# Reavaliando o Conjunto de Aplicações STAMP em um Novo Hardware Transacional

**João P. L. de Carvalho[1,2], Rafael Pizzirani Murari[1], Alexandro Baldassin[1]**

[1] Instituto de Geociências e Ciências Exatas – Universidade Estadual Paulista (UNESP)
Av. 24A, 1515, Bela Vista – Rio Claro – SP – Brazil

[2]Bolsista FAPESP, Processo 2014/00534-8

{joaopaulo, alex}@rc.unesp.br, rafaelpmurari@gmail.com

***Abstract.*** *Over the past four years, IBM® and Intel® have made available processors with transactional memory support. The majority of the evaluation conducted with these processors used mainly STAMP applications and considered mostly the abort causes for the applications as a whole. This work presents a per-transaction analysis of STAMP and contrasts different performance metrics to determine the fundamental reasons of Haswell's low performance for some applications. The main results show that a single transaction dominates execution time, has low commit rates because either it exceeds the restricted hardware capacity or it has too many conflicts when running in hardware.*

***Resumo.*** *Nos últimos quatro anos, IBM® e Intel® disponibilizaram processadores com suporte para memória transacional. A maioria dos trabalhos avaliaram esses processadores usando as aplicações STAMP e consideraram apenas as causas de cancelamentos das aplicações como um todo. Neste trabalho, apresenta-se uma análise por transação do STAMP e contrasta-se diferentes métricas de desempenho para determinar os motivos fundamentais pelo baixo desempenho do Haswell™ em algumas aplicações. Em resumo, os resultados mostram que uma transação domina o tempo de execução e tem poucas efetivações porque excede a capacidade restrita do processador, ou gera muitos conflitos quando executada em hardware.*

## 1. Introdução

As mudanças nos microprocessadores dos últimos 40 anos foram transparentes à pilha de *software*, permitindo que, mesmo sem mudanças no modelo de programação (sequencial), essa continuasse alcançando maior desempenho a cada nova geração de microprocessadores. Isso só foi possível graças ao avanço tecnológico que aumentou a velocidade de operação e reduziu o tamanho dos transistores. Um número maior de transistores em uma mesma área tornou possível implementar unidades mais complexas de processamento dotadas de mecanismos que otimizam a execução, como por exemplo a execução de múltiplas instruções independentes em paralelo (ILP, *Instruction Level Parallelism*). Todavia, em um programa típico, o número de instruções independentes é geralmente reduzido pela forma como o mesmo foi escrito, reduzindo as oportunidades de exploração do ILP [Wall 1991]. Aumentar a janela de instruções pode amenizar esse efeito, no entanto esta solução esbarra na barreira de memória (*Memory Wall*) [Wulf and McKee 1995]. Além disso, a barreira de energia (*Power Wall*) [Barroso and Holzle 2007] limitou o aumento da frequência de operação dos processadores, o número de transistores que podem

permanecer ligados simultaneamente e a velocidade de transferência de dados entre a memória e o processador. Em resumo, os processadores com fluxo único de execução não serão capazes de atender a crescente demanda por desempenho. As microarquiteturas com múltiplas unidades de execução (*multicore*) são a alternativa encontrada pela indústria às abordagens baseadas em fluxo único.

A utilização de arquiteturas *multicore* mostrou-se desafiadora, uma vez que encontrar paralelismo em algumas aplicações é difícil e escrever programas paralelos requer mudanças na forma sequencial como pensamos sobre *software*. De fato, utilizar diretamente primitivas como travas (*locks*) e semáforos para sincronizar os acessos às variáveis compartilhadas é uma tarefa não apenas complexa como também passível de erros de difícil detecção. Claramente, os modelos e ferramentas atualmente disponíveis são insuficientes para exploração significativa de todos os núcleos de execução, fazendo-se necessárias novas abordagens [Sutter and Larus 2005]. Um novo modelo de programação concorrente, conhecido como Memória Transacional (TM, *Transactional Memory*) [Harris et al. 2010], abstrai o controle de concorrência em sistemas de múltiplas linhas de execução (*threads*) ao usar o conceito de *transação* da área de Banco de Dados.

### 1.1. Motivação

Nos últimos anos o interesse na área de TM, antes restrito apenas à academia, vem ganhando a atenção da indústria de microprocessadores. Realmente, nos últimos quatro anos duas grandes fabricantes lançaram no mercado processadores com suporte em *hardware* para execução especulativa baseada no modelo transacional. Como exemplo podemos citar o *Blue Gene/Q*™ e o *POWER8*™ [Wang et al. 2012, Le et al. 2015], ambos processadores destinados aos ambientes de servidores e computação de alto desempenho com suporte em *hardware* para Memória Transacional (HTM), lançados respectivamente em 2012 e 2014 pela IBM®. Outro exemplo de máquina com suporte a HTM é o processador *Haswell*™ lançado pela Intel® em Junho de 2013. Diferentemente dos processadores lançados pela IBM®, o processador da Intel® pode ser empregado em uma gama maior de dispositivos, incluindo dispositivos móveis como *smartphones* e *tablets*.

Uma grande barreira encontrada recentemente para a avaliação do desempenho e programabilidade do modelo transacional está no número reduzido de aplicações escritas utilizando esse modelo. O conjunto de aplicações (*benchmark*) mais utilizado atualmente, o STAMP [Minh et al. 2008] (*Stanford Transactional Applications for Multi-Processing*), é de 2008 e não vem recebendo atualizações nos últimos anos. Resultados preliminares com esse *benchmark* usando o suporte de HTM dos processadores novos mostram desempenho tímido para a grande maioria das aplicações [Yoo et al. 2013]. Em geral, essa análise é feita de forma superficial, apontando recursos limitados do hardware (e.g., tamanho da cache) como a principal razão para a perda de desempenho. No entanto, não há na literatura uma análise sobre as características do *benchmark* STAMP que estão impossibilitando o bom desempenho do suporte transacional em hardware.

Como exemplo, a Figura 1 mostra o ganho em desempenho (*speedup*) das 8 aplicações do pacote STAMP. Cada aplicação é analisada com 4 configurações de *thread* e 3 abordagens para paralelização: (i) biblioteca transacional em *software* (`NOrec`); (ii) suporte em *hardware* da Intel (`RTM`); e (iii) uma trava global (`LOCK`). Como mostra a Figura 1, algumas aplicações do pacote STAMP não são aceleradas quando executadas utilizando o suporte disponível no processador *Haswell*™ (RTM), mais claramente as

aplicações `Labyrinth` e `Yada`. Saber exatamente quais os fatores que estão limitando esse desempenho é importante, mas até o momento não é encontrado na literatura uma análise mais profunda desses fatores.

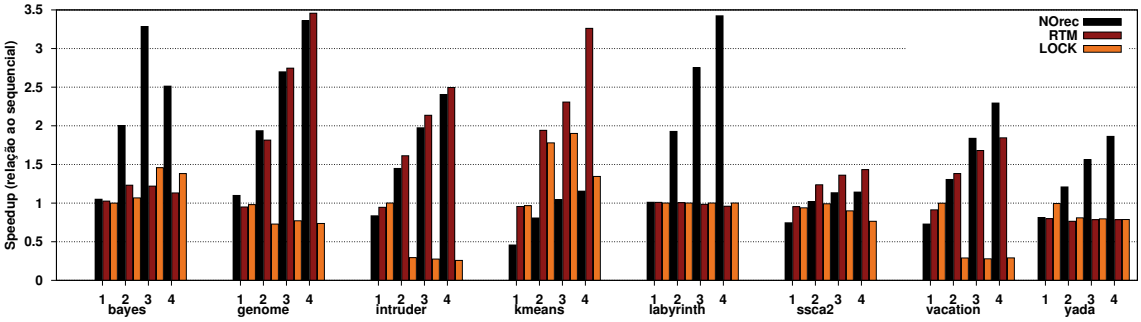


**Figura 1. Speedup das aplicações STAMP em cada abordagem de paralelização**

## 1.2. Contribuições

Objetivando identificar as razões que impedem o suporte do *Haswell*™ de acelerar algumas aplicações do STAMP, neste trabalho será apresentada uma caracterização inédita na literatura. Ao invés de considerar monoliticamente as aplicações e analisar apenas as causas de cancelamento das transações [Diegues et al. 2014], nossa abordagem caracteriza as aplicações considerando cada uma de suas transações com métricas variadas de desempenho. Em resumo, as duas principais contribuições deste trabalho são:

- Uma caracterização por transação das aplicações do STAMP considerando as seguintes métricas de desempenho: taxa de efetivação, tamanho dos conjuntos de leitura e escrita, além do número de ciclos;
- Análise e projeção do desempenho obtido nos seguintes processadores com suporte transacional, baseando-se em suas características e no desempenho observado no *Haswell*™: *Blue Gene/Q*™, *POWER8*™ e *zEC12*™.

Em geral, nossos resultados apontam que as aplicações do pacote STAMP que perdem em desempenho possuem uma única transação que domina o tempo de execução e tem poucas efetivações porque excede a capacidade restrita do processador, ou então gera muitos conflitos quando executada em *hardware*.

O restante do trabalho foi organizado da seguinte maneira. A Seção 2 apresenta um resumo dos conceitos, *benchmarks* e trabalhos relacionados. A Seção 3 descreve a metodologia de avaliação experimental utilizada, enquanto a Seção 4 mostra uma caracterização em nível de transação do pacote STAMP. O trabalho é concluído na Seção 5, onde destacamos os principais pontos da nossa análise.

## 2. Conceitos e Trabalhos Relacionados

Memória transacional é um modelo de computação paralela que utiliza o conceito de transação para implementar operações atômicas e abstrair os acessos à memória compartilhada em uma seção crítica. O conceito de transação foi primeiramente introduzido por Eswaran et al. [Eswaran et al. 1976]. Lomet et al. [Lomet 1977] foi pioneiro na utilização de transações para facilitar a codificação e sincronização de processos. No modelo de TM, assim como na área de banco de dados, as transações possuem os seguintes atributos: atomicidade, consistência e isolamento. Uma operação atômica é executada sem que seus

estados intermediários sejam observados pelo sistema. Ou seja, ou a operação é bem sucedida, nesse caso as modificações são ditas efetivadas, ou é reiniciada, suas modificações são descartadas, e para o sistema é como se nada tivesse acontecido (uma propriedade conhecida comumente como *tudo ou nada*).

Sistemas transacionais devem implementar dois mecanismos principais: *versionamento* de dados e *detecção de conflitos*. O versionamento de dados é o mecanismo responsável por gerenciar as duas versões dos dados acessados pela transação, ditas corrente e especulativa. O versionamento pode ser *diferido* ou *direto*. No versionamento *diferido* a versão especulativa é armazenada localmente à transação e a corrente é mantida com visibilidade global. Por sua vez, no versionamento *direto* a versão corrente é mantida local à transação, enquanto a especulativa é armazenada na região compartilhada. Manter essas duas versões se faz necessário pela natureza otimista da execução transacional e, em caso de cancelamento, possibilitar que as modificações sejam desfeitas restaurando os dados para sua versão corrente.

O mecanismo de detecção de conflitos determina quando um conflito é detectado. Há um conflito entre duas transações quando há uma intersecção entre seus conjuntos de leitura e escrita (posições de memória lidas e escritas). Se o conflito é detectado no momento que as operações de leitura ou escrita são efetuadas, dizemos que o sistema possui detecção *antecipada*. Se a detecção é diferida até o momento de efetivação da transação, dizemos que a detecção é *tardia*. O *gerenciador de contenção* é o módulo de um sistema transacional que decide qual ação deve ser tomada quando um conflito é detectado.

Nesse trabalho é utilizada uma biblioteca transacional em software (STM), a NOrec [Dalessandro et al. 2010]. A STM NOrec utiliza validação por valor para detecção de conflitos, que ocorre de forma antecipada, e adota uma estratégia diferida de versionamento para as escritas. Em caso de cancelamento a biblioteca simplesmente reinicia a transação sem espera, sendo uma política de gerenciamento de contenção denominada *suicida*.

### 2.1. Suporte em Hardware para TM

Os processadores disponíveis hoje com suporte para TM implementam execução baseada em transações de "melhor esforço", já que o *hardware* não garante a efetivação da transação. Uma exceção é o *Blue Gene/Q*™, que garante a efetivação da transação utilizando um mecanismo em *hardware* que serializa as transações. O *zEC12*™ também possui um modo de execução transacional restrito: transações com até 32 instruções e que acessam no máximo 256KB são garantidas de eventualmente efetivar. Os processadores *POWER8*™, *Haswell*™ e *zEC12*™ possuem instruções para iniciar, finalizar e cancelar transações. No *Blue Gene/Q*™, o mecanismo de controle das transações é transparente ao programador já que o código das transações é apenas anotado utilizando *pragmas*. Por essa razão, no *Blue Gene/Q*™ o programador não tem controle do fluxo de execução quando a transação é cancelada. Já os demais processadores necessitam de um caminho alternativo (*fallback*) que é executado após o cancelamento da transação, sendo este responsável por garantir o progresso do sistema.

Os quatro processadores discutidos neste trabalho utilizam o protocolo de coerência de *cache* na detecção de conflitos, que acontece de forma antecipada. No

*Blue Gene/Q*™ a detecção de conflitos também pode ser configurada para funcionar de maneira tardia. O versionamento é diferido (*lazy*), os conjuntos de escrita e leitura são armazenados nas *caches*, ou em estruturas similares (*write buffers*). As escritas especulativas são privados aos *cores*, dessa forma as modificações são invisíveis às demais transações. Como pode-se observar, as implementações são bastante similares, diferenciando-se notadamente em dois aspectos: (i) granularidade da detecção de conflitos e (ii) capacidade de leitura e escrita transacionais. Ambos aspectos serão melhor detalhados na Subseção 4.2, quando é feita uma projeção dos resultados obtidos no *Haswell*™ para os demais processadores.

Os experimentos neste trabalho foram realizados apenas no processador *Haswell*™. Por ser um processador convencional é mais acessível que os demais, destinados a máquinas servidoras. No *Haswell*™ o suporte transacional faz parte da extensão TSX (*Transactional Synchronization Extension*), e pode ser utilizada pelas instruções RTM (*Reduced Hardware Transactions*) ou pelo suporte para omissão de travas em *hardware* (HLE, *Hardware Lock Elision*). Na implementação utilizada cada transação pode reiniciar até 5 vezes e, caso a transação não consiga ser efetivada ou em caso de conflito não transiente (ex.: *overflow* da *cache*), o progresso é garantido pelo código de *fallback* que utiliza uma trava global para serializar a transação. O efeito de "manada" (*lemming effect*) [Dice et al. 2009] é solucionado atrasando o início das transações canceladas pelo fato da trava global estar bloqueada. Apesar de simples, essa é a implementação mais eficaz disponível na literatura.

### 2.2. Benchmarks

Os primeiros sistemas de TM foram avaliados com *microbenchmarks*, compostos por pequenas aplicações que estressam operações em estruturas de dados, ou aplicações paralelizadas por especialistas utilizando primitivas de sincronização, sendo minuciosamente adaptadas para usar sistemas de TM (SPEC, *Standard Performance Evaluation Corporation*, e SPLASH-2 [Woo et al. 1995]). O primeiro conjunto de *benchmarks* com aplicações reais paralelizadas já pensando no modelo de programação de TM foi o STAMP [Minh et al. 2008]. As regiões críticas foram criadas e anotadas pensando nas habilidades de um programador pouco experiente na área de paralelismo. Outros *benchmarks* foram propostos para avaliar sistemas de TM, como por exemplo os geradores de aplicações sintéticas Eigenbench [Hong et al. 2010] e CLOMP-TM [Schindewolf et al. 2012]. Todavia, a grande maioria dos trabalhos adota as aplicações do STAMP. O uso quase que exclusivo das aplicações STAMP revela a carência de aplicações para avaliar os novos sistemas de TM.

As 8 aplicações do STAMP foram selecionadas por representarem diversos domínios de aplicação e não serem facilmente paralelizáveis. Também foram escolhidas pela diversidade de características que diretamente influenciam no desempenho de sistemas de TM: transações longas (*Labyrinth* e *Yada*) e curtas (*Kmeans e SSCA2*), que acessam poucos (*Genome*, *Intruder* e *Bayes*) ou muitos dados (*Vacation*). Apesar da diversidade de características, as aplicações foram paralelizadas seguindo um padrão comum. Basicamente, cada aplicação remove uma tarefa de uma estrutura compartilhada, processa a tarefa em paralelo e, possivelmente, adiciona na estrutura compartilhada as sub-tarefas resultantes. Por exemplo, na aplicação `Yada` as tarefas são triângulos que devem ser ajustados para satisfazer as propriedades de Delaunay [Ruppert 1995]. Na

aplicação `Labyrinth` as tarefas são pares de pontos que devem ser conectados por caminhos que não se interceptam [Lee 1961].

### 2.3. Trabalhos Relacionados

O primeiro trabalho a caracterizar o desempenho do suporte transacional do *Haswell*™ foi o de Yoo et al. [Yoo et al. 2013]. Os resultados para as aplicações do STAMP são similares aos nossos, como mostrados na Figura 1. Os autores limitam-se a apresentar os resultados do tempo de execução normalizados em relação à versão sequencial, além de uma tabela com taxas de cancelamentos para cada aplicação. No entanto, nenhuma análise é feita sobre as condições que causam os cancelamentos e os motivos para perda de desempenho em algumas das aplicações, notadamente o `Labyrinth` e o `Yada`.

Outros trabalhos [de Carvalho et al. 2013, Diegues et al. 2014, Goel et al. 2014] também analisam o impacto no consumo de energia do pacote STAMP usando o *Haswell*™. Em particular, o trabalho de Goel et al. [Goel et al. 2014] realiza uma caracterização que explora os limites do hardware para características como tamanhos ótimos para transações e taxa de contenção. Alguns desses trabalhos [Diegues et al. 2014, Goel et al. 2014] chegam a apresentar as causas dos cancelamentos por aplicação do STAMP usando os registradores de desempenho, mas nenhuma análise em nível de transação é realizada.

Mais recentemente, Nakaike et al. [Nakaike et al. 2015] fazem uma análise do suporte transacional dos quatro sistemas comerciais existentes: IBM Blue Gene/Q, IBM *z*Enterprise, IBM POWER8 e Intel *Haswell*™. Como nos trabalhos anteriores, o pacote STAMP é usado e apenas uma análise em nível de aplicação é conduzida. É interessante notar que muitos desses trabalhos chegam a observar uma limitação das aplicações do pacote STAMP, propondo inclusive mudanças nas estruturas de dados [Nakaike et al. 2015] ou na forma que o alocador de memória trabalha [Goel et al. 2014].

## 3. Metodologia de Avaliação

Com o objetivo de identificar o que impede o suporte em *hardware* de acelerar algumas das aplicações do STAMP, o código das mesmas foram instrumentados para extrair algumas métricas de desempenho para cada uma de suas transações. Para isso uma biblioteca que programa os contadores de desempenho da PMU (*Performance Monitoring Unit*) foi desenvolvida. Disponível nos processadores Intel® a partir da linha *Pentium*™, a PMU é composta de registradores específicos de modelo que permitem extrair informações de desempenho que auxiliam o programador e o compilador a produzirem programas mais eficientes. Além de monitoramento de desempenho, os MSRs (*Model Specific Registers*) podem ser utilizados na depuração de programas e configuração da CPU.

Neste trabalho foram utilizados os dois tipos de contadores da PMU: os programáveis e os fixos. Contadores fixos são reservados aos eventos arquiteturais, como número de instruções e ciclos, os quais o fabricante garante que estarão presentes nos novos modelos do processador. Por sua vez, os contadores programáveis (não-arquiteturais) são utilizados para os demais eventos da CPU e para os contadores relacionados às extensões experimentais introduzidas em um modelo específico, como a extensão transacional do *Haswell*™ (TSX). O número de contadores e suas respectivas larguras são específicos de modelo. No *Haswell*™ todos os contadores tem largura igual a 48bits, sendo que existem três contadores fixos e quatro contadores programáveis por *thread*.

**Tabela 1. Eventos da PMU utilizados**

| | | | |
|---|---|---|---|
| Ciclos e Instruções | | `INST_RETIRED.ANY` | Total de instruções executadas |
| | | `CPU_CLK_UNHALTED.THREAD` | Total de ciclos transcorridos |
| Início e Término da Transação | RTM | `RTM_RETIRED.START` | Total de transações iniciadas |
| | | `RTM_RETIRED.COMMIT` | Total de transações finalizadas |

Além dos eventos brevemente descritos na Tabela 1, as aplicações foram instrumentadas também para coletar o tamanho do conjunto de leitura e escrita de cada transação quando executadas com o STM NOrec. O tamanho desses conjuntos foi obtido pelo número de elementos da estrutura de dados utilizada para implementá-los. O conjunto de leitura foi implementado como uma lista onde são realizadas apenas inserções, ou seja, múltiplas leituras de um mesmo endereço são armazenadas como elementos diferentes [Dalessandro et al. 2010]. Dessa forma, o tamanho da lista representa o número total de leituras da transação. Uma tabela *hash* foi utilizada para implementar o conjunto de escrita, e assim o tamanho da tabela representa o número de endereços acessados e é um limite inferior do número de escritas da transação. O objetivo de medir o tamanho destes conjuntos é estimar o tamanho das transações com relação ao acesso à memória e seu possível impacto na *cache*, utilizada para armazenar os acessos especulativos das transações em *hardware*.

**Tabela 2. Características do Sistema Empregado nos Experimentos**

| | | |
|---|---|---|
| Processador | CPU<br>Tecnologia<br>Hyperthreading | Intel® Core™ i7-4770 3.4GHz<br>22nm<br>yes |
| Caches | L1 | 64Kb (32Kb L1d + 32Kb L1i), 8-way set associative, 64bytes cache line |
| | L2 | 256Kb, 8-way set associative, não-inclusiva |
| | L3 | 8Mb, 16-way set associative, inclusiva e compartilhada por todos os *cores* |
| Memória | 16Gb (8Gb x 2) DDR3 1333MHz | |
| Discos | SSD | 120Gb SATA-3 6Gb/s |
| | HDD | 500Gb SATA-2 7200RPM 3Gb/s, 64Mb cache |

As aplicações do STAMP foram executadas em um processador Intel® Core™ i7-4770, com 4 *cores* e 2 *threads* por *core*, utilizando o conjunto de entrada recomendado para sistemas reais (vide Tabela IV em [Minh et al. 2008]). As aplicações `Kmeans` e `Vacation` foram executadas com a versão de entrada que gera alta contenção. O tempo de execução, valores dos contadores e tamanho do *read/write-set* apresentados na Seção 4 são valores médios de 20 execuções. O código da biblioteca RSTM [Marathe et al. 2006] na versão 7.0 (Julho/2011) foi utilizado para executar as aplicações utilizando o STM NOrec. O compilador GCC (GNU C Compiler) versão 4.8.3 (Abril/2014), que possui funções intrínsecas para as instruções RTM e macros para as anotações HLE, foi utilizado para compilar tanto as aplicações quanto a biblioteca transacional em *software*. O nível de otimização 3 (`-O3`) foi habilitado para compilar a biblioteca em *software* e as aplicações. Os experimentos foram conduzidos em uma instalação Linux típica, CentOS 6.5 *base server* e kernel versão 3.10. A Tabela 2 apresenta maiores detalhes do processador, como níveis e tamanhos das *caches* e frequência máxima de operação.

## 4. Resultados Experimentais

Nesta seção será apresentada a análise da caracterização de cada transação do pacote STAMP (4.1) e, baseado nas características de cada sistema, uma projeção do resultado obtido em alguns HTMs disponíveis hoje (4.2).

### 4.1. Análise dos Resultados

Nossa abordagem para análise dos resultados é averiguar o que acontece com as aplicações do pacote STAMP no nível de transação. Para isso, conduzimos experimentos para colher dados de execução de cada transação, processo ainda não realizado na literatura. A ideia é averiguar o que acontece com as transações da aplicação, colhendo informações que possam ser úteis para otimização da execução no hardware transacional.

**Tabela 3. Porcentagem de ciclos e taxa de finalização por transação (RTM)**

| Applicação | Total de Ciclos (%) | | | | | Taxa de Commits (%) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | t1 | t2 | t3 | t4 | t5 | t1 | t2 | t3 | t4 | t5 |
| `Genome` | 78,84 | 0,09 | 20,92 | 0,09 | 0,07 | 91,10 | 93,32 | 85,76 | 94,88 | 99,62 |
| `Intruder` | 5,25 | 91,59 | 3,15 | – | – | 94,32 | 75,18 | 95,43 | – | – |
| `Kmeans` | 81,40 | 18,59 | 0,00 | – | – | 94,73 | 98,96 | 99,42 | – | – |
| `Bayes` | 2,33 | 2,38 | 41,08 | 19,63 | 25,78 | 17,91 | 6,20 | 0,16 | 7,35 | 5,24 |
| `Labyrinth` | 7,79 | 92,21 | 0,00 | – | – | 41,96 | 0,00 | 70,83 | – | – |
| `SSCA2` | 0,00 | 0,00 | 100,0 | – | – | 100,0 | 87,50 | 99,91 | – | – |
| `Vacation` | 89,69 | 7,56 | 2,75 | – | – | 26,08 | 14,56 | 44,05 | – | – |
| `Yada` | 4,07 | 2,88 | 87,56 | 2,41 | 3,07 | 48,71 | 72,38 | 1,64 | 70,30 | 58,02 |

Primeiramente, medimos a porcentagem do tempo (ciclos) gasto em cada transação, além de sua taxa de efetivação (*commit*). Esses dados estão apresentados na Tabela 3. Note que mostramos apenas as 5 transações mais importantes, sendo que apenas `Yada` (6) e `Bayes` (15) apresentam um número maior. Colocamos em destaque, para cada aplicação, a transação que gastou mais ciclos de processamento e a respectiva taxa de efetivação [1]. De imediato é possível observar um padrão: a presença de uma transação que domina o tempo de execução (esse padrão também se repete nas execuções com a biblioteca em *software* NOrec). Fazendo uma correlação desses dados com o desempenho (Figura 1), notamos que as aplicações com menor desempenho, `Labyrinth`, `Bayes` e `Yada`, possuem uma taxa de efetivação extremamente baixa, 0%, 0,15% e 1,64% respectivamente, como aponta a tabela. Por outro lado, as aplicações que obtiveram desempenho superior com o uso do suporte transacional, notadamente `Genome`, `Kmeans` e `SSCA2`, têm na maioria das vezes sua transação de maior tempo sempre efetivando (91,10%, 94,73%, 99,91%, respectivamente).

Para explicar porque as transações das aplicações `Bayes`, `Labyrinth` e `Yada` têm um alto nível de falhas, executamos os experimentos colhendo informações dos registradores do *Haswell*™ que determinam alguns tipos de cancelamentos. Infelizmente, estes tipos não são muito abrangentes, sendo classificados como: (i) conflito por dados

[1] É importante observar que os valores mostrados são médias aproximadas. Desta forma, o valor $0,00$ não necessariamente indica um zero absoluto, mas sim que a média calculada foi extremamente baixa e se aproxima do zero.

e/ou capacidade, (ii) conflitos por capacidade do conjunto de escrita, (iii) cancelamentos explícitos, (iv) instruções não suportadas, e (v) nenhuma das causas anteriores. Como pode-se ver, não é possível diferenciar entre falhas por capacidade de leitura e conflito de dados. Muitas vezes as falhas podem ser espúrias, o que complica ainda mais a análise. Com o objetivo de caracterizar melhor as diferentes causas das falhas, foram coletados dados com os tamanhos dos conjuntos de leitura e escrita usando a biblioteca de STM `NOrec`, como mostra a Tabela 4.

**Tabela 4. Tamanho médio dos conjuntos de leitura e escrita por transação (NOrec)**

| Applicação | Conjunto de Leitura | | | | | Conjunto de Escrita | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | t1 | t2 | t3 | t4 | t5 | t1 | t2 | t3 | t4 | t5 |
| `Genome` | 60,03 | 1,57 | 80,09 | 4,00 | 4,06 | 0,02 | 1,00 | 1,99 | 2,00 | 2,92 |
| `Intruder` | 4,99 | 80,52 | 3,07 | – | – | 1,00 | 2,63 | 0,03 | – | – |
| `Kmeans` | 32,67 | 1,00 | 2,00 | – | – | 16,84 | 1,00 | 1,00 | – | – |
| `Bayes` | 74,85 | 12,59 | 63,10 | 8,86 | 90,77 | 6,28 | 1,00 | 0,00 | 0,00 | 4,45 |
| `Labyrinth` | 7,96 | 343,26 | 4,99 | – | – | 0,99 | 343.19 | 2,00 | – | – |
| `SSCA2` | 1,00 | 1,00 | 1,00 | – | – | 1,00 | 1,00 | 2,00 | – | – |
| `Vacation` | 415,80 | 382,63 | 126,03 | – | – | 6,72 | 15,35 | 6,49 | – | – |
| `Yada` | 7,32 | 1,00 | 293,97 | 1,00 | 3,69 | 2,09 | 0,00 | 48,93 | 1,00 | 1,06 |

Como discutido na Seção 3, o tamanho médio do conjunto de escrita representa a média do total de escritas. O tamanho do conjunto de escrita é um limite inferior do total de escritas pela forma como o mesmo foi implementado. As colunas em cinza na Tabela 4 evidenciam os valores para as transações que dominam o tempo de execução. Pode-se observar que nas aplicações `Labyrinth` e `Yada`, que não foram aceleradas com o suporte do *Haswell*™, o número de escritas e leituras é cerca de 150 a 300 vezes maior que nas aplicações `Kmeans` e `SSCA2`, onde o RTM melhor desempenha. Pelo fato da aplicação `Vacation` possuir um número maior de leituras que as aplicações `Labyrinth` e `Yada`, e ainda assim ter apresentado aceleração, aponta que cancelamento por capacidade não parece ser a razão que impede a aceleração de tais aplicações. De fato, constatamos que a transação que domina o tempo de execução da aplicação `Yada` tem mais de 70% dos cancelamentos causados por conflito de dados.

Na aplicação `Labyrinth`, o grande número de leituras e escritas causam cancelamentos por conflito e capacidade que totalizam mais de 60% dos cancelamentos. Outra razão de cancelamento na aplicação `Labyrinth` é o longo tempo em transação, em média mais de 100M de ciclos na transação dominante. Este resultado vai ao encontro dos publicados em [Goel et al. 2014], no qual constatou-se que transações com mais de 10M ciclos sempre são canceladas, e justificam as zero efetivações da transação dominante (vide Tabela 3). Em resumo, o baixo desempenho obtido utilizando-se o suporte em *hardware* deve-se à presença de transações que: (i) causam *overflow* na *cache* L1, (ii) aumentam as chances de conflito de dados e (iii) excedem o *quantum* atribuído ao processo pelo sistema operacional. Em qualquer um desses casos a efetivação da transação certamente não ocorrerá.

Essa caracterização do STAMP também nos permite fazer mais algumas

observações. Por exemplo, o número de transações estáticas é baixo, com a maioria das aplicações contendo apenas 3. Como possível consequência, a maior parte do tempo fica concentrada em apenas uma transação. Os dados também mostram que nesse caso valeria a pena averiguar a possibilidade de dividir uma transação dominante em outras, com o objetivo de aliviar o uso dos recursos de hardware. De fato, abordagens recentes de memória transacional híbrida começaram a explorar essa direção [Xiang and Scott 2015, Matveev and Shavit 2015].

### 4.2. Projeção dos Resultados em Outros HTMs

Na Subseção 4.1 pode-se concluir que os conflitos, sejam eles estruturais (*overflow* dos *buffers* especulativos) ou de dados (reais ou causados pela organização da *cache*), e as longas transações são os fatores que limitam o desempenho do suporte presente no *Haswell*™. Como os conflitos estão diretamente associados às características da implementação, podemos inferir com base nos resultados anteriores o comportamento que seria observado nos demais HTMs. Na Tabela 5 temos, para cada HTM, a granularidade da detecção de conflitos, tamanho da linha de *cache* do processador, e a capacidade dos *buffers* especulativos que armazenam os conjuntos de leitura e escrita das transações. Os dados referentes à capacidade dos *buffers* foram obtidos por Nakaike et al. [Nakaike et al. 2015].

**Tabela 5. Características dos HTMs disponíveis hoje**

| Característica | HTMs | | | |
|---|---|---|---|---|
| | *Blue Gene/Q*™ | *zEC12*™ | *POWER8*™ | *Haswell*™ |
| Detecção de Conflito | 8-128bytes | 256bytes | 128bytes | 64bytes |
| Cap. de Leitura (*core*) | 1.5MB | 1MB | 8KB | 4MB |
| Cap. de Escrita (*core*) | 1.5MB | 8KB | 8KB | 22KB |
| *Cache* L1 | 16KB, 8-way | 96KB, 6-way | 64KB | 32KB, 8-way |
| *Cache* L2 | 32MB, 16-way | 1MB, 8-way | 512KB, 8-way | 256KB, 8-way |

Como pode ser observado na Tabela 5, o processador *Haswell*™ possui a maior capacidade de leitura dos HTMs, mesmo tendo as menores *caches*. Este dado revela que, além das *caches*, outra estrutura em *hardware* deve auxiliar a manutenção do estado especulativo nas leituras. Os já frequentes cancelamentos por capacidade observados no *Haswell*™ serão ainda maiores nos demais processadores devido a menor capacidade de leitura. Os falsos conflitos também tendem a aumentar uma vez que, geralmente, quanto menor o espaço maiores são as chances de dados diferentes serem mapeados para mesma linha de *cache*. A granularidade grossa da detecção de conflitos também aumentará os cancelamentos por falsos conflitos. O *Blue Gene/Q*™, por sua vez, possui a maior capacidade de escrita. Esta característica pode justificar o *speedup* da aplicação `Yada`, ainda que modesto, de pouco mais 1.5x quando executada com 8 *threads* [Nakaike et al. 2015].

## 5. Conclusão

Neste trabalho foi apresentada uma reavaliação de desempenho e caracterização das transações do pacote STAMP no processador com suporte transacional *Haswell*™. A análise revelou que, mesmo um suporte reduzido em *hardware* de TM, pode apresentar desempenho comparável (5 de 8 aplicações) ao de uma biblioteca em *software* de TM. Também mostrou-se que algumas aplicações não são aceleradas pelo *hardware* (3 de 8 aplicações). Objetivando determinar as razões que impedem a aceleração dessas

aplicações pelo suporte transacional do *Haswell*™, conduziu-se uma caracterização por transação das aplicações STAMP. A caracterização apresenta um padrão das aplicações até então desconhecido: existência de uma transação que domina o tempo de execução. Além disso, as aplicações não aceleradas possuem transações muito longas e com um grande *footprint* de memória. Com base nos resultados e nas características, foi possível extrapolar os resultados da caracterização e concluir que as aplicações `Yada` e `Labyrinth`, como estão codificadas, não serão aceleradas em nenhum HTM presente nos processadores disponíveis atualmente.

## Referências

Barroso, L. A. and Holzle, U. (2007). The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37.

Dalessandro, L., Spear, M. F., and Scott, M. L. (2010). NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming*, pages 67–78.

de Carvalho, J. P. L., Baldassin, A., and Azevedo, R. (2013). Reassessing the energy efficiency of software transacional memory on commodity processors. In *WSCAD-SSC 2013*, Porto de Galinhas, Ipojuca, Pernambuco.

Dice, D., Lev, Y., Moir, M., and Nussbaum, D. (2009). Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168.

Diegues, N., Romano, P., and Rodrigues, L. (2014). Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 3–14. ACM.

Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633.

Goel, B., Titos-Gil, R., Negi, A., Mckee, S., Stenstrom, P., et al. (2014). Performance and energy analysis of the restricted transactional memory implementation on haswell. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 615–624. IEEE.

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

Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., and Olukotun, K. (2010). Eigenbench: A Simple Exploration Tool for Orthogonal TM Characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–11, Washington, DC, USA. IEEE Computer Society.

Le, H., Guthrie, G., Williams, D., Michael, M., Frey, B., Starke, W., May, C., Odaira, R., and Nakaike, T. (2015). Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development*, 59(1):8–1.

Lee, C. Y. (1961). An algorithm for path connections and its applications. *Electronic Computers, IRE Transactions on*, (3):346–365.

Lomet, D. B. (1977). Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137.

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

Matveev, A. and Shavit, N. (2015). Reduced Hardware NOrec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71, New York, NY, USA. ACM.

Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K. (2008). STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46.

Nakaike, T., Odaira, R., Gaudet, M., Michael, M. M., and Tomari, H. (2015). Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 144–157, New York, NY, USA. ACM.

Ruppert, J. (1995). A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of algorithms*, 18(3):548–585.

Schindewolf, M., Bihari, B., Gyllenhaal, J., Schulz, M., Wang, A., and Karl, W. (2012). What Scientific Applications Can Benefit from Hardware Transactional Memory? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 90:1–90:11, Los Alamitos, CA, USA. IEEE Computer Society Press.

Sutter, H. and Larus, J. (2005). Software and the concurrency revolution. *Queue*, 3(7):54–62.

Wall, D. W. (1991). Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188, New York, NY, USA. ACM.

Wang, A., Gaudet, M., Wu, P., Amaral, J. N., Ohmacht, M., Barton, C., Silvera, R., and Michael, M. (2012). Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 127–136.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995). The SPLASH-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM.

Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24.

Xiang, L. and Scott, M. L. (2015). Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 76–86, New York, NY, USA. ACM.

Yoo, R. M., Hughes, C. J., Lai, K., and Rajwar, R. (2013). Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–11. IEEE.