Reavaliando a Eficiência Energética de Memória Transacional em Processadores Convencionais

João P. L. de Carvalho, Alexandro Baldassin
UNESP – Univ Estadual Paulista
Rio Claro, SP – Brazil
jaopaulolc@gmail.com, alex@rc.unesp.br

Rodolfo Azevedo
Institute of Computing, UNICAMP
Campinas, SP – Brazil
rodolfo@ic.unicamp.br

***Resumo*—Memória Transacional (TM) é um mecanismo recente de sincronização que objetiva ao mesmo tempo facilitar o desenvolvimento de aplicações concorrentes e fornecer desempenho. A maioria dos trabalhos em TM focam avaliação de desempenho, negligenciando outras métricas como consumo de energia. Trabalhos anteriores analisaram a eficiência energética de implementações de TM através de ambientes simulados com modelo de execução simplificado. Este trabalho apresenta uma reavaliação do consumo de energia e desempenho de uma implementação moderna de memória transacional em processadores convencionais. Será descrito e discutido uma técnica de medição de energia baseada em Registradores Específicos de Modelo (MSRs), técnica essa utilizada para obter e analisar o consumo energético de aplicações transacionais do *benchmark* STAMP, bem como de suas respectivas implementações baseadas em travas. Também será discutido o comportamento do consumo de energia e desempenho devido à influência de diferentes gerenciados de frequência disponíveis em configurações típicas de sistemas Linux. Nossos resultados mostram que memória transacional em software de fato reduz o consumo de energia quando comparada a implementações baseadas em travas grossas em 7 das 8 aplicações estudadas em um processador Intel® Sandy Bridge™.**

***Keywords*-Programação concorrente, Análise de desempenho e energia; Memória Transacional;**

I. Introdução

Ao longo dos anos, os microprocessadores sofreram atualizações em suas microarquiteturas objetivando atender à crescente demanda por desempenho. Além disso, o avanço tecnológico impulsionado pela Lei de Moore permitiu frequências de operação cada vez mais altas, aumentando de forma transparente o desempenho do software. No entanto, a chegada do *power wall* freou de forma brusca o aumento da velocidade de operação dos processadores. Avanços na microarquitetura também esbarraram no limite do ILP (*Instruction Level Parallelism*), explorado de forma exaustiva até então. A alternativa encontrada para esses problemas foi investir em microarquiteturas com múltiplos núcleos de execução (*multicore*).

A chegada dos processadores *multicore* tem causado uma verdadeira revolução no software. É aparente que as ferramentas e modelos atuais para programação concorrente estão aquém do nível necessário para a exploração eficiente de todos os núcleos de processamento [1], requerendo abordagens novas. Uma dessas abordagens é conhecida como *Memória Transacional* (TM) [2], que usa o conceito de transação como a principal abstração para prover controle de concorrência em sistemas com múltiplas linhas de execuções (*threads*).

A grande maioria dos sistemas transacionais costumam ser avaliados de acordo com seu desempenho, desconsiderando o impacto do consumo de energia. Entendemos que a análise desse impacto é importante se TM pretende ser empregado em escala comercial, como apontado recentemente por empresas como a IBM e a Intel [3], [4]. A preocupação com questões energéticas não é algo novo [5], [6], mas apenas recentemente tem-se observado uma atenção maior por parte da indústria [7] e da academia em sua investigação. Uma manifestação concreta dessa preocupação é a interface RAPL (*Running Average Power Limit*) [8] adicionada à microarquitetura Intel® a partir do modelo *Sandy Bridge*™.

Fica evidente, desta forma, a importância de se conduzir uma análise do consumo de energia em sistemas transacionais. A maioria dos trabalhos existentes tem empregado plataformas de simulação, apresentando como principal limitação a simplificação do modelo arquitetural. Neste trabalho fazemos uma reavaliação do consumo energético em sistemas com memória transacional em software (STM) em processadores comerciais. Nosso trabalho traz algumas novidades, como a análise dos gerenciadores de frequências comumente encontrados em sistemas Linux.

Mais especificamente, apresentamos as seguintes contribuições:

- implementação de uma API (*Application Programming Interface*), baseada na interface RAPL, para avaliação do consumo de energia em processadores atuais (Seção III-B);
- avaliação do consumo de energia e desempenho de todas as 8 aplicações do pacote transacional STAMP (*Stanford Transactional Applications for Multi-Processing*) [9] (Seção IV-B). Nossa análise considera ambas as versões convencional, baseada em travas (*locks*), e transacional, baseada na biblioteca TinySTM [10]. Os resultados apontam que, quando comparado à versão convencional, STM tem melhor desempenho e custo energético em 7 das 8 aplicações;
- análise do impacto de diversas políticas de gerenciamento de frequência no consumo de energia e desempenho (Seção IV-C). Constatamos que o uso de uma política diferente da padrão adotada pelo sistema operacional melhora o EDP (*Energy Delay Product*) de 3 das 8 aplicações transacionais estudadas, cons-

tatando que nem sempre usar a configuração padrão é a melhor opção.

Este artigo está organizado da seguinte forma. A Seção II apresenta uma breve introdução sobre memória transacional. A Seção III discute os principais conceitos utilizados no trabalho com relação a consumo de energia e apresenta nossa metodologia de medição baseada na RAPL. Em seguida, a Seção IV analisa os principais resultados obtidos para o pacote de aplicações STAMP, seguido de trabalhos relacionados, na Seção V, e nossas conclusões, na Seção VI.

II. Memória Transacional: Uma Breve Introdução

Memória transacional é um modelo de computação paralela que abstrai a região compartilhada da memória em uma entidade similar a um banco de dados. Assim, operações nessa região (leitura e escrita) possuem os seguintes atributos, analogamente às transações em banco de dados: atomicidade, consistência e isolamento. Um quarto atributo, durabilidade, completa o acrônimo ACID (Atomicidade, Consistência, Isolamento e Durabilidade), mas como as operações em memória transacional ocorrem em uma memória volátil, ele não é considerado. Transação é um conceito da área de banco de dados, sendo introduzido primeiramente por Eswaran et al. [11] em 1976. No ano seguinte, sua definição e propriedades inspirou uma abordagem de programação concorrente, baseada em operações atômicas para facilitar a codificação e sincronização de processos [12].

Dizer que uma operação/transação em memória é atômica significa que estados intermediários não são observados pelo sistema. Ou seja, ou a operação é bem sucedida, ou é reiniciada e para o resto do sistema é como se nada tivesse acontecido (uma propriedade conhecida comumente como *tudo ou nada*). O termo Memória Transacional (TM) foi cunhado por Herlihy e Moss [13] em 1993. Implementações em *hardware* e em *software* foram propostas (os leitores devem consultar a referência [2] para um tratamento completo sobre o tema). Recentemente, a Intel® adicionou suporte básico para Memória Transacional em seu novo modelo chamado *Haswell*®, o que mostra o investimento na área, até então reservado à academia, por parte da indústria de microprocessadores.

Dois conceitos principais são usados na implementação de sistemas transacionais: *versionamento de dados* e *detecção de conflitos*. O versionamento de dados é necessário devido à natureza otimista da execução transacional. Desta forma, é necessário armazenar duas versões para cada dado acessado pela transação: uma especulativa e outra antiga. Se o dado especulativo é armazenado em um buffer interno e o antigo é armazenado na memória principal, dizemos que a implementação usa versionamento *deferido*. Do contrário (dado especulativo na memória principal e antigo em buffer interno), dizemos que o versionamento é *direto*.

Por sua vez, a detecção de conflitos determina quando um conflito é detectado. Um conflito é caracterizado pela tentativa de leitura ou modificação de uma região de memória previamente lida ou modificada por outra transação ainda não finalizada. 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 postergada 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 conhecida por TinySTM [10]. Nessa implementação, a estratégia de versionamento de dados padrão é deferida e a detecção de conflitos é realizada de forma antecipada. O TinySTM usa uma implementação bloqueante para detectar os conflitos, conhecida por ETL (*Encounter-Time Locking*) – os dados acessados pela transação são mapeados para uma tabela de travas quando o primeiro acesso é efetuado. A política de gerenciamento de contenção adotada pelo TinySTM por padrão é conhecida como *suicide*, fazendo com que a transação que detectou o conflito seja abortada e reiniciada imediatamente.

III. Gerenciamento e consumo de energia

Nessa seção serão introduzidos os conceitos básicos envolvidos na realização desse trabalho com relação ao gerenciamento e consumo de energia. Começamos descrevendo o suporte disponível em sistemas Linux para gerenciamento de energia, passando pelo suporte microarquitetural para realização das medições e terminando com a apresentação da nossa API.

A. Políticas para Gerenciamento de Frequência

Atualmente, o *kernel* do Linux implementa cinco políticas de gerenciamento de frequência, são elas: *powersave*, *userspace*, *conservative*, *ondemand* e *performance*. O objetivo dessas políticas é estender as funcionalidades do módulo *acpi-cpufreq* de maneira a permitir que a frequência de operação possar ser modificada dinamicamente conforme as diretrizes de cada política. A Tabela I apresenta uma breve explicação para cada uma dessas políticas.

As políticas *conservative* e *ondemand* operam entre a frequência mínima e a máxima. A diferença é que o salto de frequência na primeira ocorre de forma gradativa, ou seja, o processador permanece um intervalo de tempo nas frequências intermediárias antes de chegar na correspondente à carga de trabalho atual do sistema [1]. Já na segunda o salto é direto, ou seja, da frequência atual para a correspondente à carga de trabalho e vice-versa. É importante salientar que as frequências disponíveis de operação da CPU são definidas de fábrica e variam de processador para processador. A política *ondemand* é usada como padrão na maioria dos sistemas Linux para *desktops*. A *conservative*

[1] O *kernel* verifica a carga de trabalho do sistema regularmente (na ordem de alguns ms) para decidir se a frequência de operação atual da CPU dever ser alterada.

Tabela I: Políticas para gerenciamento de frequência

Política	Descrição
powersave	Configura a frequência de operação para a mínima disponível. A frequência permanece a mínima independente da carga de trabalho da CPU
userspace	Sua função é permitir que qualquer aplicação rodando com o UID 0 (privilégio de superusuário) escolha qual frequência a CPU deve operar
conservative	Aumenta e diminui, gradativamente, a frequência conforme a carga de trabalho da CPU. O passo e o limiar da carga de trabalho no qual a redução ou aumento da frequência ocorrem podem ser configurados
ondemand	Similar à *conservative*, porém o aumento ou redução da frequência é discreto – ou seja, não ocorre de forma gradativa, e sim direta
performance	Similar à *powersave*, porém a máxima frequência disponível é fixada ao invés da mínima

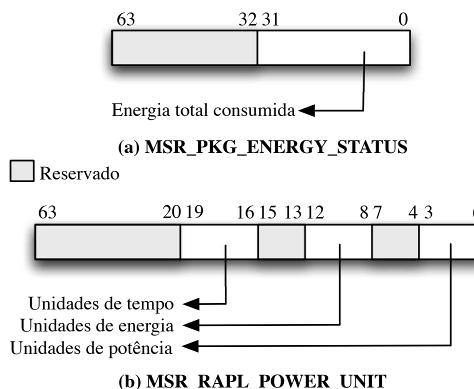

Figura 1: Registradores da interface RAPL: (a) registrador que armazena o contador; (b) registrador que armazena as unidades.

é mais atraente em sistemas portáteis, nos quais o tempo de vida da bateria é importante.

Atualmente, existem basicamente duas maneiras de escolher qual gerenciador de frequência, daqui em diante referenciado por *governor*, o sistema deve usar: pelo uso de aplicações utilitárias disponíveis no sistema (e.g. aplicações do pacote *cpufrequtils*); ou pela edição dos arquivos de configuração dos módulos do *kernel*, localizados geralmente em *"/sys/devides/cpu/"*. A primeira é recomendada quando deseja-se observar o comportamento da aplicação – todas as suas *threads* – em cada *governor* disponível, enquanto que a segunda é recomendada para aplicações nas quais deseja-se permitir que cada uma das suas *threads* faça a escolha em tempo de execução.

B. Registradores Específicos de Modelo e a RAPL

Os registradores específicos de modelos, ou MSRs (da sigla em inglês), são registradores adicionados à microarquitetura para facilitar a depuração de aplicações, permitir o monitoramento de desempenho e configurar as características da CPU. O conceito originou-se na microarquitetura *Pentium*™, quando a Intel® introduziu registradores para testar a TLB (*Translation Look-Aside Buffer*), algo novo para a época. Tais registradores sempre tiveram um carácter experimental, o que não garantia sua permanência nos modelos futuros. Ao longo dos anos, com base na sua aceitação por parte dos usuários e comprovação de sua utilidade, novos registradores foram adicionados, e outros retirados.

Provavelmente motivado pela preocupação, não apenas por parte da academia, com o consumo energético de aplicações, um grupo de registradores específicos de modelo foi adicionado e está presente a partir da microarquitetura *Sandy Bridge*™ para auxiliar na tarefa de medir o consumo de energia. No manual do desenvolvedor, tal grupo é referenciado por RAPL (*Running Average Power Limit*) [8], [14]. A RAPL é uma interface constituída de contadores de consumo de energia, dissipação de potência e tempo, o que permite o monitoramento do desempenho e imposição de restrições mais severas de consumo de energia. Nesse trabalho é utilizado apenas o contador de consumo de energia. A Figura 1 mostra uma representação visual da interface.

O registrador da Figura 1a é um contador de 32 bits que é atualizado a cada $1ms$, aproximadamente. O registrador da Figura 1b armazena as unidades da interface RAPL. A energia consumida, em joules, por um trecho de programa é dada pela seguinte expressão:

$$E = (C_{depois} - C_{antes}) * 2^{(-EU)}$$

onde C_{depois} e C_{antes} representam os valores obtidos através da leitura do registrador de contagem (Figura 1a) depois e antes do trecho de código, e EU (*Energy Unit*) é dado pelos bits 12:8 do registrador `MSR_RAPL_POWER_UNIT` (Figura 1b). As unidades da interface RAPL são dependentes de arquitetura. Para o processador utilizado neste trabalho o valor de EU é 16, provendo uma granularidade de $15,3$ microjoules. É importante salientar que, devido à limitação de tamanho do contador (inteiro de 32 bits), se faz necessário um cuidado com o tempo de *wraparound* – aproximadamente $60s$, conforme documentado no manual do desenvolvedor da Intel.

No que diz respeito aos MSRs para coleta de dados de consumo de energia, o suporte nativamente existente é o mapeamento que o *kernel* faz dos registradores da RAPL em *character devices* – arquivos especiais que podem ser acessados (leitura e escrita) como se fossem arquivos convencionais. Assim, caso deseje-se coletar esses dados é necessário realizar operações *bitwise* no valor dos registradores mapeados.

Fica evidente o inconveniente de se utilizar essa abordagem diretamente, pois além de a aplicação necessitar acessar essas entidades de baixo nível diretamente e realizar operações *bitwise*, o código da aplicação torna-se não portável, devido ao fato de cada modelo de processador possuir multiplicadores de unidade diferentes (armazenados nos registradores da RAPL, vide Figura 1b). Visando melhorar esse aspecto da infraestrutura, uma API que encapsula a complexidade do nível mais baixo foi desenvolvida.

C. Metodologia e API para Medição da Energia

Para obtenção dos dados de consumo de energia apresentados na seção IV, foi necessária a instrumentação do código das aplicações de modo a extrair esses dados dos MSRs. Desta forma, uma API foi desenvolvida para encapsular as operações de baixo nível – operações de mascaramento e *bitwise*, bem como leitura dos *character devices* mapeados pelo *kernel* – em funções de alto nível. A seguir é descrito o papel de cada uma das funções da API.

- **msrInit()**: inicializa as variáveis internas da API como, por exemplo, o multiplicador de magnitude da unidade de energia armazenado no registrador mostrado pela Figura 1b;
- **msrGetCounter()**: retorna um inteiro sem sinal que representa o valor do contador de consumo energético, vide Figura 1a;
- **msrDiffCounter(antes, depois)**: retorna um ponto flutuante de dupla precisão que representa o valor, em joules, da diferença entre os dois valores passados como parâmetro.

O procedimento para coleta de energia de um trecho de código usando a API procede da seguinte maneira: a função *main()* de cada uma das aplicações chama a função *msrInit()*; em seguida, duas chamadas da função *msrGetCounter()*, uma antes e outra depois do trecho em questão, retornam o valor do contador; ao final, obtém-se a quantidade de energia consumida invocando a função *msrDiffCounter()* com os valores do contador obtidos anteriormente.

Como as aplicações estudadas não possuem tempo de execução superior a 60s, a rotina *msrDiffCounter()* não trata esse caso. O único cuidado que tivemos de tomar é quando o valor do contador tomado após a região a ser medida é menor do que o valor do contador tomado antes da mesma região. A rotina *msrDiffCounter()* detecta essa situação e ajusta os valores corretamente.

Como dito anteriormente, existem basicamente duas maneiras de se alterar o *governor*: utilizando aplicações utilitárias disponíveis no sistema ou edição dos arquivos dos módulos do *kernel*. A primeira maneira foi a adotada nesse trabalho, pois desejou-se observar os resultados da aplicação como um todo. Portanto, o *script* que faz a submissão das aplicações e coleta dos dados da execução executa cada aplicação com os 5 *governors* disponíveis. A alteração se dá antes da aplicação executar, minimizando assim possíveis flutuações provocadas por latências do *kernel*.

Uma das limitações de nossa metodologia é que o contador da interface RAPL contabiliza o consumo apenas dos núcleos de processamento e dos três níveis de *cache* encapsulados no *chip*. Dessa forma, não é contabilizado o custo fora do *chip* (memória principal e dispositivos de entrada e saída). Essa limitação não prejudicou o nosso trabalho, pois o número de faltas no último nível de *cache* não ultrapassou 3 (dados obtidos com a aplicação *perf*), mostrando que o custo com memória principal das aplicações é baixo. Além disso, os trechos de código instrumentados não possuem operações de entrada e saída, já que o processamento transacional proíbe tais operações dentro das transações. A complementação desse trabalho, levando em conta o consumo da memória principal, é de pretensão dos autores em trabalhos futuros.

IV. Resultados Experimentais

Nessa seção são apresentados os resultados da reavaliação do desempenho e consumo de energia característicos de sistemas transacionais e também os baseados em travas com granularidade grossa. Para coleta dos dados usamos a metodologia e API introduzidas anteriormente.

A. Configuração e Aplicações

Para a análise foi escolhido o pacote de aplicações conhecido como STAMP [9]. Este pacote é composto por 8 aplicações com uma boa diversidade de parâmetros transacionais tais como tempo de execução, tempo em transação e nível de contenção, sendo o conjunto de aplicações mais utilizado na análise de TM. A biblioteca transacional usada é a TinySTM [10], versão 1.0.4, considerada estado da arte da pesquisa em STM. A configuração usada dessa biblioteca é a padrão, usando tipo de versionamento deferido e detecção de conflitos antecipada. O gerenciador de contenção usado é o *suicide*, que faz com que a transação que detectou o conflito seja reiniciada imediatamente.

Os dados apresentados nessa seção foram obtidos em uma máquina com processador Intel® i7-2700K com *hyperthreading* habilitado, perfazendo um total de 8 núcleos. As frequências mínima e máxima de operação são iguais a 1.6 e 3.5Ghz, respectivamente. A máquina conta com um total de 8Gb de memória RAM e usa um ambiente Linux típico, com *kernel* versão 3.2.0. As aplicações e biblioteca STM foram compiladas usando o `GCC` versão 4.6.3, com nível três de otimização (`-O3`). Cada experimento foi executado 10 vezes para obtenção dos resultados apresentados nessa seção.

Para dar credibilidade aos nossos resultados, apresentamos na Tabela II os valores médios de tempo e energia obtidos, juntamente com o desvio padrão, correspondentes aos experimentos da Seção IV-B. Como pode ser observado, em geral não há uma alta variabilidade nos resultados. A aplicação `Bayes`, devido ao seu carácter não-determinístico, é a única a produzir valores mais elevados (esse comportamento já é conhecido na comunidade de TM).

Nossa análise é feita em duas partes. Na primeira, discutida na Seção IV-B, analisamos o impacto no tempo de execução e consumo de energia de uma configuração padrão, onde o *governor* do sistema operacional é o *ondemand* (habilitado por padrão). Essa análise nos mostra o tempo de execução e consumo de energia das aplicações quando executadas em um ambiente convencional, experimentado pela maioria dos usuários. Em seguida, na Seção IV-C, analisamos o que é alterado quando o *governor* é trocado. Esse experimento nos permite analisar o impacto do *governor* no comportamento do desempenho e custo energético.

Tabela II: Média e desvio padrão das aplicações para cada número de threads

Aplicação	Energia (J)								Tempo (s)							
	1 thread		2 threads		4 threads		8 threads		1 thread		2 threads		4 threads		8 threads	
	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
bayes	187.3480	0.8372	158.1610	13.3631	123.6270	51.7013	175.8910	45.1597	7.8672	0.0178	4.4902	0.3036	2.3345	0.6438	3.4645	1.0722
genome	145.2930	0.2764	107.8910	1.3174	93.1700	2.0166	76.9540	1.2761	6.5841	0.0152	3.3587	0.0427	1.6803	0.0509	1.1419	0.0270
intruder	434.0900	5.9061	378.0480	2.7646	374.4590	1.4000	323.0130	1.2545	19.7210	0.1990	11.1185	0.0694	6.1921	0.0392	4.6893	0.0139
kmeans	75.3427	1.0804	61.3877	4.8018	61.0469	7.7342	63.7385	5.3278	3.0024	0.0466	1.5434	0.1204	0.8593	0.1106	0.7492	0.0634
labyrinth	1069.3200	4.6891	890.6150	4.5752	882.4310	2.7745	865.8080	11.0221	44.3639	0.0144	23.2282	0.0445	12.6857	0.0289	11.1578	0.1368
ssca2	175.4770	4.1517	151.9150	0.7897	150.8510	0.1925	119.8030	0.2443	8.0193	0.1917	4.5870	0.0096	2.6523	0.0066	1.9983	0.0075
vacation	569.6060	12.2348	486.9540	5.1531	453.2090	1.8620	354.7100	0.4917	24.9548	0.4864	13.6970	0.1618	7.0970	0.0159	4.8513	0.0095
yada	186.3380	2.1990	205.6920	2.0293	230.1490	5.7160	235.2660	1.4731	7.7535	0.0806	5.4696	0.0138	3.4365	0.0979	3.0315	0.0176

B. *Análise do Desempenho com Configuração Padrão*

Os gráficos da Figura 2 mostram, para cada aplicação estudada, o tempo de execução em segundos (gráfico mais à esquerda), o consumo de energia em joules (gráfico central) e o produto energia-latência (EDP) (gráfico mais à direita) normalizado em relação à execução sequencial. São mostrados os resultados para a execução transacional (STM) e a baseada em travas (LOCK), variando o número de *threads* entre 1 e 8. Nos gráficos de tempo e consumo também é apresentado uma linha base respectiva à execução sequencial como referência (SEQ). Como os resultados de EDP estão normalizados em relação à execução sequencial, um valor menor do que 1 representa um resultado melhor do que a sequencial (e vice-versa).

Os gráficos da Figura 2 mostram que em 7 das 8 aplicações STAMP, a abordagem que utiliza STM teve melhor desempenho e menor consumo de energia que a abordagem que utiliza travas de granularidade grossa. A única exceção é a aplicação `K-Means`, onde a abordagem com travas foi melhor que à com STM. Isso se deve a dois fatos: primeiro que a aplicação `K-Means` possui regiões críticas muito breves e, segundo, que o restante do código concorrente pode ser executado totalmente em paralelo sem a necessidade de sincronização. Desse modo, o *overhead* introduzido pela STM nessas pequenas regiões deteriora o desempenho e a minimização desse *overhead* no uso de travas apresenta melhor resultado.

Ao que tange consumo de energia, apenas as aplicações `Genome`, `Bayes` e `Labyrinth` consumiram menos energia que a versão sequencial quando utilizaram STM, ao passo que as demais consumiram mais, apesar de apresentarem redução com o aumento do número de *threads*. As aplicações `Yada`, `Ssca2` e `K-Means`, em especial, mostraram aumento de consumo de energia, ou redução pouco significativa, mesmo apresentando menor tempo de execução, o que exemplifica o porque uma avaliação de desempenho apenas sob à ótica de tempo de resposta é uma análise incompleta e não caracteriza o real desempenho do objeto em análise. Esse resultado mostra a necessidade de se compreender e descobrir quais fatores influenciam a eficiência energética das aplicações, bem como entender como essa influência se dá.

C. *Análise Comparativa com Múltiplos Governors*

No gráfico da Figura 3 são apresentados o ganho em EDP de diferentes *governors* quando comparados ao *ondemand*, usado na análise anterior, para as aplicações com STM. O ganho é calculado dividindo-se o EDP de uma dada aplicação com a política *ondemand* pela mesma aplicação com uma das seguintes políticas: *powersave*, *conservative*, *userspace* e *performance*. Desta forma, valores acima de 1 representam ganhos maiores. Os dados apresentados foram coletados das execuções com 8 *threads* por ser o cenário onde existe maior concorrência e por também ser o cenário onde se espera que o uso de sistemas transacionais exiba melhores resultados. O módulo *userspace*, como já dito, permite a escolha da frequência de operação pelo usuário. Os resultados aqui apresentados foram obtidos com a frequência fixada em 2.6Ghz. Essa configuração permite a análise de um cenário onde a frequência está em um valor intermediário ao observado quando as políticas *powersave* e *performance* estão ativas.

O gráfico mostra que, no geral, o desempenho da aplicação piora com a mudança do *governor*. A redução da frequência de operação aumenta o tempo de execução e não reduz o consumo energético significativamente, aumentando assim o EDP e reduzindo o ganho. De maneira análoga, o aumento da frequência de operação aumenta, desproporcionalmente, o consumo de energia sem redução significativa do tempo de execução, o que também leva à um maior EDP e menor ganho. Entretanto, foi observado que algumas aplicações podem se beneficiar do aumento de frequência. As aplicações `Genome`, `Ssca2` e `Yada` apresentaram redução do tempo de execução com aumento (`Yada`) e redução (`Genome` e `Ssca2`) pouco significativos do consumo de energia.

A diminuição do ganho em EDP observado em quase todas as aplicações pode ser justificado, também, pelo fato do TinySTM, e STMs em geral, operar de maneira otimista, ou seja, as *threads* das aplicações sempre permanecem ativas mesmo que essa atividade seja reiniciar a transação abortada, prosseguir por um tempo e abortar novamente em um ciclo. As aplicações que obtiveram ganho foram as que apresentam a característica de estabilidade de consumo energético com o aumento do número de *threads*, como podemos observar, por exemplo, na aplicação `Ssca2` no gráfico da Figura 2.

V. TRABALHOS RELACIONADOS

Esta seção descreve os dois tipos principais de trabalhos correlatos abordados no artigo: técnicas de medição de energia usando a interface RAPL, e caracterização do consumo de energia e desempenho de sistemas com memória transacional.

A introdução de contadores para medição de energia diretamente em processadores é algo recente, tendo sido disponibilizado pela interface RAPL [8]. Um dos primeiros trabalhos a investigar o consumo de energia em

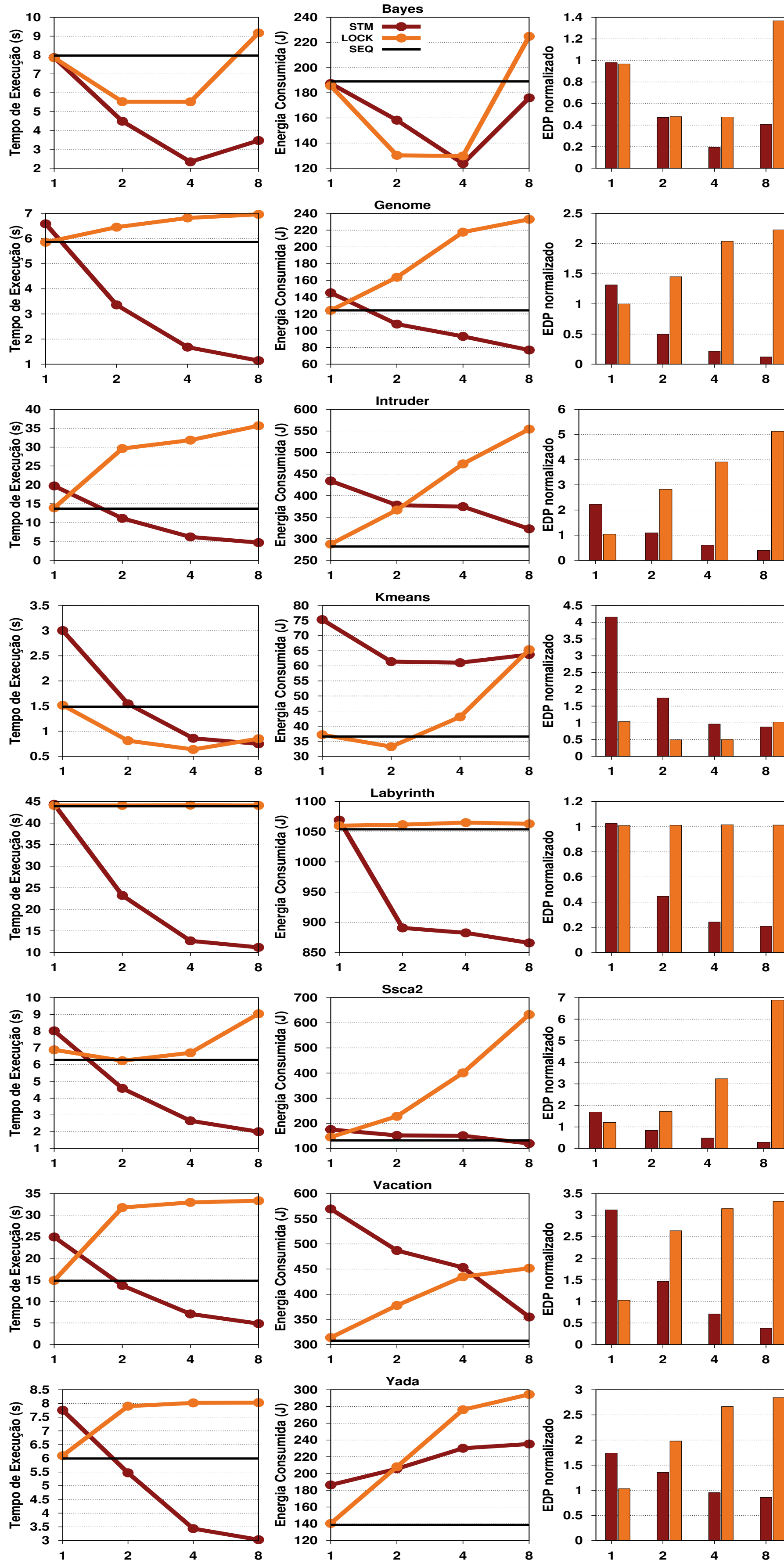

Figura 2: Resultados com *governor* padrão (*ondemand*)

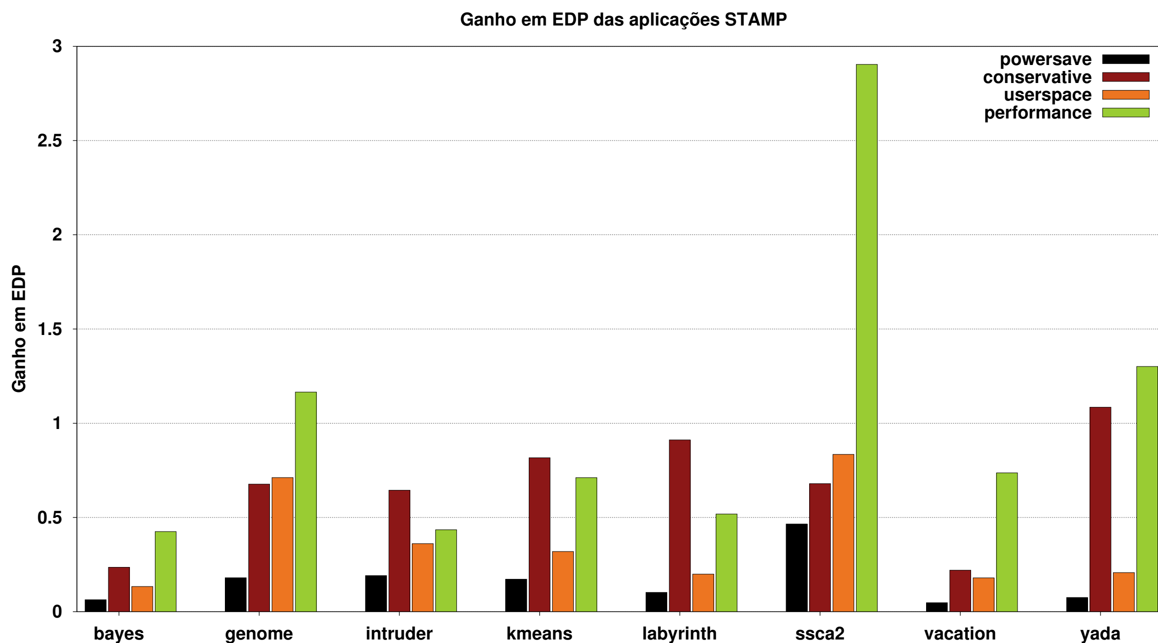

Figura 3: Ganho de desempenho de cada *governor* com relação ao padrão (*ondemand*) com 8 threads

aplicações paralelas usando a interface RAPL é devido a Lien et al. [15]. Eles analisam o impacto no consumo de energia e desempenho de diferentes tipos de paralelização e vetorização. Cebrian et al. [16] estenderam o trabalho anterior para contabilizar também o consumo devido a componentes fora do *chip*, como dispositivos de entrada e saída e memória principal. Ambos trabalhos concluem que o uso de vetorização é o principal fator para a redução do gasto energético. Apesar de nossa abordagem não contabilizar o consumo de energia fora do *chip*, foi observado que a extrema maioria dos acessos ao último nível de cache não causa faltas, e portanto o impacto extra-*chip* é minimizado. Esse mesmo argumento é usado pelo trabalho de Cebrian et al. [15].

Hahnel et al. [17] reportam as experiências com o uso da interface RAPL para medir o consumo de trechos de código curtos, da ordem de alguns milissegundos. Como as medidas coletadas através dessa interface são atualizadas aproximadamente a cada $1ms$, é importante conhecer qual a confiabilidade da abordagem quando blocos de código possuem a mesma ordem do período com que os registradores são atualizados. Nossa metodologia mede o consumo da região executada concorrentemente, que possui tempo de execução maior que esse intervalo. Desta forma, não tivemos problemas com a frequência de atualização dos registradores.

A medição e análise do consumo de energia em sistemas transacionais foram inicialmente desenvolvidas em ambientes simulados, abordando sistemas com suporte em hardware para TM [18], [19]. Posteriormente sistemas em software também foram avaliados, usando a mesma infraestrutura de simulação [20], [21], [22]. Apesar de as análises iniciais serem válidas, ambientes simulados trabalham com algumas simplificações que podem afetar o resultado final. Vale ressaltar que a maioria desses trabalhos iniciaram-se em um tempo onde a medição direta de energia era muito complicada de ser efetuada. Com o advento da interface RAPL e instrumentos para medição de energia, a situação vem se alterando.

Rico et al. [23] mostram o comportamento energético de um subconjunto do pacote STAMP para três bibliotecas de STM. Eles se utilizam do BMC (*Baseboard Management Controller*) presente no sistema computacional empregado nos experimentos para avaliar o consumo de energia. Nosso trabalho avança esse estudo ao considerar diferentes gerenciadores de frequência e aplicações baseadas em travas. O trabalho mais próximo do apresentado nesse artigo deve-se a Gautham et al. [24]. Os autores também utilizam um método de medição baseado na RAPL, considerando as aplicações do pacote STAMP em ambas versões transacionais e com travas. No entanto a aplicação `Labyrinth` foi omitida. Diferente deste último, nosso trabalho analisa o impacto em EDP das aplicações transacionais, mostrando que para algumas a troca do *governor* é compensadora no que tange a EDP.

VI. CONCLUSÃO

A reavaliação realizada neste artigo mostrou que, de fato, o uso de memória transacional como mecanismo de extração de paralelismo aumenta o desempenho das aplicações, reduzindo não apenas o tempo de execução mas também a energia consumida quando comparado a métodos tradicionais baseados em travas. Como foi apresentado, em 7 das 8 aplicações estudadas a vantagem do emprego de STM foi clara. No entanto, somente 3 aplicações conseguiram ganhos significativos de energia se comparadas às suas versões sequenciais. Este resultado mostra que, se o consumo de energia realmente for de

extrema importância, talvez a versão sequencial seja o suficiente.

Este trabalho também analisou o impacto de diferentes políticas de controle de frequência disponíveis em sistemas Linux típicos. Em especial, mostramos que a troca da política padrão para uma outra melhorou o EDP em 3 das 8 aplicações estudadas, com ganhos significativos de até 3x. Esta observação permite concluir que a política de gerenciamento da frequência de operação dos processadores pode ter um impacto significativo no sistema e, desta forma, deve ser levada em consideração no projeto e análise de algoritmos transacionais.

AGRADECIMENTOS

Os autores agradecem o apoio financeiro da FAPESP no contexto dos projetos 2011/19373-6 e 2013/03053-8.

REFERÊNCIAS

[1] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005.

[2] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*, 2nd ed. Morgan & Claypool Publishers, Jun. 2010.

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

[4] *Intel® Architecture Instruction Set Extensions Programming Reference*, Intel Corporation, Feb. 2012.

[5] L. A. Barroso and U. Holzle, "The case for energy-proportional computing," *IEEE Computer*, vol. 40, no. 12, pp. 33–37, Dec. 2007.

[6] N. Firasta, M. Buxton, K. Nasri, and S. Kuo, *Intel® AVX: New Frontiers in Performance Improvements and Energy Efficiency*, May 2008.

[7] P. Larsson, *Intel® Energy-Efficient Software Guidelines*, June 2012.

[8] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, 2010, pp. 189–194.

[9] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Sep. 2008, pp. 35–46.

[10] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, Feb. 2008, pp. 237–246.

[11] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, Nov. 1976.

[12] D. B. Lomet, "Process structuring, synchronization, and recovery using atomic actions," in *Proceedings of an ACM conference on Language design for reliable software*, Mar. 1977, pp. 128–137.

[13] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, Jun. 1993, pp. 289–300.

[14] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Jan 2013, vol. 3B.

[15] H. Lien, L. Natvig, A. A. Hasib, and J. C. Meyer, "Case studies of multi-core energy efficiency in task based programs," in *ICT as Key Technology Against Global Warming*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7453, pp. 44–54.

[16] J. M. Cebrian, L. Natvig, and J. C. Meyer, "Improving energy efficiency through parallelization and vectorization on Intel Core i5 and i7 processors," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov. 2012, pp. 675–684.

[17] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using RAPL," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, Jan. 2012.

[18] T. Moreshet, R. I. Bahar, and M. Herlihy, "Energy reduction in multiprocessor systems using transactional memory," in *Proceedings of the 2005 international symposium on Low power electronics and design*, Aug. 2005, pp. 331–334.

[19] C. Ferri, S. Wood, T. Moreshet, R. I. Bahar, and M. Herlihy, "Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems," *Journal of Parallel and Distributed Computing*, vol. 70, no. 10, pp. 1042–1052, Oct. 2010.

[20] F. Klein, A. Baldassin, G. Araujo, P. Centoducatte, and R. Azevedo, "On the energy-efficiency of software transactional memory," in *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design*, Sep. 2009, pp. 1–6.

[21] A. Baldassin, F. Klein, G. Araujo, R. Azevedo, and P. Centoducatte, "Characterizing the energy consumption of software transactional memory," *IEEE Computer Architecture Letters*, vol. 8, no. 2, pp. 56–59, 2009.

[22] A. Baldassin, J. P. L. de Carvalho, L. A. G. Garcia, and R. Azevedo, "Energy-performance tradeoffs in software transactional memory," in *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, 2012, pp. 147–154.

[23] T. M. Rico, M. L. Pilla, and A. R. D. Bois, "Energy consumption on software transactional memories," in *Anais do Simposio em Sistemas Computacionais*, Oct. 2012, pp. 194–201.

[24] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan, "The implications of shared data synchronization techniques on multi-core energy efficiency," in *Proceedings of the 2012 Workshop on Power-Aware Computing and Systems*, Oct. 2012, p. 6.