Proposta de um escalonador de transações para TinySTM aplicado a arquiteturas NUMA

Michael A. Costa[1]*, **Mauricio L. Pilla**[1], **André Rauber Du Bois**[1]

{macosta,pilla,dubois}@inf.ufpel.edu.br

[1]Laboratory of Ubiquitous and Parallel Systems – CDTec
Universidade Federal de Pelotas
Caixa Postal 354 – CEP 96001-970 – Pelotas, RS – Brasil

Resumo. *Memórias Transacionais são uma alternativa aos métodos de sincronizações baseados em* locks*. As arquiteturas* NUMA *tem a vantagem de adicionar mais processadores sem aumentar o gargalo de acesso ao barramento. Visando melhorar o desempenho das memórias transacionais em arquiteturas* NUMA *este trabalho apresenta a proposta de um escalonador de transações para* TinySTM *aplicado a arquiteturas* NUMA*.*

1. Introdução

As arquiteturas paralelas estão onipresentes nas plataformas computacionais modernas. Para que as aplicações possam extrair o melhor desempenho das arquiteturas paralelas, o código deve explorar ao máximo o poder computacional oferecido pelas diversas unidades de processamento. Porém, a programação paralela está longe de ser uma atividade trivial. Memórias Transacionais (TM) apresentam uma abstração que facilita o desenvolvimento de programas concorrentes.

Memórias transacionais usam o conceito de transações, parecidas com aquelas presentes em banco de dados para realizar a sincronização entre *threads* concorrentes. Na programação utilizando *Software Transactional Memory* (STM), o acesso à memória compartilhada é realizado dentro de uma transação executada atomicamente.

Em memórias transacionais, caso duas transações acessem a mesma área de memória compartilhada, uma das transações é cancelada, assim, gerando um *abort*. O aumento do paralelismo existente nas arquiteturas NUMA (*Non-Uniform Memory Access*) proporciona um aumento no número de contenção, ocasionando conflitos e *aborts* nas STMs. A biblioteca de STM *TinySTM*, apresenta tratamento de conflitos no acesso à memória, mas não evita que os *aborts* ocorram, assim, degradando o desempenho de execução de programas.

As arquiteturas NUMA possuem múltiplos processadores, compostos por vários núcleos e blocos de memória dedicados, a sua principal característica é o custo de tempo diferente para o acesso à memória. A característica da *TinySTM* descrita acima, torna-se mais agravante nas arquiteturas NUMA devido às diferentes latências de acesso à memória, impedindo que o programa utilize com total eficiência os recursos disponíveis.

Neste artigo, propõe-se um escalonador de transações consciente da arquitetura NUMA para a biblioteca de STM *TinySTM*, com objetivo de reduzir o número de *aborts* e otimizar o desempenho.

*Bolsista de Iniciação Ciêntifica da FAPERGS

O artigo é dividido da seguinte forma: A Seção 2 apresenta o conceito de Memórias Transacionais e aborda a biblioteca de STM *TinySTM*. A Seção 3 apresenta as motivações para a atual proposta de escalonador. A Seção 4 apresenta trabalhos relacionados e detalha a proposta deste trabalho. Por fim, a Seção 5 mostra as conclusões e principais contribuições.

2. Memórias Transacionais (TM)

Memórias Transacionais são mecanismos de sincronização que permitem execuções atômicas e isoladas sobre dadoes em memória compartilhada. As TMs são estudadas para no futuro substituírem as sincronizações baseadas em *locks*.

Na programação utilizando STMs, todo o acesso à memória compartilhada é realizado dentro de transações e todas as transações são executadas atomicamente em relação a transações concorrentes. Estas transações utilizam os mecanismos de versionamento de dados e detecção de conflitos para manter a consistência dos dados.

O versionamento de dados é responsável pelo gerenciamento dos dados. Ele armazena tanto o valor do dado no início de uma transação como também o valor do dado modificado durante a transação, para garantir a propriedade de atomicidade [Baldassin 2009].

Mecanismos de detecção de conflitos verificam a existência de operações conflitantes durante uma transação. Um conflito ocorre quando duas transações estão acessando um mesmo dado na memória e pelo menos uma das transações está fazendo uma operação de escrita [Baldassin 2009].

As TMs podem ser implementadas em *Software* (STM) ou em *Hardware* (HTM). Para esta proposta foi estudada a biblioteca de STM *TinySTM*.

A *TinySTM* [Felber et al. 2008] é uma implementação de STM para as linguagens C e C++. Seu algoritmo é baseado em outros algoritmos de STM como o TL2 (*Transactional Locking* 2) [Dice et al. 2006], mas apresenta diferentes formas de versionamento e gerenciamento de contenção.

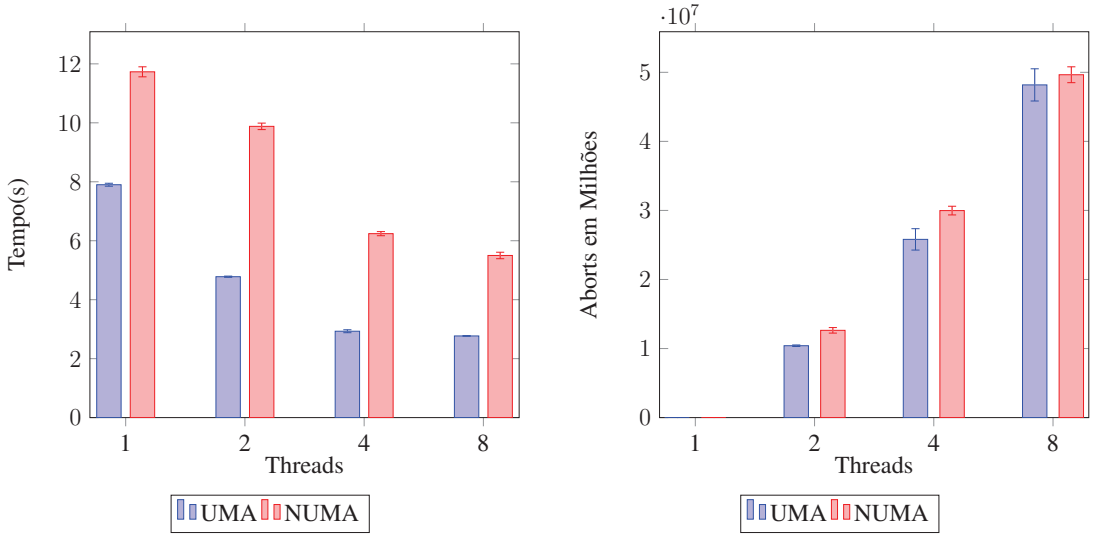
A *TinySTM* apresenta três estratégias de versionamento distintas que podem ser utilizadas, sendo que duas utilizam versionamento atrasado (*write-back*) e uma utiliza versionamento adiantado (*write-through*). Para a implementação do gerenciamento de contenção, a *TinySTM* apresenta quatro estratégias, sendo utilizada a *CM_Suicide* como sua estratégia padrão. Nesta estratégia a transação que detecta o conflito é interrompida imediatamente.

3. Motivação

Dependendo da localização física da memória em relação aos processadores, o tempo de acesso a uma posição de memória pode ser uniforme ou não. Surgem então as arquiteturas denominadas de *UMA (Uniform Memory Access)* e *NUMA (Non-Uniform Memory Access)* [Carissimi et al. 2007]. Máquinas *NUMA* tem a vantagem de agregar maior paralelismo ao adicionar mais processadores sem aumentar o gargalo de acesso ao barramento.

Porem as STMs apresentam um pior desempenho em arquiteturas NUMA quando comparadas a arquiteturas UMA. Isto ocorre em razão dos *aborts* em máquinas NUMA possuírem maior custo devido as diferentes latências de acesso à memória.

Para este trabalho foram realizados testes a fim de comparar o desempenho da biblioteca *TinySTM* em arquiteturas *UMA* com arquiteturas *NUMA*. Como pode ser visto na Figura 1, o *benchmark Intruder* utilizando a biblioteca *TinySTM* tem pior desempenho de tempo quando executados em arquiteturas NUMA.



(a) Média de Tempo **(b)** Média de Aborts

Figura 1. Comparação entre as arquiteturas UMA e NUMA usando o benchmark Intruder.

Para a execução destes testes foram utilizadas duas máquinas, uma com arquitetura NUMA de processador Intel Xeon E5620 com 2 nodos de 4 *cores* e 8 *threads* em *hyper threading* cada, totalizando 16 *threads*, com memória RAM de 12GB, e uma máquina de arquitetura UMA com processador Intel Core i7 2600 com 4 *cores* e 8 *threads* em *hyper threading*, com memória RAM de 8GB. Foram realizadas 60 execuções para cada conjunto de 1, 2, 4 e 8 *threads* em cada uma das máquinas.

Tendo em vista essa diferença no tempo de execução das STMs, propõe-se um escalonador STM para arquiteturas *NUMA*, que avalie as ocorrências de *aborts* e o custo de acesso aos dados, para então escalonar as tarefas de forma a obter melhor desempenho.

4. Proposta

Bibliotecas de STM como a *SwisSTM* utilizam mecanismos para prever conflitos em transações, porém, isto acaba por inserir maior custo para execução dos programas. Em [Nicácio et al. 2012] foi proposto um escalonador de transações dinâmico denominado *LUTS* com heurísticas de detecção de conflitos, onde o escalonador de transações evita a ocorrência de *aborts* no decorrer de sua execução.

Em [Dolev et al. 2008], um escalonador denominado *CAR-STM* é apresentado, o qual mantém uma fila para cada *thread*. Em tempo de execução, o escalonador insere as transações abortadas na fila da transação conflitante, assim, reduzindo o número de *aborts* através da serialização destas transações.

Esses trabalhos propõem reduzir o número de *aborts* escolhendo melhores formas de distribuir as transações entre os *cores* disponíveis na arquitetura utilizada, assim, obtendo um melhor desempenho no tempo de execução.

Diante disto, propõe-se a criação de um escalonador baseado nos trabalhos citados acima, onde cada núcleo disponível na arquitetura possuirá uma fila de transações. O escalonador utilizará dos mecanismos de detecção de conflitos e versionamento de dados já existentes na biblioteca para identificar a ocorrência de um *abort*.

O escalonador a ser desenvolvido irá levar em conta não somente a serialização das transações conflitantes, mas também a latência de acesso aos dados conflitantes na memória, por exemplo, com a ocorrência de um conflito a biblioteca realiza um *abort* e passa o controle para o escalonador, este, compara o custo de acesso à memória das transações conflitantes, caso o custo da transação abortada seja maior, o escalonador migra a transação abortada para à mesma fila de execução pertencente a transação conflitante.

Por meio deste mecanismo, haverá a serialização das transações conflitantes sem perder o paralelismo existente na arquitetura, também haverá a redução de tempo no acesso aos dados por meio de uma melhor distribuição de transações, com isto, será reduzido o número de *aborts* e melhorado o tempo de execução de programas que utilizam STM quando aplicado as arquiteturas NUMA.

5. Conclusão

Neste artigo foi apresentado o comportamento das STMs quando executados em arquiteturas NUMA a fim de propor um escalonar *NUMA-Aware*. Para isto, foram estudados trabalhos relacionados a STM, *TinySTM*, arquiteturas NUMA e escalonadores.

A principal contribuição deste trabalho está na proposta de um escalonador que em tempo de execução avalia onde alocar as transações conflitantes, para obter melhor tempo de execução e menor número de conflitos.

Referências

Baldassin, A. J. (2009). Explorando memória transacional em software nos contextos de arquiteturas assimétricas, jogos computacionais e consumo de energia. Dissertação de doutorado, Universidade Estadual de Campinas.

Carissimi, A., Dupros, F., Méhaut, J.-F., and Polanczyk, R. V. (2007). Aspectos de programação paralela em máquinas numa. In *Minicurso do Workshop em Sistemas Computacionais de Alto Desempenho*.

Dice, D., Shalev, O., and Shavit, N. (2006). Transactional locking II. In *DISC 2006*, pages 194–208.

Dolev, S., Hendler, D., and Suissa, A. (2008). Car-stm: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 125–134, New York, NY, USA. ACM.

Felber, P., Fetzer, C., and Riegel, T. (2008). Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246, New York, NY, USA. ACM.

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