Caracterização Inicial do Comportamento de Aplicações Transacionais em Arquiteturas NUMA

André Libório[1], **Alexandro Baldassin**[1]

[1]Universidade Paulista Júlio de Mesquita Filho (UNESP) - Brasil

{andre.lb.ferraz, alexandro.baldassin}@unesp.br

Abstract. *This article seeks to make an initial characterization of transaction memory use in high-performance NUMA systems and, through the application of different mechanisms of dynamic memory allocation and thread mapping, obtain a comparative performance profile. The results of the characterization show that there are benefits in using a single CPU socket and, to better exploit NUMA architectures, some changes are further required in the transactional algorithm.*

Resumo. *Este artigo busca fazer uma caracterização inicial de memória transacional em sistemas NUMA de alto desempenho e, por meio da aplicação de diferentes mecanismos de alocação de memória dinâmica e de mapeamento de threads, obter um perfil de desempenho comparativo. Os resultados mostram que existem benefícios no uso de um único socket de processador e que, para melhor explorar arquiteturas NUMA, algumas mudanças são necessárias no algoritmo transacional.*

1. Introdução

Com o advento do paralelismo em processadores modernos, houve a necessidade de desenvolver métodos eficientes para otimizar software para essas plataformas [Gove 2011]. Ainda que o objetivo seja simples, sua execução não é: uma paralelização adequada de um programa enfrenta diversas dificuldades, como compreender a estrutura de dados em questão e selecionar trechos específicos onde a paralelização possa ser vantajosa, utilizando bibliotecas dedicadas como a API OpenMP [M. Müller 2009] da linguagem C por exemplo. Outro ponto a ser considerado é o uso de regiões críticas, as quais podem ser utilizadas em situações de concorrência nos acessos à variáveis e, portanto, podem influenciar o desempenho da aplicação dependendo de sua implementação. Em vista disso, a memória transacional (*Transactional Memory* – TM) passa a ser uma estratégia viável. Estes e outros pontos devem ser levados em conta ao se cogitar paralelizar um código, uma vez que existem situações nas quais a paralelização pode não ser viável, podendo, talvez contra intuitivamente, gerar uma piora considerável de desempenho [Herlihy and Shavit 2008].

Para solucionar parte das dificuldades com programação paralela elencadas previamente, surge o conceito de TM que já se tornou bastante utilizada no meio acadêmico [Harris et al. 2010]TM utiliza o conceito de *transação* para executar operações de maneira atômica, funcionando como uma alternativa mais atualizada para sistemas paralelos se comparado a mecanismos tradicionais como os *locks* e variáveis condicionais. Uma transação possui as propriedades ACI (Atomicidade, Consistência e Isolamento),

semelhante às transações típicas de Banco de Dados [Gray 1981], mas sem a durabilidade. Assim, o conjunto de instruções que perfaz uma transação ou é todo ele efetivado, ou seja, as alterações parecem acontecer de forma atômica, ou então a transação é cancelada, fazendo com que as alterações sejam desfeitas e a transação reiniciada. Tal tecnologia ainda possui a vantagem de ser facilmente compreendida, ressaltando ainda que sua implementação não requer uma reestruturação completa do código. O mecanismo transacional pode ser implementado em hardware (HTM), software (STM), ou ainda de forma híbrida (HyTM). Para as finalidades deste artigo, consideramos apenas STM.

Este artigo busca por meio das aplicações STAMP (*Stanford Transactional Applications for Multi-Processing*) [Minh et al. 2008], ou seja, benchmarks de aplicações diversas que utilizam memória transacional, verificar o desempenho destas em sistemas NUMA de alto desempenho, comparando diferentes mecanismos para alocação de memória dinâmica e de mapeamento de threads. Em particular, além de mecanismos nativos do Linux como o First Touch (`FT`), são considerados os mecanismos Round Robin (`RR`), Compact (`COMP`), Scatter (`SCAT`) [Kleen 2004].

Os resultados iniciais mostram que, em geral, o mecanismo de mapeamento de threads `COMP` mostra um melhor desempenho em 4 das 7 aplicações analisadas. No entanto, não nota-se muita diferença de desempenho entre os mecanismos de alocação de memória estudados. Isso talvez se deva ao fato da máquina utilizada nos experimentos não apresentar um fator NUMA significativo.

Este trabalho é organizado da seguinte forma. A Seção 2 faz uma contextualização sobre as tecnologias e políticas aqui abordadas, assim como alguns trabalhos relacionados. A Seção 3 apresenta a metodologia utilizada para a caracterização e os resultados experimentais. Por fim, o trabalho é concluído na Seção 4.

2. Contextualização e Trabalhos Relacionados

Memória transacional é um mecanismo relativamente recente para sincronização de programas paralelos [Harris et al. 2010]. A ideia de usar transações é emprestada de Banco de Dados [Gray 1981], garantindo que um bloco de código seja executado de forma atômica, isolada e consistente. Para isso, as instruções de load/store precisam ser instrumentadas. No caso de implementações HTM, isso é feito automaticamente. Porém, as implementações em software precisam interceptar todos os acessos aos dados compartilhados. A grande vantagem de STM é sua flexibilidade, uma vez que diferentes algoritmos podem ser testados.

Ainda que apresente benefícios consideráveis, uma aplicação pode se submeter a uma considerável piora de desempenho se a técnica (TM) não for adequadamente aplicada. Os principais motivos para uma situação como esta seria um mal aproveitamento da concorrência já existente no código e se a granularidade dos locks for convertida de maneira inadequada para TM. Outro ponto a se considerar são as garantias dadas pela implementação TM sobre o progresso das transações e, por fim, se o código em questão possui uma sobrecarga de processamento sequencial. O uso em particular de STM pode gerar uma perda de desempenho significativa [Cascaval et al. 2008], já que as instruções de load/store precisam ser instrumentadas, como dito anteriormente.

Dentre os conceitos aqui abordados, deve-se ainda ressaltar a importância do fator NUMA (*Non-Uniform Memory Access*), o qual pode gerar alterações significativas

no comportamento de programas paralelizados [Lameter 2013]. Tal fator se refere ao aumento de latência de memória gerado pela arquitetura NUMA, ou seja, a situação na qual fisicamente em um ou mais processadores, núcleos específicos possuem canais de memória RAM dedicados, fazendo com que outros núcleos tenham que realizar acessos indiretos a tais endereços de memória.

Não há na literatura muitos trabalhos analisando o desempenho de aplicações transacionais em arquiteturas NUMA. Uma primeira análise sobre o controle de concorrência em TM no contexto de NUMA foi apresentado por Mohamedin et al. [Mohamedin et al. 2018]. Mais recentemente, Pasqualin et al. [Pasqualin et al. 2020a] fizeram uma caracterização do comportamento de algumas aplicações transacionais em arquiteturas NUMA. Com base nessa análise, os autores propuseram uma técnica para escalonamento de threads que potencialmente beneficiaria aplicações transacionais [Pasqualin et al. 2020b]. Adicionalmente ao trabalho de Pasqualin et al., nosso trabalho também investiga o impacto da alocação de memória.

3. Caracterização

Esta seção tem como objetivo realizar uma caracterização inicial das aplicações do pacote STAMP [Minh et al. 2008]. Consideramos tanto o aspecto de alocação de memória como o de mapeamento de threads.

3.1. Metodologia

Além da política padrão do sistema operacional utilizado, representado aqui como First Touch (`FT`), foram utilizadas duas técnicas de alocação de memória (Round Robin (`RR`) e Compact (`COMP RAM`)), e outras duas para mapeamento de threads (Scatter (`SCAT`) e Compact (`COMP TH`)). Para as configurações não especificadas, subentende-se o uso da política padrão do sistema operacional. O `FT` é a técnica padrão utilizada por sistemas operacionais Linux, na qual a página de memória é alocada apenas quando a mesma é referenciada.

Considerando as configurações com mudanças na alocação de memória, o `RR` é um dos algoritmos de escalonamento clássicos aplicados a cada nó, ou seja, subconjunto de núcleos. Neste caso em específico, a memória é alocada de forma intercalada entre as memórias físicas próximas de cada nó. Já o `COMP RAM` procura realizar a alocação de memória de maneira local ao *socket*, privilegiando alocações em pentes de memória mais próximos do núcleo que requisita a alocação.

No caso de políticas voltadas a threads, foi utilizado o `SCAT`, que busca evitar a sobrecarga de nós e, por isso, tem a característica de espalhar as threads sendo utilizadas por todos os nós disponíveis, fazendo com que a comunicação de informações entre as threads seja prejudicada em prol de uma maior capacidade de cache dedicada as threads em execução enquanto o processo não alocar todas às threads do(s) processador(es) em questão. Por fim, o `COMP TH` possui a característica de agrupar as threads sendo executadas na menor quantidade de nós possíveis, possibilitando, uma menor latência entre os dados alocados.

Para realizar os experimentos foi utilizado um sistema equipado com dois processadores Intel Xeon Gold 5220, totalizando 36 núcleos físicos e 72 lógicos, com o sistema operacional CentOS 7.7 com a política energética no modo *performance* e o compilador

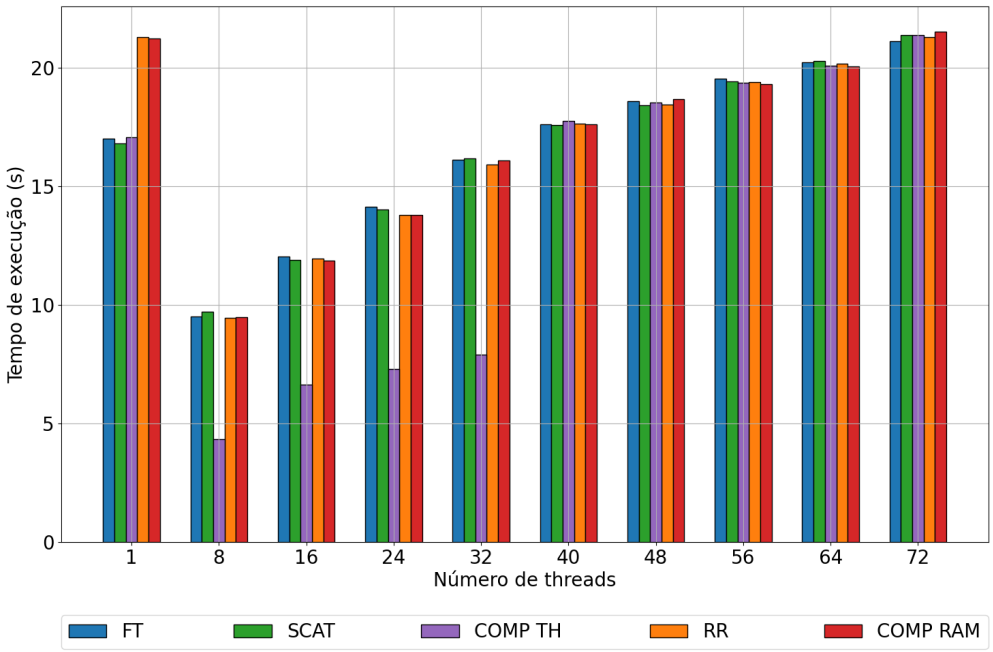


Figura 1: Gráfico representando os testes realizados na aplicação Intruder.

GCC 7.3.1. É válido ressaltar que a variável `numa_balance`, implementada nativamente em sistemas operacionais Linux para acentuar as penalidades atribuídas às arquiteturas NUMA, fora desativada.

Os experimentos foram feitos utilizando aplicações do benchmark STAMP, que consiste em aplicações paralelas, que abrangem áreas distintas da computação e utilizam diversas características para a execução em TM, suportada pela biblioteca transacional TinySTM [Felber et al. 2008] em sua configuração padrão (*encounter-time locking*). Os resultados foram obtidos por meio de 10 execuções individuais de cada aplicação com as diferentes configurações para alocação de memória (`FT`, `RR`, `COMP RAM`) e mapeamento de threads (`FT`, `SCAT`, `COMP TH`). Essas configurações são feitas através da biblioteca `libnuma` [Kleen 2004].

O artigo ainda utiliza o alocador TCMalloc da Google para a linguagem C, que possui uma implementação mais avançada para sistemas multi-núcleos, visto que trabalhos anteriores notaram uma grande diferença de desempenho entre alocadores de memória [Baldassin et al. 2015]. Inicialmente são apresentados os 3 resultados mais significativos, respectivos às aplicações Intruder, SSCA2 e Yada.

3.2. Análise do Comportamento

Analisaremos com maiores detalhes as aplicações com resultados mais destacados, sendo elas: Intruder (Figura 1), o SSCA2 (Figura 2), o Yada (Figura 3). Já as demais aplicações contidas na STAMP não apresentaram comportamentos muito distintos em comparação às demais já citadas. Para fins de exposição apenas, as mesmas são representadas na Figura 4.

Com base nos resultados obtidos, pode-se observar que, nas configurações que impactam a política de threads, sendo essas a `FT`, `SCAT` e `COMP TH`, houve uma melhora

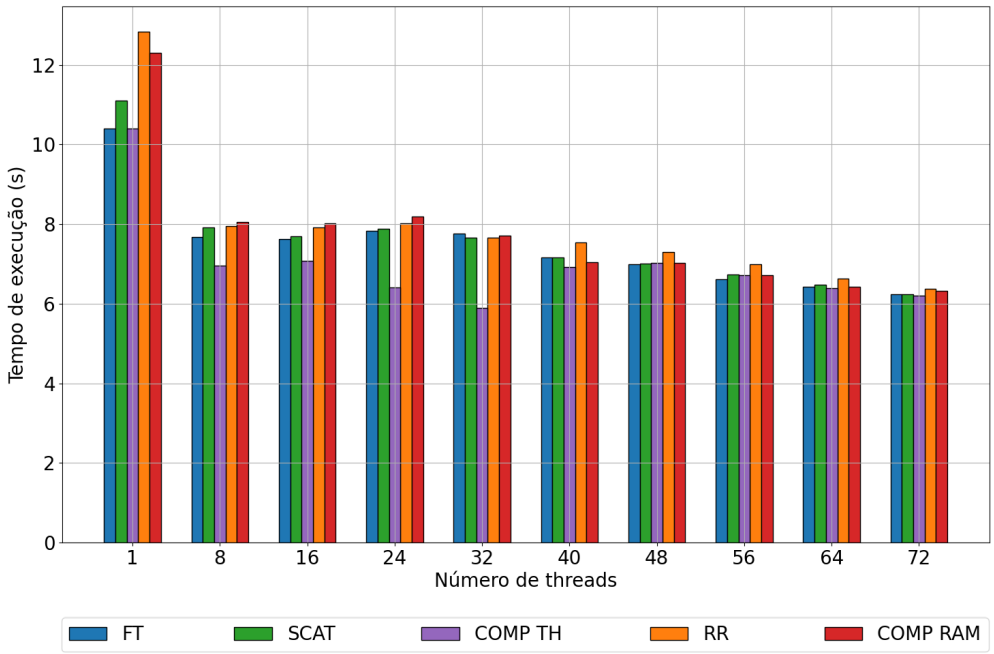


Figura 2: Gráfico representando os testes realizados na aplicação SSCA2.

generalizada no caso da configuração `COMP TH` nas situações com até 32 threads; neste caso em específico, se referindo às threads contidos em apenas um único *socket*. Como a política com a melhora de desempenho mais significativo é a que mantém os acessos mais locais, a latência de acesso à memória se mostra como um fator de grande importância, e deve ser considerado para obter um melhor desempenho.

Já considerando as políticas relacionadas à alocação de memória dinâmica, sendo essas `FT`, ou default, `RR` e `COMP RAM`, nos casos utilizando apenas 1 thread, é possível observar que os testes de `RR` e `COMP RAM` possuem um pior desempenho com relação ao `FT`. Tal diferença pode ser atribuída a um acréscimo de latência, uma vez que o `FT` neste caso realiza a alocação de memória de maneira ótima, ou seja, aloca a memória mais próxima da thread em execução e as demais políticas não se atêm a tal restrição.

Além disso, exclusivamente na aplicação Yada (Figura 3), o comportamento dos testes utilizando `RR` e `COMP RAM` apresentam uma leve vantagem no tempo de execução com 24 e 32 threads, não seguindo a tendência apresentada até então, onde excluindo o `COMP TH`, conforme já enfatizado, os demais apresentam um desempenho bastante similar.

4. Conclusão

Com esta caracterização inicial, é possível observar que as aplicações STAMP ainda são possíveis de otimização, uma vez que o resultado nativo não é o mais eficiente, demonstrando que estudos ainda são necessários para atingir uma certa maturidade da tecnologia e desenvolver seu uso comercial.

De maneira geral, observa-se que neste trabalho, apesar das diferenças apontadas, é possível concluir que mesmo com a análise de resultados da Seção 3.2 as diferenças não

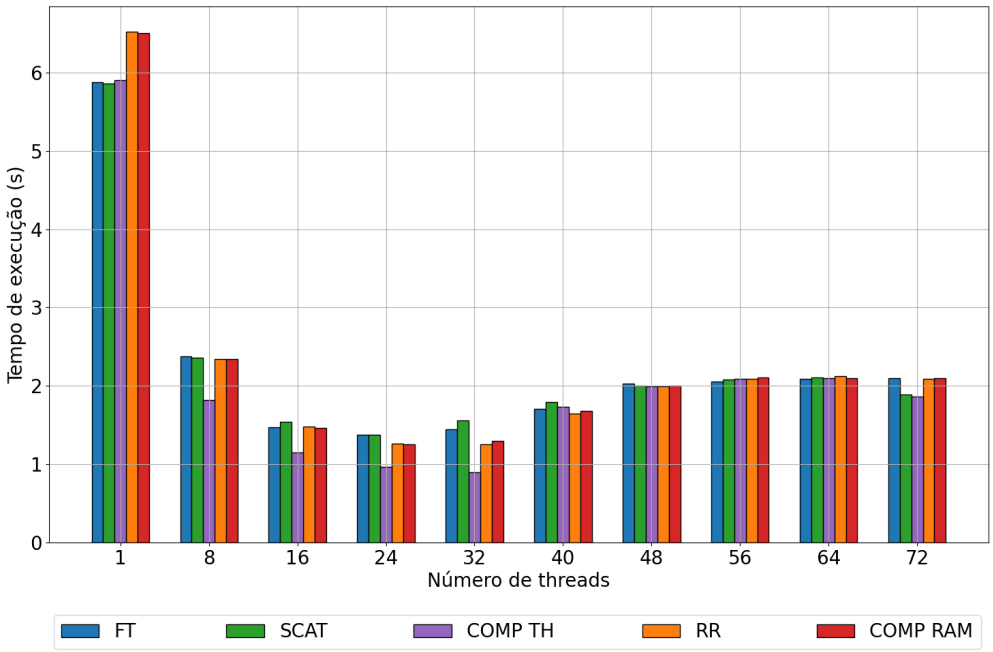


Figura 3: Gráfico representando os testes realizados na aplicação Yada.

são tão significativas quanto o esperado, o que pode ser atribuído ao baixo fator NUMA da máquina em questão. Por isso, mais testes serão necessários para obter uma conclusão mais detalhada, o que será feito em um trabalho futuro com o foco no desenvolvimento de uma técnica que explore a localidade de threads e/ou memória para aumentar o desempenho.

Referências

Baldassin, A., Borin, E., and Araujo, G. (2015). Performance implications of dynamic memory allocators on transactional memory systems. In *Proceedings of the 20th ACM PPoPP*, pages 87–96.

Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., and Chatterjee, S. (2008). Software transactional memory: Why is it only a research toy? the promise of stm may likely be undermined by its overheads and workload applicabilities. *Queue*, 6(5):46–58.

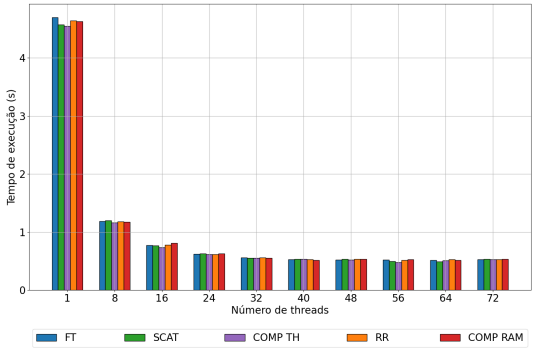
Felber, P., Fetzer, C., and Riegel, T. (2008). Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pages 237–246.

Gove, D. (2011). *Multicore Application Programming*. Pearson Education, Inc.

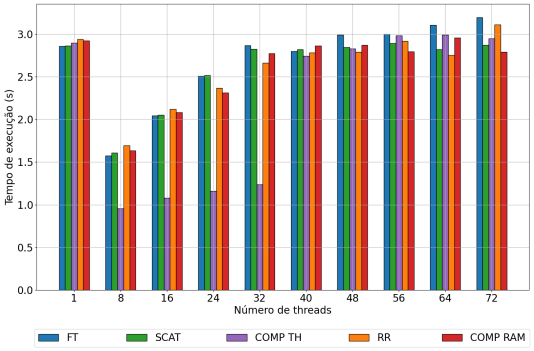
Gray, J. (1981). The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Databases*, pages 144–154.

Harris, T., Larus, J., and Rajwar, R. (2010). *Transactional Memory*. Morgan & Claypool Publishers, 2 edition.

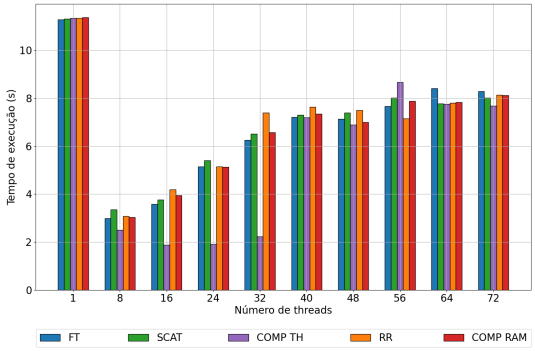
Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.



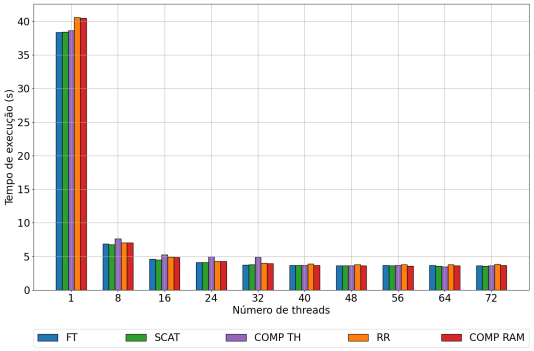
(a) Genome



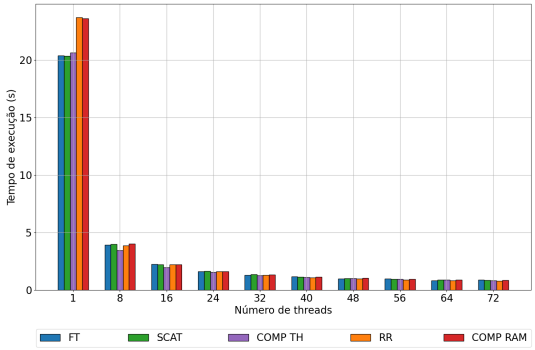
(b) Kmeans-High



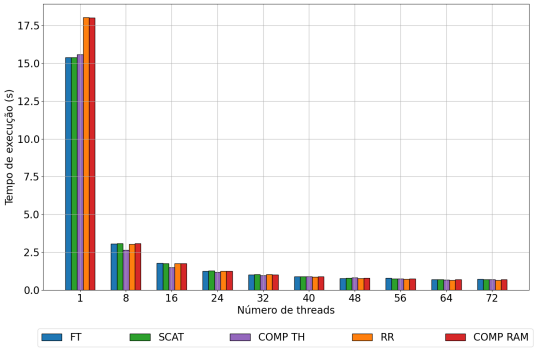
(c) Kmeans-Low



(d) Gráfico representando os testes realizados na aplicação Labyrinth



(e) Vacation-High



(f) Vacation-Low

Figura 4: Resultados com as demais aplicações do pacote STAMP.

Kleen, A. (2004). *A NUMA API for Linux*. SUSE Labs.

Lameter, C. (2013). NUMA (non-uniform memory access): An overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51.

M. Müller, B. Supinski, B. C. (2009). *Evolving OpenMP in an Age of Extreme Parallelism*. Springer.

Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K. (2008). STAMP: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46.

Mohamedin, M., Peluso, S., Kishi, M. J., Hassan, A., and Palmieri, R. (2018). Nemo: Numa-aware concurrency control for scalable transactional memory. In *Proceedings of the 47th ICPP*.

Pasqualin, D. P., Diener, M., Du Bois, A. R., and Pilla, M. L. (2020a). Characterizing the sharing behavior of applications using software transactional memory. In Wolf, F. and Gao, W., editors, *Benchmarking, Measuring, and Optimizing*, pages 3–21.

Pasqualin, D. P., Diener, M., Du Bois, A. R., and Pilla, M. L. (2020b). Online sharing-aware thread mapping in software transactional memory. In *2020 IEEE 32nd SBAC-PAD*, pages 35–42.