações em espera de modo a economizar energia, estratégia semelhante ao apresentado no trabalho de Li, Martinez e Huang [38].

No trabalho de Klein *et al.* [36], um estudo sobre consumo de energia em Memória Transacional em Software foi feito em um ambiente simulado utilizando memórias *scratchpads*. Implementações de Memória Transacional em *Software* utilizando essa estrutura foram criadas para

reduzir o consumo de energia em situações de alta contenção, sendo esses experimentos analisados utilizando algumas aplicações do *benchmark* STAMP. Esse trabalho pode ser considerado complementar ao trabalho de Ferri et al. [18], os quais utilizam a mesma estrutura de *scratchpads* mas fazem uma análise de energia utilizando suporte de Memória Transacional em *Hardware*.

Análise semelhante ao primeiro trabalho de Baldassin *et al.* [8] foi feita por Gaona-Ramírez *et al.* [19], porém analisando o funcionamento e consumo de energia em Memória Transacional em *Hardware*. Esse trabalho analisa dois Sistemas de Memória Transacional em *Hardware* conhecidos: o LogTM *eager-eager* e o Scalable TCC *lazy-lazy* (versionamento de dados-detecção de conflitos). Os autores destacam que a redução no tempo de execução não é sempre proporcional a redução do consumo de energia.

Em um trabalho mais recente de Baldassin *et al.* [7], são investigados o consumo de energia em diferentes tipos de algoritmos de memória transacional combinados às políticas de gerenciamento de contenção (SUICIDE, BACKOFF, DELAY), incluindo ainda testes com um DVFS aprimorado, utilizando ambiente simulado para o levantamento do consumo de energia executando aplicações do *benchmark* STAMP. Rico, Pilla e Bois [52] avaliam o consumo de energia em três bibliotecas de memória transacional em software: TL2, SwissTM e TinySTM, utilizando dados de energia fornecidos por um microcontrolador presente na placa mãe de servidores (*Baseboard Management Controller* – BCM). A avaliação foi feita em aplicações do *benchmark* STAMP.

Já utilizando a interface RAPL presente em processadores Intel Sandy Bridge [22], Gautham *et al.* [20] fazem uma comparação do consumo de energia entre o sistema de memória transacional em *software* da Universidade de Rochester (RSTM) [41] utilizando o al-

goritmo Swiss, travas tradicionais e *spinlocks*, executando aplicações do *benchmark* STAMP. Assim como em avaliações anteriores, o uso de memória transacional em software apresentou menor consumo de energia em comparação ao uso de travas. Neste trabalho também é apresentado como a utilização de estados de conservação de energia presente nos processadores influencia no desempenho e consumo.

Recentemente, Carvalho, Baldassin e Azevedo [6] reavaliaram o consumo de energia em Memória Transacional em *Software* através da interface RAPL, utilizando a biblioteca TinySTM e o *benchmark* STAMP. Complementarmente, variações das políticas de gerenciamento de energia presentes no kernel do Linux também foram consideradas nos testes, além da criação de uma API para facilitar a leitura dos dados da interface RAPL de baixo nível.

Em um trabalho mais recente, Rughetti, Sanzo e Pellegrini [53] avaliam o desempenho e consumo de energia em configurações de Memória Transacional em *Software* auto adaptativas. Variações da biblioteca TinySTM incorporadas a um algoritmo de aprendizado incluindo os escalonadores ATS e Shrink [55] são avaliados em uma plataforma de processador AMD para servidores utilizando sistema operacional Linux.

Com o objetivo de se aproveitar das tecnologias de DVFS para reduzir o consumo de energia, Issa, Romano e Brorsson criaram o gerenciador de contenção Green-CM [33]. O gerenciador Green-CM é baseado na política *backoff*, mas esta é implementada de maneira híbrida de modo a alternar entre fazer a *thread* esperar um spin (mantendo o processador ativo) ou fazer a transação dormir. Além disso um estudo energético comparando diferentes gerenciadores de contenção foi realizado utilizando os *benchmarks* STAMP, Stmbench7 e Memcached em um ambiente AMD com processador Opteron 6272 de 16 núcleos.

Os trabalhos relacionados até aqui somente fizeram análises de consumo de energia e desempenho utilizando principalmente gerenciadores de contenção, sendo o objetivo inicial dessa pesquisa investigar o consumo de energia em heurísticas baseadas em escalonadores.

Um trabalho preliminar [34] foi realizado nesse projeto de pesquisa, onde foram avaliados escalonadores de transação, dando destaque ao escalonador LUTS[49], onde os dados de tempo e consumo dos escalonadores foram analizados utilizando o *benchmark* STAMP. O escalonador LUTS teve o melhor desempenho nas três aplicações avaliadas, entretanto o escalonador ATS teve o melhor consumo de energia em duas aplicações.

Capítulo 3

A Heurística LUTS-Dynamic-Serializer

O objetivo desse capítulo é apresentar a heurística desenvolvida como parte do trabalho de mestrado para redução do consumo de energia e tempo de execução em sistemas transacionais. Como apontado anteriormente, ainda não existe na literatura nenhum trabalho nesse sentido voltado para os escalonadores de transação.

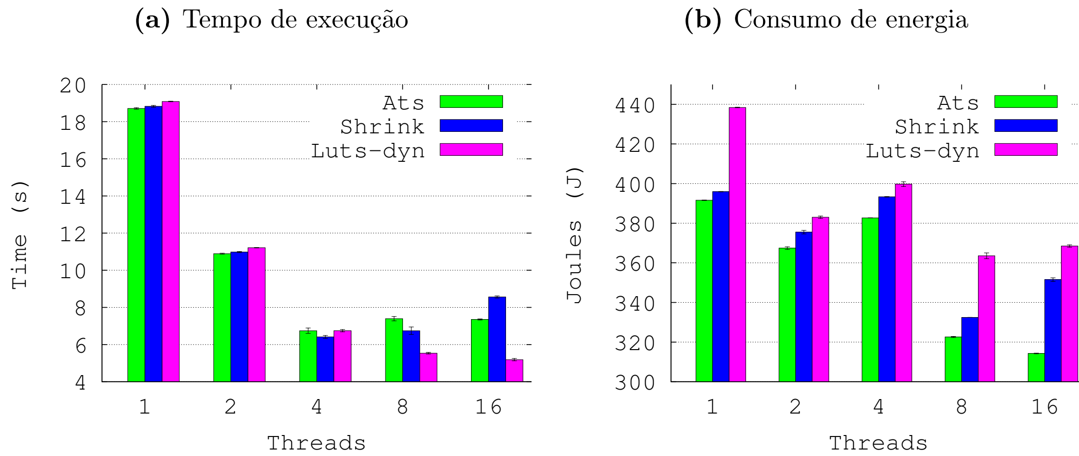
3.1 Abordagem

A ideia inicial para a heurística era utilizar técnicas de DVFS para a redução da frequência de *threads* que não possuíssem a prioridade de execução (por exemplo, porque suas transações tinham um alto índice de falhas). Essa troca de frequência seria realizada pelo escalonador. O controle de frequência mais centrado no hardware presente nos processadores Intel e a sincronicidade entre os núcleos, como descrito anteriormente no Capítulo 2, dificultou esse controle direto.

Resultados preliminares para algumas aplicações do STAMP mostraram que alguns escalonadores forneciam bom consumo de energia, mas com alto tempo de execução. Já outras possuiam bom tempo de

Figura 3.1: Testes preliminares: Consumo de energia e Tempo da aplicação STAMP-Intruder

(a) Tempo de execução **(b)** Consumo de energia



Fonte: Autor

execução, mas com alto consumo de energia. A Figura 3.1 mostra essa situação para a aplicação Intruder do pacote STAMP. É possível notar na Figura 3.1b um maior consumo do escalonador LUTS quando utilizadas 8 e 16 *threads*, enquanto o escalonador ATS possui o menor consumo nesses casos. Uma situação inversa pode ser vista na Figura 3.1a, onde foi analisado o tempo de execução. Um cenário ideal seria a combinação do menor consumo do escalonador ATS com o menor tempo de execução do escalonador LUTS.

O relativamente baixo consumo de energia geralmente conseguido com o escalonador ATS deve-se ao fato da utilização de *mutexes* para serialização. Isso faz com que o sistema operacional coloque a *thread* em estado de espera, reduzindo implicitamente a frequência de operação do processador que estava executando aquela *thread* e consequentemente reduzindo o consumo de energia. No entanto, o *overhead* de colocar a *thread* em estado de espera e acordá-la faz com que o tempo de execução fique maior. Uma alternativa seria então empregar um *spinlock*, que adota um esquema de espera ocupada para obtenção da trava ou *lock*. Com isso, a thread não é retirada de execução, evitando

o *overhead* do tempo por um lado, porém consumindo mais energia por outro.

Com o intuito de prover um melhor compromisso entre consumo de energia e tempo de execução, uma nova heurística para o LUTS foi então desenvolvida, chamada neste trabalho de LUTS-Dynamic-Serializer. Na busca por uma heurística que melhorasse o EDP no escalonador LUTS, foi observado que seria uma boa ideia manter as *threads* acordadas em um *spinlock* e assim evitar o *overhead* de adormecer uma *thread* e acordá-la novamente com o uso de *mutexes*, como acontece com o ATS.

Enquanto verdadeiro para algumas aplicações, para outras esse modo de serialização pode se tornar inviável. É preciso então encontrar uma forma de fazer com que as transações não desperdicem ciclos do processador indefinidamente em cenários de alta contenção. Nesses casos é necessário limitar a quantidade de vezes em que a transação fica no *spinlock*, utilizando *mutex* em caso contrário.

Uma abordagem que poderia ser empregada é adotar alguns limites fixos para permanência em *spinlock* e transição para *mutex*. Note, no entanto, que tal abordagem tem uma importante deficiência: esses limites podem variar de uma aplicação para outra. Há então a necessidade de um controle dinâmico que acompanhe o comportamento das transações durante a execução. Para isso foi desenvolvida a heurística descrita a seguir.

3.2 Heurística

A solução apresentada aqui é derivada do escalonador LUTS integrado a uma heurística originalmente proveniente do ATS denominada anteriormente como CILUTS [48]. Para a heurística criada considerou-

se a serialização das transações assim como ocorre no escalonador ATS original, para melhor aproveitamento das técnicas de DVFS, porém adicionando a opção de serializar também com *spinlock*, desta forma alternando entre *mutex* e *spinlock* dinamicamente.

O *spinlock* tem custo operacional mais baixo e pode oferecer vantagem quando a espera pela prioridade na fila de transação é baixa. No entanto, fazer com que a transação aguarde nessa fila pode desperdiçar ciclos do processador já que as threads ficam constantemente consultando a sua vez de entrar na fila, consumindo energia desnecessariamente. Por outro lado, serializar com *mutex* pode economizar energia por tirar a *thread* de execução e permitir ainda que estados mais altos de ”turbo” sejam alcançados, pois nesse caso somente uma transação deverá estar ativa. Porém, a perda de tempo para se acordar a *thread* pode fazer com que se perca desempenho se a transação anterior terminar em um período pequeno e a contenção for relativamente baixa.

A solução apresentada é melhor revista nas subseções seguintes onde o algoritmo para a heurística é descrito. Na sequência, o processo de escolha dos melhores valores para alternar entre *spinlock* e *mutex* é apresentado.

3.2.1 Algoritmo

A sequência de algoritmos a seguir integram a heurística proposta. A heurística é uma modificação do algoritmo CILUTS para utilizar *spinlock* além do *mutex* tradicional. A serialização do escalonador ATS está presente juntamente com o controle de *threads* do LUTS, enquanto há um acompanhamento dos dados de tempo e quantidade de falhas das transações que são utilizados para definir quando utilizar a serialização com *spinlock* ou *mutex*.

No algoritmo 1 (`stm_prepare`), a transação está iniciando ou

reiniciando depois de um rollback ou *commit*. Se a intensidade de contenção for maior que 70 é então iniciado o processo de serialização (esse parâmetro é original do próprio ATS). Como a transação pode ter vindo de um *rollback*, a linha 2 identifica se a mesma já tinha adquirido o *lock* antes. Caso já tenha o *lock*, nada é feito e o algoritmo chega ao fim; caso contrário o algoritmo continua na linha 3.

Algorithm 1 stm_prepare

```
1:  if CI > 70 then
2:    if tx.lockAdquirido = 0 then
3:      if tx.tipoDeLock = spinlock then
4:        while fila ≠ true and contador < spinMaximo and modoGlobal =
          spinlock do
5:          contador = contador + 1
6:          Tenta mudar fila para true
7:        end while
8:        if fila ≠ true then
9:          tx.tipoDeLock = mutexlock
10:         modoGlobal = mutexlock
11:       else
12:         tx.lockAdquirido = 1
13:       end if
14:     end if
15:     if tx.tipoDeLock = mutexlock then
16:       Tenta pegar lock com mutex
17:       tx.lockAdquirido = 1
18:     end if
19:   end if
20: end if
21: if threadID = 0 then
22:   Armazena o tempo atual em tempoInicial
23: end if
```

A variável `tx.tipoDeLock` é uma variável local da transação e é sempre iniciada com *spinlock*. A variável `contador` é uma variável local inicializada com 0. As constantes `spinlock` e `mutexlock` representam 0 e 1 respectivamente. Já a variável `modoGlobal` é uma variável global que força todas as transações a seguirem o mesmo modo.

Se `tx.tipoDeLock` for igual a `mutexlock` então o algoritmo

segue para a linha 15. Sendo `tx.tipoDeLock` igual a `spinlock`, a transação irá tentar a preferência na fila mudando o valor da variável `fila` para `true` ou 1. No código isso é feito com uma operação atômica fornecida pelo GCC [3] baseada na *"Processor-specific Application Binary Interface"* original dos processadores Intel Itanium [10].

A transação ficará no laço das linhas 4 – 7 enquanto: (i) não conseguir mudar a variável fila; (ii) o valor de `contador` for menor que o número máximo de spins (calculado no algoritmo 2 a seguir); e (iii) `modoGlobal` indicar o uso de `spinlock`. Caso o algoritmo saia do laço e a transação ainda não tenha conseguido a preferência na variável `fila`, ela mudará o seu `tx.tipoDeLock` para `mutexlock`, além do `modoGlobal` para `mutexlock`, interrompendo a tentativa de outras transações de tentar a fila com *spinlock*. Nesse caso a transação deve tentar adquirir a trava com *mutex*, continuando o algoritmo na linha 15.

Caso a transação tenha sucesso com o *spinlock* ou *mutex*, a variável `tx.lockAdquirido` será alterada para 1 nas linhas 12 e 17, respectivamente. Ao final do algoritmo, se a *thread* executando for a *thread* 0, o tempo atual é então salvo em `tempoInicial`.

A Figura 3.2 representa o funcionamento deste algoritmo de forma simplificada.

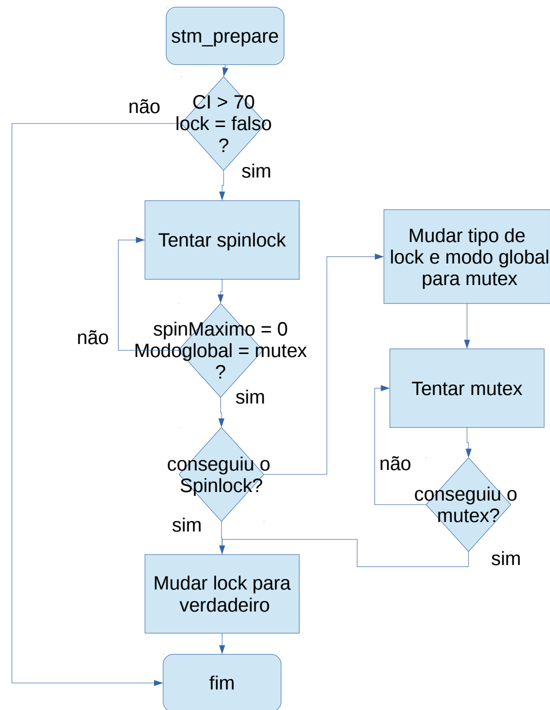
No algoritmo 2 (`stm_commit`), se a transação estiver executando na *thread* 0 ela deve recalcular o spin máximo utilizando o tempo médio das transações e o número de *aborts* (linhas 1–24). As transações atualizam adequadamente a intensidade de contenção nas linhas 25 e 26, liberam o respectivo *lock* caso necessário nas linhas 27 a 33, e finalmente decidem sobre o tipo de *lock* nas linhas 34 a 37.

Para o tempo médio das transações são acumulados os tempos de 100 *commits* desde o tempo inicial até o tempo final na variável `tempoAcumulado` (linha 2). O contador é incrementado a cada *commit*

Algorithm 2 stm_commit

1: **if** $threadID = 0$ **then**
2: $tempoAcumulado = tempoAcumulado + (tempoAtual - tempoInicial)$
3: $contador = contador + 1$
4: **if** $contador > 100$ **then**
5: $tempoMedio = tempoAcumulado \div 100$
6: $abortsTotal = abortsAtual - abortsInicial$
7: **if** $abortsTotal = 0$ **then**
8: $indice = tempoMedio$
9: **else**
10: $indice = tempoMedio \div abortsTotal$
11: **end if**
12: **if** $indice > indiceReferencia$ **then**
13: calcular novo $spinMaximo$
14: **else**
15: $spinMaximo = 0$
16: **end if**
17: **if** $spinMaximo < 0$ **then**
18: $indiceReferencia = indice$ {valor usado para evitar que um valor menor que zero seja calculado novamente}
19: **end if**
20: $abortsInicial = abortsAtual$
21: $tempoAcumulado = 0$
22: $contador = 0$
23: **end if**
24: **end if**
25: $CC = 0$
26: $CI_n = \alpha \times CI_{n-1} + (1 - \alpha) \times CC$ {atualiza intensidade de contenção}
27: **if** $tx.lockAdquirido = 1$ **then**
28: **if** $tx.tipoDeLock = spinlock$ **then**
29: mudar $fila$ para $false$
30: **else**
31: liberar $lock$ mutex
32: **end if**
33: **end if**
34: **if** $CI < limiarDeTransicao$ **then**
35: $tx.tipoDeLock = spinLock$
36: $modoGlobal = spinLock$
37: **end if**

Figura 3.2: Fluxograma de funcionamento simplificado do algoritmo 1 (stm_prepare)



Fonte: Autor

(linha 3). Se o contador for maior que 100 (linha 4) então o `tempoMedio` será o `tempoAcumulado` dividido por 100. O `abortsTotal` é calculado fazendo-se a subtração da quantidade de *aborts* atual (`abortsAtual`) da de *aborts* inicial (`abortsInicial`), na linha 6. A variável $abortsInicial$ recebe valor zero no começo da aplicação, e é atualizado com o valor de *aborts* atual a cada intervalo de medição. O índice a ser utilizado na fórmula do `spinMaximo` é calculado nas linhas 7 a 11, fazendo-se a divisão de `tempoMedio` pela quantidade de *aborts*. A verificação na linha 7 é feita para evitar uma divisão por zero.

A variável global `indiceReferencia`, presente na linha 12, foi adicionada para evitar que seja calculado o `spinMaximo` novamente sabendo-se que o índice encontrado será menor que zero. Se o spin máximo encontrado for menor que zero, o valor do índice deve ser guardado em uma variável de referência que será utilizada na próxima

medição (linhas 17–19). As variáveis que armazenam a quantidade de *aborts*, o tempo acumulado e o contador são ajustadas nas linhas de 20 a 22.

A intensidade de contenção é calculada nas linhas 25 e 26. Nas linhas 27 a 32 o *lock* apropriado é liberado caso necessário (de acordo com `tx.tipoDeLock`). Finalmente, nas linhas 34 a 37 o tipo de *lock* é trocado caso a intensidade de contenção atual for menor que um valor pré-estabelecido dado por `limiardeTransicao`. A variável `limiardeTransicao` define o valor de CI em que haverá a troca entre o modo de *mutex* para o modo de *spinlock*. Se a intensidade de contenção for menor que o valor definido a essa variável, a transação deve voltar a utilizar *spinlock* e forçar as outras transações a utilizarem o mesmo modo alterando a variável `modoGlobal`.

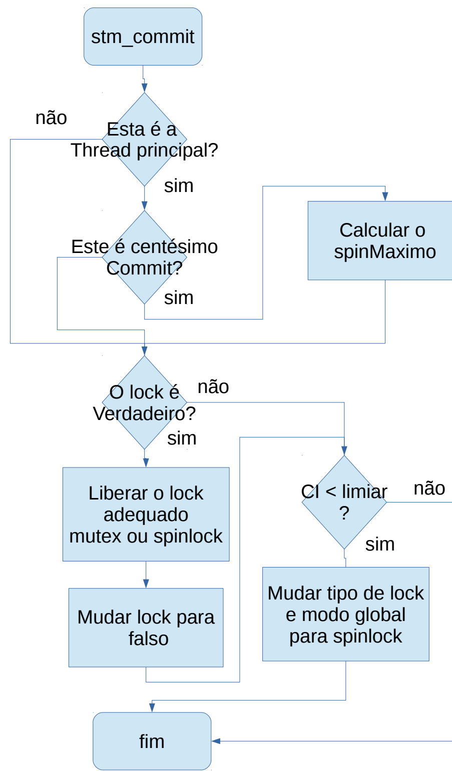
A Figura 3.3 representa o funcionamento deste algoritmo de forma simplificada.

Um exemplo de funcionamento da heurística aqui descrita é mostrado na Figura 3.4 para 4 *threads* (T). As setas representam o progresso de execução de cada thread T no instante de tempo t.

No momento $t0$ temos as transações sendo executadas livremente em paralelo enquanto a intensidade de contenção(CI) é menor do que 70. No momento $t1$, a transação na *thread* 1 identifica alta contenção e inicia a serialização com *spinlock* e, como nesse ponto ela é a única a serializar, a transação logo inicia. Entre os pontos $t1$ e $t2$ outras transações tendem a serializar com *spinlock*, até que no instante $t2$ a transação na *thread* 0 chega ao limite máximo de *spinlock* e muda o método de serialização para mutex, forçando a transação na *thread* 3 a seguir o mesmo modo mesmo que ela não tenha chegado ao limite de *spinlock*.

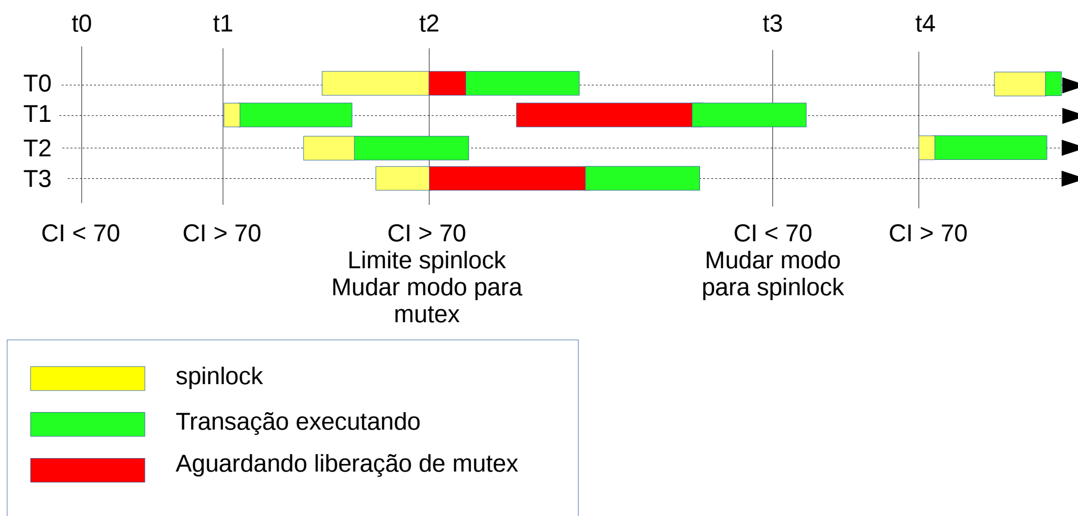
Entre os pontos $t2$ e $t3$ as transações estão no modo de seriali-

Figura 3.3: Fluxograma de funcionamento simplificado do algoritmo 2 (stm_commit)



Fonte: Autor

Figura 3.4: Exemplo de funcionamento com 4 *threads*



Fonte: Autor

zação com *mutex*. Esse estado é modificado quando uma transação na *thread* 2 (a partir do momento $t3$) identifica que a contenção diminuiu e então configura o modo global para *spinlock* novamente ($t4$). Note que é possível duas transações adquirirem os *locks* em duas filas diferentes (*spinlock* ou *mutex*) por um curto período de tempo.

3.2.2 Definição da fórmula de spin máximo

Esta sub-seção descreve a metodologia realizada para a criação de uma equação para obtenção de um valor máximo de spin que se adeque ao comportamento das transações. Esta equação é utilizada no Algoritmo 2, linha 13, e foi obtida a partir de diversos testes descritos abaixo, onde puderam ser analisados os melhores resultados para cada configuração.

Inicialmente foram feitos testes com versões do algoritmo com e sem a transição de *spinlock* para *mutex*, com valores fixos para `spinMaximo`, a fim de se observar o comportamento das transações. Não fazer a transição para *mutex* depois de um certo tempo fez com que certas aplicações do STAMP não chegassem ao fim ou levassem a tempos muito altos para serem concluídas. O consumo de energia também se elevou consideravelmente.

Também foram feitos testes quanto ao uso da variável `globalMode`. Essa variável força todas as *threads* a usarem um mesmo modo, indicando qual é o modo atual. O não uso dessa variável permite que se tenha na prática duas filas concorrentes: uma com *mutex* e outra com *spinlock*. Ao final, deciciu-se por evitar que isso aconteça, para que as técnicas de "turbo"fossem melhor aproveitadas.

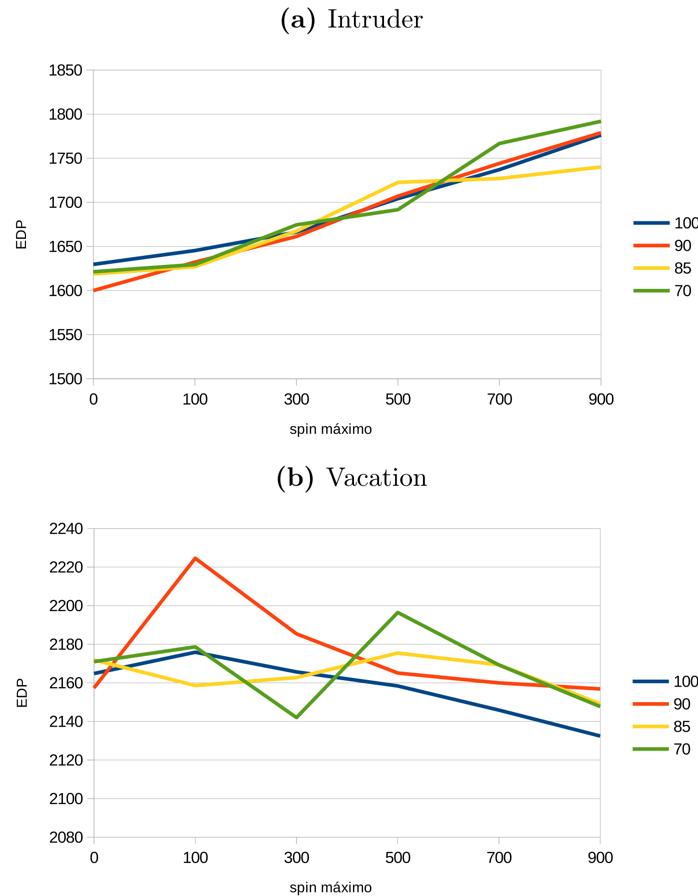
Como o valor de spin pode variar de aplicação para aplicação, foram realizados testes de sensibilidade utilizando inicialmente os valores de 500, 1000, 2000 e 5000 como limite. Os resultados obtidos com

os valores acima de 1000 se mostraram impraticáveis para algumas aplicações. A partir da constatação anterior foram realizados novos testes com valores menores. Os novos valores utilizados foram: 0, 100, 300, 500, 700 e 900. A partir dos resultados obtidos foi possível identificar aplicações que tiveram o seu melhor desempenho (energia e tempo) em faixas de valores distintos de limite para *spinlock*.

Aplicações como Genome, Kmeans, Intruder e Yada tiveram melhores resultados com números de spins baixos, enquanto outras com número de spins mais altos ou intermediários. Um número de spin baixo significa que a transação ficará pouco tempo em *spinlock*, alternando rapidamente para *mutex*.

As Figuras 3.5a e 3.5b ilustram os testes realizados com os limites de *spinlock* fixos para as aplicações Intruder e Vacation. O eixo x representa os valores utilizados como limite para *spinlock*, e o eixo y representa os valores de EDP. As linhas representam os testes utilizando diversos valores para a variável `limiardeTransicao`. É possível notar os comportamentos distintos entre as duas aplicações.

Figura 3.5: Testes de sensibilidade com valores máximos de spin fixos para aplicações STAMP com 8 *threads*



(a) Intruder

(b) Vacation

Fonte: Autor

O escalonador LUTS-Dynamic já usa o tempo médio das aplicações para decidir qual de suas heurísticas utilizar. Porém, utilizar somente esse parâmetro (tempo) para o caso da heurística que propõe otimizar o EDP mostrou-se ineficiente. Desta forma, a quantidade de *aborts* foi considerada para acompanhar o comportamento das transações, com o objetivo de identificar qual valor máximo de spin utilizar para cada aplicação dinamicamente.

Acompanhando também a estatística de *aborts*, viu-se que a mesma poderia ser utilizada para identificar as aplicações em que se deveria aplicar um número diferente de spin máximo. Foi identificado

que as aplicações em que a relação entre o tempo médio e a quantidade de *aborts* fosse baixa (como, por exemplo, em transações com tempo reduzido e alta contenção), o número de spin máximo poderia ser maior. Já para transações com alto número de *aborts* comparado ao tempo médio, pouco tempo deveria ser gasto em *spinlock*.

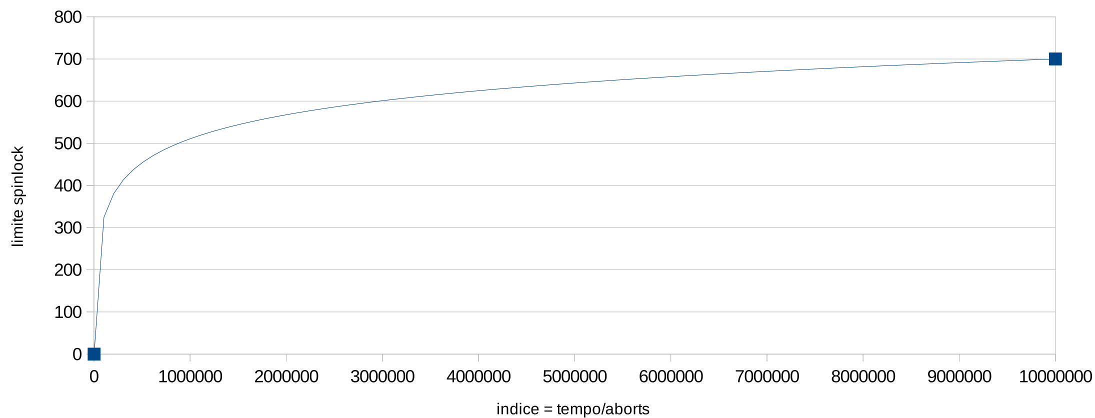
Observando os melhores valores de spin e essa relação, foi possível perceber que esses dados poderiam ser utilizados para o cálculo de spin máximo. A relação de tempo médio e a quantidade de *aborts* foi observado para as aplicações do STAMP, onde foi definido um limiar inferior (valor em que abaixo dele o *spinlock* máximo deveria ser zero).

Transferindo esses valores para um plano cartesiano onde o eixo x é a relação tempo médio/*aborts* e y é o número máximo em um *spinlock*, uma reta crescente pode ser traçada. Porém, para a aplicação Labyrinth, o valor resultante de y nessa reta seria um valor muito alto, devido ao tempo médio das transações ser da ordem de milhares de vezes maior que as outras aplicações do STAMP. A solução foi então utilizar a projeção de uma curva logarítmica crescente, que é utilizada para o cálculo de $spinMaximo$ presente no algoritmo 3, e descrita pela Equação 3.1.

$$spinMaximo = multiplicador \times log(indice) - correcao \tag{3.1}$$

Na Equação 3.1, o índice é a razão entre o tempo médio e o número de *aborts*, $multiplicador$ e $correcao$ definem o formato e posição da curva no plano cartesiano (Figura 3.6), onde no eixo x se encontram os valores para o índice e no eixo y se encontram os valores de spin máximos a serem utilizados.

Figura 3.6: Projeção de curva de melhores valores máximos de spinlock



3.3 Discussão

Uma técnica semelhante à descrita aqui foi utilizada no gerenciador de contenção Green-CM [33]. Esse gerenciador aprimora a política back-off fazendo a transação atrasar seu reinício em um *spinlock* ou então coloca a *thread* para dormir com uma chamada de sistema. O tempo de espera é calculado dinamicamente baseado nas informações de *commit* e consumo de energia. Se o tempo de espera calculado for maior que um limite estabelecido, a transação é posta para dormir. A heurística desenvolvida neste trabalho se difere do Green-CM, principalmente por ser aplicada a um escalonador de transações, e por usar a média de tempo e número de *aborts* para decidir quando uma transação deve desistir do *spinlock* e usar um *mutex*.

Inicialmente para acompanhar o comportamento das transações tentou-se a leitura da energia consumida pelas transações. O tempo curto em que uma transação ocorre inviabilizou esse tipo de acompanhamento devido à alta granularidade desse dado em razão do tempo de leitura dos registradores da interface RAPL. Mesmo quando se considerou o intervalo de tempo acumulado utilizado para o levantamento

do tempo médio das transações, para transações longas esse intervalo tende a ser suficiente, porém o mesmo não acontece para transações curtas. O gerenciador de contenção Green-CM reserva *threads* para fazer esse acompanhamento enquanto a aplicação está sendo executada para modificar seu funcionamento durante a execução das transações. Esse método foi descartado devido ao uso de um processador com menor quantidade de núcleos utilizado no processo de desenvolvimento, o que levaria a um *overhead* adicional nesse caso.

As medições realizadas consideraram o uso de 8 *threads* em um processador com 8 *threads* lógicas (4 núcleos físicos). Processadores mais rápidos ou com configuração diferente podem requerer outros valores, já que os cálculos realizados consideram o tempo de processamento. A mesma observação pode ser realizada com o escalonador LUTS-Dynamic original, onde a escolha sobre qual heurística utilizar se baseia no tempo das transações.

Capítulo 4

Resultados experimentais

Este capítulo apresenta os resultados obtidos comparando o escalonador LUTS-Dynamic-Serializer com outros escalonadores utilizando o *benchmark* STAMP e a biblioteca de Memória Transacional em *Software* TinySTM. Como ponto de partida, são apresentadas as tecnologias envolvidas e configurações do ambiente de experimento.

4.1 TinySTM

Foi adotado neste trabalho o uso da biblioteca de Memória Transacional em *Sofware* TinySTM [17]. A biblioteca TinySTM é uma implementação de Memória Transacional em *Software* com granularidade baseada em palavras, que utiliza uma tabela *hash* contendo referências aos locais bloqueados na memória (tabela de *locks*). Também utiliza um relógio global para identificar a versão dos dados acessados. Para detecção de conflitos é usado a política *eager*, chamada de *encounter-time locking* (ETL). Nos experimentos é usado o versionamento diferido (modo padrão da biblioteca). O modo de resolução de conflitos padrão para gerenciamento de contenção utiliza a política SUICIDE, que reinicia uma transação imediatamente após encontrar um conflito.

Tabela 4.1: Aplicações presentes no *benchmark* STAMP

Aplicação	Descrição
Bayes	Algoritmo para aprendizado de uma estrutura de rede bayesiana
Genome	Recria um genoma a partir de cadeias de DNA
Intruder	Algoritmo para detectar invasão em redes a partir de reconhecimento de padrões
Kmeans	Agrupa dados objetos em um espaço N-dimensional em k clusters
Labyrinth	Encontra caminhos em um labirinto tridimensional
Ssca2	Conjunto de kernels/aplicações que operam em um multigrafo acíclico. O STAMP se baseia no primeiro dos kernels que cria representações de grafos
Vacation	Simula uma aplicação de reserva de passagens online
Yada	Algoritmo para refinamento de redes de Delaunay

Fonte: [43]

4.2 *Benchmark* STAMP

O STAMP (*Stanford Transactional Applications for Multiprocessing*) [43] é uma suíte de *benchmarks* composta por 8 aplicações escolhidas especificamente para cobrir certas características transacionais, como tempo em transação, nível de contenção e tamanho dos conjuntos de leitura/escrita As oito aplicações presentes na suíte são descritas na Tabela 4.1.

Criada para ser uma suíte de referência, o seu desenvolvimento se guiou em três princípios: largura, profundidade e portabilidade. Largura define que a suíte deve conter aplicações variadas que em sua maioria devem ser paralelizadas e que possam ser beneficiadas pela concorrência otimista dos sistemas de Memória Transacional.

A profundidade é explorada com a escolha de aplicações que apresentam transações de características diversas, que se diferenciam em níveis de contenção, tamanho de transações, além de conjuntos de leitura e escrita de tamanhos variados. Os autores destacam as diferenças entre

Tabela 4.2: Características qualitativas das aplicações do *benchmark* STAMP Aplicação Comprimento da transação

Aplicação	Tamanho da transação (instruções)	Conjunto de leitura e escrita	Tempo gasto em transações	Contenção
Bayes	longa	grande	alto	alta
Genome	média	médio	alto	baixa
Intruder	curta	médio	médio	alta
Kmeans	curta	pequeno	baixo	baixa
Labyrinth	longa	grande	alto	alta
Ssca2	curta	pequeno	baixo	baixa
Vacation	média	médio	alto	baixa/média
Yada	longa	grande	alto	média

Fonte: [43]

Tabela 4.3: Argumentos utilizados na execução das aplicações do *benchmark* STAMP

Aplicação	Argumento
Genome	-g16384 -s64 -n16777216
Intruder	-a10 -l128 -n262144 -s1
Kmeans	-m15 -n15 -T0.00001 -i random-n65536-d32-c16.txt
Labyrinth	-i random-x512-y512-z7-n512.txt
Ssca2	-s20 -i1.0 -u1.0 -l3 -p3
Vacation	-n4 -q60 -u90 -r1048576 -T4194304
Yada	-a15 -i ttimeu1000000.2

Fonte: Autor

cada aplicação que pode ser conferida na Tabela 4.2.

O STAMP foi criado para ser facilmente portável para diferentes ambientes e tipos de sistemas de Memória Transacional, sendo aplicado a sistemas de Memória Transacional em *Software*, *Hardware* e híbrido. Para os resultados mostrados neste capítulo, foram utilizados os argumentos presentes na Tabela 4.3.

Devido ao comportamento não reproduzível da aplicação Bayes já observada anteriormente por outros autores [49][12], essa aplicação também será omitida dos testes com o *benchmark* STAMP.

4.3 Configuração do ambiente de experimento

Com o objetivo de avaliar diferentes cenários entre escalonadores de diferentes tipos, os testes foram realizados utilizando a configuração de *hardware* descrita na Tabela 4.4. O sistema operacional utilizado nos experimentos é um CentOS, com *Kernel* 2.6.32 e o compilador é o GCC (*GNU Compiler Collection*) versão 4.47. Os aplicativos do *benchmark* STAMP foram compilados utilizando a biblioteca de Memória Transacional em *Software* TinySTM [17], juntamente com os escalonadores ATS, LUTS-Dynamic e LUTS-Dynamic-Serialyzer. Também foram analisados os dados do escalonador Shrink, porém seus resultados estão sendo omitidos por não apresentarem nenhum comportamento diferente dos demais.

Tabela 4.4: Configuração de *Hardware* da plataforma de testes

Componente	**Configuração**
Processador	Intel Core I7 2600k, *cache* nível 3 8MB, *HyperThreading* (4 núcleos físicos + 8 núcleos lógicos) arquitetura Sandy Bridge
Memória	8GB ram

Fonte: Autor

A configuração dos escalonadores utilizados e gerenciador de contenção padrão são dados a seguir:

- ATS: Seguindo trabalho anterior [49], os valores de α e de limiar foram definidos como 0,75 e 0,7 respectivamente;
- LUTS-Dynamic: O escalonador utiliza cem mil ciclos para identificar qual heurística utilizar (transações longas ou curtas). No caso de transações curtas o limiar é definido em 0,5 e o valor α é igual a 0,75.

- LUTS-Dynamic-Serializer: A proposta dessse trabalho de mestrado, utilizando a fórmula 4.1.

$$spinMaximo = 175.5 \times log(indice) - 993 \tag{4.1}$$

Nessa última fórmula, *indice* é a divisão do tempo médio das transações pelo número de *aborts* para o cálculo do número máximo de spins antes de mudar para o modo de *mutex*. Utiliza também o limiar de 90 para fazer a transição novamente para *spinlock*. Para medir o consumo de energia foi utilizada a infraestrutura apresentada no Capítulo 2, Seção 2.3.1.

Cada aplicação foi executada 10 vezes com os escalonadores apresentados anteriormente, utilizando-se 4, 8 e 16 *threads*, assumindo-se um intervalo de confiança de 95%. Essas configurações de *threads* foram escolhidas considerando os recursos do processador utilizado, que possui 4 núcleos físicos e 8 núcleos lógicos (*Hyperthreading*). Todos os testes foram realizados considerando-se o *governor ondemand*.

4.4 Análise dos resultados

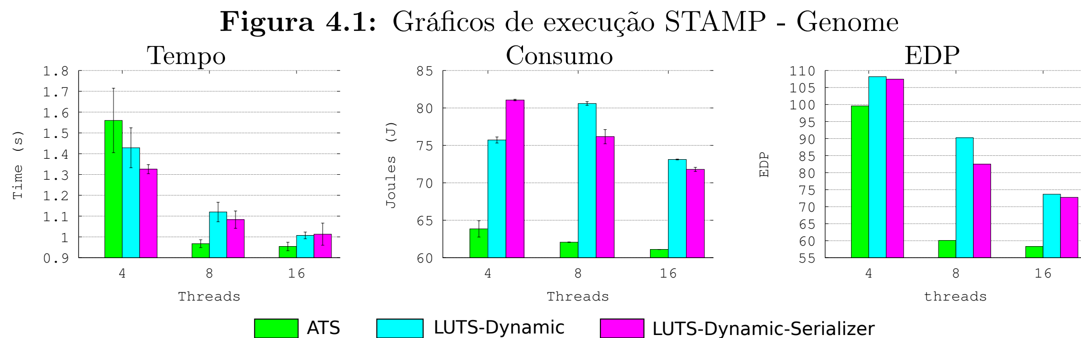
Nesta seção é feita a análise dos resultados obtidos. As Figuras de 4.1 até 4.7 apresentam, para cada aplicação do STAMP, gráficos para tempo total de execução (em segundos), consumo de energia (em Joules) e o EDP resultante. Note que, para todos os gráficos, quanto menor o valor melhor o resultado.

Inicialmente é analisado o comportamento de cada aplicação, como descrito a seguir.

Genome(Figura 4.1): Nessa aplicação tem-se o escalonador ATS apresentando melhores tempos de execução e consumo de energia

e, consequentemente, o melhor EDP para as três configurações de *threads*. Os escalonadores LUTS-Dynamic e LUTS-Dynamic-Serializer obtiveram resultados bem similares, considerando-se as margens de erro, com pequena vantagem ao último no cálculo do EDP. Obvervou-se que para essa aplicação em particular, a infraestrutura do LUTS (escalonador) foi responsável pela perda de desempenho e aumento de energia obtidos, visto que o tempo de execução é muito baixo para amortizar o custo do escalonamento.

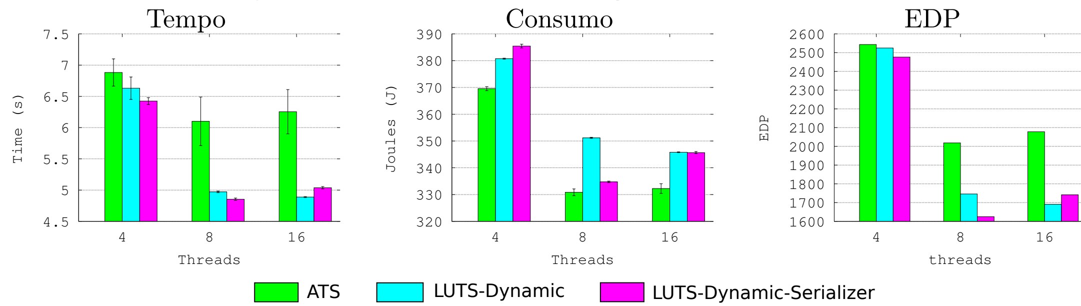
Figura 4.1: Gráficos de execução STAMP - Genome



Fonte: Autor

Intruder(Figura 4.2): Na configuração de 8 *threads* tem-se que o escalonador ATS teve tempo de execução alto e baixo consumo de energia, sendo que o contrário aconteceu para o LUTS-Dynamic, enquanto que o LUTS-Dynamic-Serializer apresentou melhores valores nesses dois aspectos o que resultou em um melhor EDP entre os três escalonadores. Já com 16 threads o escalonador LUTS-Dynamic teve melhor EDP.

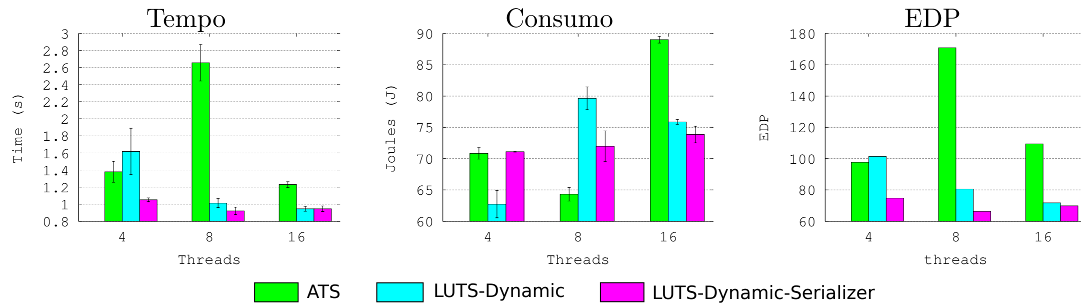
Figura 4.2: Gráficos de execução STAMP - Intruder



Fonte: Autor

Kmeans(Figura 4.3): Já a aplicação Kmeans teve comportamento parecido ao da aplicação Intruder, com o LUTS-Dynamic-Serilizer apresentando o melhor EDP para as três configurações de *threads*.

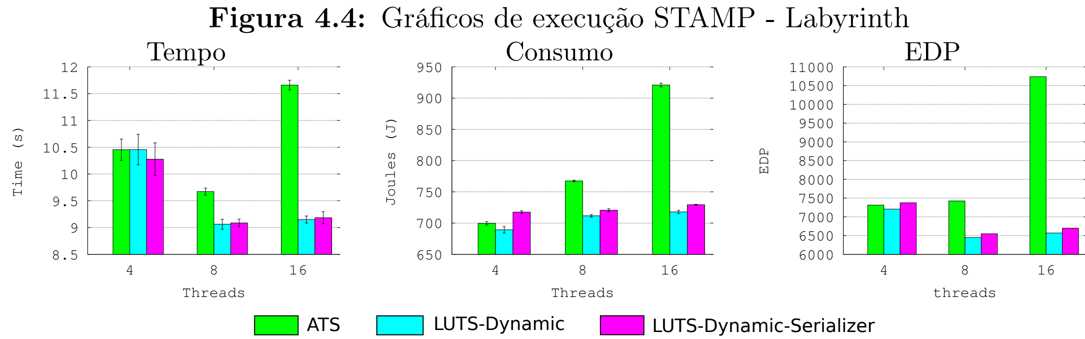
Figura 4.3: Gráficos de execução STAMP - Kmeans



Fonte: Autor

Labyrinth(Figura 4.4): O LUTS-Dynamic-Serializer seguiu próximo (mas com pior EDP) ao LUTS-Dynamic original que no caso desta aplicação tende a usar a heurística hash mais complexa. O escalonador ATS teve pior desempenho nessa aplicação, mantendo EDP parecido com 4 e 8 *threads*, enquanto o EDP aumentou em torno de 70% para 16 *threads*.

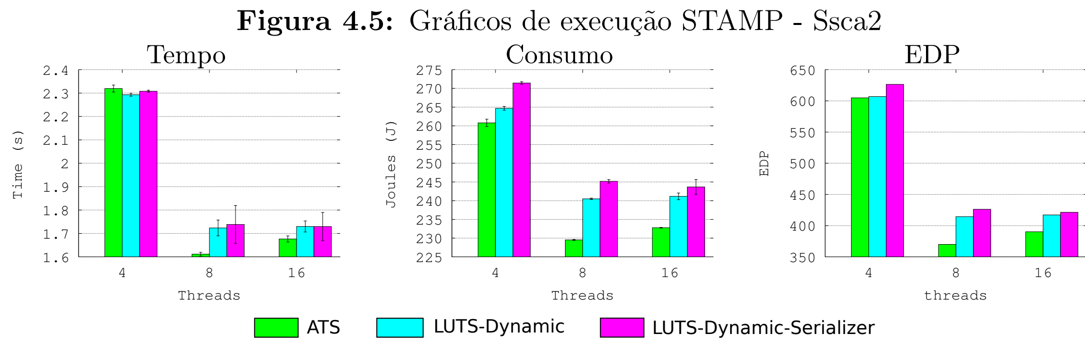
Figura 4.4: Gráficos de execução STAMP - Labyrinth



Fonte: Autor

SSCA2(Figura 4.5): Para essa aplicação o escalonador ATS teve os melhores valores em todos os casos observados, enquanto o LUTS-Dynamic se manteve ligeiramente melhor que a heurística aqui apresentada.

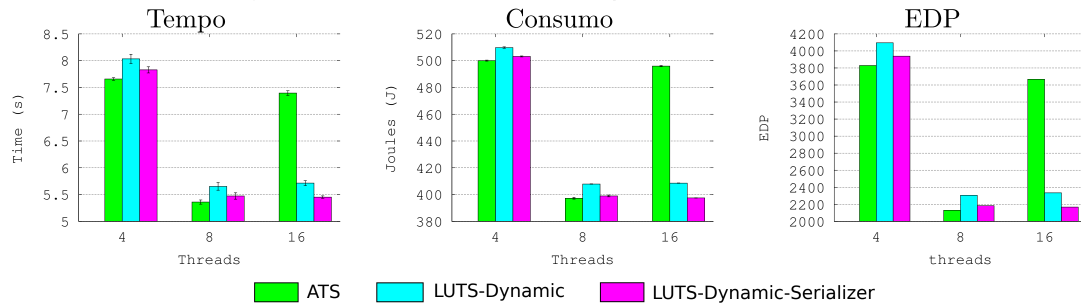
Figura 4.5: Gráficos de execução STAMP - Ssca2



Fonte: Autor

Vacation(Figura 4.6): A heurística proposta (LUTS-Dynamic-Serializer) teve desempenho similar ao escalonador ATS quando considerado os valores de EDP para 8 *threads*. Com 16 *threads*, o escalonador ATS tem EDP elevado, enquanto o LUTS-Dynamic-Serializer mantém o seu EDP próximo aos valores com 8 *threads* devido ao controle de *threads* da estrutura do LUTS que dispara somente o número máximo de *threads* que o processador suporta.

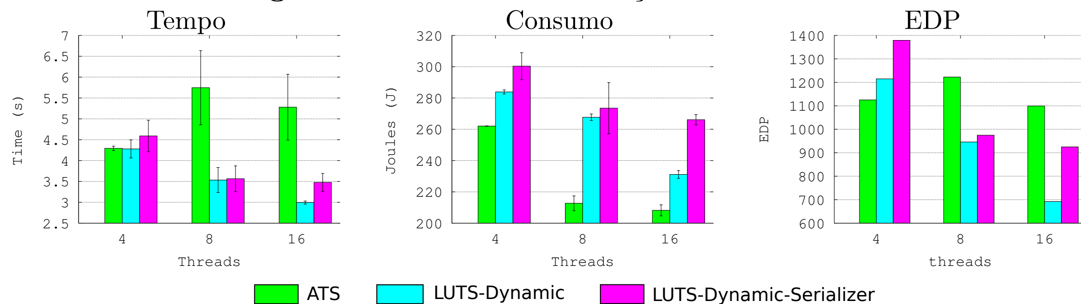
Figura 4.6: Gráficos de execução STAMP - Vacation



Fonte: Autor

Yada(Figura 4.7): O escalonador LUTS-Dynamic apresentou os melhores valores de EDP para 8 e 16 *threads*. O escalonador ATS teve para 8 e 16 *threads* tempo de execução alto e os menores valores de consumo de energia, sendo que o inverso ocorreu em partes com o LUTS-Dynamic e LUTS-Dynamic-Serializer. O LUTS-Dynamic apresentou o melhor EDP entre os três escalonadores com a configuração de 16 *threads*.

Figura 4.7: Gráficos de execução STAMP - Yada



Fonte: Autor

A Tabela 4.5 faz um resumo dos ganhos obtidos com a aplicação da nova heurística para as sete aplicações do STAMP utilizando 8 *threads*, que é a contagem de núcleos lógicos no processador utilizado. Esse comparativo é destacado aqui por corresponder a quantidade de *threads* utilizada na análise do comportamento das transações durante

a criação e refinamento dos valores utilizados no cálculo de spin máximo pela heurística. Valores positivos indicam um ganho da heurística apresentada em comparação com uma outra abordagem, enquanto valores negativos indicam uma perda.

Tabela 4.5: Comparativo EDP para as aplicações STAMP

	8 *threads*		16 *threads*	
Aplicação	**LUTS-Dynamic**	**ATS**	**LUTS-Dynamic**	**ATS**
Genome	8,61%	-37,33%	1,24%	-24,79%
Intruder	6,94%	19,49%	-3,01%	16,20%
Kmeans	17,76%	61,21%	2,63%	36,17%
Labyrinth	-1,53%	11,79%	-1,97%	37,62%
SSCA2	-2,83%	-15,22%	-1,02%	-7,99%
Vacation	5,28%	-2,56%	7,17%	40,91%
Yada	-3,06%	20,26%	-33,66%	15,84%

Fonte: Autor

É possível observar que tivemos ganhos no EDP de cerca de 17% na aplicação Kmeans se comparado ao LUTS-Dynamic e de cerca de 61% se comparado ao ATS. Ganhos menores foram obtidos nas aplicações Genome, Intruder e Vacation em comparação ao LUTS-Dynamic, e nas aplicações Intruder, Labyrinth e Yada em comparação ao ATS. Houve leve piora no EDP para as aplicações Labyrinth, SSCA2 e Yada em comparação ao LUTS-Dynamic, e leve piora para as aplicações Genome, Ssca2 e Vacation. Embora o EDP para o Genome tenha sido pior em relação ao ATS (em torno de 37%), a heurística proposta se manteve melhor que o LUTS-Dynamic nesse caso, conforme pode ser visto na Figura 4.1.

Se levarmos em consideração o uso de 16 *threads*, a heurística proposta apresenta pouca vantagem em comparação ao LUTS-Dynamic, ganhando em EDP nas aplicações Genome, Kmeans e Vacation, perdendo em EDP nas outras aplicações com destaque para a aplicação Yada, onde o LUTS-Dynamic teve EDP 33% menor. Em comparação

ao ATS, a heurística proposta apresenta os melhores valores de EDP na maioria dos casos, só perdendo em EDP nas aplicações Genome e SSCA2.

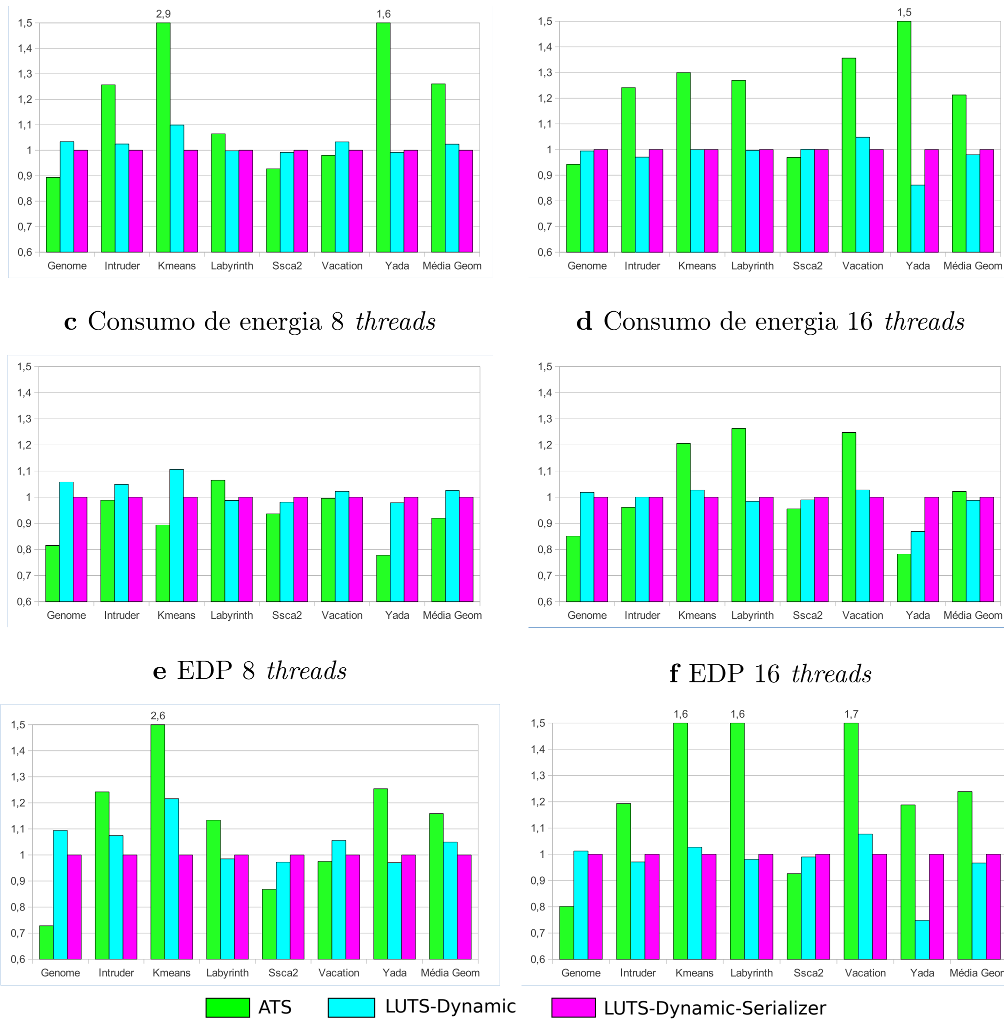
Na Figura 4.8 são apresentados os gráficos com os valores obtidos de tempo de execução, consumo de energia e EDP normalizados em relação à heurística proposta. Valores acima de 1 indicam piores resultados em comparação com o LUTS-Dynamic-Serializer, enquanto valores menores que 1 indicam melhorias. O último grupo de dados em cada gráfico indica a média geométrica entre todas as aplicações apresentadas.

Analisando os gráficos para 8 threads, principalmente o último item que indica a média geométrica, podemos observar que o escalonador ATS teve o maior tempo de execução e o menor consumo de energia na média, porém obteve EDP 15,8% maior em comparação ao LUTS-Dynamic-Serializer. O LUTS-Dynamic obteve tempo e consumo levemente maior que a nossa heurística, gerando um EDP 4,9% maior.

Analisando as execuções com 16 threads no gráfico normalizado, podemos observar que o LUTS-Dynamic foi melhor em tempo de execução (Figura 4.8b) e em consumo de energia (Figura 4.8d) na maioria dos casos, gerando um EDP menor em 3,3% quando observamos a média geométrica. Já o escalonador ATS teve maior tempo de execução, e consumo próximo ao LUTS-Dynamic-Serializer na média geométrica, gerando um EDP 23% maior.

Figura 4.8: Gráficos normalizados com LUTS-Dynamic-Serializer

a Tempo de execução 8 *threads* **b** Tempo de execução 16 *threads*



Fonte: Autor

4.5 Discussão

O escalonador ATS teve melhor EDP quando aplicado às aplicações Genome e SSCA2 que tem transações de tamanho médio/curto e pouca contenção, enquanto a heurística aqui apresentada teve melhor EDP quando aplicada às aplicações Intruder e Kmeans as quais possuem tamanhos de transação curto e contenção baixa (é indicado na Tabela 4.2

Tabela 4.6: Características das aplicações STAMP e melhores escalonadores encontrados para execução com 8 *threads*

Aplicação	**Instruções**	**Conjunto leitura - escrita**	**Tempo transa-ções**	**Contenção**	**Escalonador**
Genome	média	médio	alto	baixa	ATS
Intruder	curta	médio	médio	alta	LUTS-DS
Kmeans	curta	pequeno	baixo	baixa	LUTS-DS
Labyrinth	longa	grande	alto	alta	LUTS
Ssca2	curta	pequeno	baixo	baixa	ATS
Vacation	média	médio	alto	baixa/média	ATS
Yada	longa	grande	alto	média	LUTS

Fonte: Autor e [43]

que a aplicação Intruder possui alta contenção, mas isso somente ocorre nos estágios finais de sua execução). Isso aconteceu principalmente porque estas aplicações (Intruder e Kmeans) se previlegiaram melhor da fila com *spinlock* por terem transações pequenas, e pelo tempo de *spinlock* ser suficientemente pequeno em comparação ao tamanho das transações, sendo que raramente elas se alternam para *mutex*.

O mesmo não aconteceu no caso do Genome e SSCA2. Como estas aplicações têm tempo de transação maior, elas gastaram mais energia com o *spinlock* antes de alternar para *mutex* do que o necessário. É importante notar que o escalonador ATS utiliza *mutex* para serializar e o LUTS-Dynamic não serializa as transações. A heurística apresentada aqui ainda foi melhor que o LUTS-Dynamic para a aplicação Vacation, que possui tamanho de transação média e média contenção. Já o escalonador LUTS-Dynamic teve melhor EDP para aplicações de transações longas como Labyrinth e Yada, principalmente pelo uso de sua heurística para transações longas.

A Tabela 4.6 identifica qual escalonador foi melhor para cada aplicação considerando-se o uso de 8 threads. Foi feita uma junção

entre a Tabela 4.2 e os resultados obtidos facilitando a caracterização dos escalonadores, mostrando para cada configuração o escalonador mais indicado a ser aplicado.

Dessa forma, uma melhor análise sobre o comportamento das transações e suas diferentes fases, incluindo um refinamento da equação para identificar todos os casos, se faz necessário. Além disso, a heurística de hash do LUTS-Dynamic poderia ser integrado ao LUTS-Dynamic-Serializer de modo a obter-se melhor desempenho em transações longas.

Capítulo 5

Conclusões

Essa dissertação apresentou um levantamento de trabalhos anteriores relacionados a sistemas de Memória Transacional que abordaram o consumo de energia. Através da análise desse consumo nos escalonadores ATS e LUTS-Dynamic, foi proposta uma nova heurística utilizando o escalonador LUTS como base, objetivando-se a redução do consumo de energia. A heurística criada, denominada LUTS-Dynamic-Serializer, alterna de forma dinâmica entre duas formas de serialização (*spinlock* e *mutex*), utilizando o tempo médio das transações e a quantidade de aborts para escolha do melhor modo. A abordagem desenvolvida aqui assemelha-se com a técnica conhecida como Green-CM, porém esta foi aplicada a gerenciadores de contenção.

Considerando-se a média geométrica entre as aplicações do benchmark STAMP avaliadas, o LUTS-Dynamic-Serializer obteve menor EDP que o LUTS e que o ATS em 4,95% e 15,8% respectivamente com a configuração de 8 threads, enquanto foi menos eficiente que o escalonador LUTS em 3,3% e mais eficiente que o escalonador ATS em 23% com a configuração de 16 threads.

Levando em conta que o processador utilizado nos testes possuía a limitação de 8 *threads*, esse trabalho dá margem de exploração para melhoria de EDP também considerando casos de falso paralelismo,

enquanto que com o uso de 8 *threads* tivemos resultados satisfatórios que favorecem o uso da heurística na maioria das aplicações estudadas.

5.1 Trabalhos futuros

Como trabalhos futuros podem ser explorados os seguintes tópicos:

- Refinamento da heurística com informações de consumo de energia;
- Uso de técnicas de aprendizado de máquina para melhor acompanhamento das variáveis e melhor definição dos valores de spin máximos;
- Fazer um estudo em ambiente heterogêneo com processador multinúcleos e APU integrada.

Bibliografia

[1] Intel turbo boost technology 2.0. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html. Accessed: 2016-02-29.

[2] Intel® turbo boost technology frequency table. http://www.intel.com/content/www/us/en/support/processors/000005523.html. Accessed: 2016-02-29.

[3] Legacy __*sync* built-in functions for atomic memory access. http://https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html. Accessed: 2016-02-29.

[4] K. Accardi. Balancing power and performance in the linux kernel. Em *LinuxCon Europe 2015*. Linux Foundation, 2015.

[5] M. Alioto, E. Consoli e G. Palumbo. Metrics and design considerations on the energy-delay tradeoff of digital circuits. Em *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pp. 3150–3153. IEEE, 2009.

[6] A. Baldassin, J. de Carvalho e R. Azevedo. Reavaliando a eficiência energética de memória transacional em processadores convencionais. *Anais do XIV Simpósio em Sistemas Computacionais (WSCAD-SSC)*, pp. 69–76, 2013.

[7] A. Baldassin, J. P. De Carvalho, L. A. Garcia e R. Azevedo. Energy-performance tradeoffs in software transactional memory. Em *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pp. 147–154. IEEE, 2012.

[8] A. Baldassin, F. Klein, G. Araujo, R. Azevedo e P. Centoducatte. Characterizing the energy consumption of software transactional memory. *Computer Architecture Letters*, 8(2):56–59, 2009.

[9] L. A. Barroso e U. Hölzle. The case for energy-proportional computing. *Computer*, (12):33–37, 2007.

[10] I. I. P.-s. A. Binary. Interface (abi). *Intel Corporation, May*, 2001.

[11] M. Campbell. Intel to bring back fivr after skylake and kaby lake cpus. `http://www.overclock3d.net/articles/cpu_mainboard/intel_to_bring_back_fivr_after_skylake_and_kaby_lake_cpus/1`, 2015. Accessed: 2016-02-29.

[12] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier et alli. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. Em *Proceedings of the 5th European conference on Computer systems*, pp. 27–40. ACM, 2010.

[13] J. Demmel e A. Gearhart. Instrumenting linear algebra energy consumption via on-chip energy counters. *UC at Berkeley, Tech. Rep. UCB/EECS-2012-168*, 2012.

[14] D. Dice, O. Shalev e N. Shavit. Transactional locking ii. Em *Distributed Computing*, pp. 194–208. Springer, 2006.

[15] S. Dolev, D. Hendler e A. Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. Em *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pp. 125–134. ACM, 2008.

[16] A. Dragojević, R. Guerraoui, A. V. Singh e V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. Em *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pp. 7–16. ACM, 2009.

[17] P. Felber, C. Fetzer e T. Riegel. Dynamic performance tuning of word-based software transactional memory. Em *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 237–246. ACM, 2008.

[18] C. Ferri, A. Viescas, T. Moreshet, R. Bahar e M. Herlihy. Energy efficient synchronization techniques for embedded architectures. Em *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pp. 435–440. ACM, 2008.

[19] E. Gaona-Ramírez, R. Titos-Gil, J. Fernández e M. E. Acacio. Characterizing energy consumption in hardware transactional memory systems. Em *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pp. 9–16. IEEE, 2010.

[20] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran e K. Veezhinathan. The implications of shared data synchronization techniques on multi-core energy efficiency. Em *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*, 2012.

[21] R. Guerraoui, M. Kapalka e J. Vitek. Stmbench7: a benchmark for software transactional memory. Relatório técnico, 2006.

[22] P. Guide. Intel® 64 and ia-32 architectures software developer's manual, 2011.

[23] D. Hackenberg, R. Schone, T. Ilsche, D. Molka, J. Schuchart e R. Geyer. An energy efficiency feature survey of the intel haswell processor. Em *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 896–904. IEEE, 2015.

[24] M. Hähnel, B. Döbel, M. Völp e H. Härtig. Measuring energy consumption for short code paths using rapl. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[25] T. Harris, A. Cristal, O. S. Unsal, E. Ayguade, F. Gagliardi, B. Smith e M. Valero. Transactional memory: An overview. *IEEE micro*, (3):8–29, 2007.

[26] T. Harris, J. Larus e R. Rajwar. *Transactional memory*, volume 5. Morgan & Claypool Publishers, 2010.

[27] M. Herlihy e J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. pp. 289–300, junho de 1993.

[28] M. Herlihy e N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2008.

[29] J. Hopper et alli. Using the linux cpufreq subsystem for energy management. *IBM blueprints*, 2009.

[30] M. Horowitz, T. Indermaur e R. Gonzalez. Low-power digital design. Em *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pp. 8–11. IEEE, 1994.

[31] Intel. Desktop 4th gen intel® core™ processors datasheet, 2015.

[32] Intel. Desktop 5th gen intel® core™ processors datasheet, 2015.

[33] S. Issa, P. Romano e M. Brorsson. Green-cm: Energy efficient contention management for transactional memory. Em *Parallel Processing (ICPP), 2015 44th International Conference on*, pp. 550–559. IEEE, 2015.

[34] A. M. Junior e A. J. Baldassin. Primeiros estudos sobre consumo de energia em escalonadores de transações. Em *VI Escola Regional de Alto Desempenho de São Paulo*. CEACPAD-SBC, 2015.

[35] L. B. Kish. End of moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3):144–149, 2002.

[36] F. Klein, A. Baldassin, G. Araujo, P. Centoducatte e R. Azevedo. On the energy-efficiency of software transactional memory. Em *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*, p. 33. ACM, 2009.

[37] M. Larabel. Intel publishes new linux p-state driver. `http://www.phoronix.com/scan.php?page=news_item&px=MTI5Mzc`, 2013. Accessed: 2016-02-29.

[38] J. Li, J. F. Martinez e M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. Em *Software, IEE Proceedings-*, pp. 14–23. IEEE, 2004.

[39] D. Lo e C. Kozyrakis. Dynamic management of turbomode in modern multi-core chips. Em *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 603–613. IEEE, 2014.

[40] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall e G. Muller. Scheduling support for transactional memory contention management. Em *ACM Sigplan Notices*, volume 45, pp. 79–90. ACM, 2010.

[41] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III e M. L. Scott. Lowering the overhead of nonblocking software transactional memory. Em *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[42] A. J. Martin. Towards an energy complexity of computation. *Information Processing Letters*, 77(2):181–187, 2001.

[43] C. C. Minh, J. Chung, C. Kozyrakis e K. Olukotun. Stamp: Stanford transactional applications for multi-processing. Em *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 35–46. IEEE, 2008.

[44] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp. 114 ff. *IEEE Solid-State Circuits Newsletter*, 3(20):33–35, 2006.

[45] T. Moreshet, R. Bahar e M. Herlihy. Energy reduction in multiprocessor systems using transactional memory. Em *Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on*, pp. 331–334. IEEE, 2005.

[46] T. Moreshet, R. I. Bahar e M. Herlihy. Energy-aware microprocessor synchronization: Transactional memory vs. locks. Em *Fourth Annual Boston-Area Architecture Workshop*, p. 21. Citeseer, 2006.

[47] A. Naveh, D. Rajwan, A. Ananthakrishnan e E. Weissmann. Power management architecture of the 2nd generation intel® core™ microarchitecture, formerly codenamed sandy bridge. Em *Hot Chips*, volume 23, p. 0, 2011.

[48] D. Nicácio, A. Baldassin e G. Araújo. Luts: a weight user-level transaction scheduler. Em *Algorithms and Architectures for Parallel Processing*, pp. 144–157. Springer, 2011.

[49] D. Nicácio, A. Baldassin e G. Araújo. Transaction scheduling using dynamic conflict avoidance. *International Journal of Parallel Programming*, 41(1):89–110, 2013.

[50] K. Olukotun e L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[51] T. L. K. Organization. Intel p-state driver. `https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt`, 2015. Accessed: 2016-02-29.

[52] T. M. Rico, M. L. Pilla e A. R. Du Bois. Energy consumption on software transactional memories. Em *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*, pp. 194–201. IEEE, 2012.

[53] D. Rughetti, P. Di Sanzo e A. Pellegrini. Adaptive transactional memories: Performance and energy consumption tradeoffs. Em *Network Cloud Computing and Applications (NCCA), 2014 IEEE 3rd Symposium on*, pp. 105–112. IEEE, 2014.

[54] N. Shavit e D. Touitou. Software transactional memory. Em *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. PODC '95, 1995.

[55] M. F. Spear. Lightweight, robust adaptivity for software transactional memory. Em *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pp. 273–283. ACM, 2010.

[56] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber e D. Dice. The turbo diaries: Application-controlled frequency scaling explained. Em *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 193–204, 2014.

[57] X. Wang, Z. Ji, C. Fu, M. Hu e X. Yang. Software transactional memory in multicore processors. Em *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on*, pp. 1–4. IEEE, 2009.

[58] R. M. Yoo e H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. Em *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pp. 169–178. ACM, 2008.

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

TERMO DE REPRODUÇÃO XEROGRÁFICA

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

São José do Rio Preto, 10 de Maio de 2016

Assinatura do autor