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

Dissertação

Análise de Consumo de Energia e Desempenho de Memórias Transacionais em *Software* em Ambiente de Computação Real

TIMÓTEO MATTHIES RICO

Pelotas, 2013

TIMÓTEO MATTHIES RICO

Análise de Consumo de Energia e Desempenho de Memórias Transacionais em *Software* em Ambiente de Computação Real

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal de Pelotas, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. Maurício Lima Pilla
Coorientador: Prof. Dr. André Rauber Du Bois

Pelotas, 2013

Dados de catalogação na fonte:
Leda Cristina Peres Lopes – CRB: 10/2064
Universidade Federal de Pelotas

R487a Rico, Timóteo Matthies

Análise de Consumo de Energia e Desempenho de Memórias Transacionais em *Software* em Ambiente de Computação Real / Timóteo Matthies Rico. – Pelotas, 2013. – 83 f: gráf. – Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal de Pelotas. Centro de Desenvolvimento Tecnológico. Pelotas, 2013. – Orientador Maurício Lima Pilla; Coorientador André Rauber Du Bois.

1. Programação Concorrente. 2. Memória Transacional em *Software*. 3. Consumo de Energia. I. Pilla, Maurício Lima. II. Du Bois, André Rauber. III. Título.

CDD: 005.43

Banca examinadora:

Prof. Dr. Benhur de Oliveira Stein - UFSM

Prof. Dr. Júlio Carlos Balzano de Mattos - UFPEL

Prof. Dr. Gerson Geraldo Homrich Cavalheiro - UFPEL

Dedico este trabalho à minha família.

AGRADECIMENTOS

A Deus pela força, determinação, segurança, e aprendizado constante. Sem Ele, nada disso seria possível.

Muitas foram às pessoas que diretamente e indiretamente me acompanharam e incentivaram durante esse tempo de estudo. Citar todas aqui não seria possível, porém algumas tiveram um diferencial para que eu pudesse chegar ao fim dessa etapa. A todas elas, minha imensa gratidão.

Aos meus pais José Antônio Rico e Liamar Matthies Rico, pela educação, segurança e incentivo.

A meu orientador Prof. Dr. Maurício Lima Pilla e ao meu coorientador Prof. Dr. André Rauber Du Bois, pela disponibilidade, colaboração, conhecimentos transmitidos e incentivo.

Aos colegas do Laboratório de Sistemas Paralelos e Ubíquos da UFPel, em especial ao colega e amigo Rodrigo Medeiros Duarte, pela disponibilidade, colaboração e ajuda.

Aos professores e funcionários do Programa de Pós-Graduação em Computação da UFPel que me acompanharam em diversos momentos no decorrer do curso. Em especial, ao Prof. Dr. Gerson Geraldo Homrich Cavalheiro e ao Prof. Dr. Adenauer Corrêa Yamin, pelas palavras de incentivo.

Aos colegas do curso de Pós-Graduação em Computação, pela convivência e troca de experiências.

À Universidade Federal de Pelotas e ao curso de Pós-Graduação em Computação.

Milhões viram a maçã cair, mas só Newton perguntou por quê.
— BERNARD M. BARUCH

RESUMO

RICO, Timóteo Matthies. **Análise de Consumo de Energia e Desempenho de Memórias Transacionais em *Software* em Ambiente de Computação Real**. 2013. 83 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal de Pelotas, Pelotas.

Com o advento de arquiteturas multiprocessadas novos desafios foram introduzidos ao desenvolvimento de *software*. Dentre estes desafios, realizar a sincronização necessária para evitar condições de corrida entre os fluxos de execução, é uma das principais dificuldades na programação concorrente.

Um novo mecanismo de sincronização, denominado Memória Transacional, tem sido desenvolvido por pesquisadores de programação concorrente com o objetivo de reduzir as dificuldades e limitações encontradas em mecanismos de sincronização tradicionais. Por se tratar de uma alternativa recentemente proposta, pouco se conhece a respeito do consumo de energia de Memórias Transacionais, em especial de implementações em *software*.

Nesse contexto, o presente trabalho apresenta a análise e caracterização do consumo de energia e desempenho de quatro importantes bibliotecas de Memória Transacional em *Software* (STM), TL2, TinySTM, SwissTM e AdaptSTM, utilizando-se o *benchmark* STAMP. Diferente de outros trabalhos, as execuções não foram simuladas mas executadas em um ambiente de computação real.

Resultados obtidos mostram a SwissTM como a biblioteca mais eficiente em termos de consumo de energia e desempenho, seguida pela AdaptSTM, TinySTM e TL2, na maioria dos cenários de execução utilizando-se até 8 *threads*. Constata-se que a escalabilidade das STMs utilizadas está relacionada diretamente à particularidade das estratégias de detecção e resolução de conflitos empregada por cada biblioteca.

Nesta perspectiva, verifica-se que em aplicações com transações curtas a AdaptSTM mostra-se a biblioteca mais eficiente. Em aplicações com transações médias, a SwissTM apresenta a melhor escalabilidade. Em cenários com longas transações e sob média/alta contenção a TL2 apresenta os melhores resultados. A TinySTM, por sua vez, mostra-se a biblioteca menos eficiente em termos de escalabilidade na maioria dos cenários, exibindo bons resultados somente em aplicações que apresentem mínimas taxas de cancelamentos.

Palavras-chave: Programação Concorrente, Memória Transacional em *Software*, Consumo de Energia.

ABSTRACT

RICO, Timóteo Matthies. **Energy Consumption and Performance of Software Transactional Memories in a Real Computing Environment**. 2013. 83 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal de Pelotas, Pelotas.

With the advent of multicore architectures, new challenges to software development were raised. Among those, one of the main issues in concurrent programming is related to the synchronization required to avoid race conditions.

Transactional Memories have been developed by concurrent programming researchers in order to reduce dificulties and limitations found in traditional synchronization mechanisms. As it is a more recent approach to synchronization, little is known about energy consumption of Transactional Memories, in special of software implementations.

In this context, this work presents the analysis and characterization of energy consumption and performance of four important Transactional Memory libraries: TL2, TinySTM, SwissTM, and AdaptSTM, using the STAMP benchmark. A differential to other works is that results were obtained for a real computational environment and not simulated.

Results show that SwissTM is the most efficient library of the four in terms of energy consumption and performance, followed by AdapSTM, TinySTM, and TL2 in this order, for most of the execution scenarios and 8 threads at most. STM's scalability is directly tied to the strategies for detection and resolution of conflicts. In this perspective, AdaptSTM is the best STM for applications with short transactions. SwissTM presents the best results for medium transactions. Long transactions and medium/high contention are best handled by TL2. On the other hand, TinySTM shows the worst scalability for most scenarios, with good results only for applications with very small abort rates.

Keywords: Concurrent Programming, Software Transaction Memories, Energy Consumption.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade Consistência Isolamento Durabilidade
AMBA AHB	*Advanced Microcontroller Bus Architecture - Advanced High-performance Bus*
BMC	*Baseboard Management Controller*
DVFS	*Dynamic Voltage and Frequency Scaling*
EDP	*Energy-Delay Product*
HTM	*Hardware Transactional Memory*
HyTM	*Hybrid Transactional Memory*
IPMI	*Intelligent Platform Management Interface*
JPEG	*Joint Photographic Experts Group*
LAN	*Local Area Network*
LSA	*Lazy Snapshot Algorithm*
MPSOC	*Multiprocessor System on Chip*
NoC	*Network on Chip*
SGI	*Silicon Graphics International*
SIMPLE	*Simple Multiprocessor Platform Environment*
SRAM	*Static Random Access Memory*
SSCA2	*Scalable Synthetic Compact Applications 2*
STAMP	Stanford *Transactional Applications for Multi-Processing*
STM	*Software Transactional Memory*
TCP/IP	*Transmission Control Protocol / Internet Protocol*
TL2	*Transactional Locking* 2
TM	*Transactional Memory*
W	Watt

SUMÁRIO

1 INTRODUÇÃO

A utilização de sistemas computacionais faz-se cada vez mais presente em atividades das mais diferentes naturezas. Diversas áreas do conhecimento utilizam-se de aplicações que oferecem suporte automatizado para diversas tarefas, cujo suporte executivo tanto baseia-se em *hardware* como *software*. Nos lares, cresce de igual modo a utilização de dispositivos móveis e computadores pessoais, que servem, fundamentalmente, como equipamentos para lazer, comunicação e trabalho. A computação, por fazer-se cada vez mais presente na sociedade, torna-se um elemento catalisador do crescimento desta mesma, sendo os computadores representantes de uma considerável fatia do consumo de energia gerada no globo.

O crescimento contínuo das aplicações na Internet está conduzindo ao rápido crescimento dos *data centers*, sendo o consumo de energia desses predominantemente devido a dois fatores: servidores e refrigeração (MURUGESAN, 2008; LAMB, 2009). Indicou-se através de um estudo que há um crescimento em média de 20% ao ano nos custos para funcionamento de *data centers*. Nos Estados Unidos, o custo de consumo de *data centers* dobrou do ano de 2000 para 2006, para $4,5 bilhões (HAMM, 2008). Portanto, a busca pela redução e eficiência energética tornou-se uma estratégia fundamental em questão, visto que a emissão dos gases que poluem a atmosfera do planeta e aumentam o efeito estufa são auxiliados pela matriz energética mundial (FORREST; KAPLAN; KINDLER, 2008; CARDOSO, 2006; REIS; CUNHA, 2006).

Com enfoque no aquecimento global, pesquisadores de computação têm tomado consciência sobre o consumo de energia em sistemas computacionais e os impactos causados por esses no meio ambiente, gerando então uma área de pesquisa ainda recente, a Computação Sustentável (*Green Computing*) (KURP, 2008). O principal objetivo é prover um conjunto de práticas voltadas para a computação de modo a minimizar o impacto ambiental. O meio ambiente beneficia-se da computação sustentável por meio do consumo racional e eficiente de energia, menor uso de materiais nocivos na fabricação de computadores e através do

incentivo na reutilização e reciclagem de componentes de *hardware* (MURUGESAN, 2008). Tais práticas fomentam melhorias nas atividades de empresas e clientes, efetivando reduções em custos e contribuem com maior preservação ambiental.

O aumento da eficiência energética é importante em todo o contexto computacional, independente do *hardware* e *software* utilizado. Com o advento de arquiteturas *multicore*, a utilização de programação paralela no desenvolvimento de *software* é fundamental na melhoria da eficiência, por fazer melhor proveito dos recursos de *hardware* disponíveis e assim aumentar o desempenho das aplicações (MÓR et al., 2010).

Para que execuções concorrentes possam cooperar na realização de trabalho útil, há necessidade dessas se comunicarem. Um modelo de comunicação muito utilizado no desenvolvimento de *software* paralelo é o baseado em memória compartilhada. Neste modelo, os diferentes fluxos de execução (*threads*) possuem endereços de memória que são mapeados no mesmo espaço de endereçamento, o que torna os valores neles armazenados visíveis a todas as *threads* pertencentes a um mesmo processo. Assim, este meio de comunicação possibilita que um valor seja modificado por uma das *threads* enquanto é utilizado por outra, o que ocasiona estados inconsistentes de memória. Nesta perspectiva, há necessidade de utilizar-se mecanismos de sincronização para garantir a integridade e consistência entre as execuções concorrentes, evitando então que uma variável seja modificada enquanto em uso por algum fluxo (ANDREWS, 2000).

Os mecanismos de sincronização são tradicionalmente implementados com uso de *locks* explícitos pelo programador, onde certas variáveis são associadas à proteção (entrada e saída) da seção crítica, ou a condição para postergação ou avanço de determinadas tarefas. Embora muito utilizada, esta abordagem é propensa a erros e pode apresentar uma série de problemas tais como instabilidade no sistema, dificuldade de composição e baixo desempenho (JONES, 2007; MCKENNEY et al., 2010).

As dificuldades e limitações encontradas na sincronização baseada em *locks* fizeram ressurgir o interesse por parte de pesquisadores da área de programação concorrente na busca por métodos de sincronização mais simples e eficientes. Em especial, a ideia de usar transações como mecanismo de sincronização em linguagens convencionais ganhou nova e revigorada força (BALDASSIN, 2009). Neste novo mecanismo, denominado Memória Transacional (*Transactional Memory* - TM), a tarefa de codificação de *software* paralelo é substancialmente diferente se comparada com a utilização de *locks* explícitos, visto que não é necessário adquirir e liberar *locks* para garantir a proteção da seção crítica (HARRIS; LARUS; RAJWAR, 2010).

Nesta perspectiva, permite-se uma programação mais simples pois o programador apenas precisa delimitar a seção crítica, deixando para o sistema de execução transacional a obrigação de implementar eficientemente a sincronização em baixo nível. As principais vantagens do modelo transacional estão no aumento do nível de abstração, no potencial ganho de desempenho e escalabilidade do código, e na praticidade do uso de técnicas já conhecidas em engenharia de *software*, como a composição de código, no contexto de programação concorrente (BALDASSIN, 2009).

As implementações de TM são comumente divididas em três modelos: *Hardware Transactional Memory* (HTM), *Software Transactional Memory* (STM) e *Hybrid Transactional Memory* (HyTM) (HARRIS; LARUS; RAJWAR, 2010). As implementações das três abordagens devem garantir os conceitos de atomicidade e isolamento entre as execuções das transações. Os modelos de HTM garantem estes conceitos através de modificações e extensões em alguns componentes da arquitetura computacional (processador, *cache*, barramento e protocolos de coerência). A principal vantagem deste modelo é o desempenho, pelo fato de implementar as transações diretamente em *hardware*. No entanto, como desvantagem, as HTMs são consideradas complexas, dificultando a adoção efetiva em processadores comerciais. As implementações em *software* empregam bibliotecas, compiladores e/ou sistemas de execução (*runtime systems*) para garantir a execução atômica e isolada das transações. Como principal vantagem desta abordagem, tem-se uma maior flexibilidade na implementação aliada à execução em máquinas atuais e subsequentemente a portabilidade deste modelo, visto que alterações na arquitetura computacional não fazem-se necessárias. A abordagem híbrida combina os modelos de HTM e STM com intuito de obter o melhor resultado com a união de ambas abordagens (alto desempenho e recursos ilimitados). No que diz respeito a TMs, o escopo desta dissertação restringe-se às STMs.

As STMs garantem a execução atômica e isolada das transações (seções críticas) concorrentes através de dois mecanismos chave: *(i)* detecção/resolução de conflitos e *(ii)* versionamento de dados (HARRIS; LARUS; RAJWAR, 2010). Há ocorrência de conflitos entre transações concorrentes quando no mesmo intervalo de tempo duas ou mais transações acessam o mesmo dado compartilhado e ao menos um dos acessos é de escrita. Em caso de conflito, um componente do sistema transacional, denominado gerenciador de contenção, é invocado para resolver os conflitos e garantir o progresso do sistema. O versionamento de dados lida com o gerenciamento das diferentes versões dos dados (originais e especulativas) manipulados por uma transação. Em casos de efetivação da transação, os dados especulativos são armazenados na memória, caso contrário

permanecem os originais e a transação é cancelada e reexecutada.

Para alcançar um desempenho que torne o uso de TM viável, o entendimento da influência dos requisitos de implementação nesses sistemas é cada vez mais importante. Outro fator de extrema relevância na avaliação de novas abordagens é o consumo de energia. No entanto, por se tratar de uma alternativa recentemente proposta, pouco se conhece a respeito do consumo de energia de TM, em especial de implementações em *software*. Além disso, as poucas pesquisas que já analisaram o consumo de energia de STMs foram realizadas estritamente em ambientes computacionais simulados.

Nesse contexto, este trabalho visa suprir a deficiência da literatura, apresentando a análise do consumo de energia e desempenho de quatro bibliotecas de STM, em ambiente de computação real.

1.1 Objetivos

Este trabalho tem como principal objetivo **analisar e caracterizar o consumo de energia e desempenho de aplicações baseadas em quatro bibliotecas de STMs, sob diversos cenários de execução em ambiente de computação real**.

Tem-se como objetivos específicos:

- Analisar o tempo de execução e consumo de energia e verificar se a hipótese que o consumo de energia é proporcional ao tempo de execução é verdadeira;
- Verificar se o número de cancelamentos de transações eleva-se conforme aumenta-se o número de *threads*;
- Identificar a implementação de STM mais eficiente em termos de consumo de energia e desempenho para diferentes cenários de execução.

1.2 Estrutura do texto

O trabalho que segue está organizado em cinco capítulos. O Capítulo 2 apresenta os conceitos e o mecanismo de funcionamento de Memórias Transacionais. Além disso, nesse capítulo também são descritos em detalhes as características das bibliotecas de STM utilizadas nos experimentos. Esse capítulo serve como base para as discussões apresentadas nos que o seguem. No Capítulo 3 são apresentados os trabalhos relacionados que avaliam o consumo de energia e desempenho de Memórias Transacionais. No Capítulo 4 concentra-se o cerne

desta dissertação. Nesse capítulo são apresentadas a metodologia e a caracterização dos resultados obtidos na medição do consumo de energia e desempenho das bibliotecas de STM utilizadas. Por fim, no Capítulo 5 são apresentadas as conclusões, os trabalhos futuros, e as publicações realizadas durante o desenvolvimento do presente trabalho.

Nos anexos A e B são mostradas as médias de desempenho e consumo de energia obtidas nos experimentos, em forma de tabelas. Nos anexos C, D e E são apresentados o percentual de desvio-padrão de desempenho, consumo de energia e cancelamentos de transações. No anexo F é apresentado um mini-tutorial de configuração do microcontrolador utilizado para capturar os dados de consumo de potência, e a configuração da interface utilizada para obter estes dados.

2 MEMÓRIAS TRANSACIONAIS

A Memória Transacional (*Transactional Memory* - TM) oferece uma alternativa atraente aos *locks*, executando seções críticas de forma transacional (HARRIS; LARUS; RAJWAR, 2010). Este modelo de sincronização introduz os conceitos de execução especulativa e detecção de conflitos. Por conflito, entende-se que uma transação modificou um valor em memória compartilhada utilizado por outra transação simultaneamente. Se ao final da execução da transação não houver nenhum conflito com as demais transações em execução, as modificações são aplicadas à memória. Na ocorrência de conflitos, as modificações decorrentes da transação são descartadas e a transação é reexecutada. Por deixar a detecção de conflitos a cargo do sistema transacional, o programador apenas delimita as transações (seções críticas) e o sistema de execução transacional garante que estas sejam executadas de forma atômica e isolada.

O conceito de transação, criado e desenvolvido em sistemas de banco de dados, refere-se a um conjunto de operações que formam uma única unidade de trabalho lógica, satisfazendo as propriedades ACID (**A**tomicidade, **C**onsistência, **I**solamento e **D**urabilidade) (SILBERSCHATZ; KORTH; SUDARSHAN, 1999).

A **propriedade de atomicidade** define que as instruções pertencentes ao escopo da transação serão todas efetivadas com sucesso ou todas descartadas. Quando a transação for efetivada com sucesso o resultado da execução é permanente, e diz-se que a transação sofreu *commit*. Quando ocorrer falha/conflito na transação, diz-se que esta sofreu *abort*, e os valores pertencentes à execução da transação são descartados.

A **propriedade de consistência** assegura a integridade dos dados, garantindo que uma transação levará o sistema de um estado consistente (pré-transação) a outro estado consistente (pós-transação).

A **propriedade de isolamento** implica que transações concorrentes não poderão verificar resultados intermediários produzidos por outras transações, de modo que o resultado de execução de diversas transações seja equivalente ao resultado de execução destas mesmas em ordem serial.

A **propriedade de durabilidade** indica que as mudanças efetivadas por transações são permanentes e resistentes a eventuais falhas no sistema.

O conceito de transação no contexto de memórias transacionais é basicamente o mesmo, com exceção para a propriedade de durabilidade. O conceito de durabilidade pode ser ignorado em memórias transacionais, pois em seu contexto os dados serão armazenados em memória volátil, ao contrário de sistemas de banco de dados onde os dados são armazenados em memória persistente.

O termo "Memória Transacional" foi cunhado em 1993 por Herlihy e Moss para designar: *"uma nova arquitetura para microprocessadores que objetiva tornar a sincronização livre de bloqueio tão eficiente (e fácil de usar) quanto técnicas convencionais baseadas em exclusão mútua"* (HERLIHY; MOSS, 1993). Este termo define qualquer mecanismo de sincronização que utilize o conceito de transação para coordenar acessos à memória compartilhada.

Embora TM tenha recentemente se popularizado, a iniciativa de usar transações como forma de estruturação de programas concorrentes existe há mais de três décadas (LOMET, 1977). Todavia, o conceito não se popularizou na época principalmente por não se conhecer uma implementação eficiente. Além disso, os mecanismos de sincronização baseados em *locks* explícitos eram bastante pesquisados naquela época, contribuindo para que o trabalho de Lomet (1977) não ganhasse maior notoriedade (BALDASSIN, 2009).

Em relação ao uso de *locks*, a memória transacional apresenta as seguintes vantagens (MCKENNEY et al., 2010):

- **Facilidade de programação**: A programação torna-se mais fácil porque o programador não a tem responsabilidade de garantir a sincronização, e sim em especificar quais blocos de código devem ser executados atomicamente, cabendo ao sistema transacional a implementação do mecanismo de sincronização.

- **Escalabilidade**: Transações que acessem um mesmo dado para leitura podem ser executadas concorrentemente. Também podem ser executadas de modo paralelo as transações que modifiquem partes distintas de uma mesma estrutura de dados. Essa característica tem como vantagem garantir que mais desempenho seja obtido com o aumento do número de processadores porque o nível de paralelismo exposto é maior.

- **Composabilidade**: Transações suportam naturalmente a composição de código. Para criar uma nova operação baseando-se em outras já existentes, deve-se invocá-las dentro de uma nova transação, característica conhecida como *aninhamento*. O sistema transacional irá garantir que tais operações sejam executadas de forma atômica.

Na próxima Seção será apresentada a memória transacional sob a perspectiva do programador, em relação a linguagem e semântica. A Seção 2.2 apresenta o mecanismo de funcionamento de Memórias Transacionais. Na Seção 2.3 são descritos em detalhes as características das bibliotecas de STM utilizadas nos experimentos. Por fim, na Seção 2.4 são apresentadas as observações finais deste capítulo.

2.1 Linguagem e semântica

Aspectos referentes à linguagem e semântica são importantes por indicarem o grau de expressividade da linguagem e definir univocamente o significado de cada uma de suas construções (BALDASSIN, 2009).

No contexto de memórias transacionais, as seguintes três construções são atualmente aceitas: *atomic*, *retry* e *orElse*.

2.1.1 Construção *atomic*

A construção `atomic`, ou em algumas linguagens `atomically`, representa o bloco atômico, o qual é responsável por delimitar o escopo que deve ser executado por uma transação (HARRIS; FRASER, 2003). Uma grande vantagem do bloco atômico é que não é necessário criar manualmente variáveis específicas para controlar e bloquear a seção crítica de um programa, diferentemente de outras construções, como *locks*. O sistema transacional será responsável por garantir a sincronização do bloco, deixando a encargo do programador apenas especificar quais serão os blocos atômicos e não *como* sincronizá-los.

Na Figura 1 mostra-se um exemplo de utilização do bloco atômico, sendo delimitado pela palavra *atomic* (linhas 2 a 6).

```
1    public void deposita(double quantia) {
2        atomic{
3            if (quantia > 0) {
4                this.saldo = this.saldo + quantia;
5            }
6        }
7    }
```

Figura 1: Bloco atômico

A composição de transações é ilustrada na Figura 2, através de um método que transfere um valor entre duas contas bancárias. Pelo fato da operação ser efetuada de forma atômica e isolada, é garantido que nenhuma outra transação verá o estado intermediário na qual o valor sacado de uma conta (linha 3) ainda não tenha sido depositado na outra (linha 4). Caso haja conflito em alguma das

operações (saque ou depósito), o sistema transacional abortará toda transação, descartando as alterações em ambas operações.

```
1     public void transferencia(ContaBancaria contaOrigem, ContaBancaria
          contaDestino, Double quantia) {
2          atomic{
3               contaOrigem.saque(quantia);
4               contaDestino.deposito(quantia);
5          }
6     }
```

Figura 2: Composição em memória transacional

2.1.2 Construção *retry*

A construção `retry` define que a transação seja cancelada, desfazendo todas as ações intermediárias (HARRIS et al., 2005). Quando algum dado compartilhado recentemente acessado pela transação é modificado, a transação é reexecutada. Por exemplo, suponha-se a remoção de elemento de um *buffer*. Sempre que um *buffer* estiver vazio, a transação invoca a construção `retry`, fazendo com que a transação seja cancelada e reexecutada quando a variável `itens` for modificada por outra transação. Este exemplo é ilustrado na Figura 3.

A fim de comparar uma construção transacional com outra abordagem de sincronização, o mesmo exemplo de remoção de elemento de um *buffer* será implementado em Java utilizando monitores, conforme mostra na Figura 4. O uso da primitiva `synchronized` indica que a *thread* que invocar o método `remover` deverá obter o bloqueio associado ao objeto *buffer*, antes de prosseguir com a operação de remoção. A aquisição e liberação de bloqueios é implícita em Java, quando o método for qualificado como `synchronized`. O método `wait` serve para garantir que não seja removido um elemento de um *buffer* vazio e o método `notifyAll` notifica as outras *threads* que um elemento foi removido do *buffer*.

```
1     public int remover() {
2         atomic {
3             if (itens == 0)
4                 retry;
5             itens--;
6             return buffer[itens];
7         }
8     }
```

Figura 3: Remoção de elemento de *buffer* codificada com a construção *retry*. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

A partir dos dois exemplos mostrados nas Figuras 3 e 4, observa-se um problema comum com bloqueios: baixa concorrência e, consequentemente, baixo

```
1     public synchronized int remover() {
2        int resultado;
3        while (itens == 0)
4           wait();
5        itens--;
6        resultado = buffer[itens];
7        notifyAll();
8        return resultado;
9     }
```

Figura 4: Remoção de elemento de *buffer* codificada em Java com monitor. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

desempenho. Como um bloqueio associado ao objeto *buffer* é adquirido ao invocar o método `remover`, qualquer outro método do objeto que eventualmente seja invocado será bloqueado até que o método libere o bloqueio. Ou seja, duas ou mais operações não acontecerão concorrentemente no *buffer* mesmo quando há uma possibilidade de paralelismo. Por exemplo, é possível que uma operação de inserção ocorra simultaneamente a uma de remoção se ambas acessarem elementos disjuntos no *buffer*. Portanto, o uso de transações consegue explorar mais paralelismo, pois a detecção de conflitos é feita de forma dinâmica (em tempo de execução).

2.1.3 Construção *orElse*

A construção `orElse` permite que transações sejam compostas como alternativas para execução, de modo que somente uma transação seja executada entre várias (HARRIS et al., 2005).

Por exemplo, suponha-se um sistema que remova elementos de um *buffer*. O método de remoção pode ser bloqueado (invocar `retry`) se o *buffer* estiver vazio, conforme mostrado na Figura 3. Uma implementação possível usando `orElse` para esta situação é mostrada na Figura 5. Nesta implementação, o sistema transacional tentará remover elementos entre dois *buffers*. Caso o *buffer* 1 estiver vazio o sistema tentará remover o elemento do *buffer* 2. Caso os dois *buffers* estiverem vazios, o bloco atômico será bloqueado até que ao menos um dos *buffers* contenha um elemento para poder ser removido.

```
1    public void exemploOrElse() {
2       atomic {
3             { x = buffer1.remover(); }
4          orElse
5             { x = buffer2.remover(); }
6       }
7    }
```

Figura 5: Uso da construção *orElse*. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

2.1.4 Níveis de isolamento

O nível de isolamento está relacionado à interação entre código transacional e código não transacional. Existem dois tipos de isolamento: **atomicidade forte** (*strong atomicity*) e **atomicidade fraca** (*weak atomicity*) (BALDASSIN, 2009). No modelo de **atomicidade forte**, o acesso a um dado compartilhado fora de uma transação é consistente com os acessos efetuados ao mesmo dado dentro da transação. No modelo de **atomicidade fraca** o acesso a um mesmo dado compartilhado, dentro e fora de uma transação, pode causar condição de corrida e portanto o resultado é não-determinístico.

2.2 Mecanismo de funcionamento

As primeiras implementações de STMs são não-bloqueantes (HERLIHY et al., 2003; HARRIS; FRASER, 2003; FRASER, 2004; HARRIS et al., 2005; MARATHE; Scherer III; SCOTT, 2005; MARATHE et al., 2006). Este tipo de implementação exclui qualquer uso de *locks*, já que esses podem induzir estado de espera (BALDASSIN, 2009). No entanto, constatou-se através de pesquisas que as abordagens de STMs não-bloqueantes apresentam baixo desempenho, se comparadas a utilização de *locks* explícitos (ENNALS, 2006). A partir desse importante resultado a maioria das implementações propostas passaram a utilizar *locks*, de maneira implícita.

O sistema transacional garante a atomicidade e isolamento baseado em dois mecanismos chave: *(i)* versionamento de dados; e *(ii)* detecção e resolução de conflitos.

2.2.1 Versionamento de dados

O versionamento de dados lida com o gerenciamento das diferentes versões dos dados (originais e especulativas) acessados por uma transação. A **versão original** corresponde ao valor do dado antes do início da transação. A **versão especulativa** representa o valor intermediário sendo trabalhado pela transação.

Caso a transação seja efetivada, os **valores especulativos** tornam-se os valores correntes e são armazenados, e os valores originais são descartados. Do contrário, descarta-se os especulativos e mantém-se os originais, e a transação é reexecutada (BALDASSIN, 2009).

Existem duas formas de realizar o versionamento de dados: *(i)* adiantado ou direto (*eager versioning* ou *direct update*); e *(ii)* atrasado ou deferido (*lazy versioning* ou *deferred update*) (HARRIS; LARUS; RAJWAR, 2010).

No **versionamento de dados adiantado**, os valores especulativos são armazenados diretamente na memória, enquanto que os valores originais são armazenados em um *undo log*. No caso de uma transação ser abortada, os dados que foram gravados na memória inicialmente deverão ser convertidos para suas versões antigas (originais), através do *undo log*.

No **versionamento de dados atrasado**, o armazenamento de novos dados é realizado em um *buffer* local, e a memória continua com os valores originais. Os dados especulativos são transferidos do *buffer* para a memória somente quando a transação for confirmada.

Ilustra-se na Figura 6 um exemplo de ambos os tipos de versionamento de dados. Inicialmente, o conteúdo da variável `X` na memória corresponde ao valor 100 e os respectivos *undo log* e *buffer* estão vazios. Quando uma transação atribui o valor 77 à variável `X`, no versionamento adiantado altera-se imediatamente o conteúdo da memória e armazena-se o valor original localmente (*undo log*), enquanto no versionamento atrasado somente salva-se o valor especulativo em memória local (*buffer*). No momento da efetivação da transação, a única ação necessária no versionamento adiantado é invalidar o *undo log*, pois o dado especulativo já está na memória. Já no versionamento atrasado, é necessário primeiramente mover o valor especulativo do *buffer* para a memória do sistema. Uma situação inversa acontece em caso de cancelamento da transação. Neste caso, no versionamento adiantado precisa-se restaurar as mudanças efetuadas na memória, movendo para esta o valor original presente no *undo log*. No versionamento atrasado só é necessário descartar os valores especulativos.

Conclui-se que o versionamento adiantado torna-se mais eficiente em casos em que a transação é confirmada, pois os dados especulativos já estão na memória. Por outro lado, quando a transação é cancelada, deve-se restaurar os valores originais para a memória, que estão armazenados no *undo log*. No versionamento atrasado a situação é oposta, visto que transações confirmadas necessitam transferir os dados especulativos do *buffer* para a memória, enquanto que transações canceladas não necessitam de nenhuma operação.

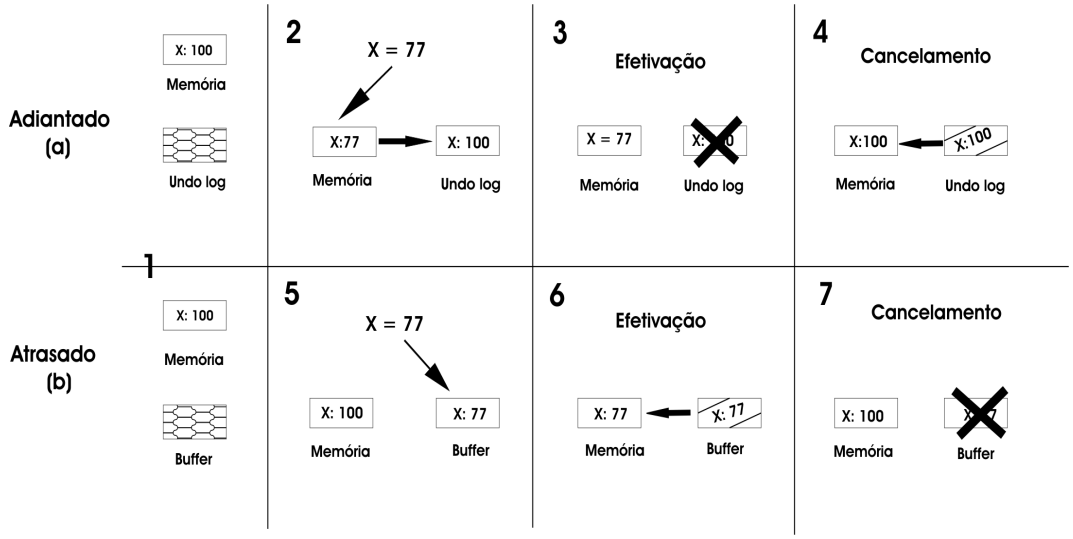


Figura 6: Exemplo ilustrando versionamento adiantado (a) e atrasado (b). Fonte: (BALDASSIN, 2009)

2.2.2 Detecção de conflitos

Para detecção de conflitos, uma transação geralmente mantém um conjunto de leitura (*read set*) com os endereços dos dados lidos, e um conjunto de escrita (*write set*) com os endereços dos dados que foram alterados. As STMs usam barreiras de leitura e escrita para interceptar os acessos aos dados compartilhados e manter os conjuntos de leitura e escrita (HARRIS; LARUS; RAJWAR, 2010).

Há um conflito entre duas ou mais transações quando a intersecção entre o conjunto de leitura e o conjunto de escrita de transações diferentes é não vazia. Em outras palavras, quando no mesmo intervalo de tempo duas ou mais transações acessam o mesmo dado compartilhado e ao menos um dos acessos é de escrita (BALDASSIN, 2009).

O sistema transacional efetua a detecção de conflitos baseando-se na granularidade de detecção. A granularidade pode ser em três níveis: **objeto**, **palavra** e **linha de *cache*** (BALDASSIN, 2009). A **granularidade por nível de objetos** pode reduzir o *overhead* em termos de espaço e tempo para detecção de conflitos. No entanto, este nível permite detectar falsos conflitos, ou seja, casos em que o sistema detectará conflitos quando duas transações operam em diferentes partes de um mesmo objeto. A **granularidade por nível de palavras** elimina a detecção de falsos conflitos, por outro lado, necessita-se de maior custo e espaço para a detecção. A **granularidade por nível de linha de *cache*** provê um acordo entre a frequência de detecção de falsos conflitos e o *overhead* em termos de tempo e espaço. No entanto, necessita-se de alterações na estrutura de *hardware* para prover a *cache* transacional, alterando seu protocolo de

coerência.

Assim como no versionamento de dados, há dois modos de detecção de conflitos: *(i)* adiantado/pessimista (*eager/pessimistic conflict detection*); e *(ii)* atrasado/otimista (*lazy/optimistic conflict detection*) (HARRIS; LARUS; RAJWAR, 2010).

No modo de **detecção de conflitos adiantado**, verifica-se a ocorrência de conflito no momento em que cada posição de memória é acessada. Desta forma pode-se evitar que uma transação condenada a abortar execute desnecessariamente, mas por outro lado, pode-se cancelar transações, que dependendo do progresso de outras, poderiam ser confirmadas com sucesso. Para exemplificar este modo de detecção, os cenários mostrados na Figura 7 serão considerados. No caso 1, `T1` e `T2` manipulam conjuntos de dados disjuntos e portanto não há nenhum conflito. No caso 2 tem-se uma operação de escrita depois de uma leitura. A transação que escreveu (`T2`) faz com que a outra transação (`T1`) seja cancelada. O caso 3 mostra a situação em que, após ser cancelada, `T1` volta a executar e por sua vez cancela a transação `T2`. Neste exemplo, pode-se notar que as transações `T1` e `T2` estão em estado de *livelock*, desta forma, a efetivação de tais transações é indefinida.

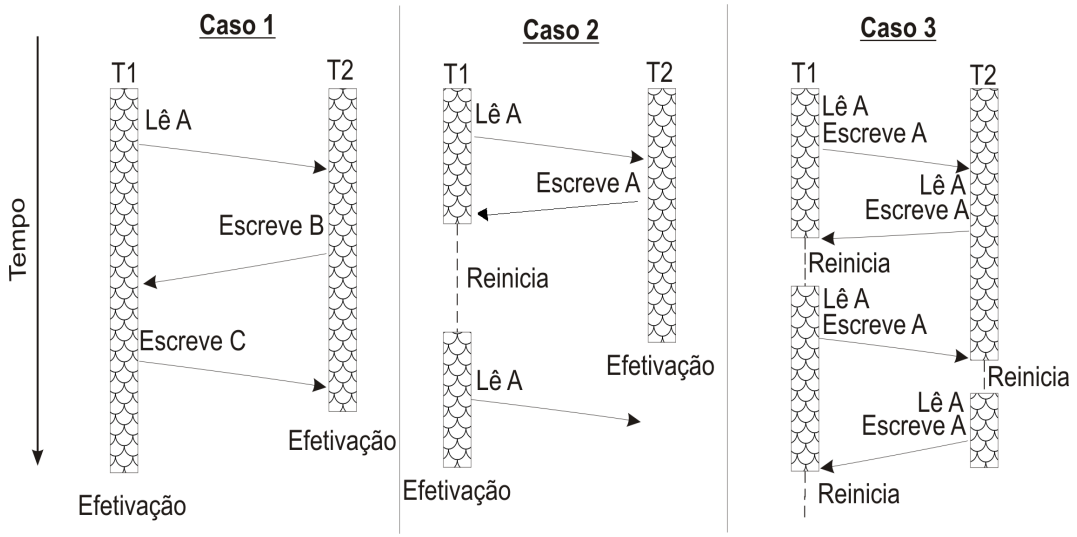


Figura 7: Detecção de conflitos em modo adiantado. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

Na **detecção atrasada** o sistema detecta se há conflito somente no momento de efetivação da transação. Assim, permite-se que transações conflitantes prossigam na esperança do conflito não se efetivar de fato. De modo a exemplificar a detecção atrasada, os cenários mostrados na Figura 8 serão considerados. No caso 1, as transações acessam um conjunto de dados disjuntos, não ocasionando conflitos. No caso 2, `T2` lê uma variável escrita por `T1`. Quando `T1`

sofre efetivação, T2 nota o conflito e é reiniciada. No caso 3 não ocorre nenhum conflito, pois T1 apenas lê uma variável que T2 está escrevendo e esta sofre efetivação após T1. O caso 4 mostra a situação em que, após ser cancelada, T1 volta a executar. No entanto, diferentemente da detecção adiantada, este caso não gera situação de *livelock*, porém, poderia acontecer situação de *starvation*. Se T1 fosse muito longa poderia ser sempre cancelada por outras transações e nunca conseguir efetivar suas alterações.

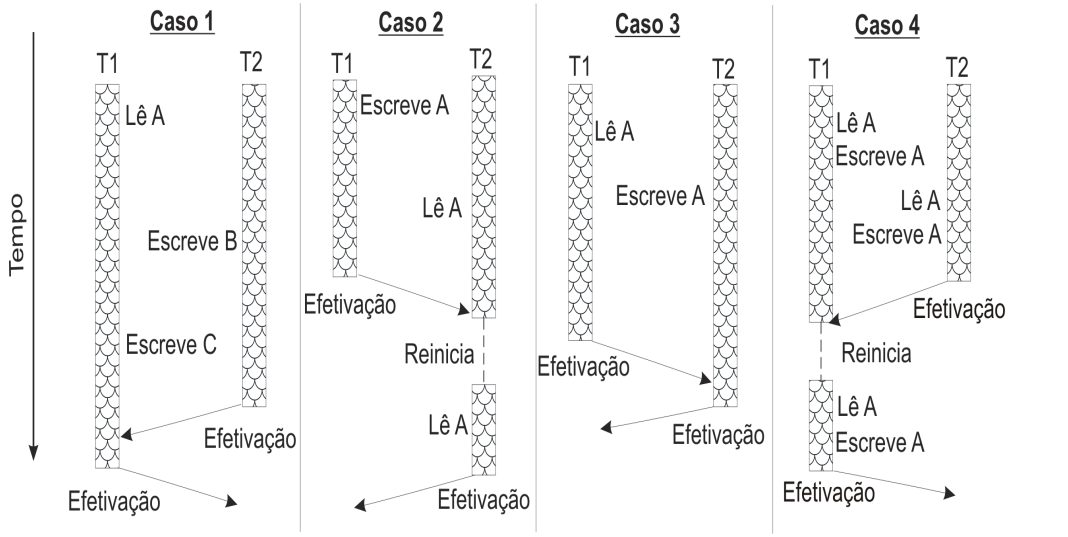


Figura 8: Detecção de conflitos em modo atrasado. Fonte: (RIGO; CENTODUCATTE; BALDASSIN, 2007)

2.2.3 Resolução de conflitos

Em caso de conflito, um componente do sistema transacional, denominado gerenciador de contenção, é invocado para resolver os conflitos entre as transações e assim garantir o progresso do sistema. Este componente implementa uma ou mais políticas de resolução de conflitos, as quais ditam caso deva-se abortar a transação que detectou o conflito, as outras transações conflitantes, e caso atrasar ou não a reexecução das transações abortadas (HARRIS; LARUS; RAJWAR, 2010).

Algumas propostas de TM possuem um método fixo para gerenciamento de contenção, ao passo que outras propostas tratam o problema como um aspecto modular do sistema, podendo ser alterado para melhorar o desempenho de um programa sob determinada carga de trabalho (KRONBAUER; RIGO, 2009). Dentre as diversas políticas existentes, destacam-se:

- **Tímido**: a transação que detectou o conflito é cancelada e reexecutada imediatamente. É o mecanismo mais simples de gerenciamento de contenção.

- **Guloso**: o gerenciador de contenção guloso atribui um marcador de tempo (*timestamp*) a cada transação no início de sua primeira tentativa de execução transacional. Em caso de conflito, a transação que possuir o *timestamp* mais antigo prevalece. No entanto, se uma transação mais nova está bloqueada pela mais antiga e detecta que esta também está bloqueada, esperando por um recurso adquirido por outra transação, então a transação mais nova cancela a mais antiga e prossegue a execução.

- ***Backoff***: a transação cancelada aguarda por um período (linear ou exponencial) de acordo com o número de seus cancelamentos para então ser reexecutada.

- ***Delay***: a transação abortada será reexecutada somente após o *lock* que causou o conflito ser liberado.

2.3 Implementações de STMs

O primeiro modelo de STM foi proposto por Shavit e Touitou (1995). Desde então, diferentes propostas e implementações de STMs têm sido desenvolvidas e testadas. Nesta perspectiva, observa-se um grande avanço nos sistemas de STMs desde a primeira proposta, e novos trabalhos e aprimoramentos estão surgindo na área.

Nas quatro próximas subseções serão detalhadas as principais características de quatro relevantes bibliotecas de STMs utilizadas pelo presente trabalho, TL2, TinySTM, SwissTM e AdaptSTM.

2.3.1 TL2

A implementação TL2, por padrão, utiliza versionamento de dados atrasado, granularidade em nível de palavra, detecção de conflitos otimista, e sincronização baseada em *locks* implícitos. Além disso, esta implementação faz uso de um relógio de versão global, denominado GV4, que funciona como um mecanismo de versão para os dados compartilhados (DICE; SHALEV; SHAVIT, 2006).

O relógio global é incrementado de forma atômica por todas transações confirmadas que escrevem na memória, e lido por todas transações. No início de cada transação, o relógio global é lido e copiado para uma variável local da *thread* chamada *read version* (rv). Em cada leitura ou escrita dos dados da transação é verificada a consistência dos dados, comparando-se o rv da *thread* com a versão do dado correspondente à localização de memória. Se o rv é menor do que o número da versão do dado, a transação é abortada pois a localização de memória foi modificada após a transação ter sido iniciada.

Apesar da redução do custo de validação utilizando-se o relógio global, este mecanismo pode se tornar um gargalo conforme o número de *threads* aumenta, pois maior será a contenção em relação ao incremento do relógio.

A TL2 utiliza o gerenciador de contenção tímido que aborta prontamente a transação que detectou o conflito. Além disso, emprega um mecanismo de *backoff* que atrasa a reexecução da transação por um tempo após esta ter sido abortada três vezes. Se a transação abortar novamente, o tempo de *backoff* é incrementado de forma linear (padrão) ou exponencial. Este mecanismo tem como objetivo evitar o cancelamento consecutivo de uma transação.

2.3.2 TinySTM

A implementação TinySTM é baseada no algoritmo LSA (*Lazy Snapshot Algorithm*), o qual possui características similares às do TL2 (FELBER; FETZER; RIEGEL, 2008). Utiliza-se um relógio global, a sincronização é baseada em *locks* implícitos, a granularidade de detecção de conflito é em nível de palavra, o versionamento de dados atrasado e gerenciador de contenção tímido.

No entanto, diferentemente da implementação TL2, a TinySTM por padrão realiza a detecção de conflito de modo adiantado. Desta forma, evita-se que uma transação condenada a abortar continue executando e assim desperdiçando recursos computacionais.

Assim como na maioria das implementações de STM baseadas em palavra, a TinySTM utiliza um vetor de *locks* para gerenciar o acesso concorrente à memória. Cada *lock* protege uma parte do espaço de endereçamento.

2.3.3 SwissTM

A SwissTM busca obter bom desempenho tanto para transações curtas e estruturas de dados pequenas, como para transações grandes e cargas de trabalho complexas. Para isso, esta implementação emprega uma estratégia mista de detecção de conflitos e gerenciador de contenção, utiliza-se versionamento de dados atrasado, granularidade em nível de palavra, e sincronização baseada em *locks* implícitos. Além disso, faz uso de um relógio global para o processo de validação dos dados (DRAGOJEVIC; GUERRAOUI; KAPALKA, 2009).

A detecção de conflitos de transações escrita/escrita é feita de modo adiantado, a fim de prevenir que transações condenadas a abortar continuem executando. Em transações de leitura/escrita o sistema transacional detecta conflitos de forma atrasada, na esperança do conflito não se efetivar de fato, e assim permitir de forma otimista que mais paralelismo seja obtido.

O gerenciador de contenção usa o esquema tímido para conflitos detectados em transações curtas ou somente leitura. Em transações mais complexas, o sis-

tema transacional muda dinamicamente para o gerenciador guloso (*greedy*), que dá prioridade para a transação mais antiga, evitando assim a possibilidade de *starvation*. Além disso, em transações canceladas devido a conflitos de escrita-escrita utiliza-se um *backoff* linear por um período proporcional ao número de seus cancelamentos sucessivos.

2.3.4 AdaptSTM

A implementação AdaptSTM tem como principal característica adaptar suas estratégias de execução conforme o cenário apresentado e em tempo de execução. Esta adaptação tem como objetivo obter os mais eficientes parâmetros de execução tais como versionamento de dados, gerenciador de contenção, tamanho do *write-set*, e outros, para cada cenário (PAYER; GROSS, 2011).

O sistema transacional coleta dados estatísticos de execução das transações, e então define a melhor estratégia de acordo com o tipo de contenção a partir das estatísticas coletadas. O sistema de adaptação usa a média das últimas 64 transações para calcular a taxa de cancelamentos, e desta forma verificar o tipo de contenção.

A AdaptSTM emprega versionamento de dados atrasado em cenários de alta contenção, pois este modo oferece pouco custo em casos de cancelamentos de transações. Em ambientes de baixa contenção, utiliza-se versionamento de dados adiantado, visto que o custo de efetivação é menor do que no modo atrasado.

Essa implementação é baseada em *locks* implícitos, utiliza granularidade em nível de palavra e faz uso de um relógio global para o processo de validação dos dados. Por padrão, a AdaptSTM utiliza detecção de conflito adiantado, emprega o gerenciador de contenção tímido em cenário de baixa contenção e usa *backoff* exponencial em cenário de alta contenção.

2.4 Observações finais

Este capítulo apresentou os conceitos, aspectos referentes à linguagem e semântica, mecanismo de funcionamento e implementações de Memórias Transacionais.

A sincronização baseada em memórias transacionais garante maior abstração na codificação de programas paralelos, pois tira da responsabilidade do programador o modo como as sincronizações serão feitas, ficando a encargo do sistema que implementa as memórias transacionais garantir a sincronização em baixo nível. Por implementar as sincronizações em baixo nível, o sistema transacional facilita a codificação de programas paralelos, além de reduzir as dificuldades e limitações encontradas no uso de *locks*.

Mostrou-se que para cada mecanismo de funcionamento (versionamento de dados e detecção/resolução de conflitos) uma determinada estratégia pode ser eficiente em determinados cenários porém inefetiva em outros. Neste contexto, as implementações que adaptam suas estratégias de versionamento de dados e detecção/resolução de conflitos em tempo de execução, tornam-se as mais eficientes, pois buscam obter os melhores parâmetros de execução conforme o cenário.

As implementações de STM se diferenciam principalmente pelas estratégias de versionamento de dados, detecção e resolução de conflitos (BALDASSIN, 2009). Na Tabela 1 são sintetizadas as principais características das STMs utilizadas no presente trabalho.

Tabela 1: Características das STMs.

STM	**Versionamento de dados**	**Detecção de conflitos**	**Gerenciador de contenção**
AdaptSTM	adiantado (baixa contenção) atrasado (alta contenção)	adiantado	tímido (baixa contenção) *backoff* exponencial (alta contenção)
SwissTM	atrasado	adiantado (escrita-escrita) atrasado (leitura-escrita)	tímido (transações curtas e de leitura) guloso (transações complexas) *backoff* linear (conflito escrita-escrita)
TinySTM	atrasado	adiantado	tímido
TL2	atrasado	atrasado	tímido *backoff* linear (após 3 cancelamentos)

3 TRABALHOS RELACIONADOS

Este capítulo apresenta o estado da arte da análise de consumo de energia e desempenho de TMs.

Nas próximas sete seções serão apresentados os principais trabalhos relacionados ao contexto desta dissertação. Por fim, na Seção 3.8 serão apresentadas as observações finais deste capítulo.

3.1 Reduzindo o consumo de energia em um sistema multiprocessado usando HTM

O trabalho de Moreshet, Bahar e Herlihy (2005) avalia o consumo de energia de HTM em comparação a *locks (spinlock)*, sendo pioneiro em avaliar o consumo energético em HTM.

Sua implementação baseia-se no modelo original de HTM (HERLIHY; MOSS, 1993), no qual há uma *cache* transacional completamente associativa para armazenar os valores especulativos. Utiliza a plataforma de simulação SIMICS (MAGNUSSON et al., 2002) para modelar um sistema com processadores UltraSparc II conectados por barramento, assumindo dados de energia e latência para uma memória compartilhada *off-chip*, e sistema operacional Linux.

Duas diferentes abordagens para a HTM utilizada foram experimentadas, sendo que a diferença entre ambas está no método utilizado para resolução de conflitos em transações. A primeira abordagem é o padrão da HTM utilizada, reexecuta as transações conflitantes em paralelo (baseado em *backoff* aleatório). A segunda foi proposta pelos próprios autores, sendo um mecanismo para execução de modo serial de todas transações pendentes durante o conflito.

Através dos resultados obtidos neste trabalho mostrou-se que a versão utilizando *locks* consumiu 10x mais energia que as abordagens de HTM utilizadas. Tal resultado deve-se a significativa fração de energia consumida pela hierarquia de memória. Em relação às HTMs utilizadas, a que utilizou a execução serial para resolução de conflitos mostrou-se mais eficiente em termos de energia. No

entanto, esta é uma abordagem pessimista, na qual há uma contenção do paralelismo.

Embora os resultados obtidos neste trabalho mostrem que a HTM utilizada é eficiente em termos de consumo de energia, o trabalho apresentou alguns problemas que tornam seus resultados pouco atraentes. Apenas um *microbenchmark* (desenvolvido pelos próprios autores) com apenas uma configuração do sistema (quatro processadores) é utilizado para os experimentos. Além disso, não é descrito como é implementado o mecanismo de execução serial das transações conflitantes.

3.2 Comparação de desempenho e consumo de energia de HTM e *locks*

O trabalho de Moreshet, Bahar e Herlihy (2006) apresenta uma extensão do trabalho anterior dos mesmos autores (2005). Utilizando a mesma HTM e plataforma de simulação, avalia o consumo de energia e desempenho de HTM em comparação a *locks (spinlock)* através de quatros aplicações do *benchmark* SPLASH-2 (WOO et al., 1995).

Neste trabalho são apresentados maiores detalhes em relação à execução serial das transações conflitantes, visto que no trabalho anterior apenas foi citado esse novo mecanismo. Com o sistema transacional de execução serial, um conflito ainda resulta em retornar ao estado anterior. No entanto, durante alta contenção, em vez de reexecutar as transações depois do conflito, o sistema transacional irá esperar uma transação completar antes de outra começar sua execução. Após o período conflitante, o sistema transacional voltará a emissão de transações em paralelo.

Inicialmente, constatou-se através dos resultados obtidos no *benchmark* SPLASH-2 que, na maioria das aplicações onde substitui-se os *locks* por transações, reduziu-se os acessos à *cache* e à memória principal, reduzindo assim o consumo de energia e melhorando o desempenho. Entretanto, por considerar que as aplicações do SPLASH-2 possuem taxas de conflitos baixa, foi utilizado um *microbenchmark*, com duas configurações para simular diferentes cenários de conflitos. Nas duas configurações do *microbenchmark*, ambas as implementações de memória transacional (*backoff* e serial) superaram em termos de desempenho e eficiência energética à versão utilizando *locks*, cerca de 10x mais, no melhor caso.

Em uma das configurações do *microbenchmark*, a memória transacional que utilizou execução serial para os casos de conflito superou em termos de performance e eficiência energética a memória transacional que utilizou a técnica

de *backoff* para resolução de conflitos. Porém, em outra configuração, a versão transacional serial mostrou-se ligeiramente inferior em relação à versão transacional utilizando *backoff*, em termos de desempenho. Desta forma, o modo de execução serial das transações conflitantes mostra-se conservador, incorrendo em penalidades no desempenho.

Através dos resultados mostrados neste trabalho verificou-se que a abordagem de HTM é promissora em termos de desempenho e energia. A execução serial para transações conflitantes em ambientes de alta contenção é eficiente em termos de energia, porém, para outros casos, a produtividade do sistema pode ser reduzida. No entanto, são dados poucos detalhes em relação ao *microbenchmark* e suas duas configurações e a avaliação do modo serial é feita somente neste *microbenchmark*, tornando novamente os resultados pouco atraentes.

3.3 Caracterizando o consumo de energia em STM

O trabalho de Baldassin et al (2009) é pioneiro em caracterizar o consumo de energia em STM. O processo de caracterização foi conduzido em um ambiente simulado amplamente usado na literatura, conhecido como MPARM (LOGHI; PONCINO; BENINI, 2004). Em relação à plataforma utilizada duas observações fazem-se necessárias. Primeiramente, as memórias empregadas (tanto privada como compartilhada) são baseadas na tecnologia SRAM (*Static Random Access Memory*). Segundo, os acessos feitos à memória compartilhada não são retidos na *cache*, visto que a implementação do barramento usado não possui suporte para coerência de *cache*.

As aplicações utilizadas na caracterização fazem parte do pacote de *benchmark* STAMP (CAO MINH et al., 2008), disponibilizado pela Universidade de Stanford. Foram utilizadas as oito aplicações presentes no pacote, com diferentes configurações, visando representar diversos cenários transacionais com diferentes tamanhos de transação, tempos em transação, tamanho do conjunto de leitura e escrita, e graus de contenção. De igual modo, uma versão sequencial (não utilizando transações) foi executada para comparação à STM.

A implementação da STM utilizada nos experimentos foi a TL2 (DICE; SHALEV; SHAVIT, 2006), configurada de dois modos (ambos usados nos experimentos): TL2-*lazy* (versionamento atrasado e detecção atrasada) e TL2-*eager* (versionamento adiantado e detecção adiantada). Duas estratégias do gerenciamento de contenção são providas, baseadas na técnica conhecida como *backoff*: após três cancelamentos consecutivos, a transação cancelada é postergada por um tempo (exponencial ou linear) proporcional ao número total de seus cancelamen-

tos.

O procedimento de medição de energia foi dividido pelos componentes básicos de uma STM, possibilitando assim, avaliar quais os elementos mais custosos e orientar o projeto do algoritmo de forma a otimizá-lo.

Os experimentos realizados foram executados na plataforma de simulação com dois cenários diferentes. Inicialmente, utilizou-se apenas um processador nos experimentos, e posteriormente, para que os custos devidos aos cancelamentos das transações fossem contabilizados, foram utilizados 8 processadores.

Mostrou-se através dos resultados dos experimentos que a implementação que utiliza versionamento adiantado (TL2-*eager*) tem um custo de energia ligeiramente menor do que a técnica com versionamento atrasado (TL2-*lazy*). No entanto, em relação à execução sequencial, para algumas aplicações, o custo total introduzido pelas transações chegou perto de 5x na utilização de um processador.

Na utilização de 8 processadores, além das diferentes técnicas de versionamento de dados, as duas configurações de *backoff* (exponencial e linear) foram contabilizadas. De modo geral, observou-se que para aplicações com alta taxa de cancelamento, nos casos em que o esquema de espera linear é usado, o custo de cancelamentos domina. Porém, no emprego do tempo de espera exponencial, o custo de *backoff* passa a prevalecer. Em uma aplicação específica do pacote de *benchmark*, utilizando-se TL2-*lazy* e tempo de espera exponencial, o custo total de energia chegou a cerca de 50x o da versão sequencial.

Com base nos resultados providos pela caracterização, devido ao alto consumo de energia em casos de alta contenção, os autores propuseram a utilização de uma estratégia que visa reduzir o consumo em casos em que estados de espera são induzidos por cancelamentos de transações, podendo ser usada em qualquer STM na qual o gerenciador de contenção adote políticas de resolução de conflitos baseada em espera. A estratégia proposta é baseada na técnica conhecida como *Dynamic Voltage and Frequency Scaling* (DVFS) (KAXIRAS; MARTONOSI, 2008). Através desta técnica, a voltagem e frequência de operação dos núcleos de processamento podem ser reduzidas em tempo de execução, de acordo com a demanda computacional, obtendo-se assim, a redução do consumo de energia em ordem quadrática (WECHSLER, 2006; KAXIRAS; MARTONOSI, 2008).

A estratégia funciona da seguinte maneira: antes de entrar no modo de *backoff*, o processador correspondente é colocado em modo de baixo consumo de potência, através da redução de sua frequência e voltagem. O processador então cumpre seu tempo em estado de espera (linear ou exponencial) em modo de baixa potência. Quando o tempo de *backoff* termina, a frequência e voltagem

originais são restauradas. O custo para alternância entre diferentes estados no processador é pequeno, requerendo apenas um ciclo por mudança, o que não aproxima-se das implementações atuais em processadores reais.

Os experimentos que utilizaram a estratégia descrita obtiveram uma redução efetiva do consumo de energia para as aplicações com alto grau de contenção, em média de 45%, chegando a 87% de redução para uma aplicação específica.

3.4 Análise de desempenho e consumo de energia de STM e *locks*

O trabalho de Klein et al (2010) avalia o consumo de energia e desempenho de STM em comparação a *locks*, utilizando a mesma plataforma de simulação, pacote de *benchmark*, metodologia e implementação de STM, apresentada no trabalho de Baldassin et al (2009), descrito na Seção 3.3.

As aplicações baseadas em *locks* foram implementadas como um típico mecanismo de *spinlock*. Foram substituídas as marcações de transações pelas operações de aquisição e liberação do *lock* global, pois o *benchmark* utilizado não provê uma versão baseada em *locks* para as aplicações.

Os experimentos foram realizados em dois cenários, em relação à implementação baseada em *locks*. No primeiro cenário, não foram contabilizados os ciclos e energia desperdiçados devido à espera ocupada. No segundo, os ciclos e energia gastos na espera ocupada foram contabilizados, através de penalidades em ciclos (variando de 1k a 10k ciclos) quando o bloqueio não pode ser adquirido prontamente.

Algumas afirmações podem ser feitas com base nos resultados obtidos. No primeiro cenário, a abordagem que utilizou *locks* superou a STM em energia e desempenho, em praticamente todas aplicações. No entanto, para algumas aplicações, a STM mostrou-se competitiva em termos de energia e desempenho, em especial para aplicações com taxas de conflitos baixa. Em aplicações com altas taxas de conflitos, a utilização de STM foi em média 3x menos eficiente em relação aos *locks*, chegando cerca de 22x em uma aplicação específica. No segundo cenário, de modo geral, o desempenho da STM mostrou-se inferior em relação à utilização de *locks*. No entanto, a diferença entre o desempenho de ambas abordagens poderia ser reduzido quando incorrer penalidades altas para o mecanismo de *locks*. Além disso, mostrou-se que para algumas aplicações transacionais, com baixas taxas de conflito, a utilização de STM pode se tornar atraente dependendo dos custos relacionados às penalidades dos *locks*. Para aplicações com alta contenção, a utilização de *locks* mostra-se a melhor abordagem, mesmo que elevadas penalidades sejam impostas.

3.5 Avaliação de consumo de energia e desempenho de HTM em sistemas embarcados multiprocessados

O trabalho de Kunz (2010) avalia o consumo de energia e desempenho do modelo LogTM em comparação ao mecanismo de *locks*, em um sistema embarcado multiprocessado. Além disso, avalia o desempenho e consumo de energia de diferentes políticas de gerenciadores de contenção.

A interconexão entre os núcleos de processamento foi baseada em uma arquitetura de rede chamada NoC (*Network on Chip*), sendo esta uma alternativa às interconexões tradicionais como barramentos e chaves *crossbar* (BENINI; DE MICHELI, 2002; BJERREGAARD; MAHADEVAN, 2006). Uma NoC é uma estrutura de comunicação formada por diversos roteadores conectados entre si de acordo com determinada topologia. Cada roteador é associado a algum elemento do conjunto da rede (processadores, memórias, entre outros). As NoCs permitem elevado grau de paralelismo na comunicação, pois os enlaces da rede podem operar de modo simultâneo sobre diferentes pacotes de dados. Apesar das soluções tradicionais (barramentos e *crossbar*) apresentarem vantagens em termos de simplicidade, compatibilidade e latência, os limites físicos impostos pelo fio, questões relacionadas à escalabilidade e largura de banda apontam para a NoC como a melhor alternativa para futuras gerações de processadores *many-core* (BJERREGAARD; MAHADEVAN, 2006).

O sistema computacional utilizado na realização dos testes utilizou-se de 2, 4, 8, 16 e 32 processadores, todos operando com frequência de 100 MHz. Os tamanhos de *cache* L1 usadas foram de 512, 1024, 2048 e 8192 *bytes*, sendo estas completamente associativas e baseadas no protocolo de coerência MSI baseado em diretório. A configuração da NoC é baseada no roteamento XY, chaveamento *Wormhole*, topologia Grade (*mesh*), arbitragem *Round-robin* e frequência de operação de 100 MHz.

A implementação e os experimentos realizados neste trabalho foram feitos na plataforma virtual SIMPLE (BARCELOS; BRIÃO; WAGNER, 2007). O consumo de energia foi obtido levando-se em consideração a energia da rede, memórias, *caches* e processadores do sistema. A energia da NoC foi calculada através da biblioteca Orion (WANG et al., 2002), enquanto que a energia das memórias e das *caches* foi calculada com a ferramenta Cacti (WILTON; JOUPPI, 1996).

Para os experimentos em aplicações de baixa contenção, três *benchmarks* foram utilizados: Multiplicação de Matrizes, Codificador JPEG e Estimação de Movimento. Para o cenário de alta contenção, utilizou-se a aplicação LeeTM, que utiliza alta demanda de sincronização durante a maior parte de sua execução (ANSARI et al., 2008). Esta aplicação possui paralelismo intrínseco, no

entanto, este paralelismo é difícil de expressar utilizando-se *locks*. Para avaliar o comportamento do sistema para o caso em que não há paralelismo disponível para as transações, desenvolveu-se neste trabalho uma modificação da LeeTM para forçar artificialmente a contenção. Com a modificação realizada, a quantidade de paralelismo que a TM pode explorar é a mesma que a versão baseada em *locks*, o que permite analisar o *overhead* de performance e energia de cada mecanismo (TM e *locks*) para o pior caso em termos de paralelismo desta aplicação. A aplicação LeeTM com inibição do paralelismo é chamada de LeeTM-IP.

Nos experimentos realizados para aplicações de baixa contenção, o gerenciador de contenção da memória transacional utilizada baseou-se na técnica de *backoff* com valores fixos. O tempo de espera usado foi de 5000 ciclos para 2, 4 e 8 processadores, 10000 ciclos para 16 processadores e 25000 ciclos para 32 processadores. Através dos resultados obtidos para o cenário de baixa contenção, mostrou-se que, na maioria dos casos, para até 4 processadores, a memória transacional apresentou-se ligeiramente inferior em relação aos *locks*, em termos de desempenho. Na utilização a partir de 6 processadores obteve-se um aumento de performance da memória transacional à medida que o número de processadores aumentava, superando os *locks* em todos os casos, chegando a 30% no melhor caso. Em relação à energia consumida, nas três aplicações e em todas configurações testadas, a memória transacional foi eficiente em comparação aos *locks*, apresentando reduções de energia em todos os casos. De maneira geral, os resultados de energia seguem os ganhos de performance à medida que o número de processadores do sistema foi aumentado, e, em muitos casos, os ganhos de energia ultrapassaram os de desempenho, indicando que a potência média também foi reduzida.

No cenário de alta contenção, como primeiro experimento, foi feita uma análise para encontrar o valor ótimo para o tempo de espera de *backoff* (fixo, exponencial e linear aleatório) para diferentes configurações do sistema (número de processadores), utilizando-se da aplicação LeeTM. Para isso, encontrou-se o parâmetro que resulta no menor tempo de execução para cada política de *backoff* de cada configuração do sistema.

Como alternativa ao encontro da melhor heurística para ajustar o tempo de espera de *backoff*, os autores propuseram uma nova estratégia de gerenciamento de contenção, chamada de *Abort Handshake*. A solução proposta é baseada em sinalização entre transações de forma que a transação cancelada seja notificada quando for seguro reiniciar para evitar que ocorra o mesmo conflito. De modo a exemplificar o funcionamento, quando a transação `T1` aborta após conflitar com a transação `T0`, `T1` envia uma mensagem sinalizando seu cancelamento à `T0` e aguarda até receber o sinal de `T0` notificando o momento de sua

reexecução. `T0` então adiciona `T1` em sua lista de transações abortadas. Quando finalizar a execução, `T0` envia o sinal para `T1` permitindo que ela reinicie a execução. A fim de evitar situações de *deadlock*, uma transação pode esperar apenas por uma transação de maior prioridade. Além disso, uma transação que tenha abortado outra pode ser abortada por uma terceira transação; uma irá notificar a outra após o *commit* e assim sucessivamente.

Esta solução serializa por completo a execução de duas transações quando uma delas aborta ao permitir que esta reinicie apenas após o *commit* da outra. É uma solução conservadora, já que a transação que abortou pode ter trabalho para executar de forma independente antes de solicitar o bloco conflitante novamente. Além disso, é possível que a transação abortada nem mesmo solicite o bloco conflitante novamente se a decisão de requisitar o bloco depender de dados alterados por uma terceira transação neste meio tempo.

Em seguida, realizou-se um experimento em ambiente de alta contenção comparando as aplicações LeeTM e LeeTM-IP, utilizando *locks*, memória transacional com o mecanismo *Abort Handshake* e memória transacional com os valores ótimos para cada política de *backoff* (fixo, exponencial e linear randômico). Como resultados deste experimento verificou-se que não houve vantagens em termos de desempenho e energia na versão das aplicações utilizando *locks* para mais de 4 processadores. Nenhuma das políticas de *backoff* pode ser escolhida como vencedora, pois todas tiveram resultados parecidos e a melhor depende de cada caso. Em situações de alta contenção, o *backoff* exponencial aumenta muito o tempo de espera das transações, prejudicando significamente a performance e elevando o consumo de energia.

O mecanismo *Abort Handshake* apresentou ganhos de performance e energia na maioria dos resultados, atingindo reduções do tempo de execução em 20% e redução de energia de 53%, se comparados com a melhor política de *backoff* em cada caso. Entretanto, para alguns casos, algumas peculiaridades podem afetar o desempenho do *Abort Handshake*. Embora que para 2 processadores o mecanismo *Abort Handshake* foi inferior em termos de energia e desempenho em relação a todas as políticas de *backoff*, a partir de 4 processadores, o *Abort Handshake* apresentou ganhos significativos de desempenho e energia, na maioria dos casos. No entanto, observou-se que na maioria dos resultados, reduziu-se drasticamente o desempenho e elevou-se o consumo de energia do mecanismo *Abort Handshake*, em comparação a *locks* e a memória transacional com *backoff*, na utilização de 32 processadores. Os autores apontaram a necessidade de analisar a eficiência do mecanismo *Abort Handshake* utilizando-se mais de 32 processadores.

3.6 Comparação entre implementações de semáforos e memórias transacionais em relação ao consumo de energia

O trabalho de Mittal e Nitin (2011) avalia o consumo de energia e taxa de *cache miss* de semáforos (*spinlock* e bloqueante) e memórias transacionais, na memória *cache* compartilhada.

Utilizou-se uma arquitetura computacional *multicore*, com 4 *cores*, cada um com sua memória privada e duas memórias *caches* compartilhadas. A arquitetura foi simulada com base no conjunto de ferramentas SimpleScalar (BURGER; AUSTIN, 1997). Três *benchmarks* foram utilizados como aplicações bases para os testes: Árvores Rubro-Negras, Transformada Rápida de Fourier e SPLASH-2 (WOO et al., 1995).

De modo geral, os resultados obtidos nos testes realizados mostraram que a sincronização por semáforos bloqueantes obteve menos *cache miss* em comparação a *spinlocks* e transações. Em termos de consumo de energia, memórias transacionais e semáforos bloqueantes têm um consumo equivalente, porém muito eficiente em comparação a *spinlocks*.

Apesar de indicar que a sincronização por semáforos bloqueantes apresenta melhores taxas de *cache miss* em comparação a *spinlocks* e transações, e que em termos de energia se equipara a memórias transacionais, o trabalho apresenta alguns problemas que tornam seus resultados pouco atraentes. Não são descritas quais aplicações do *benchmark* SPLASH-2 foram utilizadas, utilizou-se apenas uma configuração da arquitetura computacional (4 *cores*), e não foi explicitado o tipo de memória transacional usada.

3.7 Análise do consumo de energia e desempenho da implementação TinySTM

O trabalho de Baldassin et al (2012) analisa o consumo de energia e desempenho da implementação TinySTM, em uma plataforma computacional simulada.

Para avaliar a eficiência de diferentes estratégias de implementação do sistema transacional, foram utilizadas três configurações providas pela TinySTM: CTL (versionamento de dados e detecção de conflito atrasado), ETL-*lazy* (versionamento de dados atrasado e detecção de conflito adiantado) , ETL-*eager* (versionamento de dados e detecção de conflito adiantado). Além disso, para cada configuração, três gerenciadores de contenção foram utilizados, tímido, *backoff* e *delay*.

Utilizou-se a plataforma de simulação MPSoC (LOGHI; PONCINO; BENINI, 2004), configurada com processador ARMv7 de 200 MHz com 8 *cores*, *cache*

de instrução L1 com 8 KB, *cache* de dados L1 com 4 KB, memória privada e compartilhada SRAM de 16 MB cada, e barramento AMBA AHB. A avaliação das diferentes configurações de STM baseou-se no *benchmark* STAMP, com argumentos recomendados para ambiente simulado, utilizando-se exatamente os 8 *cores* disponíveis na plataforma configurada.

De modo geral, a configuração ETL-*lazy* e ETL-*eager* foram similares, e apresentaram os melhores resultados, tanto em termos de consumo de energia como desempenho. Entre as três políticas de resolução de conflito analisadas, a estratégia tímido apresentou os melhores resultados, na maioria dos casos.

Devido às estratégias de resolução de conflito *backoff* e *delay* não terem apresentado bons resultados nos experimentos realizados, os autores aplicaram a técnica DVFS em conjunto a estes gerenciadores de contenção, de modo semelhante ao trabalho apresentado por Baldassin et al (2009), descrito na Seção 3.3. O principal objetivo foi reduzir o consumo de energia através do tempo que a transação aguarda para reexecutar. A integração da técnica DVFS com os gerenciadores de contenção *backoff* e *delay* mostrou-se eficiente, apresentando um ganho de EDP (*Energy-Delay Product*) máximo e médio de 59% e 16%, respectivamente. Em particular, mostrou-se que aplicações que apresentam alta contenção e que possuam transações longas são as mais beneficiadas pelo uso de DVFS.

3.8 Observações finais

Este capítulo apresentou as principais pesquisas que avaliam o consumo de energia e/ou desempenho de aplicações, relacionadas a mecanismos de sincronização em ambiente de memória compartilhada.

Diferentes técnicas e implementações de mecanismos de sincronização foram avaliadas através dos sete trabalhos descritos anteriormente. Nesta perspectiva, as seguintes constatações podem ser feitas, com base nos resultados obtidos. Observou-se que o nível de contenção influencia consideravelmente o desempenho e consumo de energia das aplicações. De modo geral, altas taxas de contenção incorrem em custos adicionais em termos de energia e penalidades em desempenho nas memórias transacionais, devido à resolução de conflito das transações conflitantes e suas reexecuções. Para o cenário de alta contenção, as estratégias utilizadas pelo gerenciador de contenção em transações garantem a efetivação de reduções de custos adicionais de energia e penalidades no desempenho.

Em relação à comparação entre os mecanismos de sincronização, as implementações de STM, de modo geral, apresentaram desempenho inferior e maior

consumo de energia se comparadas ao uso de *locks*, em cenários de alta contenção. No entanto, em aplicações com baixo nível de contenção, as implementações de STM mostraram-se competitivas, superando em desempenho e eficiência energética a sincronização por *locks* em muitos casos. As implementações de HTM, por outro lado, mostraram-se eficientes tanto em cenários de alta quanto baixa contenção, superando a abordagem *locks* em muitas vezes em termos de eficiência energética e desempenho, na maioria dos casos.

O presente trabalho difere-se dos demais apresentados neste Capítulo em três aspectos principais. Primeiro, este trabalho analisa o consumo de energia e desempenho de bibliotecas de STM em ambiente de computação real. Outros trabalhos analisaram o consumo de energia de implementações de STM, no entanto, realizaram esta análise em ambiente computacional simulado, conforme pode ser observado nas seções 3.3, 3.4 e 3.7. Segundo, do melhor de nosso conhecimento, este trabalho é o primeiro a analisar o consumo de energia das implementações SwissTM e AdaptSTM. Terceiro, avalia-se a eficiência das STMs variando-se de 1 até 64 *threads*. Tal avaliação é de suma importância, pois verifica-se a escalabilidade das implementações, em especial em cenário de alta contenção.

No capítulo a seguir concentra-se o cerne desta dissertação, onde serão detalhados a metodologia e os resultados obtidos na avaliação do consumo de energia e desempenho das bibliotecas de STM utilizadas.

4 ANÁLISE DE CONSUMO DE ENERGIA E DESEMPENHO

É inegável a importância de técnicas para redução do consumo de energia em sistemas embarcados e, atualmente, esta tendência se espalhou também para *data centers* e sistemas *desktop* (BALDASSIN, 2009). No entanto, a análise do consumo de energia de implementações de STM é pouco explorada e, quando realizada, é baseada em ambientes computacionais simulados (BALDASSIN et al., 2009; KLEIN et al., 2009, 2010; BALDASSIN et al., 2012).

Nesse contexto, este capítulo apresenta a análise e caracterização do consumo de energia e desempenho de quatro bibliotecas de STM, em ambiente de computação não-simulado.

Na próxima seção, será descrita a metodologia empregada na realização dos experimentos. Na Seção 4.2 serão mostrados e discutidos os resultados obtidos. Por fim, na Seção 4.3 serão apresentadas as observações finais do capítulo.

4.1 Metodologia

Com objetivo de avaliar diferentes configurações de STM, quatro bibliotecas foram utilizadas, TL2 (versão 0.9.6), TinySTM (1.0.3), SwissTM (20110815) e AdaptSTM (0.5.2). Escolheu-se estas quatro implementações por comporem o estado da arte em STM, apresentarem características diferentes no projeto do sistema transacional e serem disponibilizadas para uso gratuitamente. Também executou-se uma versão sequencial das aplicações do *benchmark*, com a finalidade de analisar o *overhead* e a eficiência das STMs em relação à execução sequencial.

Os experimentos foram realizados em um servidor SGI Altix, com configurações apresentadas na Tabela 2. Cada configuração de execução (STM ou sequencial, aplicação de *benchmark*, quantidade de *threads*) foi executada 10 vezes. Após o término de execução os dados de consumo de energia e desempenho foram integrados e as médias calculadas. O desvio padrão foi pequeno

em relação à média, na maioria dos casos, conforme pode ser observado nos anexos C, D e E.

Tabela 2: Configuração do servidor utilizado nos experimentos.

Modelo do processador	Intel Xeon E5620
Frequência (GHz)	2.4
Cores	4 físicos e 4 lógicos
***Cache* (MB)**	12
Memória DDR3 (GB)	6
HD (GB)	500
Placa de rede	Realtek 8201N
Sistema operacional	SUSE Linux SP11
Compilador	G++ 4.5.2
Consumo de potência em ociosidade (W)	132

Na subseção a seguir será detalhado o *benchmark* utilizado na realização dos experimentos. Na Subseção 4.1.2 será descrito o modo realizado para coleta do consumo de energia.

4.1.1 *Benchmark* STAMP

A avaliação das STMs baseou-se nas aplicações do *benchmark* STAMP (*Stanford Transactional Applications for Multi-Processing*) versão 0.9.10, com os parâmetros de execução sugeridos para sistemas não-simulados (CAO MINH et al., 2008).

Este *benchmark* consiste em um conjunto de oito aplicações desenvolvidas para uso específico em pesquisas relacionadas a avaliações de TMs. Estas aplicações são provenientes de diferentes domínios de pesquisa, possuem possibilidade de configuração e são paralelizáveis em escalas diferentes. As aplicações são escritas em linguagem C, em duas versões: sequencial e paralela/transacional. A versão transacional é basicamente a sequencial marcada com macros para indicar o início e fim das transações.

A Tabela 3 resume, para cada aplicação, os argumentos de linha de comando usados nos experimentos, o domínio da aplicação e uma breve descrição sobre sua funcionalidade.

Conforme pode ser observado na Tabela 3, executou-se todas as aplicações com os maiores tamanhos possíveis de entrada. Além dos argumentos mostrados nesta tabela, todas as aplicações recebem como parâmetro o número de *threads* na qual o código deverá ser executado.

As características das aplicações do STAMP que influenciam a execução utilizando TMs são: tamanho da transação, conjunto de leitura e escrita, tempo em transação e o nível de contenção. Na Tabela 4 são resumidas estas características.

Tabela 3: Aplicações do *benchmark* STAMP - argumentos usados nos experimentos, domínio e breve descrição. Fonte: (CAO MINH et al., 2008)

Aplicação	Argumentos	Domínio	Breve descrição
Bayes	-v32 -r4096 -n10 -p40 -i2 -e8 -s1	aprendizado de máquina	aprendizado de estrutura de rede Bayesiana
Genome	-g16384 -s64 -n16777216	bioinformática	sequenciamento de genes
Intruder	-a10 -l128 -n262144 -s1	segurança	detecção de ataques em redes
Kmeans	-m40 -n40 -t0.00001 -i random-n65536-d32-c16	mineração de dados	algoritmo de clusterização
Labyrinth	-i random-x512-y512-z7-n512	engenharia	algoritmo de roteamento
SSCA2	-s20 -i1.0 -u1.0 -l3 -p3	científico	conjunto de operações em grafos
Vacation	-n2 -q90 -u98 -r1048576 -t4194304 (baixa cont.) -n4 -q60 -u90 -r1048576 -t4194304 (média cont.)	processamento de transações *online*	sistema para reservas de passagens
Yada	-a15 -i ttimeu1000000.2	científico	algoritmo para refinamento de malhas Delaunay

Tabela 4: Características das aplicações do *benchmark* STAMP. Fonte: (CAO MINH et al., 2008)

Aplicação	Tamanho transação	Conjunto leitura/escrita	Tempo em transação	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

Utilizou-se o *benchmark* STAMP por ser disponibilizado para uso gratuito, apresentar aplicações realistas e robustas e pelas STMs utilizadas possuírem integração com este. Além disso, este *benchmark* é um dos mais utilizados atualmente na avaliação de TMs.

4.1.2 Medição do consumo de energia

O consumo de energia foi coletado através de um microcontrolador especializado embutido na placa-mãe, presente na maioria dos servidores, denominado *Baseboard Management Controller* (BMC) (ZHUO; YIN; RAO, 2004). Este microcontrolador é capaz de capturar os dados de diversos sensores da placa-mãe como parâmetros de temperatura do sistema, velocidade do ventilador, estado do sistema, voltagem, consumo de energia, dentre outros.

Durante cada execução, em média 33 amostras de potência por segundo foram capturadas pelo BMC, e foram acessadas através da interface *Intelligent Platform Management Interface* (IPMI), utilizando-se a biblioteca FreeIPMI versão 1.1.3 (FREEIPMI CORE TEAM, 2013).

Após o término de execução, os dados de consumo de energia e desempenho

foram integrados e as médias calculadas.

4.2 Resultados

Nas próximas oito subseções são mostrados e discutidos os resultados obtidos para cada aplicação. Na Subseção 4.2.9 são apresentadas as relações de consumo de energia e desempenho, com base nos resultados obtidos. Na Subseção 4.2.10 são discutidos os principais aspectos relacionados à escalabilidade das STMs utilizadas. Por fim, na Subseção 4.2.11 são apresentadas algumas discrepâncias entre os resultados obtidos e as características das aplicações do *benchmark* STAMP.

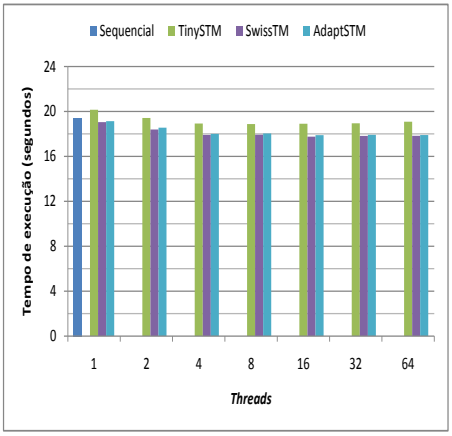
4.2.1 Resultados da aplicação Bayes

Na aplicação Bayes utilizando-se a biblioteca TL2 excedeu-se a memória disponível no sistema. Devido a este fato, experimentou-se também utilizar os parâmetros de execução médio nesta aplicação. No entanto, utilizando-se estes parâmetros, o tempo médio da fração paralela do código foi de 0,005 segundos, muito abaixo do tempo de execução total (3,5 segundos). Desta forma, a análise da eficiência das STMs ficaria comprometida nesta configuração, devido ao baixo tempo da fração paralela do código. Nesta perspectiva, optou-se por utilizar os maiores parâmetros e omitir os resultados da TL2 nesta aplicação.

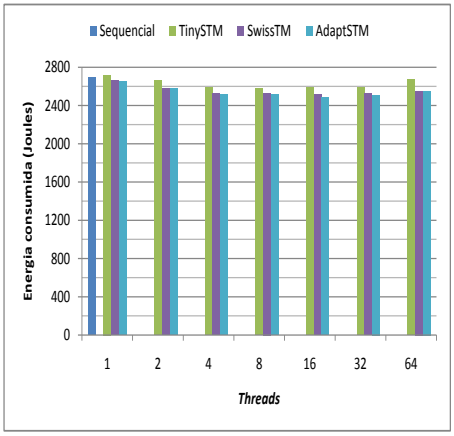
Nota-se que o tempo de execução sequencial foi maior do que a AdaptSTM e SwissTM utilizando-se uma *thread*, conforme pode ser observado na Figura 9. A explicação para este caso decorre do fato da aplicação Bayes possuir transações longas, o que amortiza o *overhead* das primitivas transacionais.

A aplicação Bayes não apresentou bons resultados de *speedup*. Independente da configuração de execução utilizada o *speedup* máximo foi cerca de 1,09. Todavia, o *speedup* não depende somente da biblioteca de STM, mas também das características da aplicação que está utilizando o sistema transacional, conforme descrito na Subseção 4.1.1. Um fator que justifica em parte o baixo desempenho é o fato de algumas aplicações não serem totalmente paralelas. Especificamente na aplicação Bayes, o tempo médio da fração paralela do código foi de 1 s, muito abaixo do tempo de execução total.

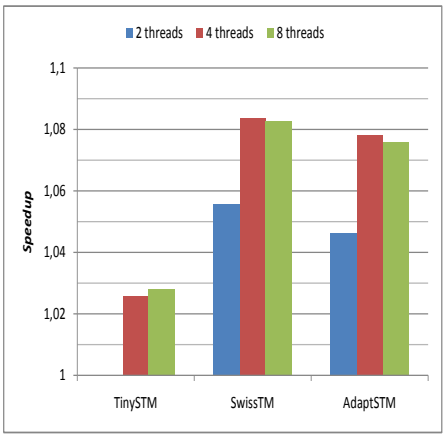
De modo geral, a SwissTM e a AdaptSTM foram similares, e apresentaram os melhores resultados nesta aplicação, tanto em termos de consumo de energia quanto desempenho. A TinySTM, por sua vez, apresentou consumo de energia e desempenho ligeiramente maior que as demais STMs, pois obteve-se as maiores taxas de cancelamento, conforme pode ser observado na Tabela 5. Este resultado é devido principalmente ao uso exclusivo do gerenciador de contenção



(a) Tempo de execução



(b) Energia consumida



(c) *Speedup*

Figura 9: Resultados da aplicação Bayes

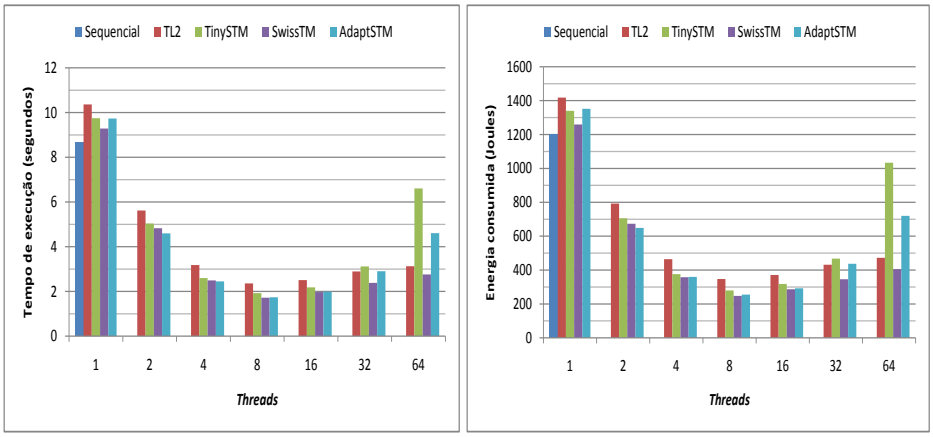
tímido nesta biblioteca. Este gerenciador de contenção cancela a transação que detectou o conflito e a reexecuta imediatamente. De modo a evitar cancelamentos consecutivos, algumas STMs empregam a estratégia de *backoff*. No entanto, este mecanismo não é utilizado pela TinySTM.

Tabela 5: Quantidade de cancelamentos na aplicação Bayes.

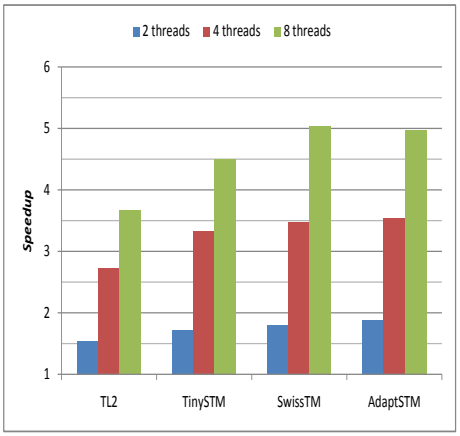
STM	**2 *threads***	**4 *threads***	**8 *threads***	**16 *threads***	**32 *threads***	**64 *threads***
TinySTM	0	1	605913	5035206	9328996	25806090
SwissTM	0	1	5	4454	18966	21304
AdaptSTM	0	2	3	1304	602	1475

4.2.2 Resultados da aplicação Genome

De modo geral, as quatro bibliotecas de STM apresentaram resultados similares na aplicação Genome utilizando-se até 32 *threads*, conforme pode ser observado na Figura 10.



(a) Tempo de execução (b) Energia consumida



(c) *Speedup*

Figura 10: Resultados da aplicação Genome

Na utilização de 64 *threads*, verifica-se que a AdaptSTM apresentou quantidade de cancelamento ligeiramente menor que a SwissTM, conforme pode ser observado na Tabela 6. No entanto, a SwissTM apresentou o melhor resultado de desempenho e consumo de energia neste caso. O motivo da eficiência da SwissTM neste cenário deve-se ao uso do gerenciador de contenção guloso, o qual é mais eficiente em relação ao tímido em transações médias/longas, pois a transação que está executando a mais tempo tem prioridade sobre as outras.

Além disso, o tempo de *backoff* empregado pela SwissTM (linear) é menor do que o utilizado pela AdaptSTM (exponencial).

Tabela 6: Quantidade de cancelamentos na aplicação Genome.

STM	2 *threads*	4 *threads*	8 *threads*	16 *threads*	32 *threads*	64 *threads*
TL2	4500	11563	31899	76970	173772	227141
TinySTM	6992	49582	221517	2771439	6702914	20762005
SwissTM	923	3804	14376	22985	81278	224546
AdaptSTM	1896	3077	10068	26301	83524	180780

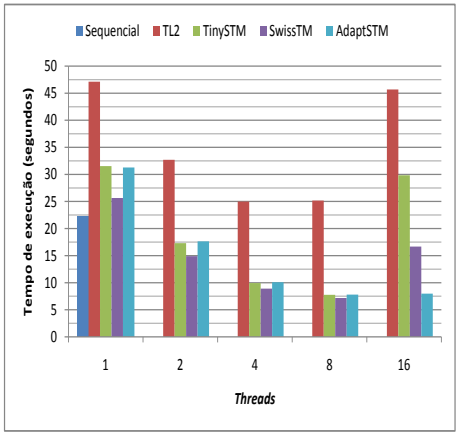
4.2.3 Resultados da aplicação Intruder

Em relação à aplicação Intruder, observa-se que as quatro STMs obtiveram pouca melhora de desempenho na passagem de 4 para 8 *threads*, conforme pode ser observado na Figura 11. Esse comportamento é devido à alta contenção intrínseca nesta aplicação.

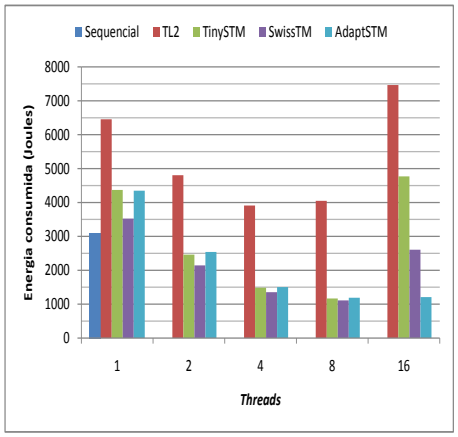
O aumento do número de *threads* incorre em maiores taxas de conflitos entre as transações, na maioria dos casos. Este resultado afeta negativamente o desempenho e o consumo de energia, pois maior será o desperdício de processamento devido à reexecução das transações canceladas.

A AdaptSTM exibiu a menor quantidade de cancelamentos em todas configurações de execução na aplicação Intruder, conforme pode ser observado na Tabela 7. Utilizando-se 64 *threads*, esta biblioteca apresentou cerca de 7x menos cancelamentos do que a TL2. Nesta perspectiva, especificamente acima de 8 *threads* a AdaptSTM apresentou resultados satisfatoriamente melhores em relação às demais STMs. A eficiência desta biblioteca nesses casos é devido ao uso do gerenciador de contenção *backoff* exponencial. Em transações curtas, o tempo de espera imposto por este mecanismo mostra-se eficiente para evitar cancelamentos consecutivos.

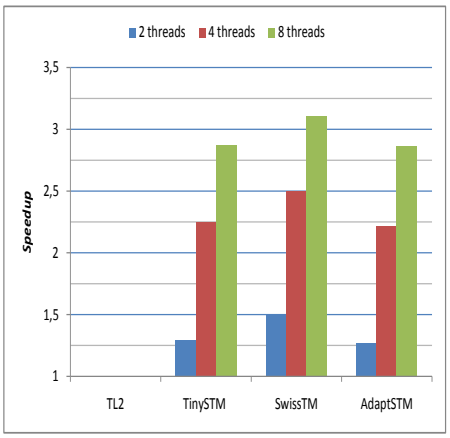
A SwissTM apresentou menor quantidade de cancelamentos em relação à TinySTM nesta aplicação, na utilização de 64 *threads*. No entanto, a SwissTM foi menos eficiente em desempenho e consumo de energia que a TinySTM, neste cenário, conforme pode ser observado na Figura 12. Este resultado é devido ao uso do *backoff* linear na SwissTM. Este mecanismo impõe um tempo de espera para a transação cancelada em conflito de escrita-escrita reexecutar. A TinySTM, por sua vez, utiliza exclusivamente o gerenciador de contenção tímido, o qual reexecuta a transação cancelada imediatamente.



(a) Tempo de execução



(b) Energia consumida

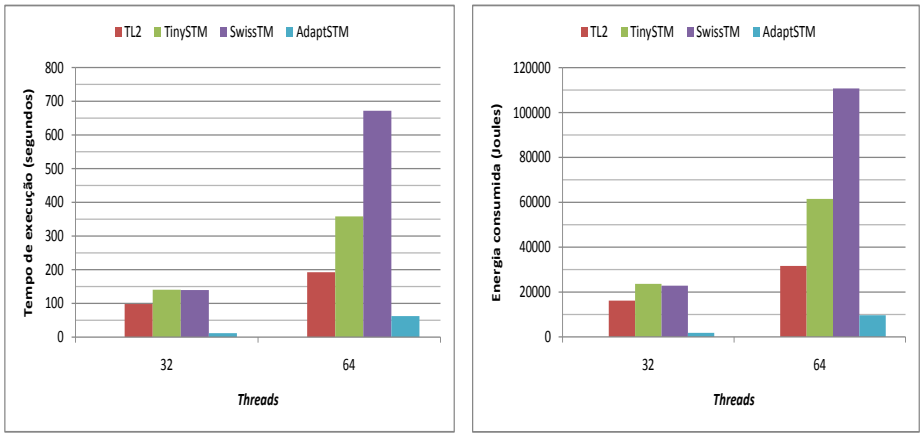


(c) *Speedup*

Figura 11: Resultados da aplicação Intruder

Tabela 7: Quantidade de cancelamentos na aplicação Intruder.

STM	**2 *threads***	**4 *threads***	**8 *threads***	**16 *threads***	**32 *threads***	**64 *threads***
TL2	9983580	22180394	28981036	37068659	51444798	77808506
TinySTM	4776343	16537412	40401667	1857277463	10786718602	24722854129
SwissTM	230474	1018706	3091874	4447138	44054329	230173952
AdaptSTM	228448	981145	2826840	2676221	3392367	10689151



(a) Tempo de execução (b) Energia consumida

Figura 12: Resultados da aplicação Intruder utilizando-se 32 e 64 *threads*

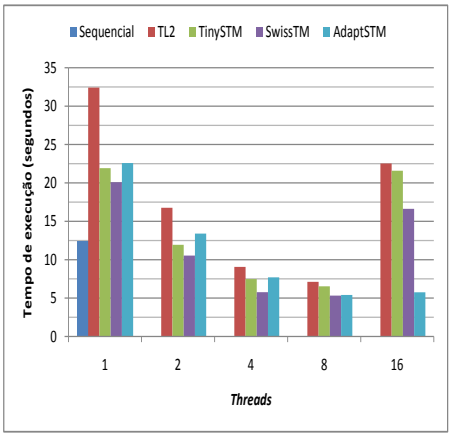
4.2.4 Resultados da aplicação Kmeans

Na aplicação Kmeans, a SwissTM apresentou o melhor desempenho entre as demais STMs, utilizando-se até 8 *threads*, conforme pode ser observado na Figura 13. Em relação ao consumo de energia, a SwissTM apresentou os melhores resultados para até 4 *threads*. Na utilização de 8 *threads*, a AdaptSTM apresentou consumo de energia ligeiramente menor que a SwissTM. Nota-se que a SwissTM obteve a menor quantidade de cancelamentos entre as demais STMs nestes cenários, conforme mostra-se na Tabela 8. Este resultado é a principal razão para a eficiência desta biblioteca nestes casos.

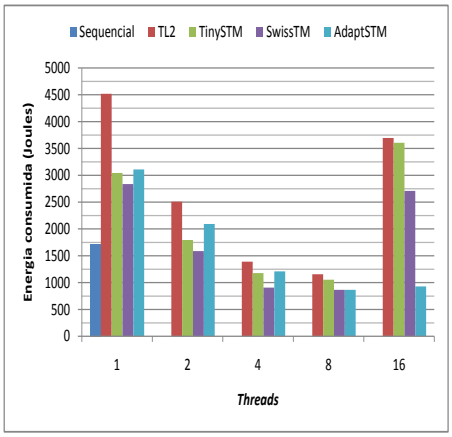
Tabela 8: Quantidade de cancelamentos na aplicação Kmeans.

STM	**2 *threads***	**4 *threads***	**8 *threads***	**16 *threads***	**32 *threads***	**64 *threads***
TL2	672379	1963606	3792486	18990268	30053948	54936329
TinySTM	566674	6546489	15312750	1679959871	5696425207	16853220360
SwissTM	77253	236174	710829	5737003	20408321	54109251
AdaptSTM	4378510	16207152	19592168	17497750	10056180	11750103

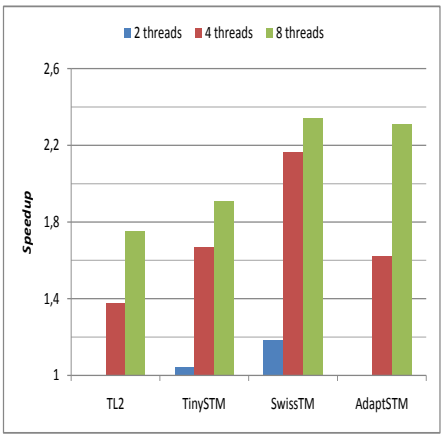
A AdaptSTM, por sua vez, apresentou quantidade de cancelamento maior do que a TL2 e a SwissTM, utilizando-se até 16 *threads*. No entanto, constata-se que conforme aumenta-se o número de *threads*, a taxa de cancelamentos da AdaptSTM cresce em menor proporção em relação aos cancelamentos apresentados pela TL2 e SwissTM, neste cenário. Este fato resulta em aumento de desempenho e eficiência energética da AdaptSTM conforme aumenta-se o número



(a) Tempo de execução



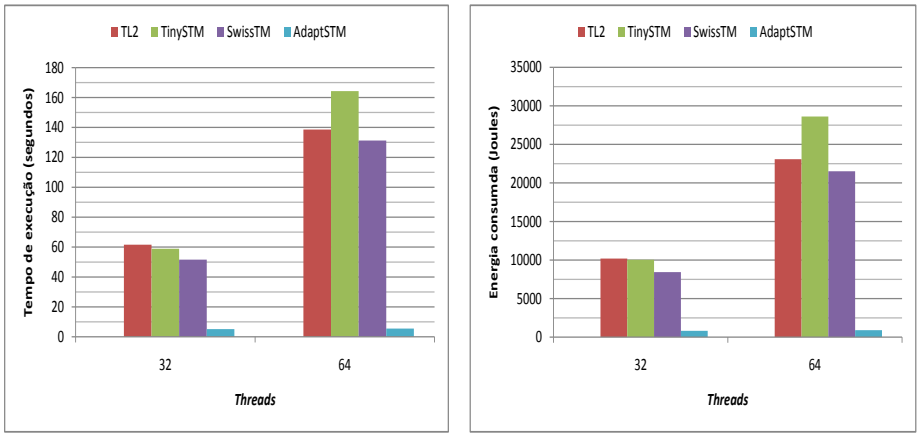
(b) Energia consumida



(c) *Speedup*

Figura 13: Resultados da aplicação Kmeans

de *threads*, conforme pode ser observado na Figura 14. Este resultado é devido ao gerenciador de contenção empregado pela AdaptSTM (*backoff* exponencial).



(a) Tempo de execução (b) Energia consumida

Figura 14: Resultados da aplicação Kmeans utilizando-se 32 e 64 *threads*

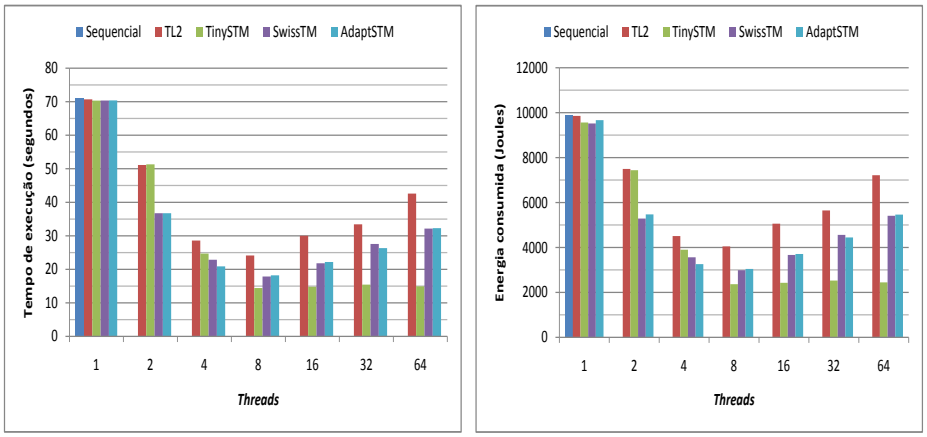
4.2.5 Resultados da aplicação Labyrinth

Conforme pode ser observado na Tabela 9, verifica-se que a quantidade de cancelamentos obtida na aplicação Labyrinth foi baixo, em comparação as outras aplicações. Apesar da TinySTM ter obtido as maiores taxas de cancelamentos (cerca de 12x mais que as demais STMs) nesta aplicação, especificamente acima de 4 *threads* esta biblioteca apresentou os melhores resultados. Nesta perspectiva, constata-se que quando o número de cancelamentos é baixo, a utilização exclusiva do gerenciador tímido empregado pela TinySTM é eficiente.

O uso de *HyperThreading* (8 *threads*) também mostrou-se eficiente para a maioria das aplicações. No melhor caso (TinySTM/Labyrinth), foi cerca de 70% mais eficiente em termos de desempenho em relação à execução com 4 *threads*, conforme pode ser observado na Figura 15.

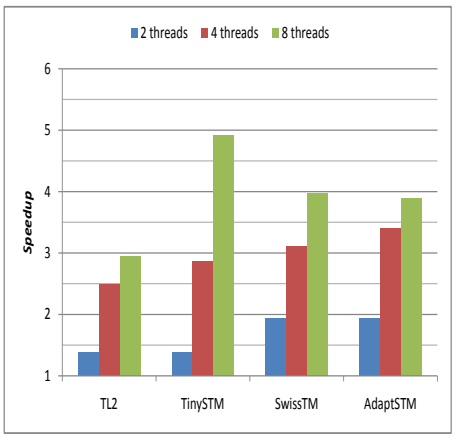
Tabela 9: Quantidade de cancelamentos na aplicação Labyrinth.

STM	**2 *threads***	**4 *threads***	**8 *threads***	**16 *threads***	**32 *threads***	**64 *threads***
TL2	39	103	202	354	397	644
TinySTM	318	980	2194	4007	5749	7037
SwissTM	19	48	103	188	279	404
AdaptSTM	19	45	102	193	277	407



(a) Tempo de execução

(b) Energia consumida



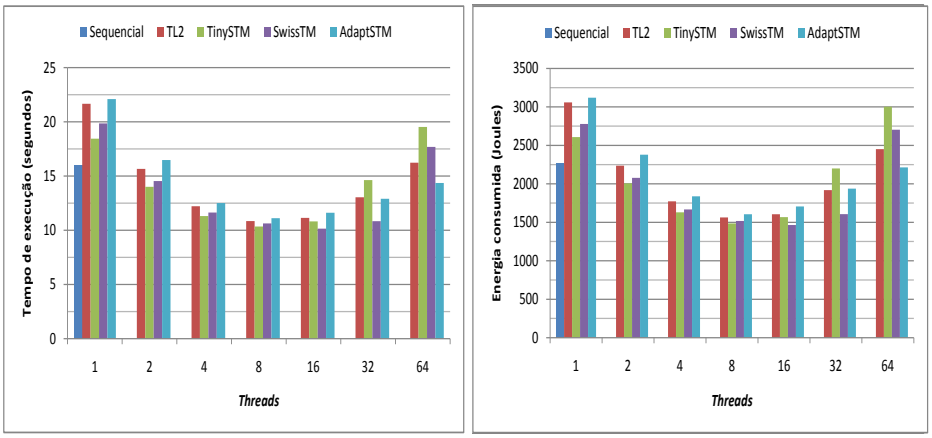
(c) *Speedup*

Figura 15: Resultados da aplicação Labyrinth

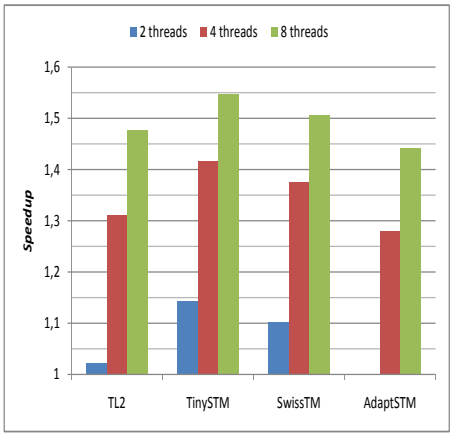
4.2.6 Resultados da aplicação SSCA2

As STMs utilizadas apresentaram baixos resultados de *speedup* em todas configurações na aplicação SSCA2, conforme pode ser observado na Figura 16. O *speedup* máximo foi cerca de 1,6. Este resultado é devido ao baixo grau de paralelismo nesta aplicação.

Verifica-se que acima de 8 *threads* aumentou-se substancialmente a quantidade de conflitos em todas STMs nesta aplicação, conforme pode ser observado na Tabela 10. Todavia, este resultado não prejudicou significamente o desempenho e consumo de energia, devido a esta aplicação possuir transações curtas, apresentar tempo em transação baixo, e o conjunto de leitura/escrita ser pequeno.



(a) Tempo de execução (b) Energia consumida



(c) *Speedup*

Figura 16: Resultados da aplicação SSCA2

Tabela 10: Quantidade de cancelamentos na aplicação SSCA2.

STM	2 *threads*	4 *threads*	8 *threads*	16 *threads*	32 *threads*	64 *threads*
TL2	24	74	188	126885	640884	1372055
TinySTM	96	342	27636	33460224	348922640	768077330
SwissTM	34	121	290	1101	303209	2428425
AdaptSTM	19	45	119	141727	858114	1101611

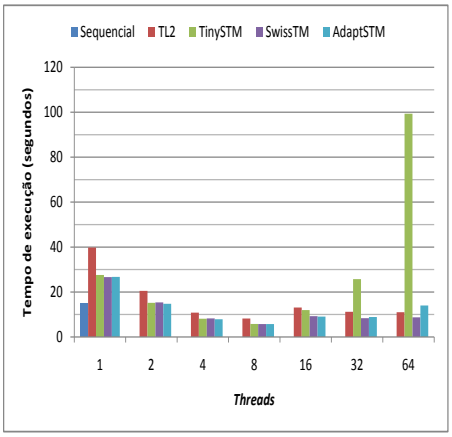
4.2.7 Resultados da aplicação Vacation

As aplicações Vacation (baixa contenção) e Vacation (média contenção) apresentaram resultados de desempenho e consumo de energia similares, conforme pode ser observado nas Figuras 17 e 18. Além disso, a quantidade de cancelamentos também foi semelhante em ambas configurações (baixa e média contenção), porém em escalas diferentes, conforme pode ser observado na Tabela 11.

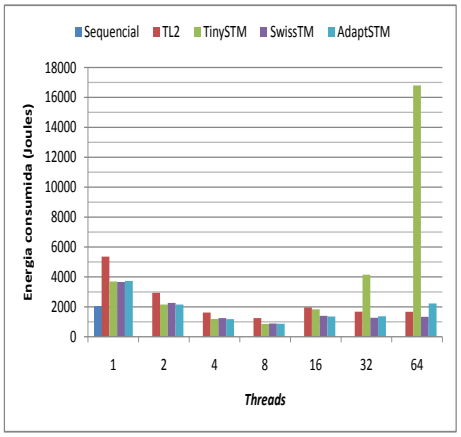
De modo geral, a TinySTM, SwissTM e AdaptSTM foram similares e apresentaram os melhores resultados na aplicação Vacation (baixa e média contenção), utilizando-se até 16 *threads*. No entanto, acima de 16 *threads*, a TinySTM foi cerca de 9x menos eficiente no consumo de energia e desempenho que a TL2, SwissTM e AdaptSTM, nesta aplicação. Este resultado é devido ao elevado número de cancelamentos obtidos pela TinySTM nestes cenários.

Tabela 11: Quantidade de cancelamentos na aplicação Vacation.

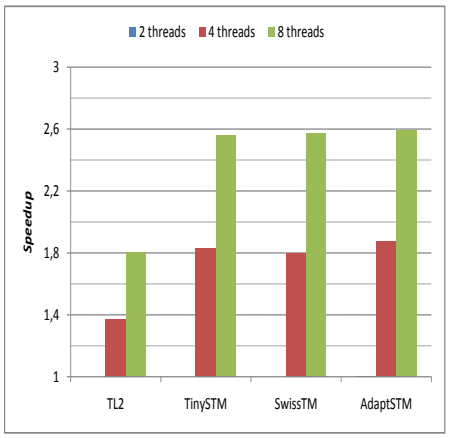
STM	2 *threads*	4 *threads*	8 *threads*	16 *threads*	32 *threads*	64 *threads*
Vacation (baixa contenção)						
TL2	4058	13332	31781	58735	48616	76375
TinySTM	1901	7567	20999	22501422	185065427	732763701
SwissTM	179	548	1263	6399	19442	91607
AdaptSTM	229	745	1742	29757	110268	405870
Vacation (média contenção)						
TL2	58673	191287	356970	2029087	604324	703663
TinySTM	9685	34261	106942	13607363	131033008	569615608
SwissTM	354	1057	2551	9529	22922	78672
AdaptSTM	1431	4023	8608	37371	137648	485521



(a) Tempo de execução

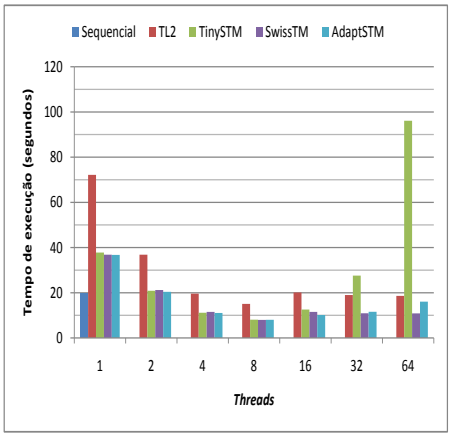


(b) Energia consumida

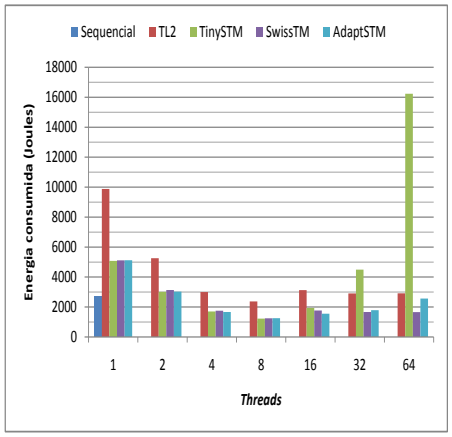


(c) *Speedup*

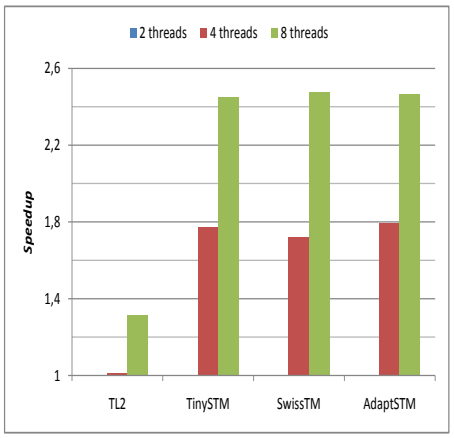
Figura 17: Resultados da aplicação Vacation (baixa contenção)



(a) Tempo de execução



(b) Energia consumida

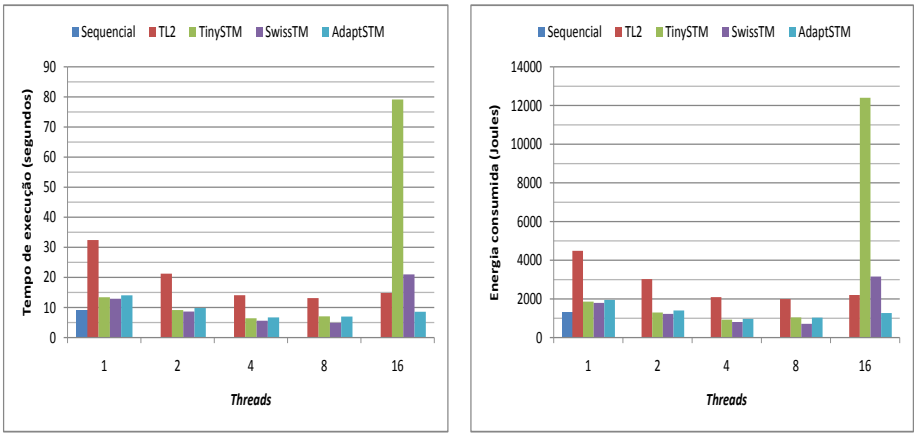


(c) *Speedup*

Figura 18: Resultados da aplicação Vacation (média contenção)

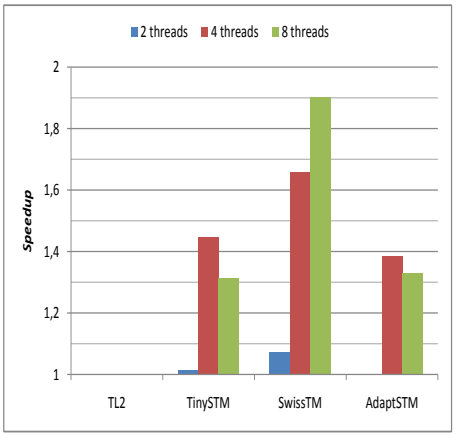
4.2.8 Resultados da aplicação Yada

Verifica-se que na aplicação Yada as quatro STMs obtiveram pouca melhora de desempenho na passagem de 4 para 8 *threads*, conforme pode ser observado na Figura 19. Em alguns casos, até mesmo degradou-se o desempenho. A principal razão para o baixo desempenho refere-se ao alto nível de contenção intrínseco nesta aplicação. Conforme descrito anteriormente, esta característica incorre em maiores taxas de conflitos entre as transações, à medida que aumenta-se o número de *threads*, degradando-se consequentemente o desempenho e consumo de energia.



(a) Tempo de execução

(b) Energia consumida



(c) *Speedup*

Figura 19: Resultados da aplicação Yada

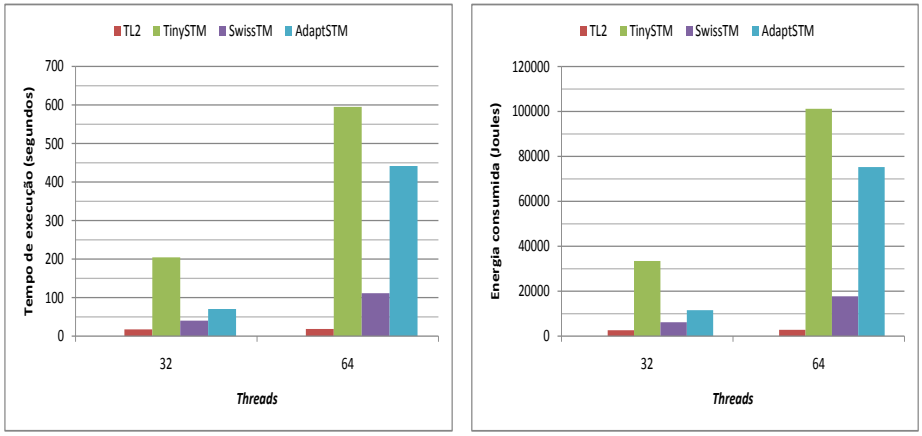
Apesar da TL2 ter apresentado os piores *speedups* na maioria das aplicações com até 8 *threads*, em alguns casos esta biblioteca é eficiente. Especifi-

camente na aplicação Yada utilizando-se 32 e 64 *threads* nota-se que a TL2 foi a implementação mais eficiente, seguida pela SwissTM, AdaptSTM e TinySTM, respectivamente, conforme pode ser observado na Figura 20. A principal razão para este resultado deve-se ao modo de detecção de conflitos empregado pela TL2. Nota-se que sob alta contenção, a detecção de conflitos atrasada é o modo mais eficiente, pois gera-se menos conflitos em relação ao modo adiantado, conforme pode ser observado na Tabela 12. Verifica-se que conforme aumenta-se o número de *threads* no modo de detecção atrasada, o número de cancelamentos de transações aumenta em menor proporção em relação ao modo adiantado ou até mesmo chega a diminuir em alguns casos, o que não acontece na detecção adiantada. Resultado similar ao descrito anteriormente é observado na aplicação Intruder com 32 e 64 *threads*, conforme pode ser observado na Figura 12.

Tabela 12: Quantidade de cancelamentos na aplicação Yada.

STM	**2 *threads***	**4 *threads***	**8 *threads***	**16 *threads***	**32 *threads***	**64 *threads***
TL2	2376218	3917966	4936525	2581383	1714519	1950572
TinySTM	21427335	41557333	71683810	3257881456	13407604392	49992149368
SwissTM	184249	391000	622608	1624121	2909019	6027714
AdaptSTM	338854	531068	779272	1960833	17182020	42501396

A SwissTM também apresentou um desempenho melhor do que a AdaptSTM e TinySTM na aplicação Yada com 32 e 64 *threads*, devido principalmente à utilização da detecção de conflito atrasada em situações de leitura-escrita e ao uso do gerenciador de contenção guloso. As bibliotecas AdaptSTM e TinySTM não foram eficientes na aplicação Yada com 32 e 64 *threads*. O motivo possível para a ineficiência da AdaptSTM nestes cenários deve-se ao tempo de *backoff* não ter sido suficiente nas transações longas apresentadas nesta aplicação, causando-se assim cancelamentos consecutivos de uma mesma transação. A ineficiência da TinySTM nestes casos deve-se ao uso exclusivo do gerenciador de contenção tímido.



(a) Tempo de execução (b) Energia consumida

Figura 20: Resultados da aplicação Yada utilizando-se 32 e 64 *threads*

4.2.9 Relação entre desempenho e energia

De modo geral, o consumo de energia seguiu a mesma tendência do tempo de execução. No entanto, verifica-se em muitos casos que o maior consumo de energia não corresponde ao maior tempo de execução, conforme pode ser observado na Tabela 13. De igual modo, nem sempre o menor consumo de energia equivale ao menor tempo de execução.

Nota-se que a AdaptSTM utilizando-se 64 *threads* na aplicação Intruder apresentou tempo de execução de 62,25 segundos, e consumiu 9636,575 Joules. A SwissTM, por sua vez, na aplicação Labyrinth utilizando-se 1 *thread* apresentou consumo de energia ligeiramente menor que o descrito anteriormente, consumindo 9519,103 Joules. Todavia, a SwissTM neste cenário apresentou tempo de execução de 70,359 segundos, cerca de 15% maior que o tempo de execução da AdaptSTM na aplicação Intruder com 64 *threads*. Semelhantemente ao caso anterior, a TinySTM na aplicação Bayes com 64 *threads* apresentou tempo de execução de 19,092 segundos, e consumiu 2670,323 Joules. A TL2, por sua vez, na aplicação Yada utilizando-se 64 *threads* apresentou tempo de execução ligeiramente menor que o descrito anteriormente. No entanto, o consumo de energia foi cerca de 10% maior que o apresentado pela TinySTM na aplicação Bayes com 64 *threads*.

Nessa perspectiva, constata-se que embora o tempo de execução tenha grande influência sobre o consumo de energia, ambos não são diretamente proporcionais. O motivo possível para este resultado deve-se aos recursos computacionais utilizados por cada execução.

Outro fator que interfere no consumo de energia é a quantidade de *threads* utilizadas. Verifica-se em vários casos onde aumenta-se a quantidade de *threads*, embora ganhe-se desempenho consome-se maior energia em relação à utilização de menos *threads*. Nota-se que a TL2 na aplicação Intruder utilizando-se 16 *threads* apresentou menor tempo de execução em relação à utilização de 1 *thread* nesta aplicação. No entanto, utilizando-se 16 *threads* o consumo de energia foi cerca 16% maior que na utilização de 1 *thread*. Outros exemplos similares podem ser observados na Tabela 13. O motivo possível para esse resultado é devido ao *overhead* dos recursos alocados pelas *threads* e do escalonamento dessas nos *cores*.

Na Tabela 14 são mostrados os valores máximos de potência em diferentes configurações (número de *threads*) obtidos nos experimentos. Observa-se que conforme aumenta-se o número de *threads*, o pico de energia aumenta ligeiramente, na maioria dos casos. O valor máximo de potência chegou a 192 W, e foi obtido utilizando-se 32 *threads*.

Tabela 13: Relação do tempo de execução e consumo de energia.

Configuração de execução	**Tempo de execução (segundos)**	**Energia consumida (Joules)**
TinySTM/Bayes/2 *threads*	19,419	**2660,601**
TinySTM/Bayes/64 *threads*	**19,092**	2670,323
TL2/Intruder/1 *thread*	47,130	**6455,105**
TL2/Intruder/16 *thread*	**45,692**	7465,823
TinySTM/Labyrinth/4 *threads*	24,718	**3900,231**
Tl2/Labyrinth/8 *threads*	**24,142**	4048,293
AdaptSTM/Vacation (baixa)/16 *threads*	9,053	**1355,152**
AdaptSTM/Vacation (baixa)/32 *threads*	**8,896**	1369,637
TL2/Vacation (média)/32 *threads*	18,998	**2906,298**
TL2/Vacation (média)/64 *threads*	**18,590**	2912,125
TinySTM/Bayes/64 *threads*	19,092	**2670,323**
TL2/Yada/64 *threads*	**18,616**	2816,864
SwissTM/Bayes/64 *threads*	17,833	2551,790
AdaptSTM/Bayes/64 *threads*	17,904	**2551,543**
SwissTM/SSCA2/64 *threads*	**17,682**	2704,336
Sequencial/Genome	8,647	**1200,214**
SwissTM/Vacation (baixa)/32 *threads*	**8,339**	1272,226
TL2/Genome/2 *threads*	5,619	**792,380**
AdaptSTM/Kmeans/32 *threads*	**5,144**	822,877
SwissTM/Intruder/2 *threads*	14,900	**2142,656**
TL2/Yada/16 *threads*	**14,832**	2199,407
SwissTM/Intruder/16 *threads*	**16,681**	2608,192
TL2/Yada/32 *threads*	17,335	**2603,511**
AdaptSTM/Intruder/64 *threads*	**62,250**	9636,575
TinySTM/Labyrinth/1 *thread*	70,324	9561,255
SwissTM/Labyrinth/1 *thread*	70,359	**9519,103**
AdaptSTM/Labyrinth/1 *thread*	70,395	9668,543
SwissTM/Kmeans/1 *thread*	20,089	**2836,107**
TinySTM/SSCA2/64 *threads*	**19,529**	3003,794
TL2/Labyrinth/4 *threads*	28,579	**4510,442**
SwissTM/Labyrinth/32 *threads*	**27,567**	4562,365
SwissTM/Vacation (baixa)/4 *threads*	8,244	**1245,943**
AdaptSTM/Vacation (média)/8 *threads*	**8,027**	1252,813

Tabela 14: Picos de potência obtidos nos experimentos.

	1 *thread*	**2 *threads***	**4 *threads***	**8 *threads***	**16 *threads***	**32 *threads***	**64 *threads***
Pico de energia (W)	146	158	168	176	184	192	180

4.2.10 Caracterização de escalabilidade das STMs utilizadas

A reexecução de transações canceladas devido a conflitos é um dos principais fatores que prejudicam a escalabilidade de aplicações baseadas em STM. Nesta perspectiva, mostrou-se que conforme aumenta-se o número de *threads* eleva-se a quantidade de cancelamentos entre transações, na maioria dos casos. Constata-se que, em especial em aplicações de alta contenção (Intruder e Yada), o número de cancelamentos de transações aumenta substancialmente conforme utiliza-se mais *threads*, na maioria dos casos. No *benchmark* STAMP, o número de transações é dividido igualmente entre as *threads* utilizadas. Assim, esses resultados são devidos ao número de transações por *threads* e pelo escalonamento dessas no sistema.

A TL2 apresentou as menores taxas de crescimento de conflitos na maioria dos casos, seguida pela AdaptSTM e SwissTM, nesta ordem. A TinySTM, por sua vez, obteve as maiores taxas de cancelamento na maioria das configurações de execução. Mesmo apresentando as maiores taxas de cancelamento, este resultado não prejudicou significamente o desempenho desta biblioteca, em alguns casos. Todavia, utilizando-se 32 e 64 *threads* a TinySTM apresentou os piores resultados entre as demais, na maioria das aplicações.

Constata-se que a escalabilidade das STMs utilizadas está relacionada diretamente a suas estratégias de detecção e resolução de conflitos. Nesta perspectiva, verifica-se que em aplicações com transações curtas a AdaptSTM mostra-se a biblioteca mais eficiente. Este resultado é devido principalmente ao uso do gerenciador de contenção *backoff* exponencial empregado pela AdaptSTM. Em aplicações com transações médias a SwissTM apresenta a melhor escalabilidade, devido principalmente a utilização do gerenciador de contenção guloso e a detecção de conflitos atrasada em casos de leitura-escrita. Em cenários com longas transações e sob média/alta contenção a TL2 apresenta os melhores resultados. Este resultado é devido principalmente pela utilização de detecção de conflitos atrasada. A TinySTM, por sua vez, mostra-se eficiente somente em aplicações que apresentem mínimas taxas de cancelamentos, pois utiliza-se exclusivamente o gerenciador de contenção tímido. Na Tabela 15 é sintetizado o motivo da eficiência em determinados cenários das STMs utilizadas.

Tabela 15: Relação entre a eficiência das STMs utilizadas e o cenário de execução.

STM	Cenário	Principal motivo
AdaptSTM	transações curtas	uso de *backoff* exponencial (alta contenção)
SwissTM	transações médias	gerenciador de contenção guloso detecção de conflitos atrasada (leitura-escrita)
TL2	transações longas e sob média/alta contenção	detecção de conflitos atrasada
TinySTM	mínimas taxas de cancelamento	uso exclusivo do gerenciador de contenção tímido

4.2.11 Discrepâncias entre resultados e características do *benchmark* STAMP

Comparando-se as características das aplicações do *benchmark* STAMP e os resultados apresentados neste capítulo, identifica-se variações entre os resultados e as características.

Conforme as características das aplicações do STAMP mostradas na Tabela 4, a aplicação Labyrinth apresenta alta contenção. No entanto, esta aplicação apresentou os melhores resultados de *speedup* para até 8 *threads*. Além disso, não é notada uma grande diminuição de desempenho acima de 16 *threads* em nenhuma das STMs, o que não ocorreu em outras aplicações de média e alta contenção. A aplicação Yada, por sua vez, apresentou altíssimas taxas de cancelamentos. Todavia, esta aplicação é caracterizada como de média contenção. Por fim, a aplicação Bayes apresentou baixas taxas de cancelamentos na maioria das configurações de execução, não condizendo assim com sua característica de alta contenção. Discrepâncias similares às descritas anteriormente também foram apresentadas por outros trabalhos (RUI, 2012; MOREIRA, 2010).

4.3 Observações finais

Este capítulo detalhou a metodologia e os resultados obtidos no desenvolvimento desta dissertação, apresentando a análise e caracterização do consumo de energia e desempenho de aplicações baseadas em STMs em ambiente de computação real.

Analisando-se a execução sequencial e a execução baseada nas bibliotecas de STMs verifica-se que o *overhead* introduzido pelas STMs é alto, utilizando-se as aplicações do *benchmark* STAMP. Embora a utilização de STM facilite a codificação de *software* concorrente, a implementação desses sistemas não é simples. É necessário realizar a sincronização em baixo nível de forma transparente ao programador. Nessa perspectiva, a utilização de STM introduz a necessidade de operações primitivas que inicializam, cancelam e efetivam as transações, as-

sim como detectar possíveis conflitos.

Apesar do alto custo imposto pelas primitivas do sistema transacional e das características das aplicações que influenciam negativamente o desempenho, a maioria das configurações de execução obtiveram ganhos de desempenho com o aumento do número de *threads* (até 8 *threads*). A STM que mais apresentou *speedup* foi a SwissTM, na maioria dos cenários de execução com até 8 *threads*. A TL2, por sua vez, apresentou os piores *speedups*. Nas aplicações Intruder e Yada esta biblioteca não apresentou ganhos em relação à execução sequencial.

Ficou evidenciado que a eficiência de uma determinada estratégia do projeto de STM está diretamente relacionada às características das aplicações que influenciam a execução de TMs. Assim, nenhuma estratégia supera as demais em todos os cenários. Nessa perspectiva, as implementações que adaptam seu mecanismo de funcionamento com objetivo de obter os melhores parâmetros de execução de acordo com o cenário, tendem a obter melhor desempenho e eficiência energética.

Outros trabalhos analisaram o consumo de energia e desempenho em STM. Estes trabalhos avaliaram as implementações TL2 e TinySTM, utilizando-se o *benchmark* STAMP com parâmetros sugeridos para sistemas simulados, conforme descrito nas seções 3.3, 3.4 e 3.7. Diferentemente dos resultados apresentados neste capítulo, os trabalhos que avaliaram a TL2 não obtiveram ganhos na passagem de 4 para 8 *threads*, especificamente na aplicação Vacation (média contenção). O trabalho que avaliou a TinySTM, por sua vez, não apresentou ganhos na passagem de 4 para 8 *threads* na aplicação Intruder. Além disso, este trabalho apresentou maior *speedup* nas aplicações Bayes e Kmeans utilizando-se 4 e 8 *threads*. Os motivos possíveis para essa divergência nos resultados estão relacionados às plataformas computacionais usadas, os parâmetros de execução utilizados no *benchmark* e questões relacionadas às *caches*.

5 CONCLUSÕES

Uma das principais dificuldades no desenvolvimento de *software* paralelo é garantir a correta sincronização entre as *threads*, necessária para evitar condições de corrida entre os fluxos de execução. As TMs têm sido desenvolvidas por pesquisadores de programação concorrente com o objetivo de reduzir as dificuldades e limitações encontradas em mecanismos de sincronização tradicionais, em especial aqueles baseados em *locks*.

O aumento da eficiência energética é de suma importância em todo o contexto computacional, independente do *hardware* e *software* utilizado, pois efetiva-se reduções em custos, maximiza-se o tempo útil das baterias de dispositivos móveis, diminui o impacto ambiental, entre outros benefícios. No entanto, por se tratar de uma alternativa recentemente proposta, pouco se conhece a respeito do consumo de energia de TM, em especial de implementações em *software*.

Nesse contexto, o presente trabalho apresentou como principal contribuição a análise e caracterização do consumo de energia e desempenho de quatro importantes bibliotecas de STM, TL2, TinySTM, SwissTM e AdaptSTM, utilizando-se o *benchmark* STAMP. Diferentemente de outros trabalhos, as execuções não foram simuladas mas executadas em ambiente de computação real, suprindo então a deficiência na literatura.

Resultados obtidos mostram a SwissTM como a biblioteca mais eficiente em termos de consumo de energia e desempenho, seguida pela AdaptSTM, TinySTM e TL2, na maioria dos cenários de execução utilizando-se até 8 *threads*. Em relação à escalabilidade destas bibliotecas, constata-se que a SwissTM mostra-se a biblioteca mais eficiente em aplicações com transações de tamanho médio. Em aplicações com transações curtas a AdaptSTM apresenta a melhor escalabilidade. A TL2, por sua vez, mostra-se muito eficiente em aplicações com transações longas e sob média e alta contenção. Por fim, a TinySTM mostra-se a biblioteca menos eficiente na maioria dos cenários, exibindo bons resultados de escalabilidade somente em aplicações que apresentem mínimas taxas de cancelamentos.

A reexecução de transações canceladas devido a conflitos é um dos principais fatores que afetam negativamente o desempenho e consumo de energia de aplicações baseadas em STM. Nesta perspectiva, mostrou-se um problema relacionado à escalabilidade das bibliotecas utilizadas. Constata-se que, conforme aumenta-se o número de *threads* eleva-se substancialmente a quantidade de cancelamentos entre as transações, na maioria dos casos. Todavia, cada biblioteca de STM utilizada mostrou-se eficiente em termos de escalabilidade em cenários diferentes, devido à particularidade de suas estratégias de detecção e resolução de conflitos.

Mostrou-se também que nem sempre as execuções que apresentam o maior consumo de energia apresentam o maior tempo de execução. De igual modo, verifica-se que o menor consumo de energia não corresponde ao menor tempo de execução, em muitos casos. Nessa perspectiva, constata-se que embora o tempo de execução tenha grande influência sobre o consumo de energia, ambos não são proporcionais.

Embora facilitem a programação, as STMs utilizadas apresentaram ineficiência em alguns casos, em especial sob alta contenção. Além disso, nenhuma estratégia de implementação supera as demais em todos os cenários. Nesse contexto, os resultados apresentados no presente trabalho podem servir como guia para desenvolvedores de algoritmos transacionais, visando-se os melhores parâmetros de execução para cada cenário, buscando-se assim a melhora do desempenho e eficiência energética das implementações de STMs.

5.1 Trabalhos futuros

Por se tratar de um novo mecanismo de sincronização, TM é alvo de muita pesquisa, e muitos desafios precisam ser vencidos em várias frentes para melhor caracterizá-la.

As implementações TL2, TinySTM, SwissTM e AdaptSTM foram utilizadas neste trabalho em seu modo de operação padrão. No entanto, estas permitem em sua instalação configurar alguns parâmetros, como por exemplo, as estratégias adotadas para versionamento de dados, detecção/resolução de conflitos, tamanho do vetor de *locks*, entre outros parâmetros. Nesta perspectiva, tem-se como trabalhos futuros analisar as mesmas implementações de STMs utilizadas neste trabalho variando os parâmetros de instalação mencionados anteriormente, contemplando assim um maior número de configurações do sistema transacional.

Embora existam diversas estratégias e pesquisas sobre gerenciamento de contenção, grande parte das pesquisas avaliam poucas estratégias sob um

mesmo contexto, especificamente quando avalia-se o consumo de energia. Assim, a avaliação de várias estratégias de gerenciamento de contenção sob o mesmo contexto, em termos de consumo de energia e desempenho, seria de grande valia, visto que o gerenciador de contenção é um componente essencial no contexto do sistema transacional, em especial em casos de alta contenção.

A comparação de aplicações baseadas em STM e *locks* sob o mesmo contexto computacional já foi realizada em outros trabalhos. No entanto, nesta comparação utilizou-se apenas a biblioteca de STM TL2. Além disso, os experimentos foram conduzidos em um ambiente computacional simulado. Nessa perspectiva, tem-se como trabalhos futuros realizar comparações de desempenho e consumo de energia de aplicações baseadas em bibliotecas de STMs e *locks*, em ambiente computacional não-simulado, com a finalidade de comparar a eficiência de ambos mecanismos de sincronização.

Mostrou-se nesse trabalho que houve discrepância entre alguns resultados e as características do *benchmark* STAMP. Divergências similares também foram apresentadas por outros trabalhos. Nessa perspectiva, sugere-se uma classificação mais pormenorizada das características do *benchmark* STAMP, visto que este é um dos *benchmarks* mais utilizados na avaliação de TMs.

5.2 Publicações

Durante o desenvolvimento do presente trabalho os seguintes artigos foram publicados:

- Análise do Consumo de Energia e Desempenho de Memórias Transacionais em Software em Cenário de Alta Contenção.
  Timóteo M. Rico, Maurício L. Pilla, André R. Du Bois, Rodrigo M. Duarte.
  In: 13ª Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD-RS 2013 - Fórum de PG).

- Análise do Consumo de Energia e Desempenho de Memórias Transacionais em Software sob Baixa Contenção.
  Timóteo M. Rico, Maurício L. Pilla, André R. Du Bois.
  In: III Seminário de Pesquisa em Computação da UFPel (SPC-UFPel 2012).

- Avaliação do Consumo de Energia e Desempenho de Memórias Transacionais em Software.
  Timóteo Matthies Rico, Rodrigo Medeiros Duarte, Maurício Lima Pilla, André Rauber Du Bois.
  In: XIV Encontro de Pós-Graduação da UFPel (ENPOS-UFPel 2012).

- Energy Consumption on Software Transactional Memories.
  Timóteo M. Rico, Maurício L. Pilla, André R. Du Bois.
  In: I Workshop em Modelos de Programação Paralela – MPP 2012 (junto com WSCAD-SSC 2012).

- Computação Sustentável e Mecanismos de Sincronização.
  Timóteo M. Rico, Felipe L. Teixeira, Maurício Lima Pilla, André Rauber Du Bois.
  In: 12ª Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD-RS 2012 - Fórum de PG).

O autor também participou da escrita do texto de um minicurso, intitulado "Programação de Máquinas *Multicore* usando Memórias Transacionais em *Software*", ministrado na 13ª Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD-RS 2013).

REFERÊNCIAS

ANDREWS, G. R. **Foundations of multithreaded, parallel, and distributed programming**. Boston: Addison Wesley, 2000.

ANSARI, M.; KOTSELIDIS, C.; WATSON, I.; KIRKHAM, C.; LUJáN, M.; JARVIS, K. Lee-TM: A non-trivial benchmark suite for transactional memory. In: IN PROCEEDINGS OF THE 8TH INTERNATIONAL CONFERENCE ON ALGORITHMS AND ARCHITECTURES FOR PARALLEL PROCESSING, ICA3PP, 2008. **Anais...** [S.l.: s.n.], 2008.

BALDASSIN, A.; CARVALHO, J. P. de; GARCIA, L. A.; AZEVEDO, R. Energy-Performance Tradeoffs in Software Transactional Memory. In: COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD), 2012 IEEE 24TH INTERNATIONAL SYMPOSIUM ON, 2012. **Anais...** [S.l.: s.n.], 2012. p.147–154.

BALDASSIN, A. J. **Explorando Memória Transacional em Software nos Contextos de Arquiteturas Assimétricas, Jogos Computacionais e Consumo de Energia**. 2009. Tese (Doutorado em Ciência da Computação) — Universidade Estadual de Campinas, Campinas, SP, Brasil.

BALDASSIN, A.; KLEIN, F.; ARAUJO, G.; AZEVEDO, R.; CENTODUCATTE, P. Characterizing the Energy Consumption of Software Transactional Memory. **IEEE Comput. Archit. Lett.**, Washington, DC, USA, v.8, p.56–59, July 2009.

BARCELOS, D.; BRIÃO, E. W.; WAGNER, F. R. A hybrid memory organization to enhance task migration and dynamic task allocation in NoC-based MPSoCs. In: INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 20., 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.282–287. (SBCCI '07).

BENINI, L.; DE MICHELI, G. Networks on Chips: A New SoC Paradigm. **Computer**, Los Alamitos, CA, USA, v.35, p.70–78, January 2002.

BJERREGAARD, T.; MAHADEVAN, S. A survey of research and practices of Network-on-chip. **ACM Comput. Surv.**, New York, NY, USA, v.38, June 2006.

BURGER, D.; AUSTIN, T. M. The SimpleScalar tool set, version 2.0. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.25, p.13–25, June 1997.

CAO MINH, C.; CHUNG, J.; KOZYRAKIS, C.; OLUKOTUN, K. STAMP: Stanford Transactional Applications for Multi-Processing. In: IISWC '08: PROCEEDINGS OF THE IEEE INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION, 2008. **Anais...** [S.l.: s.n.], 2008.

CARDOSO, F. **Efeito estufa**: Por que a Terra morre de calor. [S.l.]: Editora Terceiro Nome, 2006.

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional Locking II. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED COMPUTING (DISC 2006), 20., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.194–208.

DRAGOJEVIC, A.; GUERRAOUI, R.; KAPALKA, M. Stretching transactional memory. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2009., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.155–165. (PLDI '09).

ENNALS, R. **Software Transactional Memory Should Not Be Obstruction-Free**. [S.l.]: Intel Research Cambridge Tech Report, 2006. (IRC-TR-06-052).

FELBER, P.; FETZER, C.; RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 13., 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.237–246. (PPoPP '08).

FORREST, W.; KAPLAN, J.; KINDLER, N. Data Centers: How to cut carbon emissions and costs. **McKinsey on Business Technology**, [S.l.], v.14, n.6, 2008.

FRASER, K. **Practical lock-freedom**. [S.l.]: University of Cambridge, 2004. (Relatório Técnico 579).

FREEIPMI CORE TEAM. **GNU FreeIPMI**. Disponível em: `http://www.gnu.org/software/freeipmi/`. Acessado em: Abril 2013.

HAMM, S. It's too darn hot. **Businessweek. com**, [S.l.], 2008.

HARRIS, T.; FRASER, K. Language support for lightweight transactions. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMING, SYSTEMS,

LANGUAGES, AND APPLICATIONS, 18., 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p.388–402. (OOPSLA '03).

HARRIS, T.; LARUS, J.; RAJWAR, R. Transactional memory. **Synthesis Lectures on Computer Architecture**, [S.l.], p.1–263, 2010.

HARRIS, T.; MARLOW, S.; JONES, S.; HERLIHY, M. Composable memory transactions. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2005. **Proceedings...** [S.l.: s.n.], 2005. p.48–60.

HERLIHY, M.; LUCHANGCO, V.; MOIR, M.; SCHERER III, W. N. Software transactional memory for dynamic-sized data structures. In: PRINCIPLES OF DISTRIBUTED COMPUTING, 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p.92–101. (PODC '03).

HERLIHY, M.; MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In: OF THE 20TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1993, New York, NY, USA. **Proceedings...** ACM, 1993. p.289–300. (ISCA '93).

JONES, S. Beautiful concurrency. **Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly)). O'Reilly Media, Inc**, [S.l.], p.385–406, 2007.

KAXIRAS, S.; MARTONOSI, M. **Computer Architecture Techniques for Power-Efficiency**. [S.l.]: Morgan & Claypool Publishers, 2008. (Synthesis Lectures on Computer Architecture).

KLEIN, F.; BALDASSIN, A.; ARAUJO, G.; CENTODUCATTE, P.; AZEVEDO, R. On the energy-efficiency of software transactional memory. In: ANNUAL SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN: CHIP ON THE DUNES, 22., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.33:1–33:6. (SBCCI '09).

KLEIN, F.; BALDASSIN, A.; MOREIRA, J.; CENTODUCATTE, P.; RIGO, S.; AZEVEDO, R. STM versus lock-based systems: an energy consumption perspective. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 16., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.431–436. (ISLPED '10).

KRONBAUER, F.; RIGO, S. Experimentos com Gerenciamento de Contenção em uma Memória Transacional com Suporte em Software. **Anais do X Simpósio em Sistemas Computacionais WSCAD-SSC**, São Paulo, SP, Brasil, p.44–51, 2009.

KUNZ, L. **Memória Transacional em Hardware para Sistemas Embarcados Multiprocessados Conectados por Redes-em-Chip**. 2010. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brasil.

KURP, P. Green computing. **Commun. ACM**, New York, NY, USA, v.51, p.11–13, Oct. 2008.

LAMB, J. **The Greening of IT**: How Companies Can Make a Difference for the Environment. 1st.ed. [S.l.]: IBM Press, 2009.

LOGHI, M.; PONCINO, M.; BENINI, L. Cycle-accurate power analysis for multiprocessor systems-on-a-chip. In: ACM GREAT LAKES SYMPOSIUM ON VLSI, 14., 2004, New York, NY, USA. **Proceedings...** ACM, 2004. p.410–406. (GLS-VLSI '04).

LOMET, D. Process structuring, synchronization, and recovery using atomic actions. In: ACM SIGPLAN NOTICES, 1977. **Anais...** [S.l.: s.n.], 1977. v.12, n.3, p.128–137.

MAGNUSSON, P. S.; CHRISTENSSON, M.; ESKILSON, J.; FORSGREN, D.; HALLBERG, G.; HOGBERG, J.; LARSSON, F.; MOESTEDT, A.; WERNER, B. Simics: A full system simulation platform. **Computer**, [S.l.], v.35, n.2, p.50–58, 2002.

MARATHE, V. J.; Scherer III, W. N.; SCOTT, M. L. Adaptive Software Transactional Memory. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED COMPUTING, 19., 2005, Cracow, Poland. **Proceedings...** [S.l.: s.n.], 2005. p.354–368. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.

MARATHE, V. J.; SPEAR, M. F.; HERIOT, C.; ACHARYA, A.; EISENSTAT, D.; III, W. N. S.; SCOTT, M. L. Lowering the overhead of nonblocking software transactional memory. In: FIRST ACM SIGPLAN WORKSHOP ON LANGUAGES, COMPILERS, AND HARDWARE SUPPORT FOR TRANSACTIONAL COMPUTING, 2006. **Anais...** [S.l.: s.n.], 2006.

MCKENNEY, P. E.; MICHAEL, M. M.; TRIPLETT, J.; WALPOLE, J. Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.44, p.93–101, August 2010.

MITTAL, S.; NITIN. A Resolution for Shared Memory Conflict in Multiprocessor System-on-a-Chip. **International Journal of Computer Science Issues, IJCSI**, [S.l.], v.8, p.503–507, July 2011.

MOREIRA, J. B. C. G. **Análise do Consumo de Energia em STMs e uma Plataforma de Simulação Multiprocessada com Abastração Híbrida**. 2010. Dissertação (Mestrado em Ciência da Computação) — Universidade Estadual de Campinas, Campinas, SP, Brasil.

MORESHET, T.; BAHAR, R. I.; HERLIHY, M. Energy reduction in multiprocessor systems using transactional memory. In: LOW POWER ELECTRONICS AND DESIGN, 2005., 2005, New York, NY, USA. **Proceedings...** ACM, 2005. p.331–334. (ISLPED '05).

MORESHET, T.; BAHAR, R. I.; HERLIHY, M. Energy-aware microprocessor synchronization: Transactional memory vs. locks. **In 4th Workshop on Memory Performance Issues, held in conjunction with the International Symposium on High-Performance Computer Architecture**, [S.l.], 2006.

MÓR, S. D. K.; ALVES, M. A.; LIMA, J. V. F.; MAILARD, N. B.; NAVAUX, P. O. A. Eficiência Energética em Computação de Alto Desempenho: Uma Abordagem em Arquitetura e Programação para Green Computing. **XXX Congresso da Sociedade Brasileira de Computação, CSBC**, [S.l.], p.346–360, 2010.

MURUGESAN, S. Harnessing Green IT: Principles and Practices. **IT Professional**, Piscataway, NJ, USA, v.10, p.24–33, January 2008.

PAYER, M.; GROSS, T. Performance Evaluation of Adaptivity in Software Transactional Memory. In: PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE (ISPASS), 2011 IEEE INTERNATIONAL SYMPOSIUM ON, 2011. **Anais...** [S.l.: s.n.], 2011. p.165–174.

REIS, L. dos; CUNHA, E. da. **Energia elétrica e sustentabilidade**: aspectos tecnológicos, socioambientais e legais. [S.l.]: Manole, 2006.

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

RUI, F. F. **Uma Avaliação Comparativa de Sistemas de Memória Transacional de Software e seus Benchmarks**. 2012. Dissertação (Mestrado em Ciência da Computação) — Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, RS, Brasil.

SHAVIT, N.; TOUITOU, D. Software transactional memory. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 1995, New York, NY, USA. **Proceedings...** ACM, 1995. p.204–213. (PODC '95).

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Sistema de Banco de Dados, 3ª edição**. [S.l.]: Editora Pearson, 1999.

WANG, H.-S.; ZHU, X.; PEH, L.-S.; MALIK, S. Orion: a power-performance simulator for interconnection networks. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 35., 2002, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society Press, 2002. p.294–305. (MICRO 35).

WECHSLER, O. Setting New Standards for Energy-Efficient Performance. **White Paper, Inside Intel Core Microarchitecture**, [S.l.], p.4–5, 2006.

WILTON, S. J. E.; JOUPPI, N. P. CACTI: An Enhanced Cache Access and Cycle Time Model. **IEEE Journal of Solid-State Circuits**, [S.l.], v.31, p.677–688, 1996.

WOO, S. C.; OHARA, M.; TORRIE, E.; SINGH, J. P.; GUPTA, A. The SPLASH-2 programs: characterization and methodological considerations. In: COMPUTER ARCHITECTURE, 22., 1995, New York, NY, USA. **Proceedings...** ACM, 1995. p.24–36. (ISCA '95).

ZHUO, H.; YIN, J.; RAO, A. V. Remote Management with the Baseboard Management Controller in Eighth-Generation Dell PowerEdge Servers. **Magazine of Dell Power Solutions**, [S.l.], p.26–29, 2004.

ANEXO A RESULTADOS DO TEMPO DE EXECUÇÃO (SEGUNDOS)

STM	segundos 1 *thread*	segundos 2 *threads*	segundos 4 *threads*	segundos 8 *threads*	segundos 16 *threads*	segundos 32 *threads*	segundos 64 *threads*
Bayes							
Sequencial	19,414						
TinySTM	20,151	19,419	18,928	18,882	18,910	18,943	19,092
SwissTM	19,051	18,391	17,914	17,930	17,759	17,824	17,833
AdaptSTM	19,132	18,553	18,008	18,044	17,890	17,915	17,904
Genome							
Sequencial	8,647						
TL2	10,361	5,619	3,176	2,358	2,506	2,891	3,126
TinySTM	9,745	5,035	2,597	1,921	2,180	3,118	6,606
SwissTM	9,282	4,827	2,490	1,716	1,973	2,382	2,760
AdaptSTM	9,731	4,594	2,448	1,739	1,991	2,901	4,604
Intruder							
Sequencial	22,309						
TL2	47,130	32,694	24,993	25,171	45,692	98,272	192,525
TinySTM	31,542	17,296	9,937	7,777	29,838	140,548	358,161
SwissTM	25,649	14,900	8,914	7,181	16,681	139,659	671,757
AdaptSTM	31,269	17,638	10,081	7,797	7,981	11,796	62,250
Kmeans							
Sequencial	12,477						
TL2	32,410	16,762	9,067	7,118	22,526	61,601	138,622
TinySTM	21,913	11,935	7,478	6,537	21,579	58,894	164,332
SwissTM	20,089	10,540	5,763	5,328	16,628	51,636	131,240
AdaptSTM	22,586	13,396	7,694	5,401	5,759	5,144	5,524
Labyrinth							
Sequencial	70,937						
TL2	70,689	51,137	28,579	24,142	30,103	33,414	42,589
TinySTM	70,324	51,331	24,718	14,435	14,793	15,424	14,936
SwissTM	70,359	36,742	22,846	17,850	21,806	27,567	32,156
AdaptSTM	70,395	36,741	20,884	18,208	22,173	26,325	32,278
SSCA2							
Sequencial	16,016						
TL2	21,668	15,655	12,219	10,841	11,145	13,042	16,230
TinySTM	18,448	14,004	11,308	10,346	10,815	14,630	19,529
SwissTM	19,847	14,536	11,637	10,633	10,145	10,834	17,682
AdaptSTM	22,099	16,479	12,506	11,111	11,610	12,902	14,356
Vacation (baixa contenção)							
Sequencial	14,817						
TL2	39,801	20,497	10,804	8,211	13,105	11,217	11,026
TinySTM	27,581	15,186	8,101	5,792	11,964	25,741	99,322
SwissTM	26,662	15,370	8,244	5,759	9,274	8,339	8,722
AdaptSTM	26,756	14,770	7,901	5,722	9,053	8,896	14,007
Vacation (média contenção)							
Sequencial	19,816						
TL2	72,178	36,850	19,618	15,058	20,095	18,998	18,590
TinySTM	37,759	20,883	11,159	8,080	12,539	27,599	96,115
SwissTM	36,856	21,214	11,508	8,004	11,536	10,897	10,847
AdaptSTM	36,731	20,387	11,037	8,027	10,174	11,591	16,061
Yada							
Sequencial	9,291						
TL2	32,407	21,266	14,077	13,135	14,832	17,335	18,616
TinySTM	13,403	9,147	6,415	7,070	79,132	204,497	595,095
SwissTM	12,919	8,663	5,602	4,887	20,999	40,139	111,237
AdaptSTM	14,034	9,823	6,712	6,982	8,582	70,263	441,511

ANEXO B RESULTADOS DO CONSUMO DE ENERGIA (JOULES)

STM	Joules 1 *thread*	Joules 2 *threads*	Joules 4 *threads*	Joules 8 *threads*	Joules 16 *threads*	Joules 32 *threads*	Joules 64 *threads*
Bayes							
Sequencial	2695,150						
TinySTM	2719,253	2660,601	2589,666	2582,037	2584,772	2586,808	2670,323
SwissTM	2661,298	2575,071	2523,386	2531,054	2510,223	2524,848	2551,790
AdaptSTM	2647,765	2581,268	2519,364	2519,034	2484,766	2509,630	2551,543
Genome							
Sequencial	1200,214						
TL2	1417,678	792,380	464,186	347,377	370,898	431,265	472,128
TinySTM	1339,979	705,478	376,312	279,895	317,967	467,309	1033,679
SwissTM	1259,081	673,611	357,693	247,524	286,744	345,864	405,879
AdaptSTM	1351,884	648,709	359,567	255,307	292,141	437,380	719,953
Intruder							
Sequencial	3096,645						
TL2	6455,105	4804,363	3909,861	4049,038	7465,823	16171,320	31640,980
TinySTM	4369,589	2461,348	1485,169	1168,737	4769,541	23634,450	61481,730
SwissTM	3523,095	2142,656	1354,274	1112,347	2608,192	22822,100	110729,200
AdaptSTM	4349,236	2541,618	1505,631	1191,555	1210,492	1828,756	9636,575
Kmeans							
Sequencial	1725,280						
TL2	4517,701	2508,816	1389,354	1155,769	3695,612	10193,410	23080,070
TinySTM	3042,267	1793,077	1177,451	1053,339	3605,295	10034,770	28611,100
SwissTM	2836,107	1586,741	906,826	865,113	2708,937	8438,168	21521,450
AdaptSTM	3108,399	2091,364	1209,293	865,074	928,427	822,877	894,273
Labyrinth							
Sequencial	9903,128						
TL2	9855,721	7498,793	4510,442	4048,293	5058,681	5649,249	7216,971
TinySTM	9561,255	7440,519	3900,231	2364,408	2425,350	2527,843	2448,474
SwissTM	9519,103	5286,700	3564,163	2987,786	3667,840	4562,365	5408,845
AdaptSTM	9668,543	5473,512	3256,842	3044,253	3711,681	4446,515	5461,804
SSCA2							
Sequencial	2262,676						
TL2	3058,675	2236,449	1773,423	1563,599	1604,491	1918,364	2451,583
TinySTM	2608,451	2005,463	1629,567	1485,832	1568,153	2199,809	3003,794
SwissTM	2779,099	2077,721	1667,833	1518,089	1464,351	1605,838	2704,336
AdaptSTM	3119,833	2378,772	1838,473	1604,545	1706,189	1938,380	2213,713
Vacation (baixa contenção)							
Sequencial	2037,844						
TL2	5359,922	2936,913	1622,321	1251,597	1964,880	1675,260	1670,290
TinySTM	3695,958	2156,412	1191,847	858,843	1830,571	4154,775	16801,570
SwissTM	3661,437	2259,694	1245,943	882,641	1395,257	1272,226	1330,696
AdaptSTM	3730,442	2155,345	1176,561	870,528	1355,152	1369,637	2222,484
Vacation (média contenção)							
Sequencial	2716,885						
TL2	9882,320	5259,999	3000,971	2372,915	3128,798	2906,298	2912,125
TinySTM	5079,623	3011,642	1702,589	1226,283	1940,679	4491,885	16233,660
SwissTM	5117,416	3135,088	1750,622	1243,741	1768,568	1668,564	1659,829
AdaptSTM	5118,333	3024,474	1667,273	1252,813	1552,663	1788,759	2562,142
Yada							
Sequencial	1298,354						
TL2	4488,836	3025,939	2093,823	1992,218	2199,407	2603,511	2816,864
TinySTM	1859,014	1298,014	929,172	1045,314	12402,270	33460,950	101195,200
SwissTM	1793,740	1225,890	806,278	712,490	3157,629	6162,511	17741,680
AdaptSTM	1947,661	1402,171	966,824	1029,917	1269,319	11551,670	75233,500

ANEXO C PERCENTUAL DO DESVIO-PADRÃO DO TEMPO DE EXECUÇÃO (SEGUNDOS)

STM	1 *thread*	2 *threads*	4 *threads*	8 *threads*	16 *threads*	32 *threads*	64 *threads*
Bayes							
Sequencial	0,903						
TinySTM	0,343	1,141	0,854	0,988	0,294	1,181	2,014
SwissTM	0,076	0,092	0,515	0,772	0,341	0,735	0,628
AdaptSTM	0,382	0,422	0,503	0,731	0,606	0,646	0,637
Genome							
Sequencial	0,156						
TL2	0,144	2,391	2,455	4,322	2,322	5,060	6,453
TinySTM	0,099	3,868	1,611	5,418	3,599	11,403	14,216
SwissTM	0,157	3,216	0,568	5,385	2,924	2,283	5,035
AdaptSTM	0,231	2,647	0,423	2,846	4,275	8,420	13,166
Intruder							
Sequencial	0,187						
TL2	0,449	0,356	0,435	0,714	3,434	10,059	9,655
TinySTM	0,373	0,547	1,405	1,072	2,209	6,479	4,245
SwissTM	0,633	0,927	0,458	0,481	3,021	10,393	4,995
AdaptSTM	0,605	0,298	0,472	0,947	0,575	8,542	12,834
Kmeans							
Sequencial	0,058						
TL2	0,600	7,957	5,568	12,742	19,310	16,151	15,705
TinySTM	0,344	13,367	25,922	27,476	19,843	17,139	31,534
SwissTM	1,951	12,366	15,121	12,787	15,268	22,926	29,993
AdaptSTM	1,727	15,385	11,822	13,881	24,536	22,218	16,874
Labyrinth							
Sequencial	0,032						
TL2	0,145	16,052	19,441	5,388	6,163	7,036	3,915
TinySTM	0,084	0,279	7,060	4,331	2,448	3,192	1,597
SwissTM	0,029	0,301	10,480	1,936	2,799	2,435	3,562
AdaptSTM	0,124	0,289	1,585	3,526	2,814	3,454	3,447
SSCA2							
Sequencial	0,360						
TL2	0,179	0,079	0,037	0,123	1,300	9,323	10,083
TinySTM	0,285	0,102	0,146	1,014	2,302	17,725	6,909
SwissTM	0,440	2,306	0,168	0,080	0,474	2,014	7,206
AdaptSTM	0,390	0,137	0,067	0,190	2,191	4,669	3,268
Vacation (baixa contenção)							
Sequencial	0,200						
TL2	0,238	0,243	0,258	0,700	0,379	0,312	0,339
TinySTM	0,307	0,345	0,826	0,514	1,987	6,113	5,410
SwissTM	0,298	0,234	0,103	1,340	0,518	0,300	0,431
AdaptSTM	0,514	0,693	0,291	0,358	0,526	1,326	6,477
Vacation (média contenção)							
Sequencial	0,184						
TL2	0,195	0,290	0,209	0,287	0,649	1,061	1,191
TinySTM	0,265	0,247	0,534	0,388	2,299	4,984	6,428
SwissTM	0,411	0,344	2,096	1,004	1,042	1,168	0,468
AdaptSTM	0,499	0,669	1,040	0,401	0,519	1,483	5,410
Yada							
Sequencial	0,934						
TL2	0,224	0,544	1,151	1,558	1,982	1,751	1,880
TinySTM	0,379	0,568	0,468	2,068	8,730	3,320	4,342
SwissTM	0,192	0,130	0,229	0,814	3,782	7,510	14,524
AdaptSTM	0,759	0,506	0,496	1,240	1,197	10,433	7,308

ANEXO D PERCENTUAL DO DESVIO-PADRÃO DO CONSUMO DE ENERGIA (JOULES)

STM	1 *thread*	2 *threads*	4 *threads*	8 *threads*	16 *threads*	32 *threads*	64 *threads*
Bayes							
Sequencial	1,046						
TinySTM	0,531	1,117	1,014	0,965	0,506	1,327	2,144
SwissTM	0,820	0,321	0,523	1,083	0,315	1,295	1,310
AdaptSTM	1,176	0,738	0,550	0,886	1,043	1,418	1,419
Genome							
Sequencial	0,183						
TL2	1,139	2,484	2,702	4,301	2,439	5,092	7,132
TinySTM	0,222	4,231	1,572	5,612	3,336	11,849	14,955
SwissTM	0,133	3,231	0,824	5,447	2,880	8,674	5,192
AdaptSTM	0,303	2,552	0,425	2,493	4,278	9,093	13,488
Intruder							
Sequencial	1,091						
TL2	1,657	0,366	0,579	0,575	3,757	10,223	10,291
TinySTM	1,856	0,687	1,398	1,497	2,327	6,976	4,285
SwissTM	1,958	0,920	0,729	0,784	3,206	10,686	5,864
AdaptSTM	1,428	0,610	0,457	1,213	0,538	8,927	34,387
Kmeans							
Sequencial	0,531						
TL2	0,874	8,003	5,601	13,373	19,945	15,763	15,956
TinySTM	1,306	13,813	27,117	28,307	20,032	17,439	31,988
SwissTM	1,847	12,954	15,821	13,443	15,706	22,946	30,338
AdaptSTM	1,821	23,961	12,010	14,670	25,680	23,411	17,460
Labyrinth							
Sequencial	0,281						
TL2	0,540	14,701	19,853	5,147	6,334	6,851	3,930
TinySTM	0,140	0,944	7,067	4,520	2,574	3,299	1,600
SwissTM	2,212	1,097	10,576	2,514	2,804	2,740	3,785
AdaptSTM	0,856	0,456	1,634	3,615	2,912	3,490	3,476
SSCA2							
Sequencial	0,440						
TL2	2,010	0,653	0,979	0,727	1,662	10,110	11,343
TinySTM	0,252	0,306	0,485	1,004	2,673	20,042	7,217
SwissTM	1,674	2,450	1,388	0,677	0,618	2,087	8,068
AdaptSTM	0,346	1,081	0,760	0,566	2,323	5,320	3,698
Vacation (baixa contenção)							
Sequencial	0,205						
TL2	0,915	0,245	0,174	0,935	0,541	0,805	0,813
TinySTM	0,307	0,571	0,872	0,563	2,311	6,414	5,399
SwissTM	0,586	0,569	1,298	1,270	0,603	0,523	0,436
AdaptSTM	1,027	0,931	0,374	0,333	0,568	1,405	6,318
Vacation (média contenção)							
Sequencial	0,632						
TL2	1,928	0,402	0,288	0,400	0,690	1,378	1,593
TinySTM	0,629	1,201	0,477	1,380	3,178	5,390	6,585
SwissTM	1,690	0,760	1,757	0,986	1,177	1,388	0,655
AdaptSTM	0,474	1,165	1,171	0,924	0,686	2,106	5,618
Yada							
Sequencial	1,885						
TL2	2,040	0,688	1,277	1,558	1,029	0,803	1,952
TinySTM	0,649	0,678	0,692	2,142	8,884	3,374	4,728
SwissTM	1,169	0,372	0,805	0,689	3,699	7,610	14,680
AdaptSTM	0,600	0,915	0,960	1,235	1,407	10,923	7,437

ANEXO E PERCENTUAL DO DESVIO-PADRÃO DA QUANTIDADE DE CANCELAMENTOS

STM	2 *threads*	4 *threads*	8 *threads*	16 *threads*	32 *threads*	64 *threads*
Bayes						
TinySTM		72,008	205,421	97,721	140,125	135,250
SwissTM		105,409	40,182	302,858	85,938	146,686
AdaptSTM		106,598	73,509	186,678	241,313	275,770
Genome						
TL2	6,931	4,786	27,560	11,355	23,758	23,799
TinySTM	13,629	4,544	54,715	128,839	42,799	35,970
SwissTM	9,729	2,266	47,560	21,808	56,246	22,994
AdaptSTM	6,150	2,488	12,708	19,252	41,010	32,142
Intruder						
TL2	0,307	0,383	0,467	2,051	8,479	8,431
TinySTM	2,227	1,791	1,293	3,132	4,328	3,073
SwissTM	0,970	0,389	0,856	4,649	9,559	4,401
AdaptSTM	1,131	0,480	3,056	1,264	12,118	7,655
Kmeans						
TL2	7,868	5,687	12,301	18,459	16,364	15,925
TinySTM	16,861	25,423	21,257	19,622	17,530	31,855
SwissTM	15,277	14,981	12,659	16,202	25,052	30,489
AdaptSTM	34,121	23,232	31,346	54,810	42,937	72,710
Labyrinth						
TL2	5,996	16,938	7,836	10,484	19,593	7,667
TinySTM	22,448	38,409	20,253	21,915	18,470	7,586
SwissTM	5,379	8,757	10,458	7,808	14,902	6,712
AdaptSTM	5,261	10,184	11,702	5,991	33,708	7,524
SSCA2						
TL2	30,174	16,106	13,983	32,265	21,606	18,802
TinySTM	11,365	24,090	307,792	54,751	63,892	17,482
SwissTM	16,455	7,459	4,610	116,955	28,272	20,003
AdaptSTM	20,032	16,194	8,853	101,288	65,057	77,252
Vacation (baixa contenção)						
TL2	7,891	4,402	9,967	19,141	10,332	12,782
TinySTM	11,207	4,487	4,053	14,081	13,958	7,647
SwissTM	5,633	2,491	1,817	10,794	3,184	4,350
AdaptSTM	6,030	3,623	3,285	7,339	7,962	8,096
Vacation (média contenção)						
TL2	4,817	2,393	1,512	3,931	25,478	18,215
TinySTM	4,927	5,421	66,035	20,924	10,680	8,788
SwissTM	9,441	3,348	2,629	13,144	22,348	8,045
AdaptSTM	5,569	5,545	7,228	8,803	10,131	8,369
Yada						
TL2	0,541	0,868	1,071	4,213	4,204	9,808
TinySTM	1,924	0,518	4,533	9,447	3,984	4,255
SwissTM	0,307	0,706	0,499	4,299	11,316	11,529
AdaptSTM	0,407	0,491	13,119	7,739	7,563	8,869

ANEXO F CONFIGURAÇÃO DO BMC E IPMI PARA LEITURA DA POTÊNCIA CONSUMIDA

Primeiramente deve-se verificar se o sistema computacional possui suporte para a leitura de dados referente à potência consumida. Este suporte refere-se a uma ligação física entre o BMC presente na placa-mãe do computador e a fonte de alimentação do mesmo.

O BMC pode ser acessado de duas formas, via *Serial over* LAN ou via LAN. No modo de acesso via *Serial over* LAN, a comunicação é feita através de um protocolo serial pela porta dedicada ao BMC. No acesso via LAN, a comunicação é baseada em TCP/IP. Neste anexo será exemplificado a configuração para utilizar o sistema em rede LAN.

Para ativar o sistema BMC e definir as configurações de rede, deve-se configurar o *firmware* do IPMI, através do *setup* do computador. O caminho da configuração de rede do BMC no *setup* encontra-se no menu *Advanced*/IPMI *Configuration*/*Set* LAN *Configuration*. Note-se que, em outros sistemas computacionais o caminho para esta configuração pode ser diferente.

Uma vez ativo e configurado, pode-se verificar se o IP do IPMI está acessível através de uma solicitação na rede. Alguns sistemas disponibilizam uma interface *web* para gerência dos recursos do BMC. Esta interface pode ser acessada utilizando-se o IP do IPMI diretamente no navegador.

Para acessar os recursos disponíveis pelo BMC é necessário utilizar uma biblioteca que se comunique com este. No presente trabalho utilizou-se o pacote FreeIPMI versão 1.1.3 (FREEIPMI CORE TEAM, 2013).

Para solicitar as informações de potência consumida da fonte utiliza-se o seguinte comando dessa biblioteca: `ipmi-dcmi -h 10.0.1.101 -u admin -p admin --get-system-power-statistics`. O parâmetro `-h` indica o *host* onde o sistema se encontra (IP do IPMI). O parâmetro `-u` refere-se ao usuário do sistema IPMI. O parâmetro `-p` refere-se à senha deste sistema. O usuário e senha do sistema IPMI podem ser verificados no manual do sistema BMC/IPMI da placa-mãe do computador utilizado. Por fim, o comando `--get-system-power-statistics` informa a potência consumida atual, entre outras informações.