UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LEONARDO KUNZ

Memória Transacional em Hardware para Sistemas Embarcados Multiprocessados Conectados por Redes-em-Chip

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Flávio Rech Wagner
Orientador

Porto Alegre, dezembro de 2010.

"Success is not final, failure is not fatal: it is the courage to continue that counts."

Winston Churchill

AGRADECIMENTOS

Primeiramente, aos meus pais pelo apoio e dedicação incondicionais que foram fundamentais para mim, e ao meu irmão, pelo companheirismo e motivação.

Ao meu orientador Flávio Rech Wagner, pelo exemplo de dedicação e entusiasmo à pesquisa, e por todos os ensinamentos, apoio, confiança e ajuda nos momentos críticos.

Ao colega Gustavo Girão, por todo o apoio e pelas inúmeras e valiosas discussões a respeito do trabalho.

Aos outros colegas do Laboratório de Sistemas Embarcados como Tomás, Santini, Daniel, Ronaldo, Mateus, Victor, Mônica, Ulisses, Thiago, Elias, Marco, João, entre outros, que contribuíram de alguma forma para o desenvolvimento deste trabalho.

Por fim, a todos meus amigos e familiares que fizeram parte da minha vida nos últimos anos e a todos que contribuíram para minha formação pessoal e profissional.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ACK	*Acknowledgement*
EDP	*Energy-Delay Product*
HTM	*Hardware Transactional Memory*
ILP	*Instruction-Level Parallelism*
JPEG	*Joint Pictures Expert Group*
JVM	*Java Virtual Machine*
LSE	Laboratório de Sistemas Embarcados
MPSoC	*Multi-processor System-on-Chip*
NACK	*Negative acknowledgement*
NoC	*Network on Chip*
SIMPLE	*SIMPLE Multiprocessor Platform Environment*
SPM	*Scratchpad Memory*
STM	*Software Transactional Memory*
TC	*Transactional Cache*
TCC	*Transactional Coherence and Consistency*
TLP	*Thread-Level Parallelism*
TM	*Transactional Memory*
UTM	*Unbounded Transactional Memory*
VC	*Victim Cache*
VTM	*Virtual Transactional Memory*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A Memória Transacional (TM) surgiu nos últimos anos como uma nova solução para sincronização em sistemas multiprocessados de memória compartilhada, permitindo explorar melhor o paralelismo das aplicações ao evitar limitações inerentes ao mecanismo de *locks*. Neste modelo, o programador define regiões de código que devem executar de forma atômica. O sistema tenta executá-las de forma concorrente, e, em caso de conflito nos acessos à memória, toma as medidas necessárias para preservar a atomicidade e isolamento das transações, na maioria das vezes abortando e re-executando uma das transações.

Um dos modelos mais aceitos de memória transacional em hardware é o LogTM, implementado neste trabalho em um MPSoC embarcado que utiliza uma NoC para interconexão. Os experimentos fazem uma comparação desta implementação com *locks*, levando-se em consideração performance e energia do sistema. Além disso, este trabalho mostra que o tempo que uma transação espera para reiniciar sua execução após ter abortado (chamado de *backoff delay on abort*) tem impactos significativos na performance e energia. Uma análise deste impacto é feita utilizando-se de três políticas de *backoff*. Um mecanismo baseado em um *handshake* entre transações, chamado *Abort handshake*, é proposto como solução para o problema.

Os resultados dos experimentos são dependentes da aplicação e configuração do sistema e indicam ganhos da TM na maioria dos casos em relação ao mecanismo de *locks*. Houve redução de até 30% no tempo de execução e de até 32% na energia de aplicações de baixa demanda de sincronização. Em um segundo momento, é feita uma análise do *backoff delay on abort* na performance e energia de aplicações utilizando três políticas de *backoff* em comparação com o mecanismo *Abort handshake.* Os resultados mostram que o mecanismo proposto apresenta redução de até 20% no tempo de execução e de até 53% na energia comparado à melhor política de *backoff* dentre as analisadas. Para aplicações com alta demanda de sincronização, a TM mostra redução no tempo de execução de até 63% e redução de energia de até 71% em comparação com o mecanismo de *locks.*

Palavras-Chave: Memória transacional em hardware, sistemas-em-chip multiprocessados, redes-em-chip, sistemas embarcados.

Hardware Transactional Memory for NoC-based Multi-core Embedded Systems

ABSTRACT

Transactional Memory (TM) has emerged in the last years as a new solution for synchronization on shared memory multiprocessor systems, allowing a better exploration of the parallelism of the applications by avoiding inherent limitations of the lock mechanism. In this model, the programmer defines regions of code, called transactions, to execute atomically. The system tries to execute transactions concurrently, but in case of conflict on memory accesses, it takes the appropriate measures to preserve the atomicity and isolation, usually aborting and re-executing one of the transactions.

One of the most accepted hardware transactional memory model is LogTM, implemented in this work in an embedded MPSoC that uses an NoC as interconnection mechanism. The experiments compare this implementation with *locks*, considering performance and energy. Furthermore, this work shows that the time a transaction waits to restart after abort (called *backoff delay on abort*) has significant impact on performance and energy. An analysis of this impact is done using three backoff policies. A novel mechanism based on handshake of transactions, called *Abort handshake*, is proposed as a solution to this issue.

The results of the experiments depends on application and system configuration and show TM benefits in most cases in comparison to the *locks* mechanism, reaching reduction on the execution time up to 30% and reduction on the energy consumption up to 32% on low contention workloads. After that, an analysis of the *backoff delay on abort* on the performance and energy is presented, comparing to the *Abort handshake* mechanism. The proposed mechanism shows reduction of up to 20% on the execution time and up to 53% on the energy, when compared to the best backoff policy. For applications with a high degree of synchronization, TM shows reduction on the execution time up to 63% and energy savings up to 71% compared to *locks*.

Keywords: Hardware transactional memory, multiprocessor system-on-chip, network-on-chip, embedded systems.

1 INTRODUÇÃO

Com o surgimento de sistemas MPSoC (do inglês, *Multiprocessor System-on-Chip*), tornou-se necessário um mecanismo de sincronização eficiente, combinado com um modelo de programação simples para explorar ao máximo o potencial destes sistemas. Em um ambiente de memória compartilhada, programas *multi-thread* tipicamente utilizam o mecanismo de *locks* para sincronização. Entretanto, este mecanismo pode ocasionar *deadlocks* e inversões de prioridade. Além disso, o uso de *locks* para delimitar seções críticas longas limita o paralelismo, enquanto que seções críticas curtas que refinem porções paralelas do código são difíceis de programar e podem acarretar em um *overhead* significativo.

Uma alternativa promissora para sincronização é Memória Transacional (TM), proposta por Herlihy (1993). Neste modelo, o programador define regiões de código chamadas de transações que devem executar de forma atômica e em isolamento. O sistema executa as transações de forma especulativa, e, para garantir a atomicidade em caso de conflitos nos acessos aos dados, uma das transações deve abortar, implicando no descarte de suas alterações e em sua re-execução.

A TM pode ser implementada tanto em software quanto em hardware. O modelo LogTM (MOORE, 2006) é um dos mais aceitos modelos de memória transacional em hardware (HTM). Diversos trabalhos avaliam seu desempenho (TITOS, 2008) e sugerem aprimoramentos (YEN, 2007)(BOBBA, 2008)(TITOS, 2009). Entretanto, este modelo ainda não foi explorado no contexto de sistemas embarcados, apesar de ser uma escolha aparentemente adequada para este contexto devido a algumas de suas características. Transações abortam apenas quando um risco de *deadlock* é detectado, o que reduz o número de *aborts* e conseqüentemente a energia gasta. Além disso, o LogTM é implementado sobre um protocolo de coerência de cache baseado em diretório, que é adequado para *rede-em-chip* (NoC, do inglês *Network-on-Chip*), logo, adequado também para MPSoCs de muitos processadores.

Aborts representam um gargalo em sistemas de TM. Primeiramente, porque todo o trabalho realizado durante a transação se torna inútil. Segundo, porque há tempo e energia envolvida ao se desfazer das alterações. *Aborts* são necessários para a resolução de conflitos. Entretanto, *aborts* repetidos entre duas transações são desnecessários, exceto pelo último.

Uma questão crítica que há no modelo LogTM é quanto ao tempo uma transação deve aguardar após abortar antes de reiniciar sua execução. Este tempo, chamado aqui de *backoff delay on abort*, se subestimado, pode causar vários outros *aborts* e até mesmo comprometer o progresso da execução, enquanto que um tempo superestimado pode prejudicar a performance.

1.1 Objetivos

O objetivo deste trabalho é implementar o modelo LogTM em um sistema MPSoC embarcado baseado em NoC e avaliar sua performance e energia em comparação ao mecanismo de *locks*. Além disso, este trabalho tem por objetivo mostrar que o *backoff delay on abort* que otimiza a performance não é facilmente previsível e propor uma solução simples e eficiente para este problema, levando em consideração o impacto desta solução na energia do sistema.

1.2 Contribuições

Baseado em nosso conhecimento, este trabalho é o primeiro a implementar um modelo de memória transacional utilizando uma NoC como mecanismo de interconexão. Além disso, é o primeiro a avaliar a energia do modelo LogTM, levando em consideração sua implementação em um MPSoC baseado em NoC.

Além de explicar o problema do *backoff delay on abort*, é feita uma análise do impacto na performance e energia que o mau dimensionamento deste tempo pode ter no sistema. Três políticas de *backoff* são utilizadas para esta análise.

O trabalho ainda propõe um mecanismo simples, baseado em um *handshake* entre transações, para solucionar o problema do *backoff delay on abort*. Este mecanismo é chamado de *Abort Handshake*.

A primeira parte dos experimentos apresenta ganhos da TM na maioria dos casos. Houve redução em relação aos mecanismo de *locks* de até 30% no tempo de execução e 32% na energia de aplicações de baixa demanda de sincronização. Em um segundo momento, é feita uma análise do *backoff delay on abort* na performance e energia de aplicações utilizando três políticas de *backoff* em comparação com o mecanismo de *Abort handshake.* Os resultados mostram que o mecanismo *Abort handshake* apresenta redução de até 20% no tempo de execução e de até 53% na energia comparado à melhor política de *backoff* dentre as analisadas. Para aplicações com elevado grau de sincronização, a TM mostrou redução no tempo de execução de até 63% e redução de energia de até 71%.

1.3 Organização do texto

Os conceitos básicos para compreensão do trabalho são apresentados no Capítulo 2. O Capítulo 3 faz uma análise dos trabalhos relacionados. Em seguida, é apresentado no Capítulo 4 a implementação do modelo LogTM junto com uma descrição da plataforma em que foi implementado. O Capítulo 5 descreve os experimentos do trabalho, dividindo-os em duas partes principais: a primeira aborda uma comparação entre TM e *locks* utilizando aplicações de baixa contenção. A segunda utiliza aplicações de elevado grau de sincronização para analisar o *backoff delay on abort* com três políticas diferentes e com o mecanismo proposto neste trabalho. São ainda apresentados resultados da comparação entre as políticas e entre TM e *locks.*

Por fim, no último capítulo, são apresentadas as conclusões e propostas de trabalhos futuros.

2 MECANISMOS DE SINCRONIZAÇÃO EM MEMÓRIA COMPARTILHADA

Este capítulo apresenta uma fundamentação teórica, abordando conceitos básicos para compreensão do trabalho.

2.1 Sistemas MPSoC

Durante muitos anos, o aumento de performance dos processadores foi sustentado pelo aumento da freqüência de *clock* e pela exploração de paralelismo no nível de instrução (ILP, do inglês *Instruction-Level Parallelism*). Entretanto, fatores tecnológicos e de dissipação de potência atualmente limitam a aumento de freqüência enquanto que o paralelismo existente no nível de instrução para único fluxo de execução já foi explorado próximo do limite.

Por outro lado, a capacidade de integração em um único chip segue de acordo com a Lei de Moore, permitindo dobrar o número de transistores a cada dois anos, aproximadamente. Isso tornou possível o surgimento de sistemas MPSoC ou *multi-core*, que são sistemas que possuem dois ou mais processadores no mesmo chip e permitem com isso explorar o paralelismo no nível de *thread* (TLP, do inglês *Thread-Level Parallelism*). Este panorama tem provocado uma mudança no paradigma em que tanto fabricantes de processadores como programadores têm focado na programação paralela como forma de obter ganhos de performance (SUTTER, 2005).

O modelo de comunicação normalmente utilizado em sistemas MPSoC é o de memória compartilhada, que permite comunicação entre processos e *threads* através de leituras e escritas em um espaço de endereçamento compartilhado.

Sistemas MPSoC utilizam memórias cache locais para salvar dados de uso freqüente. Isto reduz o impacto da latência de acesso aos dados na memória compartilhada, entretanto, resulta em múltiplas cópias de uma mesma posição de memória no sistema. Para garantir que todas as cópias estejam sempre atualizadas, é necessário um mecanismo de coerência de cache, que pode ser baseado em diretório ou *snoop*. Os protocolos de coerência *snoop* transmitem em *broadcast* os endereços de memória para todas as caches, enquanto que protocolos baseados em diretório centralizam o controle dos blocos mantendo um diretório em cada nodo de memória.

Atualmente, sistemas MPSoC com dois e quatro processadores são comuns, e sistemas com centenas de processadores são esperados em um futuro próximo. À medida que se aumenta o número de elementos de processamento, é necessário encontrar um mecanismo de interconexão eficiente. Barramentos, normalmente utilizados, não são eficientes em termos de performance e consumo de energia para

muitos elementos. Como alternativa, surge um mecanismo de interconexão baseado em redes de computadores conhecido como *rede-em-chip* (NoC, do inglês *Network-on-Chip*). NoC (BENINI, 2002) é uma estrutura de comunicação formada por diversos roteadores conectados entre si de acordo com determinada topologia (ex. *mesh*, *torus*, cubo). Cada roteador é associado a algum elemento da rede (processadores, memórias, blocos IP especializados). NoCs permitem um elevado grau de paralelismo na comunicação, já que os *links* da rede podem operar simultaneamente com diferentes pacotes de dados.

2.2 Sincronização em Memória Compartilhada

Para tornar uma aplicação paralela é necessário dividi-la em múltiplas tarefas que podem ser desempenhadas através de *threads*, que são fluxos de execução independentes que compartilham o mesmo espaço de endereçamento da memória. Em um único processador, as *threads* normalmente são executadas de forma intercalada de acordo com alguma política de escalonamento, enquanto que em sistemas multiprocessados é possível executá-las de forma verdadeiramente simultânea, tirando-se proveito do paralelismo para reduzir o tempo de execução.

Ao programar com *threads*, o programador deve utilizar algum mecanismo de sincronização para coordenar a execução das *threads* a fim de garantir a consistência dos dados compartilhados e evitar situações em que a ordem de execução das *threads* possa influenciar no resultado da aplicação, conhecidas como **condições de corrida**. (no texto, as afirmações sobre sincronização de *threads* também são válidas para sincronização entre processos).

Condições de corrida podem acontecer em casos aparentemente simples como, por exemplo, atualizações concorrentes em uma mesma conta em um sistema bancário. A Figura 2.1 ilustra a situação em que duas *threads* sem sincronização tentam depositar R$ 50 e R$ 100, respectivamente, em uma conta com saldo de R$ 30 representada pelo endereço A da memória. No exemplo apresentado, o código que efetua o depósito, ao ser compilado, gera uma seqüência de três instruções. Caso a *thread* 0 carregue o valor inicial da conta (R$ 30), e, antes de salvar o valor acrescido de 50 (R$ 80), a *thread* 1 também leia o valor inicial para somar 100 (resultando em R$ 130), apenas uma das *threads* não terá seu depósito acrescentado na conta. Se a *thread* 0 salvar o valor 80 primeiro, este será sobrescrito pela *thread* 1, resultando em um valor de R$ 130 na conta ao final da execução, e não R$ 180, como seria o correto.

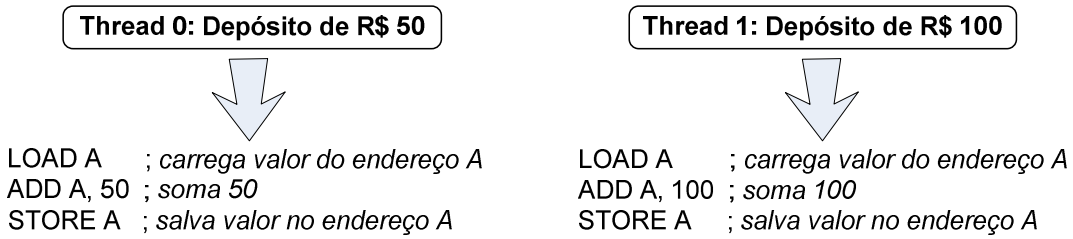

Figura 2.1: Exemplo de um sistema bancário com condição de corrida

O modelo mais intuitivo de sincronização para recursos compartilhados é a serialização (DIJKSTRA, 1965). Regiões de código que acessam dados compartilhados, chamadas de **seções críticas**, não devem executar de forma intercalada no tempo. O mecanismo de sincronização deve garantir a **exclusão mútua**, de forma que as *threads*

não executem suas seções críticas ao mesmo tempo. Este modelo de sincronização permite que o programador deixe inconsistências nos dados temporariamente sem expô-las às outras *threads*.

2.3 Locks

Locks é um mecanismo de sincronização entre *threads* que garante a consistência dos dados através da exclusão mútua. Uma variável compartilhada é utilizada para indicar se alguma *thread* está executando a seção crítica.

Existem diferentes implementações de *locks* que podem ser classificadas como **bloqueantes**, que retiram o processo ou *thread* do escalonamento, ou como **espera-ocupada** (*spin-locks*), que testam repetidamente variáveis compartilhadas até que possam prosseguir com a execução da seção crítica. A sincronização com espera-ocupada é fundamental para programação paralela em sistemas multiprocessados e é preferida sobre a bloqueante quando o *overhead* do escalonamento ultrapassa o tempo de espera, quando os recursos do processadores não são necessários para outras tarefas ou quando a solução bloqueante é inapropriada ou impossível, como por exemplo, no *kernel* de um sistema operacional (MELLOR-CRUMMEY, 1991). S*pin-locks* podem acarretar em um elevado tráfego no mecanismo de interconexão e em muitos acessos à memória. *Locks* bloqueantes, entretanto, são mais complexos do que *spin-locks*, especialmente quando incluem propriedades adicionais para garantir o acesso à seção crítica de forma justa entre as *threads*. Semáforos e monitores são exemplos de *locks* bloqueantes com semânticas mais ricas para sincronização.

Em Anderson (1990), é feita uma análise entre diversas alternativas de implementação de *spin-locks* para sistemas multiprocessados e propõem uma alternativa para reduzir o tráfego na rede.

Para ser implementado, o mecanismo de *locks* necessita que o hardware dê suporte a operações atômicas (que executam de forma indivisível do ponto de vista das outras *threads*) como *test-and-set*, *compare-and-swap*, *fetch-and-add* ou *load-linked/store-conditional*. Algumas variações como *test-and-test-and-set* (RUDOLPH, 1984) podem ser utilizadas com o objetivo de reduzir tráfego na rede e número de acessos à memória.

Apesar de ser o mecanismo de sincronização mais comumente utilizado, o modelo de *locks* possui diversas desvantagens:

Deadlocks - Para evitar que ocorram *deadlocks* ao requisitar *locks* em múltiplos objetos, os *locks* devem ser adquiridos em uma ordem específica para evitar qualquer dependência cíclica. Além de tornar a programação suscetível a erros, introduz um *overhead* para coordenar a aquisição de *locks*.

Inversão de Prioridade – Ocorre quando uma *thread* de alta prioridade é bloqueada ao tentar adquirir um *lock* que outra *thread* de baixa prioridade possui. Somente quando a *thread* de baixa prioridade terminar de executar sua seção crítica a de alta prioridade poderá prosseguir.

Contenção *x Overhead* – O programador deve estar atento à importância da granularidade da sincronização com *locks*. Seções críticas longas, que protegem toda a estrutura de dados, causam contenção às outras *threads*, limitando o paralelismo da execução. Por outro lado, a divisão em seções críticas curtas pode ser uma tarefa complexa para o programador e aumentar o *overhead* da sincronização. Além disso,

locks têm uma abordagem pessimista, já que uma *thread* deve sempre solicitar acesso exclusivo a um objeto compartilhado mesmo que durante a execução nenhuma outra *thread* acesse o mesmo objeto.

2.4 Sincronização não-bloqueante

Uma alternativa para sincronização baseada em *locks* é a sincronização não-bloqueante (TAUBENFELD, 2006), que utiliza algoritmos livres de *locks* (*lock-free*) para melhorar a performance e evitar os problemas intrínsecos dos *locks* no acesso a estruturas de dados compartilhadas. Os algoritmos não-bloqueantes baseiam-se no princípio de que conflitos de sincronização são raros e devem ser tratados como exceção. Quando um conflito é detectado, a operação é executada novamente do início.

Os algortimos de sincronização não-bloqueantes evitam *deadlocks* e inversões de prioridade, são tolerantes a falhas de *threads*, e não sofrem de degradação de performance por preempção no escalonamento, *page faults* ou *cache misses*.

Estes algoritmos são específicos da estrutura de dados, por isso aplicam-se normalmente a estruturas relativamente simples como filas, pilhas, *heaps* e listas encadeadas. São pouco utilizados na prática por serem complexos e possuírem diversas limitações, mas podem ser adequados para aplicações paralelas de larga escala. Apesar de evitarem *deadlocks*, *livelocks* são um problema em potencial. Além disso, muitos algoritmos utilizam primitivas de sincronização atômicas poderosas como LL/SC/VL (*Load-Linked, Store-Conditional, Validate*) e DCAS (*Double Compare-and-Swap*), raramente suportadas em hardware.

2.5 Memória Transacional

Memória transacional (LARUS, 2006) é uma alternativa promissora para sincronização e pode ser vista como uma generalização de algoritmos não-bloqueantes. Baseia-se na idéia de transações de bancos de dados para aplicá-la a sincronização em memória compartilhada para sistemas *multi-core*. Neste modelo, o programador define regiões de código, chamadas transações, para serem executadas atomicamente e de forma isolada (alterações são visíveis às outras *threads* apenas no final da execução). O sistema executa as transações de forma especulativa, e para garantir a atomicidade quando ocorrem conflitos nos acessos dos dados compartilhados, uma das transações deve abortar, descartando suas modificações na memória e processador, e executar novamente do início da transação. Caso não haja nenhum conflito durante a execução, a transação termina de forma bem sucedida (*commit*), tornando as alterações permanentes no sistema e visíveis às outras *threads*.

O conceito de memória transacional foi proposto pela primeira vez por (LOMET 1977), que observou que uma abstração similar a transação de banco de dados seria um bom mecanismo para garantir a consistência dos dados entre vários processos. A primeira implementação prática de memória transacional foi proposta por (HERLIHY 1993), através de um mecanismo para permitir acesso a estruturas de dados sem *locks*.

Muitas decisões de implementação compõem o espaço de projeto de um sistema de memória transacional. As principais são descritas nas subseções a seguir.

2.5.1 Detecção de conflitos

A detecção de conflitos pode classificada como *eager* ou *lazy*. A detecção *eager* verifica a cada leitura ou escrita da transação se houve algum conflito com outra transação, enquanto que a abordagem *lazy* espera até o fim da execução da transação para verificar se houve algum conflito.

2.5.2 Gerenciamento de versões

Para suportar a execução das transações, o sistema necessita de um mecanismo para gravar as escritas especulativas. A forma como o sistema trata as versões especulativas dos dados é chamada de gerenciamento de versões (*version management*). Duas abordagens são utilizadas: *eager* ou *lazy*. Na primeira, a transação atualiza diretamente a memória enquanto grava em um *log* os dados anteriores à modificação. As informações contidas no *log* serão necessárias para desfazer as atualizações no caso da transação abortar. Na segunda abordagem, as alterações são mantidas em um *buffer* privado até que a transação finalize bem sucedida, que só então são transferidas para a memória compartilhada.

2.5.3 Granularidade da transação

A granularidade da transação pode ser definida como a unidade de armazenamento sobre a qual o sistema detecta conflitos. Os níveis de granularidade podem variar desde o nível de objeto (*arrays*, listas, etc), a campos individuais nas estruturas de dados ou até linhas da cache.

Granularidades maiores, como no nível de objeto, aumentam o compartilhamento falso (*false sharing*), ou seja, casos em que o sistema detectará como conflitos acessos ao mesmo objeto, por exemplo, mesmo que esses acessos não representem conflito por estarem acessando campos diferentes do objeto.

Por outro lado, granularidades mais finas normalmente acarretam um custo maior para a detecção de conflitos.

2.5.4 Hardware *versus* software

O sistema de memória transacional pode ser implementado tanto em hardware quanto em software, ou ainda de forma híbrida. Historicamente, as primeiras implementações de memória transacional foram baseadas em hardware (HTM). Memórias transacionais em software (STM) foram propostas por pesquisadores com o objetivo principal de superar limitações inerentes às primeiras implementações em hardware.

Implementações em software são mais flexíveis que em hardware, podem ser utilizadas em processadores já existentes e permitem o uso de algoritmos mais sofisticados. Por outro lado, possuem baixa performance.

Já implementações em hardware possuem alta performance e menor consumo de potência e energia. Porém, possuem limitações impostas pelas estruturas fixas de hardware.

Algumas implementações híbridas (KUMAR, 2006) tentam combinar características das duas abordagens para superar limitações das implementações em hardware e melhorar a performance da implementação em software. Algumas delas são baseadas em software com aceleradores em hardware ou então auxiliadas por hardware. Estas

últimas são baseadas em hardware durante execução normal e recaem em software no caso de transações excederem os recursos de hardware disponíveis.

2.5.5 Memória Transacional em Software (STM)

Muitos pesquisadores são céticos quanto à viabilidade da memória transacional em software. O trabalho de Cascaval (2008) aponta diversos fatores que limitam sua performance, sugerindo que STM é apenas um "brinquedo" de pesquisa. Dragojevic (2010) contra-argumenta, mostrando que STM tem potencial para acelerar código paralelo. Entretanto, sua análise limita-se a uma comparação com a execução seqüencial das aplicações sem fazer uma comparação direta com mecanismo de *locks*, tornando a argumentação a favor de STM questionável.

Os principais fatores responsáveis pelo *overhead* e dificuldade de aceitação das diversas implementações de STM destacados nos dois trabalhos citados anteriormente são:

Custo de sincronização – Cada leitura ou escrita na memória feita dentro de uma transação é executada através de uma chamada de rotina da STM, já que esta deve manter controle de cada acesso feito pela transação. Com isso, junto das instruções de *load* e *store* são adicionadas dezenas de outras instruções.

Sobre-instrumentação pelo compilador – Os programadores precisam inserir chamadas da STM para início e fim de transação no código e substituir todos os acessos à memória dentro de transações por chamadas das STM para leitura e escrita na memória. Este processo, chamado de **instrumentação**, pode ser manual, quando os programadores substituem manualmente referências à memória pelas chamadas da STM, ou através de um compilador. Apesar de esta última opção reduzir significativamente a complexidade da programação, ela pode reduzir a performance dos programas já que o compilador não consegue determinar precisamente quais instruções acessam dados compartilhados e logo instrumenta o código de forma conservadora, provocando chamadas desnecessárias.

Código binário legado – Como as STMs devem observar toda a atividade da memória nas transações para garantir a atomicidade e isolamento, normalmente não podem suportar código binário legado que não foi instrumentado, como bibliotecas de terceiros.

Semântica – Para evitar *overheads* ainda maiores, acessos não-transacionais (leituras e escritas fora de transações) não são instrumentados. Isso enfraquece a semântica das transações, necessitando que o programador seja mais cauteloso para evitar acesso a dados compartilhados fora de transações.

2.5.6 Memória Transacional em Hardware (HTM)

Memória transactional em hardware não apresenta os problemas de performance da STM. Entretanto, sofre com dois principais problemas: primeiro, elevados custos de implementação e verificação que acarretam em riscos de projeto elevados para justificarem atualmente sua implementação pela indústria, e segundo, as limitações impostas pelas estruturas fixas de hardware que, quando não impõem limitações ao modelo de programação, acarretam perda significativa da performance e aumentam a complexidade da implementação.

A seguir são apresentados os principais modelos de HTM.

2.5.6.1 - Herlihy e Moss

O trabalho de Herlihy e Moss (HERLIHY, 1993) é a primeira implementação de memória transacional. Permite que um número limitado de leituras e escritas sejam feitas de forma atômica. Para isso, introduzem seis novas instruções ao processador que operam sobre os dados acessados na transação: *load-transactional*, *load-transactional-exclusive*, *store-transactional*, *commit*, *abort*, e *validate*. Utiliza ainda, além da cache normal, uma cache transacional que serve como *buffer* para armazenar os dados especulativos até que a transação termine (*commit* ou *abort*). Para transações que ultrapassam a capacidade de armazenamento do *buffer*, sugere um mecanismo em software para tratar esta situação. Esta cache é completamente associativa e só propaga os valores especulativos para os outros processadores e para a memória no *commit* da transação.

O protocolo de coerência de cache detecta conflitos nos acessos aos blocos. O programador deve utilizar a instrução *validate* continuamente para verificar se houve algum conflito. O programador ainda é responsável por salvar os registradores de estado e garantir o progresso das transações em caso de conflitos.

2.5.6.2 - Transactional Coherence and Consistency (TCC)

O modelo *Transactional Memory Coherence and Consistency* (TCC) (HAMMOND, 2004) é um modelo de memória transacional em hardware que utiliza execução de transações continuamente. Diferente de outras propostas que utilizam memória transacional apenas como uma forma de sincronização não-bloqueante, esta utiliza transações definidas pelo programador como a unidade básica de sincronização, coerência e consistência de memória.

Utiliza gerenciamento de versões *lazy*, armazenando as alterações da transação em um *buffer* privado, e detecção de conflitos *lazy*, detectando-os apenas na fase de *commit*. Cada transação armazena os dados de suas escritas à memória (chamado de conjunto de escritas) em um *buffer* local (a própria cache L1) que devem ser transferidos para a memória principal de forma atômica quando a transação completar sua execução. Ao finalizar a execução da transação, o hardware deve arbitrar globalmente para a permissão de *commit* e gravar o conjunto de escrita na memória compartilhada. Assim que a permissão é concedida, o processador envia os dados em *broadcast* para o resto do sistema. As outras transações lêem estes dados de *broadcast* para manter coerência de cache e detectar se foram usados dados modificados pela transação que finalizou, o que significaria que houve uma violação de dependência provocando a re-execução da transação.

Ao combinar todas as escritas de uma transação juntas, minimiza-se a sensibilidade à latência, já que menos mensagens e arbitragem entre os processos são necessárias. Por outro lado, é necessária uma maior largura de banda para transmissão para evitar que a fase de *commit* se torne um gargalo.

O modelo impõe uma ordem seqüencial entre *commits* de transações ao invés de impor ordem entre leitura e escritas individuais, como na maioria dos modelos de consistência. Todas as referências à memória de uma transação aparecem como sendo anteriores às referências da transação que realizou o *commit* posteriormente, mesmo que as referências tenham ocorrido de forma intercalada no tempo. Além de simplificar o protocolo de consistência de memória, cria a ilusão de execução por um único processsador.

O programador deve dividir o código em transações que podem executar concorrentemente. Este processo é parecido com a paralelização convencional, na qual os programadores devem encontrar e marcar regiões paralelas, com a diferença que não é necessário garantir que as regiões paralelas sejam independentes já que o TCC irá identificar as violações de dependência em tempo de execução. Transações devem ser escolhidas para minimizar o número de conflitos, já que estes impedem o ganho de performance dado pela paralelização. Transações grandes são preferíveis já que amortizam o *overhead* de *commit* e inicialização, mas também aumentam o trabalho perdido em caso de conflito. Opcionalmente, o programador pode ainda especificar uma ordem entre as transações.

Esta proposta apresenta baixa escalabilidade por exigir a serialização dos *commits* das transações, além de necessitar de mecanismos de *broadcast* e arbitragem global.

2.5.6.3- Unbounded Transactional Memory (UTM)

UTM (ANANIAN, 2005) é a primeira HTM a permitir que a transação exceda limites do *buffer* de hardware e a permitir troca de contexto durante a transação. Propõe extensões arquiteturais para separar o estado da transação e detecção de conflitos das caches e do protoloco de coerência de cache. Para isso, introduz uma estrutura de dados residente em memória para manter os dados da transação, incluindo conjunto de leitura e escrita de todas as transações do sistema. Em cada bloco de memória do sistema são acrescentados bits de acesso e ponteiro para o *log* que armazena os dados anteriores à transação. Cada leitura ou escrita, inclusive fora de transação, deve verificar o ponteiro e os bits de acesso para detectar conflitos. Com isso, são necessários vários acessos adicionais à memória para leituras e escritas das transações.

O mesmo trabalho de (ANANIAN, 2005) também propõe uma versão simplificada chamada de LTM, pois os próprios autores consideram a UTM complexa demais para ser implementada. A LTM não permite trocas de contexto nem paginação dos dados transacionais e também apresenta baixa performance para transações que excedem o *buffer* da cache.

2.5.6.4- Virtual Transactional Memory (VTM)

O trabalho de RAJWAR (2005) propõe a VTM, que tem como foco a virtualização dos recursos do hardware no sentido de esconder do programador limitações dependentes da plataforma como tamanho de buffers, *quantum* de escalonamento, *page faults* e migração de processos. VTM utiliza estruturas de dados residentes na memória virtual que armazenam os dados de transações que ultrapassaram o limite da cache ou excederam seu tempo de escalonamento. O estado das transações que não excedem o tamanho dos *buffers* são mantidas em hardware para não afetar a performance desses casos, diferentemente do que ocorre na UTM, em que a performance do caso normal também é afetada.

2.5.6.5– TCC Escalável (Scalable TCC)

Em (CHAFI, 2007) é proposto o modelo TCC Escalável, uma variação do TCC (HAMMOND, 2004) para sistemas baseados em diretório. Ao contrário de outras propostas escaláveis de memória transacional, esta detecta conflitos apenas quando a transação está pronta para o *commit* (detecção de conflitos *lazy*) para garantir uma implementação livre de *livelocks* sem necessidade de intervenções no nível do usuário com políticas de contenção.

O modelo TCC original opera sob a condição de que a execução das transações podem ser simultâneas (sobrepostas no tempo), mas apenas uma pode efetuar o *commit* por vez. *Commits* seqüenciais limitam a concorrência e se tornam um gargalo quando há muitos processadores. A variação escalável do TCC proposta baseia-se na condição de que quando não há conflitos, as transações podem sobrepor completamente suas fases de execução e *commit*.

O uso de múltiplos diretórios permite diversas otimizações. Primeiro, apesar de cada diretório permitir o *commit* de uma única transação por vez, múltiplas transações podem fazer o *commit* em paralelo em diretórios diferentes. Assim, um número maior de diretórios permite um grau maior de paralelismo. *Commits* paralelos valem-se da localidade dos acessos de cada transação. Segundo, permitem um protocolo de *write-back* que move dados entre nodos apenas no compartilhamento verdadeiro ou substituição de dados na cache. Por último, permitem filtragem do tráfego de *commit* e eliminam a necessidade de mensagens de invalidação em *broadcast*.

Os diretórios são usados para verificar quais processadores possuem dados lidos de forma especulativa. Quando um processador está em sua fase de validação, adquire um identificador da transação (TID) e só prossegue para sua fase de *commit* quando for garantido que nenhum outro processador irá violar a transação.

Algumas regras devem ser seguidas para garantir o funcionamento correto do protocolo com *commits* paralelos. Primeiro, escritas conflitantes para o mesmo endereço são serializadas. Segundo, uma transação com TID deve aguardar o *commit* de todas as outras transações com TIDs menores antes de realizar seu *commit*.

O algoritmo pode ser descrito através de cinco etapas principais:

1. Obter TID – Um agente centralizado fornece um ID para a transação. Os TIDs impõem uma ordem nos *commits* das transações.

2. Testar diretórios do conjunto de escrita – Para cada diretório no conjunto de escrita de uma transação é enviada uma mensagem de teste para verificar se transações anteriores (com TIDs menores) já enviaram suas escritas para o diretório. Para cada diretório que não faz parte do conjunto de escrita da transação, é enviada uma mensagem de *skip* para que o diretório saiba que não precisará aguardar escritas desta transação.

3. Enviar mensagens *mark* – Para cada linha da cache no conjunto de escrita é enviada uma mensagem de *mark* ao diretório correspondente. Esta mensagem comunica ao diretório que a linha da cache marcada terá seu estado alterado para *Owned* quando a mensagem final de *commit* for enviada pela transação.

4. Testar diretórios do conjunto de leitura – Para cada diretório do conjunto de escrita é enviada uma mensagem de teste para verificar se as linhas da cache correspondentes já receberam notificação de escrita por transações anteriores. Se a verificação for bem sucedida, a transação terá certeza de que não terá que abortar devido a alguma escrita de uma transação anterior. Testes são enviados periodicamente até serem bem sucedidos.

5. Enviar mensagens de *commit* – Uma mensagem de *commit* é enviada para cada diretório do conjunto de escrita da transação. As linhas da cache marcadas terão seu estado modificado para *Owned* e mensagens de invalidação são enviadas para as outras caches que compartilham as mesmas linhas. Estas mensagens de invalidação podem

fazer com que uma transação mais nova (TID menor) seja abortada se a linha da cache em questão pertencer ao conjunto de leitura da transação.

Em resumo, o algoritmo utiliza primeiro um agente centralizado para impor uma ordem nas transações. Duas transações podem proceder com o algoritmo de *commit* em paralelo desde que os diretórios de seus conjuntos de leitura e escrita sejam distintos. Caso as transações precisem acessar o mesmo diretório, serão serializadas baseadas nos TIDs das transações. O algoritmo é livre de *deadlocks* e *livelocks* já que as transações são assinaladas com TIDs crescentes por um agente centralizado e uma transação nunca é forçada a esperar por outra com TID maior.

2.5.6.6 – LogTM

Considerando que *commits* são mais freqüentes do que *aborts*, LogTM (MOORE, 2006) utiliza gerenciamento de versão *eager* permitindo que transações escrevam diretamente na memória enquanto dados antigos da memória são salvos em uma estrutura de *log*. Quando uma transação finaliza com *commit*, os novos dados tornam-se permanentes na memória. Quando aborta, a memória deve ser restaurada com os dados antigos salvos na estrutura de *log*. Esta abordagem torna *commits* rápidos, mas penaliza *aborts*.

LogTM utiliza detecção de conflitos *eager*, o que significa que detecta conflitos a cada leitura ou escrita, ao invés de esperar até o fim da transação. A detecção de conflitos é feita através das requisições de bloco, necessitando apenas de algumas modificações no protocolo de coerência de cache.

Em um *cache miss*, o controlador da cache requisita o bloco ao diretório, o qual possui uma tabela que indica qual bloco está em qual cache do sistema. Se alguma outra cache do sistema possuir cópia do bloco requisitado, então a requisição é encaminhada para os processadores que possuem cópia do bloco para que estes possam detectar o conflito. Os processadores podem negar o pedido em caso de conflito ou então podem agir conforme o protocolo original, invalidando ou enviando o bloco em *write-back*, dependendo da situação.

Cada cache utiliza dois bits por bloco para identificar quais blocos foram lidos ou escritos (R e W, respectivamente) durante a transação para permitir a detecção de conflitos. Quando a transação finaliza, os bits são zerados.

Quando o controlador da cache detecta algum conflito, envia uma mensagem de NACK para o processador que fez a solicitação que causou o conflito. Este, por sua vez, ao receber a mensagem de NACK entra em *stall*, caso não haja dependência cíclica que possa causar *deadlock*, ou aborta, caso uma dependência cíclica seja detectada.

Quando a transação entra em *stall*, ela segue enviando requisições até que o bloco seja liberado pela outra transação. No caso da transação abortar, todos os dados modificados na memória são restaurados com os valores salvos no *log*, o processador faz um *rollback* do seu estado e a transação executa novamente a partir do início após um período de espera.

O LogTM suporta transações ilimitadas já que blocos transacionais substituídos da cache são mantidos em um estado *sticky* no diretório, permitindo que solicitações sejam encaminhadas para o processador "dono" do bloco para a detecção de conflitos.

2.5.6.7– LogTM-SE

LogTM-SE (YEN, 2007) é uma evolução do modelo LogTM com o objetivo de permitir virtualização, suportando substituição de blocos da cache, aninhamento de transações ilimitado (transações dentro de transações), mudança e migração de contexto e paginação dos dados transacionais. Para isso, utiliza uma detecção de conflitos baseada em *signatures*, semelhante ao modelo Bulk (CEZE, 2006). A utilização de *signatures* separa a detecção de conflitos das caches, evitando assim que estruturas críticas do projeto de processadores como *arrays* e *tags* das caches L1 sejam alteradas por uma idéia ainda emergente como memória transacional. Além disso, permite o suporte a virtualização ao permitir que os estados das transações sejam acessíveis por software e sejam salvos em trocas de contexto e aninhamento de transações.

Signature é uma forma compacta de representar os endereços lidos e escritos das transações. É baseada em um filtro de Bloom, que permite verificar se algum elemento (endereço) está contido em um conjunto (conjunto de leitura ou escrita), o que indica conflito entre transações. Permite falso positivo, o que causa conflitos desnecessários, entretanto, falso negativo não é possível, garantindo assim a execução correta sem violar a atomicidade. A taxa de falsos conflitos, e conseqüentemente a performance, dependem do tamanho das *signatures* e da codificação *hash* utilizadas. Transações longas também tendem a aumentar a taxa de conflitos, penalizando a performance.

2.5.6.8– TokenTM

TokenTM (BOBBA, 2008) também utiliza o gerenciamento de versões do LogTM em conjunto com um novo mecanismo de detecção de conflitos baseados em *tokens*, adaptado da coerência de cache por *tokens*. Este mecanismo permite virtualizar as transações sem penalizar transações longas como acontece com o LogTM-SE.

Transações devem adquirir *tokens* associados aos blocos para leitura e escrita. Caso não consiga adquirir os *tokens* necessários, a transação detecta um conflito. Os *tokens* são liberados quando a transação é finalizada. As informações dos *tokens* são salvos em meta-estados tanto em hardware, para permitir detecção de conflitos rápida, como na memória em *logs* acessíveis por software.

Para permitir compartilhamento de blocos para leitura, introduz um mecanismo de fissão e fusão de meta-estados. Transações pequenas que mantêm todos seus dados na cache liberam os *tokens* em tempo constante. Transações longas, porém, devem percorrer o *log* para liberar os tokens, tanto em um *commit* como em um *abort*, o que pode representar um *overhead* significativo.

TokenTM evita modificações no protocolo de coerência de cache, já que estes são difíceis de verificar. Os dados de meta-estado são transmitidos com as mensagens do protocolo padrão, evitando novas mensagens como NACKs e estados *sticky* como no LogTM.

2.5.6.9- Considerações Finais

A Tabela 2.1 sintetiza as principais informações dos modelos de memória transacional em hardware apresentados, incluindo as seguintes características de suporte a virtualização de transações:

- Transações Ilimitadas (TI) - O modelo permite que transações possuam tamanho ilimitado, ou seja, o tamanho de seus conjuntos de escrita e leitura não está limitado ao tamanho de um *buffer* em hardware (cache).

- **Chaveamento de Contexto (CC)** – Permite que haja troca de contexto durante a execução de uma transação, sem que esta precise ser abortada.

- Paginação (P) – Dados pertencentes ao conjunto de escrita e leitura da transação podem ser salvos no disco pela memória virtual do sistema operacional durante a transação, podendo ser alocados novamente na memória em endereços físicos diferentes.

- Aninhamento (A) – Permite que o programador "aninhe" uma transação dentro de outra.

Tabela 2.1: Resumo das características dos principais modelos de HTM

Modelo	**Principais Características**	**Coerência de cache**	**TI**	**CC**	**P**	**A**
Herlihy & Moss (1993)	- Gerenciamento de versão *lazy* - Detecção de conflitos *eager* - Primeira implementação de memória transacional - Utiliza uma cache transacional (TC) para salvar os dados especulativos da transação	*Snoop*				
TCC (2004)	- Gerenciamento de versão *lazy* - Detecção de conflitos *lazy* - Execução contínua de transações - Substitui protocolo de coerência de cache - Livre de *deadlocks* e *livelocks*	*Snoop*				
UTM (2005)	- Gerenciamento de versão *eager* - Detecção de conflitos *eager* - Exige estruturas de hardware demasiadamente complexas	Diretório	✘	✘	✘	
LTM (2005)	- Gerenciamento de versão *lazy* - Detecção de conflitos *eager* - Simplificação "implementável" da UTM	Diretório	✘			
VTM (2005)	- Gerenciamento de versão *eager* - Detecção de conflitos *eager* - Alta complexidade e baixa performance - Transações longas são tratadas em software	Diretório	✘	✘	✘	

LogTM (2006)	- Gerenciamento de versão *eager* - Detecção de conflitos *eager* - Armazena em um *log* o endereço e valor antigo a cada escrita - Aloca espaço na memória virtual para o *log* - Modifica protocolo de coerência de cache - *Commits* rápidos mas *aborts* lentos - Requer políticas de controle de contenção para evitar *livelocks*	Diretório	✘			
TCC Escalável (2007)	- Gerenciamento de versão *lazy* - Detecção de conflitos *lazy* - Adaptação do TCC para sistemas baseados em diretório - Permite *commits* paralelos em diferentes diretórios, desde que suas transações não tenham conflitos entre si - Livre de *deadlocks* e *livelocks*	Diretório				
LogTM-SE (2007)	- Gerenciamento de versão *eager* - Detecção de conflitos *eager* - Utiliza *signatures* para representar os conjuntos de escrita e leitura de forma compacta, porém esse mecanismo permite falsos positivos provocando conflitos desnecessários	Diretório	✘	✘	✘	✘
TokenTM (2008)	- Gerenciamento de versão *eager* - Detecção de conflitos *eager* - Utiliza *logs* como na LogTM em conjunto com novo esquema de detecção de conflitos baseado em *tokens* - Associa um meta-estado a cada bloco da memória - Não modifica a semântica do protocolo de coerência de cache	Diretório	✘	✘	✘	✘

O modelo LogTM foi escolhido para ser implementado neste trabalho por apresentar um conjunto de características aparentemente adequadas para um MPSoC embarcado:

- Transações só são abortadas em conflitos quando há risco de *deadlock* (possível ciclo de dependências é detectado). Transações abortadas representam trabalho inútil e conseqüentemente energia desperdiçada.

- Utiliza um protocolo de coerência de cache baseado em diretório, o que o torna adequado para uma NoC, já que a comunicação é feita ponto-a-ponto. Protocolos de

coerência de cache *snoop* necessitam de mensagens de *broadcast*, as quais não são suportadas de forma eficiente em uma NoC.

- Não necessita de hardware complexo para ser implementado em comparação com os demais modelos de memória transacional.

- Serve de base para outros modelos como LogTM-SE e TokenTM, que fazem uso de mecanismos de detecção de conflitos mais avançados para permitir a virtualização das transações. Embora o LogTM não permita chaveamento de contexto, paginação e nem aninhamento de transações (permite somente transações ilimitadas), estas restrições não representam grandes desvantagens no contexto de sistemas embarcados.

3 TRABALHOS RELACIONADOS

Esta seção apresenta trabalhos que levam em consideração consumo de energia de implementações de HTM. Trabalhos sobre STM não serão abordados já que não são atraentes do ponto de vista energético e foram deixados de fora do escopo deste trabalho. Uma análise do consumo de energia de um sistema STM pode ser vista em Klein (2009).

Todos os trabalhos aqui descritos baseiam sua implementação em algum dos modelos clássicos de memória transacional descritos no Capítulo 2. Além de avaliarem o consumo de energia e performance, os trabalhos apresentados neste capítulo propõem alguma modificação arquitetural com o objetivo de reduzir o consumo de energia do sistema.

Não foi encontrada na literatura nenhuma implementação de HTM que utilize uma NoC como mecanismo de interconexão. Isto deixa espaço para contribuição deste trabalho, já que é interessante avaliar o impacto que um mecanismo de memória transacional terá em um sistema interconectado através de uma NoC, que permite maior paralelismo na comunicação. O mecanismo de interconexão de sistemas multiprocessados representa um componente chave que afeta diretamente a performance e energia do sistema. Além disso, acredita-se que, para explorar todo o potencial da memória transacional, é necessário reduzir o gargalo de comunicação entre os processadores, já que este é um fator que limita a performance de aplicações paralelas.

3.1 Moreshet 2005

O trabalho de Moreshet (2005) é o pioneiro em avaliar consumo de energia de memória transacional em hardware. Sua implementação baseia-se no modelo de Herlihy (1993), no qual há uma cache transacional completamente associativa para salvar os valores especulativos. Utiliza o simulador Simics (MAGNUSSON, 2002) para modelar um sistema com processadores UltraSPARC II conectados por barramento, assumindo dados de energia e latência para uma memória compartilhada *off-chip*.

O trabalho apresenta também um mecanismo para serializar a execução de transações após um conflito com o objetivo de evitar novos conflitos e conseqüentemente reduzir a energia para os casos em que a taxa de conflitos é alta. Apesar de indicar que a memória transacional pode reduzir a energia em comparação com o mecanismo de *locks*, e que seu mecanismo de execução serial pode reduzi-lo ainda mais, o trabalho apresenta diversos problemas que tornam seus resultados pouco convincentes. Apenas um *microbenchmark* com apenas uma configuração de sistema (quatro processadores) é utilizado para obtenção dos dados. Não são explicados detalhes

de como é implementado o mecanismo de execução serial e, além disso, não são mostrados resultados de performance, apenas resultados de energia.

3.2 Moreshet 2006

Moreshet (2006) apresenta uma extensão do trabalho (MORESHET, 2005) descrito anteriormente. Utilizando a mesma arquitetura, avalia o consumo de energia da memória transacional em comparação com *locks* através de quatro *benchmarks* do SPLASH-2 (WOO, 1995). Apenas uma das aplicações (fmm) apresenta um consumo de energia maior com TM, enquanto que nas outras a TM mostra ganhos significativos de energia. Por considerar que as aplicações do SPLASH-2 possuem taxas de conflito muito baixas, utiliza um *microbenchmark* com duas configurações que supostamente apresentam taxas de contenção altas.

Apesar de apresentar mais detalhes do modo de execução serial e considerar sua performance, além da energia, a avaliação do modo serial é feita somente através do *microbenchmark*, o que novamente torna seus resultados pouco convincentes. Além disso, são dados poucos detalhes sobre o *microbenchmark* e suas duas configurações.

3.3 Ferri 2007

O primeiro trabalho a avaliar energia de uma HTM em uma plataforma embarcada é o de Ferri (2007). Assim como os trabalhos citados anteriormente, seu modelo de HTM é baseado no de Herlihy (1993). Utiliza o simulador MPARM para simular um sistema com uma quantidade variável de processadores ARM7 interconectados por um barramento compatível com a arquitetura de comunicação AMBA. O protocolo de coerência de cache (MESI) é implementado através de dispositivos *snoop* conectados às portas mestres do barramento. Um exemplo de configuração do sistema é apresentado na Figura 3.1. Cada *core* possui uma cache L1 com mapeamento direto de 8KB e uma memória privada de 256KB. Há no sistema uma memória compartilhada de 256KB.

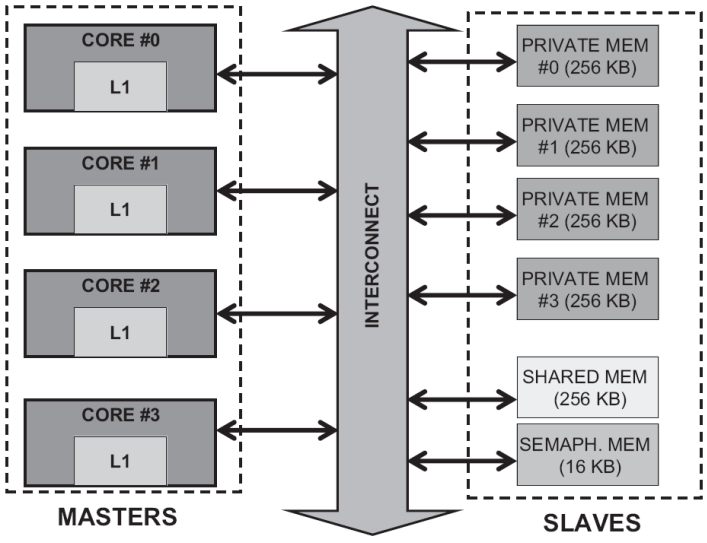

Figura 3.1: Configuração do sistema para quatro processadores. Retirado de Ferri (2007).

Uma visão geral da arquitetura de cada core é mostrada na Figura 3.2. Além da cache L1, há uma cache transacional (TC) completamente associativa de 512B para armazenar os dados da transação. Uma memória *scratchpad* (SPM) de 128B serve para salvar os registradores no início da transação.

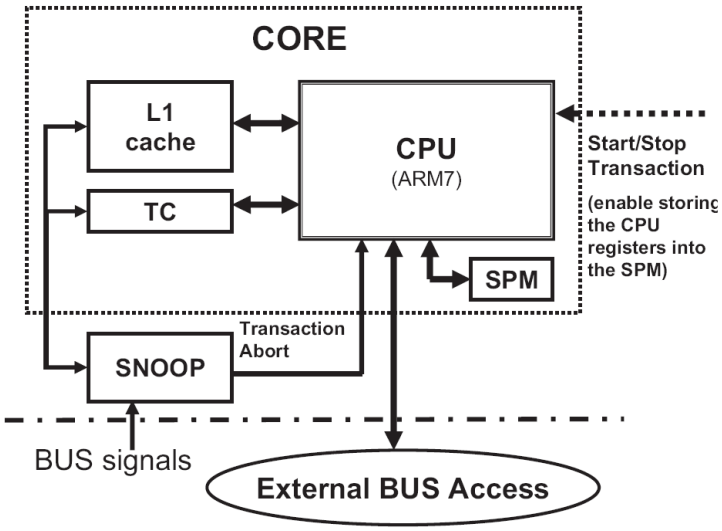

Figura 3.2: Visão geral da arquitetura. Retirado de Ferri (2007).

Em caso de conflito, o dispositivo *Snoop* notifica o processador que deve abortar e invalidar os dados na TC escritos pela transação. Após restaurar os registradores com os dados na SPM, o processador entra em modo de baixa potência por um período de *backoff* randômico exponencial, ou seja, o tempo de espera é um valor randômico no intervalo definido pelo tempo de espera inicial (menor do que 100 ciclos) que dobra a cada vez que há um novo conflito.

Para avaliar os ganhos de performance e energia da memória transacional, o trabalho utiliza um *microbenchmark* em que os processadores acessam uma matriz subdividida em regiões sobrepostas. Os dados das regiões sobrepostas são acessados dentro de seções críticas. Quatro configurações para o *microbenchmark* são apresentadas, classificadas quanto ao número de acessos dentro e fora das seções críticas, de forma que a porção de acessos dentro de seções críticas varia de 5% até 85%. Os experimentos foram feitos com 4, 8 e 10 processadores.

Os resultados mostram que para a configuração de alta contenção (85% de acessos dentro da seção crítica) há uma redução de até 31% na energia do sistema e redução de 62% no tempo de execução para 8 processadores. Com taxas de contenção baixas (5%) as diferenças de performance são insignificantes e há um maior consumo de energia pela TM. A principal razão para isso está no consumo da TC, que é acessada em paralelo com a cache L1 mesmo quando o processador não está em transação. Este consumo representa em torno de 20% do total de energia do sistema e é comparável ao consumo dos processadores.

3.4 Ferri 2008

Ferri (2008) estende o trabalho anterior (FERRI 2007) levando em consideração um conjunto maior de *benchmarks* (*red-black tree* e *skip-list*, além do mesmo *microbenchmark*) e adotando um mecanismo que possibilita desligar a TC com o objetivo de economizar energia quando esta não é utilizada.

Para suportar transações que excedam a capacidade da TC em sua arquitetura, Ferri (2008) utiliza uma abordagem semelhante a do modelo TCC (HAMMOND, 2004). Ao ocorrer um *overflow* da cache, todas as outras transações do sistema são paradas para permitir que a transação que excedeu a capacidade da TC utilize diretamente toda a hierarquia de memória para seus dados. O trabalho justifica esta abordagem por considerar raros os casos de *overflow* da TC, e, por ser um sistema embarcado com restrições de recursos, um sistema mais elaborado para tratar estes casos não se justificaria. Além disso, aplicações embarcadas normalmente são conhecidas anteriormente, o que permitiria dimensionar a TC de forma a evitar casos de *overflow*.

Para permitir que a TC seja desligada ao final de uma transação, é necessário fazer o *write-back* dos dados. Para isso, o trabalho apresenta duas alternativas, uma em que os dados são enviados para a cache L2 e outra em que são enviados para a L1. Enviar para a cache L2 aumenta o tráfego na rede e, como há grandes chances de que esses dados serão utilizados no futuro, o processador terá que buscá-los novamente na L2. A alternativa de salvar os dados na L1 é mais complicada, já que um *write-back* para a L1 pode causar um *write-back* para a L2.

Os resultados mostram que, à medida que aumentam o número de cores e a proporção de tempo gasto nas seções críticas, maior é a vantagem da TM sobre *locks*. Esta vantagem se deve à maior quantidade de tráfego necessária para manter a consistência no mecanismo de *locks*, que justifica a energia adicional gasta pela TC na maioria dos casos. A política de desligamento da TC pode resultar em economia de até 17% de energia, entretanto, para aplicações em que a TC é muito utilizada, esta política pode trazer desvantagens. A abordagem de enviar os dados da TC em *write-back* para a cache L1 mostrou-se mais eficiente do que o *write-back* para a L2.

3.5 Ferri 2010-a

Após constatar o elevado consumo de energia da TC nos trabalhos anteriores, Ferri (2010-a) investiga uma modificação em sua arquitetura para suprimir a necessidade da TC. Para isso, utiliza a cache L1 para salvar também os dados especulativos. Como a associatividade da cache L1 é restrita devido a fatores de consumo de energia, a solução encontrada para reduzir a probabilidade de substituição de dados transacionais (*overflow*) é a de utilizar uma *victim cache* (VC)(JOUPPI, 1990). Conforme descrito nos trabalhos anteriores, nesta arquitetura o *overflow* de dados transacionais da cache faz com o sistema entre em modo de execução serial pouco eficiente, bloqueando a execução de todas as outras transações e permitindo que a transação que ultrapassou a capacidade de sua cache utilize toda a hierarquia de memória. A VC é acessada somente quando ocorre *miss* na cache L1, permitindo assim um menor consumo de energia se comparada com a TC, que é acessada em paralelo com a L1. O sistema entra em modo de execução serial apenas quando há *overflow* da VC.

O trabalho compara a nova arquitetura (chamada de *TM-victim*) com as duas arquiteturas propostas anteriormente: *vanilla-TM*, arquitetura original proposta em Ferri

(2007) e *TM-aggressive-L1WB*, proposta em Ferri (2008). Para a avaliação, utiliza três aplicações do STAMP (*Vacation*, *K-means* e *Genome*) e ainda dois *benchmarks* de operações em estruturas de dados complexas: *Red-Black Tree* e *Skiplist*. O trabalho explora ainda diferentes tamanhos e associatividades das caches, fazendo uma análise abrangente das arquiteturas e explorando os *trade-offs* envolvidos nas diferentes configurações das caches e nos casos de *overflow*.

3.6 Ferri 2010-b

Ferri (2010-b) apresenta um resumo das diversas propostas e decisões de projeto dos trabalhos anteriores, estendendo a análise das propostas através de um conjunto maior de dados. Além disso, propõe modificações na detecção de conflitos.

Segundo os autores, aplicações com altas taxas de conflitos têm baixa performance com detecção de conflitos *eager*. Como em seu sistema o processador que faz a requisição sempre vence em caso de conflito, transações longas podem ser abortadas repetidamente por transações curtas. Esta situação pode gerar altas taxas de *abort*, baixa performance e elevado consumo de energia. Para evitar isso, sugere que o sistema entre em um modo de execução serial (também chamado de serialização forçada) assim que a taxa de *aborts* ultrapassar determinado limite. Esta abordagem é atrativa por sua simplicidade, já que o hardware já possui suporte para execução serial para tratar casos de *overflow*.

A segunda alternativa proposta é adotar detecção de conflitos *lazy*. Apesar de mais complexa, essa alternativa permite filtrar falsos conflitos (*write-after-read*) que não podem ser detectados com a abordagem *eager*. Na forma como foi implementada nesta arquitetura, os conflitos são detectados à medida que ocorrem, mas sua resolução é adiada para o fim da transação. Cada processador mantém uma tabela para identificar conflitos com outros processadores. Ao finalizar a execução da transação, o processador inicia a fase de *commit* fazendo com que todos os processadores em transação do sistema suspendam sua execução. Então, o processador no *commit* transmite em *broadcast* no barramento o conjunto de endereços escritos, fazendo com que os outros processadores abortem caso houver um conflito.

A política de serialização forçada resultou em melhora de até 17% do EDP (do inglês, *energy-delay product*), enquanto que a abordagem de detecção de conflitos *lazy* reduziu o número de *aborts* de 29% para 9%, em média, e melhorou o EDP em 14%. Entretanto, para algumas aplicações a abordagem *lazy* resultou em um EDP 30% pior devido ao *overhead* da fase de *commit*. De forma geral, a detecção de conflitos *lazy* melhora a performance de aplicações com taxas altas de conflito, mas prejudica as de taxas baixas e aumenta a complexidade do hardware.

3.7 Sanyal 2009

Sanyal (2009) propõe um mecanismo para economizar energia e melhorar a performance do modelo TCC-Escalável (CHAFI, 2007). Sua abordagem consiste em desligar os processadores através da técnica de *clock-gating* após o *abort* de uma transação. Seu objetivo é evitar que as mesmas transações entrem em conflito novamente e causem um novo *abort*, o que significa energia desperdiçada e possível degradação da performance de outras transações por aumentar a contenção no mecanismo de interconexão e na memória do sistema.

O mecanismo proposto no trabalho é uma política de gerenciamento de contenção baseada em *backoff* combinada com mecanismo de *clock-gating*. Os processadores que abortam entram em *clock-gating* por um período que depende do número de *aborts* que a transação sofreu e do estado da transação conflitante. Em cada diretório é acrescentada uma tabela que contém informações sobre os processadores abortados usados para controlar o período de *clock-gating*, calculado conforme a fórmula da Figura 3.3.

$$W_t = W_0(2^{\lceil \lg N_a \rceil} + 2^{\lceil \lg N_r \rceil})$$

Figura 3.3: Cálculo do período de *clock-gating*. Retirado de Sanyal (2009).

O período de *clock-gating* (W_t) depende de uma constante inicial (W_0) que deve ser ajustada levando-se em consideração o número de processadores no sistema. Segundo os autores, W_0 deve ser pequena para sistemas com muitos processadores (presumindo que a taxa de *aborts* será alta) e grande para sistemas de pequena escala. O cálculo depende ainda do número de *aborts* que a transação sofreu (N_a) e um valor chamado de *renew count* (N_r). Ao expirar o tempo de *clock-gating*, o diretório envia mensagem ao processador que causou o *abort* para verificar se a transação que está executando possui um *id* (endereço do contador de programa de início da transação) igual ao da transação que causou o *abort*. Se os *ids* forem iguais, o contador *renew count* é incrementado e o processador continua em *clock-gating,* renovando seu período de espera após um novo cálculo de W_t.

É importante ressaltar aqui que a política de renovar o período de *clock-gating* se uma transação com *id* igual ao da que provocou o *abort* estiver executando não é uma serialização das transações. Apesar de os *ids* poderem ser iguais, o que indica um *loop* no código, a transação não é a mesma. Em um sistema com detecção de conflitos e gerenciamento de versão *lazy* como o TCC-Escalável, uma transação aborta outra apenas quando estiver em sua fase de *commit* após terminar sua execução. Qualquer conflito entre transações causa *abort* e re-execução da que não finalizou, provocando naturalmente a serialização das transações conflitantes.

A simulação do sistema é feita no simulador M5 utilizando configurações de 4, 8 e 16 processadores Alpha 21264 com cache de 64KB de associatividade *2-way*. Os processadores e diretórios do sistema são interconectados por um barramento do tipo *split-transaction*. Três aplicações de *benchmark* do STAMP foram utilizadas: *genome*, *yada* e *intruder*.

Os resultados para as três aplicações e três configurações de sistema (4, 8 e 16 processadores) apresentaram uma média ganhos de performance de 4% e redução de energia de 19%.

3.8 Considerações Finais

Os trabalhos apresentados neste capítulo mostram os principais trabalhos que abordam consumo de energia de sistemas de memória transacional. A maioria dos trabalhos possui uma implementação baseada no modelo de Herlihy, com exceção de Sanyal (2009), que implementa o modelo TCC-Escalável. Não há nenhum trabalho que avalie o consumo de energia de um sistema baseado no LogTM. Apenas os trabalhos de Ferri consideram o uso de memória transacional em um sistema embarcado, enquanto que os outros trabalhos fazem sua avaliação em sistemas de propósito geral.

4 IMPLEMENTAÇÃO LOGTM

Este capítulo descreve a implementação do LogTM na plataforma SIMPLE. A Seção 4.1 apresenta uma breve descrição da plataforma SIMPLE e a Seção 4.2 explica detalhes da implementação. Em seqüência, a Seção 4.3 apresenta o problema do *backoff delay*, enquanto que uma solução proposta para este problema é apresentada na Seção 4.4.

4.1 SIMPLE

A implementação e os experimentos realizados neste trabalho foram feitos em uma plataforma virtual chamada de SIMPLE (*SIMPLE Multiprocessor Platform Environment*) (BARCELOS, 2008), desenvolvida no Laboratório de Sistemas Embarcados (LSE) da Universidade Federal do Rio Grande do Sul. Esta plataforma foi descrita em SystemC e emula um ambiente MPSoC baseado em NoC com precisão de ciclo.

A plataforma SIMPLE é composta por um número parametrizável de processadores Java e suporta diferentes organizações de memória definidas em tempo de projeto, como distribuída, compartilhada, compartilhada distribuída e uma organização de memória fisicamente centralizada, mas logicamente distribuída. Além disso, as configurações permitem que o usuário especifique tamanhos de cache, políticas de substituição, associatividade e tamanho de bloco.

4.1.1 FemtoJava

O processador utilizado na plataforma SIMPLE é uma descrição em SystemC do processador FemtoJava multiciclo (ITO, 2001). O FemtoJava é um processador que executa nativamente *bytecodes* Java através de uma máquina de pilha compatível com a especificação da Máquina Virtual Java (JVM, do inglês *Java Virtual Machine*). Utiliza arquitetura Harvard, possuindo, assim, a cache de dados separada da cache de instruções.

4.1.2 SoCIN

Como mecanismo de interconexão, é utilizada uma descrição em SystemC da rede SoCIN (ZEFERINO, 2003). A comunicação é feita através de pacotes compostos por *flits* (*Flow Control Unit*).

A topologia da SoCIN é de grade (*mesh*) 2-D. Cada roteador possui cinco portas bi-direcionais (norte, sul, leste, oeste e local) nas quais há *buffers* somente na entrada. O roteamento dos pacotes é determinístico, baseado no ordenamento por dimensão. Cada pacote possui um valor XY que indica o número de roteadores que deve percorrer até o

destino em cada dimensão, com o sentido indicado por um bit. Os pacotes percorrem a rede primeiramente na direção X para depois percorrerem a direção Y, evitando assim que ocorram *deadlocks*.

A SoCIN é uma rede baseada em chaveamento de pacotes do tipo *wormhole*. Caso haja congestionamento na rede, ou o nodo de destino não leia os dados da rede imediatamente, os *flits* do pacote são armazenados nos *buffers* dos roteadores que estão no caminho entre a origem e o destino.

4.1.3 Memória compartilhada

O modelo de memória utilizado neste trabalho é o de memória compartilhada, que consiste em uma memória de dados em um dos nodos da rede, enquanto que os processadores situam-se nos outros nodos da rede, conforme ilustrado na Figura 4.1. Para reduzir o impacto da latência de acesso à memória na performance, cada um dos processadores possui uma cache de dados. As caches utilizadas são memórias completamente associativas e seu algoritmo de substituição de blocos é definido em tempo de projeto, podendo-se escolher o algoritmo LRU (*Least Recently Used*), FIFO (*First In First Out*), FIFO *second chance*, LFU (*Least Frequently Used*) ou aleatório. O protocolo de coerência de cache utilizado é descrito na seção seguinte.

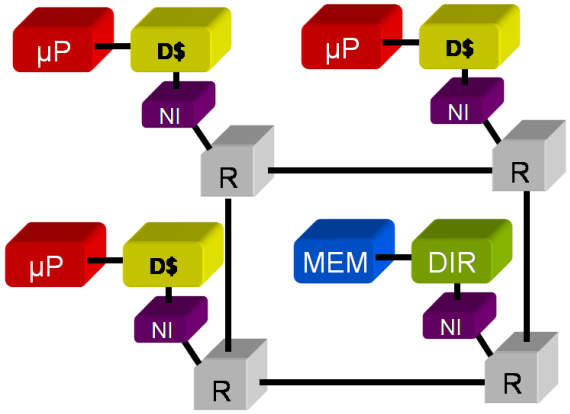

Figura 4.1: Modelo de Memória Compartilhada. Retirado de Girão (2008)

Quando ocorre um *miss* em um acesso à cache, o controladores das memórias caches enviam requisições através da rede para o diretório situado no nodo de memória solicitando blocos para leitura ou escrita de dados. Devido a características do FemtoJava, que deve receber a resposta de um acesso à memória em um ciclo, o processador é mantido em *clock-gating* até o recebimento do dado solicitado pela cache.

Como esta versão do FemtoJava possui pilha mapeada em memória, há uma memória local em cada processador reservada para este fim. Um módulo verifica o endereço de leitura ou escrita do processador e identifica se o acesso deve ser feito na pilha ou na memória cache.

A memória de instruções é local, sendo que cada processador possui todo o código da aplicação. A memória de instruções é de 8KB.

4.1.4 Coerência de cache

Para garantir a coerência dos dados das caches no modelo de memória compartilhada, utiliza-se no SIMPLE uma solução baseada em diretório. O diretório,

situado no nodo de memória, armazena informações de quais caches possuem cópia dos blocos da memória e se a cópia da memória está desatualizada. Para isto, faz uso de uma tabela *full-map* chamada STA, conforme mostra a Figura 4.2.

BLOCO	P0	P1	P2	P3	Sujo
0	1	1	1	0	0
1	0	1	0	0	1
2	0	0	1	0	1
3	0	0	1	0	0

Figura 4.2: Tabela STA do diretório.

Esta tabela utiliza, para cada bloco da memória, um bit por processador para indicar o status do bloco (se o processador possui cópia válida), e mais um bit por bloco chamado de *dirty*, significando que a cópia da memória está desatualizada. Neste caso, apenas uma das caches pode possuir cópia do bloco. Nas requisições de blocos, o diretório faz uso desta tabela para encaminhar invalidações ou pedidos de *write-back* forçado para as caches que possuem cópia do bloco.

Sempre que ocorrer um *read-miss* ou *write-miss* em uma cache e esta estiver totalmente ocupada, o controlador da cache deve desfazer-se de um dos blocos para possibilitar o recebimento do novo bloco. A escolha do bloco que será retirado da cache é feita de acordo com a política de substituição escolhida. Se o bloco estiver modificado, (cache possui permissão de escrita) ele deve ser enviado em *write-back* para o diretório. Se não, apenas uma mensagem de *page replacement* é enviada para informar ao diretório que a cache não possui mais o bloco substituído. Só então o controlador da cache fará a solicitação do bloco necessário para a leitura ou escrita do processador.

Quando ocorre um *read-miss* em uma das caches, o controlador da cache envia uma requisição de bloco para leitura (também chamado de GETS) para o diretório. O comportamento do diretório para este caso é ilustrado no diagrama da Figura 4.3.

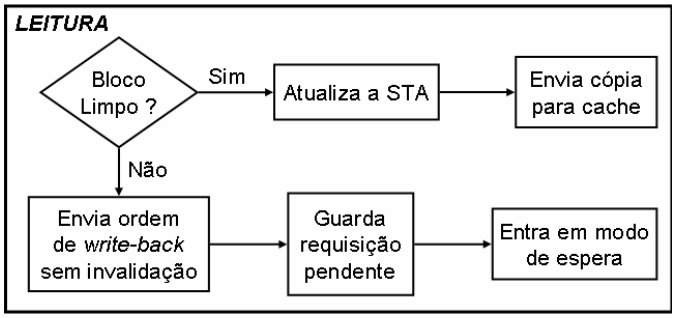

Figura 4.3: Tratamento do diretório para solicitação de leitura. Retirado de Girão (2008)

Caso o bloco esteja limpo (a cópia da memória está atualizada), o diretório atualiza a tabela de status (STA) para indicar que o processador que solicitou agora possui cópia do bloco e então envia a cópia do bloco para o processador. Se o bloco estiver sujo (*dirty*), envia ordem de *write-back* sem invalidação, o que significa que o processador que possui o bloco deve enviar sua cópia para o diretório perdendo a permissão de escrita, mas pode manter sua cópia na cache. O diretório aguarda o recebimento do *write-back* do bloco em um modo de espera, sem atender novas solicitações, até que tenha finalizado a solicitação pendente. Ao receber o bloco em *write-back*, o diretório re-encaminha para o processador que solicitou o bloco.

O diretório possui dois *buffers* para armazenar pacotes recebidos: um para *write-backs* e mensagens de *page replacement* e outro para solicitações (GETS, GETX e GETP). Isso possibilita que no modo de espera o diretório não atenda nenhuma nova solicitação, porém *write-backs* e *page replacements* continuam sendo tratados.

Quando ocorre um *write-miss* em uma das caches, o diretório recebe uma solicitação de escrita (GETX). Conforme pode ser visto na Figura 4.4, se o bloco estiver limpo, o diretório invalida todas as cópias já que agora o processador que solicitou deve ter uma cópia exclusiva para poder escrever no bloco. A tabela STA é então atualizada para indicar qual processador possui o bloco e o bit *dirty* é setado. Em seguida, a cópia é enviada para o processador que solicitou o bloco.

Caso o bloco esteja sujo, o diretório encaminha uma ordem de *write-back* com invalidação, o que significa que a cache que possui o bloco com permissão de escrita deve invalidá-lo e enviá-lo para o diretório. Novamente, o diretório aguarda o bloco em modo de espera.

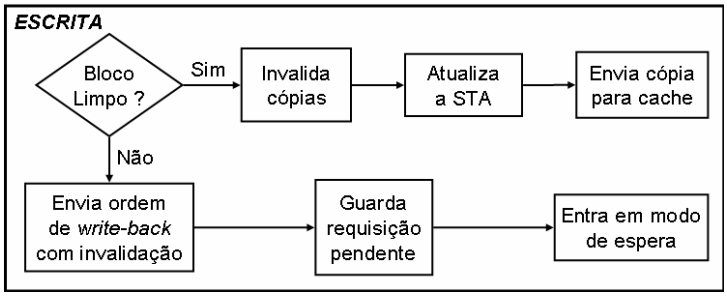

Figura 4.4: Tratamento do diretório para solicitação de escrita. Retirado de Girão (2008)

Outro tipo de solicitação possível é o de permissão de escrita (GETP), descrito na Figura 4.5. Ele é enviado para o diretório quando o processador deseja fazer uma escrita e a cache possui cópia do bloco para leitura, mas não de forma exclusiva. Se a cache ainda possuir cópia do bloco, o diretório deve enviar invalidações para todas as demais cópias e atualizar a tabela antes de enviar a permissão de escrita para a cache. Entretanto, entre a solicitação da permissão de escrita pela cache e o tratamento da solicitação pelo diretório, pode ocorrer uma situação em que a cópia do bloco em questão seja invalidada pelo diretório por uma solicitação de outro processador. Neste caso, o diretório deve ignorar o pedido de permissão de escrita, já que o controlador da cache verifica quando é invalidado um bloco que aguarda perdido de permissão e refaz a solicitação, desta vez enviando um pedido de escrita no bloco (GETX).

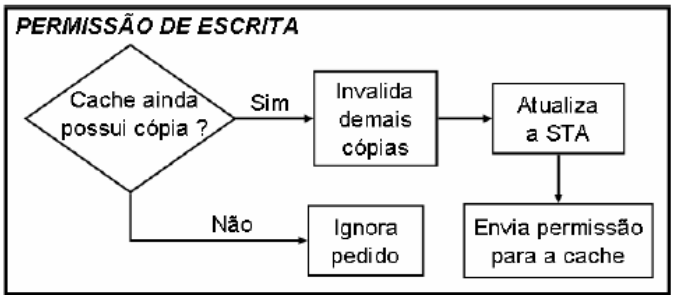

Figura 4.5: Tratamento de solicitação de permissão de escrita. Retirado de Girão (2008)

O tratamento de um *write-back* pelo diretório é mostrado no diagrama da Figura 4.6. Se o diretório está no modo de espera (existe pendência) e o bloco recebido é o bloco

pendente, o diretório atualiza a memória e, após atender a requisição pendente e atualizar a tabela, sai do modo de espera. Caso não exista pendência, ou o bloco recebido em *write-back* não é o pendente, o diretório apenas atualiza a memória e a tabela e permanece no estado de espera.

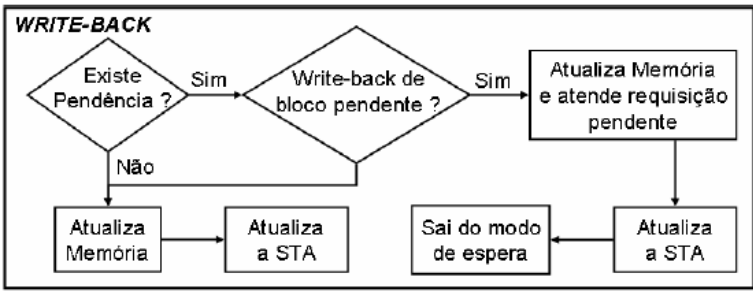

Figura 4.6: Tratamento de *write-back*. Retirado de Girão (2008)

4.1.5 *Locks*

Para garantir a consistência dos dados no modelo de memória compartilhada, a plataforma dá suporte ao mecanismo de *locks* através das operações *down* e *up* sobre variáveis *mutex*. A operação *down* testa se a variável *mutex* está liberada para que o processador entre na seção crítica, enquanto que a operação *up* libera o *mutex*. Como estas operações devem ser atômicas, são mapeadas para as operações *test-and-set* e *test-and-reset*, respectivamente.

A operação *down* faz com que o processador sinalize para o controlador da cache efetuar um *test-and-set* na posição de memória da variável *mutex*. Uma mensagem de *test-and-set* é enviada para o diretório, e este, por sua vez, realiza um teste no endereço da variável *mutex* indicada. Caso seja zero, o valor é setado para um, senão, mantém o valor. Uma mensagem de resposta é enviada para a cache informando se a operação de *test-and-set* foi bem sucedida ou não. Caso a variável não esteja liberada, o controlador solicitará novamente o *test-and-set*.

A operação *up* utiliza o *test-and-reset*, semelhante ao *test-and-set*, porém não é efetuado nenhum teste. O diretório simplesmente seta o valor da variável para zero e envia uma mensagem de confirmação para o controlador da cache ao terminar.

4.2 Detalhes de implementação do LogTM

Esta seção mostra detalhes específicos da implementação do LogTM deste trabalho. O trabalho de Moore (2007), a descrição mais detalhada do LogTM encontrada na literatura, faz uma análise de diversos *trade-offs* na implementação. Entretanto, possivelmente para evitar restringir sua descrição a um protocolo e organização específicos, omite muitos detalhes de implementação, principalmente quanto à detecção de conflitos. Muitos dos detalhes da implementação e sua descrição detalhada também são contribuições deste trabalho.

4.2.1 Detecção de Conflitos

O mecanismo de detecção de conflitos do LogTM exige modificações no protocolo de coerência de cache. O protocolo descrito nesta seção é a adaptação ao LogTM do protocolo MSI descrito na seção 4.1.4.

Nas caches do sistema existem dois bits por bloco que permitem rastrear quais blocos foram acessados pela transação. O bit R indica que o bloco foi lido na transação e o bit W indica que o bloco foi escrito na transação. Todos os bits R e W da cache são limpos quando a transação finaliza com *commit* ou é abortada. Esses bits representam uma espécie de "posse" pela transação dos blocos escritos ou lidos, já que o bit R setado impede a invalidação do bloco e o bit W impede a invalidação e o compartilhamento, garantindo que não haja violação do isolamento dos dados até o fim da transação (*commit*) ou até que a transação libere a posse através de um *abort*. No texto, serão utilizados os termos "posse para leitura" e "posse para escrita" de um bloco por uma transação para se referir aos bits R e W, respectivamente. O termo "permissão de escrita" significa que determinada cache possui cópia exclusiva do bloco e pode escrever nele, o que não implica em "posse" se o acesso ao bloco for feito fora de transação.

Para permitir a detecção de conflitos em blocos acessados pela transação que foram substituídos nas caches do sistema, um bit de *sticky* por bloco é acrescentado na tabela STA do diretório (Figura 4.7). Este bit de *sticky* é setado quando um bloco escrito pela transação é substituído e enviado em *write-back* para o diretório. Isto permite que requisições feitas por outros processadores possam ser encaminhadas ao processador "dono" do bloco (o qual possui posse de escrita) para a detecção de conflitos.

Em cada controlador da cache é acrescentado um bit de *overflow*, setado sempre que há substituição de um bloco acessado em transação e limpo no *commit* ou *abort*. Requisições encaminhadas pelo diretório em blocos *sticky* verificam o bit de *overflow*, e caso ainda esteja setado, um conflito é detectado.

BLOCO	P0	P1	P2	P3	Sujo	Sticky
0	1	1	1	0	0	0
1	0	1	0	0	1	0
2	0	0	1	0	0	1
3	0	0	1	0	0	0

Figura 4.7: Tabela STA do diretório LogTM.

A Figura 4.8 mostra o fluxograma que descreve o funcionamento do controlador da cache, já adaptado para a detecção de conflitos do LogTM, para o caso em que um pedido de leitura é feito pelo processador.

Caso seja necessário substituir algum bloco com permissão de escrita da cache, deve-se verificar se ele foi acessado pela transação. Em caso positivo, o bit de *overflow* da cache será setado e o bloco enviado em *write-back sticky*, sinalizando que ficará no estado *sticky* no diretório. No caso em que não houve acesso no bloco pela transação, este pode ser enviado em *write-back* normal. Se a cache não possuir permissão de escrita do bloco substituído, há duas possibilidades: na primeira, se o bloco não foi lido pela transação, é enviado para o diretório uma mensagem de *page replacement* para informar que o bloco foi invalidado na cache; caso tenha sido lido pela transação, o bloco é simplesmente invalidado sem envio de *page replacement* e o bit de *overflow* é setado. Dessa forma, o diretório continuará a encaminhar invalidações à cache, permitindo que esta detecte conflito caso a invalidação seja referente ao bloco que não está mais na cache.

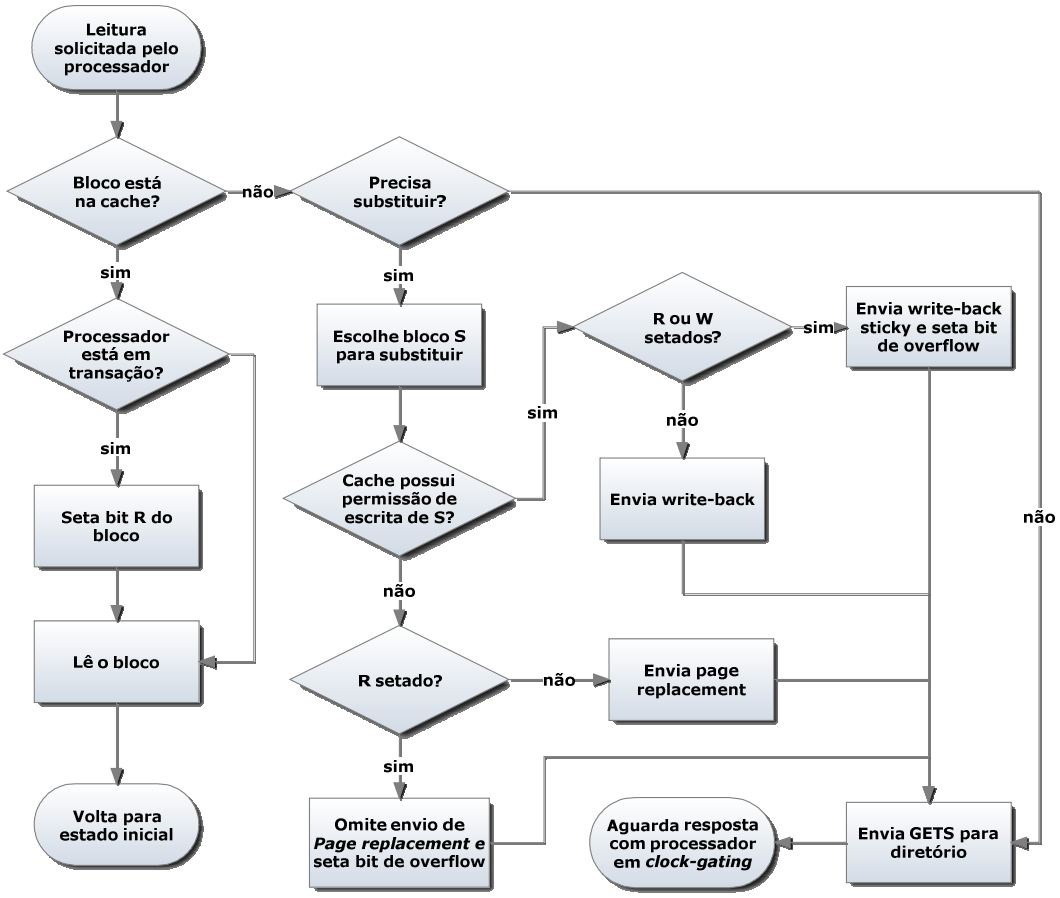

Figura 4.8: Leitura solicitada pelo processador

As solicitações de escritas pelo processador são tratadas de forma semelhante, como pode ser visto na Figura 4.9.

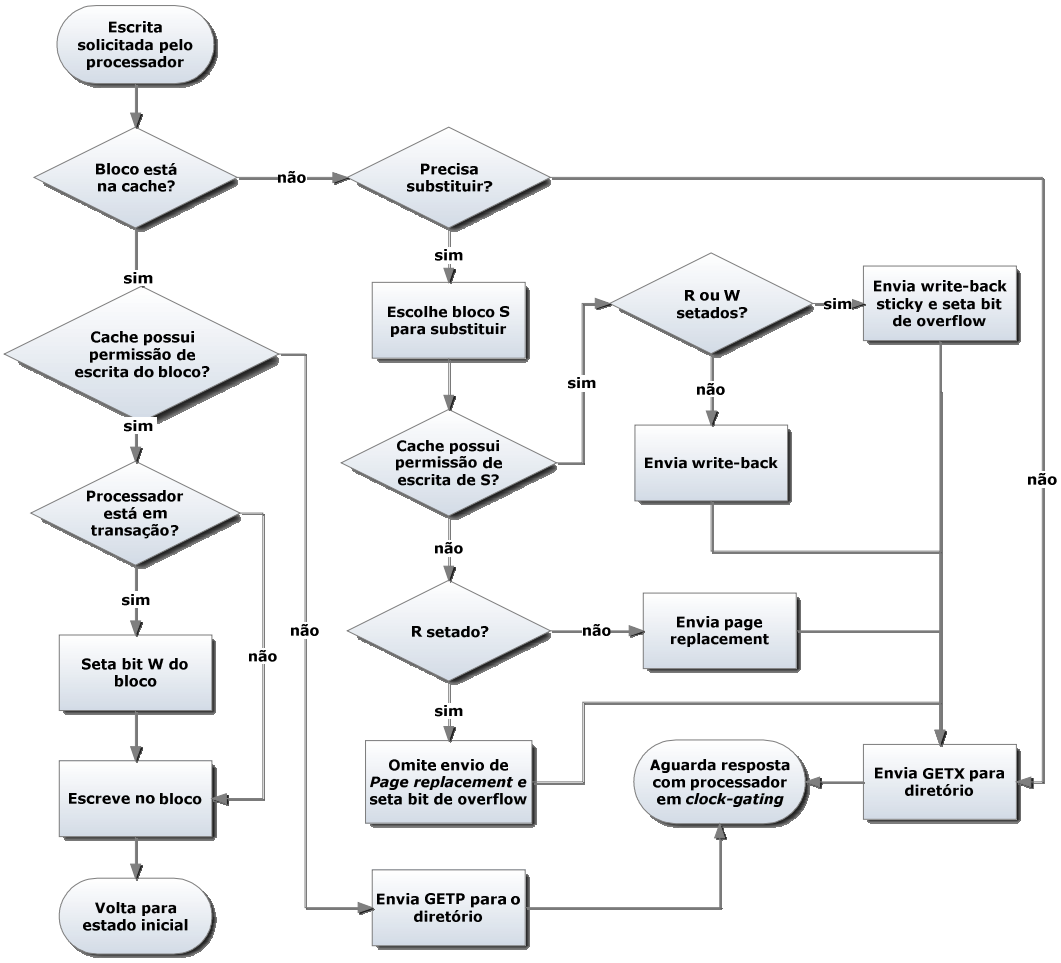

Figura 4.9: Escrita solicitada pelo processador

Conforme visto anteriormente na seção 4.1.4, existem três tipos de requisições que o controlador da cache pode fazer ao diretório: GETS, GETX e GETP. As Figuras 4.10, 4.11 e 4.12 mostram o tratamento destas requisições pelo diretório. O diretório possui dois modos de espera em que pode entrar durante o tratamento das requisições: WAIT INVALIDATE RESPONSES e WAIT TRANSACTION RESPONSE. Durante um destes modos, o diretório não atende novas requisições, apenas aguarda respostas das caches e recebe mensagens de *page replacement* e *write-backs*.

Existem três mensagens que o diretório pode enviar para as caches: INVALIDATE, que solicita invalidação de bloco, e FWD_GETS e FWD_GETX, que são pedidos de leitura e escrita, respectivamente, re-encaminhados pelo diretório para o processador que possui o bloco. Ao contrário do protocolo de coerência original, INVALIDATEs necessitam confirmação (ACK ou NACK), pois, em caso de conflito, a cache não irá invalidar seu bloco.

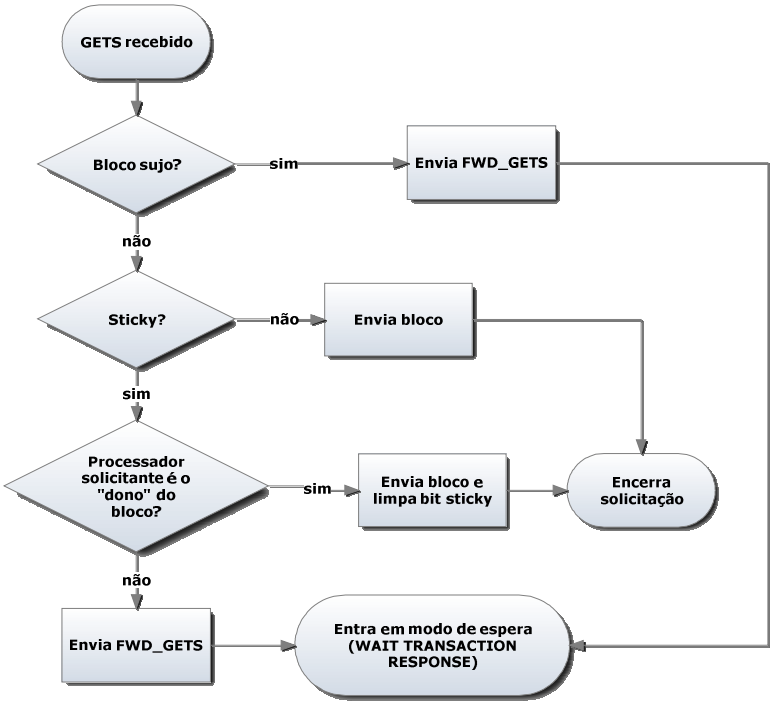

Figura 4.10: Tratamento de GETS pelo diretório

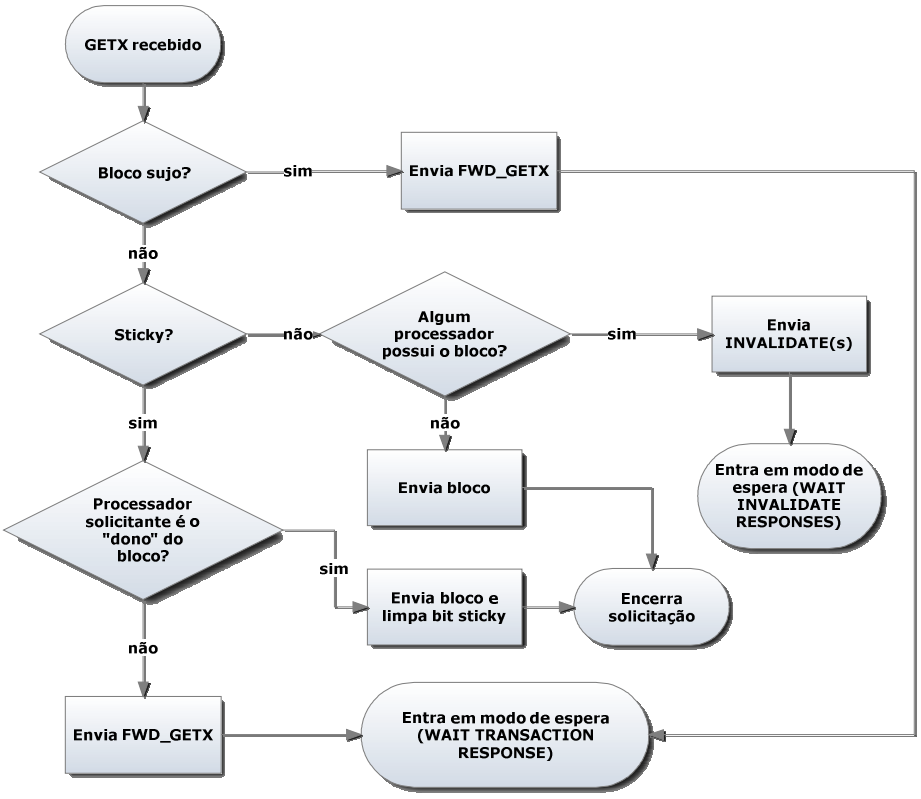

Figura 4.11: Tratamento de GETX pelo diretório

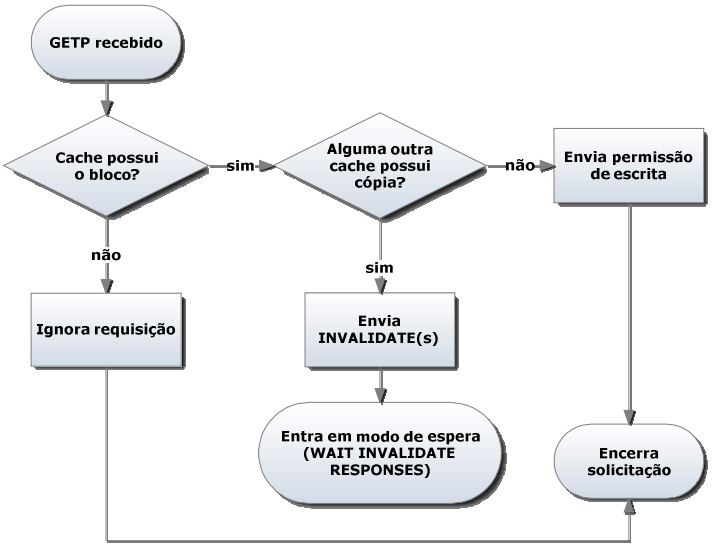

Figura 4.12: Tratamento de GETP pelo diretório

Os tratamentos das mensagens de FWD_GETS, FWD_GETX e INVALIDATE pelo controlador da cache são descritos nas Figuras 4.13, 4.14 e 4.15, respectivamente. Quando algum conflito é detectado, uma mensagem de NACK é enviada diretamente ao processador que fez a solicitação, enquanto que mensagens de confirmação (ACK, CLEAN ou WRITE-BACK) são enviadas ao diretório. O processador que fez a solicitação, ao receber NACK do processador que detectou o conflito, envia uma mensagem de NACK para o diretório.

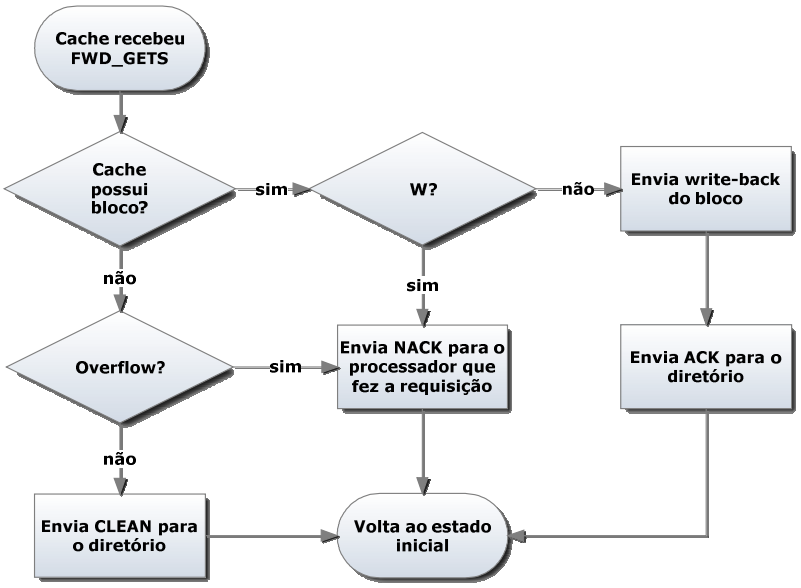

Figura 4.13: Tratamento de FWD_GETS pela cache

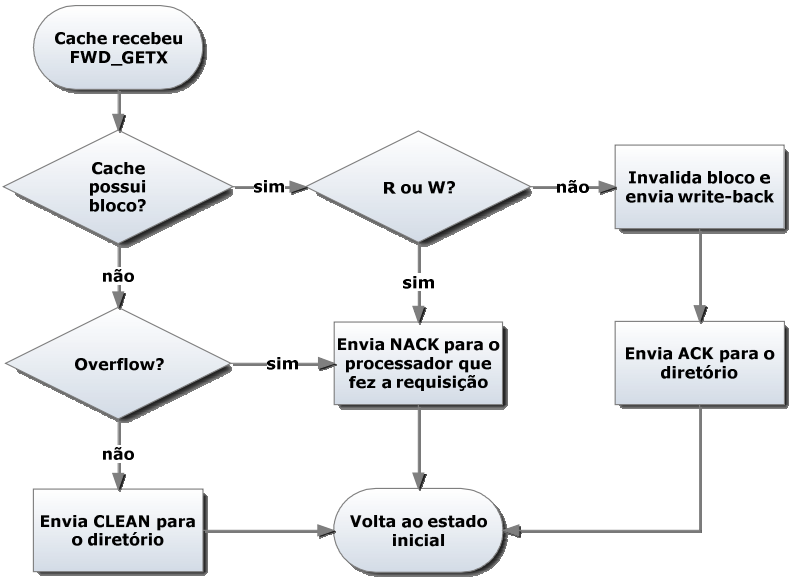

Figura 4.14: Tratamento de FWD_GETX pela cache

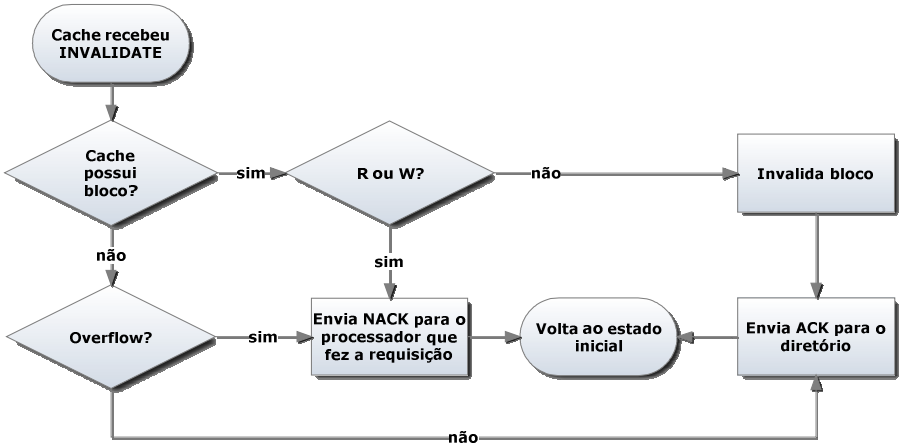

Figura 4.15: Tratamento de INVALIDATE pela cache

Os modos de espera que o diretório utiliza para tratar respostas das caches são descritos nas Figuras 4.16 e 4.17. O estado WAIT INVALIDATE RESPONSES, descrito na Figura 4.16, aguarda respostas de invalidações (ACKs ou NACKs). No caso em que o bloco é compartilhado para leitura, o estado aguarda resposta de todos os processadores que possuem cópia do bloco.

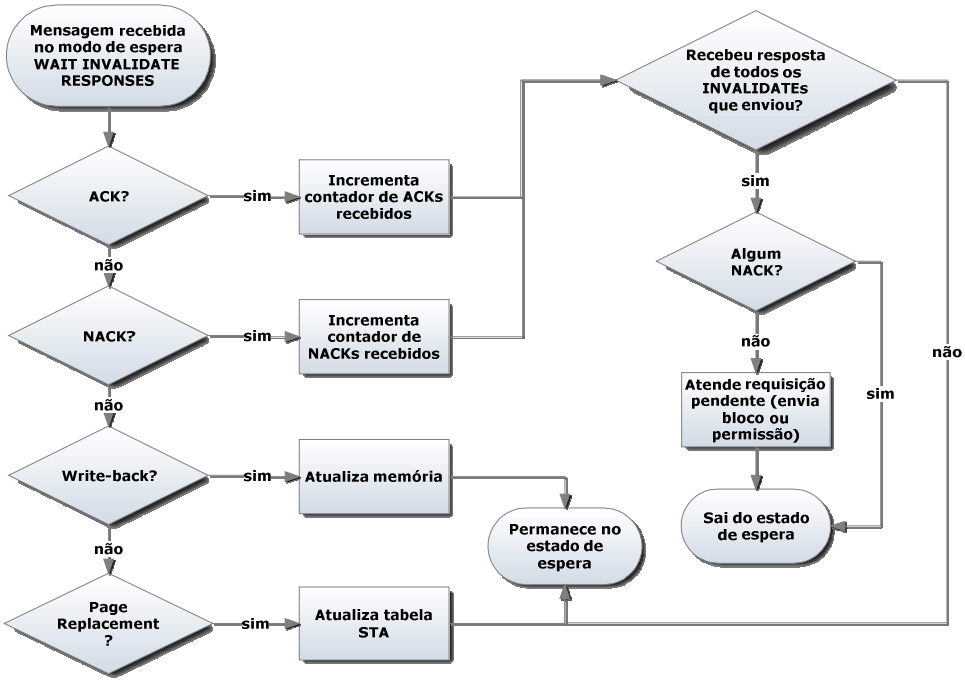

Figura 4.16: Modo de espera WAIT INVALIDATE RESPONSES

O modo WAIT TRANSACTION RESPONSES aguarda a resposta de FWD_GETS ou FWD_GETX, que podem ser CLEAN, NACK ou WRITE-BACK.

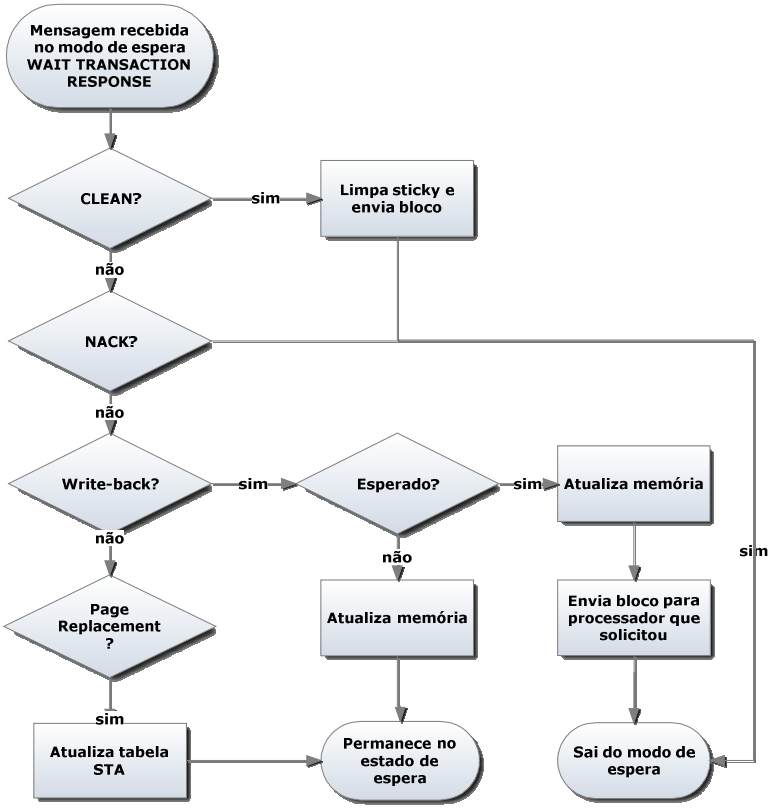

Figura 4.17: Modo de espera WAIT TRANSACTION RESPONSE

O tratamento de mensagens de NACK recebidas pelas caches faz parte da resolução de conflitos e é abordado na seção 4.2.3.

4.2.2 Gerenciamento de versão

O LogTM possui gerenciamento de versão *eager*, em que valores novos (especulativos) são salvos diretamente na hierarquia de memória enquanto que valores antigos são salvos em algum local alternativo. No LogTM, valores de registradores são salvos através de um mecanismo de *checkpoint* do processador, descrito na seção 4.2.2.1, enquanto que os valores antigos da memória são salvos em um *log*, descrito na seção 4.2.2.2.

4.2.2.1 Mecanismo de Checkpoint e Rollback do FemtoJava

Para permitir que uma transação aborte e reinicie, é necessário um mecanismo de *checkpoint* e *rollback*. O mecanismo deve salvar o estado do processador, ou seja, seu conjunto de registradores, no início da transação, de forma que a execução possa voltar para o ponto de início de transação como se não houvesse executado previamente.

Apesar de ser possível suportar o *rollback* por software, optou-se por incluir no processador suporte em hardware para evitar perda de desempenho com rotinas de salvamento, descarte do contexto no caso de *commit* e restauração de contexto em caso de *abort*.

No FemtoJava, apenas seis registradores precisam ser copiados para que o contexto seja salvo. Não é necessário salvar a pilha, já que esta nunca será menor em nenhum ponto da transação do que em seu início, desde que não haja nenhum retorno de método dentro da transação sem que a respectiva chamada do método também esteja na transação. Com isso, salvar o ponteiro da pilha (*stack pointer*) no início da transação e restaurá-lo com este valor ao abortar é suficiente para descartar todas as alterações na pilha.

A Figura 4.19 mostra as modificações do hardware para duplicar o PC original (Figura 4.18). Um novo registrador (PC_RB) é adicionado com o objetivo de salvar o PC de início de transação. O sinal *save_state*, que ativa o registrador, é acionado no início de uma transação através de uma instrução de I/O. Em caso de *abort*, o controlador da cache sinaliza através do sinal *rollback* fazendo com que o PC carregue o valor de PC_RB.

Da mesma forma, os registradores A, B, FRM, VARS e SP são duplicados para permitir que o estado seja salvo. Ao abortar, é necessário ainda que a máquina de estados de controle do processador retorne ao estado inicial já que o sinal de *rollback* ocorre durante a execução de uma instrução de acesso à memória.

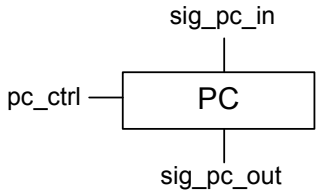

Figura 4.18: Registrador PC original

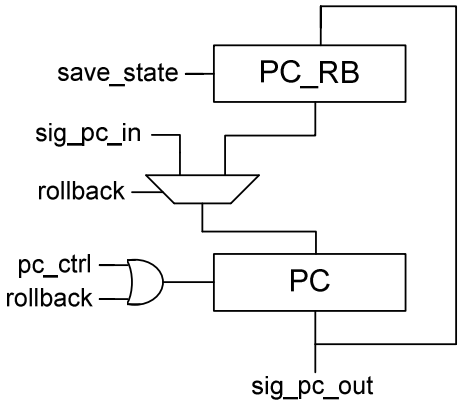

Figura 4.19: Modificações do PC para suporte do *rollback*

Para estimar o custo em área e impacto na freqüência desta solução em hardware fez-se a síntese da descrição VHDL do processador FemtoJava com o mecanismo de *checkpoint* e *rollback* para o FPGA Xilinx Virtex-II Pro xc2vp30-7ff1152. A Tabela 4.1 apresenta os valores de Slices, Flip Flops, LUTs e freqüência máxima do processador obtidos através da ferramenta Xilinx ISE 9.1. O número de LUTs utilizadas aumentou cerca de 14,51%, enquanto que a freqüência máxima do processador diminuiu 1,35%.

	Processador original	**Processador com *checkpoint* e *rollback***	**Aumento (%)**
Slices	1234	1446	17,17%
Flip Flops	678	810	19,47%
LUTs	2266	2595	14,51%
Freqüência máxima	85,503 MHz	84,348 MHz	-1,35%

Tabela 4.1: Custo do mecanismo de *checkpoint* e *rollback* em hardware para o FemtoJava

4.2.2.2 Log da transação

O *log* da transação é uma região da memória utilizada por cada *thread* para salvar os valores antigos de todos os endereços de memória modificados durante a transação. Dois registradores são utilizados para a manipulação do *log*: *log base* e *log pointer*. *Log base* define a localização na memória e é definido na inicialização do sistema. O *log pointer* marca a posição do último valor escrito do *log* e é incrementado a cada escrita.

Na implementação, é importante considerar a granularidade dos dados do *log*. A cada escrita, são salvos no *log* o endereço e os dados anteriores, podendo estes serem desde uma palavra até um bloco de memória inteiro.

Como o *log* é utilizado apenas para restaurar os valores de memória anteriores à transação, não é necessário salvar dados mais de uma vez referentes ao mesmo endereço, o que permite reduzir o número de escritas no *log*. Entretanto, esta otimização exige que sejam consultados quais dados já foram escritos no *log*. A granularidade de bloco permite que os bits W de cada bloco sejam utilizados para isso, enquanto que na granularidade de palavra o custo desta verificação é maior. Esta otimização depende da

localidade espacial dos acessos para ser vantajosa em termos de ocupação de memória do *log*.

Neste trabalho, optou-se pela granularidade de palavra que, mesmo sem otimizações para entradas duplicadas, é vantajosa para transações que escrevem poucas vezes no mesmo bloco.

No *commit* da transação, os valores do *log* são descartados. Para isso, basta zerar o *log pointer*. Ao abortar, a transação deve ler o *log* em ordem inversa da qual foi escrito e restaurar os dados da memória com os valores contidos no *log*. O LogTM especifica que esta operação seja feita por um tratador em software para que a estrutura do *log* seja salva em um espaço da memória virtual, permitindo que o *log* cresça além da capacidade da memória e seja paginado para o disco pelo sistema operacional. Entretanto, neste trabalho o mecanismo de *abort* para restauração da memória é feito pelo controlador da cache. Esta abordagem restringe o tamanho do *log* ao espaço reservado na memória física do sistema, mas dado o contexto de sistemas embarcados, esta restrição pode ser pouco significativa. Além disso, ao fazer-se esta operação inteiramente em hardware, reduz-se o tempo de execução e energia em comparação ao que seria necessário para execução do tratador em software.

É possível utilizar um *buffer* para armazenar os dados do *log* de forma que sejam copiados para a memória em um momento em que o processador não está fazendo acessos à memória, evitando que a escrita no *log* prejudique a performance da aplicação. Neste trabalho, optou-se por um *buffer* do tamanho de um bloco de memória (32 bytes). Ao ser preenchido pelo *log*, seus dados são enviados para o diretório e salvos no espaço reservado na memória para o *log* da transação. Ao abortar, o controlador da cache inicia a leitura do *log* pelos dados no *buffer* para em seguida ler os dados do *log* na memória. Dados de blocos que estão em *sticky* no diretório recebem uma mensagem com dado e endereço para restaurar a memória e deixam de ser *sticky*.

4.2.3 Resolução de conflitos

Após um conflito ser detectado, o sistema deve resolver o conflito para preservar o isolamento entre as transações. O sistema pode serializar a execução através de *stall* (parada temporária) ou abortando uma das transações. Abortar as transações em conflito pode levar o sistema a um *livelock* se todas as transações forem abortadas antes de finalizarem. Parar as transações com *stall* pode causar *deadlock* caso uma tenha que esperar pela outra indefinidamente.

O LogTM utiliza a resolução de conflitos *requester stalls* (BOBBA, 2007), em que a transação que fez a requisição que originou o conflito é responsável pela sua resolução. A transação tenta resolver o conflito através de *stall*, entretanto, se o sistema detectar alguma possível dependência cíclica que possa causar *deadlock*, a transação aborta.

Para implementar a resolução de conflitos, o sistema utiliza o método de *timestamps* distribuídos do *Transactional Lock Removal* (RAJWAR, 2002). Neste método, um *timestamp* é atribuído a cada transação no início de sua execução. Transações com *timestamps* anteriores possuem prioridade sobre transações com *timestamps* mais recentes.

Para garantir progresso do sistema e reduzir *aborts*, uma transação entra em *stall* em um conflito, abortando apenas se (1) há risco de *deadlock* e (2) é logicamente posterior à transação com a qual entrou em conflito. Risco de *deadlock* é detectado quando uma

transação está esperando uma transação mais recente e fazendo uma transação mais recente esperar. Um bit chamado *possible cycle* por processador é utilizado para isto. Este bit é setado quando uma transação envia NACK para outra anterior (de maior prioridade). Uma transação aborta quando seu bit *possible cycle* está setado e recebe NACK de uma transação anterior. Para evitar que duas transações tenham o mesmo *timestamp*, o endereço físico de rede dos processadores é concatenado ao *timestamp*.

Ao entrar em *stall* após receber NACK notificando conflito, a transação envia novamente a requisição ao diretório. O conflito se repetirá até que a transação conflitante (a qual possui "posse" do bloco, representada pelos bits R e W) liberá-lo ao fim da transação, através de um *commit* ou *abort*.

Ao abortar, a transação libera a posse de todos os seus blocos usados na transação (limpando bits de R e W) e aguarda um tempo arbitrário antes de iniciar sua execução novamente.

Neste tipo de resolução de conflitos, uma transação, ao fazer uma requisição que ocasione um conflito, não aborta diretamente a outra, mesmo que esta tenha menor prioridade. Uma transação aborta a si mesma ao fazer uma requisição que entre em conflito com outra transação de maior prioridade e houver risco de *deadlock*. Entretanto, a expressão de que uma transação abortou outra será utilizada no texto para se referir à transação de maior prioridade com a qual a transação que abortou entrou em conflito, ou, em outras palavras, a transação que enviou a mensagem de NACK que fez com que a transação abortasse.

Caso conflitos se repitam entre transações, o LogTM sugere o uso de um gerenciador de contenção (*contention manager*) em software (HERLIHY, 2003) para decidir de forma mais precisa quando uma transação deve abortar. Diversos gerenciadores de contenção em software já foram propostos, entretanto, escolher um é uma tarefa complexa já que diferentes gerenciadores de contenção são melhores para diferentes *benchmarks* e nenhum deles é melhor para todos os tipos de carga de aplicação (SCHERER, 2004).

4.2.4 Exemplos

Esta seção apresenta alguns casos para exemplificar o funcionamento da detecção e resolução de conflitos desta implementação.

A Figura 4.20 apresenta um caso em que duas transações possuem o bloco A compartilhado para leitura e tentam adquirir permissão para escreverem no bloco. A transação T0 envia GETP para o diretório (DIR), que encaminha a mensagem INVALIDATE para invalidar a cópia do bloco da transação T1. Assumindo que T0 é anterior a T1 (T0 tem maior prioridade), T1 seta sua *flag* de *possible cycle* ao enviar NACK para T0. Ao receber o NACK, T0 encaminha-o para o diretório e entra em *stall*. A segunda requisição atendida pelo diretório é de T1, que também solicita permissão para escrever no bloco (GETP). O conflito com T0 é detectado, T1 aborta já que sua *flag* de *possible cycle* está setada e recebeu NACK de uma transação de maior prioridade. A terceira solicitação ao diretório é novamente o pedido de T0, que desta vez é atendido já que T1 invalida sua cópia do bloco A após ter abortado e liberado a posse dos blocos acessados em transação. Com isso, T0 recebe permissão de escrita do diretório (WRITE PERMISSION) e prossegue sua execução.

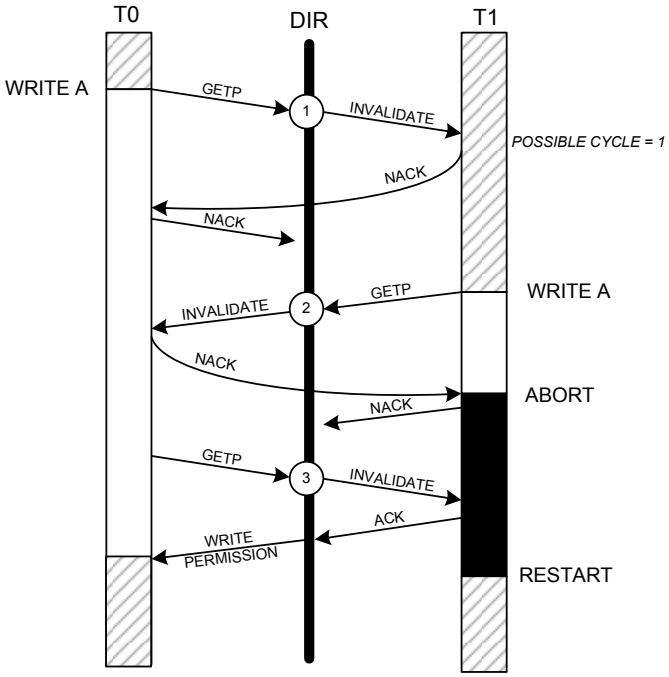

Figura 4.20: Conflito resolvido através de *abort*

A Figura 4.21 mostra um conflito detectado em um bloco que foi substituído pela cache e sua resolução através de *stall*. Inicialmente, T0 solicita e recebe o bloco A para escrita. Ao ter um *cache miss* na leitura do bloco B, a cache escolhe o bloco A para substituir e envia-o para o diretório através de um *write-back sticky* e seta seu bit de *overflow*. Em seguida, T0 envia GETS do bloco B. O diretório recebe então pedido de leitura do bloco A. Como o bloco A está em *sticky* no diretório, o pedido é encaminhado para T0 através de um FWD_GETS. Como o bit de *overflow* está setado, o conflito é detectado e T1 entra em *stall*. Ao refazer o pedido em (4), T0 já finalizou com *commit* e envia CLEAN para o diretório. O bloco então é encaminhado para T1, que prossegue sua execução.

Na Figura 4.22, três transações conseguem o bloco A para leitura compartilhada e, em seguida, T1 tenta escrever no bloco (4). O diretório envia invalidações para T0 e T2. Como T1 não está ciente do fato de que há mais de uma transação compartilhando o bloco, ao receber o primeiro NACK, vindo de T2, entra em *stall* e envia novamente GETX para o diretório. Entretanto, o NACK de T0 chega a T1 após este ter feito a nova requisição. Para evitar que um NACK atrasado seja interpretado como um NACK da nova requisição, utiliza-se um bit de seqüência que é alternado a cada requisição. Este bit é encaminhado junto com invalidações e NACKs. O processador, ao receber NACK, compara o bit de seqüência recebido ao da última requisição enviada, e, se forem diferentes, o NACK é ignorado. Antes de ser atendido pelo diretório, o GETX (5) aguarda no *buffer* do diretório até que este tenha recebido resposta a todas as invalidações. Apesar de T2 ter finalizado sua transação e invalidado seu bloco, T1 ainda entra em conflito com T0.

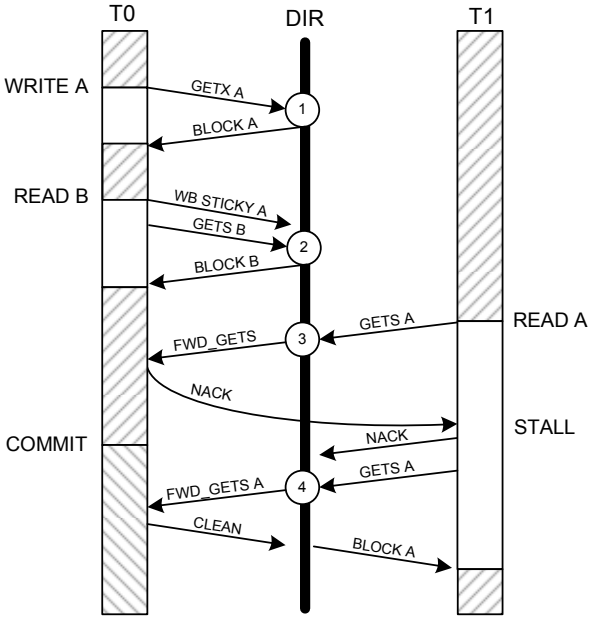

Figura 4.21: Detecção de conflito em bloco substituído na cache

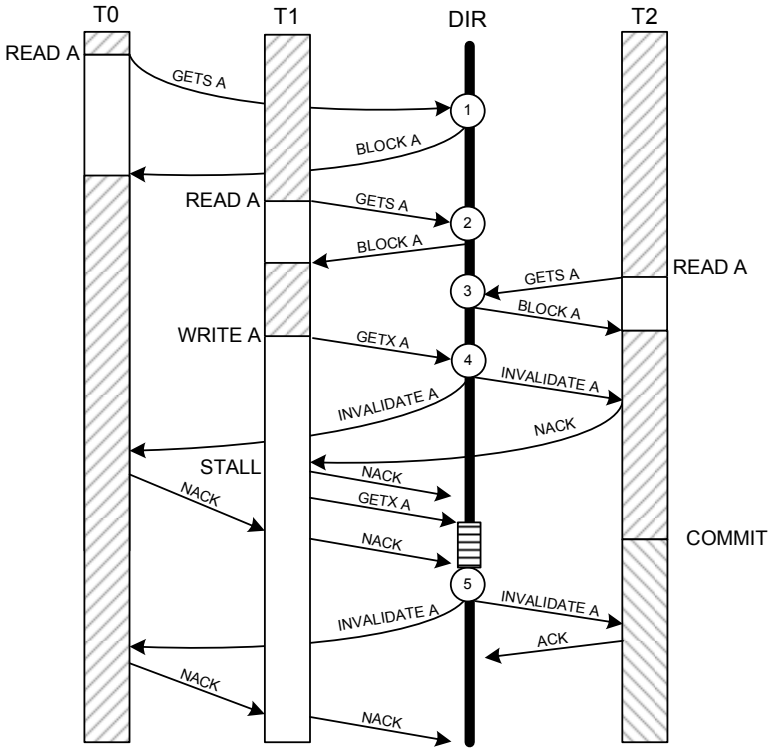

Figura 4.22: Tentativa de escrita em bloco compartilhado por três transações

4.3 *Backoff Delay on abort*

Uma questão crítica que surge no LogTM é quanto tempo uma transação deve aguardar antes de reiniciar sua execução após abortar. Neste trabalho, chamou-se esse tempo de *backoff delay on abort.* Este tempo, se subestimado, pode causar vários *aborts* desnecessários e até mesmo comprometer o progresso da execução do sistema, enquanto que se for estimado além do necessário pode prejudicar a performance.

Em um *abort*, o processador libera a posse dos blocos acessados durante a transação. A transação que fez com que a outra abortasse necessita solicitar novamente o bloco para conseguir sua posse e continuar execução. Após abortar, o processador aguarda um tempo arbitrário (*backoff delay on abort*) antes de iniciar novamente. Se a transação abortada reiniciar e conseguir novamente posse do bloco conflitante antes que a transação que abortou consiga, o mesmo conflito irá acontecer, causando um novo *abort* da transação. Esta situação não apenas prejudica a performance como pode causar um *livelock* caso ela se repita indefinidamente.

A Figura 4.23 apresenta um exemplo em que duas transações (T0 e T1, sendo que T0 possui *timestamp* anterior) possuem o bloco A compartilhado para leitura em suas caches e ambas tentam escrever no bloco. Na primeira requisição (1), T0 pede permissão para escrever no bloco enviando um GETP para o diretório, que por sua vez encaminha o pedido solicitando que T1 invalide sua cópia. T1 detecta o conflito, envia NACK para T0 e seta sua *flag* de *possible cycle*, já que T1 possui um *timestamp* posterior ao de T0. Ao receber o NACK, T0 entra em *stall* e envia novamente sua solicitação em (3). A segunda requisição que o diretório recebe (2) é um GETP de T1, que faz com que T1 aborte ao receber NACK de T0. Neste exemplo, T1 solicita o bloco A para leitura e tem *cache hit* antes que a invalidação de (3) chegue a T1, fazendo com que o conflito se repita.

A Figura 4.23 mostra dois intervalos de tempo: *retrieve time*, medido do momento em que T1 libera a posse do bloco ao abortar, até que solicita novamente o bloco e consegue antes que T0; e o intervalo *ownership change time*, que é o tempo que a transação de maior prioridade tem para conseguir a posse do bloco após a transação de menor prioridade a tenha liberado. Claramente, o *retrieve time* deve ser maior do que o *ownership change time* para que a transação de maior prioridade adquira permissão de escrita do bloco.

Considerando que a transação abortada libera a posse do bloco após completar o *rollback* do processador e restaurar a memória, o *retrieve time* depende de dois componentes principais:

1) *Backoff Delay on Abort* – intervalo de tempo arbitrário no qual o processador aguarda antes de reiniciar a transação abortada.
2) O tempo que a transação leva para acessar o bloco conflitante, incluindo uma possível busca do bloco da memória caso tenha sido invalidado por outro processador ou substituído na cache.

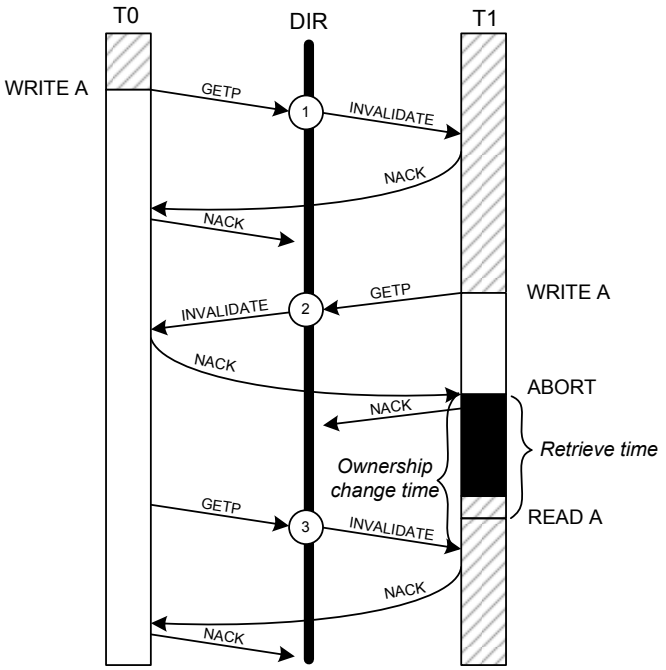

Figura 4.23: Tentativa de invalidação mal sucedida

O intervalo *ownership change time* depende de três principais componentes, considerando que a requisição do processador que abortou esteja no *buffer* de requisições do diretório no momento em que o NACK do processador que abortou chega ao diretório:

1) O tempo para enviar o NACK para o diretório, incluindo atrasos na rede.
2) O tratamento de outra(s) requisição(ões) que possa(m) ter chegado no diretório antes que a requisição da transação de maior prioridade.
3) O tempo para processar a requisição em questão, incluindo o tempo para enviar mensagens de coerência através da rede para o processador que abortou (como o INVALIDATE da terceira requisição da Figura 3.23).

A situação pode se agravar para o caso em que várias transações que compartilham o bloco para leitura tentam atualizá-lo. Para conseguir a permissão de escrita no bloco, a transação de maior prioridade precisa abortar todas as outras transações que compartilham o bloco e solicitar a permissão antes que uma delas reinicie e leia o bloco novamente. Mais precisamente, há três condições que devem ser atendidas para que a transação consiga o bloco de forma exclusiva (supondo que transações {T0, T1, .. Tn} compartilham o mesmo bloco para leitura quando tentam obter permissão exclusiva para escrita, assumindo prioridades T0 > T1 > .. > Tn):

1) T0 deve requisitar acesso exclusivo ao bloco de forma que o diretório envie invalidações para todas as transações que compartilham o bloco e estas respondam com NACK setando seus bits de *possible cycle*.
2) Todas as transações de T1 a Tn devem ter abortado.
3) T0 deve solicitar acesso exclusivo antes que alguma das transações de T1 a Tn reinicie e leia o bloco novamente.

A Figura 4.24 mostra um exemplo em que cinco transações compartilham um bloco A com posse de leitura quando todas tentam adquirir permissão de escrita (assumindo prioridades T0 > T1 > T2 > T3 > T4). Em seu estado inicial na linha 0, quando todas as *flags possible cycle* estão limpas e todos possuem posse de leitura do bloco A. A linha 1 mostra o estado dos processadores (neste caso, apenas as informações relevantes para a detecção e resolução de conflitos) após a requisição de T2 ter sido processada. T2 recebe NACK de todos os processadores e entra em *stall*. T3 e T4 setam suas flags de *possible cycle* já que enviaram NACK para transações de maior prioridade. Na linha 2, T3 aborta ao receber NACK das transações de maior prioridade (T0, T1 e T2) e por possuir seu bit de *possible cycle* setado. A primeira condição descrita acima é atendida na linha 3, quando a transação de maior prioridade (T0) entra em *stall* e faz com que todas as outras marquem suas *flags* de *possible cycle*. O bloco de T3 é invalidado após liberar sua posse ao abortar. A segunda condição é atendida apenas na linha 6, quando todas as transações já abortaram exceto pela de maior prioridade. Somente quando T0 tiver sua requisição processada novamente conseguirá permissão de escrita no bloco, atendendo a última condição. T3, a primeira transação a abortar, teve que aguardar, neste caso, cinco requisições de outras transações antes de solicitar novamente permissão de leitura do bloco A.

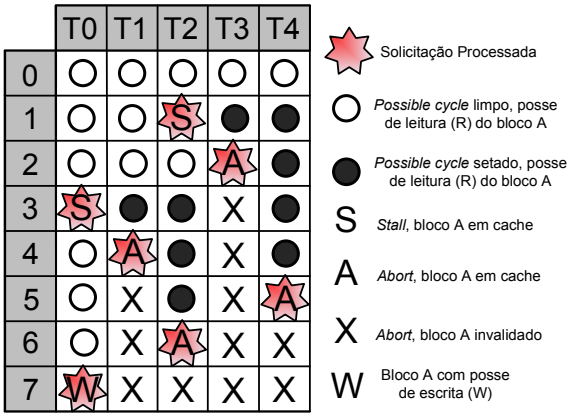

Figura 4.24: Obtenção de permissão de escrita de bloco compartilhado

Como pode-se ver, o progresso do sistema e a performance dependem de um tempo de espera (*backoff delay on abort*, que conseqüentemente afeta o *retrieve time*) contra um intervalo de tempo imprevisível (*ownership change time*) que depende de parâmetros arquiteturais (latência da rede e largura de banda, arquitetura do diretório) e características dinâmicas (contenção na rede, número de transações compartilhando determinado bloco, ordem de processamento das requisições no diretório).

Para tratar esta situação, o LogTM utiliza uma política de *backoff* linear randomizado após o *abort* e opcionalmente permite o uso de mais precisos (e complexos) gerenciadores de contenção em software se o problema persistir. Esta estratégia de *backoff* linear randomizado deixa que o sistema se "adapte" para garantir progresso, mas a performance ainda permanece uma questão de ajuste de parâmetros. Se o tempo de *backoff* for muito curto, a transação de maior prioridade terá que tentar diversas vezes enquanto que as transações de baixa prioridade abortadas reiniciarão e serão abortadas novamente, executando trabalho inútil. Com um tempo maior do que o necessário, perde-se oportunidade de explorar paralelismo, afetando a performance de todo o sistema.

4.4 Mecanismo *Abort Handshake*

Ao invés de tentar encontrar a melhor heurística para ajustar o tempo de *backoff delay on abort*, a solução proposta por este trabalho é baseada em sinalização entre transações de forma que a transação abortada seja notificada quando for seguro reiniciar para evitar que ocorra o mesmo conflito.

Duas mensagens são utilizadas para este propósito. *Signal abort* é enviado para a transação assim que aborta para notificar o processador que fez com que ela abortasse. *Release abort* é a resposta no sentido contrário para notificar quando uma transação pode reiniciar execução. Cada processador mantém uma lista de transações das quais recebeu *Signal abort* durante a transação atual e envia *Release abort* para todos os processadores em sua lista no *commit*. Esta solução foi chamada de *abort handshake.*

A Figura 4.25 mostra o caso da Figura 4.23 utilizando o *Abort Handshake*. Quando T1 aborta após o conflito com T0, envia uma mensagem de S*ignal abort* para T0 e aguarda até receber *Release abort*. T0, então, adiciona T1 em sua lista de transações que abortou. Quando finaliza a transação, T0 envia *Release abort* para T1 permitindo que ela reinicie a execução.

Para o caso em que várias transações compartilham o mesmo bloco, a transação receberá mais de um NACK para uma única solicitação. Caso aborte, a transação escolhe uma das transações de maior prioridade que enviaram o NACK para enviar o *Signal abort*. Uma transação pode esperar apenas por uma transação de maior prioridade, evitando o risco de *deadlock*. 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.

Apesar disso, esta abordagem conservadora pode economizar energia já que reduz a especulação, conseqüentemente reduz o número de mensagens que levam a contenção no diretório e na rede de interconexão, o que também pode ocasionar ganhos de performance para os casos em que a especulação não é bem sucedida.

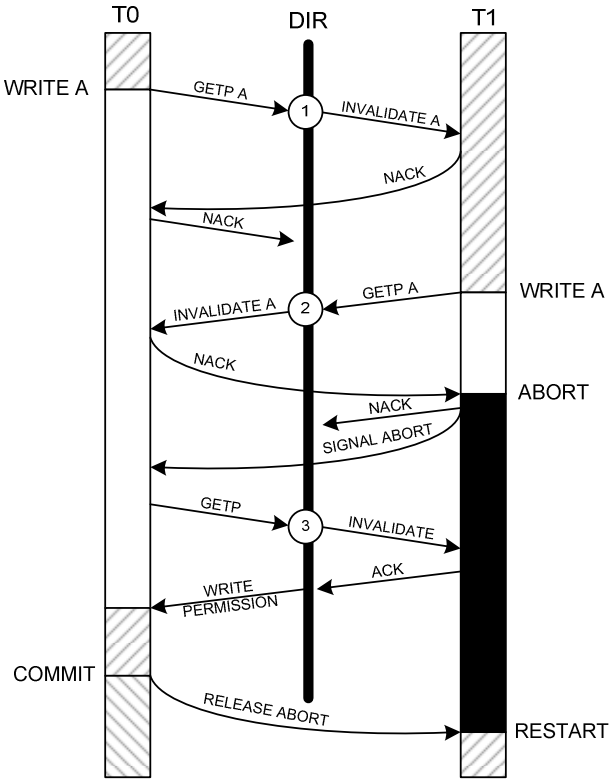

Figura 4.25: Obtenção de permissão de escrita de bloco compartilhado

5 EXPERIMENTOS

Este capítulo apresenta uma descrição dos experimentos deste trabalho, assim como os resultados obtidos com estes experimentos.

5.1 Configurações dos experimentos

Todos os experimentos deste trabalho foram realizados na plataforma virtual SIMPLE (BARCELOS, 2007), descrita na seção 4.1. Os valores de performance obtidos se referem ao número de ciclos necessários até que o último processador termine sua execução. O consumo de energia foi obtido levando-se em consideração a energia da rede (incluindo *buffers*, árbitros, *crossbar* e *links*), memórias, caches e processadores do sistema. A energia da NoC foi calculada através da biblioteca Orion (WANG, 2002), enquanto que a energia das memória e das caches foi calculada com a ferramenta Cacti (WILTON, 1996). Apenas a energia dinâmica foi considerada nos experimentos. Todos os dados de energia são referentes à tecnologia 0,18 µm.

Utilizou-se a organização de memória compartilhada, com um nodo de memória localizado na rede de forma a minimizar a distância média (medida em *hops*) dos processadores para a memória em cada uma das configurações do sistema. A Tabela 5.1 apresenta um resumo das configurações de sistema utilizadas nos experimentos.

Tabela 5.1: Configurações do sistema

Número de processadores	{2, 4, 8, 16, 32}
Freqüência dos processadores	100 Mhz
Tamanho das caches L1	512, 1024, 2048, 8192 bytes
Tamanho do bloco	32 bytes (8 palavras)
Associatividade das caches	Completamente associativa
Tamanho da memória	{32, 64} kB
Protocolo de coerência de caches	MSI baseado em diretório
Roteamento da NoC	XY
Chaveamento da NoC	*Wormhole*
Topologia da NoC	Grade (*mesh*)
Arbitragem da NoC	*Round-robin*
Freqüência de operação da NoC	100 Mhz

Utilizou-se caches completamente associativas para minimizar os casos de *overflow* dos dados utilizados pelas transações, já que a detecção de conflitos do LogTM para estes casos é pouco eficiente ao permitir falsos conflitos. Estes falsos conflitos, além de afetarem a performance, ocorrem em situações durante a execução que são difíceis de prever, logo, adicionam certa imprevisibilidade aos resultados.A freqüência da NoC tem impacto direto no tempo de *miss* da cache, enquanto que a freqüência do processador influencia o tempo de computação propriamente dito, excluindo-se tempos de *miss*. Da mesma forma, o tempo de computação pode ser reduzido se for utilizado um processador *pipeline*, em comparação aos resultados obtidos com o processador multiciclo utilizado neste trabalho. Experimentos com outros processadores e freqüências de operação são deixados como trabalhos futuros.

5.2 Aplicações de *benchmark*

Esta seção apresenta as aplicações utilizadas nos experimentos. Para as versões transacionais, as primitivas *lock* e *unlock* foram substituídas no código pelas primitivas de início e fim de transação, respectivamente.

5.2.1 Multiplicação de Matrizes

O algoritmo de Multiplicação de Matrizes é paralelizado de forma que cada processador é responsável pela multiplicação de um subconjunto de linhas da matriz A pelas colunas da matriz B. Cada matriz possui 32x32 elementos. No início da execução, um dos processadores é eleito para inicializar as matrizes.

5.2.2 Codificador JPEG

O algoritmo do Codificador JPEG possui três etapas distintas. As etapas de quantização e DCT-2D podem ser executadas em paralelo pelos processadores, enquanto que a última (codificação de entropia) deve ser executada por um processador ao final dos passos anteriores. Sendo assim, uma imagem de 64x32 pixels é dividida em blocos de 8x8 pixels e cada processador é responsável por executar as duas primeiras etapas do algoritmo em uma quantidade igual de blocos. Ao final destas etapas, o processador mestre realiza a etapa final na imagem completa.

5.2.3 Estimação de Movimento

A Estimação de Movimento é um algoritmo utilizado para codificação de vídeo. A aplicação utilizada neste trabalho implementa a parte de busca por um macrobloco em uma imagem. O algoritmo é paralelizado de forma que cada processador fique responsável por procurar o macrobloco em uma região da imagem. Nas simulações, foram utilizados um macrobloco de 8x8 pixels e uma imagem no formato QCIF (176x144 pixels).

5.2.4 LeeTM

LeeTM (ANSARI, 2008) é uma implementação do algoritmo de Lee para roteamento de circuitos com o objetivo de servir de *benchmark* para sistemas de memória transacional. O algoritmo de Lee é uma aplicação realista e não trivial que possui paralelismo intrínseco. No entanto, este paralelismo é difícil de expressar

utilizando-se o mecanismo de *locks*. Como o paralelismo depende das entradas (trilhas a serem roteadas), a TM pode aproveitar-se da especulação quando há paralelismo.

Cada trilha é uma unidade básica de trabalho, e o roteamento de cada uma delas é efetuado por uma transação. Se as trilhas forem roteadas em partes separadas do *grid* do circuito, pode-se realizar o roteamento de forma completamente paralela, mas se estiverem próximas o suficiente para uma afetar o roteamento da outra, então uma deve ser roteada após a outra para garantir que o algoritmo execute de forma correta. A versão com *locks* do algoritmo precisa adotar uma abordagem conservativa, delimitando em seções críticas o roteamento de cada uma das trilhas, e, conseqüentemente, realizando todo o trabalho de forma serial.

O algoritmo é composto por duas etapas principais: expansão e *backtracking*, ilustradas na Figura 5.1. A fase de expansão executa uma busca em largura, numerando todos os pontos do *grid* por uma onda de expansão a partir da origem. Em cada etapa da expansão, todos os pontos vizinhos à onda não assinalados são marcados com o valor da onda incrementado. Esta fase de expansão termina quando o destino for atingido ou quando todos os pontos do *grid* tiverem sido visitados. Os pontos numerados pela expansão representam a distância até a origem. Quando o destino é atingido, inicia-se a etapa de *backtracking*, que consiste em traçar uma rota de retorno do destino até a origem percorrendo uma seqüência decrescente de pontos.

O *grid* do circuito é representado por uma matriz de três dimensões, o que possibilita estender o algoritmo para circuitos de diversas camadas de roteamento. Na versão utilizada neste *benchmark* há duas camadas. É possível que uma trilha atravesse de uma camada para outra através de uma via para encontrar um melhor caminho até o destino. É possível, ainda, atribuir "pesos" para determinadas regiões do *grid*, fazendo com que o roteamento evite áreas que tendem a ficar congestionadas.

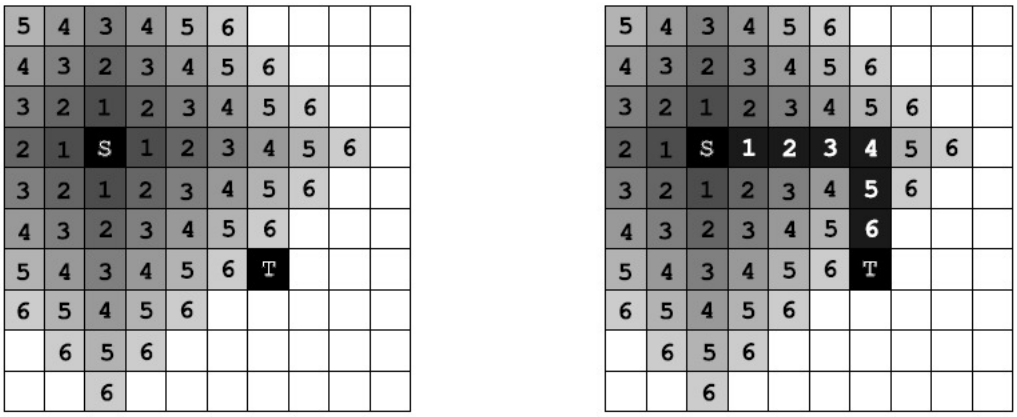

Figura 5.1: Fases de Expansão (esquerda) e *Backtracking* (direita). Retirado de Ansari (2008).

Para evitar que as etapas de expansão de duas trilhas roteadas em paralelo interfiram umas nas outras, cada processador efetua sua etapa de expansão em uma cópia local do *grid*, apenas lendo os dados do *grid* compartilhado para identificar obstáculos e trilhas roteadas anteriormente. Apenas na fase de *backtracking* há escritas no *grid* compartilhado. Se houver algum conflito na escrita da trilha com uma leitura da fase de expansão ou escrita de outra trilha no *grid* compartilhado, uma das transações deverá abortar e efetuar o roteamento novamente.

Cada processador lê de uma fila compartilhada os dados da trilha que deve ser roteada. Em seguida, inicializa seu *grid* local e então executa as fases de expansão e *backtracking*. Uma exemplificação do código do LeeTM é mostrada na Figura 5.2.

```
while (track_queue.not_empty()) { // enquanto fila das trilhas não estiver vazia
        begin_transaction();
        track = track_queue.remove_item(); // retira dados da trilha da fila
        end_transaction;
        initialize_local_grid(); // inicializa grid local
        begin_transaction();
        expansion();
        backtracking();
        end_transaction();
        }
```

Figura 5.2: Código do LeeTM.

Devido a limitações da plataforma usada neste trabalho, utilizou-se uma versão simplificada dos dados de entrada. A entrada consiste em 32 trilhas distribuídas randomicamente em um *grid* representado por uma matriz de 20x20x2 pontos de interconexões. O principal fator responsável pela limitação no tamanho do *grid* é o limite de endereçamento de 64 KB do processador FemtoJava.

Para esta aplicação, as caches foram dimensionadas de forma a evitar casos de *overflow*. Utilizou-se caches de 8KB, tamanho suficiente para manter na cache as estruturas de dados utilizadas no roteamento de cada trilha, incluindo-se uma cópia local do *grid*.

5.2.5 LeeTM-IP

Para avaliar o comportamento do sistema para o caso em que não há paralelismo disponível para as transações, desenvolveu-se neste trabalho uma modificação do LeeTM para forçar artificialmente a contenção. Para isso, adicionou-se na aplicação uma escrita em uma variável compartilhada no início de cada transação para fazer com que os processadores mantenham posse do bloco desta variável compartilhada durante toda a transação, inibindo desta forma o paralelismo enquanto são mantidos os mesmos dados de entrada para a aplicação. Com esta modificação, 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. Esta modificação da aplicação LeeTM com inibição do paralelismo é chamada de LeeTM-IP.

5.2.6 Considerações sobre as aplicações

A Multiplicação de Matrizes, O Codificador JPEG e a Estimação de Movimento são aplicações tipicamente paralelas em que a maior parte da computação pode ser feita de forma completamente paralela sem necessidade de sincronização. A sincronização é necessária, em geral, no início e final das aplicações. Estas aplicações são consideradas aplicações de baixa demanda de sincronização.

A LeeTM é considerada uma aplicação de alta demanda de sincronização, já que utiliza sincronização durante a maior parte da execução.

5.3 Resultados experimentais

Esta seção apresenta, inicialmente, os resultados dos experimentos para aplicações de baixa demanda de sincronização que comparam TM e *locks*. Em seguida, é feita uma análise do *backoff delay on abort*, apresentando-se políticas de *backoff*, uma análise de seu impacto na performance e ainda uma comparação entre a solução proposta (*abort handshake)* e as políticas de *backoff*.

5.3.1 Análise da TM e *Locks* para aplicações de baixa contenção

A primeira etapa da comparação entre TM e o mecanismo de *locks* é feita com as aplicações de Multiplicação de Matrizes, Codificador JPEG e Estimação de Movimento. Os experimentos foram realizados com 2, 4, 8, 16 e 32 processadores e três tamanhos de memória cache (512, 1024 e 2048 bytes). Utilizou-se uma memória principal de 32 KB.

A Tabela 5.2 mostra a diferença relativa no tempo total de execução das aplicações entre memória transacional e o mecanismo de *locks*. Como se pode ver, a performance da memória transacional aumenta à medida que o número de processadores aumenta. Na multiplicação de matrizes com 2P (dois processadores) e caches de 1024B, a TM perde 1,77%, mas apresenta redução no tempo de execução para 8P, 16P e 32P, alcançando 30,13% de redução com 32P e caches de 2048B. A Estimação de Movimento apresenta um comportamento semelhante, perdendo para 2P e 4P e atingindo ganho máximo com 32P.

Tabela 5.2: Diferença no tempo de execução entre TM e *locks*.

Aplicação	Cache	Número de Processadores				
		2P	4P	8P	16P	32P
Multiplicação de Matrizes	**512B**	1,86%	0,13%	-1,49%	-7,32%	-11,11%
	1024B	1,77%	0,28%	-1,62%	-7,24%	-12,66%
	2048B	0,32%	0,02%	-3,22%	-16,91%	-30,13%
Estimação de Movimento	**512B**	0,23%	0,41%	-0,24%	-5,81%	-23,10%
	1024B	0,11%	0,44%	-0,84%	-4,23%	-23,26%
	2048B	0,12%	0,32%	-0,77%	-6,01%	-24,38%
Codificador JPEG	**512B**	-1,00%	0,59%	-2,11%	-5,79%	-10,12%
	1024B	-0,15%	0,11%	-1,40%	-3,76%	-6,74%
	2048B	0,03%	0,21%	-1,15%	-2,79%	-3,43%

Um dos principais fatores responsáveis pelo ganho de performance na Multiplicação de Matrizes e na Estimação de Movimento está em uma etapa inicial que há nas duas aplicações, onde ocorre a inicialização das matrizes. Esta etapa é executada por um processador, enquanto os outros aguardam para entrar na seção crítica. Com o mecanismo de *locks*, os processadores que aguardam enviam repetidamente mensagens para o diretório até que o *lock* seja liberado. Estas mensagens irão sobrecarregar o diretório e conseqüentemente degradar a performance do processador que está executando a etapa de inicialização, já que nesta etapa há muitos acessos à memória. Com TM, os processadores que aguardam têm suas transações abortadas e aguardam um período longo antes de tentarem novamente (aproximadamente de 5000 a 25000 ciclos). Para caches maiores, estas aplicações executam mais rápido a parte paralela, logo aumentando a contribuição da TM para os ganhos totais.

O Codificador JPEG também aumenta sua performance com TM à medida que aumenta-se o número de processadores, mas apresenta alguns pontos fora da curva para 2P e 4P. Esta pequena variação pode ser explicada por uma singularidade na implementação LogTM deste trabalho. O Codificador JPEG possui uma longa etapa serial ao final que é atribuída a um dos processadores. Com *locks,* o processador eleito para executar esta parte serial é o primeiro a ter sua requisição lida pelo diretório no início da execução, que no caso será o processador com menor número de *hops* (menor latência) até a memória. Com TM, o processador eleito é o que tem a maior prioridade no sistema, definida pelo *timestamp* do início da transação, e, para *timestamps* iguais como neste caso, é definida pelo menor endereço físico de rede. O processador eleito para esta parte pode estar situado em um nodo de maior latência, conseqüentemente degradando a performance da aplicação. Esta situação também pode ocorrer com outras aplicações, entretanto as diferenças são mais evidentes com o Codificador JPEG.

A Tabela 5.3 mostra a diferença de energia do sistema entre memória transacional e *locks*. A TM apresenta redução de energia em todos os casos. De maneira geral, os resultados seguem os ganhos de performance à medida que aumenta o número de processadores do sistema, e, em muitos casos, os ganhos de energia ultrapassam os de performance, o que indica que a potência média também foi reduzida.

A Multiplicação de Matrizes apresentou as maiores reduções de energia, alcançando até 32,25% de redução para 16P e 2048B.

Tabela 5.3: Diferença na energia total do sistema entre TM e *locks*.

Aplicação	Cache	Número de Processadores				
		2P	4P	8P	16P	32P
Multiplicação de Matrizes	**512B**	-2,81%	-10,63%	-13,09%	-20,64%	-16,21%
	1024B	-2,37%	-10,81%	-14,47%	-22,21%	-16,70%
	2048B	-2,30%	-10,29%	-20,34%	-32,25%	-20,31%
Estimação de Movimento	**512B**	-0,59%	-2,17%	-3,84%	-7,01%	-18,80%
	1024B	-0,57%	-1,84%	-4,22%	-5,87%	-17,94%
	2048B	-0,38%	-1,95%	-3,66%	-7,27%	-17,35%
Codificador JPEG	**512B**	-2,64%	-2,35%	-7,26%	-5,74%	-8,08%
	1024B	-0,51%	-0,89%	-3,79%	-6,10%	-3,04%
	2048B	-0,17%	-0,39%	-2,30%	-4,19%	-2,70%

As Figuras 5.2 até 5.7 apresentam a energia consumida pelo sistema, incluindo processadores, memória, caches e NoC. As Figuras 5.2, 5.4 e 5.6 mostram a energia total do sistema, enquanto que as Figuras 5.3, 5.5 e 5.7 mostram a distribuição de energia entre os componentes do sistema. A notação 2P_L é utilizada para os resultados com dois processadores e *locks*, 2P_T para dois processadores e TM e de forma análoga para mais processadores.

À medida que o número de processadores aumenta, a energia da memória se torna mais significativa, tornando-se comparável à dos processadores. A contribuição da NoC para a energia total do sistema também aumenta à medida que há mais processadores no sistema.

A Tabela 5.4, que mostra a diferença na energia por componente do sistema para o Codificador JPEG, permite analisar quais os componentes que contribuem para a redução de energia da TM. Como se pode ver, a memória e a NoC são os principais

responsáveis pela redução da energia, chegando a 42,99% para a memória e 37,81% para a NoC com 32P.

Os resultados da TM obtidos nos experimentos apresentados nesta seção foram obtidos com valores fixos para *backoff delay on abort* (descrito na seção 4.3) em cada configuração do sistema. O tempo de espera usado foi de 5000 ciclos para 2P, 4P e 8P, 10000 ciclos para 16P e 25000 ciclos para 32P.

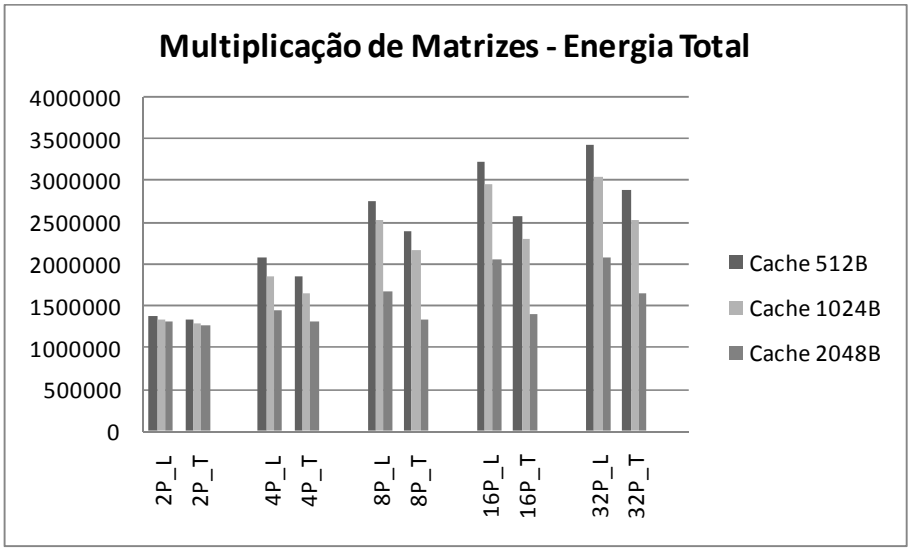

Figura 5.2: Energia total do sistema para Multiplicação de Matrizes

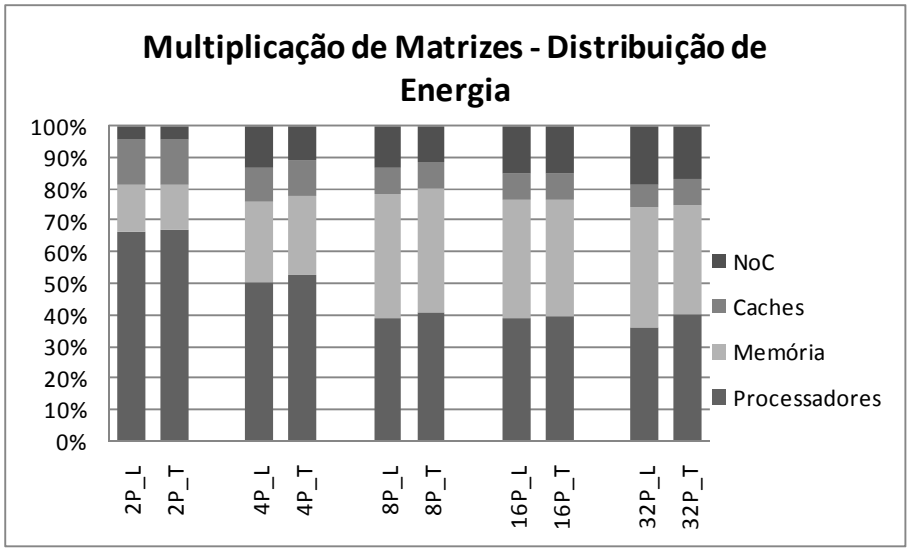

Figura 5.3: Distribuição de energia para Multiplicação de Matrizes

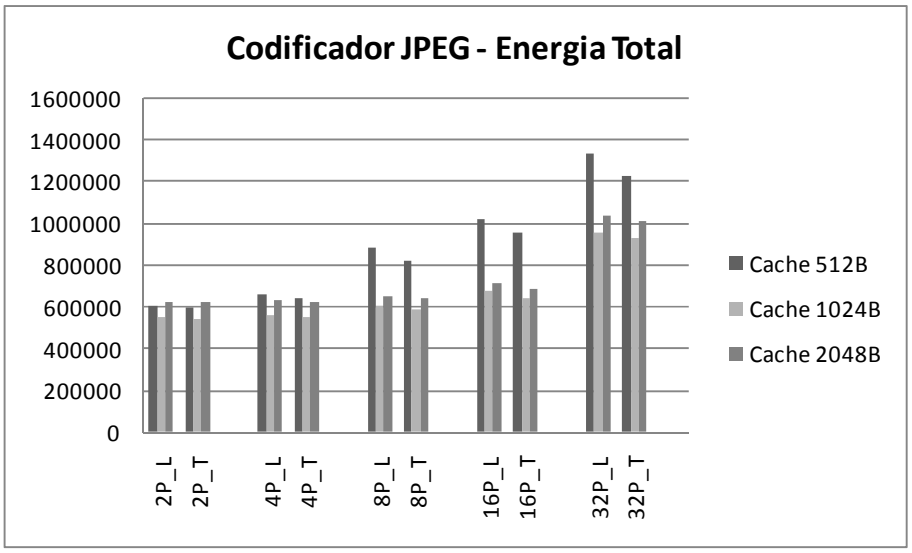

Figura 5.4: Energia total do sistema para o Codificador JPEG

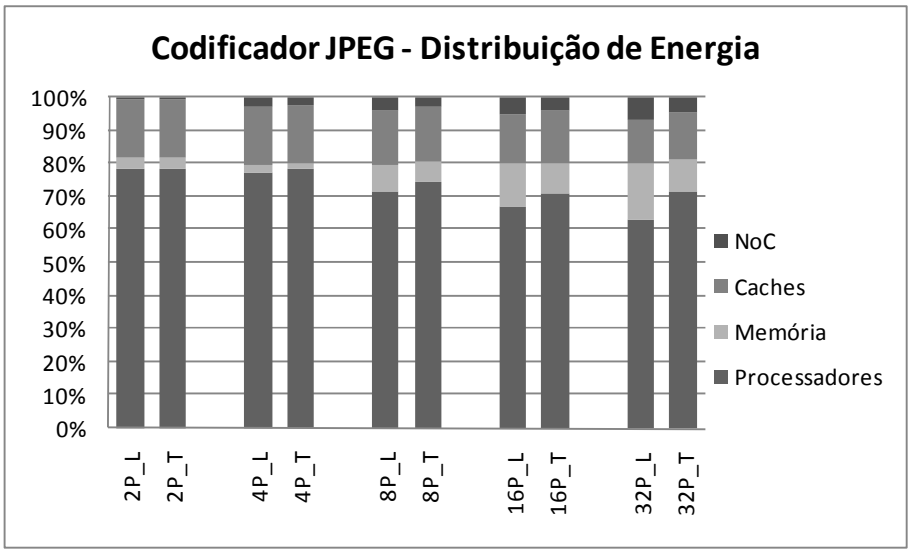

Figura 5.5: Distribuição de energia para o Codificador JPEG

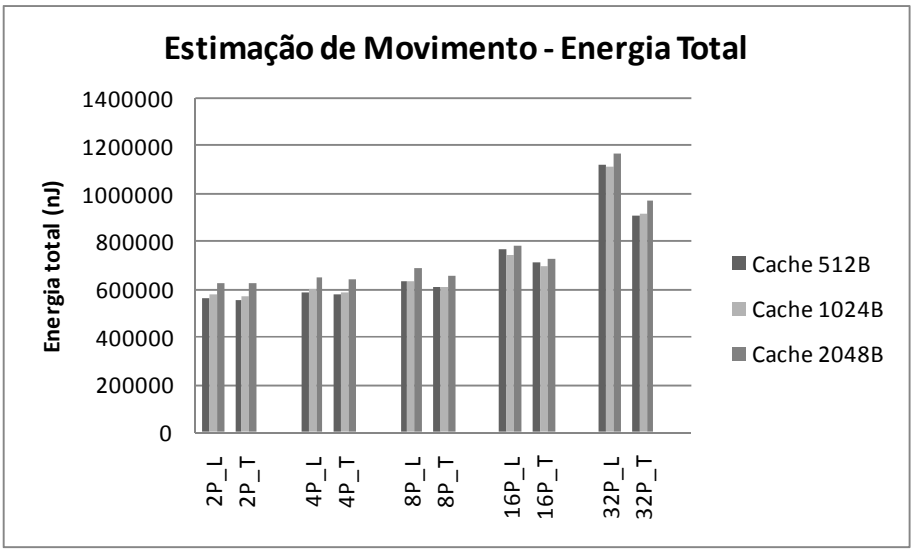

Figura 5.6: Energia total do sistema para a Estimação de Movimento

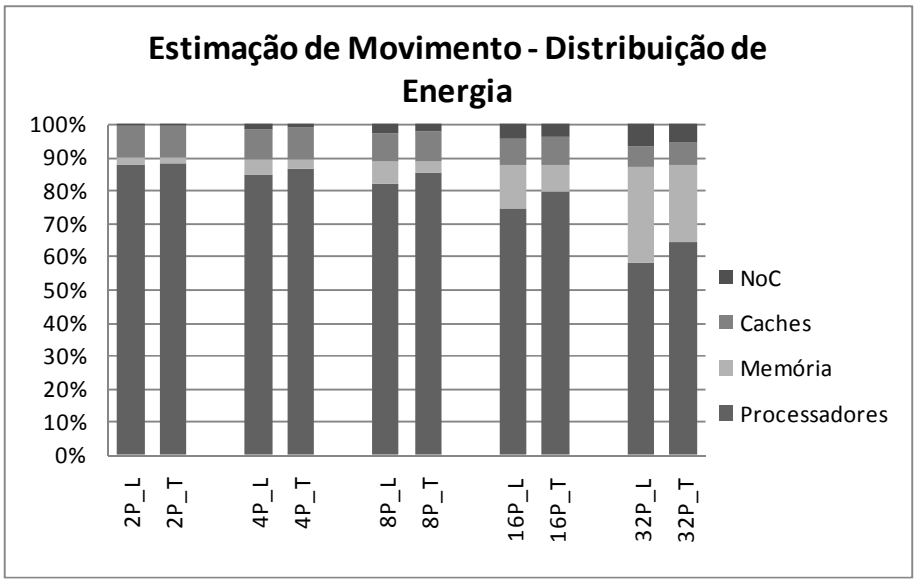

Figura 5.7: Distribuição de energia para a Estimação de Movimento

Tabela 5.4: Diferença na energia por componente entre TM e *locks* para o Codificador JPEG

	2P	**4P**	**8P**	**16P**	**32P**
Processadores	-0,26%	0,53%	-0,19%	-0,38%	9,62%
Memória	-7,47%	-20,02%	-25,95%	-33,03%	-42,99%
Caches	-0,21%	0,16%	-0,34%	-0,40%	6,86%
NoC	-2,07%	-27,47%	-35,57%	-29,30%	-37,81%
TOTAL	**-0,51%**	**-0,89%**	**-3,80%**	**-6,12%**	**-3,06%**

5.3.2 Políticas de *backoff*

As seções a seguir fazem uma análise do impacto que o *backoff delay on abort* (tempo de espera que a transação aguarda para reiniciar após abortar, descrito na seção 4.3) tem na performance e na energia do sistema utilizando TM. Para isto, três políticas de *backoff* são utilizadas: *backoff* fixo, exponencial e linear randômico. Cada uma das políticas possui um parâmetro inicial medido em ciclos como entrada, o qual é utilizado como referência para o tempo de espera, conforme descrito a seguir:

***Backoff* Fixo** – A transação aguarda sempre um valor constante de ciclos, definido pelo parâmetro de entrada.

***Backoff* Exponencial** – O tempo de espera é inicialmente o tempo definido pelo parâmetro de entrada, mas dobra a cada *abort* da transação. O *commit* da transação restaura o tempo de espera para o valor inicial.

***Backoff* Linear Randômico** – O tempo de espera é um valor randômico escolhido em cada *abort* no intervalo entre zero e o valor inicialmente igual ao parâmetro de entrada. A cada *abort*, o intervalo é incrementado com o valor inicial, fazendo com que cresça linearmente. No *commit*, o intervalo retorna ao valor inicial.

Os parâmetros iniciais utilizados nos experimentos para as políticas de *backoff* variam de 625 até 640000 ciclos, em intervalos exponenciais, sendo que cada valor é o dobro do anterior.

5.3.3 Influência do *backoff delay on abort* na performance

As Figuras 5.8 a 5.12 apresentam os gráficos de tempo de execução das aplicações com diferentes políticas de *backoff* e diferentes parâmetros de entrada para cada política. Algumas execuções entraram em *livelock* e não finalizaram. Estas execuções têm sua coluna omitida no gráfico. O objetivo destes gráficos não é escolher a melhor política de *backoff*, já que os resultados são dependentes da escolha dos parâmetros, escolhidos aqui de forma arbitrária, e sim mostrar o quanto a performance pode ser prejudicada com uma escolha de parâmetros inadequada. Além disso, espera-se ter uma idéia do comportamento de cada aplicação quanto ao dimensionamento dos parâmetros, mostrando que algumas aplicações são mais suscetíveis a variações do *backoff delay on abort* do que outras.

As aplicações Multiplicação de Matrizes, Codificador JPEG e Estimação de Movimento, por serem aplicações de baixa contenção, são menos suscetíveis, quanto à performance, à escolha dos parâmetros e políticas de *backoff*. Pode-se identificar certo padrão nestas aplicações, sendo que valores pequenos dos parâmetros não têm grande influência na performance, exceto para o *backoff* fixo, que ocasionou *livelocks* nos casos em que o parâmetro de entrada é demasiadamente pequeno. Parâmetros grandes tendem a prejudicar a performance gradativamente, principalmente na política exponencial.

A aplicação LeeTM, por executar transações na maior parte de sua execução, apresenta um comportamento interessante. Não há nenhuma tendência clara para a performance em relação ao dimensionamento dos parâmetros, mostrando uma suscetibilidade grande quanto à escolha do *backoff*.

Já a aplicação LeeTM-IP apresenta comportamento semelhante às aplicações de baixa contenção. Entretanto, este comportamento se deve a outro motivo. O tempo de *backoff* tem um impacto direto no nível de especulação do sistema, e, como esta

aplicação possui pouco paralelismo para ser explorado pela TM, a performance varia pouco para diferentes níveis de especulação. A energia da LeeTM-IP, entretanto, pode sofrer grande influência do nível de especulação.

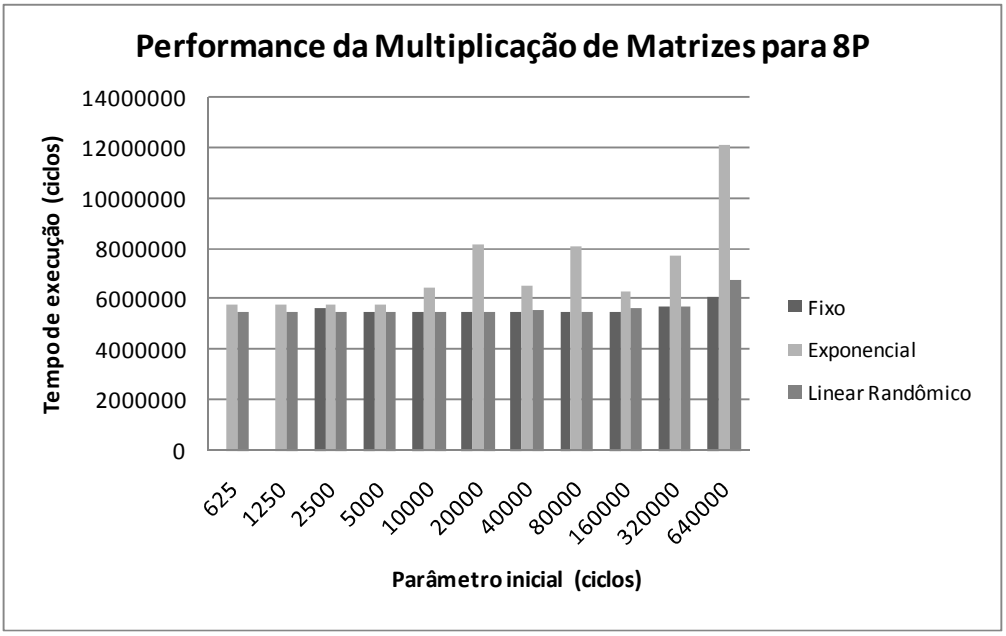

Figura 5.8: Tempo de execução da Multiplicação de Matrizes para diferentes políticas e parâmetros

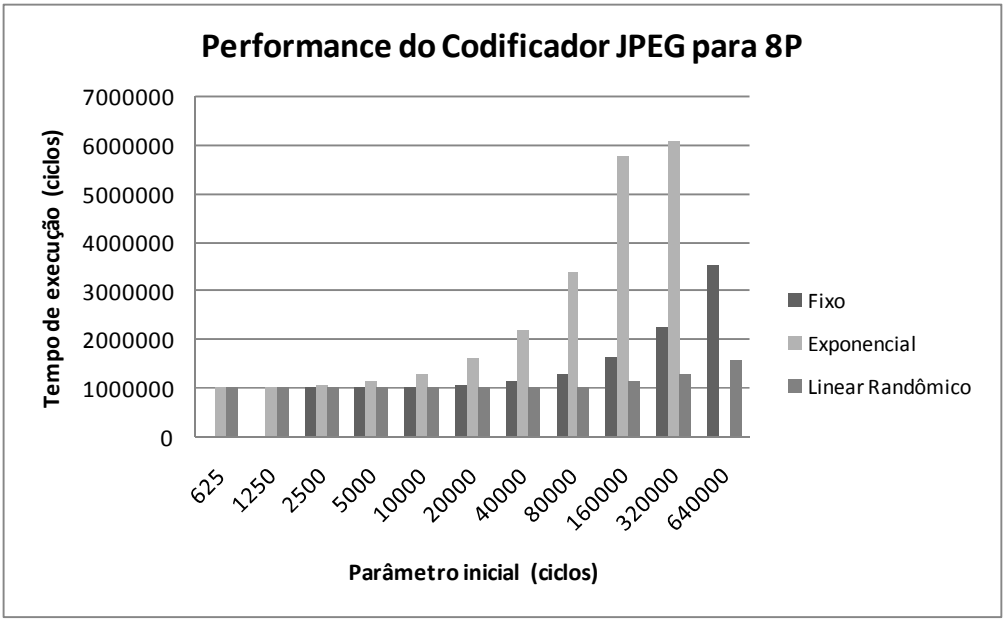

Figura 5.9: Tempo de execução do Codificador JPEG

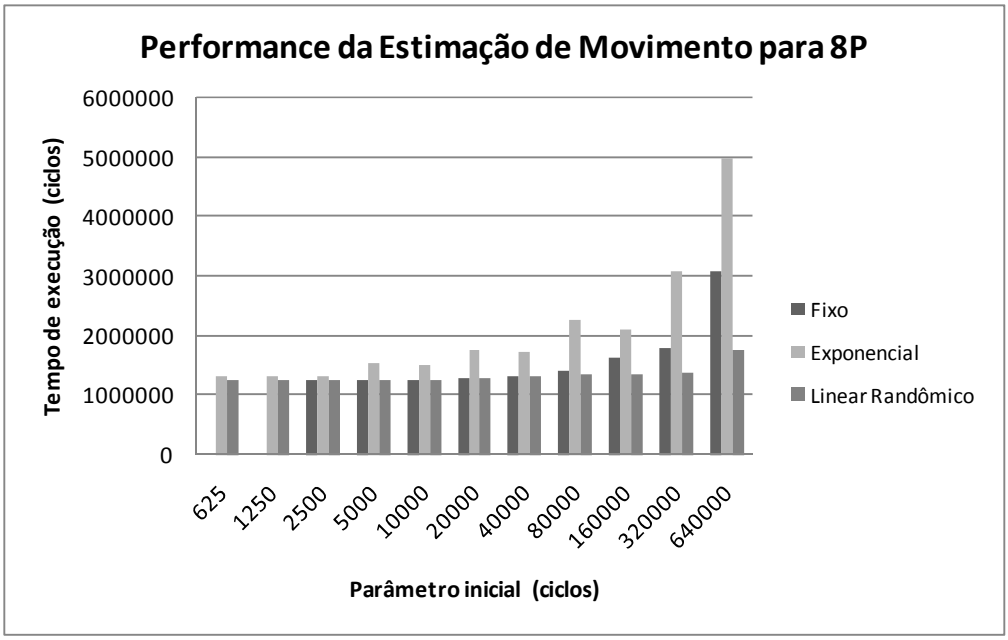

Figura 5.10: Tempo de execução da Estimação de Movimento

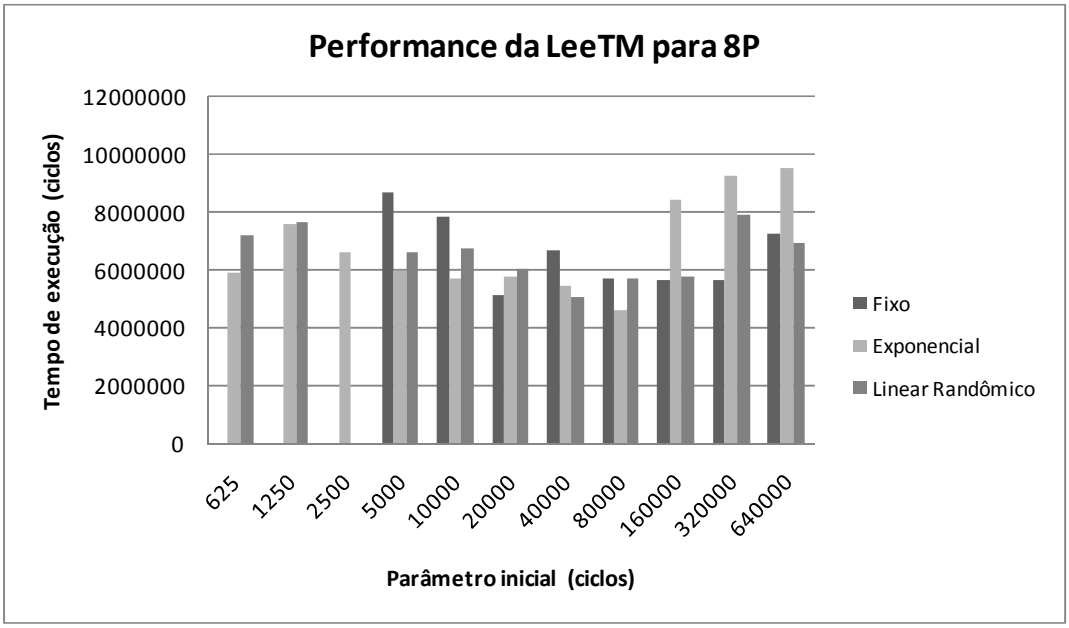

Figura 5.11: Tempo de execução da LeeTM

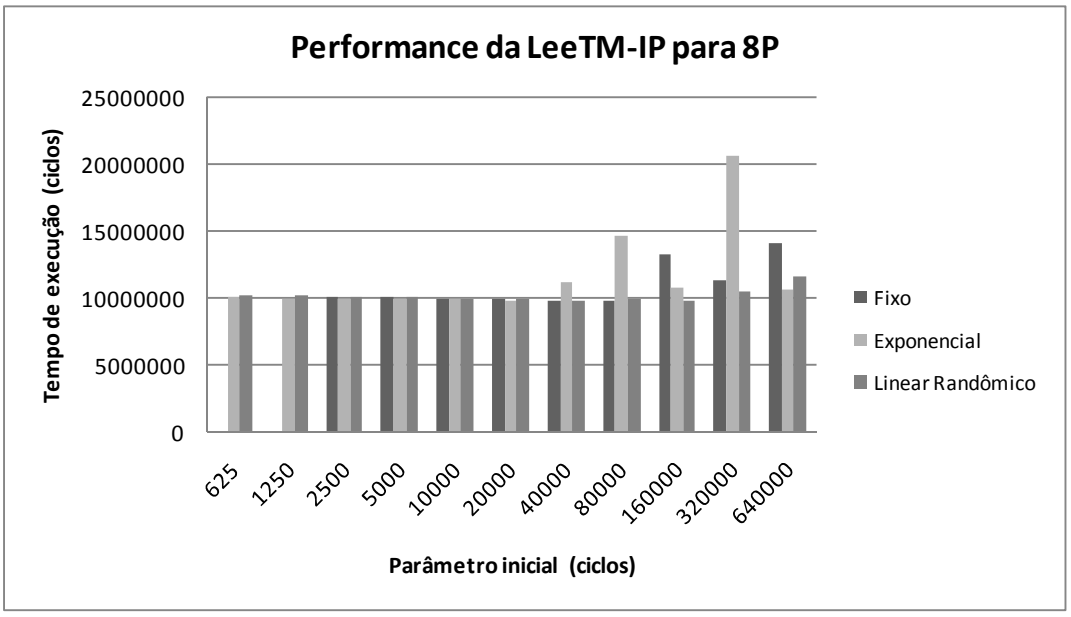

Figura 5.12: Tempo de execução do LeeTM-IP

Utilizando-se a aplicação LeeTM, é feita uma análise da dificuldade de encontrar o parâmetro ótimo para diferentes configurações do sistema. Para isso, encontrou-se o parâmetro que dá o menor tempo de execução para cada política de *backoff* de cada configuração do sistema. Os resultados são mostrados na Figura 5.13.

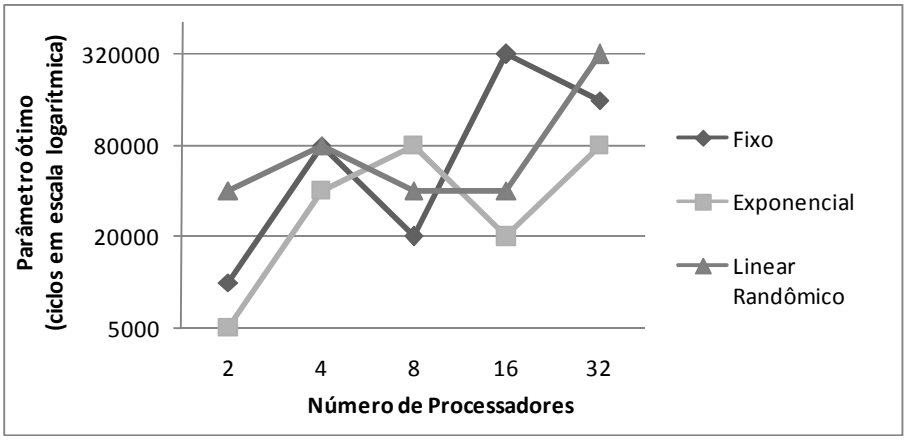

Figura 5.13: Parâmetros ótimos para cada política de *backoff* com LeeTM.

Para cada política e número de processadores, a performance ótima é obtida com valores diferentes do parâmetro de entrada. Além disso, nenhuma das políticas apresenta um comportamento previsível à medida que o número de processadores aumenta.

Para avaliar o impacto de parâmetros não-ótimos na performance, selecionou-se os parâmetros ótimos para 8P de cada política de *backoff* e executou-se a aplicação com diferentes número de processadores no sistema mantendo-se os mesmos parâmetros. A Figura 5.14 mostra os tempos de execução com os parâmetros otimizados para 8P, em valores relativos ao tempo de execução obtido com o parâmetro ótimo de cada configuração. A execução com a política de *backoff* fixo com parâmetro de 20000 ciclos resultou em *livelock* para 32P. No caso extremo, o tempo de execução de 16P com

parâmetro otimizado para 8P foi de 81,5% maior do que com o parâmetro otimizado para 16P.

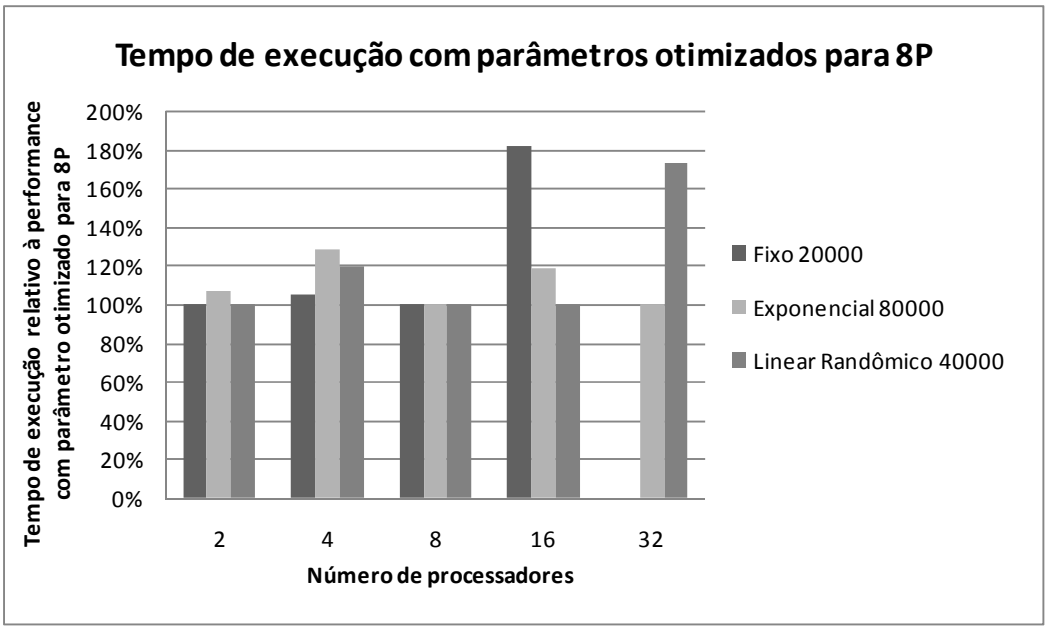

Figura 5.14: Impacto do uso de parâmetros não-ótimos no tempo de execução.

5.3.4 Análise das políticas de *backoff* e do *Abort Handshake*

Esta seção faz uma comparação das políticas de *backoff* e do mecanismo *Abort Handshake* proposto neste trabalho (detalhado na seção 4.4). Para esta comparação, utilizou-se os valores ótimos de tempo de execução obtidos neste trabalho para cada política de *backoff* e configuração do sistema. As aplicações utilizadas para esta análise foram a LeeTM e LeeTM-IP.

As Figuras 5.15 e 5.16 mostram o tempo de execução da LeeTM e da LeeTM-IP, respectivamente, com as três políticas de *backoff*, o mecanismo *Abort Handshake* e a versão com *locks*, enquanto que as Figuras 5.17 e 5.18 mostram a energia total. Pode-se ver que para a versão com *locks* das aplicações (versões com *locks* da LeeTM e LeeTM-IP são idênticas) não há vantagens em termos de performance nem energia para mais do que 4P. A explicação para isto é que o pouco paralelismo que existe para ser explorado impede a amortização do *overhead* da sincronização, que cresce com o número de processadores.

Os resultados de tempo de execução da aplicação LeeTM-IP com TM (políticas de *backoff* e *Abort Handshake*) são bem próximos da versão com *locks*, e, assim como esta, também não apresentam vantagens para mais de 4P. Entretanto, a análise dos resultados para mais processadores ajuda a entender o comportamento do sistema para cargas de aplicação de alta demanda de sincronização.

Nenhuma das políticas de *backoff* pode ser apontada como vencedora, já que todas tiveram resultados próximos e cada uma se mostrou melhor em algum caso particular. A única política que apresentou resultados divergentes é a exponencial para LeeTM-IP com 16P e 32P.

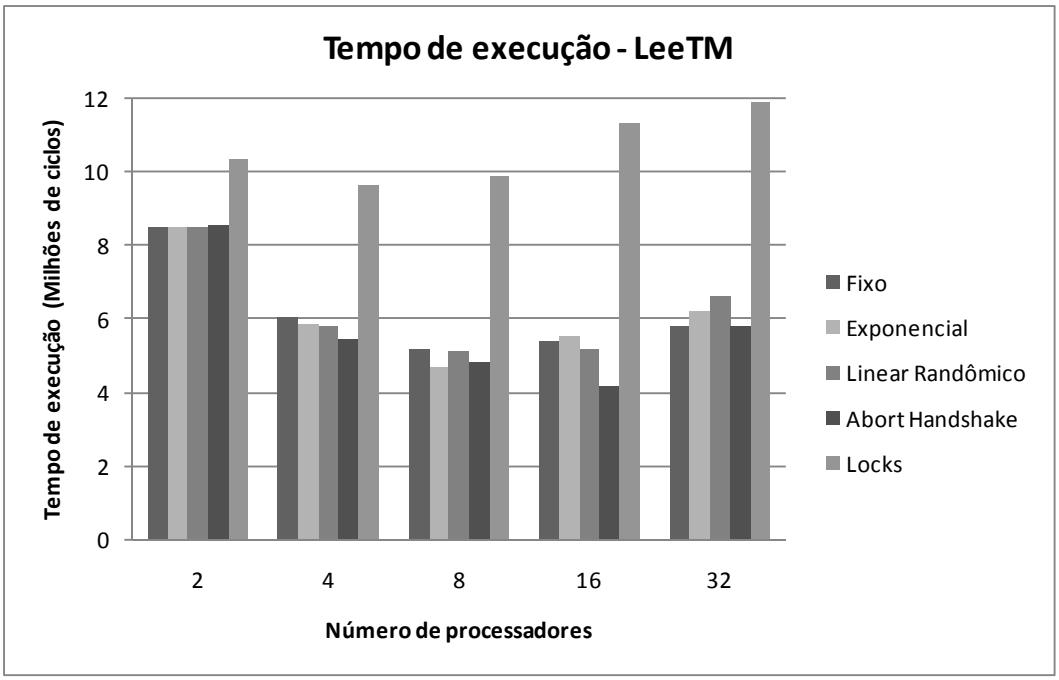

Figura 5.15: Tempo de execução com LeeTM.

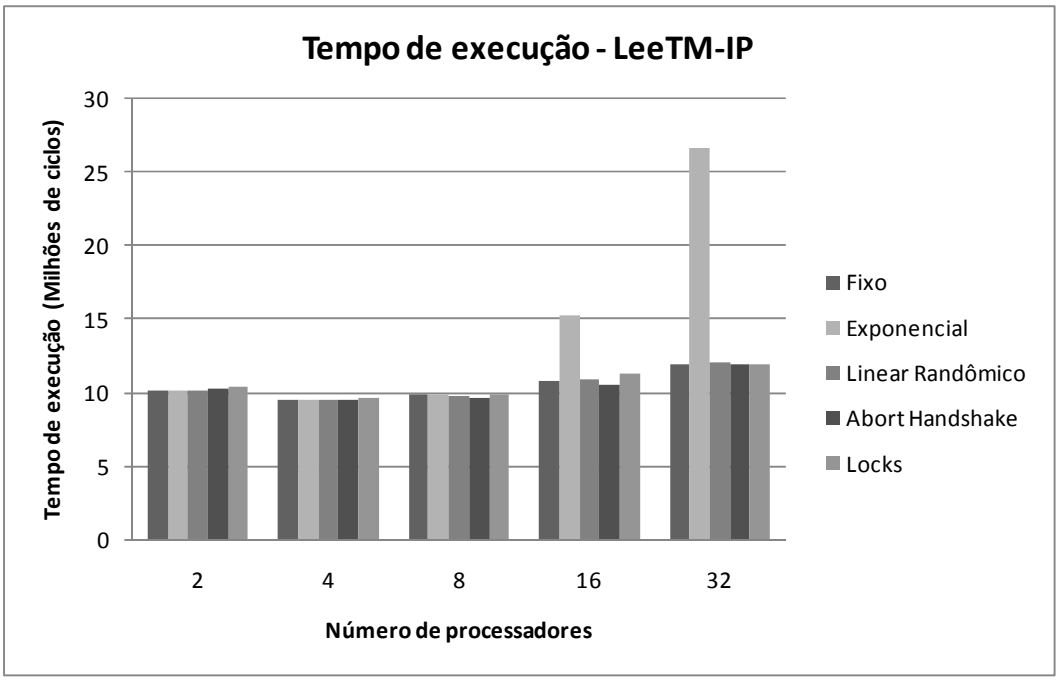

Figura 5.16: Tempo de execução com LeeTM-IP.

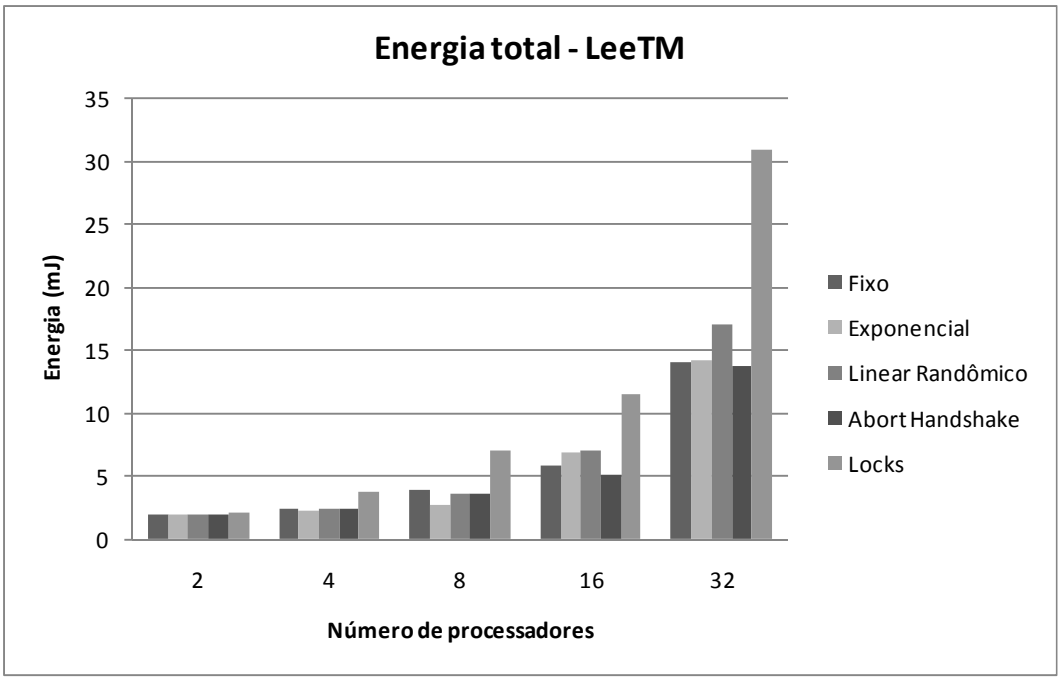

Figura 5.17: Energia total com LeeTM.

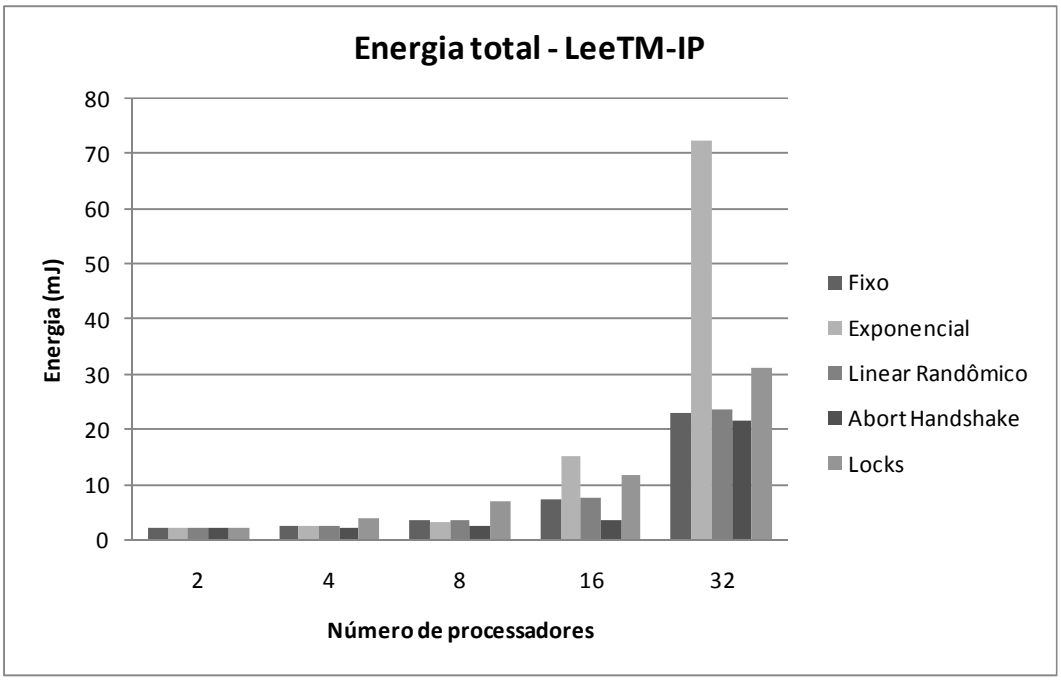

Figura 5.18: Energia total com LeeTM-IP.

A Tabela 5.4 mostra a redução no tempo de execução do mecanismo *Abort Handshake* em relação ao tempo de execução com as políticas de *backoff*. Para o LeeTM com 2P, o *Abort Handshake* prejudicou o tempo de execução em cerca de 0,5% em comparação com as três políticas, assim como o tempo de execução para 8P foi cerca de 4% pior se comparada à política exponencial. Nos outros casos, o *Abort Handshake* teve performance melhor do que todas as políticas de *backoff*, alcançando até 20,25% de redução no tempo de execução sobre a melhor política para 16P, o *backoff* linear randômico. Esta redução no tempo de execução é possível devido à menor especulação que o *Abort Handshake* proporciona. Com as políticas de *backoff*, as transações reiniciam sua execução de forma prematura, repetindo conflitos anteriores e congestionando a rede e o diretório ao executarem. O *Abort Handshake* evita que isto ocorra, mantendo os processadores das transações que abortaram parados até que as transações conflitantes tenham finalizado.

Para o LeeTM-IP, as reduções foram mais modestas em geral, já que neste caso a TM, tanto com as políticas de backoff como com o *Abort Handshake*, já explora o paralelismo próximo do limite disponível nesta aplicação. A exceção foram os resultados da comparação com o *backoff* exponencial para 16P e 32P (31% e 55%, respectivamente), mas a razão para estes valores altos se deve à baixa performance do *backoff* exponencial para estes casos. Em situações de alta contenção, o *backoff* exponencial aumenta muito o tempo de espera das transações, o que prejudica significativamente a performance. Exceto para 2P, o *Abort Handshake* não teve, em nenhum caso, performance pior do que todas as políticas de *backoff*.

A Tabela 5.5 apresenta os ganhos de energia do *Abort Handshake*. Exceto para a comparação com *backoff* exponencial e 8P, caso em que a energia do sistema com *Abort Handshake* é 33,7% maior, nos poucos casos em que há perdas na energia as diferenças são pequenas (inferiores a 1%). Nos outros casos, o *Abort Handshake* proporciona ganhos de energia. Mesmo com ganhos modestos de performance na maioria dos casos, ele apresenta ganhos significativos de energia para o LeeTM-IP, alcançando até 53,6% em relação à melhor política de *backoff*. Da mesma forma que a redução da especulação do *Abort Handshake* resulta em redução no tempo de execução, a energia também é reduzida. Para a aplicação LeeTM-IP esta redução foi mais significativa já que nesta aplicação os conflitos ocorrem mais cedo (no início das transações, ao tentarem escrever na variável compartilhada). Isso evita que transações que executaram trechos longos de código tenham que abortar.

Tabela 5.4: Ganhos de performance do *Abort Handshake* em comparação com as políticas de *backoff*.

Reduções no Tempo de Execução (LeeTM)					
	2P	**4P**	**8P**	**16P**	**32P**
Fixo	-0,51%	10,08%	6,50%	23,36%	0,03%
Exponencial	-0,51%	7,07%	-4,00%	25,32%	6,96%
Linear Randômico	-0,48%	5,71%	5,31%	20,25%	12,25%

Reduções no Tempo de Execução (LeeTM IP)					
	2P	**4P**	**8P**	**16P**	**32P**
Fixo	-1,40%	0,16%	2,17%	2,74%	-0,44%
Exponencial	-1,40%	0,27%	1,66%	31,12%	55,27%
Linear Randômico	-1,40%	0,08%	1,56%	3,47%	0,82%

Tabela 5.5: Redução de energia do *Abort Handshake* em comparação com as políticas de *backoff*.

Redução da Energia (LeeTM)					
	2P	**4P**	**8P**	**16P**	**32P**
Fixo	-0,04%	4,66%	9,86%	11,66%	2,07%
Exponencial	-0,04%	-0,78%	-33,70%	25,42%	3,18%
Linear Randômico	-0,08%	3,54%	2,70%	27,38%	19,07%

Redução da Energia (LeeTM_IP)					
	2P	**4P**	**8P**	**16P**	**32P**
Fixo	-0,66%	6,72%	23,94%	53,60%	5,25%
Exponencial	-0,66%	7,68%	20,04%	78,11%	70,06%
Linear Randômico	-0,05%	9,74%	24,67%	54,89%	8,41%

Ambas as aplicações apresentam, no início da execução, uma etapa em que todos os processadores lêem e incrementam uma variável compartilhada como forma de atribuir um identificador lógico para cada *thread* da aplicação. Como todos os processadores iniciam sua execução ao mesmo tempo e em seguida solicitam o bloco da variável para leitura em uma transação, o bloco é compartilhado para leitura por todos os processadores. Para que cada processador possa atualizar a variável, é necessário que todos os outros processadores de menor prioridade que possuem o bloco sejam abortados e tenham seu bloco invalidado. Nesta situação, ocorre uma particularidade que afeta a performance do *Abort Handshake*. Quando uma transação aborta, ela envia *Signal abort* para o primeiro processador que lhe enviou um NACK como resposta da tentativa de invalidação do bloco. Da forma como foi implementado, o diretório envia as invalidações para os processadores na ordem crescente de seus endereços de rede, que coincide com a ordem decrescente de prioridades das transações nos processadores. Assim, a primeira transação a receber invalidação é a de maior prioridade dentre as que compartilham o bloco. Logo, na maioria dos casos, o primeiro NACK recebido pela transação que irá abortar será da transação de maior prioridade, já que a latência da rede, em geral, não é suficiente para inverter a ordem dos NACKs recebidos. Dessa forma, a transação de maior prioridade, que será a próxima a conseguir atualizar o bloco, é a que recebe S*ignal abort* da maioria das transações que abortam, ficando responsável por sinalizá-las com *Release abort* após o *commit*. Esta situação faz com que estas transações sejam reiniciadas após a de maior prioridade conseguir atualizar o bloco, o que prejudica a performance, já que as transações que reiniciaram solicitam o bloco para leitura e devem ser abortadas novamente.

Esta situação adiciona um custo na performance que cresce exponencialmente com o número de processadores, sendo este o principal fator responsável pela queda abrupta da performance do *Abort Handshake* sobre as políticas de *backoff* de 16P para 32P.

A Tabela 5.6 apresenta uma comparação entre memória transacional e *locks* para as aplicações LeeTM e LeeTM-IP. Para memória transacional, utilizou-se o *Abort Handshake*, já que no geral apresentou ganhos mais consistentes em relação às políticas de *backoff*. A aplicação LeeTM obteve reduções no tempo de execução de até 63,38% e redução de energia de até 55% em relação a versão com *locks*, confirmando o ótimo potencial da TM com esta aplicação. A aplicação LeeTM-IP, conforme esperado, teve resultados de performance próximos à versão com *locks*. Ainda assim, conseguiu redução de quase 7% para 16P, perdendo apenas 0,61% para 32P. Entretanto, a LeeTM-

IP mostrou os maiores ganhos de energia, atingindo uma redução de até 71% na energia total do sistema.

Tabela 5.6: Comparação do tempo de execução e energia entre TM utilizando *Abort Handshake* e *Locks* para LeeTM e LeeTM-IP

Diferença no Tempo de Execução (TM com Abort Handshake x Locks)					
	2P	**4P**	**8P**	**16P**	**32P**
LeeTM	-17,10%	-43,32%	-50,83%	-63,38%	-51,27%
LeeTM-IP	-0,50%	-1,47%	-2,21%	-6,92%	0,61%

Diferença de Energia (TM com Abort Handshake x Locks)					
	2P	**4P**	**8P**	**16P**	**32P**
LeeTM	-3,81%	-36,84%	-48,97%	-55,63%	-55,45%
LeeTM-IP	-4,85%	-41,86%	-63,87%	-71,32%	-30,00%

5.4 Considerações Finais

Os experimentos neste capítulo foram divididos em duas partes principais. A primeira faz uma análise da performance e energia da implementação da TM em comparação com o mecanismo de *locks* com aplicações de baixa contenção. Os resultados mostram que os ganhos da TM em relação aos *locks* tendem a aumentar com o número de processadores. Estes experimentos mostraram redução no tempo de execução de até 30% e redução da energia do sistema em até 32%. A TM também reduz a energia consumida pela memória e NoC, sendo estes os principais componentes responsáveis pela redução de energia total do sistema.

A segunda parte dos experimentos leva em consideração aplicações de alta contenção em que o *backoff delay on abort* tem uma contribuição significativa para os resultados. Para isso, é feita uma análise do seu impacto nas aplicações. Três políticas de *backoff* são avaliadas e comparadas com a solução proposta neste trabalho, o mecanismo *Abort Handshake*. Nenhuma das políticas de *backoff* pode ser escolhida como vencedora, já que todas tiveram resultados parecidos e a melhor depende de cada caso. O *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, há algumas peculiaridades que podem afetar a performance do *Abort Handshake*. São necessários trabalhos futuros para analisar melhor esta situação e propor uma correção.

A comparação entre TM e *locks* para aplicações LeeTM e LeeTM-IP mostra o potencial de ganhos de performance e energia da memória transacional. A redução no tempo de execução chegou a 63,38% para LeeTM, o que representa um *speedup* de 2,73 em relação a versão com *locks*. A TM também mostrou ganhos significativos na energia, inclusive em situações em que há pouco paralelismo disponível e os ganhos de performance são modestos, como na aplicação LeeTM-IP.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi feita uma implementação do modelo LogTM de memória transacional em um sistema MPSoC baseado em NoC. A análise leva em consideração performance e energia em comparação com o mecanismo de *locks*.

O trabalho ainda estuda o impacto que o tempo que uma transação deve aguardar para reiniciar após abortar (*backoff delay on abort*) pode ter na performance e energia do sistema. Três políticas de *backoff* são utilizadas para esta análise e é proposto um mecanismo chamado *Abort Handshake*, baseado em sinalização entre transações, como alternativa às políticas de *backoff*.

Os experimentos realizados com aplicações consideradas de baixa contenção mostraram que a memória transacional apresenta ganhos de performance e energia na maioria dos casos, e estes ganhos tendem se tornar mais significativos à medida que aumenta o número de processadores no sistema. A memória transacional apresentou redução de até 30% no tempo de execução e até 32% na energia. Os resultados mostram também que a energia reduzida com memória transacional se deve principalmente à memória e à NoC. A TM evita o envio de repetidas mensagens de *test-and-set* para o diretório ao tentar acessar uma seção crítica, reduzindo o tráfego na rede e o número de acessos à memória.

A segunda parte dos experimentos utiliza aplicações de elevado grau de sincronização para mostrar a influência do *backoff delay on abort* na performance e energia do sistema. É mostrado que a escolha da política de *backoff* e o dimensionamento dos parâmetros envolvidos no tempo de espera que otimizam a performance não são facilmente previsíveis e variam conforme a configuração do sistema e aplicação. A escolha de parâmetros não-otimizados para a configuração do sistema pode acarretar em perdas significativas da performance.

Os resultados ainda mostram que é difícil escolher uma política de *backoff* vencedora dentre as analisadas. A melhor política varia conforme o caso. O mecanismo *Abort handshake* proposto apresenta redução no tempo de execução na maioria dos casos, comparando-se com a melhor política de *backoff* dentre as analisadas. Houve redução no tempo de execução de até 20% e redução na energia de até 53%.

A comparação entre TM e *locks* para aplicações de elevado grau de sincronização mostra que é possível obter reduções no tempo de execução de até 63,38% e reduções na energia de até 71%.

Existem ainda algumas peculiaridades que podem afetar a performance do *Abort Handshake*, como foi o caso das aplicações LeeTM e LeeTM-IP com 32 processadores. Como trabalho futuro, pretende-se investigar esta situação e propor uma solução.

O mecanismo *Abort Handshake* pode ser considerado conservador, já que serializa a execução das transações que conflitaram. Variações mais agressivas desta abordagem com maior nível de especulação podem ser estudadas em trabalhos futuros. Por exemplo, uma transação T2, que aguarda T1, poderia ser autorizada a reiniciar caso T1 aborte ao conflitar com T0. É possível que T0 e T2 possam executar em paralelo sem conflitos, aumentando o paralelismo, enquanto que com o *Abort Handshake* as três transações seriam serializadas.

Os trabalhos futuros também incluem experimentos com outras aplicações de *benchmark*, para que se possa estudar o comportamento do sistema para diferentes granularidades de transações e cargas de aplicação.

Pretende-se ainda analisar o desempenho com caches de menor associatividade, com as quais haveria maior índice de *overflow* da cache. Uma análise do efeito que o tamanho do bloco de cache tem na taxa de conflitos, e, conseqüentemente na performance da TM, também poderá ser realizado. Blocos pequenos aumentam a taxa de falsos conflitos, por outro lado, aumentam o *overhead* da detecção de conflitos, e, ao mesmo tempo, prejudicam a taxa de *miss* das caches por reduzir o aproveitamento da localidade espacial dos acessos à memória.

REFERÊNCIAS

ANANIAN, C. S.; *et. al*. Unbounded Transactional Memory. In: 11TH INT. SYMP. ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 2005, San Francisco. **Proceedings…** Washington, DC: IEEE Computer Society, pp. 316–327, Fev. 2005.

ANDERSON, T. E. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. **IEEE Transactions on Parallel and Distributed Systems,** [S.l.], v.1, n.1, Jan. 1990.

ANSARI, M.; *et. al.* Lee-TM: A Non-trivial Benchmark for Transactional Memory. In: 8TH INTERNATIONAL CONFERENCE ON ALGORITHMS AND ARCHITECTURES FOR PARALLEL PROCESSING (ICA3PP '08), Agia Napa, Cyprus. **Proceedings…** Heidelberg, Berlin: Springer-Verlag, p.196-207, 2008.

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: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2007, Rio de Janeiro. **Proceedings…** New York, NY: ACM, 2007. p. 282-287.

BENINI, L.; DE MICHELI, G. Networks on chips: a new SoC paradigm. **IEEE Computer**, [S.l.], v. 35, n. 1, p. 70-78, Jan. 2002.

BOBBA, J., *et. al.* Performance pathologies in hardware transactional memory. **SIGARCH Comput. Archit. News** v.35, n.2, p.81-91, 2007.

BOBBA, J.; *et. al.* TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. **ACM SIGARCH Computer Architecture News**, v.36, n.3, p. 127-138, 2008.

CASCAVAL, C.; *et. al*. Software transactional memory: why is it only a research toy?. **Communications of the ACM**, [S.l], v.51, n. 11, p. 40-46, Nov. 2008.

CEZE, L.; *et. al*. Bulk disambiguation of speculative threads in multiprocessors. In: 33RD INT. SYMP. ON COMPUTER ARCHITECTURE**,** 2006, Boston, MA. **Proceedings…** Washington, DC: IEEE Computer Society, pp. 227–238, 2006.

CHAFI, H.; *et. al*. A Scalable, Non-blocking Approach to Transactional Memory. In: 13TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE, **Proceedings…** Phoenix, Arizona: IEEE Computer Society, pp. 97-108, 2007.

DIJKSTRA, E. W. Solution of a problem in concurrent programming control. **Communications of the ACM**, [S.l.], v.8, n.9, p.569, Set. 1965.

DRAGOJEVIC, A.; *et. al*. Why STM can be more than a Research Toy?. **Communications of the ACM**, 2010.

FERRI, C.; *et. al*. A hardware/software framework for supporting transactional memory in a MPSoC environment. **ACM SIGARCH Computer Architecture News,** v.35, n.1, p.47-54, 2007.

FERRI, C.; *et. al*.. Energy Implications of Transactional Memory for Embedded Architectures. In : WORKSHOP ON EXPLOITING PARALLELISM WITH TRANSACTIONAL MEMORY AND OTHER HARDWARE ASSISTED METHODS (EPHAM'08), April 2008.

FERRI, C.; *et. al*. Energy and Throughput Efficient Transactional Memory for Embedded Multicore Systems. In: INTERNATIONAL CONFERENCE ON HIGH-PERFORMANCE EMBEDDED ARCHITECTURES AND COMPILERS (HIPEAC), January 2010. **Proceedings…**. January 2010.

FERRI, C.; *et. al*. Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems. **Journal of Parallel and Distributed Computing**. v.70, n.10, p.1042-1052, October 2010.

GIRÃO, G. **Estudo sobre o Impacto da Hierarquia de Memória em MPSoCs baseados em NoC**. 2009. 94p. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

HERLIHY, M.; *et. al.* Software Transactional Memory for Dynamic-sized Data Structures. In: 22ND ACM SYMP. ON PRINCIPLES OF DISTRIBUTED COMPUTING, Boston, Massachusetts. July 2003. **Proceedings…**. ACM, New York, NY, p.92-101. 2003.

HERLIHY, M.; MOSS, J. E. B. Transactional Memory: Architectural Support for Lock-Free Data Structures. In: 20TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, San Diego, CA**, Proceedings…** New York, NY: ACM, p. 289–300, 1993.

ITO, S.A.; CARRO, L.; JACOBI, R.P. Making Java work for microcontroller applications. **Design & Test of Computers, IEEE** Volume 18, n. 5, p.100-110.

JOUPPI, N.P.; Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. **ACM SIGARCH Computer Architecture News**, v.18, n.3, p. 364-373, 1990.

KLEIN, F.; *et. al.* On the energy-efficiency of software transactional memory. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2009, Natal, Brazil. **Proceedings…** New York, NY: ACM, p. 33:1-33:6, 2009.

KUMAR, S.; *et. al*. Hybrid Transactional Memory. In: ELEVENTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING. New York, **Proceedings…**, New York, NY: ACM, p. 209-220, 2006.

LARUS J. R.; RAJWAR, R. **Transactional Memory**. [S.l.:s.n] Morgan & Claypool, 2006.

LOMET, D.B. Process structuring, synchronization, and recovery using atomic actions. In: ACM CONFERENCE ON LANGUAGE DESIGN FOR RELIABLE SOFTWARE, Raleigh, 1977, **Proceedings…** New York, NY: ACM, p. 128–137, 1977.

MAGNUSSON, P. S.; *et. al.* Simics: A full system simulation platform. **IEEE Computer**, [S.l.], v. 35, n. 2, p. 50–58, 2002.

MELLOR-CRUMMEY, J. M.; SCOTT M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. **ACM Trans. on Computer Systems**, [S.l.], v.9, n.1, p. 21–65, 1991.

MOORE, K. E., *et. al*. LogTM: Log-based Transactional Memory. In: 12TH INT. SYMP. ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, Austin, Texas, 2006, **Proceedings…** Austin, Texas, IEEE Computer Society, pp. 254–265, 2006.

MOORE, K. E. **Log-based Transactional Memory**. 2007. 107p. Tese (Doutorado em Ciência da Computação) – University of Wisconsin – Madison.

MORESHET, T.; BAHAR, R.I.; HERLIHY, M. Energy reduction in multiprocessor systems using transactional memory. In: 2005 INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, San Diego, CA. **Proceedings…** New York, NY: ACM, p. 331–334, 2005.

MORESHET, T.; BAHAR, R.I.; HERLIHY. Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks. In: WORKSHOP ON MEMORY PERFORMANCE ISSUES, February 2006.

RAJWAR, R.; GOODMAN, J. R. Transactional lock-free execution of lock-based programs. In: 10TH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS. San Jose, California. **Proceedings…** New York, NY: ACM, p. 5-17, 2002.

RAJWAR, R., HERLIHY, M., LAI, K. Virtualizing Transactional Memory. In: 32ND ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, Madison, Wisconsin. **Proceedings…** Washington, DC, IEEE Computer Society, p. 494-505, 2005.

RUDOLPH, L. ; SEGALL, Z. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. **ACM SIGARCH Computer Architecture News**, [S.l], v.12, n.3, Jun. 1984.

SANYAL, S.; *et. al.* Clock Gate on Abort: Towards Energy-efficient Hardware Transactional Memory. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED PROCESSING, Rome, Italy. **Proceedings…** Washington, DC, USA, IEEE Computer Society, p.1-8, 2009.

SCHERER III, W.N.; SCOTT, M.L. Contention Management in Dynamic Software Transactional Memory. In: ACM PODC WORKSHOP ON CONCURRENCY AND SYNCHRONIZATION IN JAVA PROGRAMS, St. John's, NL, Canada. July 2004. **Proceedings…** July 2004.

SUTTER, H. The free lunch is over: a fundamental turn toward concurrency in software. **Dr. Dobb's Journal,** [S.l.], v.30, n.3**,** Mar. 2005. Disponível em: <http://www.gotw.ca/publications/concurrency-ddj.htm>. Acesso em out. 2010.

TAUBENFELD, G. **Synchronization algorithms and concurrent programming**. Harlow, [s.n.], Pearson Prentice Hall, 2006.

TITOS, J. R.; *et. al.* Characterization of Conflicts in Log-Based Transactional Memory (LogTM). In: 16TH EUROMICRO CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP 2008). Toulouse, France. **Proceedings…** Washington, DC, USA, IEEE Computer Society, p.30-37, 2008.

TITOS, J. R.; *et. al.* Speculation-based conflict resolution in hardware transactional memory. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED PROCESSING (IPDPS '09), Rome, Italy. **Proceedings…** Washington, DC, USA, IEEE Computer Society, p.1-12, 2009.

WANG, H. Orion: A Power-performance Simulator for Interconnection Networks. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 2002. **Proceedings...** New York: ACM, 2002. p. 294-305.

WILTON, S.; JOUPPI, N. Cacti: an enhanced access a cycle time model, **IEEE Journal of Solid State Circuits**, Vol. 31, No. 5, p.677-688, 1996.

WOO, S. C.; *et. al.* The splash-2 programs: Characterization and methodological considerations. In: 22ND INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, S. Margherita Ligure, Italy, **Proceedings…**, New York, NY: ACM, p. 24–36, 1995.

YEN, L.; *et. al.* LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In: 13TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA), Phoenix, Arizona. **Proceedings…** Washington, DC, IEEE Computer Society, p. 261-272, 2007.

ZEFERINO, C.; SUSIN, A. SoCIN: a parametric and scalable network-on-chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2003, São Paulo. **Proceedings…** Los Alamitos, CA: IEEE Computer Society 2003. pp.169-174.