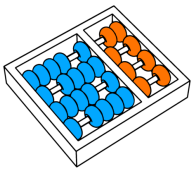
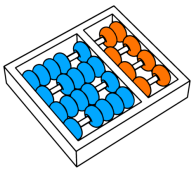


Leonardo Augusto Guimarães Garcia

“Análise e estudo de desempenho e consumo de energia de memórias transacionais em software”

CAMPINAS
2013



Universidade Estadual de Campinas
Instituto de Computação

Leonardo Augusto Guimarães Garcia

"Análise e estudo de desempenho e consumo de energia de memórias transacionais em software"

Orientador: **Prof. Dr. Rodolfo Jardim de Azevedo**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Leonardo Augusto Guimarães Garcia, sob orientação de Prof. Dr. Rodolfo Jardim de Azevedo.

Assinatura do Orientador

CAMPINAS
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

G165a

Garcia, Leonardo Augusto Guimarães, 1981-
Análise e estudo de desempenho e consumo de energia de memórias transacionais em software / Leonardo Augusto Guimarães Garcia. – Campinas, SP : [s.n.], 2013.

Orientador: Rodolfo Jardim de Azevedo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

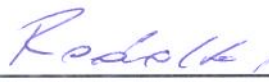
1. Memória transacional. 2. Software - Desempenho. 3. Energia - Consumo. I. Azevedo, Rodolfo Jardim de,1974-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

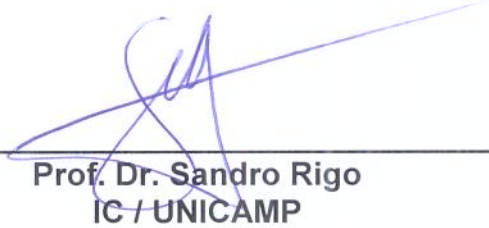
Título em outro idioma: Performance and energy consumption analysis and study on software transactional memories
Palavras-chave em inglês:
Transactional memory
Software - Performance
Energy consumption
Software - Speedup
Energy-delay analysis
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Rodolfo Jardim de Azevedo [Orientador]
Sandro Rigo
Fernando Magno Quintão Pereira
Data de defesa: 22-11-2013
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 22 de novembro de 2013, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP



Prof. Dr. Sandro Rigo
IC / UNICAMP

Prof. Dr. Fernando Magno Quintão Pereira
DCC / UFMG

Instituto de Computação
Universidade Estadual de Campinas

Análise e estudo de desempenho e consumo de energia de memórias transacionais em software

Leonardo Augusto Guimarães Garcia[1]

22 de novembro de 2013

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo (Orientador)
- Prof. Dr. Sandro Rigo
  IC - UNICAMP
- Prof. Dr. Fernando Magno Quintão Pereira
  DCC - UFMG
- Prof. Dr. Edson Borin (Suplente)
  IC - UNICAMP
- Profa. Dra. Alice Maria Bastos Hubinger Tokarnia (Suplente)
  FEEC - UNICAMP

[1]Este trabalho foi parcialmente suportado pela IBM.

Resumo

A evolução das arquiteturas de computadores nos últimos anos, com a considerável introdução de processadores com vários núcleos e computadores com vários processadores, inclusive em máquinas consideradas de baixo poder de processamento, faz com que seja primordial o desenvolvimento de novos paradigmas e modelos de programação paralela que sejam fáceis de usar e depurar pela grande maioria dos programadores de sistemas. Os modelos de programação paralela, atualmente disponíveis, são baseados em primitivas de implementação cujo uso é complexo, tedioso e altamente sujeito a erros como, por exemplo, *locks*, semáforos, sinais, *mutexes*, monitores e *threads*. Neste cenário, as Memórias Transacionais (TM) aparecem como uma alternativa promissora que promete ser eficiente e, ao mesmo tempo, fácil de programar. Muita pesquisa foi feita nos últimos anos em relação às Memórias Transacionais em Software (STM), a maior parte delas relacionada a seu desempenho, com pouca atenção dada a outras métricas importantes, como, por exemplo, o consumo energético e como este se relaciona com o tempo de execução — *energy-delay product* (EDP). Esta dissertação de mestrado faz uma avaliação destas métricas em uma STM configurada com diversas políticas de gerenciamento da TM e de utilização energética, sendo algumas destas combinações inéditas. É mostrado que os resultados para desempenho e EDP nem sempre seguem a mesma tendência e, portanto, pode ser apropriado escolher diferentes políticas de gerenciamento dependendo de qual é o foco da otimização que se deseja fazer, sendo que algumas vezes a execução sequencial pode ser melhor que qualquer execução paralela. De uma forma geral, a execução com o uso de TM foi mais rápida quando comparada com a execução sequencial em dois terços dos casos analisados e teve melhor EDP em um terço das execuções. Através desta análise foi possível derivar um conjunto mínimo de políticas de gerenciamento da TM que foram capazes de entregar melhor resultado para o conjunto de *benchmarks* estudados, além de identificar tendências sobre o comportamento dos sistemas de TM para grupos de *benchmarks* quando se varia o número de núcleos executando em paralelo e o tamanho da carga de trabalho.

Abstract

The recent unveilings on the computer architecture area, with the massive introduction of multi-core processors and computers with many processors in the last years, even in embedded systems, has brought to light the necessity to develop new paradigms and models for parallel programming that could be leveraged and are easy to debug by the majority of the developers. The current parallel programming models are based on primitives whose use is complex, tedious and highly error prone, such as locks, semaphores, signals, mutexes, monitors and threads. In this scenario, the Transactional Memories (TM) appear as a promising alternative which aims to be efficient and, at the same time, easy to program. Lots of research have been made on the past years on Software Transactional Memories (STM), the majority of them interested on the performance of such systems, with little attention given to other metrics such as energy and the relationship between energy and performance, known as the energy-delay product (EDP). This work evaluates these metrics in an STM configured with a number of TM and energy management policies, some of them new. It is shown that performance and EDP do not always follow the same trend, and, because of that, it might be appropriate to choose different management policies depending on the optimization target. It is also important to never forget about the sequential execution, as it can be more advantageous than any parallel execution in some scenarios. Overall, the execution with TM has a better performance when compared to the sequential execution in two thirds of the analysed situations, and a better EDP in one third of the scenarios. Through the analysis made in this work, it was possible to derive a minimal set of TM management policies that were able to deliver the best results with the benchmarks analysed. It was also possible to identify behavioural trends on the TM systems in certain sets of benchmarks when changing the number of cores in the execution and the workload size.

Agradecimentos

Eu gostaria de agradecer, primeiramente, à Deus e à minha família por terem me suportado por toda esta jornada. Francine, minha querida esposa, e Miguel, meu filho, por terem aguentado as demandas e preocupações que este trabalho me trouxeram. Eles certamente foram afetados diretamente por esta jornada. Gostaria, também, de agradecer aos meus pais, Euclides e Deny, que são os principais responsáveis pela pessoa que sou hoje, e à minha irmã, Ana Letícia, a melhor irmã que tenho, por todo o apoio dado durante toda minha vida.

Além disto, é claro que é impossível resumir em uma página todos os agradecimentos que eu gostaria de fazer a todos que me apoiaram nos últimos anos durante todo o processo do mestrado, até porque minha memória certamente não ajudaria nesta hora. Espero, entretanto, que todos aqueles que de alguma forma se envolveram comigo durante este trabalho saibam que sou muito grato e tive muita honra de ter realizado este caminho com vocês, independente de seus nomes estarem explicitamente citados a seguir nesta minha tentativa de lembrar de todos. Diante da vastidão do tempo e da imensidão do universo, foi um imenso prazer para mim dividir esta jornada com vocês. Espero, sinceramente, que o futuro de todos vocês seja por caminhos onde a glória e o reconhecimento sejam sempre presentes!

Agradeço ao povo do estado de São Paulo, por ter me dado a oportunidade de estudar na graduação e no mestrado em uma universidade de ponta. A UNICAMP é parte essencial da minha formação profissional e pessoal.

Agradeço, também a todo o pessoal do LSC, alunos e professores, que estiveram comigo nesta jornada nos últimos anos, que demorou muito mais que o planejado inicialmente, mas que foi extremamente prazerosa. Agradeço especialmente ao Felipe Goldstein por ter me puxado para dentro desta área de pesquisa, e ao Alexandro Baldassin e ao Felipe Klein por terem compartilhado todo seu conhecimento e me ajudado a progredir com meus trabalhos desde o início.

Agradeço à IBM, especialmente aos meus gerentes à época em que iniciei o mestrado, Alcântaro Jovanco e Antero Rivero, que me incentivaram a iniciar esta jornada, proveram mecanismos para que eu tivesse tempo para dedicar ao mestrado e que enxergam o valor que o crescimento acadêmico traz às pessoas além de como pode ser benéfica a colaboração entre academia e indústria.

Destaco, por fim, minha gratidão ao meu orientador, Rodolfo. Primeiro por ter me dado a oportunidade de participar de um programa de mestrado ao mesmo tempo em que continuava minhas atividades na indústria, sempre buscando casar minhas atividades profissionais com tópicos interessantes na área acadêmica. Eu realmente acredito que este tipo de sinergia entre a indústria e a academia, ainda rara no Brasil, é essencial para levarmos este país a um outro patamar de desenvolvimento. Depois, por toda a paciência e dedicação demonstrada nos últimos anos, continuando do meu lado em todos os momentos bons e ruins da evolução do meu trabalho de mestrado. Por toda esta grandeza de espírito, por toda a dedicação e por todo o conhecimento que compartilhou comigo nestes anos, muito obrigado!

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Códigos

Lista de Acrônimos

ACI	Atômica, Consistente e Isolada
AHB	*AMBA High-performance Bus*
AMBA	*Advanced Microcontroller Bus Architecture*
API	*Application Programming Interface*
BSD	*Berkeley Software Distribution*
CMOS	*Complementary metal–oxide–semiconductor*
CPU	*Central processing unit*
DRAM	*Dynamic random-access memory*
DVFS	*Dynamic Voltage and Frequency Scaling*
EDA	*Electronic Design Automation*
EDP	*Energy-Delay Product*
FLOPS	*Floating-point operations per second*
GCC	GNU *Compiler Collection*
GNU	GNU's Not Unix!
GPL	GNU Public License
HTC	*High Throughput Computing*
HTM	*Hardware Transactional Memory*
HyTM	*Hybrid Transactional Memory*
IC	Instituto de Computação

LSC Laboratório de Sistemas Computacionais

NAS *Numerical Aerodynamic Simulation*

NPB *NAS Parallel Benchmarks*

OMP OpenMP

OpenMP *Open Multiprocessing*

RTEMS *Real-Time Executive for Multiprocessor Systems*

SIMD *Single instruction, multiple data*

SPEC *Standard Performance Evaluation Corporation*

SPLASH *Stanford Parallel Applications for Shared-Memory*

SRAM *Static random-access memory*

STAMP *Stanford Transactional Applications for Multi-Processing*

STM *Software Transactional Memory*

SWARM Software ARM

TL2 *Transactional Locking* II

TM *Transactional Memory*

UNICAMP Universidade Estadual de Campinas

Capítulo 1

Introdução

A popularização de processadores com vários núcleos e sistemas computacionais com um ou mais destes processadores nos últimos anos, inclusive em máquinas embarcadas ou consideradas de baixo poder de processamento, ampliou significativamente a distância usualmente existente entre as tecnologias que os softwares sendo desenvolvidos conseguem explorar e aquilo que o hardware disponível consegue entregar.

Apesar de ser usualmente verificado de forma empírica que os softwares, na média, estão ao menos uma geração atrasada em relação àquilo que o hardware oferece, o que pode ser de certa forma entendido, por exemplo, devido à curva de aprendizado necessária para que programadores se acostumem com novas funcionalidades anunciadas em novos processadores, com a introdução massiva de sistemas com diversos núcleos de processamento, esta distância cresceu ainda mais. Enquanto muitos programadores ainda estão mapeando seus programas em estruturas de dados que explorem o paralelismo de dados e os ganhos que podem ser obtidos com instruções SIMD, a possibilidade de se executar múltiplas *threads* simultaneamente fazendo diversos trabalhos aumenta a complexidade não apenas lógica na estruturação de programas, como também aumenta consideravelmente a complexidade de tarefas de análise e correção de problemas em códigos já que, com várias *threads* sendo executadas simultaneamente, a gama de problemas que podem acontecer também cresce consideravelmente.

Novos paradigmas de programação paralela tem surgido para tentar endereçar este problema, tornando mais fácil com que programadores explorem paralelismo de forma mais natural e intuitiva. Dentre estas alternativas, sistemas de Memórias Transacionais (TM) aparecem como uma sugestão promissora. Este trabalho irá explicar como estes sistemas funcionam e irá estudar um aspecto ainda pouco explorado nesta área: o consumo de energia, como ele varia de acordo com parâmetros de execução e como é o relacionamento e o balanço entre esta grandeza e o desempenho de processamento.

1.1 Motivação histórica

Quando se analisa a evolução das arquiteturas de computadores, percebe-se que poucos avanços foram implementados no processo de fabricação dos elementos do núcleo de processamento nos últimos 25 anos. É claro que, neste período, vários avanços microarquiteturais foram desenvolvidos, como a popularização de *branch prediction*, *cache*, processadores superescalares, entre outros. Mas, se por um lado estes avanços puramente microarquiteturais traziam ganhos de poder de processamento, estes ganhos eram obtidos de forma pontual na geração em que estas estruturas eram introduzidas ou melhoradas. O ganho sistemático de desempenho entre gerações de processadores subsequentes se fiava em uma outra técnica.

De meados da década de 1980, quando foi introduzido o processo de fabricação de processadores baseado em CMOS, até o início dos anos 2000, a evolução das arquiteturas de computadores era fortemente atrelada à redução do tamanho do transistor, que possibilitava consequente redução do consumo de energia e aumento da frequência de operação do processador. Apoiados nestes fatos, os fabricantes de processadores eram capazes de aumentar a capacidade de processamento de tempos em tempos sem muito esforço, desde que conseguissem diminuir a escala dos transistores, o que ficou conhecido como Lei de Moore [59, 60]. Este comportamento pode ser facilmente verificado pelo gráfico da evolução das frequências de processamento na figura 1.1.

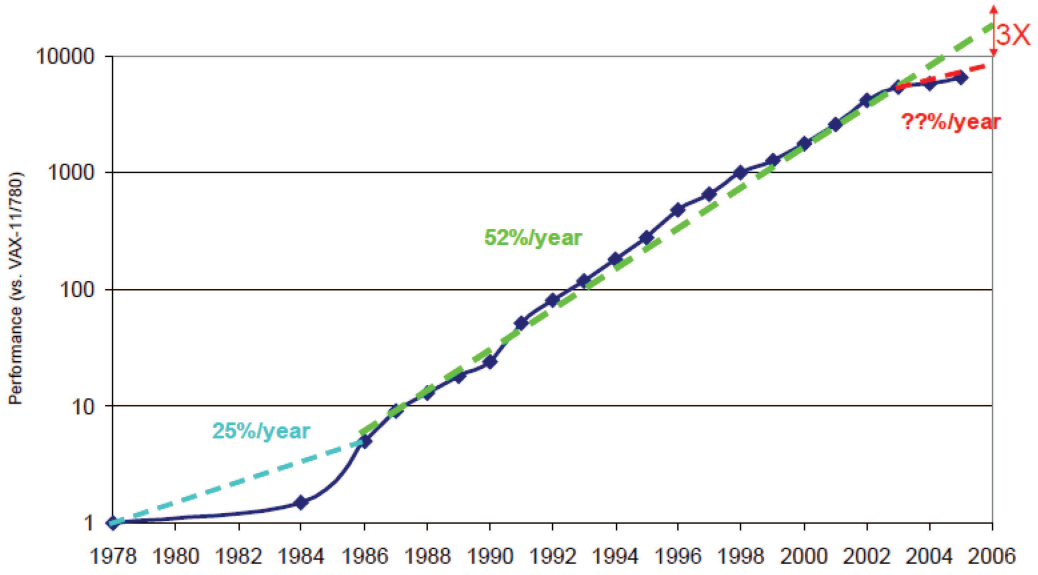


Figura 1.1: Evolução das frequências de operação dos processadores [37].

No entanto, como já era de se esperar, o processo de diminuição do tamanho dos

transistores não poderia, com a utilização da tecnologia de fabricação atual, ser aplicado indefinidamente devido às próprias limitações atômicas dos materiais envolvidos. Desta forma, por volta de 2004, verificou-se uma quebra contundente na evolução dos processadores em relação à sua frequência de operação, como também pode ser verificado na figura 1.1, terminando com o crescimento exponencial do desempenho dos computadores sequenciais [66]. Esta quebra foi consequência de três barreiras físicas encontradas pelos fabricantes de processadores diretamente relacionadas à forma como era feita a evolução das arquiteturas até então: barreira de potência, barreira de memória e barreira de frequência.

A barreira de potência reflete o fato de a dissipação de calor das tecnologias de transistor atuais ter chegado a níveis críticos que atrapalham não só o bom funcionamento dos processadores mas também o uso eficiente da energia elétrica. Este problema é ilustrado pelos dados no gráfico da figura 1.2.

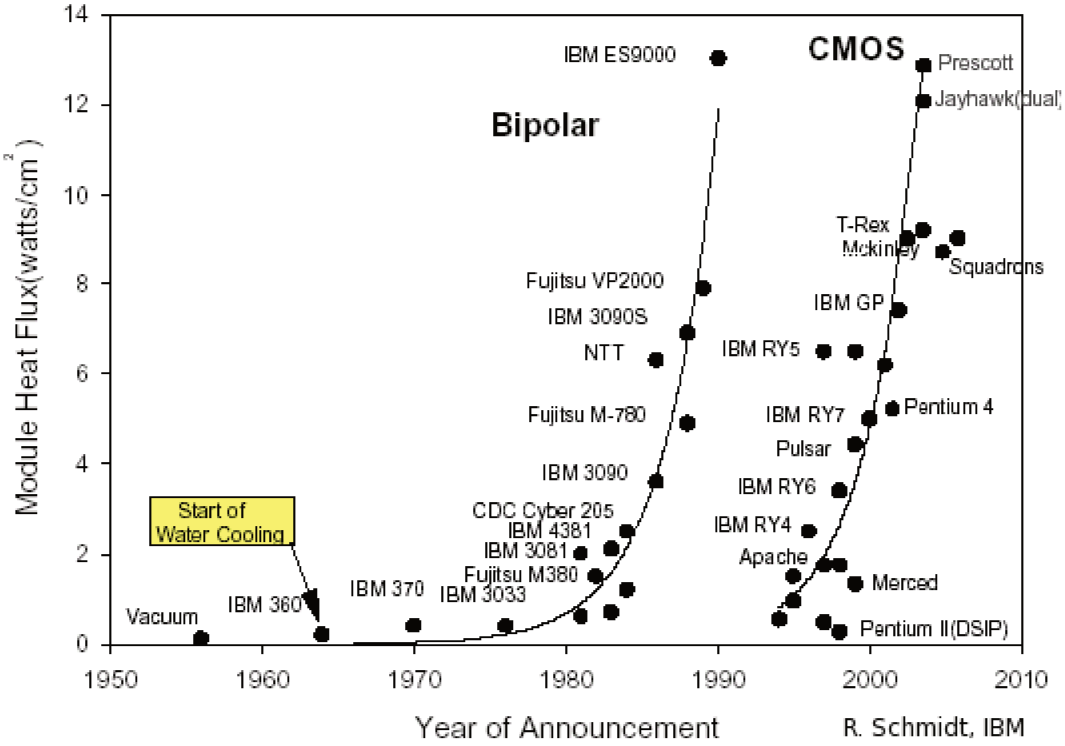


Figura 1.2: Dissipação de calor em processadores [71].

A barreira de memória reflete o fato de a frequência de trabalho dos processadores ter aumentado de forma muito mais rápida que a frequência de trabalho das memórias DRAM. Com isto, formam-se gargalos de processamento em programas que acessam

muito a memória.

Já a barreira de frequência reflete o fato de o simples aumento da frequência de trabalho dos processadores nas arquiteturas atuais causar problemas físicos devido a interferências e dissipação de calor, o que impede maiores ganhos de velocidade com o aumento de *pipelines*, por exemplo.

A saída, diante destas dificuldades, foi mudar o foco da evolução das arquiteturas de processadores. Antes o foco estava todo centrado na diminuição da escala e aumento da frequência de operação de processadores com apenas um núcleo de processamento. Agora, os fabricantes de processadores estão focando em arquiteturas com vários núcleos de processamento (*multi-core*, *many-core* e núcleos especializados).

Processadores com muitos núcleos permitem que, mesmo tendo núcleos com micro-arquiteturas mais simples, a capacidade de processamento conjunta entregue seja maior. Com núcleos mais simples, é possível a operação em alta frequência com menores voltagens, diminuindo o problema de consumo de energia e os problemas ocasionados pela barreira de potência.

Além disto, com diversos núcleos é possível ter um maior proveito de vários níveis de memória. Logo, é comum hoje, em diversas arquiteturas de processadores, vários níveis de *cache* e memória que visam amenizar os problemas ocasionados pela barreira de memória, tentando deixar os dados o mais próximo possível do núcleo onde eles serão utilizados.

Por fim, com núcleos mais simples ou até mesmo com núcleos de processamento especializados (por exemplo, núcleos para processamento criptográfico ou núcleos para processamento gráfico) é possível a operação em frequências mais altas sem que a barreira de frequência se torne um problema pois os circuitos lógicos são mais regulares nestes casos.

Esta mudança fundamental na evolução e na organização dos processadores disponíveis no mercado aumentou consideravelmente a complexidade do hardware. Nos processadores atuais, tem-se muitas estruturas disponíveis simultaneamente para processamento que podem e devem ser utilizadas através de programação paralela. No entanto, a grande maioria dos sistemas de software disponíveis ainda não usufrui de todo o poder paralelo do hardware pois ainda se fia no antigo modelo de evolução e organização das arquiteturas de computadores. Para que os sistemas de software possam usufruir de todas as vantagens providas pelas mudanças dos processadores nos últimos anos, é imprescindível alterar a forma como os sistemas de software são feitos sendo, para isto, primordial a popularização da programação paralela.

O conceito de programação paralela não é novo, com suas origens remontando à década de 1950. Entretanto, diversas dificuldades relacionadas a este modelo fizeram com que ele não se tornasse popular. Por exemplo, o fato de programas paralelos não serem determinísticos [52] faz com que a construção da lógica dos mesmos possa ser muito mais difícil que a de programas sequenciais. Além disto, os programadores mantêm-se na zona

de conforto criada pelo antigo modelo de evolução dos processadores que, de tempos em tempos, sem nenhuma alteração arquitetural considerável, era capaz de entregar mais poder de processamento na única *thread* que era normalmente executada.

Mesmo assim, diversos modelos de programação paralela foram estudados e propostos. Dentre eles destacam-se os modelos *pipeline*, *streaming*, vetorização e *function off-load* entre outros. Estes modelos podem ser agrupados em dois principais grandes grupos: paralelismo de dados e paralelismo de tarefas. O paralelismo de dados se aplica em operações que são executadas simultaneamente em um agregado de itens individuais [39, 40]. Já o paralelismo de tarefas executa computação em *threads* concorrentes que são coordenadas através de sincronização explícita. Uma boa descrição de vários modelos de programação paralela e suas diversas facetas pode ser encontrado em [58].

Todos estes modelos, de certa forma, utilizam-se das mesmas primitivas de implementação: *locks*, semáforos, sinais, *mutexes*, monitores e *threads*. Estas primitivas, apesar de permitirem a programação paralela, quase sempre se traduzem em alternativas complexas, tediosas e muito sujeitas a erros. Outro problema com estas primitivas está no fato de elas não suportarem composição, já que, quando se tenta combinar dois pedaços de código utilizando estas primitivas, é necessário saber como elas são usadas internamente em cada pedaço de código para evitar eventuais conflitos. Além disto, é bem mais difícil implementar ferramentas para a análise de programas concorrentes. Por exemplo, a combinação da análise sensível a contexto – técnica fundamental para analisar programas sequenciais que analisa funções em relação a uma pilha de execução específica – com a análise de sincronização – necessária para entender a comunicação entre *threads* – é um problema indecidível até mesmo quando analisando apenas duas *threads* [68].

Portanto, para que a programação paralela se popularize, é essencial que se explore técnicas de mais alto nível que sejam mais amigáveis aos programadores, tornando o trabalho de paralelizar um programa mais fácil, mais rápido e menos sujeito a erros.

Dentre as alternativas novas propostas, uma vem ganhando um bom destaque nos últimos anos: as Memórias Transacionais (TM). Este destaque se deve à promessa de bom desempenho aliado a facilidade de uso das TM. Uma fonte valiosa de informações sobre o estado-da-arte em sistemas de TM pode ser vista em [30, 49, 50].

1.2 Justificativa

Apesar de, como foi dito na seção 1.1, as tecnologias de processadores atuais serem muito mais eficientes energeticamente quando comparadas com as de alguns anos atrás, entregando muito mais processamento por energia consumida, ainda assim a preocupação com o consumo energético dos computadores é extremamente alta. Vários fatores contribuem para isto. Além da preocupação geral com a preservação de recursos naturais e o bem-

estar de futuras gerações, fatores econômicos também corroboram com esta preocupação. Parte de toda a energia consumida por um computador em seu processamento é dissipada em calor. Além do desperdício energético liberado em calor, este calor ainda precisa ser tratado para não danificar o próprio equipamento. Portanto, para uma infraestrutura adequada, além de se gastar dinheiro para abastecer os computadores com energia, gasta-se também muito dinheiro para refrigerar o ambiente adequadamente e manter os computadores em operação, a ponto de estudos terem apontado que o gasto em refrigeração projetado para o final de 2009 seria maior que o gasto com a compra de novos equipamentos, como pode ser visto na figura 1.3. Apesar de estes estudos não terem se confirmado totalmente, este problema torna-se ainda mais grave se pensarmos que existe uma tendência, nos últimos dez anos, de migração da utilização de máquinas isoladas para o uso de máquinas instaladas em *racks* ou até mesmo em chassis construídos de forma a aumentar drasticamente a densidade computacional por volume ocupado [46], já que a utilização de espaço em centros de processamento de dados também é outro problema crescente. Soma-se, ainda, a isto, o fato de que hoje em dia é cada vez mais comum a consolidação de vários servidores em uma única máquina física através de virtualização, o que acaba fazendo com que as máquinas tenham uma taxa de ocupação de processamento, na média, bem mais alta do que aquela que era encontrada nas máquinas que executavam apenas um sistema ou aplicativo. Estes movimentos intensificam ainda mais o problema de dissipação de calor, já que com mais máquinas por unidade de volume com uma maior taxa de utilização média, maior é o calor acumulado no ambiente e, consequentemente, maiores são os gastos com refrigeração.

O aspecto de desempenho por Watt está tão em voga nos últimos tempos que até mesmo uma hierarquia específica para ele foi criada, o Green500 [3]. Neste *site* pode-se encontrar a lista dos supercomputadores mais energeticamente eficientes do mundo. Esta lista é construída com base nos dados coletados pelo Top500 [7], que lista os computadores com maior poder de processamento do mundo, contrastando este poder de processamento com o consumo de energia. Quanto melhor a relação entre FLOPS e Watt, melhor a colocação do supercomputador no Green500.

Apesar de toda esta importância dada à eficiência energética nos dias de hoje, ainda são relativamente poucos os esforços no campo da ciência da computação para o estudo adequado do desempenho energético de sistemas e aplicações quando comparados com os vastos esforços dispendidos com a análise de desempenho relacionada ao poder de processamento de sistemas e aplicações.

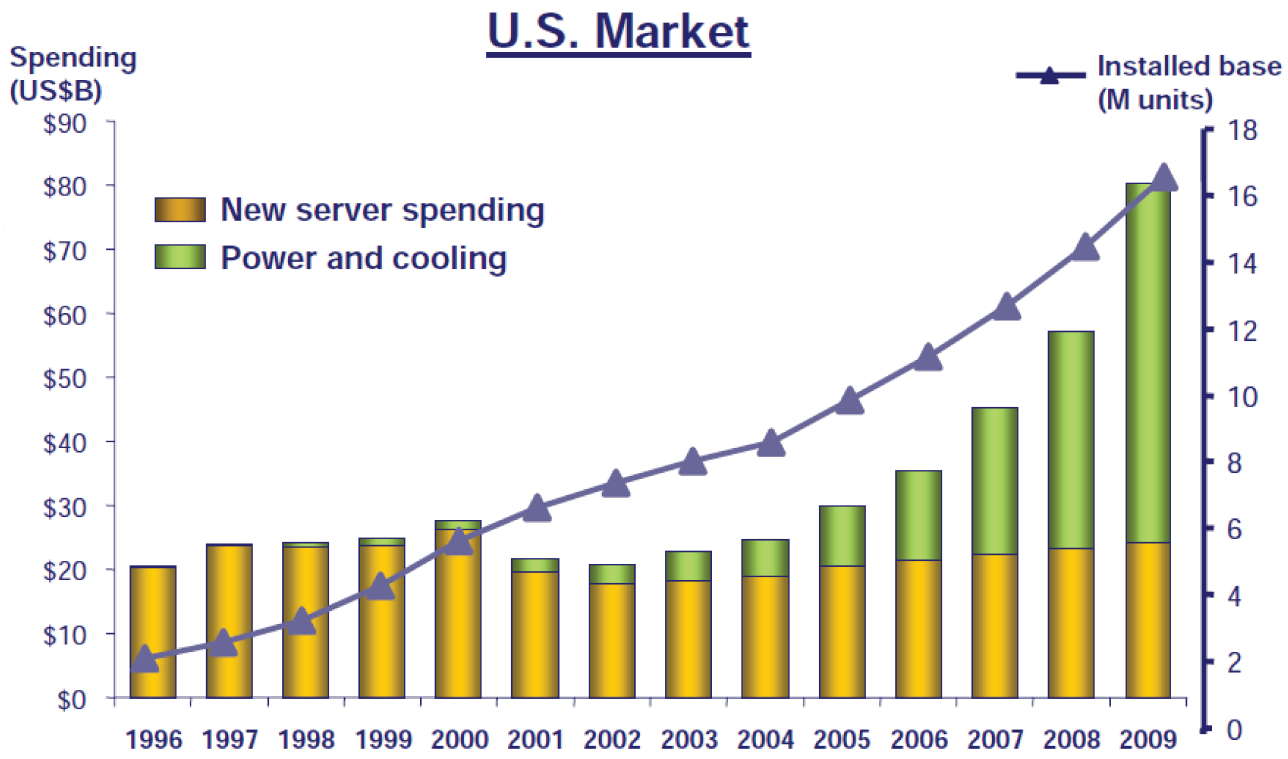


Figura 1.3: Gastos financeiros em refrigeração e novos equipamentos frente ao número de computadores instalados nos Estados Unidos da América [46].

1.3 Foco do trabalho e contribuições

Muita pesquisa foi feita nos últimos anos em relação às TM, especificamente às Memórias Transacionais em Software (STM), que serão o foco principal desta dissertação. A maior parte destas pesquisas, assim como em outras áreas da ciência da computação, é relacionada ao desempenho de processamento das TM, com pouca atenção dada a outras métricas importantes como, por exemplo, o consumo energético e a relação deste com o desempenho de processamento.

Esta dissertação de mestrado faz uma avaliação sobre o consumo de energia, o desempenho de processamento e a relação e o balanço entre estas duas grandezas em uma STM configurada com diversas políticas de gerenciamento da TM e de utilização energética, sendo algumas destas combinações inéditas, estendendo estudos anteriores registrados em [12, 13]. Para a condução deste estudo, cada uma das políticas de gerenciamento da TM foi analisada para diversos *benchmarks* executando em um sistema configurado com um, dois, quatro ou oito núcleos de processamento. Os resultados obtidos foram todos normalizados em relação aos resultados obtidos para a execução sequencial dos *benchmarks*

para que se tivesse um referencial único de comparação para cada caso.

Como já era de se esperar, não foi encontrada uma política de gerenciamento da TM que fosse mais vantajosa em todos os casos ou mesmo cenários patológicos que prejudicavam a execução em todos os *benchmarks* analisados, desde que se utilizando as políticas de gerenciamento da TM validadas como viáveis neste trabalho. Para cada aplicação, com seu conjunto de dados, alguns aspectos foram mais relevantes. Após uma detalhada análise do comportamento de cada uma das políticas de gerenciamento da TM propostas neste trabalho, para cada um dos *benchmarks* estudados, foi possível identificar e agrupar características comuns entre os diversos casos analisados. Desta forma foi possível tirar conclusões sobre quais configurações podem ou não ser mais vantajosas para determinados tipos de aplicações. Além disto, métricas diferentes levam a conclusões diferentes. Cenários que entregam um melhor *speedup* podem não ser os mesmos que entregam a melhor relação entre consumo de energia e tempo de processamento. Esta distinção deve ser levada em consideração quando se escolher a política e o número de núcleos executando a ser utilizado. Usar mais núcleos pode trazer um melhor *speedup*, mas o gasto de energia adicional associado ao aumento do número de núcleos executando pode prejudicar sobremaneira o balanço entre consumo de energia e tempo de execução a ponto de tornar esta nova configuração desfavorável sob alguns aspectos. Para determinados casos, inclusive, a paralelização do algoritmo pode se mostrar não muito vantajosa. Nestes casos, a recomendação é continuar executando os algoritmos sequencialmente, tentar alterar a forma como a paralelização do código foi implementada ou, eventualmente, procurar uma maneira diferente de paralelizar o algoritmo.

De uma forma geral espera-se, ao paralelizar um algoritmo, que o tempo de execução da aplicação diminua em relação à execução sequencial e que o gasto de energia não cresça muito. No entanto, isto nem sempre ocorre na prática. Muitas vezes, inclusive, presenciam-se comportamentos conflitantes, em que ao se aumentar o número de núcleos executando aumenta-se o consumo de energia — como seria esperado — mas diminui-se o *speedup* — ao contrário do que se esperaria nesta situação.

Através das análises feitas, foi possível concluir que, para a plataforma utilizada neste trabalho, o *speedup* aumenta com o aumento do número de núcleos executando com exceção para os casos de execução com 8 núcleos para uma grande parte das políticas de gerenciamento da TM analisadas. Este comportamento espalhado por quase todos os *benchmarks* indica mais do que um possível desajuste entre políticas de gerenciamento da TM estudadas e a execução com 8 núcleos. O fato de que as execuções com 8 núcleos em geral apresentam decréscimo de *speedup* indica que a infraestrutura utilizada está sendo saturada com esta quantidade de núcleos executando.

Da mesma forma, pode-se fazer uma análise para a relação entre o consumo de energia e o tempo de execução dos *benchmarks*. Esta relação é mais favorável quanto maior é o

número de núcleos executando, com exceção para quase todos os casos de execução com 8 núcleos.

Resumindo, para a plataforma utilizada nesta dissertação, os melhores *speedups* estão bem divididos entre execuções com quatro e oito núcleos. Já as melhores relações entre consumo de energia e tempo de execução foram encontradas em execuções com 4 núcleos. De uma forma geral, portanto, sem saber muito sobre o *benchmark* a ser executado, as chances de se obter melhor *speedup* e melhor relação entre consumo de energia e tempo de execução ao se utilizar um sistema de TM para controlar o acesso aos dados compartilhados em uma execução paralelizada na plataforma usada neste estudo seriam maiores caso o *benchmark* fosse executado em 4 núcleos.

Caso fosse analisada a melhor combinação de quantidade de núcleos executando e política de gerenciamento da TM para cada *benchmark* estudado, incluindo nesta análise as execuções sequenciais dos *benchmarks*, seria possível concluir que a execução sequencial é a mais rápida em um terço dos casos e tem a melhor relação entre consumo de energia e tempo de execução em dois terços das execuções.

Olhando mais de perto, STM mostrou-se desfavorável principalmente nos casos de aplicações com alta contenção. Estas aplicações eventualmente teriam problemas caso outras metodologias de paralelização de código fossem utilizadas também, especialmente se não fosse tomado o devido cuidado na análise de onde o código deveria ser paralelizado. Por outro lado, para aplicações com baixa contenção, o uso de TM foi vantajoso mesmo utilizando-se STM.

Mesmo assim, considerando-se que sistemas de TM podem ter implementações em hardware e considerando que estas mesmas terão um comportamento similar ao de uma STM porém com maior desempenho, pode-se utilizar os resultados obtidos para STM como forma de análise de tendências sobre o que ocorrerá quando as mesmas políticas forem implementadas em um sistema de HTM. Neste sentido, as políticas de gerenciamento da TM mais vantajosas dentre as que foram estudadas neste trabalho são as seguintes:

- `BACKOFF_LINEAR Eager`
- `BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager`
- `BACKOFF_EXPONENTIAL Eager` e `BACKOFF_LINEAR Eager` combinadas com:
  - `WAIT_ABORTER_DVFS`
  - `WAIT_ABORTER_ENABLED`
  - `DVFS_COMMIT_BEST_BALANCE`

1.3.1 Publicações

O trabalho desta dissertação está ligado a outros trabalhos publicados em conferências e escolas:

- *A Software Transactional Memory System for an Asymmetric Processor Architecture.*
  Felipe Goldstein, Alexandro Baldassin, Paulo Centoducatte, Rodolfo Azevedo e Leonardo A. G. Garcia.
  Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'08), pg 175–182, Outubro de 2008.
  Este artigo foi agraciado com o prêmio Júlio Salek Aude de melhor artigo entre todos os apresentados no SBAC-PAD'08.

- *Análise de desempenho e consumo de energia em Memórias Transacionais.*
  Leonardo A. G. Garcia, Alexandro Baldassin e Rodolfo Azevedo.
  Poster apresentado na III Escola Regional de Alto Desempenho de São Paulo (ERAD-SP 2012).

- *Energy-Performance Tradeoffs in Software Transactional Memory.*
  Alexandro Baldassin, João P. L. de Carvalho, Leonardo A. G. Garcia e Rodolfo Azevedo.
  Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2012).

1.4 Organização do texto

Esta dissertação está organizada em sete capítulos. O capítulo 2 discute os conceitos básicos relacionados aos sistemas de TM e elenca uma série de trabalhos relacionados, seja porque seguem uma linha de estudo similar, seja porque são trabalhos que servem de base para o trabalho desenvolvido nesta dissertação.

O capítulo 3 explica quais foram as variações de políticas de TM exploradas na parte experimental deste trabalho. Já o capítulo 4 descreve os experimentos que foram feitos em cima destas políticas para obtenção dos dados das simulações.

Os resultados das simulações, assim como uma análise das políticas de TM propostas e não viáveis são apresentados no capítulo 5. Já uma análise mais aprofundada das políticas de TM propostas e viáveis é apresentada no capítulo 6.

As conclusões, assim como uma discussão de possíveis trabalhos futuros, são encontradas no capítulo 7.

Capítulo 2

Conceitos básicos e trabalhos relacionados

Como visto no capítulo 1, os sistemas de TM foram propostos como um modelo de programação paralela que visa ser mais amigável ao programador, explorando técnicas de mais alto nível que tornem a programação paralela mais fácil, mais rápida e menos sujeita a erros.

Os estudos de TM, assim como vários estudos no campo da ciência da computação, tem focado majoritariamente em como algoritmos e sistemas computacionais de hardware e software podem entregar mais desempenho em relação à diminuição do tempo de execução, com poucos esforços sendo relativamente dedicados ao estudo adequado do desempenho energético.

2.1 Conceitos básicos

Os sistemas de TM foram propostos como uma alternativa de modelo de programação que visava facilitar a paralelização da execução de código. Eles também podem ser utilizados para recuperação de erros, programação em tempo-real ou programação multi-tarefa, mas este não tem sido o foco nos últimos anos. Apesar de esta técnica não ser propriamente nova, tendo sido proposta, mas sem sugestão de implementação, por Lomet, em 1977 [56], ela só voltou à tona em 1993, com Herlihy e Moss [38]. Mais recentemente, com a popularização dos processadores *multicore*, esta alternativa tem ganhado cada vez mais força, principalmente no campo da programação paralela.

Estes sistemas funcionam baseados em transações. Uma transação é um conjunto de instruções cuja execução atômica, consistente e isolada (ACI) é garantida. Mais especificamente, as transações em memória garantem as propriedades ACI para as instruções de acesso à memória (leituras e escritas), garantindo que o acesso a dados compartilhados

não gere resultados inconsistentes ou incorretos. Caso alguma parte da transação não possa ser executada com sucesso, ela toda é abortada e seu processamento, se for o caso, é reiniciado. Quando uma transação é completada, existe a garantia de que toda ela foi executada com sucesso e de que seu resultado, devido à propriedade de isolamento, é o mesmo que seria obtido se nenhuma outra transação estivesse executando concorrentemente. Para um observador externo, é como se a transação fosse uma única operação indivisível e instantânea.

Assim como no conceito largamente conhecido de transação usado em bancos de dados, as transações em memória também são utilizadas para facilitar o controle de concorrência. Caso dois trechos de código estejam acessando uma mesma área de memória de maneira conflitante, por exemplo se ambos os trechos estiverem escrevendo na mesma região de memória, uma das transações deverá ser abortada e apenas uma delas poderá prosseguir com sua execução. Desta forma, garante-se que apenas uma transação altera uma determinada região de memória por vez, garantindo, assim, a coerência e a consistência da memória. No entanto, ao contrário do que acontece com as transações em bancos de dados, no caso de programas, as transações em memória podem ter que conviver com outras partes de código que acessam dados compartilhados e não são delimitadas por transações. Este tipo de situação faz com que as propriedades ACI não sejam suficientes para especificar um sistema de TM completamente, como será visto mais adiante nesta seção.

O controle de concorrência nos sistemas de TM é feito através de pequenos trechos de códigos de programa ou instruções de hardware que são executados em pontos específicos da transação. Estes pontos ficam, pelo menos, logo antes do início da transação e após a finalização da mesma. No início da transação, é necessário avisar de alguma forma o sistema de TM que uma transação irá se iniciar. Esta operação, neste trabalho, será chamada de `TxStart`. Quando a transação é finalizada, mais uma vez, roda-se um conjunto de instruções que verifica a validade da transação e persiste os resultados em memória caso nenhum conflito tenha ocorrido. Esta operação, chamada de *commit*, também será referenciada nesta dissertação como `TxCommit`. Caso durante a transação ou durante a operação `TxCommit` seja identificado um conflito, uma operação irá abortar a transação e tomar as medidas necessárias. Esta operação será referenciada neste trabalho como `TxAbort`. Outras operações podem ser encontradas dependendo do sistema de TM sendo estudado mas pelo menos estas três operações estão sempre presentes e serão utilizadas em exemplos neste texto.

A facilidade provida pelos sistemas de TM vem do fato de que todo o controle de concorrência é feito pelo sistema. Ao programador basta delimitar o início e o fim das transações no código do programa e continuar pensando e escrevendo código sequencial, apenas se preocupando em como encadear as várias transações. Um conjunto básico de

extensões para linguagens de programação que adiciona um suporte básico a TM nestas linguagens pode ser visto em [18, 33, 35]. Além desta alternativa, a TM também pode ser implementada através do uso de bibliotecas que provem a funcionalidade do sistema de TM, como é o caso do sistema de TM que foi utilizado neste trabalho e que será melhor descrito na seção 2.2. O programador, como foi dito, independente de o sistema de TM ser implementado através de uma biblioteca ou através de extensões da linguagem de programação, apenas deve delimitar o início e o fim das transações. A implementação do controle de concorrência, feito pelo sistema de TM, pode ficar a cargo de uma biblioteca em software, de um hardware especializado, de uma mistura destas coisas, ou até mesmo de instruções específicas para TM que vem sendo inseridas em algumas microarquiteturas de propósito geral [31, 43]. Esta facilidade proporciona grande agilidade ao programador, que pode passar a se preocupar mais com a lógica de seu programa do que com os mecanismos de controle de paralelismo: com TM todos os pedaços do código executam de forma isolada.

Quando comparada com as primitivas de implementação de paralelismo indicadas na seção 1.1, as TM provem ao menos uma grande vantagem: nelas não é necessário nomear os recursos compartilhados, os dados ou os mecanismos de sincronização. Isto é bem diferente, por exemplo, do que ocorre quando se usa monitores [16, 41], em que o programador explicitamente nomeia os dados protegidos pela seção crítica, ou *locks*, em que o programador nomeia os dados que estão protegendo o mecanismo de sincronização.

As TM também suportam composição: caso duas transações interajam uma com a outra, não é necessário que elas saibam detalhes de implementação do controle de concorrência que ambas utilizam para que o funcionamento do programa seja correto.

Além disto, o não uso das primitivas de implementação de paralelismo facilita bastante a depuração de programas paralelos já que, como foi dito antes, elas são, em geral, complexas, tediosas e muito sujeitas a erros. Isto não quer dizer que depurar programas em cima de sistemas de TM não seja complexo [54]; a vantagem está mais na menor chance de se cometer erros durante o desenvolvimento com este paradigma.

Por todas estas características, o objetivo principal buscado com os sistemas de TM é eles serem tão fáceis de programar quanto *locks* de granularidade alta ao mesmo tempo em que proporcionem um desempenho tão bom quanto o que poderia ser obtido utilizando-se *locks* de granularidade fina.

No entanto, os sistemas de TM não resolvem todos os problemas para o programador. Ainda é necessário que ele ache os pontos corretos para iniciar e terminar uma transação. Caso a escolha destes pontos seja feita de forma incorreta ou displicente, é fácil obter exemplos de programas que podem entrar em estados inconsistentes ou não funcionarem por completo, como pode ser visto em [15, 26], mesmo com o uso de TM. Por exemplo, apesar de TM impedir a ocorrência de *deadlocks*, ainda é possível ter situações de *livelock*

com este tipo de controle de concorrência.

Além disto, muitos sistemas de TM atuais não permitem que código transacional faça entrada/saída de dados com entidades não controladas pelo sistema de TM, já que isto também afeta o estado do programa e o controle da transação. Caso estas operações ocorram, o sistema de controle transacional pode ficar muito mais complexo.

Outra limitação está relacionada ao fato já citado anteriormente de que as propriedades ACI não são suficientes para especificar um sistema de TM quando transações em memória convivem com outras partes de código que acessam dados compartilhados e não são delimitadas por transações. Programas com este tipo de acesso não-transacional podem conter problemas de concorrência no acesso aos dados da memória [10, 27, 65]. Códigos contendo este tipo de conflito possuem comportamento indefinido e dependente da implementação do modelo de consistência de memória sobre o qual ele está executando. Como este tipo de comportamento, em geral, indica um erro no programa, estes possíveis conflitos devem ser evitados.

Desta forma, é possível diferenciar os sistemas de TM com isolamento fraco e isolamento forte [15]. Em sistemas com isolamento fraco, código não-transacional que referencia dados acessados por uma transação pode gerar problemas de concorrência. Já em sistemas com isolamento forte, o código não-transacional é convertido em operações transacionais, garantindo que não ocorrerão problemas de concorrência.

Uma outra variação implementada em alguns sistemas de TM é a identificação dos dados compartilhados nas transações. Isto permite otimizações e melhora o desempenho já que acesso a dados privados não precisam ser controlados pelo sistema [9, 56]. Isto, no entanto, pode fazer com que o programador esqueça de declarar um dado a ser compartilhado e gere os problemas de concorrência citados anteriormente. Por outro lado, alguns tipos de análises de código feitas por compiladores podem, de maneira conservadora, identificar dados que não são compartilhados entre *threads* [14, 36].

Os sistemas de TM geralmente tratam uma transação simples como algo semelhante a um *lock* simples. Transações com encadeamentos simples poderiam necessitar o uso de múltiplos *locks* recursivos. Uma questão ainda em aberto em relação às TM é como tratar transações encadeadas de formas mais complexas. Modelos de como tratar o encadeamento de transações em sistemas de TM podem ser encontrados em [63, 75].

Outro aspecto de variação entre os sistemas de TM já propostos é a granularidade da transação, que é o tamanho da unidade de armazenamento sobre a qual é possível detectar um conflito. Sistemas totalmente baseados em software muitas vezes estendem linguagens de programação orientadas a objeto e, por causa disto, a unidade mínima de detecção de conflitos é um objeto. Para sistemas em que parte ou todo o controle é feito no hardware esta unidade pode ser uma palavra na memória ou até mesmo uma linha de memória ou de *cache*, dependendo da implementação do sistema.

Além disto, os sistemas também podem apresentar diferentes métodos de atualização dos dados alterados durante uma transação. Em alguns sistemas, esta atualização é feita diretamente na posição de memória sendo alterada (*direct update*). Neste caso, o sistema de TM precisa, de alguma forma, guardar a informação previamente presente nesta área da memória para que ela seja recuperada no caso de a transação abortar. Por outro lado, em outros sistemas, as transações fazem uma cópia local dos dados que elas alteram e, quando a transação é finalizada, ela atualiza o dado original caso não tenha ocorrido nenhum conflito (*deferred update*). Para cada caso, um conjunto diferente de procedimentos de detecção de concorrência devem ser adotados.

Devido a esta grande variedade de aspectos e decisões relacionadas à construção de um sistema de TM, uma grande variedade deles e de suas características foram propostas até hoje. Alguns exemplos podem ser vistos em [8, 9, 18, 22, 23, 28, 29, 32, 34, 35, 53, 56, 57, 70, 75].

2.2 Trabalhos relacionados

Esta dissertação de mestrado faz uma avaliação sobre o consumo de energia e o desempenho de processamento em uma STM configurada com diversas políticas de gerenciamento da TM e de utilização energética, sendo algumas destas combinações inéditas, estendendo estudos anteriores registrados em [12, 13]. Especificamente, neste trabalho também é utilizada a estratégia de *Dynamic Voltage and Frequency Scaling* (DVFS) nos sistemas de TM, como foi feito em [13], para investigar como as variações de estratégias de DVFS, aliadas a outras estratégias propostas neste trabalho, afetam a eficiência energética dos sistemas de TM.

Trabalhos similares investigaram o consumo de energia para implementações de Memórias Transacionais em hardware (HTM) [25, 61, 62]. Estes trabalhos sugerem que TM possuem vantagens sobre *locks*, porém eles são focados apenas em *microbenchmarks*. Nesta mesma linha de análise e comparação de eficiência energética, [48] estuda o consumo energético entre uma STM e a programação paralela utilizando primitivas de *lock*. O trabalho desta dissertação, ao contrário destes, está focado na comparação do consumo energético entre várias configurações de uma STM.

Um dos aspectos mais importantes de se discutir quando se avalia e compara diferentes sistemas computacionais que possuem o mesmo propósito é qual pacote de programas está sendo utilizado para se obter as medidas de desempenho. Isto porque, dependendo dos programas utilizados, um ou outro sistema pode ser beneficiado por ser mais eficiente sob determinadas condições que podem ser características de apenas alguns programas. Desta forma, é necessário ter um pacote de programas, também chamado de pacote de *benchmarks*, balanceado, que seja bastante representativo do conjunto total de aplicações

que se deseja executar no sistema sendo analisado.

Apesar de sistemas de TM serem promissores para simplificação da programação paralela e de vários sistemas terem sido propostos, a não existência de ferramentas e cargas de trabalho na forma de conjuntos de *benchmarks* adequados para analisá-los e compará-los ainda é um problema.

Sistemas de TM têm sido avaliados utilizando-se *microbenchmarks*, que não são representativos de aplicações reais, ou aplicações isoladas transformadas de *benchmarks* paralelos, como SPEC OMP2001 [11], SPLASH-2 [73] , NPB OpenMP [45], BioParallel [44], MediaBench [51] ou MineBench [64]. Estes *benchmarks* paralelos, em geral, não estressam uma grande variedade de cenários de execução sendo, muitas vezes, focados em computação de alto desempenho. Além disto, eles, em geral, são altamente otimizados para minimizar sincronização e comunicação entre *threads*, resultando em programas que raramente utilizam transações e que não representam como programadores usariam transações. O maior potencial de sistemas de TM, como foi visto, está na possibilidade de se escrever código paralelo com uso frequente de transações de granularidade alta que tenha desempenho tão bom quanto um código cuidadosamente otimizado com *locks* de pequena granularidade.

O *Stanford Transactional Applications for Multi-Processing* (STAMP) [17], um conjunto de *benchmarks* públicos para avaliação de sistemas de TM, aparece como uma alternativa promissora neste sentido. Com três objetivos em mente — (i) variedade de algoritmos e domínios de aplicação que se beneficiem de TM, (ii) uma grande variedade de características transacionais, como tamanho e duração relativa das transações e tamanho dos conjuntos de leitura e escrita e (iii) compatibilidade com um grande número de sistemas de TM baseados em hardware, software ou híbridos — o STAMP apresenta os algoritmos, estruturas de dados, estratégias de paralelização e de uso de transações, além da caracterização transacional, de 8 aplicações escritas em C com anotações de TM em macros C para facilitar o porte para os diversos sistemas de TM. Uma análise qualitativa do STAMP mostra que ele cobre uma gama interessante de cenários para TM, como pode ser visto na tabela 2.1. Por outro lado, a análise quantitativa do STAMP confronta o senso comum geralmente aceito em relação ao desempenho de TM que diz que HTM pode ser até 4 vezes mais rápida que STM e até 2 vezes mais rápida que HyTM e que *Eager* TM é mais rápido que *Lazy* TM. Executando em máquinas de um a dezesseis núcleos através de simulação 20 variações do pacote de programas em 6 sistemas (*Lazy* HTM, *Eager* HTM, *Lazy* STM, *Eager* STM, *Lazy* HyTM e *Eager* HyTM, cada um com várias características próprias), mostra-se que este senso comum em relação ao desempenho de TM nem sempre é válido. Uma explicação detalhada sobre o significado de HTM, STM, HyTM, assim como das características *Lazy* e *Eager* poderá ser encontrada no capítulo 3.

Tabela 2.1: Análise qualitativa das características de execução transacional das aplicações do STAMP. A descrição de cada característica é relativa às outras aplicações do STAMP [17].

Aplicação	Tamanho da transação	Conjunto de leitura/escrita	Tempo transacional	Contenção
`bayes`	Longa	Grande	Longo	Alta
`genome`	Média	Médio	Longo	Baixa
`intruder`	Pequena	Médio	Médio	Alta
`kmeans`	Pequena	Pequeno	Pequeno	Baixa
`labyrinth`	Longa	Grande	Longo	Alta
`ssca2`	Pequena	Pequeno	Pequeno	Baixa
`vacation`	Média	Médio	Longo	Baixa/Média
`yada`	Longa	Grande	Longo	Média

Como dito no parágrafo anterior, um dos objetivos do STAMP foi ter, dentro do pacote de *benchmarks*, uma variedade de algoritmos e domínios de aplicação que se beneficiem de TM, ao mesmo tempo em que uma amostra qualitativa do pacote de *benchmarks* mostra que ele cobre uma gama interessante de cenários de TM. Esta é uma preocupação que pode ser vista tanto no STAMP quanto em outros pacotes de *benchmarks* muito utilizados: mesmo tendo um bom conjunto de *benchmarks* para avaliação de TM, deve-se pensar no quão balanceado e representativo dos vários cenários possíveis este conjunto é. Um bom exemplo deste tipo de análise pode ser encontrada em [67] para o conjunto de *benchmarks* SPEC CPU2006, talvez o pacote de *benchmarks* mais utilizado para avaliação de arquiteturas de processadores. O SPEC CPU2006 é formado por um conjunto de programas inteiros em C e C++ (CINT2006) e por um conjunto de programas de pontos flutuantes em C, C++ e Fortran (CFP2006). Porém, restrições no tempo disponível para simulações, problemas com compiladores, bibliotecas e chamadas de sistema fazem com que muitas vezes apenas parte dos programas seja usada, o que pode levar a análises e conclusões incorretas. Uma análise estatística de um conjunto de características dependentes de microarquitetura para os programas e as suas respectivas entradas disponíveis no SPEC CPU2006 chegou, para o CINT2006, a um subconjunto representativo de 6 programas com apenas uma entrada cada de um total de 12 programas, muitos com diversas entradas. Através da análise de variáveis concluiu-se que este subconjunto agrega 94% da variância das características dos *benchmarks*. Para o CFP2006, uma análise similar, chegou a um subconjunto de oito de dezessete programas, agregando 85% da variância. Estes subconjuntos representam a maior parte da informação provida por todo o con-

junto de *benchmarks*. Usando-se o subconjunto de *benchmarks* obtido desta análise, o erro nas medidas em relação ao conjunto completo tem média 3,8% e não é maior que 8% no CINT2006. Já no CFP2006, o erro tem média 7% não sendo maior que 12%. Outro aspecto abordado nesta análise é que, apesar de algumas áreas possuírem vários programas no SPEC CPU2006, como, por exemplo, 3 programas de Inteligência Artificial no CINT2006 e 4 de Dinâmica de Fluidos no CFP2006 e outras, como *Electronic Design Automation* (EDA), não terem representantes, isto não significa que o conjunto de *benchmarks* seja desbalanceado. A ausência de programas EDA, presentes em versões anteriores do SPEC CPU, é compensada por outros programas que possuem características similares e até mesmo mais abrangentes que os programas EDA anteriormente presentes. Além disto, programas da mesma área muitas vezes possuem características de execução diferentes. Em resumo, de uma forma geral, o SPEC CPU2006 é mais abrangente que seus predecessores.

No caso do STAMP, não existe uma análise tão detalhada quanto a encontrada para o SPEC CPU2006 em [67]. Portanto, neste trabalho, todas as aplicações disponíveis no STAMP foram executadas a fim de cobrir o maior espectro possível na análise sendo feita. Além disto, até onde se sabe, o STAMP é o único pacote de *benchmarks* com cargas de trabalho genéricas focado em sistemas de TM. Portanto, apesar de várias dificuldades com ele (desde problemas na compilação dos programas até problemas com a corretude dos mesmos, que foram solucionados, como será mostrado na subseção 4.2.2), este é o pacote de *benchmarks* que será utilizado para a avaliação de TM nesta dissertação.

Dentro do STAMP, como dito anteriormente, as operações de TM são anotadas nos códigos dos programas que compõem o pacote de *benchmarks* através de macros C. Estas macros podem ser mapeadas em diversos sistemas de TM. Especificamente, estas macros podem ser mapeadas em sistemas de TM implementados em hardware, software ou híbridos. Uma melhor explicação sobre estes poderá ser encontrada no capítulo 3. No caso desta dissertação, o foco será nas STM. A implementação em software do sistema de TM utilizada neste trabalho foi a TL2 [20]: uma biblioteca que implementa um sistema de STM que é distribuída junto com o STAMP e, portanto, casa perfeitamente com as anotações em macros C que são usadas nos programas que compõem o pacote de *benchmarks* do STAMP. Maiores detalhes sobre o STAMP e a TL2 poderão ser encontrados na seção 4.2.

O objetivo deste trabalho é, portanto, executar os *benchmarks* do STAMP sobre uma plataforma específica — foi escolhida a plataforma de simulação MPARM [55] (ver seção 4.1) — e verificar como diversas políticas de gerenciamento da TM comportam-se para diversas configurações do ambiente simulado com diferentes número de núcleos executando em relação ao tempo de execução da simulação, o consumo de energia do sistema e a relação entre estas duas grandezas, medida através do *energy-delay product* EDP [42],

produzindo, assim, uma análise mais ampla do que aquelas feitas anteriormente pelos trabalhos citados no início desta seção, como pode-se ver na tabela 2.2.

Na tabela 2.2, a coluna "Sincronização" indica qual mecanismo de sincronização foi utilizado no trabalho indicado. Já as colunas "*Speedup*", "Energia" e "EDP" indicam, respectivamente, se no trabalho indicado foi feita uma análise sobre desempenho de execução, consumo energético e EDP com todas as simulações executadas ou com apenas alguns casos. Já a última coluna, "Políticas", mostra quais variações de políticas de gerenciamento da TM ou de outros mecanismos de sincronização usados nos trabalhos indicados foram utilizadas.

Tabela 2.2: Características de trabalhos relacionados.

Referência	**Sincronização**	**Tipo de TM**	**Simulador**	***Speedup***	**Energia**	**EDP**	***Benchmarks***	**Número de núcleos**	**Políticas**
[61]	TM e *locks*	HTM	Simics	Não	Sim	Não	*microbenchmarks* próprios	4	2 políticas de TM e *locks*
[62]	TM e *locks*	HTM	Simics	Sim	Sim	Não	SPLASH-2 e *microbenchmark* próprio	4	2 políticas de TM e *locks*
[25]	TM, *locks* e semáforos	HTM	MPARM	Alguns	Alguns	Alguns	7 *microbenchmarks*	depende da aplicação: 1, 2, 4, 8 e 16	4 políticas de TM, *locks* e semáforos distribuídos
[48]	TM e *locks*	STM	MPARM	Sim	Sim	Não	STAMP com TL2	1, 2, 4 e 8	4 políticas de TM e *locks*
[12, 13]	TM	STM	MPARM	Sim	Sim	Alguns	STAMP com TL2	1, 2, 4 e 8	8 políticas de TM
Esta dissertação	TM	STM	MPARM	Sim	Sim	Sim	STAMP com TL2	1, 2, 4 e 8	36 políticas de TM

Capítulo 3

Políticas de Memórias Transacionais

Como visto na seção 2.1, várias implementações e variações de sistemas de TM foram propostas. Cada uma delas tenta explorar alguma técnica ou conceito diferente, variando questões como isolamento, granularidade e forma de controle das versões de memória. Todas estas propostas se encaixam em três grandes grupos básicos de sistemas de TM:

- Memória Transacional em Hardware (HTM): são sistemas onde todo o controle de concorrência é gerenciado por dispositivos implementados no hardware. São considerados empiricamente mais eficientes em velocidade de execução mas possuem limitações intrínsecas ao desenho de hardware, sendo pouco flexíveis, de forma que é difícil ou impossível alterar facilmente alguns comportamentos dos sistemas, como os que serão vistos nas próximas seções.

- Memória Transacional em Software (STM): inicialmente propostas em [72], são sistemas onde o controle de concorrência é gerenciado por uma ou mais camadas de software como, por exemplo, através de bibliotecas que disponibilizam toda a abstração de um sistema de TM para camadas de software superiores. Apesar de muitas vezes não serem tão eficientes quanto à velocidade de execução, são mais fáceis de modificar a fim de aplicar novas políticas como as que serão vistas nas próximas seções. Por isto são bastante utilizados em estudos de análise do comportamento de sistemas de TM sob várias condições e com vários parâmetros de execução. Neste estudo este é o tipo de sistema que será utilizado.

- Memória Transacional Híbrida (HyTM): são sistemas onde o controle de concorrência é parte gerenciado por dispositivos implementados em hardware e parte gerenciado por uma ou mais camadas de software que trabalha cooperativamente com este hardware.

Apesar de estes sistemas mapearem suas características em instruções específicas do processador ou em bibliotecas de software, o comum a todos eles é a abstração de TM para o programador. É esta abstração que facilita a programação concorrente.

Hoje em dia, a tendência comercial é que a implementação de TM que deverá se sobressair para uso em alta escala é a de HyTM. Isto porque ela alia suporte direto no próprio processador e hierarquia de memória, o que provê mais rapidez ao processamento e controle do sistema de TM, com a flexibilidade de possíveis extensões e controles adicionais em software. Isto, por exemplo, é a proposta prática com os processadores de propósito geral que já anunciaram suporte a instruções específicas de TM [31, 43]. Nesta dissertação, no entanto, o foco será em sistemas de STM. Estes, além de seguirem os mesmo conceitos básicos presentes em HTM e HyTM, também possuem grande flexibilidade para implementação e experimentação de novas políticas de gerência do sistema de TM, provendo rapidez na obtenção de resultados que indicam como cada política implementada impacta tanto no desempenho em relação ao tempo de processamento quanto ao consumo de energia.

Além destas variações já vistas que, em geral, necessitam de uma ação relativamente intrusiva no sistema de TM para serem implementadas, existe uma série de outra variações cuja implementação é menos intrusiva e que podem coexistir simultaneamente no código de um mesmo sistema de STM, foco deste trabalho, sendo que o comportamento desejado pode ser selecionado em tempo de compilação ou execução através de parâmetros passados ao sistema de STM. Estas variações afetam diretamente o comportamento de um sistema de STM e podem influenciar consideravelmente não só o desempenho das aplicações como também o consumo energético durante a execução das mesmas. Chama-se, neste trabalho, estas variações acionadas através de parâmetros de compilação ou execução, de políticas.

Políticas de TM são, portanto, pequenas variações no comportamento dos sistemas de STM acionadas facilmente por meio de parâmetros de execução. Todas as políticas estão implementadas no mesmo sistema de STM através de modificações na biblioteca que será usada neste trabalho (TL2, que será explicada com maiores detalhes na seção 4.2), sendo simplesmente acionadas ou não durante uma execução. Como será visto, várias políticas podem ser combinadas em uma mesma execução, dependendo de qual aspecto do sistema de TM ela influencia. Este conceito, que aqui está sendo descrito para uma STM, pode ser também estendido para HTM, através de modificações no hardware, ou HyTM, através de modificações no hardware ou no software, apesar de este não ser o foco deste trabalho.

3.1 Controle de concorrência

Os sistemas de TM podem variar de acordo com a forma como eles tratam exceções lançadas durante uma transação e a forma como eles fazem o controle de concorrência.

Este pode ser pessimista, caso em que o sistema detecta e resolve o conflito assim que ele acontece; otimista, caso em que o sistema apenas detecta e resolve o conflito depois que ele acontece, ou ainda pode haver uma mistura de políticas pessimistas e otimistas em um mesmo sistema de TM. Neste trabalho, são analisados sistemas pessimistas e sistemas otimistas. Resolver o conflito, neste contexto, envolve abortar a transação que o identificou e tomar uma ação que, em geral, é simplesmente reiniciar a transação.

Nos sistemas de TM pessimistas, também conhecidos como *Eager* TM, uma transação pode ter a posse exclusiva de uma região de memória. Este tipo de acesso exclusivo pode gerar situações de *deadlock* e *livelock* que podem ser resolvidas com a obtenção de acesso exclusivo em uma ordem fixa ou abortando uma transação que tenha entrado neste estado.

Já nos sistemas de TM otimistas, também chamados de *Lazy* TM, várias transações podem acessar uma mesma região de memória concorrentemente. Os conflitos, caso ocorram, são detectados apenas quando a transação é finalizada.

3.2 *Backoff*

Pode não ser bom, no entanto, reiniciar a transação logo que ela é abortada, pois a chance de conflito ainda pode ser grande. Adota-se, então, um tempo de espera (*backoff*) para reiniciar a transação:

- *Backoff* linear é aquele em que o tempo de espera aumenta linearmente com o número de vezes que a transação abortou.
- *Backoff* exponencial é aquele em que o tempo de espera aumenta exponencialmente com o número de vezes que a transação abortou.

3.3 DVFS

Além das variações vistas nas seções 3.1 e 3.2, usou-se *Dynamic Voltage and Frequency Scaling* (DVFS) para mudar dinamicamente a frequência do núcleo e, assim, seu consumo energético. Neste trabalho, alterou-se o sistema de TM para variar a frequência do núcleo em certos momentos de suas operações, a fim de avaliar políticas energeticamente mais eficientes.

O processador utilizado suporta DVFS e 256 diferentes frequências de operação (mais detalhes na seção 4.1). O controle de qual é a frequência de operação do processador em um determinado momento, na plataforma utilizada neste trabalho, é feito via software. Destas frequências, quatro foram selecionadas para serem utilizadas neste experimento:

- `FULL_PERFORMANCE`: frequência de operação de 200 MHz, é a frequência máxima de operação, sendo aquela na qual o processador inicia sua operação e nela permanece a não ser que uma mudança de frequência seja requisitada via software. Esta é a frequência de operação quando se utiliza um divisor de frequência igual a 1 no processador.

- `HALF_SPEED`: frequência de operação de 100 MHz. Esta é a frequência de operação quando se utiliza um divisor de frequência igual a 2 no processador.

- `BEST_BALANCE`: frequência de operação de 66,67 MHz, é a maior frequência de operação possível com a menor voltagem suportada pelo núcleo, 0,3 V. Esta é a frequência de operação quando se utiliza um divisor de frequência igual a 3 no processador.

- `SLOWEST`: frequência de operação de 1,56 MHz, é a menor frequência de operação utilizada neste experimento. Esta é a frequência de operação quando se utiliza um divisor de frequência igual a 128 no processador.

Frequências mais baixas são suportadas pelo processador utilizado, chegando-se próximo a 781 kHz. No entanto, para fins deste experimento, não é um caso de uso real o uso de frequências tão baixas em processadores ARM executando os tipos de aplicações que estão sendo testadas aqui. A única exceção em relação a estes quatro níveis de frequência é a política `DVFS_AGGRESSIVE` que, como será visto adiante nesta seção, diminui a frequência de execução de forma variável.

Os quatro valores de frequência utilizados foram escolhidos de acordo com os seguintes critérios:

- `FULL_PERFORMANCE`: sendo esta a frequência máxima do processador utilizado nos experimentos, é natural que seja estudado o gasto energético com esta frequência que, teoricamente, entrega o maior poder de processamento.

- `HALF_SPEED` e `BEST_BALANCE`: apesar de o processador utilizado nos experimentos suportar valores para divisores de frequência entre 1 e 256, a frequência abaixa rapidamente com o aumento do divisor como pode ser visto na figura 4.1. Para o propósito deste estudo — uma análise entre desempenho de processamento e consumo energético — não parece interessante simplesmente executar as aplicações em frequências muito baixas, até porque isto não corresponde a um caso real de aplicação hoje em dia. Desta forma, foram selecionados os menores divisores possíveis (fornecendo, assim, as maiores frequências possíveis) até o ponto em que o processador atingia a mínima voltagem de operação. Apesar de o consumo energético variar também com outras variáveis, como a frequência, a partir deste ponto, a voltagem

já não afeta diretamente o consumo energético. Além disto, para o próximo valor de divisor de frequência (4), a frequência de operação cairia para 50 MHz, o que é um valor bem baixo mesmo para processadores embarcados hoje em dia.

- `SLOWEST`: este valor foi utilizado para comparação com estudos anteriores feitos em [13]. Além disto, nota-se que variando os valores do divisor de frequência entre 1 e 128 explora-se a maior parte do espaço de frequência suportado pelo processador utilizado nos experimentos deste trabalho. Variações do divisor de frequência entre 129 e 256 fazem a frequência variar entre 1,55 MHz e 781 kHz, respectivamente; um intervalo bem menor do que aquele obtido com a variação do divisor de frequência entre 1 e 128 (200 MHz e 1,56 MHz, respectivamente). Isto pode ser facilmente verificado pela figura 4.1.

Sobre estas quatro frequências de operação base, várias políticas utilizando DVFS foram implementadas. Cada uma delas explorou a variação da frequência de operação do processador em um determinado momento das operações realizadas pelo sistema de TM. A figura 3.1 mostra um conflito simples entre duas transações e ajudará na explicação de cada uma das políticas que utilizam DVFS. As operações relacionadas à transação (`TxStart`, `TxCommit` e `TxAbort`) são as mesmas explicadas na seção 2.1.

- `NONE`: as políticas de controle de concorrência e *backoff* podem ser combinadas entre si sem o uso de uma política DVFS combinada. Para facilitar o entendimento de todas as possibilidades e os cálculos e controles das quantidades de simulações feitas, considera-se neste trabalho que uma das políticas de DVFS adotadas é não utilizar DVFS.

- `DVFS_CONSERVATIVE`: nesta política diminui-se a frequência de operação do núcleo para `SLOWEST` enquanto uma transação está executando o laço do *backoff*, como pode ser visto no código 3.1. O *backoff* sempre é executado ao final da operação que aborta a transação conflitante (`TxAbort` na figura 3.1). Após a execução do *backoff*, a STM utilizada nesta dissertação tenta reiniciar a transação novamente (`TxStart` na figura 3.1). O motivador por trás desta política é que o *backoff* não executa nenhum código útil, já que é apenas uma espera. Desta forma, mesmo executando em menor frequência, é possível ajustar o número de ciclos de espera de forma que a transação espere o tempo correto para ser reiniciada pelo sistema de TM, porém o faça em uma frequência de execução menor e, portanto, gastando menos energia enquanto não está entregando nenhum trabalho útil.

- `DVFS_COMMIT`: nestas políticas diminui-se a frequência de operação do núcleo durante o *commit* da transação (`TxCommit` na figura 3.1). Esta política é inspirada no fato

de que vários estudos mostram que diminuir a frequência de operação enquanto um programa acessa intensamente a memória tende a diminuir o consumo de energia sem impactar o desempenho do programa [47]. Isto porque operações na memória são muito mais lentas que a execução de instruções no processador. Se a velocidade de execução do código do programa é ditada majoritariamente pelos acessos à memória, diminuir a frequência da execução do processador não irá impactar consideravelmente no desempenho do processador, mas irá impactar na quantidade de energia consumida. Como a fase de *commit* é, geralmente, a fase em que o sistema de TM mais acessa a memória, parece interessante diminuir a frequência de operação do processador durante a execução desta fase. Três sub-tipos deste tipo de política foram explorados:

  - `DVFS_COMMIT_LIGHT`: diminui-se a frequência de operação do núcleo para `HALF_SPEED`.
  - `DVFS_COMMIT_BEST_BALANCE`: diminui-se a frequência de operação do núcleo para `BEST_BALANCE`.
  - `DVFS_COMMIT_SLOWEST`: diminui-se a frequência de operação do núcleo para `SLOWEST`.

- `DVFS_AGGRESSIVE`: nesta política, diminui-se a frequência de operação do núcleo executando uma transação que aborta um número de vezes acima de um limiar. Após este limiar, a frequência diminui à medida que a transação aborta mais vezes, sendo que existe um controle sobre o número máximo de núcleos executando simultaneamente em baixa frequência, como pode ser visto no código 3.2. A verificação sobre se a transação abortou mais vezes que o limiar definido é feita durante a fase que aborta uma transação conflituosa (`TxAbort` na figura 3.1). Se o limiar for atingido, a frequência de execução é diminuída. Toda vez subsequente em que a mesma transação for abortada, a frequência será diminuída mais ainda, até que a transação seja finalizada (*commit*) com sucesso. A motivação por trás desta política é a de deixar as transações conflituosas mais lentas na esperança de que as outras transações que estejam conflitando com esta completem mais rapidamente, de forma que a próxima vez que esta transação conflituosa for executada, ela será executada com sucesso.

- `DVFS_ABORTER_ENABLED`: nesta política, assim como na política `DVFS_AGGRESSIVE`, diminui-se a frequência de operação do núcleo executando uma transação que aborta um número de vezes acima de um limiar. No entanto, neste caso, ao invés de uma diminuição proporcional ao número de vezes que a transação abortou, são utilizados apenas dois patamares de frequência de operação (`HALF_SPEED` e `BEST_BALANCE`)

que são acionados quando limiares determinados para o número de vezes que a transação abortou são atingidos. Neste caso também, ao contrário da política `DVFS_AGGRESSIVE`, não existe nenhum tipo de controle sobre o número máximo de núcleos simultaneamente executando em baixa frequência. O código 3.3 mostra como esta política é controlada. Assim como no caso `DVFS_AGGRESSIVE`, a verificação do número de vezes em que a transação foi abortada em relação aos limiares definidos é feita durante a fase que aborta uma transação conflituosa (`TxAbort` na figura 3.1). Desta forma, quando a transação for reiniciada pelo sistema de TM utilizado nesta dissertação após a ocorrência do conflito (`TxStart` na figura 3.1), ela já será executada em menor frequência caso os limiares tenham sido atingidos.

- `WAIT_ABORTER_ENABLED`: nesta política, baseada nas propostas descritas em [74], coloca-se o núcleo em estado de espera caso um limiar de contenção seja alcançado. Este limiar é calculado a partir do número de transações abortadas ou finalizadas com sucesso no núcleo, calculados, respectivamente, nas fases `TxAbort` e `TxCommit` da figura 3.1. Quando outras transações em outros núcleos finalizam, elas liberam a execução deste núcleo. A motivação, neste caso, é que se existe muita contenção entre duas ou mais transações executando em um determinado momento, talvez seja interessante serializar a execução das mesmas. Assim, evita-se o conflito e tem-se certeza de que todas chegarão à fase de *commit*. Desta forma, quando a primeira das transações conflituosas atingir o limiar de contenção, ela ficará em um estado de espera até que, quando outras transações finalizarem, elas liberem a transação que está em espera para executar também. Note que, apesar de esta política não fazer uso de DVFS, foi decidido colocá-la aqui nesta seção junto com as outras ao invés de separá-la em uma categoria própria pois ela (i) sugere um método de espera na execução que, apesar de não utilizar variação de frequência, tenta, assim como no caso da variação de frequência, diminuir a velocidade de execução (neste caso, mais especificamente, parar a execução efetiva) de uma transação conflitante para evitar o conflito e (ii) esta política serviu de inspiração para a próxima política que é uma variação desta com o uso de DVFS.

- `WAIT_ABORTER_DVFS`: nesta política, da mesma forma que acontece com a política `WAIT_ABORTER_ENABLED`, coloca-se o núcleo em estado de espera caso um limiar de contenção seja alcançado, porém a espera é feita em frequência de operação `BEST_BALANCE`. O código 3.4 mostra o estado de espera implementado nesta política. Os cálculos dos limiares nas fases `TxAbort` e `TxCommit` da figura 3.1 são feitos da mesma forma que explicado para `WAIT_ABORTER_ENABLED`. A diferença neste caso está mesmo apenas na utilização de DVFS durante a espera.

Note que, apesar de a figura 3.1 mostrar o comportamento hipotético de um sistema

de TM *Lazy*, as mesmas variações acima poderiam ser estendidas para memórias *Eager*, não invalidando, portanto, as políticas acima para o caso em que a TM possua um outro tipo de controle de concorrência.

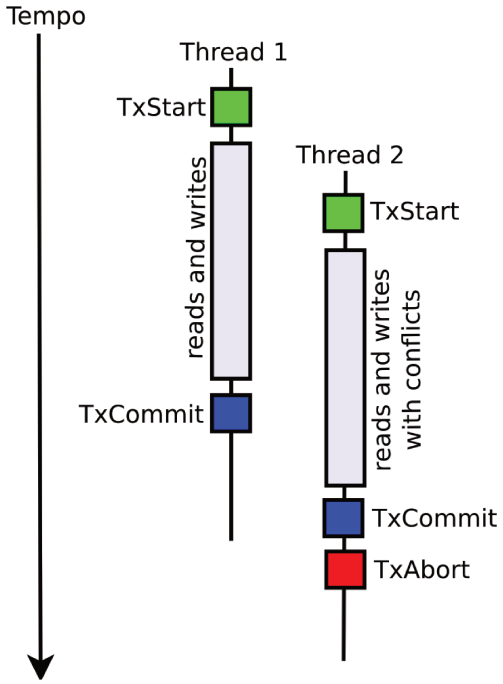


Figura 3.1: Exemplo de execução de transações conflitantes.

Código 3.1: *Backoff*.

```
__INLINE__ void
backoff (Thread* Self, long attempt)
{
#ifdef TL2_BACKOFF_EXPONENTIAL
    unsigned long long n = 1 << ((attempt < 31) ? (attempt) : (31));
    unsigned long long stall = TSRandom(Self) % n;
#else
    unsigned long long stall = TSRandom(Self) & 0xF;
    stall += attempt >> 2;
    stall *= 10;
#endif

/* Max and Min frequencies (in MHz) */
#define MAX_FREQ    200.0f
#define MIN_FREQ    1.56f


#ifdef DVFS_CONSERVATIVE
#   define FREQ_DIV     (MAX_FREQ/MIN_FREQ)
    DVFS_SCALE_SLOWEST(); /* slow down the processor while backing off */
#else
#   define FREQ_DIV     1
#endif

    /* CCM: timer function may misbehave */
    volatile typeof(stall) i = 0;
    while (i++ < stall/FREQ_DIV) {
        PAUSE();
```

```
    }

#ifdef DVFS_CONSERVATIVE
    DVFS_SCALE_FULL_SPEED(); // go back to full performance
#endif
}
```

Código 3.2: `DVFS_AGGRESSIVE`.

```
#ifdef DVFS_AGGRESSIVE
/* if the transaction is continuosly aborting, slow down its execution until it commits */
if (Self->Retries > RETRIES_THRESHOLD) {

  WAIT(-1);
    if (cores_in_low_power_mode < MAX_CORES_LOW_POWER) {
      cores_in_low_power_mode++;
      Self->in_low_power = 1;
    }
  SIGNAL(-1);

  if (Self->in_low_power)
    SELECT_FREQUENCY(Self->Retries);
}
```

Código 3.3: `DVFS_ABORTER_ENABLED`.

```
#elif DVFS_ABORTER_ENABLED

/* If the transaction is continuously aborting, slow down its execution until it commits */
/* This is a little bit more naive than DVFS_AGGRESSIVE as we are not limiting how many cores can slow
    down at a certain moment in time. */
/* We are also slowing down from the third abort, not the fourth as in DVFS_AGGRESSIVE */
/* Also, we slow down the frequency only twice:
 *   - Normal frequency: 200 MHz
 *   - Half frequency: 100 MHz
 *   - Best balance: 66 MHz
 * Each change in frequency is triggered by 3 consecutive aborts */
if (Self->Retries >= RETRIES_THRESHOLD) {
    int speed_pos = Self->Retries / RETRIES_THRESHOLD;
    if (speed_pos > DVFS_SCALE_SPEED_LEVELS - 1) {
        speed_pos = DVFS_SCALE_SPEED_LEVELS - 1;
    }
    DVFS_SCALE(dvfs_scale_speed_values[speed_pos]);
}
```

Código 3.4: `WAIT_LOW_FREQUENCY` - esperando por um *lock* em um estado de baixa frequência.

```
int WAIT_LOW_FREQUENCY(int ID, int DIVIDER)
{
  unsigned short int original_divider = get_this_core_frequency();
  while (lock[ID]) {
    scale_this_core_frequency(DIVIDER);
  }
  scale_this_core_frequency(original_divider);
  return(0);
}
```

Os três conjuntos de políticas das seções 3.1, 3.2 e 3.3 estão representados na figura 3.2. Sendo estes conjuntos de políticas ortogonais entre si, as políticas de cada uma destas seções podem ser combinadas uma a uma ao mesmo tempo em um sistema de TM. Desta forma, 36 combinações são possíveis, gerando 36 possíveis políticas diferentes.

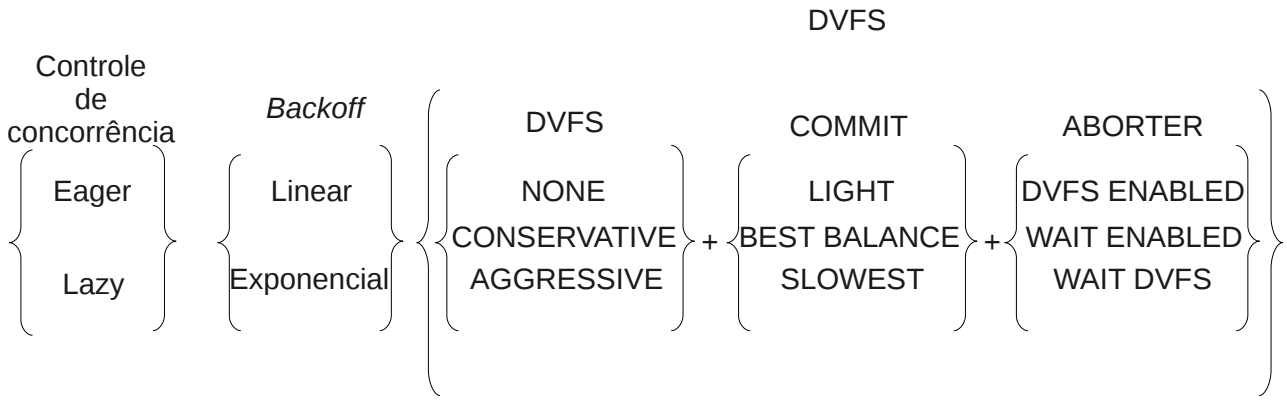


Figura 3.2: Conjuntos de políticas de TM das seções 3.1, 3.2 e 3.3

Capítulo 4

Ambiente experimental

Esta dissertação de mestrado faz uma avaliação sobre o consumo de energia e o desempenho de processamento em uma STM configurada com diversas políticas de gerenciamento da TM e de utilização energética, sendo algumas destas combinações inéditas. Em relação à utilização energética foi explorado o conceito de *Dynamic Voltage and Frequency Scaling* (DVFS) com TM. Esta funcionalidade, disponível na maior parte dos processadores hoje em dia, torna possível alterar dinamicamente a frequência de operação do processador e, consequentemente, seu consumo de energia. Este controle de qual é a frequência de operação do processador em um determinado momento, em geral, é feito via software. No caso deste trabalho, alterou-se o sistema de TM para que ele variasse a frequência de operação do processador em determinados momentos da execução das operações de TM, conforme descrito no capítulo 3, a fim de se explorar e avaliar políticas que economizassem energia ao mesmo tempo que otimizassem ao máximo o uso dos sistemas de TM.

Para que os experimentos fossem executados, foi primordial ter um conjunto de aplicativos, também conhecido como um pacote de *benchmarks*, sobre o qual as avaliações e comparações sobre consumo de energia e desempenho de processamento foram feitas. Como foi dito na seção 2.2, o único pacote de *benchmarks* atualmente disponível focado em TM é o STAMP. No entanto, para que fosse possível utilizar o STAMP para as avaliações e comparações aqui propostas, foi necessário inicialmente superar inúmeras dificuldades encontradas na compilação e correta execução dos programas. Além disto, foi feito um estudo de cada uma das aplicações disponíveis neste pacote de *benchmarks*, para que fosse possível não só entender o funcionamento delas, como também conseguir compilá-las e certificar que sua execução estava gerando resultados corretos.

Para a coleta dos dados necessários para a avaliação e comparação propostas nesta dissertação, os programas do STAMP foram alterados de forma que fosse possível contabilizar dados de desempenho de processamento e consumo energético sobre a execução do sistema de TM. Para entender como estas alterações foram feitas, é necessário entender

um pouco mais do ambiente sobre o qual as execuções do STAMP foram realizadas.

4.1 Ambiente de simulação MPARM

Para executar os *benchmarks* do STAMP neste experimento foi utilizado o MPARM [55], um simulador com precisão de ciclos e sinais de um sistema multiprocessado, baseado em processadores ARM, escrito em SystemC, que executa em arquitetura x86. As aplicações foram executadas diretamente sobre o processador, sem nenhum sistema operacional. Logo, sem trocas de contextos nem interrupções, as aplicações são executadas sempre da mesma forma e na mesma ordem.

O MPARM permite executar aplicações na plataforma simulada de forma realística, obtendo-se um comportamento funcional exato, além de ser possível extrair dados de consumo energético e de desempenho do processamento. A plataforma simulada pelo MPARM é formada pelos seguintes componentes:

1. Um número configurável de processadores ARMv7 de 32 bits, com 8 kB de *cache* de instruções (associatividade 4) e 4 kB de *cache* de dados (associatividade 4, *write-through*).

2. A memória privada destes processadores, cada uma formada por uma SRAM de 12 MB.

3. Uma memória compartilhada, formada por uma SRAM de 12 MB.

4. Um módulo de interrupção de hardware.

5. Um módulo de semáforos em hardware.

6. Um barramento de interconexão AMBA-AHB.

Os acessos feitos à memória compartilhada não são guardados na *cache* pois o barramento utilizado não implementa um protocolo de coerência de *cache*. Mesmo assim, outros trabalhos também usam o MPARM devido a outras vantagens providas pela plataforma. O consumo energético, por exemplo, é medido através da medição do consumo energético de todos os elementos que compõem a plataforma. No caso dos experimentos com DVFS feitos neste trabalho, apenas a frequência de operação do processador foi alterada, já que este é o único elemento que permite variação na frequência de operação (assim como acontece na maior parte das plataformas que suportam DVFS).

A frequência base do processador simulado é 200 MHz. Esta frequência base pode ser diminuída através do uso de um divisor de frequência que pode ser configurado com

valores entre 1 e 256. Desta forma, a frequência do processador pode variar de 200 MHz a 781,25 kHz. Esta variação de frequência em relação ao divisor pode ser vista na figura 4.1.

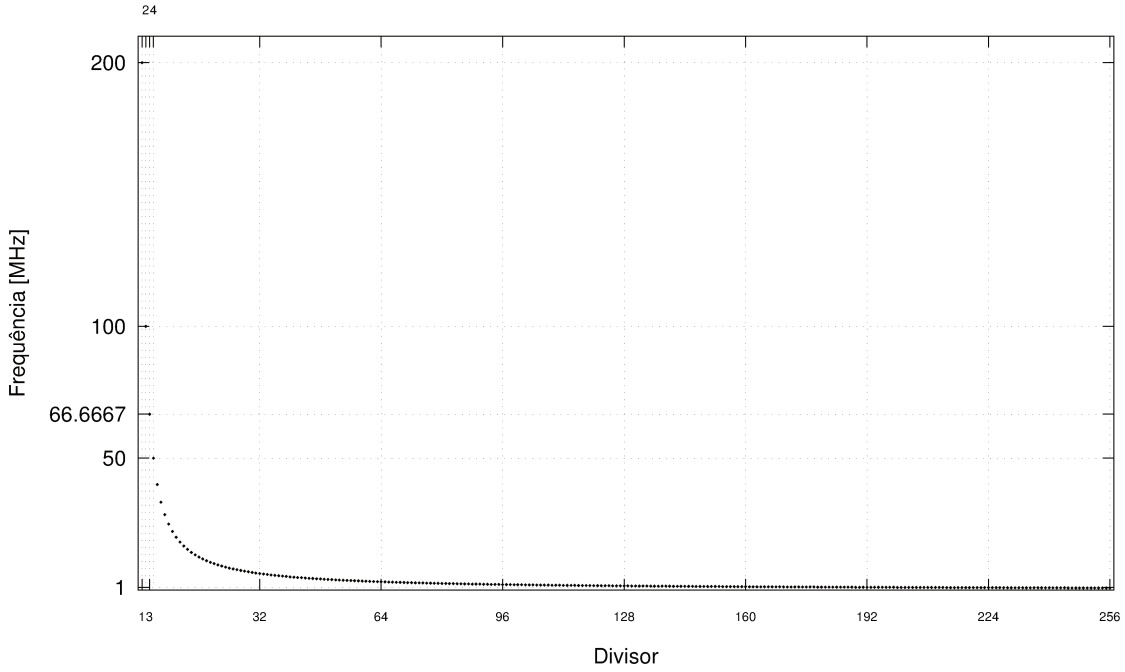


Figura 4.1: Frequência de operação do processador ARM na plataforma simulada MPARM em relação aos divisores de frequência suportados.

A frequência é controlada via software através da escrita de um novo valor no divisor. Isto pode ser facilmente feito através da função `scale_this_core_frequency` fornecida pela API disponibilizada pelo MPARM. Esta função escreve o novo valor especificado no registrador mapeado em memória que corresponde ao divisor de frequência no hardware.

Como seria de se esperar, o consumo energético do processador ARM varia com a voltagem de alimentação e com a frequência de execução, sendo que a voltagem de alimentação varia de acordo com a frequência de execução. A voltagem de alimentação varia para períodos de *clock* entre 5 ns e 11 ns, que englobam os valores de divisor de frequência 1 e 2. Com divisor de frequência igual ou maior a 3, atingimos o menor valor de voltagem de alimentação possível (0,3 V), como citado na seção 3.3.

Como foi dito no início desta seção, as aplicações do STAMP foram executadas diretamente sobre o processador, sem nenhum sistema operacional. Para que isto fosse possível, as aplicações do STAMP foram portadas para o ambiente do MPARM, já que elas foram originalmente desenvolvidas para executar em uma plataforma x86 com acesso

a um sistema operacional.

4.1.1 Ambiente de compilação e execução do simulador

O MPARM é um projeto relativamente antigo e, por causa disto, conseguir montar um ambiente de compilação e execução do simulador apresentou alguns desafios a serem superados. A plataforma MPARM, além de requerer uma versão antiga de SystemC para ser compilada, também precisa de versões antigas de compiladores e bibliotecas em 32 bits para funcionar, o que pode ser particularmente problemático em sistemas GNU/Linux de 64 bits atuais.

No apêndice C poderão ser encontrados os dados necessários para acessar todos os códigos e modificações realizadas para executar o simulador no ambiente descrito nesta subseção.

Compilação do SystemC

O MPARM é um simulador construído utilizando-se o SystemC [1]. Logo, para compilá-lo, é necessário, antes disto, compilar o SystemC. O MPARM utilizado neste experimento, no entanto, só funciona quando compilado com o SystemC 2.0.1. Esta versão do SystemC não é compilada pelo GCC 4.x atualmente disponível nas distribuições GNU/Linux, sendo necessário um GCC 3.4 para compilá-lo. Isto já impôs inicialmente uma limitação no ambiente utilizado para compilar o ambiente de simulação, já que o mesmo deveria, de alguma forma, ter disponível esta versão mais antiga do GCC. Além disto, o SystemC 2.0.1 só executa em ambientes 32 bits, o que também se contrapõe com o fato de as mais populares distribuições GNU/Linux atuais para x86 serem de 64 bits. A maneira mais fácil encontrada para sobressair a estes problemas foi a utilização de um Fedora 14, o qual disponibiliza pacotes pré-configurados que possibilitam facilmente a convivência de diversas versões de GCC na mesma instalação de GNU/Linux, além de possuir um suporte muito bom de *multilib*, que possibilita a execução de programas e bibliotecas de 32 e 64 bits no mesmo ambiente, o que inclui também o suporte a GCC em ambas as arquiteturas.

Após baixar o SystemC 2.0.1 do *site* oficial e modificar algumas opções de compilação em seu arquivo de configuração da compilação para adaptá-lo ao Fedora 14, foi utilizado o GCC 3.4 com opções de compilação de 32 bits para compilá-lo, conforme mostrado no código 4.1. Neste trecho de comandos, `<systemc-2.0.1>` é o diretório onde está o código-fonte do SystemC 2.0.1 baixado do *site* oficial.

Também do *site* oficial é possível baixar o código-fonte dos testes de regressão do SystemC 2.0.1, que são úteis para verificar se a compilação do mesmo ocorreu com sucesso. Após a compilação do SystemC 2.0.1, pode-se executar os comandos mostrados no

código 4.2 para verificar se a compilação ocorreu com sucesso. Neste trecho de comandos, `<systemc_test-2.0.1>` é o diretório onde está o código-fonte dos testes de regressão baixado do *site* oficial.

Código 4.1: Procedimento para compilação do SystemC 2.0.1 no Fedora 14.

```
sudo mkdir -p /opt/systemc-2.0.1
cd <systemc-2.0.1>
mkdir objdir
cd objdir
sudo CC=gcc34 CXX=g++34 ../configure --prefix=/opt/systemc-2.0.1 --build=i386-pc-linux --host=i386-pc-
    linux --target=i386-pc-linux
sudo make
sudo make install
sudo make check
```

Código 4.2: Procedimento para execução de testes de regressão do SystemC 2.0.1.

```
cd <systemc_test-2.0.1>
mkdir logs
cd logs
SYSTEMC_HOME=/opt/systemc-2.0.1 CC=gcc34 CXX=g++34 RT_OPTS="-m32" ../scripts/verify.pl -v systemc
```

Para executar os testes de regressão com sucesso foram necessárias algumas pequenas modificações no *script* `verify.pl` para que ele funcionasse corretamente no Fedora 14.

Após executar os testes de regressão, foi verificado que, dos 634 testes disponíveis, apenas 2 não passaram com sucesso. Estes dois testes são conhecidos por gerarem erros em alguns ambientes e, na avaliação que foi feita, isto não afetaria o funcionamento do simulador MPARM dado os requisitos que ele usa em relação ao SystemC.

Compilação do MPARM

Após compilar o SystemC 2.0.1, foi possível compilar o simulador MPARM. Para isto, também foram necessárias algumas modificações no arquivo de configuração de compilação para adaptá-lo ao ambiente baseado em Fedora 14 que foi construído. Especificamente, foram necessárias as seguintes modificações:

- Apontar o diretório onde foi instalado o SystemC 2.0.1 (`/opt/systemc-2.0.1` no caso desta dissertação).
- Especificar que será utilizado o compilador `gcc++34`.
- Especificar a opção de compilação `-m32` para compilar o código em 32 bits (arquitetura i386).

O MPARM vem com uma série de aplicações de teste que podem ser executadas para verificação da compilação com sucesso do simulador. Após compilar o simulador,

compilou-se e executou-se estas aplicações para se ter certeza de que o simulador foi compilado com sucesso no ambiente baseado em Fedora 14 construído.

É necessário um conjunto de ferramentas de compilação cruzada para compilar estas aplicações de exemplo já que o ambiente Fedora 14 executa em cima de processadores com arquitetura x86_64 e as aplicações que executarão no simulador devem ser compatíveis com a arquitetura ARM. Para isto, foi instalado, no ambiente de compilação, um conjunto de ferramentas de compilação cruzada, baseado no GCC, chamado RTEMS GCC [6]. Este conjunto de ferramentas de compilação fornece suporte à compilação cruzada para uma série de arquiteturas, inclusive a arquitetura ARM simulada pelo MPARM. Além disto, ele oferece a funcionalidade de poder ser instalado através de repositórios `yum`, suportados pelo Fedora 14, o que torna sua instalação e configuração especialmente fácil no ambiente que foi construído para a compilação dos programas que serão utilizados neste experimento. No caso do simulador MPARM que foi utilizado, a versão do conjunto de ferramentas de compilação cruzada que deve ser instalado é o RTEMS 4.6, pois esta é a última versão compatível com a versão do MPARM que foi utilizada neste experimento.

Após instalar o ARM RTEMS GCC 4.6 no ambiente baseado em Fedora 14, foi possível compilar as aplicações de exemplo do simulador MPARM que não dependem de um sistema operacional para executar (aplicações que executam diretamente sobre o processador). Por uma limitação do simulador MPARM, é possível simular sistemas com, no máximo, 12 núcleos de processamento. Acima disto o simulador tem a execução interrompida com um erro de `segmentation fault`.

Fedora 14

O Fedora [2] é um sistema operacional baseado em Linux, também conhecido como uma distribuição GNU/Linux. O seu principal patrocinador é a Red Hat.

O Fedora é uma das distribuições GNU/Linux que mais rapidamente adota novas versões de programas e aceita correções de *bugs*. Além disto, é uma das distribuições GNU/Linux com suporte a *multilib*, que possibilita a execução de programas e bibliotecas de 32 e 64 bits no mesmo sistema. Ele também não apenas fornece um GCC capaz de gerar código para ambas as arquiteturas como também fornece camadas de compatibilidade com versões anteriores do GCC. Devido a estes fatos, que casam a necessidade de se ter um ambiente o mais atual possível aliado à flexibilidade de se executar aplicações de 32 bits (necessárias para a compilação do simulador e de suas aplicações) e 64 bits (necessárias para outra atividades relacionadas a esta pesquisa) compatíveis com binários gerados por versões antigas do GCC, além, é claro, da familiaridade do autor com este ambiente, é que o Fedora foi escolhido como plataforma para compilação do simulador e das aplicações do pacote de *benchmarks* STAMP.

A escolha do Fedora 14 foi feita por ser a última versão estável do Fedora disponível

quando se iniciou o trabalho com as simulações executadas para esta dissertação. Após isto, foi tomada a decisão de não fazer a atualização do sistema de compilação para novas versões do Fedora (hoje a última versão disponível é o Fedora 19) para se garantir que sempre o mesmo código estivesse sendo utilizado em todas as execuções. No entanto, não deve ser difícil a adaptação do processo descrito nas subseções anteriores para ambientes com versões mais novas do Fedora.

Após instalar o Fedora 14 de 64 bits em uma máquina x86_64, foi necessário instalar todas as atualizações disponíveis para este sistema operacional, conforme mostrado no código 4.3.

Código 4.3: Procedimento para instalação das atualizações disponíveis no Fedora 14.

```
yum update
```

A seguir, para se certificar de que as versões corretas de GCC necessárias para compilar o SystemC e o MPARM foram instaladas, é necessário executar o comando mostrado no código 4.4.

Código 4.4: Procedimento para instalação das dependências necessárias para compilação do SystemC e do MPARM.

```
yum install compat-gcc-34 compat-libstdc++-33 compat-gcc-34-c++ compat-libf2c-34 compat-gcc-34-g77
    compat-libstdc++-296.i686 compat-libstdc++-33.i686 glibc.i686 glibc-devel.i686 libgcc.i686 libstdc
    ++-devel.i686
```

Isto certificará de que as versões de 64 bits do GCC estarão instaladas, assim como as necessárias bibliotecas de 32 bits e camadas de compatibilidade com versões mais antigas do GCC. Durante a execução do GCC, serão passadas opções de compilação para instruir o compilador a gerar código de 32 bits, compatível com o que é requisitado pelo SystemC e o MPARM sendo utilizados.

Execução do simulador

O simulador MPARM pode ser executado em qualquer ambiente que tenha disponível os binários gerados pela compilação do SystemC, os binários do MPARM e as bibliotecas de sistema de 32 bits (`libc`) necessárias tanto para a execução do SystemC quanto do MPARM gerados a partir do GCC 3.4 de 32 bits. Esta execução, portanto, pode de ser feita no ambiente Fedora 14 descrito na subseção 4.1.1 ou em qualquer outro sistema de 32 bits com todas as dependências corretamente instaladas. Para os experimentos deste trabalho, o conjunto de testes que vem com o SystemC e o MPARM foram executados no ambiente Fedora 14. Já a execução do STAMP foi feita em um outro ambiente instalado no *cluster* de máquinas disponibilizados pelo Laboratório de Sistemas Computacionais (LSC) do Instituto de Computação (IC) da UNICAMP, como será descrito na seção 4.3.

4.1.2 Medidas de tempo

Um ponto importante a ressaltar é como foram feitas as medidas de tempo de execução do simulador MPARM. Uma preocupação recorrente sobre medidas de tempos de execução em computadores é quanto a extração destas medidas afeta diretamente no valor medido, já que é gasto um determinado tempo de execução para se extrair a medida. É preciso ter certeza de que o tempo de execução gasto na extração da medida não interfere de maneira indesejável nos valores sendo medidos.

Este tipo de preocupação é especialmente necessária no caso de sistemas em que se deseja medir o tempo de execução de uma aplicação executando em espaço de usuário e a medição de tempo só pode ser feita no espaço privilegiado do processador, sendo que a troca de contexto entre o espaço de usuário e o espaço privilegiado pode levar milhares de ciclos de relógio do processador, o que certamente afeta negativamente a precisão da medida sendo extraída, especialmente considerando que a aplicação de usuário poderá não executar sem interrupção durante um tempo muito maior do que aquele gasto para o chaveamento entre o espaço de usuário e o espaço privilegiado do processador.

Neste trabalho, no entanto, como o cenário é bem mais simples, este problema está muito mais sob controle, não chegando a afetar negativamente as medidas feitas. Em primeiro lugar, não está sendo utilizado um sistema operacional dentro do simulador. Em um determinado momento, apenas o *benchmark* sendo medido está sendo executado no processador. Não há trocas de contexto. Além disto, é utilizado um ambiente simulado. Mais do que isto, as medidas de tempo são extraídas a partir da infraestrutura de simulação provida pelo MPARM e não através de alguma instrução que seja diretamente executada no processador simulado. Portanto, a sobrecarga para se efetuar as medidas é muito baixa (da ordem de alguns poucos ciclos) e não chega a ser significante frente ao número de ciclos gastos durante a execução das operações de TM. Além disto, todas as medidas são afetadas pela mesma sobrecarga mínima de desempenho. Como, além de tudo, o interesse neste trabalho é apenas nos cálculos relativos de tempo de execução e não nos seus valores absolutos, chega-se à conclusão de que estes desvios causados pela extração da medida do tempo de execução serão muito pequenos e insignificantes para o experimento que está sendo realizado.

4.2 STAMP

O *Stanford Transactional Applications for Multi-Processing* (STAMP) foi o pacote de *benchmarks* utilizado neste trabalho para executar as medidas sobre o sistema de STM sendo estudado.

No apêndice C estão os dados necessários para acessar todos os códigos e modificações

realizadas no STAMP neste trabalho.

4.2.1 Ambiente de compilação e execução do STAMP

Como descrito na subseção 4.1.1, para a compilação de aplicações que serão executadas diretamente sobre o processador ARM simulado pelo MPARM foram utilizadas as ferramentas de compilação cruzada ARM RTEMS GCC dentro do ambiente baseado em Fedora 14 que foi criado para a compilação dos aplicativos que serão executados neste experimento.

Com as aplicações do STAMP não foi diferente. O STAMP foi originalmente escrito para arquiteturas x86. Para que ele pudesse ser executado em um processador ARM, foi necessário recompilá-lo utilizando um compilador cruzado. Além disto, foi necessário adaptar o código dos *benchmarks* do STAMP para executarem sem problemas no ambiente do simulador MPARM. Isto porque, como não foi utilizado um sistema operacional no processador simulado, foi necessário alterar todas as chamadas de entrada e saída que utilizavam as bibliotecas padrão da linguagem C para chamadas equivalentes diretamente nas interfaces de entrada e saída providas pelo processador simulado. Por questões de compatibilidade com o MPARM, aqui também foi utilizado o ARM RTEMS GCC 4.6.

Para facilitar a execução dos diversos *benchmarks* nas diversas configurações desejadas, foi desenvolvido um *script* que compila todos os aplicativos do STAMP com todas as configurações selecionadas. Este mesmo *script* é capaz de enfileirar a execução de todos os `benchmarks` no *cluster* HTCondor. O HTCondor foi a plataforma de execução utilizada para executar todas as simulações de forma controlada e paralela e será detalhado na subseção 4.3.1.

4.2.2 Dificuldades com STAMP

Além dos desafios em relação ao ambiente de compilação e execução do STAMP vistos anteriormente, também existiram algumas dificuldades na compilação e execução correta do conjunto de *benchmarks*.

Três defeitos foram encontrados em aplicações do STAMP. Estes defeitos foram validados com os autores do conjunto de *benchmarks*. Infelizmente não foi feito um anúncio de uma nova versão do STAMP com a correção destes defeitos mas, nas aplicações que foram executadas neste experimento, a correção para estes defeitos foi incluída a fim de se obter resultados corretos nas execuções. Destes três defeitos, apenas um que foi corrigido no aplicativo `ssca2` era realmente um defeito que estava causando execuções com resultados incorretos. Os aplicativos `genome` e `kmeans` também possuíam um defeito cada um que, se não corrigidos, poderiam gerar situações de condições de corrida que travariam

o progresso das execuções, gerando resultados incorretos. Estas últimas foram corrigidas pela alteração de uma variável para `volatile` dentro de um *loop*.

A maior parte das alterações, no entanto, foram para adaptar o código dos aplicativos do STAMP para executarem sem problemas no ambiente do simulador MPARM, sem um sistema operacional, conforme descrito na subseção 4.2.1.

4.2.3 TL2

Uma outra modificação que foi feita no STAMP foi a instrumentação da biblioteca TL2. A TL2 é a biblioteca que, por padrão, provê suporte a TM dentro do STAMP. Seu código é distribuído junto com o código do STAMP. A TL2 fornece a implementação de uma STM completa e como tal foi utilizada neste trabalho. Além da facilidade inerente de já ter a biblioteca integrada à interface de TM do STAMP, ela também provê uma infraestrutura para sua instrumentação através do arquivo `profile.h`. Através da definição de algumas macros neste arquivo é possível, a cada operação, transacional ou não, iniciar os contadores de desempenho disponíveis na infraestrutura de simulação provida pelo MPARM e, assim, coletar informações estatísticas sobre a execução da operação. No início de cada nova operação, as estatísticas são limpas. Assim, é possível traçar um quadro detalhado de informações sobre a execução do simulador para cada instante.

Estas informações nada mais são que as informações sobre desempenho e consumo energético durante a execução dos *benchmarks* que serão analisadas posteriormente por esta dissertação.

4.3 Execução das simulações

As aplicações do STAMP foram compiladas no ambiente ARM RTEMS GCC 4.6 no Fedora 14 e foi utilizado um *script* que enfileirava todas as simulações a serem executadas com os *benchmarks* no *cluster* HTCondor.

Com este processo, cuja execução foi controlada pelo HTCondor, executou-se todas as combinações possíveis das seguintes variações:

- 13 *benchmarks* (8 aplicações do STAMP, algumas com 2 cargas de trabalho):
  - `genome`
  - `genome+`
  - `intruder`
  - `intruder+`
  - `yada`

  - `labyrinth`
  - `labyrinth+`
  - `ssca2`
  - `vacation-low`
  - `vacation-high`
  - `bayes`
  - `kmeans-low`
  - `kmeans-high`

- Todas políticas de TM do capítulo 3:
  - 2 tipos de controle de concorrência: *Eager* e *Lazy*.
  - 2 variações de *backoff*: linear e exponencial.
  - 9 variações com DVFS.

- 4 variações de números de núcleos computacionais no sistema: 1, 2, 4 e 8.

Para cada *benchmark* também foi executada uma versão sequencial do algoritmo. Desta forma, totalizaram-se 1885 simulações. Uma representação gráfica de todas estas combinações pode ser encontrada na figura 4.2.

Uma breve explicação sobre a relação entre as oito aplicações do STAMP e os *benchmarks* utilizados neste trabalho (especificamente quais cargas de trabalho e nível de contenção foram executadas sobre cada aplicação), está na lista a seguir:

- `genome`
  - `genome`: execução padrão do STAMP.
  - `genome+`: execução utilizando os parâmetros `-g512 -s32 -n32768` (maior carga de trabalho).

- `intruder`
  - `intruder`: execução padrão do STAMP.
  - `intruder+`: execução utilizando os parâmetros `-a10 -l16 -n4096 -s1` (maior carga de trabalho).

- `yada`
  - `yada`: execução padrão do STAMP utilizando o conjunto de dados `633`.

- `labyrinth`
  - `labyrinth`: execução padrão do STAMP utilizando a entrada `random-x32-y32-z3-n96.txt` (chamada também de `input1` neste trabalho).
  - `labyrinth+`: execução padrão do STAMP utilizando a entrada `random-x48-y48-z3-n64.txt` (`input2` — maior carga de trabalho).
- `ssca2`
  - `ssca2`: execução padrão do STAMP utilizando o conjunto de dados `preComputed-s13-l3-p3.c`.
- `vacation`
  - `vacation-low`: execução padrão do STAMP.
  - `vacation-high`: execução utilizando os parâmetros `-n4 -q60 -u90` (maior nível de contenção).
- `bayes`
  - `bayes`: execução padrão do STAMP.
- `kmeans`
  - `kmeans-low`: execução padrão do STAMP utilizando a entrada `random-n2048-d16-c16.c`.
  - `kmeans-high`: execução utilizando a entrada `random-n2048-d16-c16.c` e os parâmetros `-m15 -n15` (maior nível de contenção).

Apesar de a plataforma MPARM suportar simulações de sistemas com até 12 núcleos, como explicado na subseção 4.1.1, as aplicações do STAMP só rodam com um número de núcleos potência de 2. Por isto, foi possível simular apenas sistemas com até 8 núcleos.

4.3.1 HTCondor

O HTCondor é um projeto de código-livre cujo objetivo é desenvolver, implementar, implantar e avaliar mecanismos e políticas que suportem *High Throughput Computing* (HTC) em grupos de máquinas com recursos computacionais distribuídos. A comunidade em torno do projeto HTCondor provê ferramentas de software que permitem a cientistas e engenheiros aumentar a quantidade de dados computados simultaneamente [4]. O

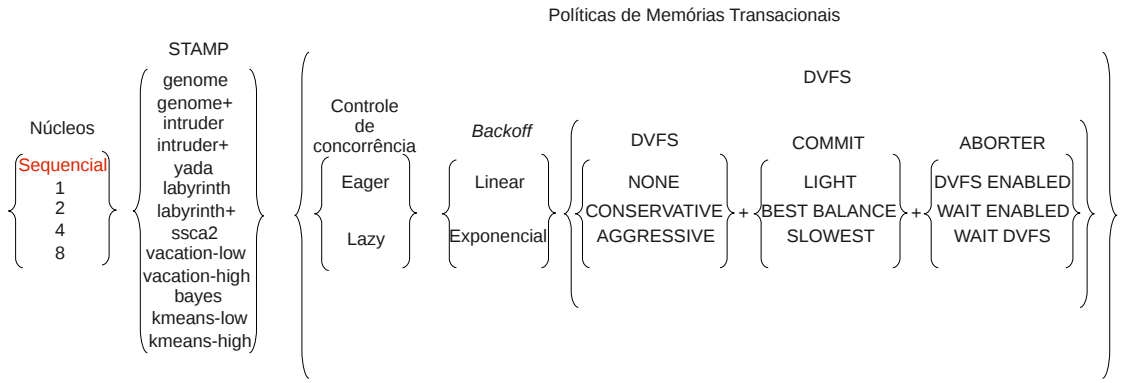


Figura 4.2: Representação de todas as simulações executadas neste trabalho, combinando 13 *benchmarks* do STAMP, 2 políticas de TM de controle de concorrência, 2 políticas de variações de *backoff* e 9 variações com DVFS, além da execução em ambientes simulados com quatro diferentes números de núcleos computacionais. Somando-se a todas estas variações, ainda foram executadas versões sequenciais de cada um dos *benchmarks* do STAMP. Estas execuções sequenciais, por não utilizarem o sistema de TM, não combinam com as diversas políticas mostradas na figura.

projeto HTCondor era conhecido apenas como Condor entre 1988 e 2012, quando ele foi renomeado para HTCondor devido a problemas legais.

O HTCondor foi utilizado para executar as inúmeras simulações deste trabalho. O HTCondor se mostrou perfeito para agilizar a execução de todas as simulações. Como um ambiente simulado foi executado, não existe nenhuma restrição em relação ao número de instâncias de simulador que poderiam executar em uma mesma máquina. Como o simulador MPARM foi escrito de forma a executar em apenas uma *thread*, independente do número de núcleos que ele esteja simulando, foi decidido executar uma instância do simulador para cada núcleo de cada máquina disponível no *cluster* HTCondor. Desta forma foi possível executar inúmeras simulações simultaneamente, agilizando a obtenção dos resultados. Mesmo assim, uma execução completa de todos os cenários exercitados em um *cluster* com mais de 100 núcleos computacionais x86_64 leva aproximadamente uma semana para ser completada. O *cluster* HTCondor utilizado no LSC conta com mais de 100 núcleos computacionais mas deve-se lembrar que, além de ele ser compartilhado com outras pessoas do laboratório, algumas vezes alguns núcleos não estavam disponíveis por estarem executando tarefas dedicadas de outros projetos. É difícil, portanto, fazer uma relação exata entre o tempo gasto para a execução de todas as simulações e quantos núcleos foram necessários para isto mas, como poderá ser visto na subseção 5.1.1, o *cluster* HTCondor agilizou sobremaneira a execução das simulações deste trabalho.

Apesar de o HTCondor disponibilizar uma série de funcionalidades que permitem agrupar diversos núcleos de uma mesma máquina e um único nó de processamento ou até mesmo interromper a execução de uma aplicação em um nó de processamento e reiniciar sua execução de onde ela parou em um outro nó (por exemplo se um nó de processamento cair por algum motivo), não foi utilizada nenhuma destas funcionalidades sofisticadas. Como dito no parágrafo anterior, como o MPARM é um simulador executado em uma única *thread*, foi utilizada a configuração padrão do HTCondor em que cada núcleo de processamento em cada máquina é definido como um nó de processamento. Também não foi feita nenhuma distinção entre os nós neste trabalho, já que todos eles eram capazes de executar o simulador por serem compatíveis com a arquitetura x86_64, mesmo sendo de microarquiteturas não necessariamente iguais. A funcionalidade de interrupção e reinício de tarefas de onde elas pararam em outros nós poderia ser interessante para este experimento, especialmente porque em simulações que podem durar até dois dias, eventualmente algum problema pode acontecer com algum dos nós de processamento. No entanto, ela não foi utilizada pois, caso esta funcionalidade fosse ativada, seria necessário recompilar o simulador MPARM com o conjunto de bibliotecas de entrada e saída disponibilizada pelo HTCondor para que as tarefas executassem no universo *standard* do HTCondor. Um universo no HTCondor nada mais é que a definição de um ambiente de execução dentro do *cluster* de nós de processamento. No universo *standard*, o HTCondor interage com as tarefas executando toda vez que elas executam entrada e saída de dados, criando pontos de checagem a partir dos quais a infraestrutura do HTCondor é capaz de reiniciar as tarefas de onde elas pararam em outro nó de processamento caso necessário. Como o simulador MPARM possui uma série de restrições sobre sua compilação como visto na subseção 4.1.1 e como ele nunca tinha sido compilado com as bibliotecas de entrada e saída do HTCondor, optou-se por utilizar a compilação padrão do simulador MPARM, utilizando as bibliotecas padrão de entrada e saída do sistema operacional. Desta forma, as execuções do simulador foram feitas no universo *vanilla* do HTCondor, o qual apenas executa as tarefas e não provê nenhuma funcionalidade de interrupção e reinicio de tarefas caso necessário. Isto fez com que fosse necessário a implementação de mecanismos de verificação da execução com sucesso das simulações, o que foi feito, no caso deste experimento, através da criação de arquivos de sinalização que indicavam se o HTCondor havia terminado com sucesso ou não uma determinada simulação. A utilização do universo *vanilla* também fez com que fosse necessário o ajuste das permissões de escrita nos diretórios onde os arquivos de saída da simulação seriam gerados. Isto porque o HTCondor executa as aplicações neste universo com permissões do usuário `nobody`. Para facilitar, no caso deste experimento, foi utilizado um sistema de arquivo que era compartilhado por todos os nós de processamento do *cluster* HTCondor e todos os diretórios utilizados tinham permissões de escrita suficientes para que as aplicações executando nos diversos

nós fossem capazes de escrever todos os arquivos de saída necessários.

O HTCondor provê também toda uma infraestrutura de controle para análise de quais tarefas estão sendo executadas em um determinado momento no *cluster*. Esta infraestrutura foi essencial durante o processo de montagem da infraestrutura de execução e para a identificação de processos que eventualmente entrassem em *livelock*. Esta infraestrutura foi usada de forma complementar aos arquivos de sinalização citados no parágrafo anterior que apenas indicavam se uma simulação havia sido completada ou não, mas não diziam se a simulação estava executando e, se este fosse o caso, onde e por quanto tempo ela estava executando.

Para identificar quais processos estão em execução em um determinado momento no *cluster* do HTCondor, foi utilizado o comando mostrado no código 4.5.

Código 4.5: Procedimento para listar processos ativos no *cluster* do HTCondor.

```
condor_q -l -run | less
```

Este comando lista quais tarefas estão executando e suas informações. Com ele é possível identificar qual *slot* — que, na configuração do *cluster* HTCondor utilizado, nada mais era que um núcleo de processamento — de qual máquina do *cluster* HTCondor está sendo utilizado para executar a simulação e qual o diretório de trabalho do comando, que, no caso deste experimento, também identificava, pelo seu nome, a política de TM que estava sendo simulada.

Com estas informações, é possível acessar a máquina específica que está executando a simulação no *cluster* HTCondor e capturar o comando completo que gerou a execução com o comando `ps`. Note que, como todas as execuções deste trabalho são simulações, acessar a máquina não gera nenhum impacto nas informações sendo coletadas durante a execução.

Enfileiramento de tarefas no HTCondor

O *script* citado no início da seção 4.3 basicamente gera um arquivo de *submit* do HTCondor. Este arquivo contém a especificação das linhas de comando que precisam ser executadas em cada nó de processamento do *cluster* (também chamado de *slot*). Posteriormente, este *script* chama o aplicativo `condor_submit` para este arquivo de *submit*. Isto insere todas as chamadas na fila de execução do *cluster* HTCondor.

Um ponto de atenção neste processo foi a passagem de parâmetros para os comandos que eram descritos no arquivo de *submit* do HTCondor. Como alguns destes parâmetros precisavam estar descritos entre aspas duplas, foi necessário tomar cuidado com a geração dos arquivos de `submit` do HTCondor pelo *script* para que os parâmetros fossem corretamente passados para as instâncias sendo executadas nos nós de processamento do

cluster HTCondor. Em relação à passagem de parâmetros, uma dica é sempre verificar a documentação do HTCondor relativa à exata versão do HTCondor que está sendo utilizada já que a passagem de parâmetros foi alterada entre diferentes versões do HTCondor, especialmente no que diz respeito ao uso de aspas para a passagem de parâmetros.

Ambiente de execução no *cluster* HTCondor

Os nós de execução do HTCondor não possuíam o ambiente Fedora 14 descrito na subseção 4.1.1 instalado. Eles possuíam uma versão de Ubuntu 10.04 32 bits. Assim, o simulador foi inicialmente compilado no ambiente Fedora 14 e os binários foram copiados para os nós do HTCondor para posterior execução das aplicações STAMP sobre o simulador. No ambiente de execução não é necessária a instalação de nenhuma ferramenta de compilação do ARM RTEMS GCC, bastando que cada nó de processamento tenha acesso ao executável gerado para a plataforma ARM, o que, como visto no início desta seção, foi possível pelo fato de todos os nós terem acesso ao mesmo sistema de arquivos compartilhado, onde eram encontradas todas as informações necessárias para a execução de todas as simulações.

Capítulo 5

Resultados das simulações

A execução dos aplicativos do STAMP no simulador MPARM sobre a infraestrutura do HTCondor descrita na seção 4.3 gera vários arquivos de saída. Por exemplo, para uma execução de bayes, com 2 núcleos, tem-se:

- `bayes-2c.condor_output`: arquivo com a saída padrão (`stdout`) gerada pela execução do aplicativo no nó HTCondor.
- `bayes-2c.condor_error`: arquivo com a saída padrão de erro (`stderr`) gerada pela execução do aplicativo no nó HTCondor.
- `bayes-2c.txt`: arquivo com um resumo de estatísticas sobre a execução do programa no ambiente simulado.
- `bayes-2c-finegrained.txt`: arquivo com uma estatística detalhada sobre a execução do programa, especificando qual tipo de operação de memória transacional está acontecendo em um determinado momento da execução em um determinado núcleo de processamento sendo simulado, quanto tempo esta execução gastou e quanto de energia ela consumiu para ser executada.

Um exemplo de cada um destes arquivos pode ser visto no apêndice A. O principal foco neste experimento foi nos arquivos `*.condor_output` e `*-finegrained.txt`. Nos arquivos que registravam a saída padrão da execução dos *benchmarks* do STAMP podia-se verificar se a execução foi completada com sucesso. Para isto foi desenvolvido um *script* que checava automaticamente estes arquivos e dizia se havia ocorrido algum erro durante a execução ou não. Isto se fez necessário dado o grande número de simulações que foram executadas durante a implementação desta dissertação (mais de dez mil simulações contando testes, falhas e execuções com sucesso). Já os arquivos com as estatísticas detalhadas sobre a execução dos *benchmarks* (arquivos *finegrained*) foram devidamente

processados por *scripts* que consolidavam as informações neles contidas em dados mais inteligíveis.

A geração dos arquivos *finegrained* foi um trabalho a parte. Para que isto fosse possível, a biblioteca TL2 foi instrumentada como explicado na subseção 4.2.3. Assim, foi possível traçar um quadro detalhado de informações sobre a execução do simulador para cada um dos *benchmarks* analisados.

O suporte padrão do simulador MPARM para registro de estatísticas detalhadas sobre a execução registra quatorze campos de dados em cada linha dos arquivos *finegrained*. Este suporte foi alterado no código do simulador MPARM para que, ao invés dos 14 campos originais, apenas 12 fossem registrados, conforme será descrito na seção A.4. Os dois campos removidos (`d-spm energy` e `i-spm energy`) registravam os valores de energia gasta em memórias do tipo *scratchpad* de dados e instruções. Como a plataforma que foi usada na simulação deste experimento não possui memórias *scratchpad*, estes valores eram sempre registrados como zero. Desta forma, não houve perda de informação ao remover estes dados dos arquivos de saída e foi possível diminuir os arquivos gerados pelas simulações. Além disto, como o simulador precisa ser compilado com um compilador de 32 bits como foi visto na subseção 4.1.1, o tamanho limite de arquivos gerados por ele durante sua execução é de 2 GB. Acima deste tamanho, o simulador passa a sobrescrever dados do arquivo de saída a partir do seu início. Como algumas simulações executadas eram muito longas, este limite de 2 GB estava sendo atingindo muitas vezes com o registro dos quatorze campos de dados. Com a retirada dos campos relativos às memórias *scratchpad* este problema foi quase que completamente eliminado como poderá ser visto posteriormente neste capítulo.

Como, mesmo assim, estes arquivos são muito grandes, facilmente atingindo a ordem de gigabytes de dados, depois de sua geração eles eram compactados com a ferramenta `bzip2`. O *script* de processamento destes arquivos é capaz de lê-los tanto em formato texto ou diretamente do arquivo `bz2` gerado a partir do arquivo *finegrained* originalmente produzido pelo simulador MPARM. Apenas a título de exemplo, uma massa de dados completa das 1885 simulações descritas na seção 4.3 ocupa cerca de 200 GB depois de todos os arquivos de saída compactados. Todos os dados gerados durante este trabalho ultrapassaram 1,4 TB. Lidar com esta quantidade de dados e fazer o devido *backup* das mesmas se mostrou um desafio a parte durante a execução deste trabalho.

Após o processamento dos arquivos *finegrained* por *scripts* são gerados quatro arquivos. Por exemplo, para uma execução de bayes, com 2 núcleos, tem-se:

- `bayes-2c-operations-output.txt`: arquivo com a saída padrão (`stdout`) da execução do *script* de processamento do arquivo *finegrained*. Este arquivo contem um resumo das operações de TM executadas e quanto de energia foi gasto por tipo de

operação, além de outras informações bem resumidas sobre as operações executadas durante a simulação.

- `bayes-2c-operations.txt`: arquivo que registra um resumo das informações mais importantes do arquivo *finegrained*. Serão referenciados como arquivo *operations* neste texto.

Além destes dois arquivos para cada um dos *benchmarks* em cada uma das configurações de TM, também são gerados dois arquivos com resultados totalizados da simulação:

- `results-globals.dat`: Para cada um dos *benchmarks* em cada uma das configurações de TM e de número de núcleos utilizados durante a execução do aplicativo, este arquivo contem dados totalizados sobre a execução relativos ao consumo de energia e tempo gasto na execução.

- `results-primitives.dat`: Para cada um dos *benchmarks* em cada uma das configurações de TM e de número de núcleos utilizados durante a execução do aplicativo, este arquivo contem dados totalizados relativos ao consumo de energia e tempo gasto na execução de cada operação primitiva de TM.

Um exemplo de cada um destes arquivos pode ser encontrado no apêndice B.

Em um determinado momento, pensou-se em importar os dados contidos nos arquivos *finegrained* e/ou *operations* para uma base de dados, de forma que fosse mais fácil fazer consultas diversas sobre os dados. No entanto, pelo menos dois problemas foram notados:

- O conjunto de dados gerados pelas simulações era muito grande, especialmente os arquivos *finegrained*, atingindo quase 400 GB de dados após processados (sem compactação). Mesmo considerando apenas os arquivos *operations*, o total de dados produzidos fica em torno de 21 GB. Migrar todos estes dados para uma base de dados consumiria um bom tempo e não seria trivial indexar e administrar uma base de dados deste tamanho.

- O tempo gasto para processar os arquivos é relativamente pequeno se comparado com o tempo total gasto nas simulações dos *benchmarks* do STAMP utilizando o MPARM. Desta forma, optou-se pelo uso de *scripts* para o processamento dos arquivos a fim de automatizar consultas e geração de gráficos com a ferramenta `gnuplot`. Além disto, o processamento dos arquivos pode ser feito de forma paralela em máquinas com vários núcleos ou pode, até mesmo, ser distribuída em um *cluster* HTCondor para acelerar o processamento.

A análise de toda esta massa de dados gerada pelas simulações foi feita através do estudo de gráficos que traduziam o desempenho e consumo energético de cada política de TM proposta para cada *benchmark* do STAMP. É importante notar que, apesar da utilização de DVFS em várias políticas de TM, os valores mensurados de tempo de execução pelo simulador utilizam um relógio de período fixo. O período deste relógio é de 5 ns e reflete a menor granularidade possível para a frequência do processador ARM simulado (200 MHz). Caso a frequência de operação seja menor que 200 MHz, o tempo gasto na execução de um ciclo de processamento do processador será a unidade básica de tempo (5 ns) vezes o divisor de frequência utilizado naquele determinado momento. Desta forma, as medidas obtidas após o processamento dos dados estatísticos capturados das execuções das simulações refletem o número de ciclos de 5 ns executados pelo simulador (independente da frequência de operação). Este valor pode ser facilmente convertido para uma unidade de tempo.

5.1 Simulações executadas

Durante o trabalho desenvolvido nesta dissertação, mais de dez mil simulações foram executadas para testar a infraestrutura do HTCondor e as diversas políticas. Nesta seção será descrito brevemente qual era o propósito de cada conjunto de simulações executado no HTCondor e qual a conclusão de cada um deles. Outras simulações isoladas para testar conceitos foram executadas, mas estas serão discutidas apenas em casos em que elas foram necessárias para elucidar quaisquer fatos encontrados durante a execução dos conjuntos de simulações gerenciados pelo HTCondor.

Cada conjunto de simulação é identificado por uma data no formato `DDMMAAHHmm` e um número identificador do conjunto de tarefas enfileiradas no HTCondor.

5.1.1 Simulação 2809111232 (104)

Conjunto de simulações originais feitas em [13]. Como esta dissertação seria uma extensão do trabalho apresentado nesse artigo, o primeiro passo era integrar o conjunto de simulações executadas para esse artigo na infraestrutura do HTCondor implementada para este trabalho.

A integração com o HTCondor e o *cluster* de máquinas do LSC foi de primordial importância para este trabalho. As simulações para o artigo [13] foram executadas em apenas duas máquinas e controladas uma a uma. Este processo demorou cerca de um mês. Com o HTCondor, todo o processo não passou de três dias.

O conjunto de políticas exploradas e simulações executadas estão representados na figura 5.1. Em *cinza e itálico* estão marcadas as políticas não exploradas neste conjunto

de simulações.

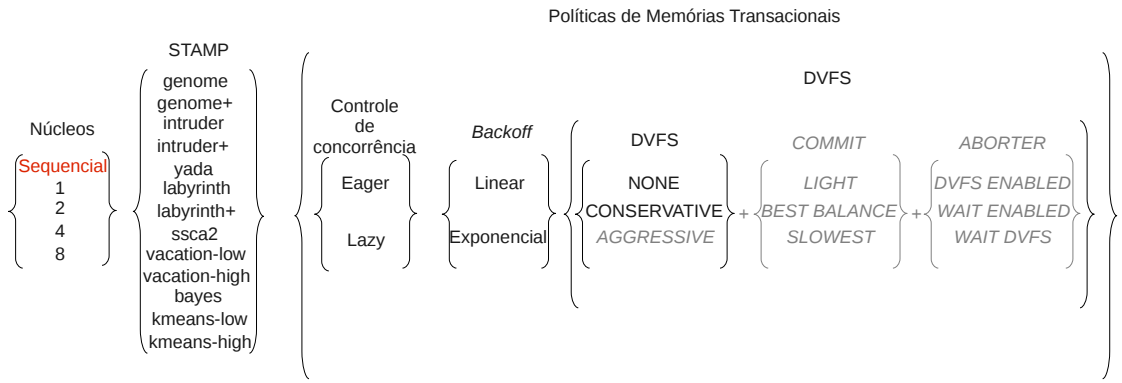


Figura 5.1: Representação de todas as simulações executadas neste conjunto, combinando 13 *benchmarks* do STAMP, 2 políticas de TM de controle de concorrência, 2 políticas de variações de *backoff* e 2 variações com DVFS, além da execução em ambientes simulados com quatro diferentes números de núcleos computacionais. Somando-se a todas estas variações, ainda foram executadas versões sequenciais de cada um dos *benchmarks* do STAMP. Desta forma, neste conjunto totalizou-se 429 simulações.

A execução da simulação do *benchmark* yada em 8 núcleos com a política de TM `BACKOFF_LINEAR Eager` não terminou com o erro "`mparm_sbrk - reserving memory failed`", que indica que houve falha na alocação de memória por falta de memória. Este erro foi reproduzível ao executar esta aplicação manualmente (sem o *cluster* HTCondor) ou mesmo em outros conjuntos de simulação através do HTCondor. O mesmo problema também será encontrado em algumas simulações futuras. Uma análise deste problema e a solução encontrada para ele poderão ser encontradas na subseção 5.1.5.

5.1.2 Simulação 2909111508 (105)

Este conjunto de simulações foi o primeiro a explorar DVFS durante o *commit* (`DVFS_COMMIT`). Nele apenas os *benchmarks* `genome`, `genome+`, `intruder` e `intruder+` foram executados.

O conjunto de políticas exploradas e simulações executadas estão representados na figura 5.2. Em *cinza e itálico* estão marcadas as políticas não exploradas neste conjunto de simulações.

Antes de executar este conjunto de simulações, durante a implementação das políticas `DVFS_COMMIT`, alguns problemas foram encontrados. Após uma melhor análise das modificações iniciais feitas no código da TL2 para adicionar suporte à política `DVFS_COMMIT`

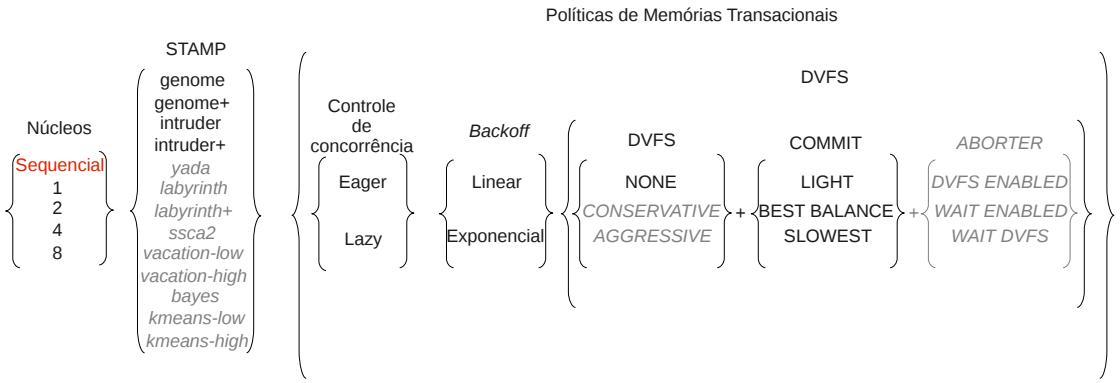


Figura 5.2: Representação de todas as simulações executadas neste conjunto, combinando 4 *benchmarks* do STAMP, 2 políticas de TM de controle de concorrência, 2 políticas de variações de *backoff* e 4 variações com DVFS, além da execução em ambientes simulados com quatro diferentes números de núcleos computacionais. Somando-se a todas estas variações, ainda foram executadas versões sequenciais de cada um dos *benchmarks* do STAMP. Desta forma, neste conjunto totalizou-se 260 simulações.

verificou-se que a alteração feita no código não estava baseada em premissas totalmente verdadeiras.

O objetivo das políticas `DVFS_COMMIT` é explorar a tese que diz que abaixar a frequência de execução do processador em trechos de código com muito acesso à memória economiza energia sem impactar consideravelmente o desempenho da execução. O código do *commit* na TL2 é dividido em duas partes:

- Na parte inicial, é verificado se a transação em questão apenas leu o valor de variáveis na memória. Se isto for verdade, um código mais simples é executado.
- Caso isto não seja verdade e a transação tenha também escrito novos valores na memória, um outro código, mais complexo, é executado.

Inicialmente achou-se que apenas a região que verificava se escritas tinham acontecido deveria ter a variação de frequência ativada, pois não parecia ser relevante a quantidade de acessos à memória executados na primeira parte do *commit*. No entanto, posteriormente, notou-se que isto não era verdade e alterou-se o código para que a política de DVFS fosse ativa durante toda a operação de *commit*. Foi este código já alterado que foi executado neste conjunto de simulações.

Este conjunto de simulações também demonstrou problemas de execução nas simulações com política `DVFS_COMMIT_SLOWEST`. Como será explicado na subseção 5.2.1, esta política foi classificada como não eficaz e, portanto, foi removida de futuras simulações.

5.1.3 Simulação 2909111756 (106)

Este conjunto de simulações tentou explorar a política `DVFS_AGGRESSIVE`. Como será visto na subseção 5.2.2, esta política não se mostrou eficaz e, portanto, foi removida de futuras simulações.

5.1.4 Simulação 2909111845 (107)

Neste caso tentou-se executar os aplicativos sem diretivas nenhumas sobre o sistema de TM. Esta execução falhou. Posteriormente, analisando o código, concluiu-se que, na verdade, não fazia sentido este tipo de execução porque a STM sempre esperava algum tipo de diretiva de execução. Nenhuma diretiva, o que significava a mesma coisa que executar o código original da TL2, era a mesma coisa que executar a STM apenas com *backoff* linear ativado, o que já estava sendo feito em outros conjuntos de simulações.

5.1.5 Simulação 2410111829 (120)

Este conjunto de simulações foi uma tentativa de utilizar o *cluster* HTCondor para executar um conjunto grande das políticas até então implementadas: políticas do artigo [13] (sem o caso `DVFS_CONSERVATIVE`, que foi excluído por simplificação) mais as políticas `DVFS_COMMIT` (sem o caso `SLOWEST`). Desejava-se testar o *cluster* HTCondor com um número maior de simulações e, ao mesmo tempo, verificar como poder-se-ia validar uma execução de uma grande massa de simulações.

O conjunto de políticas exploradas e simulações executadas estão representados na figura 5.3. Em *cinza e itálico* estão marcadas as políticas não exploradas neste conjunto de simulações. As políticas excluídas por não serem viáveis estão riscadas.

Durante a análise dos resultados deste conjunto de simulações notou-se que algumas execuções que aparentemente tinham terminado com sucesso, na verdade apresentavam erros nos arquivos que registravam a saída padrão da execução do *benchmark*. Isto provavelmente também aconteceu em conjuntos de simulações anteriores. Como cada um destes conjuntos continha uma grande quantidade de simulações, não era viável uma inspeção manual dos arquivos de saída de cada uma das simulações. A partir desta constatação surgiu a necessidade de se construir um *script* para checar automaticamente se os arquivos de saída padrão da execução dos programas continham algum erro ou se continham a saída esperada para a execução de cada um dos *benchmarks* do STAMP com sucesso.

Com isto, depois de 3,5 dias executando, alguns tipos de problemas foram detectados.

A execução do *benchmark* `intruder+` em 4 núcleos com a política de TM `BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager` conclui a execução em três núcleos mas nunca conclui a execução no quarto núcleo. Esta tarefa não apresentou o mesmo pro-

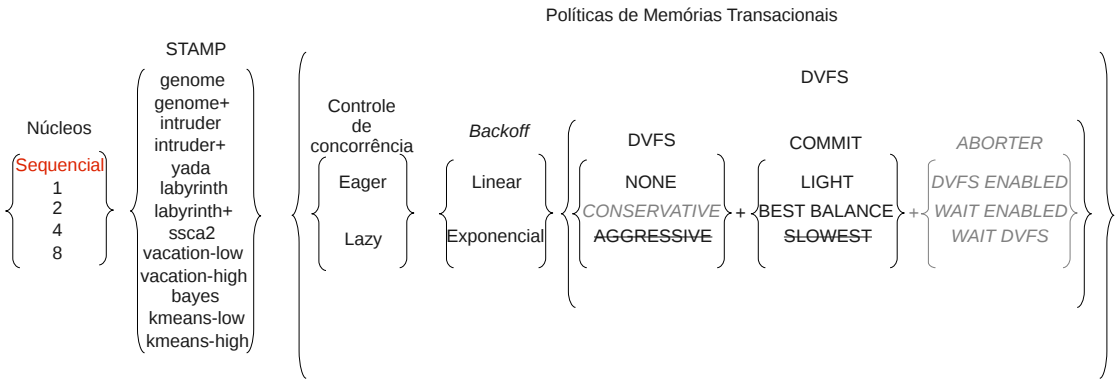


Figura 5.3: Representação de todas as simulações executadas neste conjunto, combinando 13 *benchmarks* do STAMP, 2 políticas de TM de controle de concorrência, 2 políticas de variações de *backoff* e 3 variações com DVFS, além da execução em ambientes simulados com quatro diferentes números de núcleos computacionais. Somando-se a todas estas variações, ainda foram executadas versões sequenciais de cada um dos *benchmarks* do STAMP. Desta forma, neste conjunto totalizou-se 637 simulações.

blema em outros conjuntos de simulações, então suspeita-se de algum incidente no *cluster* HTCondor durante a execução desta tarefa neste conjunto de simulações.

Além disto, algumas tarefas haviam parado sua execução com erro de falta de memória (`mparm_sbrk - reserving memory failed`). A tabela 5.1 mostra as simulações nesta situação. Uma análise mais detalhada deste erro chegou à conclusão de que ele ocorria porque a quantidade de memória compartilhada configurada no processador ARM era insuficiente para as aplicações sendo executadas. Esta quantidade, que por padrão estava configurada como 1 MB, foi alterada para 12 MB posteriormente neste trabalho. Um conjunto de simulações foi executado com esta modificação (veja subseção 5.1.10).

Outras quatro simulações tiveram um erro de mapeamento de endereços em memória (por exemplo, `Fatal error: Address 0x411c7109 cannot be mapped to any existing slave, nor (if present) to DMA/scratchpad!`, sendo que o endereço de memória sempre variava sem nenhum padrão aparente). A tabela 5.2 mostra os detalhes das simulações que apresentaram este problema. Todas estas simulações foram reexecutadas manualmente posteriormente (sem o uso do HTCondor) e apresentaram o mesmo problema.

Por último, a execução do *benchmark* `intruder+` em 8 núcleos com a política de TM `BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Lazy` apresentava arquivo *finegrained* que tinha atingido o tamanho máximo de 2 GB e, portanto, estava tendo seu conteúdo sobrescrito.

Tabela 5.1: Simulações que terminaram por falta de memória.

Benchmark	**Número de núcleos simulados**	**Política de TM**
yada	8	BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager
yada	4	BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager
yada	8	BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager

Tabela 5.2: Simulações que terminaram por impossibilidade de mapeamento de endereço.

Benchmark	**Número de núcleos simulados**	**Política de TM**
yada	2	BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager
yada	8	BACKOFF_LINEAR Eager
yada	4	BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager
intruder	8	BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager

5.1.6 Simulação 2710111046 (121)

Este conjunto de simulações foi o primeiro a explorar a política `DVFS_ABORTER_ENABLED`. Como será visto na subseção 5.2.2, esta política não se mostrou eficaz e, portanto, foi removida de futuras simulações.

5.1.7 Simulação 0111111159 (173)

Neste caso explorou-se a política `WAIT_ABORTER_ENABLED`. Esta política, que basicamente coloca em estado de espera o núcleo que esteja executando uma transação que aborta seguidamente, surgiu da percepção de que, em casos de alta contenção entre transações, a execução sequencial de cada uma das transações é mais rápida e eficiente que a execução transacional. Assim, se a contenção em um determinado momento for muito grande entre as diversas transações sendo executadas, estas vão eventualmente degenerar para o caso de execução sequencial (com todas as transações em estado de espera menos uma, que

finalmente conseguirá finalizar seu trabalho).

O conjunto de políticas exploradas e simulações executadas estão representados na figura 5.4. Em *cinza e itálico* estão marcadas as políticas não exploradas neste conjunto de simulações. As políticas excluídas por não serem viáveis estão riscadas.

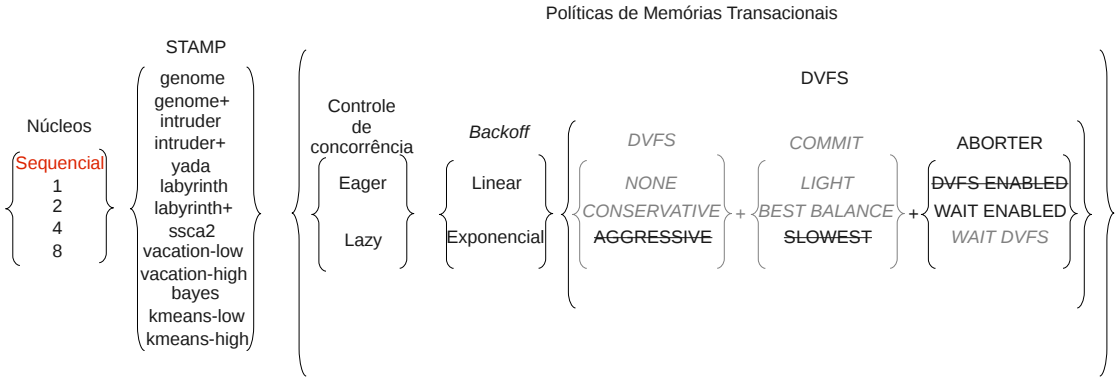


Figura 5.4: Representação de todas as simulações executadas neste conjunto, combinando 13 *benchmarks* do STAMP, 2 políticas de TM de controle de concorrência, 2 políticas de variações de *backoff* e 1 variação com DVFS, além da execução em ambientes simulados com quatro diferentes números de núcleos computacionais. Somando-se a todas estas variações, ainda foram executadas versões sequenciais de cada um dos *benchmarks* do STAMP. Desta forma, neste conjunto totalizou-se 221 simulações.

Uma outra evidência também corroborou para a elaboração desta política. Em uma primeira implementação de teste da política `DVFS_ABORTER_ENABLED` que foi executada apenas em algumas poucas simulações de forma manual (sem ajuda do HTCondor), ao invés de utilizar apenas os dois patamares de frequência de execução descritos na seção 3.3 (`HALF_SPEED` e `BEST_BALANCE`), caso a contenção continuasse acontecendo, o que acabou se provando muito corriqueiro até mesmo no caso da política `DVFS_ABORTER_ENABLED` utilizada nos experimentos com o HTCondor conforme será descrito na subseção 5.2.2, a frequência de execução do núcleo era baixada para `SLOWEST`. Nesta implementação de teste da política `DVFS_ABORTER_ENABLED` também eram utilizados valores bem pequenos para os limiares que acionavam a política. Com isto, os núcleos que executavam transações com alta contenção rapidamente passavam a operar na frequência `SLOWEST`. Como esta frequência é muito mais baixa que as outras frequências utilizadas (`FULL_PERFORMANCE`, `HALF_SPEED` e `BEST_BALANCE`), isto significava que um núcleo executando na frequência `SLOWEST` estava virtualmente parado em relação aos núcleos nas outras frequências. Neste cenário foi possível notar que, em determinados momentos, uma transação atingia a frequência `SLOWEST` (devido à alta contenção) enquanto o outro núcleo continuava

executando em uma das outras frequências utilizadas, com casos em que a execução degenerava-se e apenas um núcleo finalizava inúmeras transações em sequência, enquanto o outro núcleo não progredia, abortando sua transação seguidamente e, devido a isto, permanecendo na frequência `SLOWEST`. Este cenário prosseguia até os dados a serem consumidos finalizarem e todas as transações terminarem menos a que estava executando na frequência `SLOWEST` que, aí sim, conseguia terminar seu trabalho.

Esta implementação de teste da política `DVFS_ABORTER_ENABLED` acabou influenciando duas novas políticas:

- A política `DVFS_ABORTER_ENABLED` foi implementada da forma como descrita na seção 3.3, sem o uso da frequência `SLOWEST` e com limiares de contenção para acionamento da política maiores. Desta forma, as transações demoravam um pouco mais de tempo para terem sua frequência de execução diminuída e, quando isto acontecia, elas ainda continuavam executando em uma frequência compatível com a frequência de execução nos outros núcleos. Infelizmente estas mudanças acabaram não fornecendo melhora alguma no desempenho do sistema de TM, gerando até mesmo casos de *livelock*.

- A política `WAIT_ABORTER_ENABLED` ganhou força, já que na prática a implementação de teste da política `DVFS_ABORTER_ENABLED` estava se degenerando para um caso muito parecido com o que seria atingido com a política `WAIT_ABORTER_ENABLED`, onde uma transação fica parada caso um cenário de alta contenção seja atingido. Esta transação é novamente executada apenas quando a transação conflitante já estiver finalizada.

Nos testes preliminares feitos a política `WAIT_ABORTER_ENABLED` mostrou-se bastante promissora, especialmente pela forma como ela foi capaz de não permitir que várias transações ficassem em contenção durante muito tempo.

5.1.8 Simulação 0911111611 (175)

Este conjunto de simulações testou a política `WAIT_ABORTER_DVFS`. O conjunto de políticas exploradas e simulações executadas estão representados na figura 5.5. Em *cinza e itálico* estão marcadas as políticas não exploradas neste conjunto de simulações. As políticas excluídas por não serem viáveis estão riscadas.

Com os bons resultados da política `WAIT_ABORTER_ENABLED`, gerou-se uma expectativa em relação a esta política já que ambas são muito similares mas, no caso da política `WAIT_ABORTER_DVFS`, o núcleo em estado de espera devido à alta contenção tem sua frequência de execução diminuída, o que, empiricamente, levaria esta variação a ter um menor consumo de energia.

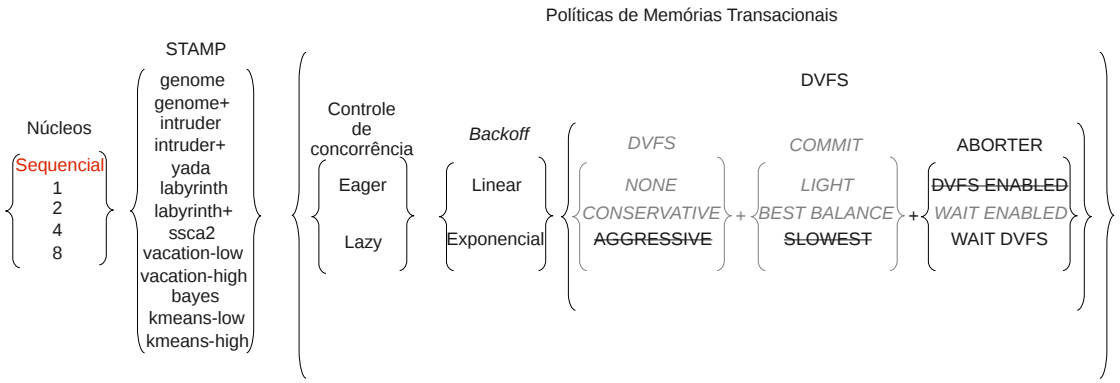


Figura 5.5: Representação de todas as simulações executadas neste conjunto, combinando 13 *benchmarks* do STAMP, 2 políticas de TM de controle de concorrência, 2 políticas de variações de *backoff* e 1 variação com DVFS, além da execução em ambientes simulados com quatro diferentes números de núcleos computacionais. Somando-se a todas estas variações, ainda foram executadas versões sequenciais de cada um dos *benchmarks* do STAMP. Desta forma, neste conjunto totalizou-se 221 simulações.

5.1.9 Simulação 1102121722 (267)

Este conjunto de simulações incluiu todas as políticas anteriores que foram testadas, inclusive aquelas que apresentaram problemas, com exceção da `DVFS_COMMIT_SLOWEST`. Com isto tentou-se obter um único conjunto de dados com todas as variações analisadas durante o trabalho a fim de se certificar que os problemas identificados anteriormente eram reproduzíveis. O conjunto de políticas exploradas e simulações executadas estão representados na figura 5.6. A política excluída por não ser viável está riscada.

Depois de 9 dias executando, várias das tarefas deste conjunto de simulações que utilizavam as políticas `DVFS_AGGRESSIVE` e `DVFS_ABORTER_ENABLED` ainda não haviam terminado, sendo que todas as simulações problemáticas apresentavam, neste estágio, arquivo *finegrained* que tinha atingido o tamanho máximo de 2 GB e, portanto, estava tendo seu conteúdo sobrescrito. A lista de simulações nesta situação poderá se encontrada no apêndice D. Isto confirma o problema anteriormente detectado com estas políticas. A execução do *benchmark* `vacation-high` em um núcleo com as políticas de TM `BACKOFF_LINEAR DVFS_AGGRESSIVE Eager` e `BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager` também apresentou erro (“`Core:  Shifter distance more than 31, i.e:46`”).

Além disto, outras poucas execuções não utilizavam políticas `DVFS_AGGRESSIVE` ou `DVFS_ABORTER_ENABLED` mas também tiveram problemas. Todas as políticas nesta situação também deram problemas no conjunto de simulações que será descrito na subseção

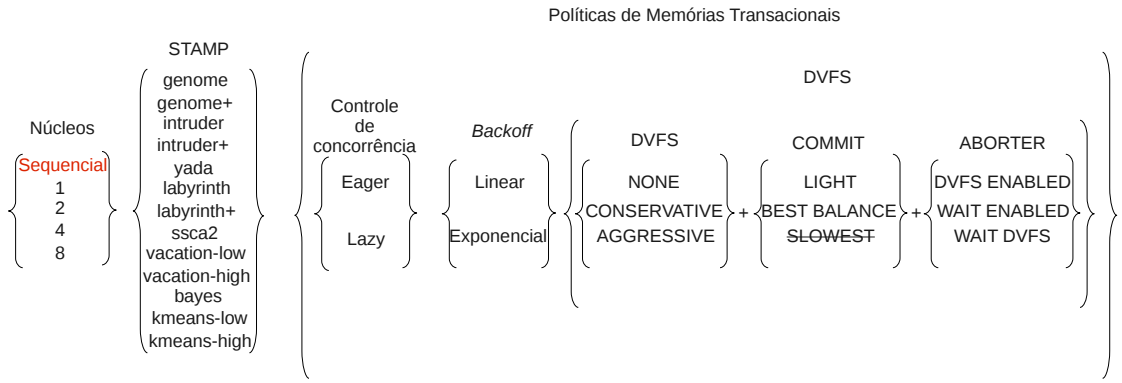


Figura 5.6: Representação de todas as simulações executadas neste conjunto, combinando 13 *benchmarks* do STAMP, 2 políticas de TM de controle de concorrência, 2 políticas de variações de *backoff* e 8 variações com DVFS, além da execução em ambientes simulados com quatro diferentes números de núcleos computacionais. Somando-se a todas estas variações, ainda foram executadas versões sequenciais de cada um dos *benchmarks* do STAMP. Desta forma, neste conjunto totalizou-se 1677 simulações.

5.1.10 e estes problemas estão lá explicados.

As outras simulações, como esperado, completaram sua execução com sucesso.

5.1.10 Simulação 1202122257 (284)

Este conjunto de simulações incluiu todas as políticas anteriores que foram testadas com sucesso. O conjunto de políticas exploradas e simulações executadas estão representados na figura 5.7. As políticas excluídas por não serem viáveis estão riscadas.

Neste conjunto de simulações foi executado um simulador MPARM com uma correção para o problema de alocação de memória enfrentado em algumas simulações anteriores. Este problema ocorria porque a quantidade de memória compartilhada configurada no processador ARM era insuficiente para as aplicações sendo executadas. Esta quantidade, que por padrão estava configurada como 1 MB, foi alterada para 12 MB neste conjunto de simulações.

Assim, o objetivo final era gerar uma massa de dados que levava em consideração todas as descobertas feitas durante as milhares de simulações feitas anteriormente. Com isto obteve-se um único conjunto de dados com todas as variações válidas possíveis a serem analisadas nesta dissertação para facilitar o processamento final dos dados e a respectiva análise dos mesmos.

Mesmo assim, algumas poucas das 1261 simulações apresentaram problemas já detec-

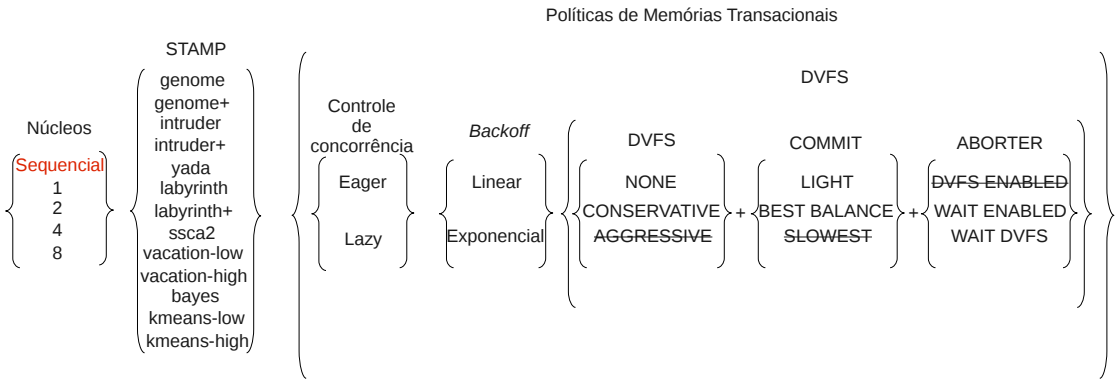


Figura 5.7: Representação de todas as simulações executadas neste conjunto, combinando 13 *benchmarks* do STAMP, 2 políticas de TM de controle de concorrência, 2 políticas de variações de *backoff* e 6 variações com DVFS, além da execução em ambientes simulados com quatro diferentes números de núcleos computacionais. Somando-se a todas estas variações, ainda foram executadas versões sequenciais de cada um dos *benchmarks* do STAMP. Desta forma, neste conjunto totalizou-se 1261 simulações.

tados anteriormente.

Algumas políticas tiveram um erro de mapeamento de endereços de memória (por exemplo, `Fatal error: Address 0x19400fdc cannot be mapped to any existing slave, nor (if present) to DMA/scratchpad!`), sendo que o endereço de memória sempre variava sem nenhum padrão aparente). Os detalhes das simulações que apresentaram este problema podem ser encontrados na tabela 5.3.

Apesar de todas as execuções da tabela 5.3 terem gerado erro na simulação descrita na subseção 5.1.9, a execução do *benchmark* `yada` em 8 núcleos com as políticas `BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager` e `BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager` apresentava antes o erro de falta de memória encontrado anteriormente também na subseção 5.1.5 (`mparm_sbrk - reserving memory failed`). Este erro foi corrigido como descrito anteriormente nesta subseção, mas estas simulações acabaram esbarrando em um outro problema. As outras duas simulações da tabela 5.3 tiveram o mesmo erro tanto neste conjunto de simulações quanto naquele descrito na subseção 5.1.9.

Este erro de mapeamento de memória nestas simulações já tinha sido encontrado outras vezes em outros conjuntos de simulações e também durante a execução de alguns programas de teste distribuídos junto com o MPARM. Foi notado, no entanto, que este erro é bem inconstante e pode desaparecer em execuções subsequentes da mesma aplicação no simulador, o que indica algum defeito no MPARM. No entanto, no caso das quatro

Tabela 5.3: Simulações que terminaram por impossibilidade de mapeamento de endereço.

Benchmark	**Número de núcleos simulados**	**Política de TM**
`yada`	8	`BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager`
`yada`	8	`BACKOFF_LINEAR Eager`
`yada`	8	`BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager`
`intruder`	4	`BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager`

simulações acima, não se conseguiu executá-las nenhuma vez sem que o erro fosse visto depois que a correção para o defeito de falta de memória — que afetava alguma das simulações acima — foi implementado.

Aliás, esta inconstância descrita no parágrafo anterior do MPARM é a causa mais provável para o fato de o conjunto de simulações com erros descrito na subseção 5.1.5 não ser exatamente o mesmo do conjunto de simulações com erros descrito nesta subseção, apesar de o conjunto de simulações com erros desta subseção ser exatamente o mesmo que o encontrado na subseção 5.1.9, depois de retirados os casos patológicos que estão listados no apêndice D. Por isto, como as simulações apresentadas nesta subseção cobrem todas as políticas viáveis sendo analisadas neste trabalho e como este é o conjunto de simulações que menos erros de execução teve, além do fato de não se ter conseguido reexecutar sem erros estas simulações após a implementação da correção para o defeito de falta de memória, este conjunto de simulações foi o conjunto considerado na análise final deste trabalho.

Além disto, algumas simulações, mesmo tendo chegado ao final da execução, geraram erros ao tentar checar os resultados com os resultados esperados pelo STAMP. A lista com os detalhes das simulações que apresentaram este problema pode ser encontrada na tabela 5.4. Todas estas simulações foram executadas novamente manualmente, sem a ajuda do *cluster* HTCondor, e apresentaram o mesmo erro. Isto mostra, por outro lado, alguma incerteza nas aplicações do STAMP além das já detectadas e documentadas na subseção 4.2.2.

Por fim, o arquivo *finegrained* da execução do *benchmark* `intruder+` em 8 núcleos com políticas de TM `BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Lazy` e `BACKOFF_LINEAR DVFS_COMMIT_LIGHT Lazy` atingiu o tamanho máximo de 2 GB e, portanto, teve seu conteúdo sobrescrito. Isto indica que estas simulações, com estas políticas, possuem execuções

Tabela 5.4: Simulações cuja checagem de resultados foi incompatível com o resultado esperado pelo STAMP.

Benchmark	**Número de núcleos simulados**	**Política de TM**	**Erro de checagem**
`yada`	2	`BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager`	*Final mesh is not valid!*
`yada`	8	`BACKOFF_LINEAR DVFS_CONSERVATIVE Eager`	*Final mesh is not valid!*
`intruder`	4	`BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager`	*'Num attack' and 'Num found' are different!*

muito extensas que vão além da capacidade de armazenamento suportadas no modelo de simulação utilizado neste trabalho.

Desta forma, das 1261 simulações executadas neste conjunto, 9 (cerca de 0,7%) apresentaram algum tipo de problema durante a execução que torna os dados coletados para elas não confiáveis. Estas simulações que apresentaram erros estarão devidamente identificadas na análise final dos dados.

5.2 Políticas não eficazes

Através da análise da execução das simulações e dos dados gerados pelas mesmas nos arquivos *finegrained* foi possível verificar que algumas políticas baseadas em DVFS não foram eficazes e levaram a execução dos *benchmarks* a estados não desejados, principalmente de *livelock*.

5.2.1 `DVFS_COMMIT_SLOWEST`

Apesar de o HTCondor gerenciar de forma muito competente a execução das simulações dentro do *cluster*, o acompanhamento da execução das tarefas conforme descrito na subseção 4.3.1 é primordial para detectar possíveis problemas.

Por exemplo, ainda durante a execução das simulações que implementavam a política `DVFS_COMMIT_SLOWEST` foi possível detectar que alguma coisa errada com esta política estava acontecendo. Isto porque algumas simulações não terminavam mesmo depois de mais de uma semana executando, quando, em geral, as simulações mais demoradas levavam no máximo 3 dias para completarem a execução.

A suspeita foi que a diminuição da velocidade de execução do *commit* para a frequência `SLOWEST` acabou fazendo com que a operação de *commit* ficasse muito lenta em relação às outras partes de código executando na frequência `FULL_PERFORMANCE`. Isto acabava gerando um cenário de alta contenção: as operações de *commit* demoravam tanto tempo para executar que acabavam conflitando em algum ponto com alguma outra transação em execução.

Isto foi posteriormente confirmado, através da introdução de mensagens de rastreamento da execução das operações de TM no código com esta política, com a reexecução destas simulações. Era possível notar que o sistema acabava entrando em *livelock* em várias simulações, com várias *threads* invalidando umas às outras em sequência, de forma a, aparentemente, não saírem mais desta situação. Um exemplo de um trecho da saída destas mensagens de rastreamento introduzidas no código pode ser visto no código 5.1. A indentação mostra de qual núcleo a mensagem vem: nenhuma indentação, núcleo 0; uma indentação, núcleo 1.

Código 5.1: *Livelock* com a política `DVFS_COMMIT_SLOWEST`.

```
Iniciando commit - diminuindo a frequência do core para slowest
    Iniciando commit - diminuindo a frequência do core para slowest
Abort - voltando para full speed
    Abort - voltando para full speed
Iniciando commit - diminuindo a frequência do core para slowest
    Iniciando commit - diminuindo a frequência do core para slowest
Abort - voltando para full speed
    Abort - voltando para full speed
```

De qualquer forma, como o uso da frequência `SLOWEST` (1,56 MHz) é muito baixo para padrões atuais, retirar esta política do conjunto de simulações executado não interferiu na significância dos resultados.

5.2.2 `DVFS_AGGRESSIVE` e `DVFS_ABORTER_ENABLED`

Também com as políticas que diminuem a frequência do núcleo executando uma transação que aborta muito, `DVFS_AGGRESSIVE` e `DVFS_ABORTER_ENABLED`, através da análise dos dados de execução, foi possível verificar que, em muitos casos, foram geradas situações de *livelock* ou situações em que a simulação era muito longa e o tamanho máximo do arquivo de saída *finegrained* gerado era atingido, comprometendo, assim, a coleta de dados com a infraestrutura disponível.

Por exemplo, algumas simulações com a política `DVFS_ABORTER_ENABLED` continuavam em execução após 5 dias. Durante a execução de testes isolados desta política foi possível notar que algumas transações chegavam a abortar milhares de vezes em sequência sem conseguirem concluir seu trabalho. Isto levava a geração de arquivos de saída *finegrained* maiores que o suportado pelo ambiente MPARM.

5.3 Problemas com aplicações do STAMP na plataforma MPARM

A aplicação `kmeans-low` do STAMP, apesar de ser executada com sucesso no MPARM e gerar saída correta, teve problemas na coleta de dados estatísticos sobre sua execução pela plataforma MPARM, gerando arquivos de saída *finegrained* com problemas. Especificamente, para esta aplicação, o MPARM não registrou nenhum dado sobre consumo energético.

Devido a isto, apesar de os dados sobre tempo de execução das operações de TM terem sido coletados, decidiu-se não analisar estes resultados isoladamente.

Capítulo 6

Análise de políticas

Das 1885 simulações descritas na seção 4.3, pode-se descartar da análise final aquelas que não foram eficazes conforme descrito nas seções 5.2 e 5.3. Desta forma, tem-se:

- 12 *benchmarks* (8 aplicações do STAMP, algumas com 2 cargas de trabalho, descartando `kmeans-low`):
  - `genome`
  - `genome+`
  - `intruder`
  - `intruder+`
  - `yada`
  - `labyrinth`
  - `labyrinth+`
  - `ssca2`
  - `vacation-low`
  - `vacation-high`
  - `bayes`
  - `kmeans-high`
- Algumas políticas de TM do capítulo 3:
  - 2 tipos de controle de concorrência: *Eager* e *Lazy*.
  - 2 variações de *backoff*: linear e exponencial.

  - 6 variações com DVFS (retiradas `DVFS_COMMIT_SLOWEST`, `DVFS_AGGRESSIVE` e `DVFS_ABORTER_ENABLED`).

- 4 variações de números de núcleos computacionais no sistema: 1, 2, 4 e 8.

Uma representação gráfica de todas estas combinações pode ser vista na figura 6.1. As políticas e os *benchmarks* excluídos por não serem viáveis estão riscados.

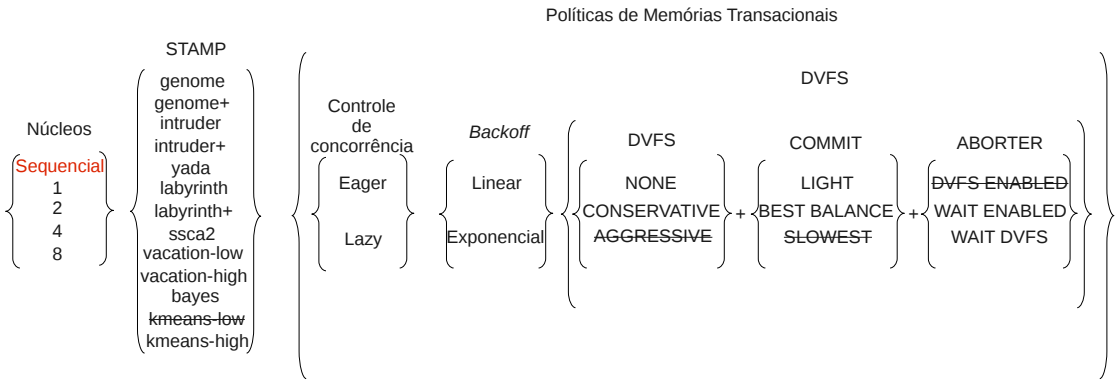


Figura 6.1: Representação de todas as simulações consideradas na análise final neste trabalho, combinando 12 *benchmarks* do STAMP, 2 políticas de TM de controle de concorrência, 2 políticas de variações de *backoff* e 6 variações com DVFS, além da execução em ambientes simulados com quatro diferentes números de núcleos computacionais. Somando-se a todas estas variações, ainda foram executadas versões sequenciais de cada um dos *benchmarks* considerados do STAMP.

Além das políticas consideradas inviáveis, também foram excluídas da análise as combinações de políticas que não geraram dados confiáveis ou cuja simulação não executou com sucesso após inúmeras tentativas, conforme explicado na subseção 5.1.10. Estas combinações estão listadas na tabela 6.1.

Todas as variações válidas, agregadas, formam um espaço de 1155 simulações com políticas que executaram com sucesso para serem analisadas em maiores detalhes.

Tabela 6.1: Simulações que encontraram problemas durante sua execução.

Benchmark	**Número de núcleos simulados**	**Política de TM**
yada	8	BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager
yada	8	BACKOFF_LINEAR Eager
yada	2	BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager
yada	8	BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager
yada	8	BACKOFF_LINEAR DVFS_CONSERVATIVE Eager
intruder	4	BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager
intruder	4	BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager
intruder+	8	BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Lazy
intruder+	8	BACKOFF_LINEAR DVFS_COMMIT_LIGHT Lazy

6.1 Energia, tempo de execução e *Energy-Delay Product*

Como já dito anteriormente, o foco deste trabalho está na análise de desempenho e consumo energético de TM, comparando-se como diferentes políticas de gerenciamento do sistema de TM afetam estas variáveis durante a execução de um conjunto de aplicativos.

Os dados de consumo energético e tempo de execução dos *benchmarks*, na plataforma de simulação utilizada neste trabalho, estão registrados nos arquivos *finegrained*, como explicado no capítulo 5. Com estes dados é possível gerar gráficos de consumo energético e *speedup* para cada execução com um determinado número de núcleos no ambiente simulado para cada política de cada um dos programas do STAMP.

Estes gráficos conseguem mostrar quais políticas são melhores em relação ao consumo energético e quais são melhores em relação ao tempo de execução. Apesar de estas grandezas estarem de certa forma associadas — já que menor tempo de execução, intuitivamente, vai implicar em menor consumo energético por parte do processador — será possível ver, através dos gráficos, que esta relação nem sempre é tão direta e podem haver variações.

Em uma situação ideal, considerando um cenário de consumo de energia constante (sem DVFS), ao aumentar o número de núcleos executando em uma aplicação com para-

lelização perfeita, haveria um *speedup* linear e a energia total gasta se manteria constante, já que o tempo de execução seria dividido pelo número de núcleos executando e cada núcleo gastaria uma quantidade de energia proporcional ao seu tempo de execução. Na prática, é muito difícil ter um *speedup* linear e vários fatores podem influenciar o consumo de energia. O tempo de execução pode sofrer grandes variações que vão depender do algoritmo sendo paralelizado, da técnica de paralelização, das sobrecargas impostas por esta técnica e da carga de trabalho sendo executada, entre outras coisas.

Este tipo de análise — consumo energético *versus* tempo de execução — é relativamente comum. Como estas duas grandezas, apesar de estarem relacionadas, não são diretamente ligadas uma à outra, uma nova medida, o *Energy-Delay Product* (EDP) [42], foi proposta para condensar as informações tanto de consumo energético quanto de tempo de execução, de forma que fique fácil mostrar qual sistema tem o melhor balanço entre estas duas grandezas.

$$EDP = E * T^w \quad (6.1)$$

A equação 6.1 mostra como calcular o EDP. Nela, E é a energia consumida, T é o tempo de execução e w é o peso relativo, geralmente de 1 a 3, que se deseja dar ao desempenho no cálculo do EDP. Quanto maior w, maior a importância do desempenho em relação ao consumo energético no cálculo do EDP. Para os propósitos deste trabalho, em que se deseja dar relevância ao consumo energético, será utilizado $w = 1$. Logo, a fórmula do EDP fica como mostrado na equação 6.2.

$$EDP = E * T \quad (6.2)$$

Como o objetivo de qualquer otimização é diminuir o consumo de energia e diminuir o tempo de execução, pode-se concluir que quanto menor o EDP, melhor é a otimização.

6.2 Normalização dos dados

Para que fosse possível uma comparação entre as diversas políticas de TM, foi necessário normalizar os dados obtidos das simulações em relação a referenciais estabelecidos para se proceder com a comparação.

Tradicionalmente, os dados sobre desempenho — medido através do *speedup* S — e consumo energético — medido através da energia consumida E — de sistemas de TM executando uma determinada política tem sido normalizados em relação à execução desta política com apenas um núcleo — indicado pelo índice $TM1c$. Com isto, todas as execuções com um núcleo têm valor um para suas medidas e as medidas das demais execuções

são mostradas de forma relativa a esta execução com um núcleo, conforme mostrado nas equações 6.3 e 6.4.

$$E_{normalizada\_TM1c} = \frac{E}{E_{TM1c}} \qquad (6.3)$$

$$S_{normalizado\_TM1c} = \frac{T_{TM1c}}{T} \qquad (6.4)$$

Para a medida da energia gasta na execução de uma simulação normalizada (equação 6.3), quanto menor que um este valor for, menos energia foi consumida na simulação em relação à execução com um núcleo. É natural esperarmos, portanto, que quanto mais núcleos utilizarmos na simulação, maior será a energia normalizada em relação à execução com apenas um núcleo, que será maior que um nestes casos.

Já para a medida de desempenho (equação 6.4), a grandeza utilizada neste trabalho foi o *speedup*. Esta já é uma medida comparativa entre duas outras grandezas de tempo — representado por T. No caso descrito aqui, calcula-se o *speedup* em relação à execução de TM com um núcleo. Neste caso, quanto maior a paralelização obtida com TM, espera-se que menor seja o tempo de execução e, consequentemente, maior seja o *speedup*. *Speedups* vantajosos são aqueles maiores que um e são estes que espera-se encontrar nas execuções com vários núcleos.

Este tipo de visualização comparativa com a execução de TM em um núcleo para uma determinada política é muito útil para se verificar como as políticas escalam quando a execução é feita com mais núcleos. Por exemplo, na figura 6.2 tem-se os dados para consumo energético e *speedup* para uma execução do *benchmark* `bayes` em que, para cada política, os resultados estão normalizados em relação à execução com um núcleo daquela política.

Todas as execuções de um núcleo na figura 6.2 tem valor um. Através deste gráfico pode-se facilmente concluir que para todas as políticas analisadas, para o *benchmark* `bayes`, o desempenho e o consumo energético para as execuções de 8 núcleos são conflitantes: pegando-se, por exemplo, a política `BACKOFF_EXPONENTIAL Eager`, é possível ver que apesar de o consumo energético aumentar de quatro para oito núcleos, o desempenho cai — gasta-se mais energia com mais núcleos, o que é intuitivo, mas obtém-se menos desempenho de processamento, o que é contra-intuitivo. De forma geral, é possível identificar este mesmo comportamento em todas ou na maior parte das políticas com todos os *benchmarks* que possuem alta contenção: `bayes`, `intruder`, `intruder+`, `labyrinth`, `labyrinth+` e `yada`.

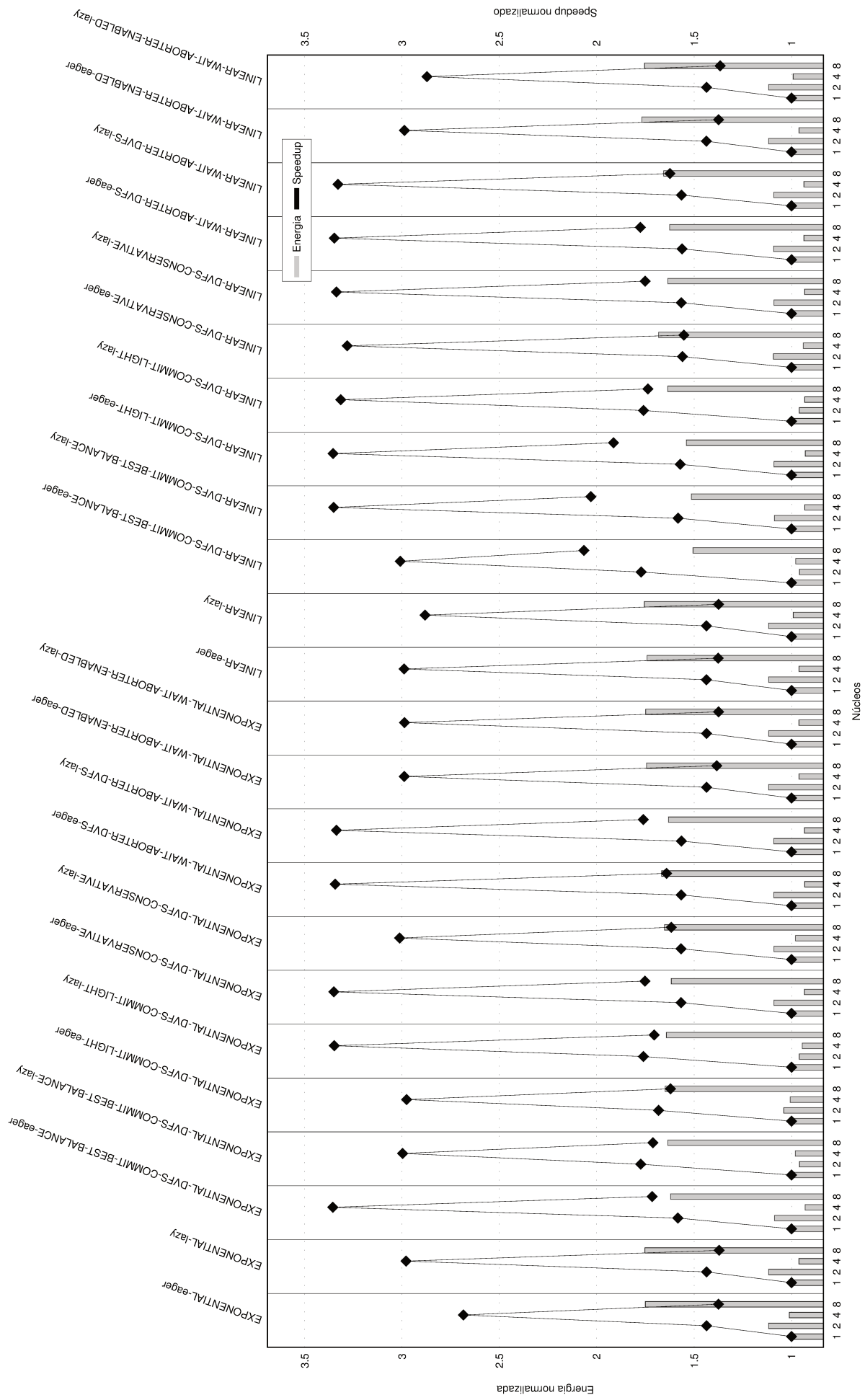


Figura 6.2: Energia e *speedup* normalizados em relação à execução com TM em um núcleo — bayes.

Uma possível explicação para a questão mostrada no parágrafo anterior pode estar em uma limitação da plataforma de simulação utilizada. O barramento de interconexão AMBA-AHB tem certas limitações e pode entrar em contenção caso muitos acessos estejam sendo feitos ao barramento simultaneamente, o que certamente acontece com aplicações cujo algoritmo já apresenta esta características de alta contenção e, portanto, precisam trocar muitas informações entre as várias *threads* executando, especialmente quando o número de núcleos aumenta.

Este tipo de normalização de dados pode ser bastante útil e gerar conclusões como a exposta anteriormente. Ele também é utilizado tradicionalmente para fazer comparações entre diferentes políticas de TM, mesmo que a escala para cada política seja diferente. Isto porque a execução de um aplicativo com TM em um único núcleo não gera conflitos e todas as transações são finalizadas com sucesso, o que também acontece em execuções sequenciais. Logo, descartar os dados da execução sequencial, que seria um denominador comum para um determinado *benchmark*, e assumir que todas as comparações serão feitas em relação a execuções utilizando TM em um núcleo não parece ser um problema. Pelo mesmo motivo, não parece ser um problema o fato de as escalas entre as políticas não serem as mesmas, já que a falta de ocorrência de conflitos faz com que a sequência de execução seja a mesma com qualquer política de TM, desde que a execução seja em um único núcleo.

No entanto, apesar dos pontos mostrados no parágrafo anterior, acredita-se que este tipo de normalização dos dados não é muito apropriado para a comparação de diversas políticas de TM diferentes devido à impressão de que a falta de um referencial comum possa ser um problema. Isto porque, mesmo não ocorrendo conflito entre transações na execução com TM em um único núcleo, a implementação de cada política de TM adiciona uma sobrecarga na execução do sistema. Esta sobrecarga é diferente para cada política de TM implementada e, acredita-se, sua influência é grande o suficiente para levá-la em consideração durante as comparações entre as diversas políticas, já que o código do sistema de TM é executado com grande frequência nos *benchmarks*. Se esta hipótese for verdadeira, comparar diversas políticas normalizadas em relação às suas execuções em um núcleo pode levar a erros de comparação.

A fim de avaliar esta questão, comparou-se os dados de consumo energético e desempenho gerados pelas simulações normalizados tanto em relação à execução com TM em um único núcleo quanto em relação à execução sequencial. A fim de facilitar esta comparação, utilizou-se a medida de EDP, que agrega as informações tanto de consumo energético quanto de desempenho. Na figura 6.3, pode-se ver a variação do EDP para o *benchmark* `bayes` em cada uma das políticas exploradas neste trabalho com valores normalizados em relação à execução sequencial e normalizados em relação à execução de TM em um núcleo para cada uma das políticas. O valor do EDP normalizado pode ser calculado

conforme mostrado na equação 6.5, onde EDP_{base} é o EDP referencial em relação ao qual a normalização está sendo calculada (execução sequencial ou execução de TM em um núcleo).

$$EDP_{normalizado} = \frac{EDP}{EDP_{base}} \qquad (6.5)$$

Conclui-se da equação 6.5 que quanto menor o EDP normalizado, melhor será a política de TM em relação ao balanço entre desempenho e consumo de energia.

O valor do EDP normalizado pode ser derivado a partir das medidas de consumo de energia e tempo de execução das simulações (medidas básicas coletadas neste experimento) com a ajuda das equações 6.5, 6.2, 6.3 e 6.4, conforme pode ser visto na equação 6.6.

$$\begin{aligned} EDP_{normalizado} &= \frac{EDP}{EDP_{base}} \\ EDP_{normalizado} &= \frac{E * T}{E_{base} * T_{base}} \\ EDP_{normalizado} &= \frac{E_{normalizada}}{S_{normalizado}} \end{aligned} \qquad (6.6)$$

Na figura 6.3 é possível notar que as medidas de EDP normalizadas em relação à execução sequencial são sempre ligeiramente superiores às medidas de EDP normalizadas em relação à execução com TM em um único núcleo. Este comportamento corrobora o que foi dito anteriormente nesta seção: a introdução de TM insere uma sobrecarga na execução. Esta sobrecarga se traduz em maior consumo energético e maior tempo de execução e, consequentemente, em maior EDP. Note que ambas as curvas, normalizadas em relação à execução sequencial e normalizadas em relação à execução de TM em um núcleo, neste caso, possuem contornos similares. Logo, no caso do *benchmark* `bayes` especificamente, a utilização da normalização em relação à execução sequencial não parece apresentar uma grande diferença em relação à análise final que pode ser obtida dos dados coletados.

No entanto, isto não é verdade para qualquer *benchmark*. Por exemplo, na figura 6.4 é possível ver que apesar de as medidas de EDP normalizadas em relação à execução sequencial serem sempre superiores às medidas de EDP normalizadas em relação à execução com TM em um núcleo e apesar de ambas as curvas terem contornos parecidos, os valores normalizados em relação à execução sequencial amplificam de maneira considerável algumas diferenças (como na política `BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Lazy` para 8 núcleos) e, em alguns casos, mostram tendências opostas (como nas políticas `BACKOFF_LINEAR Eager` e `BACKOFF_LINEAR Lazy` para 2 núcleos, que aparecem em um

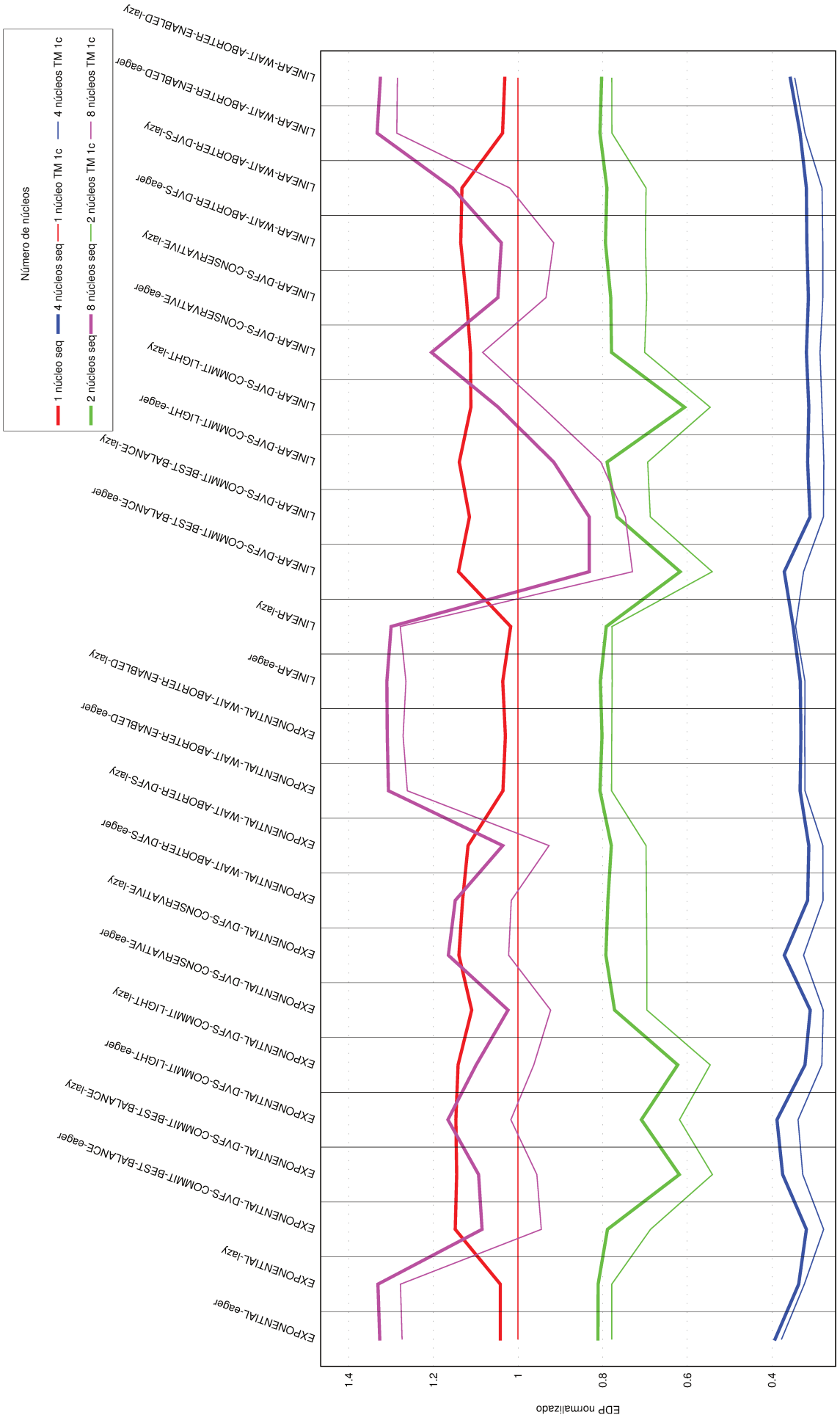


Figura 6.3: EDP normalizado em relação à execução sequencial e em relação à execução de TM em um núcleo — bayes.

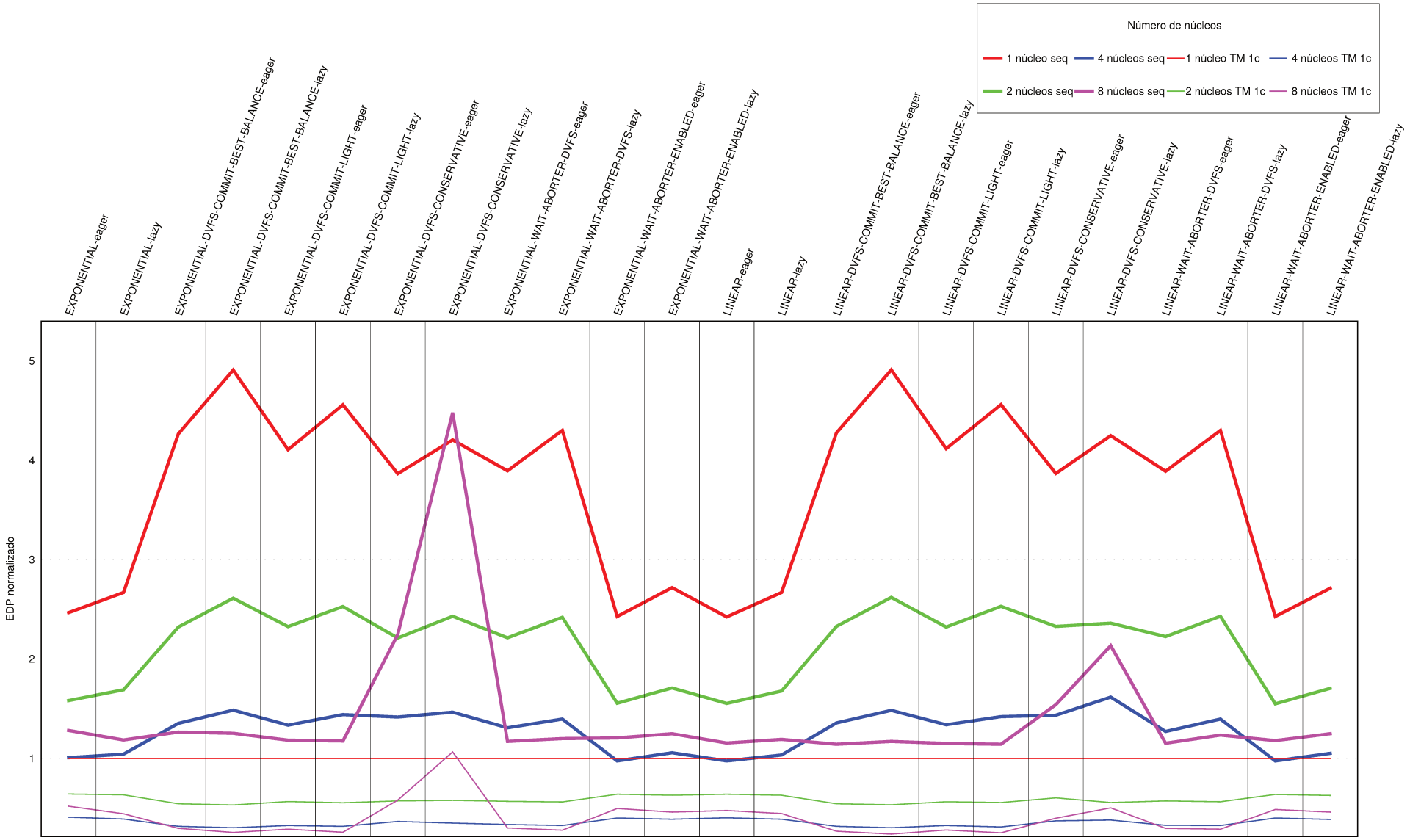


Figura 6.4: EDP normalizado em relação à execução sequencial e em relação à execução de TM em um núcleo — `genome+`.

vale no gráfico normalizado em relação à execução sequencial mas aparecem em um patamar mais elevado no gráfico normalizado em relação à execução de TM em um núcleo).

A amplificação de valores notada na figura 6.4 é ainda mais pronunciada em outros *benchmarks*, como por exemplo o `intruder` (ver figura 6.5). Esta amplificação é vantajosa na hora da comparação de políticas, pois ajuda a discernir de maneira melhor quais políticas são melhores em determinados casos. No caso da figura 6.5, a amplificação de valores aliada a alguns dados que fugiram completamente da escala usual tornam impraticável o uso de uma escala linear para comparação das políticas. Por isto, é usada neste gráfico uma escala logarítmica. Note, no entanto, que alguns valores nas curvas com 4 núcleos não aparecem no gráfico por corresponderem a simulações que terminaram com erro, conforme visto na seção 5.1.10.

Já a diferença de tendências das curvas, também mostrada na figura 6.4, apesar de bem sutil neste caso, aparece de forma bem mais nítida e acentuada em outros, como, por exemplo, no *benchmark* `labyrinth+`, que está representado na figura 6.6.

Por causa destes dois motivos, neste trabalho será utilizada a normalização dos dados em relação à execução sequencial. Este tipo de normalização é capaz de criar uma base referencial comum entre todas as execuções para um determinado *benchmark* e torna mais confiáveis as comparações que são feitas, especialmente quando este tipo de normalização mostra tendências contrárias daquelas que são encontradas nas comparações feitas com normalização dos dados em relação à execução de TM em um núcleo. Neste último caso as conclusões tiradas podem inclusive estar incorretas, já que as tendências apontadas refletem dados comparados contra diferentes referenciais e que, como já visto, não refletem necessariamente a realidade estampada quando todos os dados são refletidos contra o mesmo referencial.

6.3 Análise por *benchmark*

Nesta seção serão analisados os resultados das simulações executadas neste trabalho normalizados em relação à execução sequencial para cada um dos *benchmarks*. O foco será dado em relação aos valores de EDP, já que estes representam um balanço entre as grandezas de desempenho e consumo energético, que são as duas grandezas propostas de serem analisadas neste trabalho.

6.3.1 `bayes`

Para o *benchmark* `bayes`, é possível ver na figura 6.7 o mesmo comportamento visto anteriormente na figura 6.2: para todas as políticas analisadas o desempenho e o consumo energético para as execuções de 8 núcleos são conflitantes, com aumento do consumo

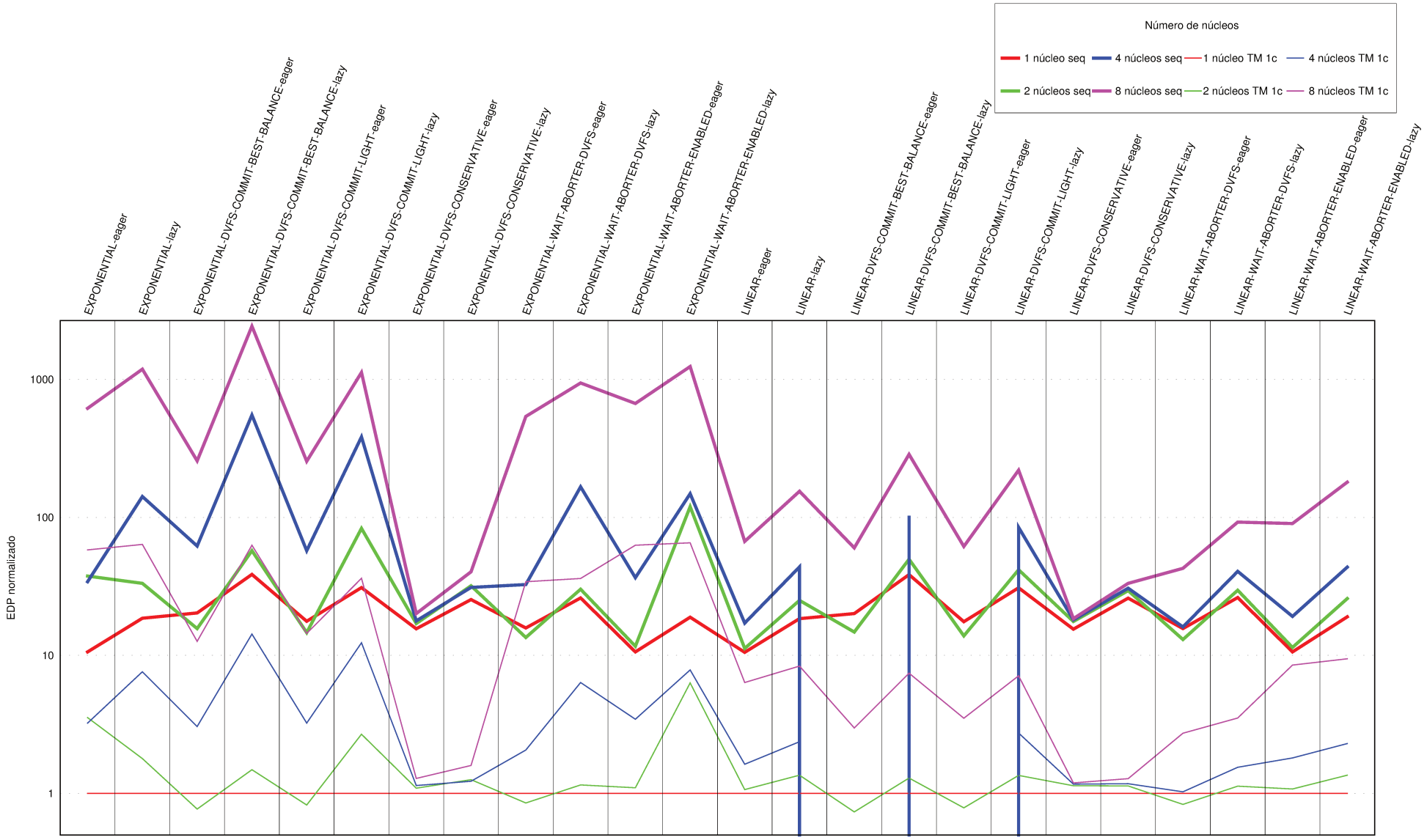


Figura 6.5: EDP normalizado em relação à execução sequencial e em relação à execução de TM em um núcleo — `intruder`. Note escala logarítmica.

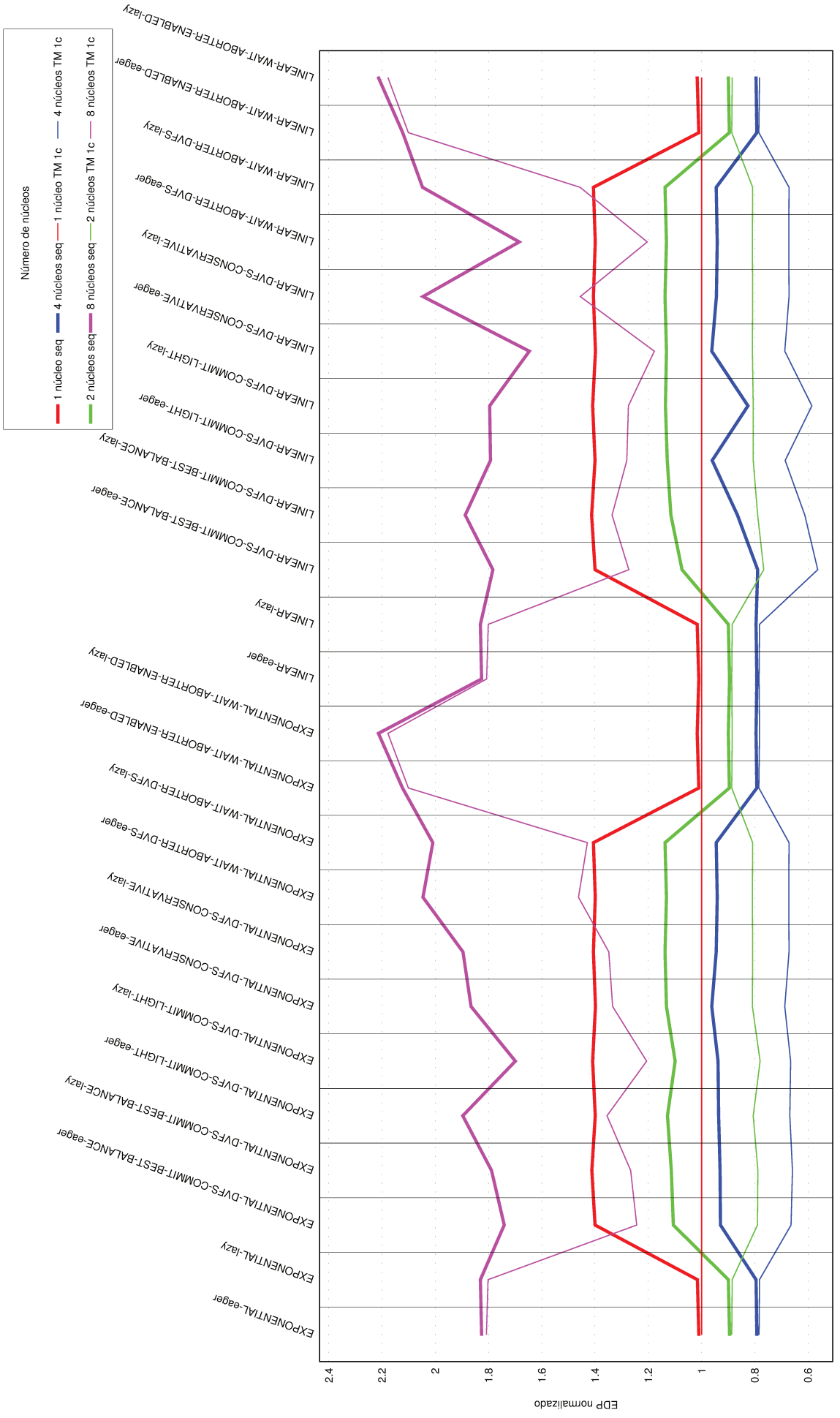


Figura 6.6: EDP normalizado em relação à execução sequencial e em relação à execução de TM em um núcleo — `labyrinth+`.

energético — como é esperado — acompanhado de um menor desempenho de processamento — o que não é esperado quando se aumenta o número de núcleos executando. Esta conclusão, obtida com o gráfico normalizado em relação à execução com TM em um núcleo também pode ser facilmente obtida do gráfico normalizado em relação à execução sequencial.

Um outro fato muito particular pode ser visualizado na figura 6.7. Para vários casos de execuções com 4 núcleos, o consumo de energia tem valor menor que um, indicando que a execução paralela gastou menos energia que a execução sequencial. Em um cenário em que os núcleos consumissem uma quantidade de energia constante — o que acaba sendo intuitivamente assumido muitas vezes — isto seria contraditório. Mas deve-se lembrar que o modelo de consumo de energia é bem mais complexo, não apenas com o uso de DVFS, mas também com inúmeras diferenças que podem acontecer durante uma execução paralela em relação a como a memória e o barramento são utilizados. Tudo isto influencia bastante a forma como a energia é gasta durante a execução e pode gerar situações como esta.

Já quando analisa-se a figura 6.8, é possível também visualizar facilmente que no caso do *benchmark* `bayes`, a execução com 8 núcleos não foi vantajosa para nenhuma política analisada. As execuções que demonstraram melhor balanço entre consumo energético e *speedup* foram as simulações com 4 núcleos, sendo que as simulações com 2 núcleos também foram vantajosas em relação às execuções com 8 núcleos sob este critério.

Na figura 6.7, o *speedup* para 2 núcleos e 8 núcleos são sempre próximos. Em alguns casos, como para a política `BACKOFF_EXPONENTIAL Eager`, o *speedup* em 2 núcleos é maior, já em outros, como para a política `BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Lazy`, o *speedup* em 8 núcleos é maior. No entanto, quando se calcula o balanço entre energia consumida e ganho de desempenho com o EDP(figura 6.8), é nítida a desvantagem das execuções com 8 núcleos devido ao consumo de energia ser bem superior nestes casos.

Ainda em relação à figura 6.8, para execuções em 2 núcleos, a combinação de políticas `BACKOFF_EXPONENTIAL Lazy` com `DVFS_COMMIT_BEST_BALANCE` e `DVFS_COMMIT_LIGHT`, assim como a combinação de `BACKOFF_LINEAR` com `DVFS_COMMIT_BEST_BALANCE Eager` e `DVFS_COMMIT_LIGHT Lazy` mostram bons resultados. A combinação `DVFS_COMMIT_BEST_BALANCE Eager`, que funciona bem com `BACKOFF_LINEAR`, não apresentou resultados bons com `BACKOFF_EXPONENTIAL`, assim como `DVFS_COMMIT_BEST_BALANCE Lazy` funcionou bem com `BACKOFF_EXPONENTIAL` mas não apresentou bons resultados com `BACKOFF_LINEAR`. Isto mostra, para uma política que diminui a frequência de execução durante a operação de *commit*, que há um balanço entre o momento da detecção do conflito e o tamanho do *backoff* que pode ser explorado: detecção precoce de conflitos vai melhor com menores tempos de *backoff*, ao passo que detecção tardia de conflitos se encaixa melhor com maiores tempos de *backoff* (*backoffs*

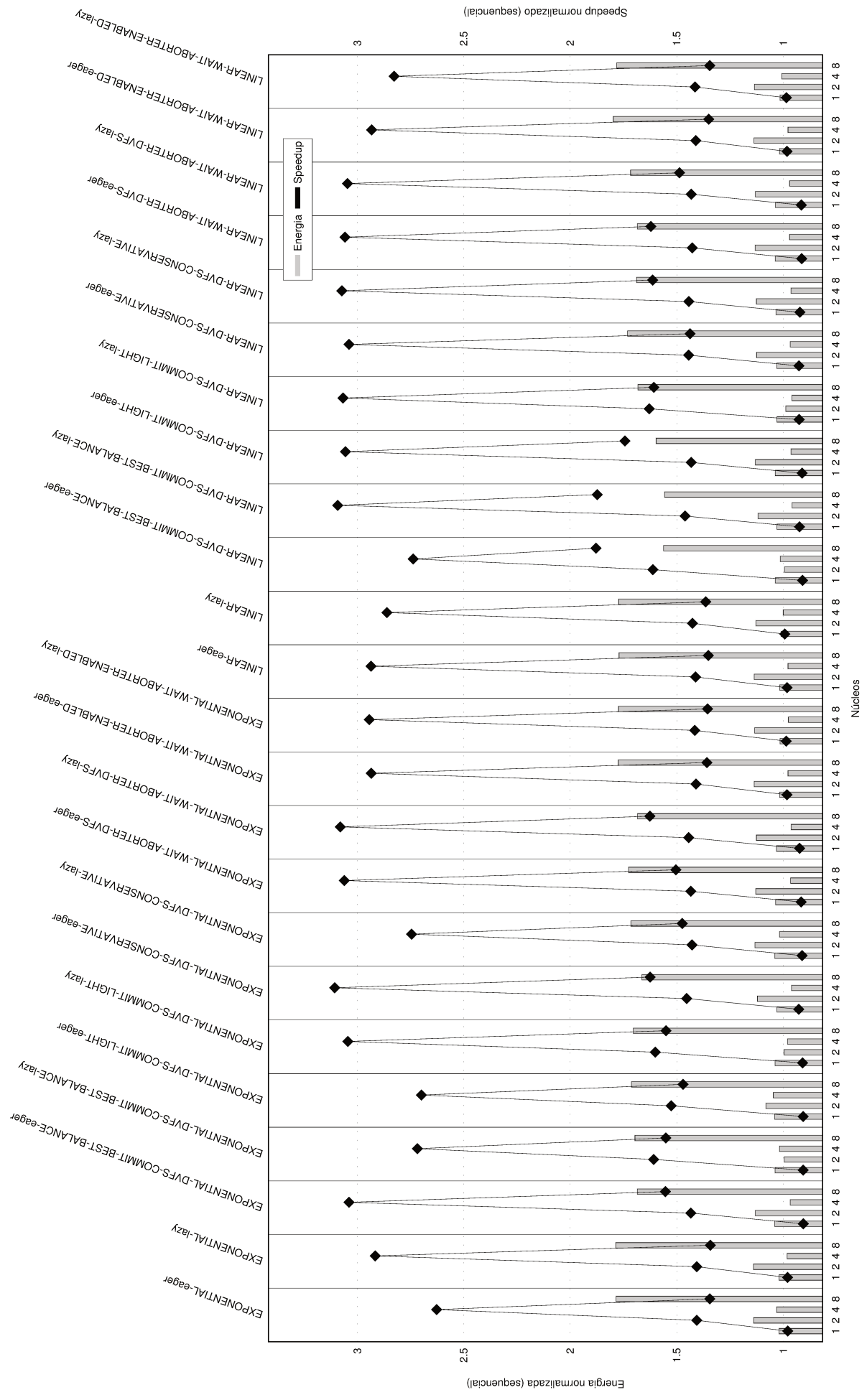


Figura 6.7: Energia e *speedup* normalizados em relação à execução sequencial — bayes.

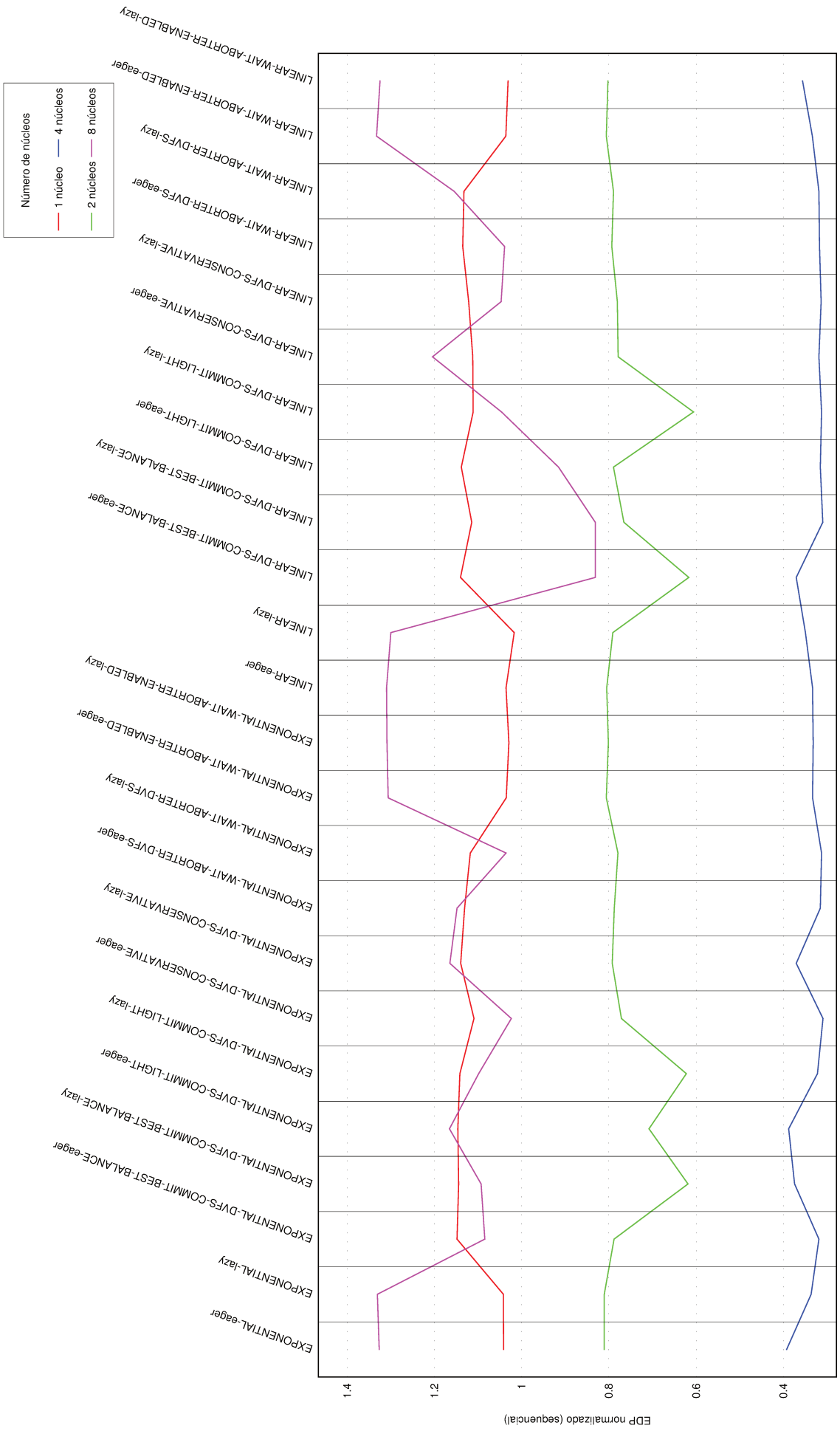


Figura 6.8: EDP normalizado em relação à execução sequencial — bayes.

exponencias tendem a ser maiores que *backoffs* lineares). Porém, quando se aumenta um pouco a frequência de execução da operação de *commit* (`DVFS_COMMIT_LIGHT`), a detecção precoce não é mais vantajosa. Este é um caso típico em que detecção tardia com diminuição moderada da frequência de execução da operação de *commit* mostrou vantagens.

Olhando de uma forma geral na figura 6.8, as políticas com `BACKOFF_LINEAR` se saíram melhores para 4 núcleos, sendo que as políticas `WAIT_ABORTER_DVFS` apresentaram o melhor desempenho executando em 4 núcleos. Ainda, conclui-se que todas as políticas para dois e quatro núcleos possuem EDP menor que um, o que indica que elas são mais eficientes que a execução sequencial.

6.3.2 `genome`

Para o *benchmark* `genome`, o comportamento mostrado na figura 6.9 está mais dentro daquilo que seria normalmente esperado: à medida que se aumenta o número de núcleos executando, aumenta-se o consumo de energia moderadamente, ao mesmo tempo em que diminui-se o tempo de execução. As principais exceções são as políticas com `DVFS_CONSERVATIVE`, que apresentaram um gasto de energia muito maior para a execução com 8 núcleos, sendo que quando combinada com `BACKOFF_EXPONENTIAL` isto ainda veio acompanhado de um severo impacto negativo no tempo de execução.

Quando analisa-se o EDP (figura 6.10) para este *benchmark*, no entanto, a execução com 8 núcleos não é muito vantajosa de uma forma geral e não apenas nos casos das políticas com `DVFS_CONSERVATIVE` como concluído anteriormente nesta seção. Isto porque, apesar de a execução com 8 núcleos entregar melhor desempenho em relação à execução de 4 núcleos na maior parte das vezes, ela consome uma quantidade de energia relativamente maior do que o ganho de desempenho relativo que ela consegue entregar e, portanto, acaba sendo menos eficiente quando leva-se em conta o EDP. Este é um caso em que o cenário que entrega o melhor desempenho (políticas com 8 núcleos com exceção de `DVFS_CONSERVATIVE`) não é o mesmo cenário que entrega o melhor balanço entre consumo de energia e desempenho (políticas com 4 núcleos em quase todos os casos).

Ainda na figura 6.10 nota-se mais três coisas. Primeiramente, nas execuções com um, dois e quatro núcleos, é possível identificar nas curvas um formato de dente-de-serra. Este formato indica que políticas *Eager* são, em geral, melhores que políticas *Lazy*. Isto indica que a detecção precoce de conflitos provê vantagem neste caso, pois, provavelmente, das vezes em que ocorre um conflito, este é detectável rapidamente durante a transação. Este tipo de contorno também aparecerá na análise de outros *benchmarks* e, no caso do `genome`, ele tende a desaparecer com o aumento do número de núcleos executando. Isto indica que, quanto maior o número de núcleos executando, menos importante é o tipo de detecção de

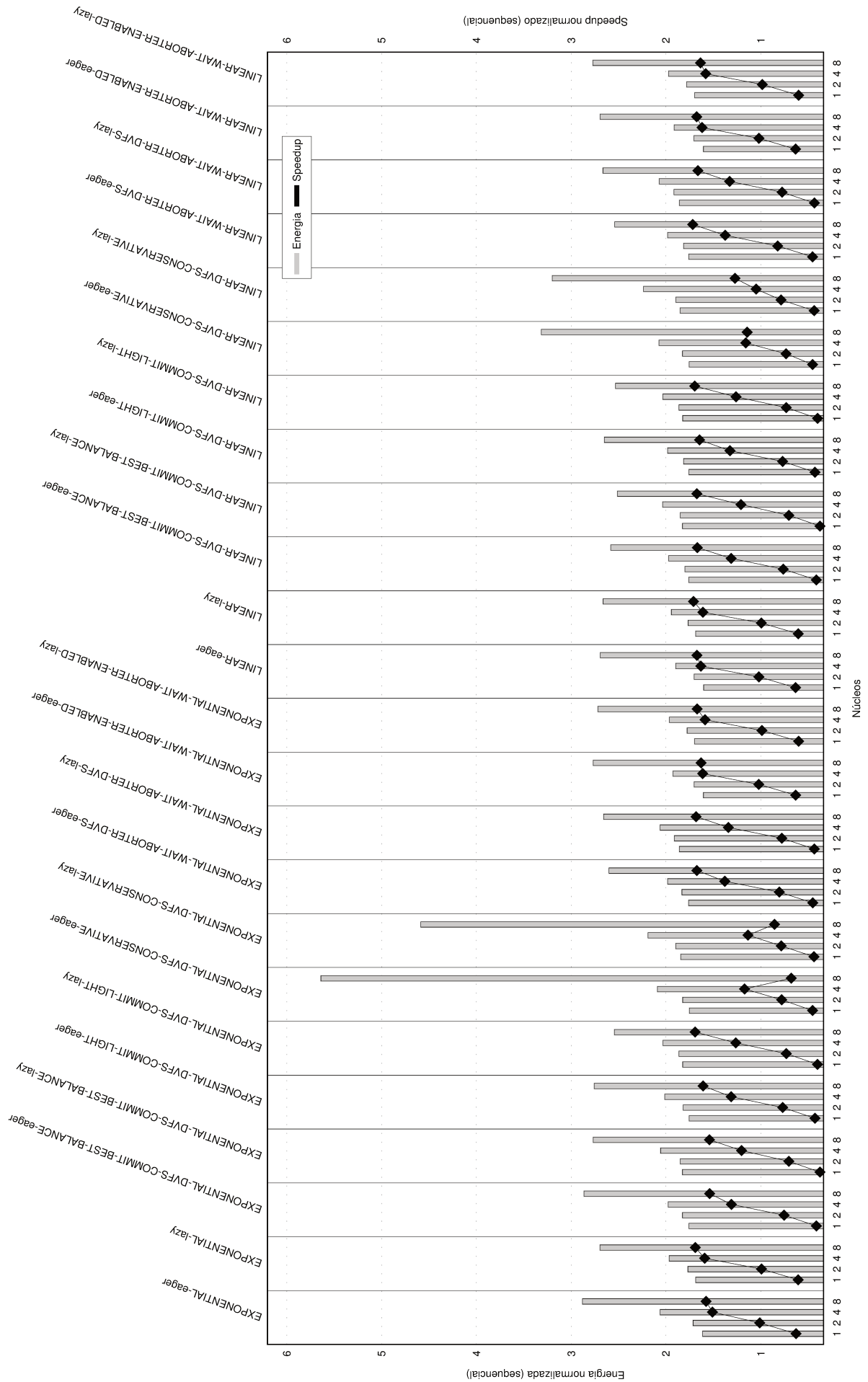


Figura 6.9: Energia e *speedup* normalizados em relação à execução sequencial — `genome`.

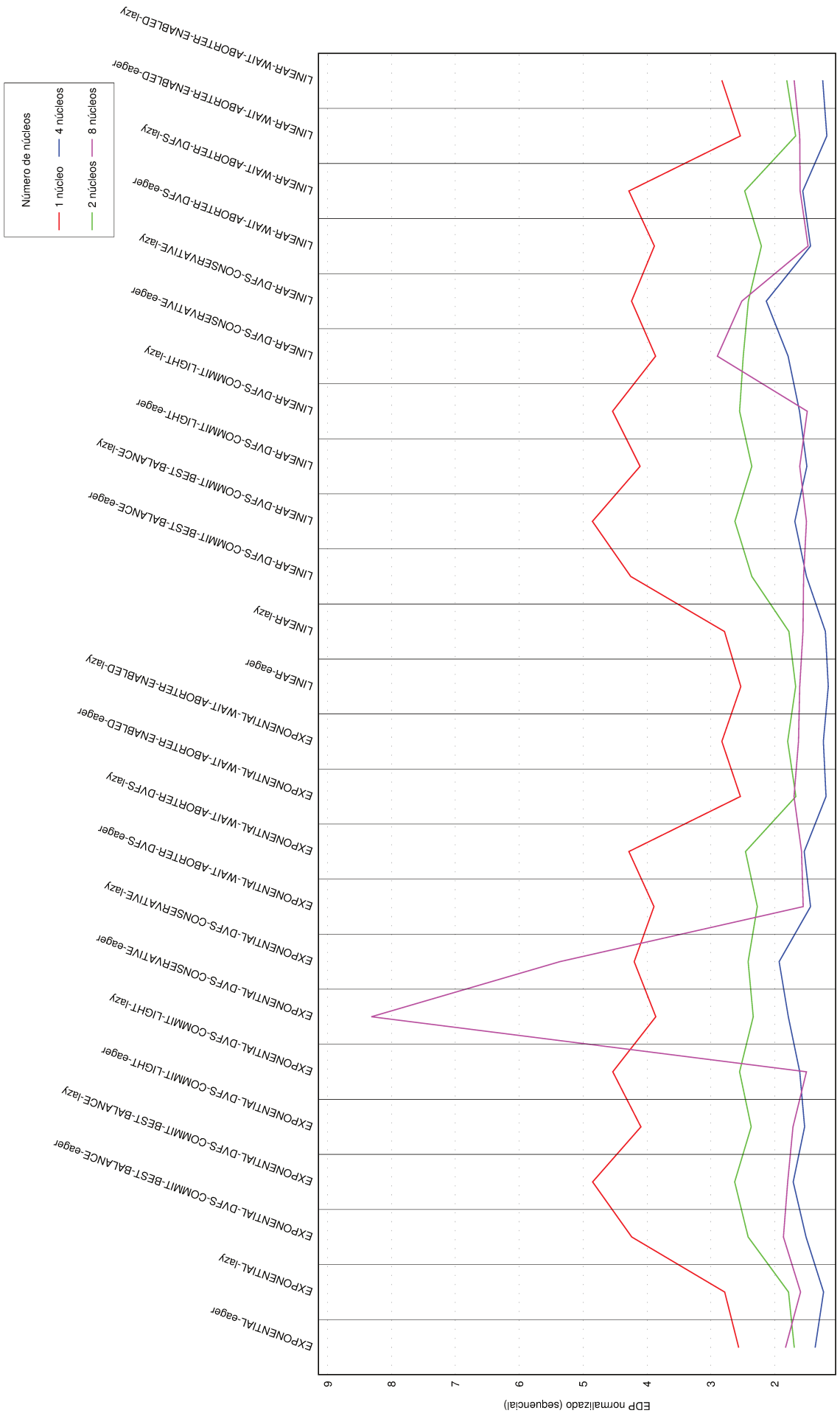


Figura 6.10: EDP normalizado em relação à execução sequencial — `genome`.

conflito utilizado (precoce ou tardio).

Além disto, não há EDP menor que um para nenhuma execução. Logo, caso se leve em conta o balanço entre consumo energético e desempenho de acordo com as premissas utilizadas neste trabalho (peso igual para as duas medidas), a execução sequencial é a mais vantajosa. Já com o uso de TM, as políticas que apresentaram melhor EDP foram as políticas sem uso de DVFS (`BACKOFF_LINEAR` e `BACKOFF_EXPONENTIAL` combinadas com *Eager* e *Lazy*) e as políticas envolvendo `WAIT_ABORTER_ENABLED`, que como foi visto na seção 3.3, também não envolve o uso de DVFS.

6.3.3 `genome+`

Para o *benchmark* `genome+`, a análise da figura 6.11 é muito parecida com aquela feita para a figura equivalente na subseção 6.3.2. A diferença é que para esta carga de trabalho as políticas com `DVFS_CONSERVATIVE` foram menos penalizadas em relação à perda de desempenho nas execuções com 8 núcleos. Isto porque provavelmente as sobrecargas impostas por estas políticas acabaram diluindo-se na carga de trabalho maior processada neste caso. Mas, ainda assim, elas mostram-se extremamente desvantajosas.

A mesma conclusão é verdade para a figura 6.12 e a análise da figura equivalente feita na subseção 6.3.2: ambas mostram comportamentos muito parecidos. Chama a atenção, no entanto, de que com esta carga de trabalho maior no *benchmark* `genome+` (já que o algoritmo utilizado é o mesmo que no caso do `genome`), houve uma melhora do EDP nas execuções com 8 núcleos em relação com o EDP nas execuções com 4 núcleos. No caso do `genome`, as execuções com 4 núcleos eram invariavelmente melhores. Agora, no `genome+`, apesar de a diferença entre os EDP das execuções de quatro e oito núcleos ainda ser pequena, as execuções com 8 núcleos levam vantagem. Isto indica que a sobrecarga imposta pela maior quantidade de comunicação necessária para a execução com 8 núcleos está sendo compensada, com esta maior carga de trabalho, com o maior poder de processamento agregado entregue.

Um outro aspecto para o *benchmark* `genome+` é que as políticas com `WAIT_ABORTER_ENABLED Eager` (com `BACKOFF_LINEAR` ou com `BACKOFF_EXPONENTIAL`) e a política `BACKOFF_LINEAR Eager` tiveram EDP menor que um na execução com 4 núcleos, mostrando-se, portanto, melhores que a execução sequencial.

De uma forma geral, com o uso de TM, as políticas que apresentaram melhor EDP foram as mesmas que as apresentadas para o *benchmark* `genome`.

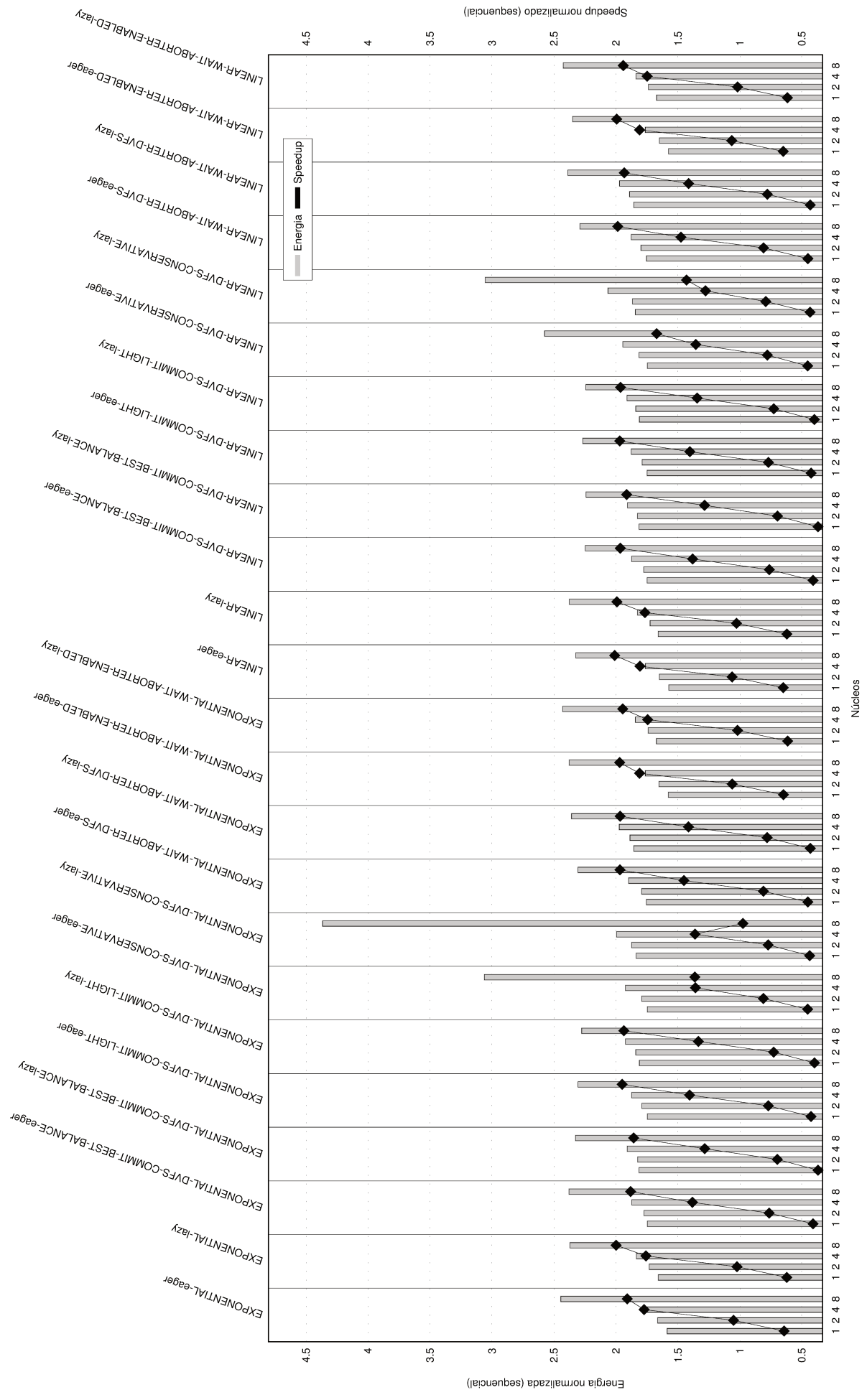


Figura 6.11: Energia e *speedup* normalizados em relação à execução sequencial — `genome+`.

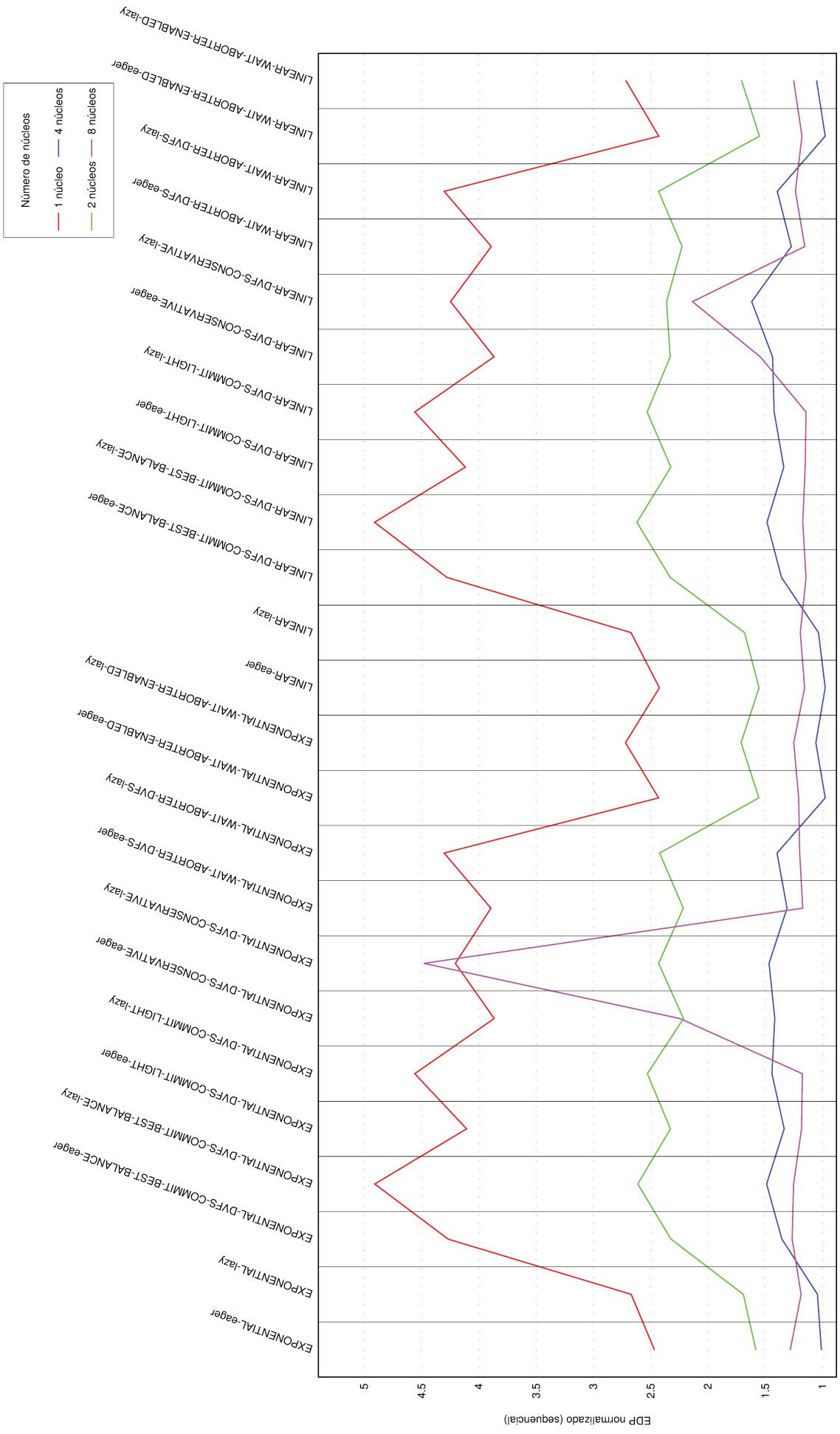


Figura 6.12: EDP normalizado em relação à execução sequencial — `genome+`.

6.3.4 `intruder`

No caso do *benchmark* `intruder`, a análise da figura 6.13 mostra que o uso de TM e o aumento do número de núcleos executando aumenta demasiadamente o consumo de energia sem prover quase nenhum ganho de desempenho. Isto é característico de algoritmos que não escalam e que apresentam alta contenção no acesso aos dados, como é o caso do `intruder`. Para estes tipos de algoritmos, o uso de TM impõe sobrecargas sem trazer ganhos.

Já em relação ao EDP (figura 6.14), de uma forma geral, quanto maior o número de núcleos maior é o seu valor. Além disto, é possível ver que quanto maior o número de núcleos, neste cenário de alta contenção, mais nítido é o contorno dente-de-serra. No caso da figura 6.14, mesmo com o uso de uma escala logarítmica, que tende a amenizar e distorcer vales e picos nas curvas, é possível ver claramente os contornos dente-de-serra — a escala logarítmica foi usada pois a escala linear tornaria impraticável a comparação dos dados devido a alguns valores que fogem da escala. Com 8 núcleos este contorno é nítido e a discrepância entre execuções com políticas *Eager* (mais eficientes) e políticas *Lazy* (menos eficientes) pode chegar a 946%, como com as políticas `BACKOFF_EXPONENTIAL` `DVFS_COMMIT_BEST_BALANCE`. Conclui-se, portanto, que um algoritmo com grande contenção como o `intruder` tem vantagens se for executado de forma sequencial, pois mesmo a combinação com melhor EDP (`BACKOFF_LINEAR Eager`) possui EDP 10,53.

Caso estivesse sendo feita a análise como tradicionalmente se faz, comparando as execuções em relação às execuções com TM em um núcleo para cada política, seria possível ver que, para algumas políticas, as execuções com 4 núcleos podem ser vantajosas. No entanto, quando comparadas com a execução sequencial, todas as políticas, independente do número de núcleos utilizado, não mostram vantagem alguma.

6.3.5 `intruder+`

As figuras 6.15 e 6.16 mostram o comportamento do *benchmark* `intruder+`. Este *benchmark* utiliza o mesmo algoritmo do *benchmark* `intruder`, mas com uma carga de trabalho maior. Aqui, também, é fácil verificar que o uso de TM e o aumento do número de núcleos executando aumenta o consumo de energia sem proporcionar ganhos de desempenho, como já era esperado para um algoritmo com alta contenção no acesso aos dados. Neste caso, porém, é possível ver uma melhora das medidas em relação ao caso do *benchmark* `intruder`, já que a maior carga de trabalho amortiza a sobrecarga imposta pelo sistema de TM e diminui a contenção, visto que a área ocupada pelos dados aumenta. Mesmo assim, todas as execuções tiveram desempenho pior que o caso sequencial e a análise é similar à que foi feita na subseção 6.3.4.

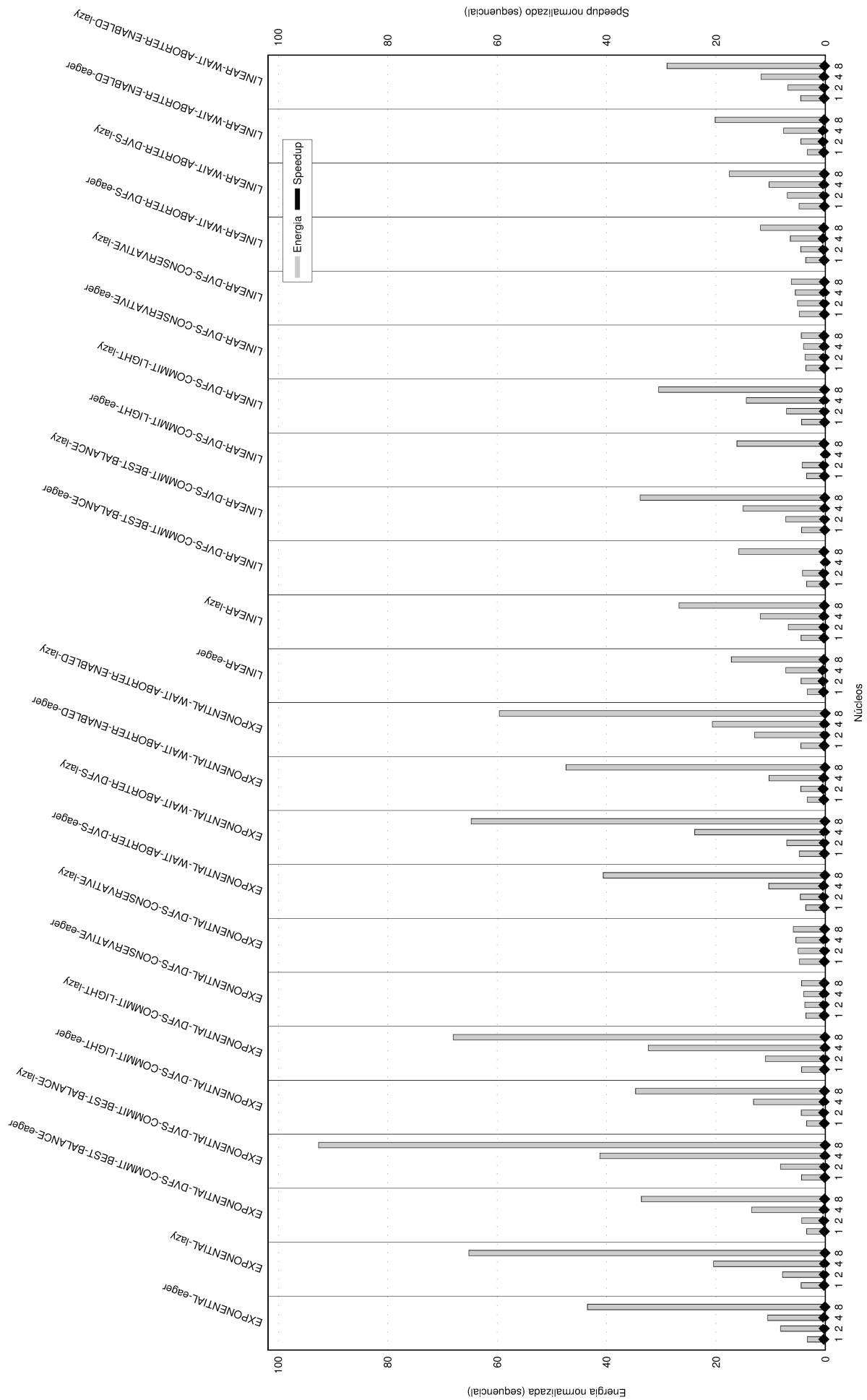


Figura 6.13: Energia e *speedup* normalizados em relação à execução sequencial — `intruder`.

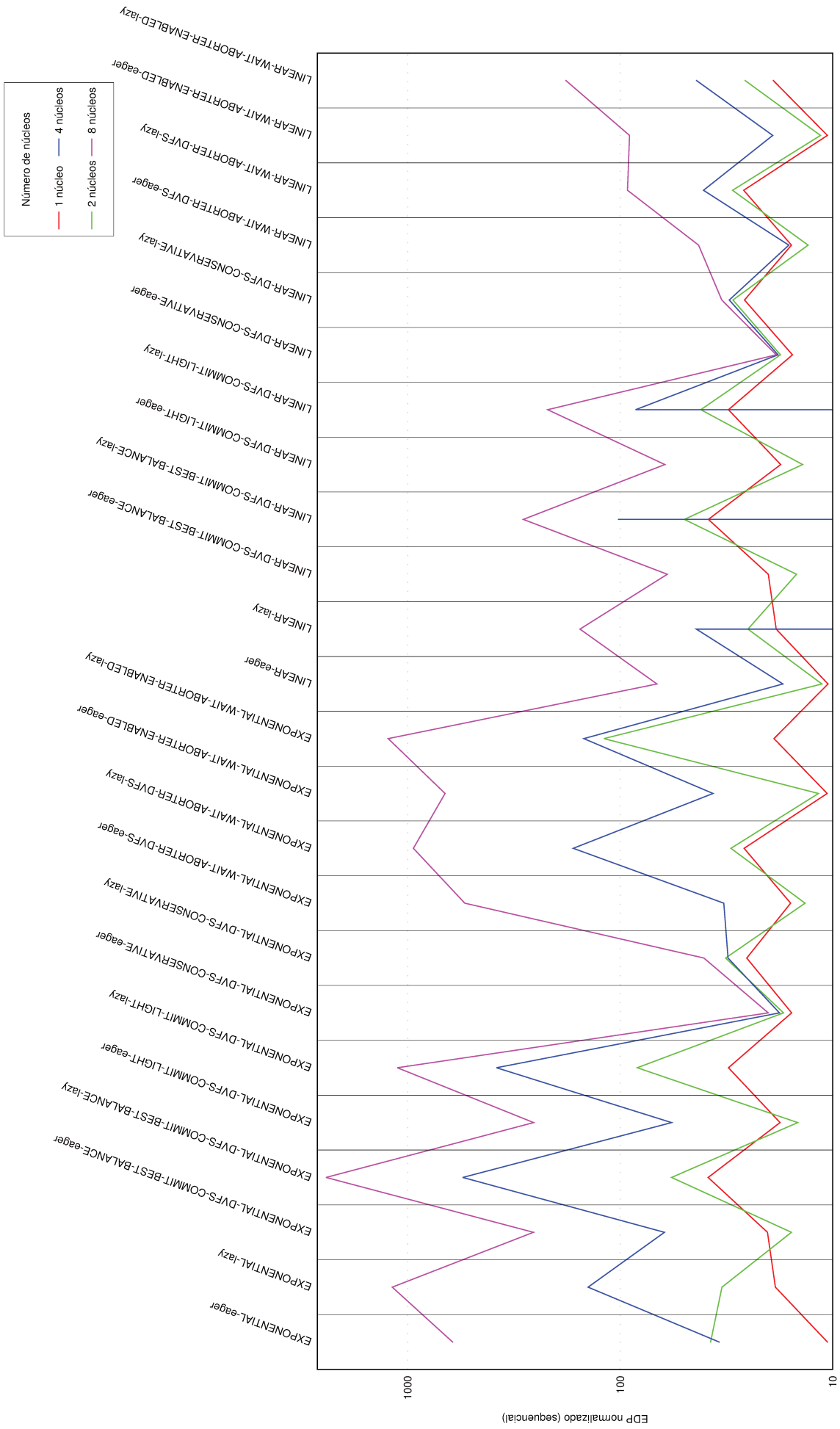


Figura 6.14: EDP normalizado em relação à execução sequencial — `intruder`. Note escala logarítmica.

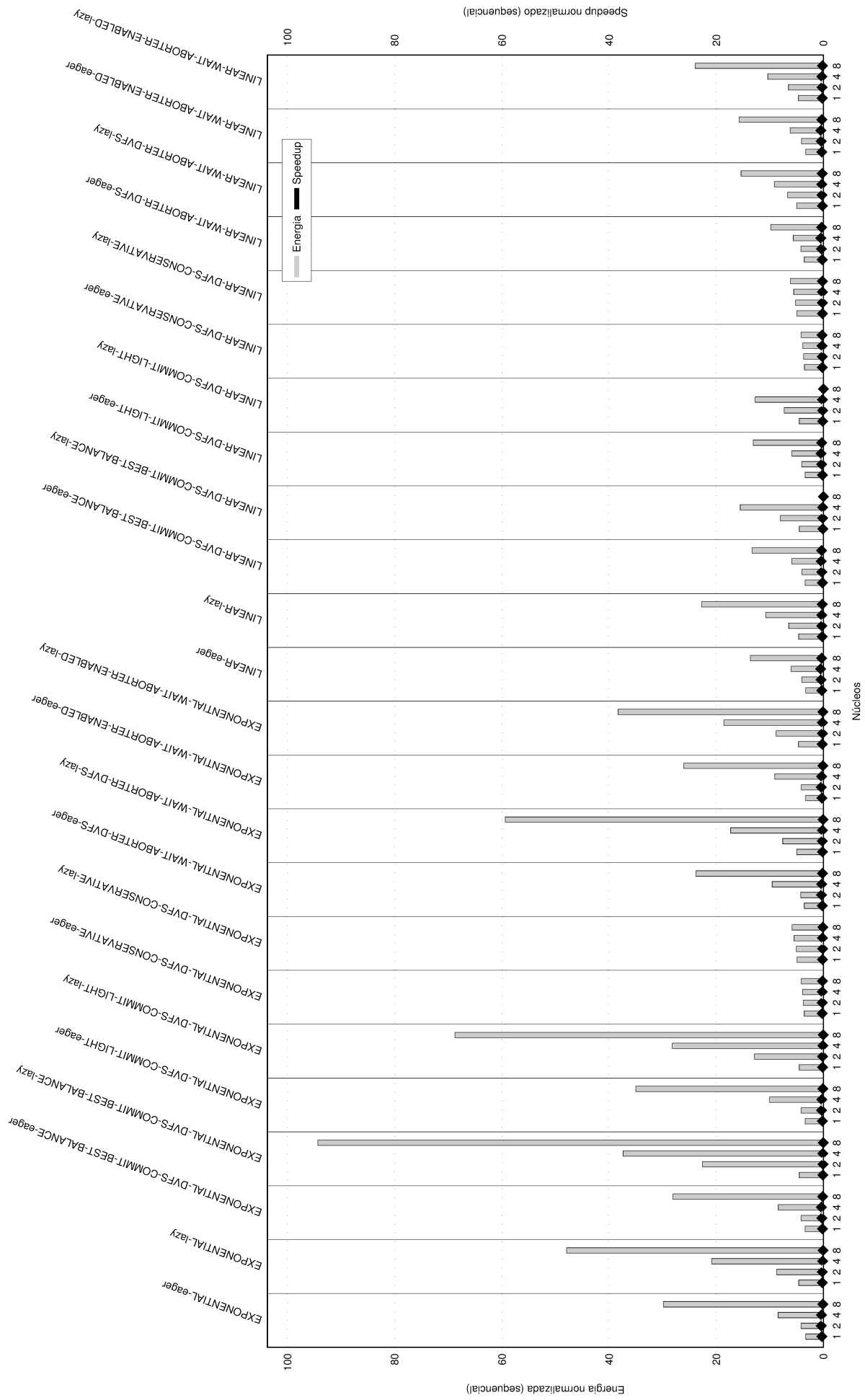


Figura 6.15: Energia e *speedup* normalizados em relação à execução sequencial — `intruder+`.

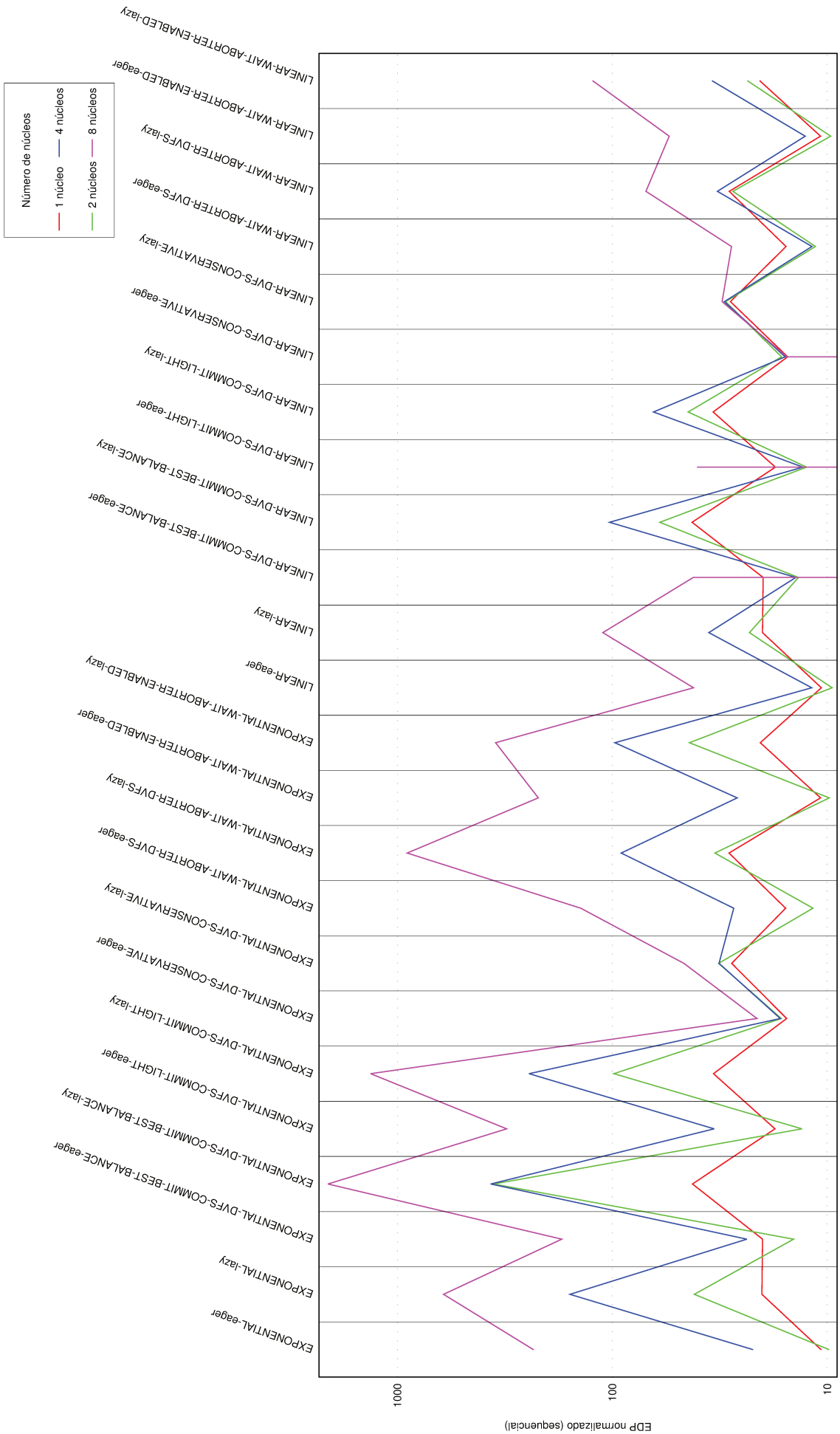


Figura 6.16: EDP normalizado em relação à execução sequencial — `intruder+`. Note escala logarítmica.

6.3.6 `kmeans-high`

Para o *benchmark* `kmeans-high` o comportamento mostrado na figura 6.17 é exatamente o que se esperaria quando se paraleliza a execução de um algoritmo: à medida que se aumenta o número de núcleos executando, aumenta-se o consumo de energia e diminui-se o tempo de execução. Como o `kmeans-high` executa um algoritmo facilmente paralelizável e que escala muito bem, observa-se que há apenas um pequeno aumento no consumo de energia à medida que se aumenta o número de núcleos executando ao mesmo tempo que se obtém um crescimento de *speedup* quase linear até 4 núcleos e ainda vantajoso até 8 núcleos.

Já na análise da figura 6.18, destacam-se várias características. Primeiramente, a execução das políticas de TM estudadas neste trabalho com apenas um núcleo mostra que o EDP foi muito próximo daquele da execução sequencial. Isto significa que a introdução de TM em um algoritmo facilmente paralelizável gera uma sobrecarga muito pequena. Ainda assim, é possível identificar o contorno dente-de-serra, especialmente nas curvas para um e 2 núcleos, indicando que políticas *Eager* são vantajosas em relação a políticas *Lazy*: a identificação precoce de conflitos nas transações é vantajosa, indicando que os conflitos, quando ocorrem, acontecem cedo na transação, dando vantagem a este tipo de política.

Além disto, nas execuções com dois e quatro núcleos é possível ver que, apesar de elas serem sempre vantajosas em relação ao EDP da execução sequencial, para as combinações de políticas sem uso de DVFS (`BACKOFF_LINEAR` e `BACKOFF_EXPONENTIAL` combinadas com *Eager* e *Lazy* e as políticas envolvendo `WAIT_ABORTER_ENABLED`), há uma elevação do EDP, mostrando que estas políticas não se adequaram tão bem quanto as outras para este *benchmark*. Este comportamento é exatamente o contrário do encontrado para o *benchmark* `genome`, onde estas políticas foram as que apresentaram melhor EDP. De qualquer forma, o aumento da quantidade de núcleos executando para oito some com todas estas variações: o algoritmo escala tão bem que as eventuais sobrecargas extras que geravam este desbalanço em algumas políticas simplesmente desaparecem e todos os valores de EDP para execuções com 8 núcleos são bem próximos uns dos outros.

Também é possível ver na figura 6.18 que quanto maior o número de núcleos, menor tende a ser o EDP, indicando que o sistema de TM, neste caso, cumpre bem o seu papel. É fato que, pegando-se a medida de EDP, os valores mais vantajosos são encontrados em execuções com 4 núcleos. No entanto, caso o interesse seja apenas em desempenho, como pode ser visto na figura 6.17, as execuções com 8 núcleos são sempre mais rápidas, proporcionando sempre mais que 4 vezes de *speedup*.

Conclui-se, portanto, que para algoritmos que escalam muito bem, a política utilizada não faz muito diferença se houver uma quantidade suficiente de núcleos e uma carga de trabalho grande o suficiente para ser processada. Neste caso, todas as eventuais sobre-

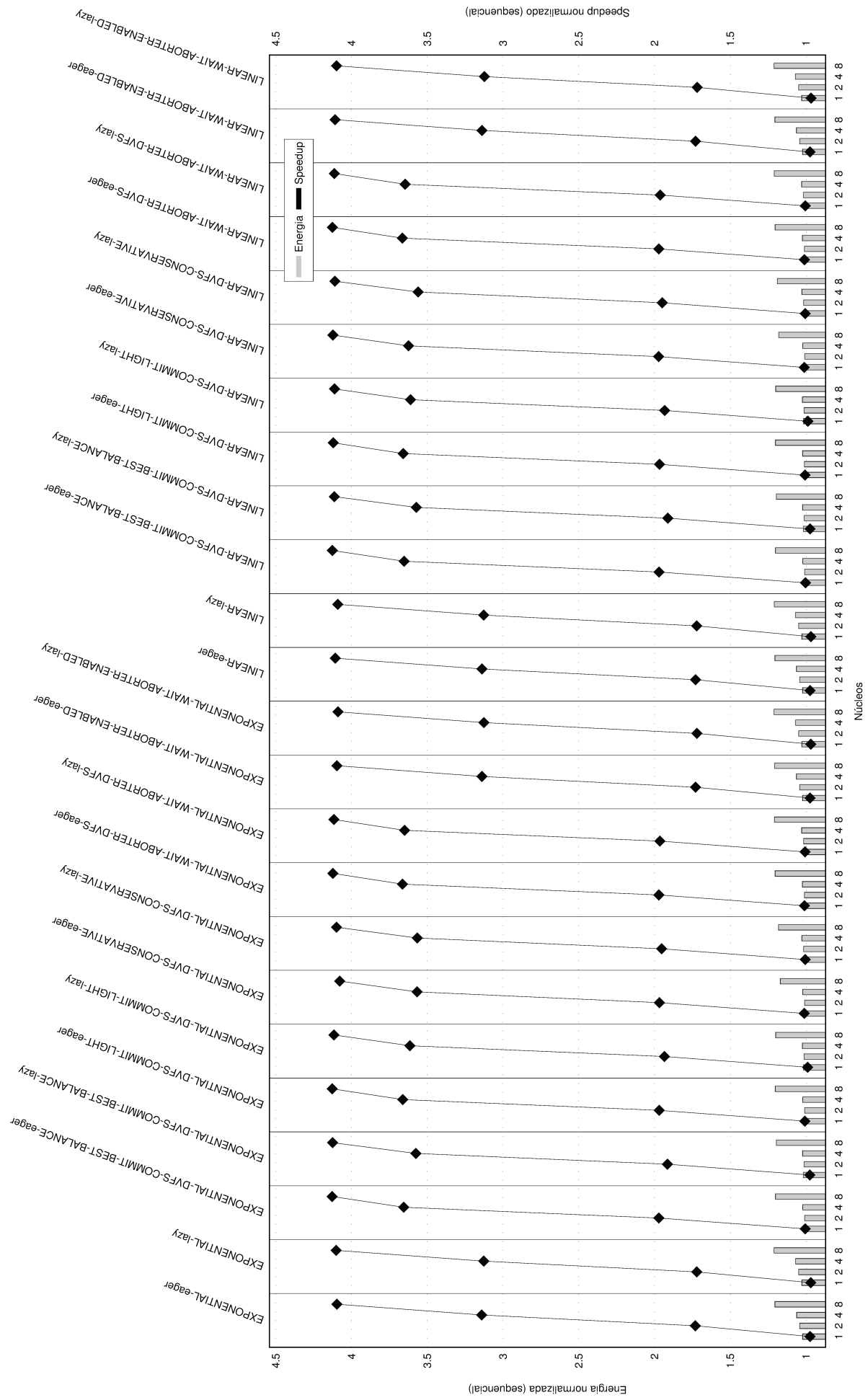


Figura 6.17: Energia e *speedup* normalizados em relação à execução sequencial — `kmeans-high`.

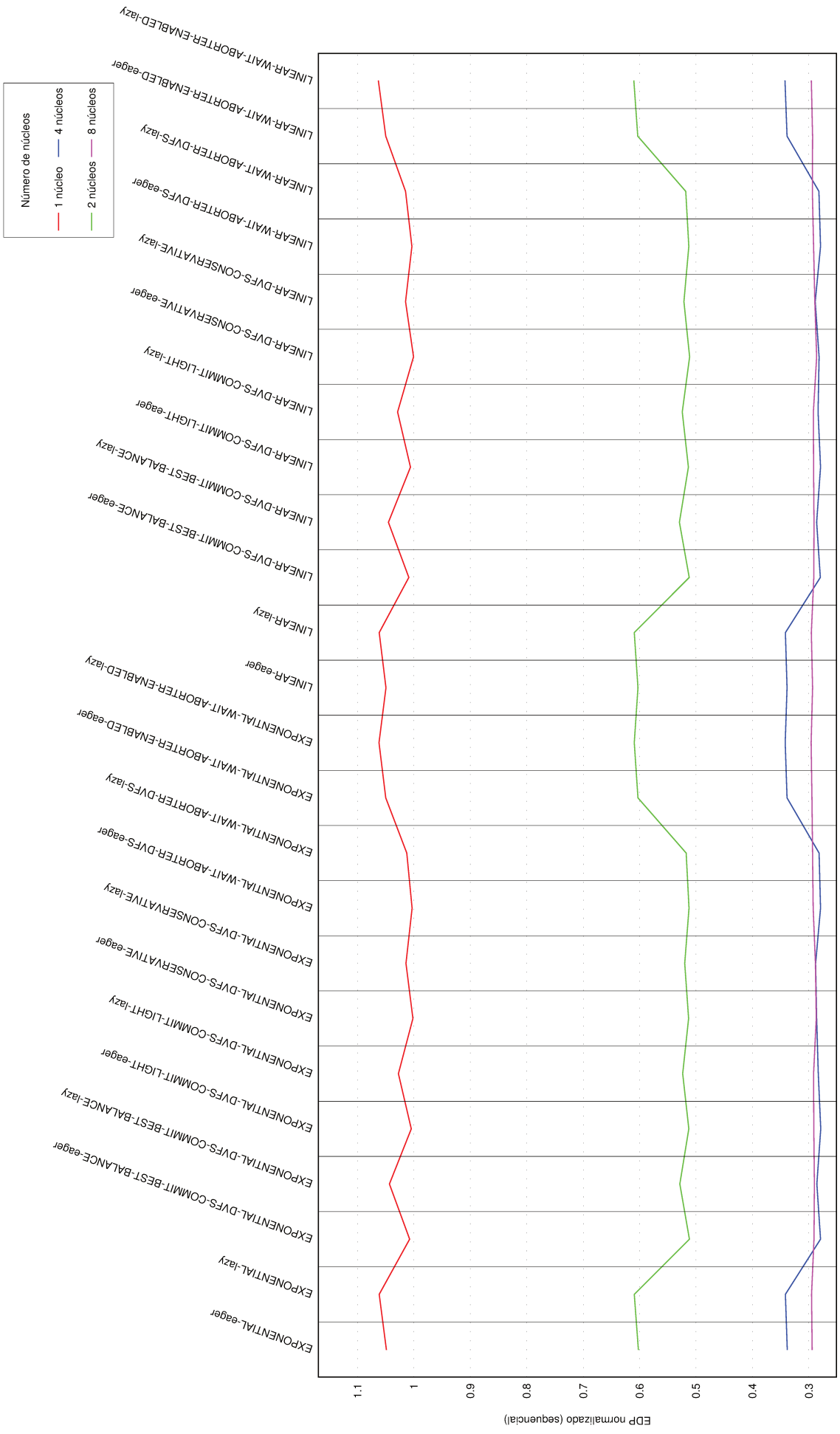


Figura 6.18: EDP normalizado em relação à execução sequencial — `kmeans-high`.

cargas introduzidas pela paralelização desaparecem. É claro que a escalabilidade de um algoritmo tem um limite, mas no caso do `kmeans-high` este limite não foi atingido nos experimentos feitos neste trabalho.

6.3.7 `labyrinth`

O *benchmark* `labyrinth` (figura 6.19), tem um comportamento parecido com o do *benchmark* `bayes` na maior parte dos casos: todas as políticas analisadas, com exceção daquelas derivadas de `DVFS_CONSERVATIVE Lazy` e `BACKOFF_EXPONENTIAL WAIT_ABORTER_DVFS` e, também, das execuções com a política `BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager` assim como `BACKOFF_LINEAR WAIT_ABORTER_DVFS Eager`, apresentaram desempenho e consumo energético para as execuções com 8 núcleos conflitantes, com aumento do consumo energético — como esperado — acompanhado de menor desempenho de processamento — como não é esperado quando se aumenta o número de núcleos executando.

O comportamento descrito no parágrafo anterior reflete diretamente na figura 6.20, onde é possível verificar que a execução com 8 núcleos não foi vantajosa para nenhuma política analisada, já que mesmo nos casos em que havia ganho de desempenho na execução com 8 núcleos pelo gráfico da figura 6.19, este ganho era pequeno comparado com o grande aumento no consumo de energia visualizado nas execuções com 8 núcleos.

Aliás, de uma forma geral, nenhum política de TM foi melhor que a execução sequencial em relação ao EDP neste caso. Com o uso de TM, para mais de um núcleo, as execuções mais vantajosas foram com 2 núcleos, sempre mais vantajosas inclusive em relação às execuções com 4 núcleos. Para 2 núcleos, as execuções mais vantajosas foram as que combinaram as políticas `BACKOFF_LINEAR` e `BACKOFF_EXPONENTIAL` com *Eager* e *Lazy*, além das execuções que incluíram a política `WAIT_ABORTER_ENABLED`, seguidas, de perto, pelas que incluíam a política `DVFS_COMMIT_BEST_BALANCE Lazy`. Normalmente, quando executa-se uma aplicação em vários núcleos, aumenta-se as chances de ocorrência de conflitos, já que há um maior número de *threads* simultâneas trabalhando com os dados. Isto parece particularmente afetar o *benchmark* `labyrinth` e, por isto, as execuções com menor número de núcleos são aquelas mais vantajosas. Além disto, as execuções que combinam as políticas `BACKOFF_LINEAR` e `BACKOFF_EXPONENTIAL` com *Eager* e *Lazy*, além das execuções que incluíram a política `WAIT_ABORTER_ENABLED` são as que inserem menor sobrecarga no código original dos programas e, ao que tudo indica, por isto são vantajosas neste caso, já que tornam a execução o mais próximo daquela que seria no caso sequencial.

6.3.8 `labyrinth+`

Para o *benchmark labyrinth+*, é possível ver na figura 6.21 uma piora no conflito já descrito para o *benchmark* `labyrinth`: agora, para todas as políticas analisadas o desempenho e

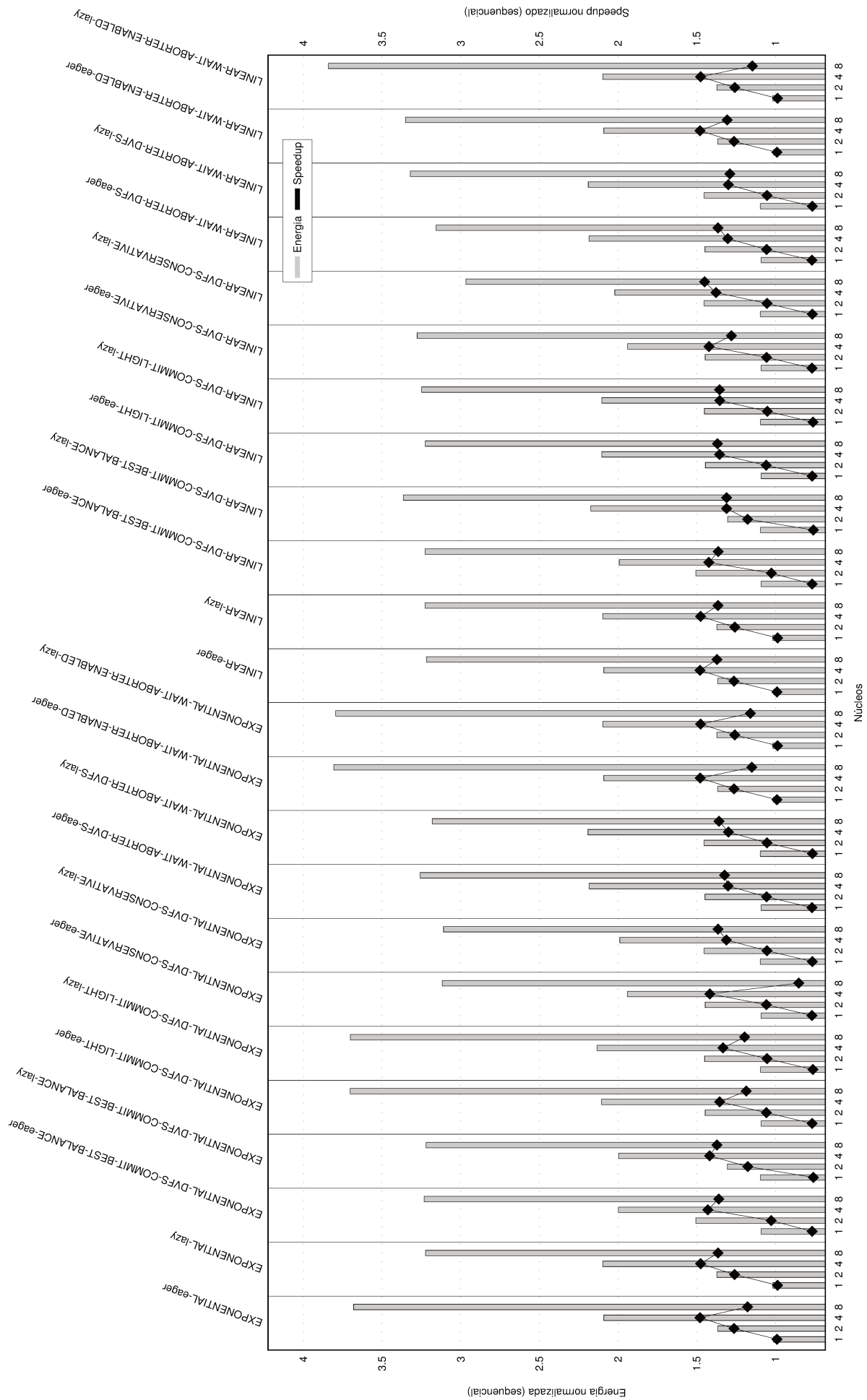


Figura 6.19: Energia e *speedup* normalizados em relação à execução sequencial — `labyrinth`.

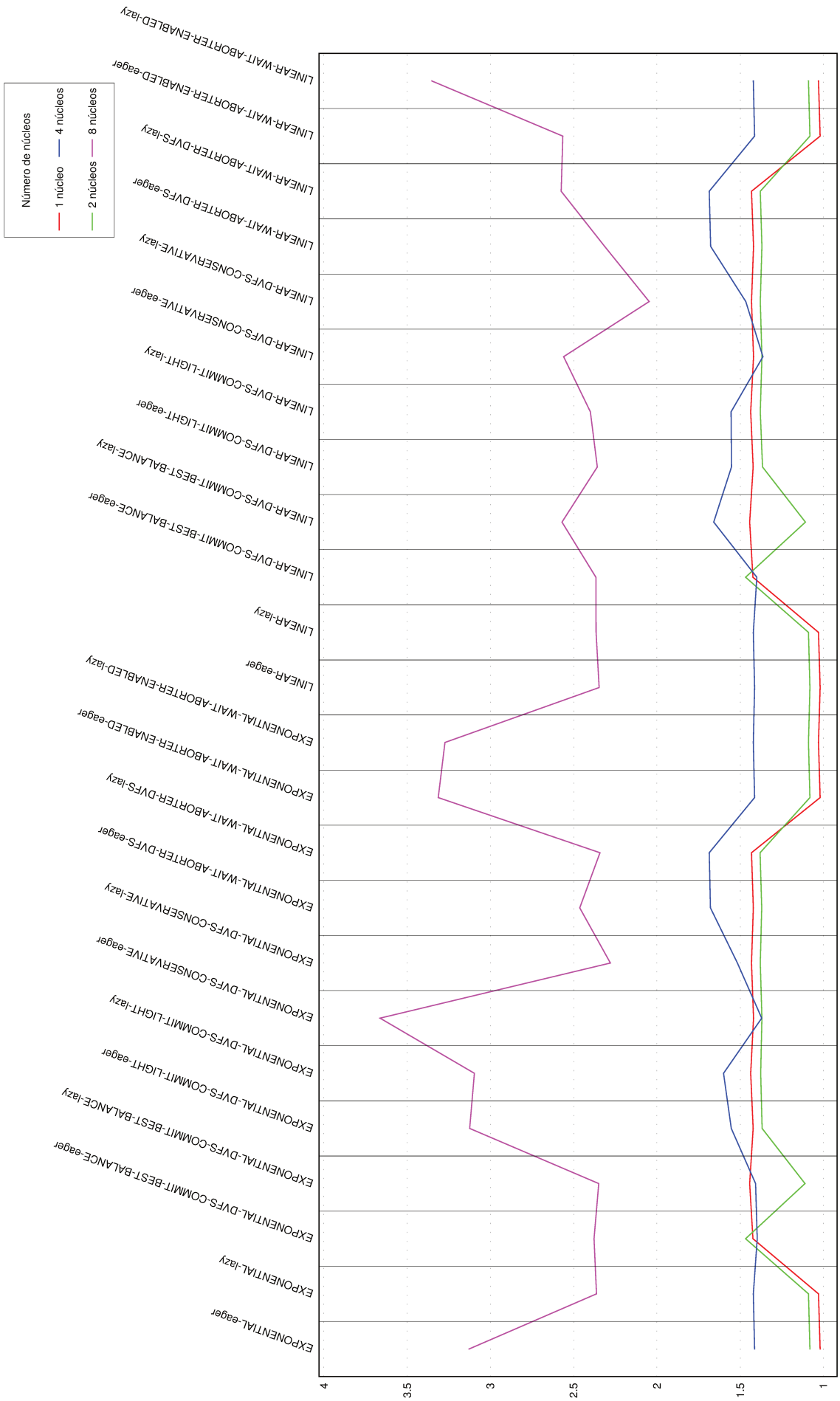


Figura 6.20: EDP normalizado em relação à execução sequencial — `labyrinth`.

o consumo de energia para execuções com 8 núcleos são conflitantes, possuindo consumo energético maior que a execução em 4 núcleos acompanhado de um menor desempenho de processamento, assim como aconteceu para o *benchmark* `bayes`. Isto reforça a análise feita na subseção 6.3.7: um aumento no número de núcleos executando para o algoritmo `labyrinth` acabou por aumentar consideravelmente a chance de conflitos e, portanto, diminuiu o desempenho de processamento.

Este tipo de comportamento descrito no parágrafo anterior novamente se reflete na figura 6.22, onde a execução com 8 núcleos não foi vantajosa em nenhum caso. No entanto, neste caso, a desvantagem em relação à execução sequencial diminuiu quando se compara com o *benchmark* `labyrinth`. Isto porque a carga de trabalho maior acaba por amenizar os impactos causados pela sobrecarga imposta na execução pelo sistema de TM. Além disto, com mais dados a serem processados diminui-se também as chances de conflitos, já que a área de trabalho na memória é maior.

Este aumento na carga de trabalho fez com que, inclusive, execuções que antes não eram tão vantajosas como aquelas que utilizam 4 núcleos passassem a sê-lo: todas as execuções com 4 núcleos tiveram medidas de EDP melhores que a da execução sequencial, ao contrário do que aconteceu para o *benchmark* `labyrinth`. Esta vantagem em relação à execução sequencial aconteceu, também, para algumas políticas sendo executadas em 2 núcleos. As políticas mais vantajosas aqui foram as mesmas descritas na subseção 6.3.7: execuções que combinaram as políticas `BACKOFF_LINEAR` e `BACKOFF_EXPONENTIAL` com *Eager* e *Lazy*, além das execuções que incluíram a política `WAIT_ABORTER_ENABLED`. A diferença é que, para o *benchmark* `labyrinth+`, `BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager` também se apresentou bem vantajosa.

6.3.9 `ssca2`

O caso do *benchmark* `ssca2` é muito similar ao visto para o algoritmo `intruder` nas subseções 6.3.4 e 6.3.5. A introdução do sistema de TM, como pode ser visto na figura 6.23, aqui também, causou um grande impacto no desempenho do algoritmo, mesmo quando executando em um único núcleo. Isto se deve, em ambos os casos, às transações serem muito pequenas. Assim, a introdução de qualquer sobrecarga acaba prejudicando o desempenho do aplicativo. Esta queda de desempenho (o *speedup* é sempre menor que um), acompanhada de um crescimento no consumo de energia sempre maior que 400% faz com que o EDP se mostre desvantajoso em relação à execução sequencial para todas as políticas, como pode ser visto na figura 6.24

Ao contrário do `intruder`, no entanto, no caso do *benchmark* `ssca2` o aumento do número de núcleos executando diminui o EDP, refletindo o fato de que, apesar de sempre ruim, o *speedup* cresce à medida que se cresce o número de núcleos executando em todos

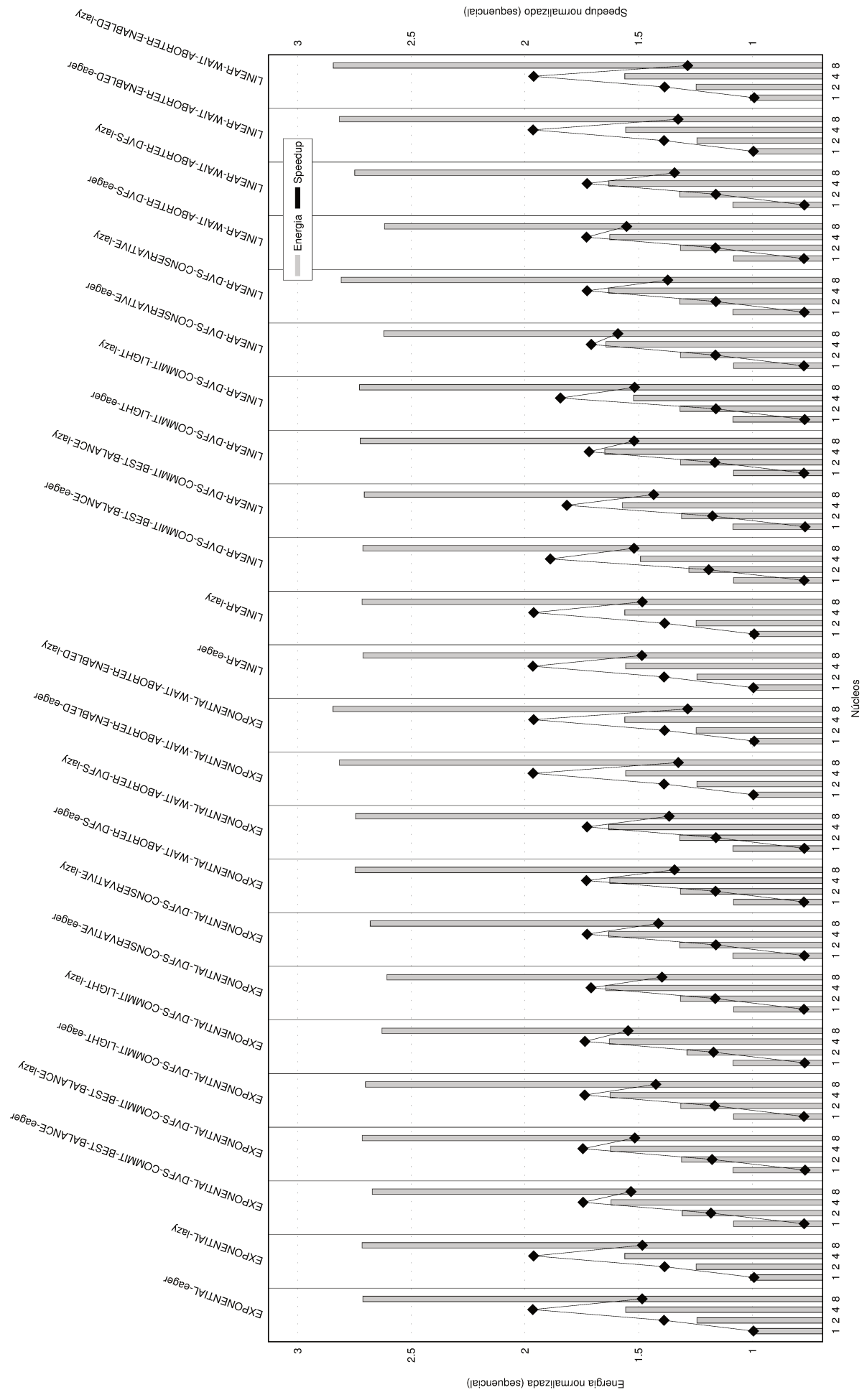


Figura 6.21: Energia e *speedup* normalizados em relação à execução sequencial — `labyrinth+`.

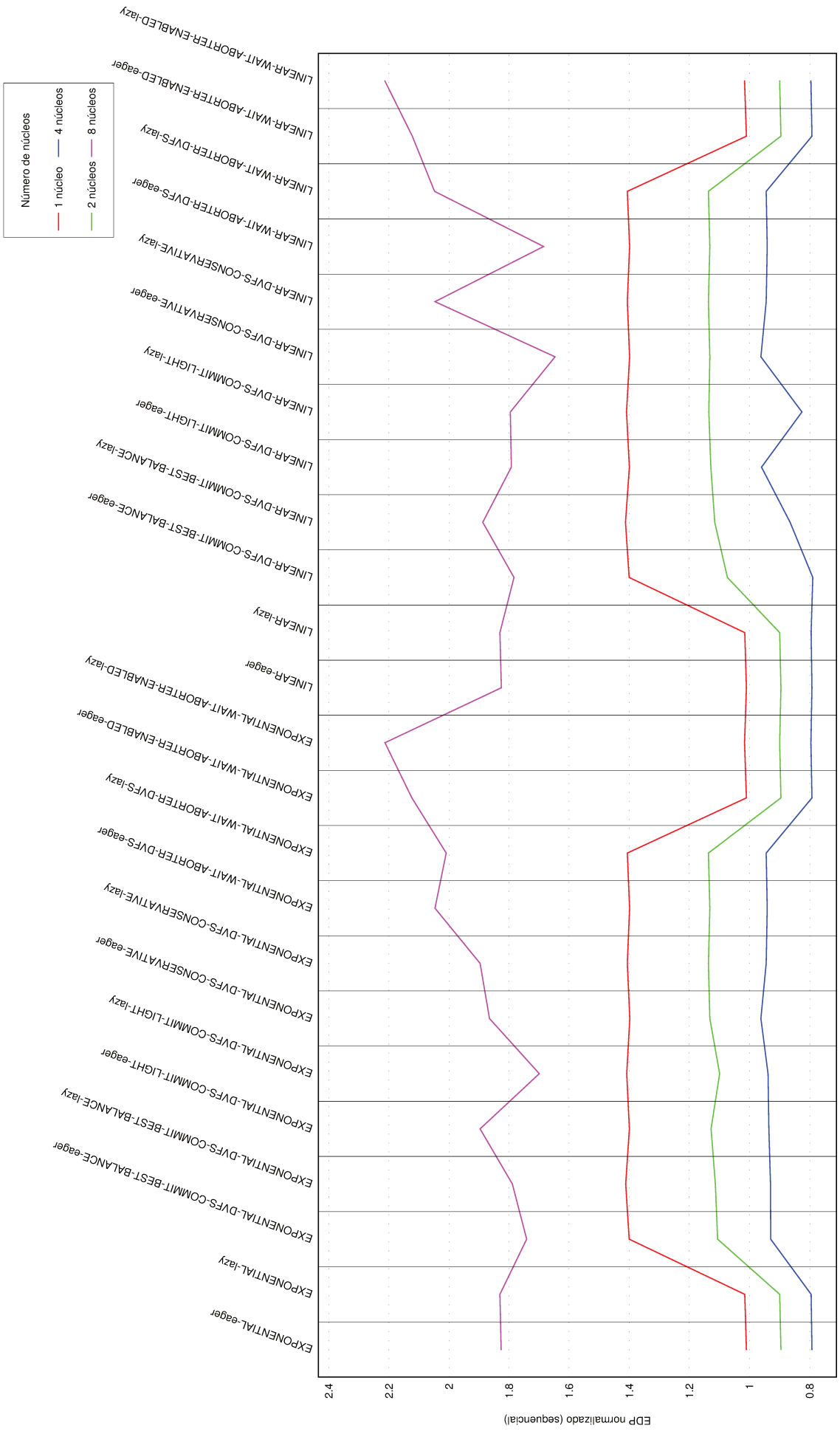


Figura 6.22: EDP normalizado em relação à execução sequencial — `labyrinth+`.

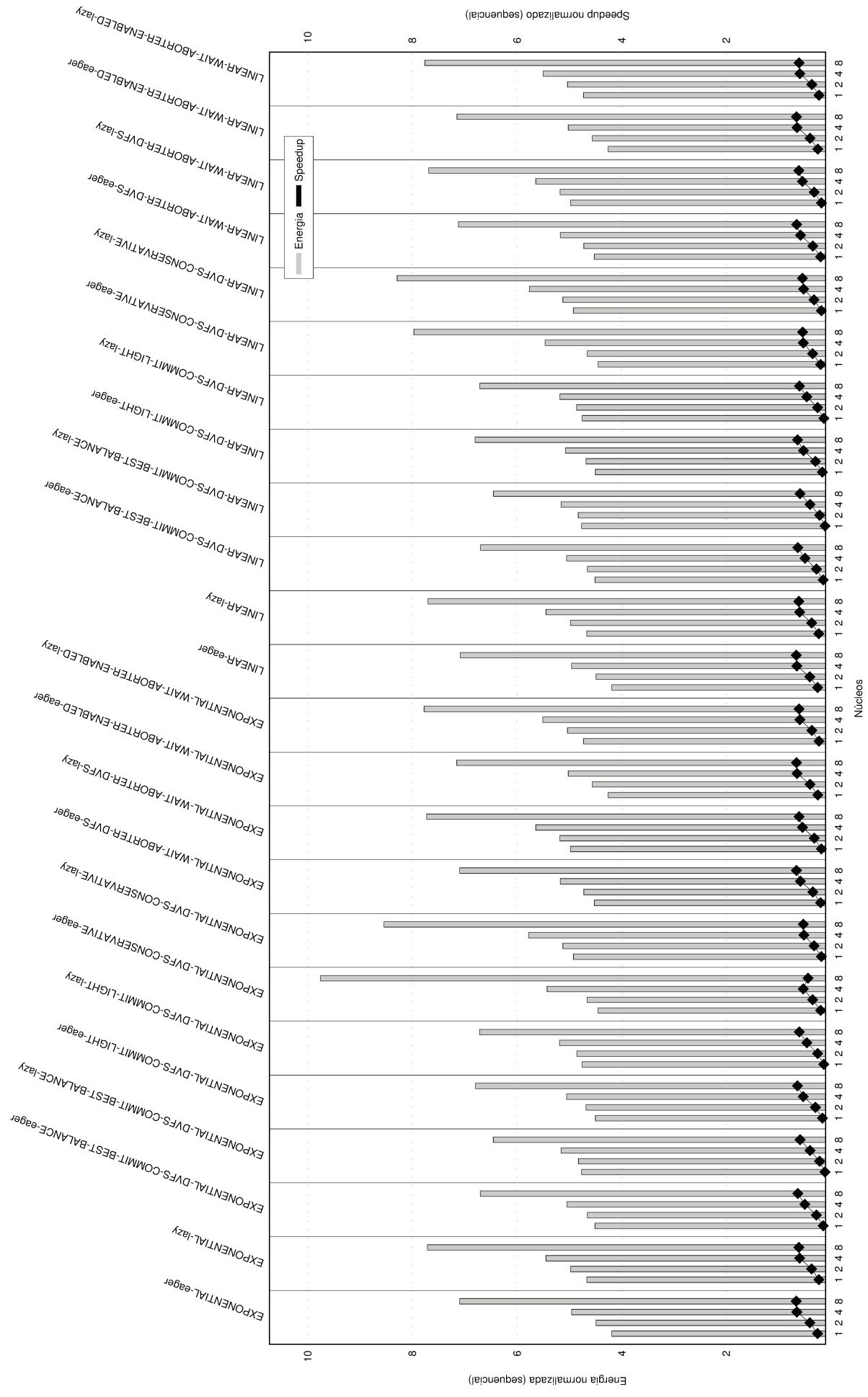


Figura 6.23: Energia e *speedup* normalizados em relação à execução sequencial — `ssca2`.

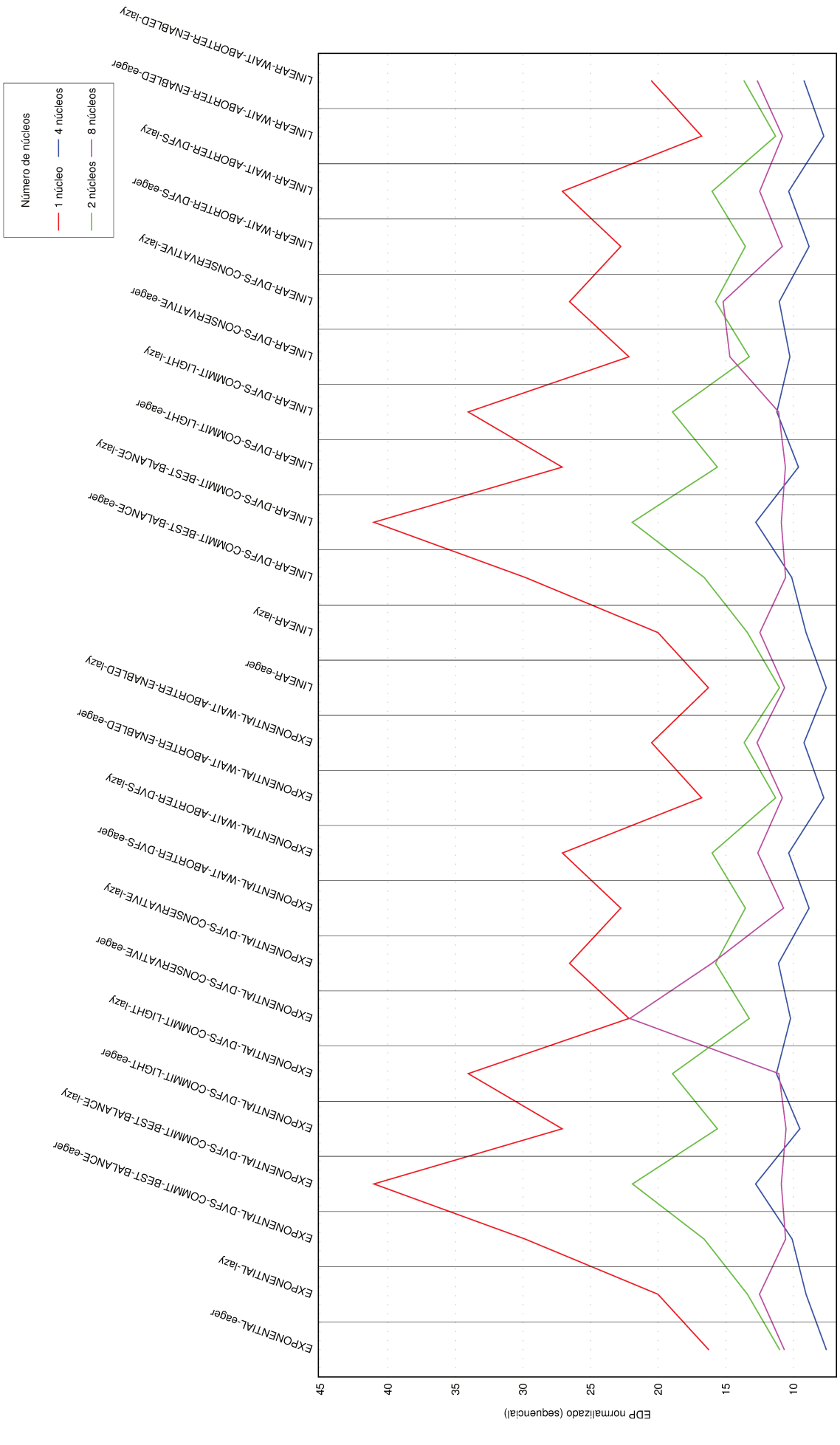


Figura 6.24: EDP normalizado em relação à execução sequencial — `ssca2`.

os casos exceto com a política BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager e o consumo de energia não cresce na mesma proporção (intruder possui mais contenção que ssca2). Este balanço entre tempo de execução e consumo de energia é cada vez mais vantajoso, de uma forma geral, se aumenta-se a quantidade de núcleos executando de um para dois e de dois para quatro. No entanto, com 8 núcleos, nem sempre o EDP é menor devido ao considerável aumento no consumo de energia experimentado com o uso desta quantidade de núcleos se comparado ao caso com 4 núcleos. Desta forma, para todas as políticas, exceto as execuções que incluem as políticas DVFS_COMMIT_BEST_BALANCE Lazy e a política BACKOFF_EXPONENTIAL DVFS_COMMIT_LIGHT Lazy, a execução com 4 núcleos é mais vantajosa.

Para o *benchmark* ssca2 também é possível identificar facilmente na figura 6.24 os contornos de dente-de-serra em todas as curvas. Assim, mesmo com transações pequenas, quanto antes se detecta o conflito mais vantajoso é e, por isto, as políticas *Eager* se mostram mais eficientes em relação ao EDP.

Por fim, apesar de o EDP melhorar com o aumento do número de núcleos executando desde que o número de núcleos não ultrapasse quatro, a conclusão final é a mesma que foi encontrada para o algoritmo intruder nas subseções 6.3.4 e 6.3.5: o ssca2 também é melhor executado sequencialmente. As políticas com melhor EDP neste caso, BACKOFF_LINEAR Eager e BACKOFF_EXPONENTIAL Eager possuem EDP 7,58 (normalizado em relação à execução sequencial).

6.3.10 vacation-high

O caso do *benchmark* vacation-high é similar aos casos dos algoritmos ssca2 e intruder. A introdução de TM, como pode ser vista na figura 6.25 causou um grande impacto no desempenho do algoritmo, mesmo quando executando em um único núcleo, indicando, assim como no ssca2, transações pequenas que, neste caso, quando executadas em paralelo, causam grande contenção no acesso aos dados. O *speedup* neste caso é sempre menor que um. No entanto, no caso do *benchmark* vacation-high, percebe-se na figura 6.25 que, apesar de o consumo de energia sempre aumentar quanto maior for a quantidade de núcleos envolvidos na execução, o mesmo não pode ser falado a respeito do *speedup*. Para execuções que envolvem políticas com BACKOFF_EXPONENTIAL, o desempenho quase sempre cai quando se compara a execução entre quatro e oito núcleos, indo contra aquilo que seria esperado no caso de execuções em paralelo com mais núcleos. Isto só não é verdade para as políticas BACKOFF_EXPONENTIAL DVFS_COMMIT_BEST_BALANCE Eager, BACKOFF_EXPONENTIAL DVFS_COMMIT_LIGHT Eager e BACKOFF_EXPONENTIAL WAIT_ABORTER_DVFS Eager. Já para execuções que envolvem políticas com BACKOFF_LINEAR, o desempenho quase sempre sobe quando se compara a execução entre quatro e oito núcleos, como se es-

pera no caso de execuções em paralelo com mais núcleos. Isto só não é verdade para as políticas `BACKOFF_LINEAR Eager`, `BACKOFF_LINEAR DVFS_CONSERVATIVE Eager` e `BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager`. Isto indica que, apesar de o algoritmo `vacation` parecer não apresentar boas características de escalabilidade com este nível de contenção utilizado para o *benchmark* `vacation-high`, ele é mais escalável, de uma forma geral, quando se utiliza políticas baseadas em *backoff* linear. Mesmo com esta divergência de comportamento, quando se olha apenas os números de *speedup*, é possível encontrar políticas baseadas em *backoff* exponencial que sejam quase tão boas quanto políticas baseadas em *backoff* linear.

O gráfico de EDP (figura 6.26) reflete o impacto negativo causado pela introdução de TM no *speedup* do *benchmark* `vacation-high`. Todos os EDP são piores que o do caso sequencial. No entanto, assim como para o *benchmark* `ssca2` e ao contrário do que acontece com o `intruder`, o aumento do número de núcleos executando diminui o EDP, com exceção para as execuções com 8 núcleos. Isto porque o algoritmo `vacation`, assim como o `ssca2`, possui uma contenção menor que a do algoritmo `intruder`. Para 8 núcleos, como foi visto no parágrafo anterior, o *speedup* é fortemente afetado pelo tipo de *backoff* utilizado. Mesmo assim, quando se calcula o EDP, ele é sempre pior que os EDP para a mesma política executando com 4 núcleos. Isto porque, como pode ser visto na figura 6.25, o consumo de energia para as execuções com 8 núcleos é muito maior que aquele nas execuções com 4 núcleos, chegando a ser 90% maior para a política `BACKOFF_EXPONENTIAL Lazy`. Desta forma, os melhores EDP são encontrados para as execuções com 4 núcleos.

Na figura 6.26 também é fácil identificar os contornos de dente-de-serra em todas as curvas. Além disto, a vantagem do *backoff* linear sobre o *backoff* exponencial para este *benchmark* notada anteriormente nesta subseção durante a análise da figura 6.25 também pode ser vista no gráfico de EDP. Apesar de as curvas parecerem repetir seu contorno para as mesmas políticas entre o lado esquerdo (*backoff* exponencial) e o lado direito (*backoff* linear) do gráfico, o dente-de-serra do lado direito é mais suave, com picos mais baixos e vales menos profundos, indicando um melhor balanço entre desempenho e consumo de energia para as execuções baseadas em políticas com *backoff* linear.

Por fim, apesar de a execução sequencial se mostrar vantajosa em todos os casos analisados para este *benchmark*, a execução com TM com melhor EDP foi com a política `BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager` em 4 núcleos, com valor 4,37 (normalizado em relação à execução sequencial).

6.3.11 `vacation-low`

O caso do *benchmark* `vacation-low` não é, no geral, muito diferente do `vacation-high`. A introdução de TM, aqui também, como pode ser visto na figura 6.27, causou um grande

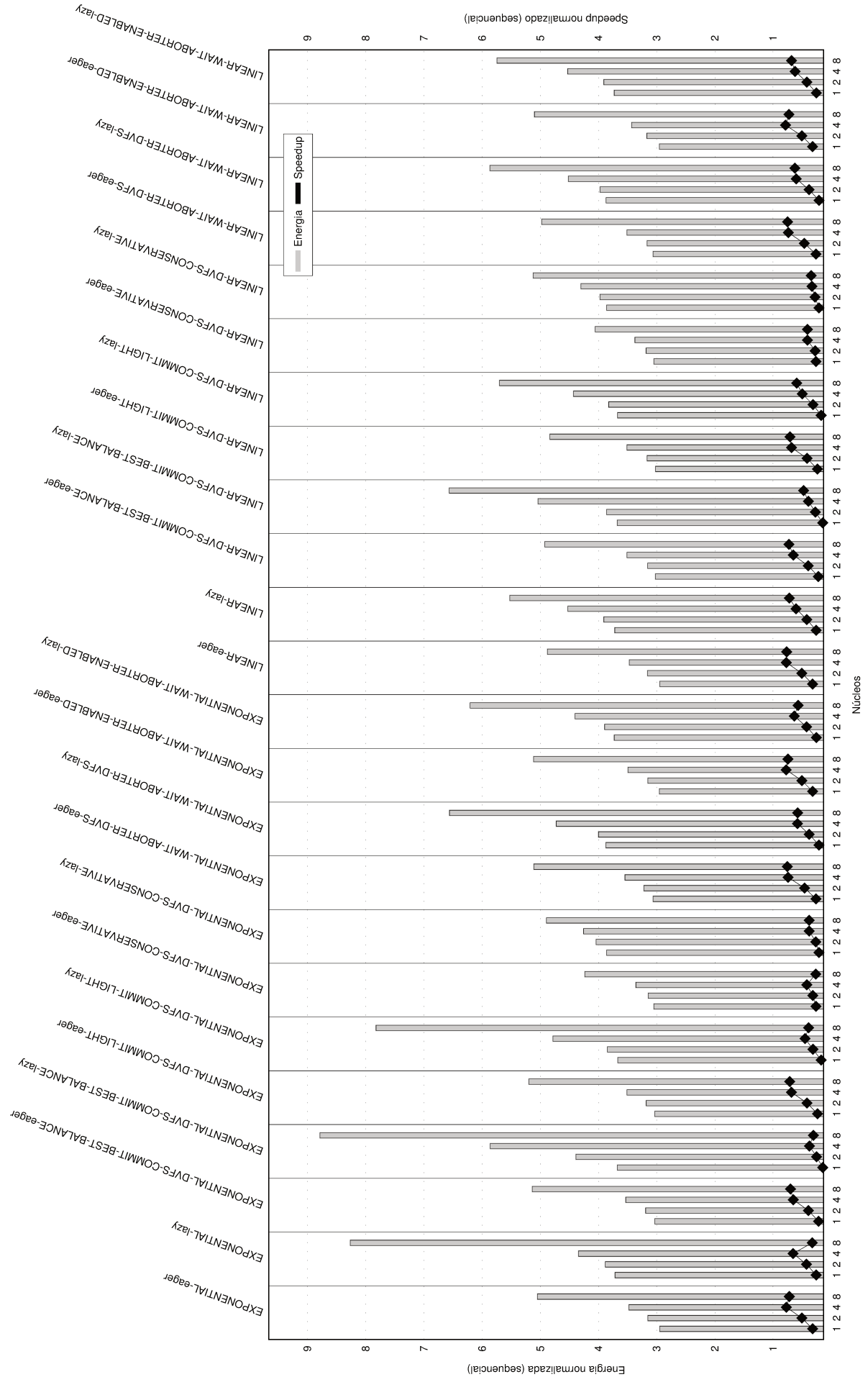


Figura 6.25: Energia e *speedup* normalizados em relação à execução sequencial — `vacation-high`.

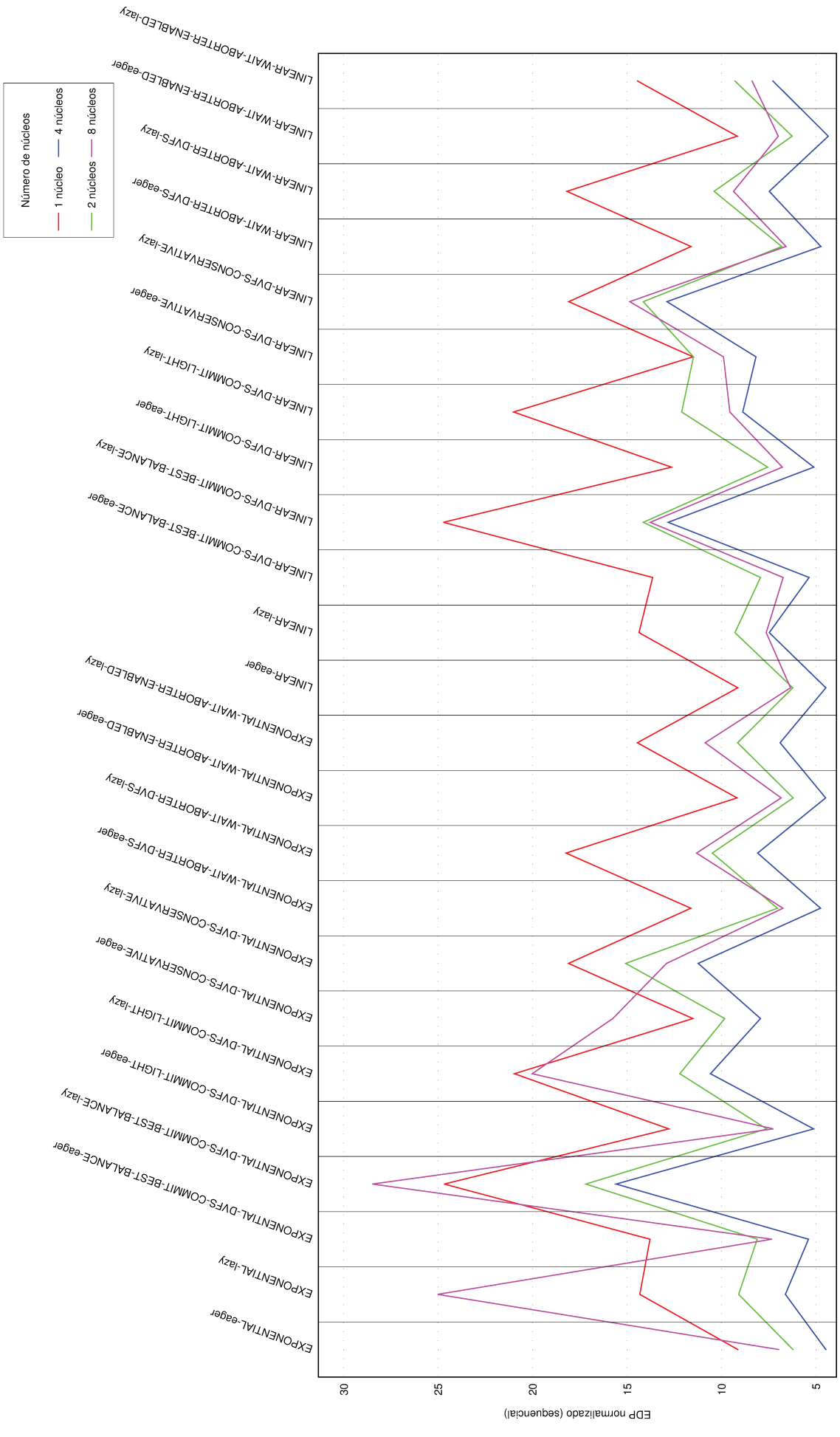


Figura 6.26: EDP normalizado em relação à execução sequencial — `vacation-high`.

impacto no desempenho do algoritmo, mesmo quando executando em um único núcleo. A diferença é que no caso do *benchmark* `vacation-low`, todas as execuções com 8 núcleos tiveram *speedup* maior que um quando normalizados em relação à execução sequencial, já que este *benchmark* possui menor contenção que o `vacation-high`, com exceção das execuções com variações da política `DVFS_CONSERVATIVE`, que se mostrou muito desvantajosa para esta carga de trabalho. Para algumas execuções como, por exemplo, `BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager`, o *speedup* para a execução com 4 núcleos foi maior que o para a execução com 8 núcleos. No entanto, na maior parte das políticas testadas, o *speedup* aumentou conforme se aumentava o número de núcleos executando, conforme é esperado para execuções em paralelo. Para esta carga de trabalho, o algoritmo `vacation` parece não mostrar a diferenciação entre políticas com *backoff* exponencial e com *backoff* linear apresentada no *benchmark* `vacation-high`. Isto se deve ao fato de esta carga de trabalho, que gera menos contenção, ter se apresentado mais escalável, como esperado, atingindo inclusive *speedups* vantajosos em relação à execução sequencial como visto. Desta maneira, a influência que o tipo de *backoff* havia demonstrado na escalabilidade para o cenário com maior contenção acabou não ficando visível neste cenário com menor contenção.

O gráfico de EDP na figura 6.28 também reflete o impacto negativo causado pela introdução de TM no *speedup*, assim como visto para o *benchmark vacation-high*. Todos os EDP mostram execuções piores que a do caso sequencial. Assim como no caso do *benchmark* `vacation-high`, o aumento do número de núcleos executando diminui o EDP, com exceção para as execuções com 8 núcleos e para as execuções com políticas `DVFS_CONSERVATIVE`, que, como foi visto no parágrafo anterior, possuem comportamento atípico para este *benchmark*. Inclusive, é possível identificar facilmente nas curvas de EDP com dois, quatro e oito núcleos, um pico nos valores relativos às políticas `DVFS_CONSERVATIVE`.

Para as execuções com 8 núcleos, assim como no caso do *benchmark* `genome`, algumas conclusões podem ser tiradas do gráfico da figura 6.28. Primeiramente, aqui também, as políticas com `DVFS_CONSERVATIVE` apresentaram resultado bem pior que as outras políticas, como já comentado. Além disto, caso se esteja interessado simplesmente em desempenho e retirando as políticas com `DVFS_CONSERVATIVE` da análise, o tempo de execução das políticas de TM em 8 núcleos é, na maior parte dos casos, inferior ao tempo de execução das mesmas políticas executando em 4 núcleos. No entanto, quando se analisa o balanço entre desempenho e consumo de energia através do EDP, as execuções com 4 núcleos são mais vantajosas. Isto porque, apesar de a execução com 8 núcleos entregar, na maior parte das vezes, melhor *speedup* quando comparada à execução com 4 núcleos, ela consome uma quantidade de energia relativamente maior do que o ganho de desempenho relativo que ela consegue entregar, tendo, portanto, um EDP menos eficiente. Esta coincidência de características entre o *benchmark* `genome` e o `vacation-low` é um

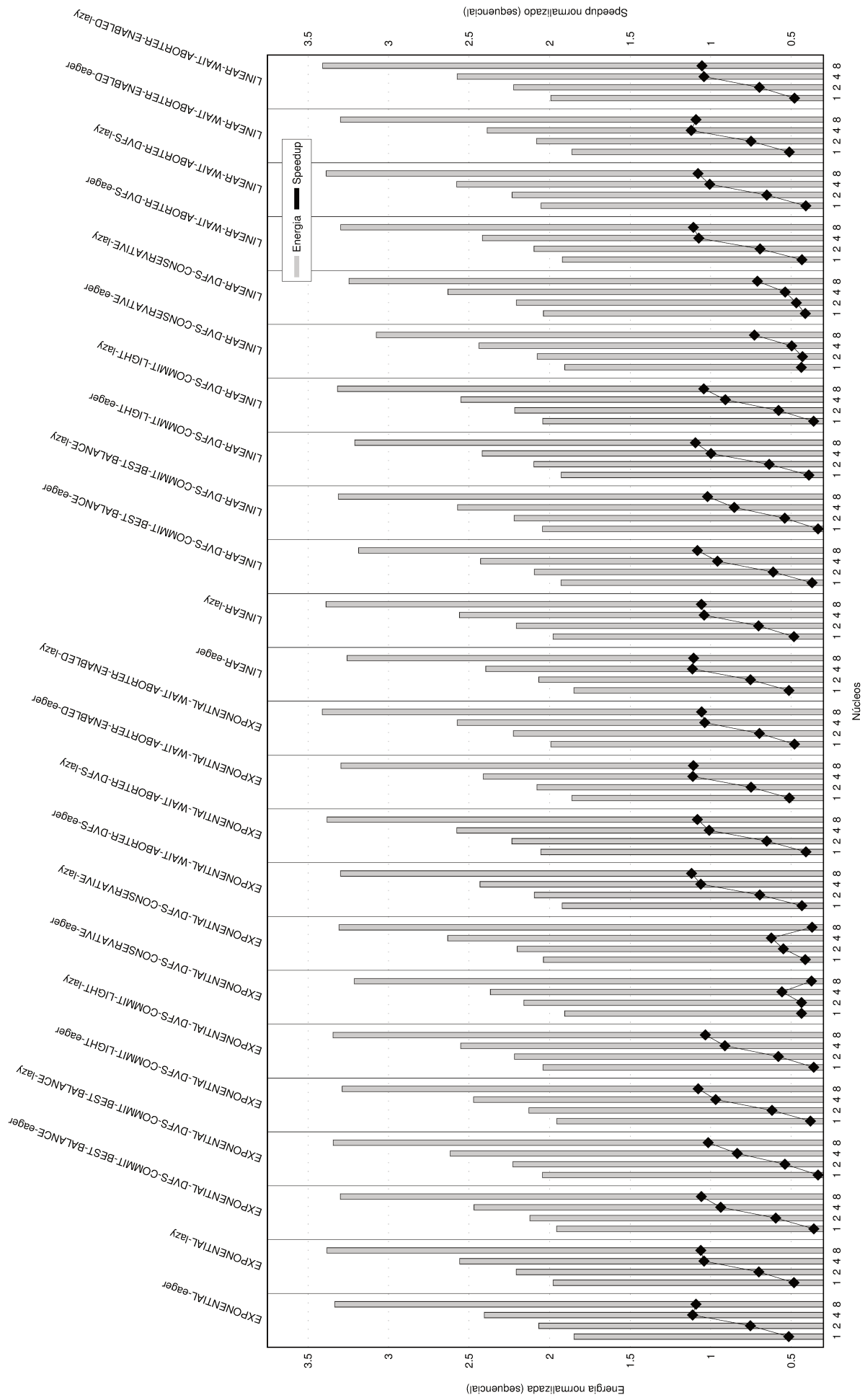


Figura 6.27: Energia e *speedup* normalizados em relação à execução sequencial — `vacation-low`.

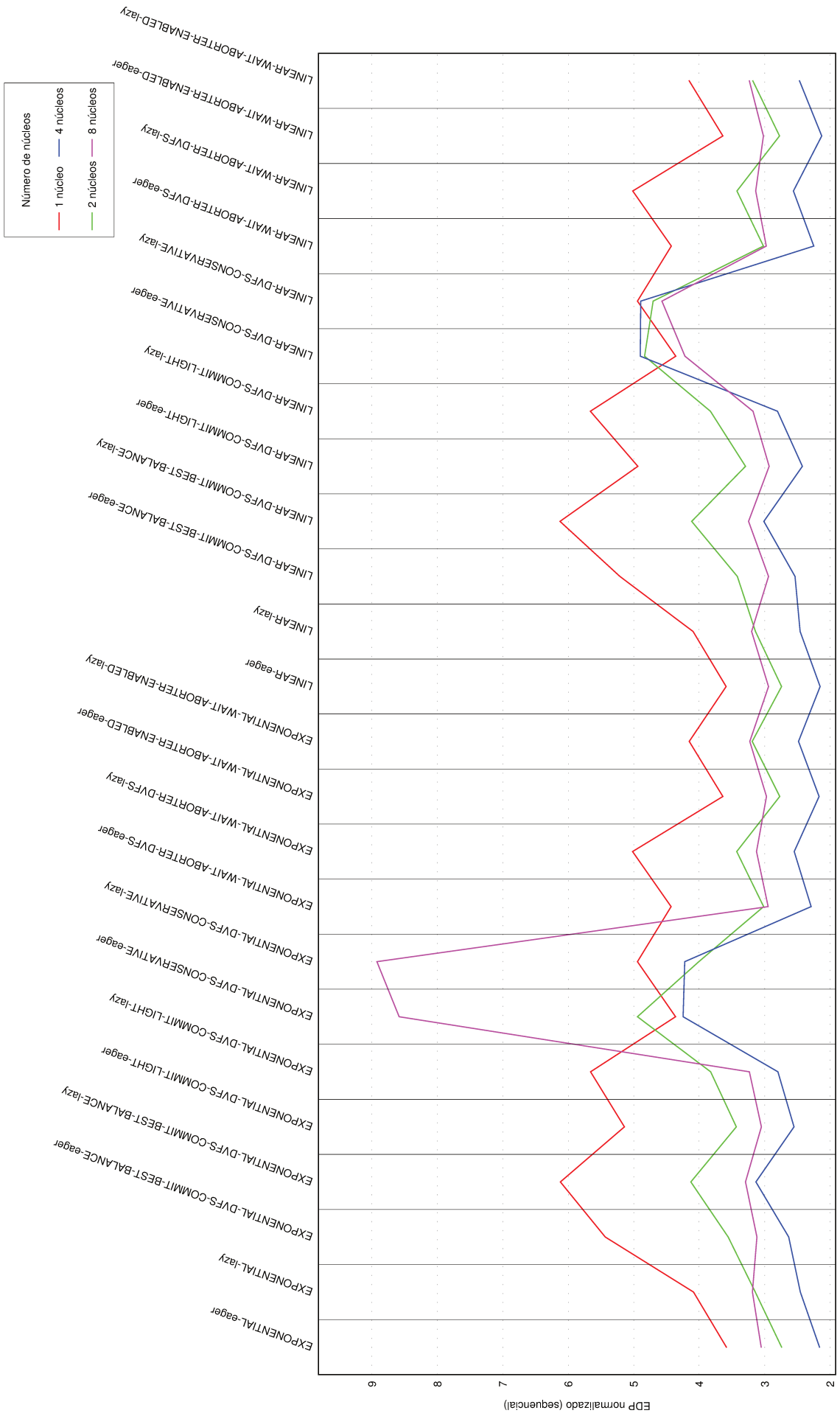


Figura 6.28: EDP normalizado em relação à execução sequencial — `vacation-low`.

reflexo do fato de ambos terem características de execução (tamanho das transações e nível de contenção, por exemplo) parecidas.

Assim como no *benchmark* `vacation-high`, na figura 6.28 é possível identificar os contornos de dente-de-serra em todas as curvas de EDP, indicando, mais uma vez, uma vantagem das políticas *Eager*.

Por fim, apesar de a execução sequencial se mostrar vantajosa em todos os casos analisados para este *benchmark*, a execução com TM com melhor EDP foi com a política `BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager` em 4 núcleos, com valor 2,13 (normalizado em relação à execução sequencial). Exatamente a mesma política que foi mais vantajosa no caso do *benchmark* `vacation-high`.

6.3.12 `yada`

O *benchmark* `yada`, como foi visto no início deste capítulo, foi o conjunto de execuções que mais problemas tiveram, o que prejudica um pouco sua análise final. Mesmo assim, através da figura 6.29 é possível ver que este aplicativo, com esta carga de trabalho, possui uma escalabilidade muito ruim: para todos os casos com dados válidos, o *speedup* para a execução com 8 núcleos é menor que o *speedup* para a execução com 4 núcleos. Isto quando o melhor *speedup* não é o da execução com um núcleo, como é o caso da política `BACKOFF_EXPONENTIAL Eager`. No entanto, apesar da escalabilidade ruim, para sete políticas o *speedup* ficou maior que um, mostrando que TM pode executar mais rápido que o caso sequencial. Estas políticas tinham `BACKOFF_LINEAR` combinado com as políticas *Eager* ou *Lazy*, sozinhas ou juntas com `WAIT_ABORTER_DVFS`, ou combinadas com a política *Eager* e um das seguintes outras políticas: `DVFS_CONSERVATIVE`, `DVFS_COMMIT_LIGHT` ou `DVFS_COMMIT_BEST_BALANCE`. O maior *speedup* (`BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager` executando em 4 núcleos) ficou 14% melhor que a execução sequencial.

Já em relação ao EDP, quando considerando apenas os dados válidos, é possível ver pela figura 6.30 que quanto maior o número de núcleos executando, maior o EDP para uma determinada política. Isto indica uma grande contenção no acesso aos dados. Contenção que piora quanto maior é o número de transações executando simultaneamente. Para o caso de 8 núcleos, este sintoma é ainda pior devido não somente ao fato de o *speedup* nestes casos poder ser menor que nas execuções com 4 núcleos como visto no parágrafo anterior mas também pelo fato de, como pode ser visto na figura 6.29, o consumo de energia ser consideravelmente maior para as execuções com 8 núcleos. Por exemplo, para a política `BACKOFF_EXPONENTIAL DVFS_COMMIT_LIGHT Lazy`, a combinação de um *speedup* quase 59% menor em relação à execução com 4 núcleos, aliado a um gasto de energia 276% maior na execução com 8 núcleos em relação à execução com 4 núcleos, acabou gerando

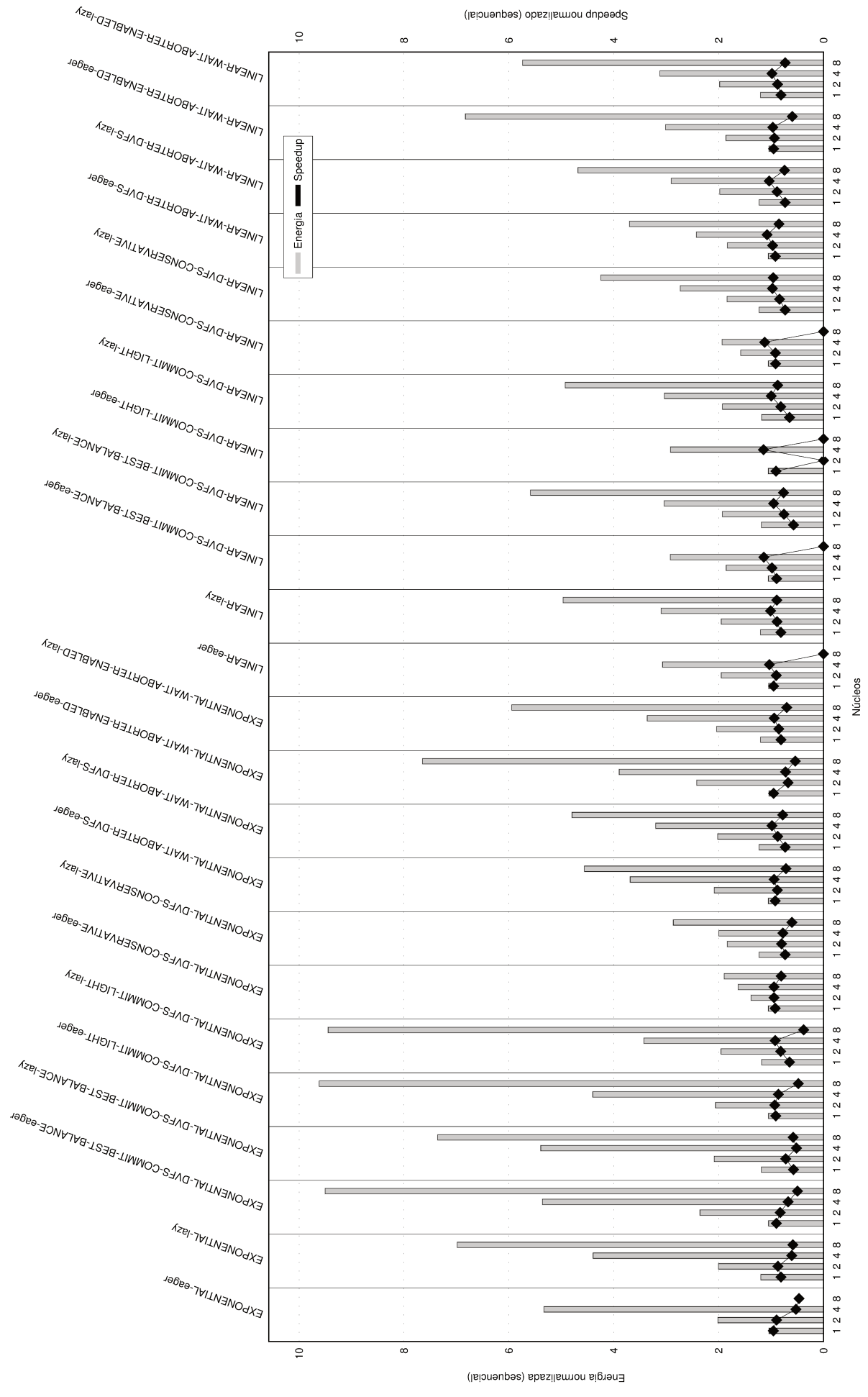


Figura 6.29: Energia e *speedup* normalizados em relação à execução sequencial — yada.

um EDP 669% maior na execução com 8 núcleos em relação à execução com 4 núcleos. Para o EDP, ao contrário do que aconteceu para o *speedup*, nenhuma execução foi mais vantajosa que o caso sequencial.

Na figura 6.30 é possível ver, para as curvas de um, dois e quatro núcleos, os contornos de dente-de-serra. Para a curva de 8 núcleos a falta de dados prejudica esta análise com fidelidade. Os picos e vales, porém, no caso do *benchmark* `yada` são bem menos expressivos que em alguns outros casos, indicando uma menor influência da forma de detecção de conflitos na eficiência do sistema de TM para este *benchmark*, o que pode ser um efeito do fato de a contenção, neste algoritmo, apesar de alta, não ser das maiores entre os *benchmarks* estudados.

As combinações de políticas que geraram o melhor EDP para o caso do *benchmark* `yada` foram *Eager* combinada com `BACKOFF_LINEAR` ou `BACKOFF_EXPONENTIAL` de forma isolada ou ainda combinados com `WAIT_ABORTER_ENABLED`. O valor do EDP para estes casos ficou em 1,09 (normalizado em relação ao caso sequencial).

6.4 Comparação entre políticas de Memórias Transacionais

Nesta seção serão combinadas as diversas análises feitas na seção 6.3 de forma a serem encontradas tendências e conclusões mais gerais sobre as políticas de TM testadas neste trabalho e não apenas conclusões válidas para cada um dos *benchmarks* do STAMP analisados.

Na tabela 6.2 é listado o comportamento de cada um dos *benchmarks* analisados neste trabalho em relação ao *speedup*. A coluna "*Speedup* aumenta" indica se o *speedup* aumenta com o aumento do número de núcleos executando conforme seria esperado. Neste momento, não está sendo analisado se o *speedup* é maior que um, o que indicaria que a execução paralela é mais rápida que a sequencial. Está sendo analisado simplesmente a tendência de aumento do *speedup* com o aumento do numero de núcleos executando. Como pode se ver na tabela, com exceção dos casos extremos (`kmeans-high` que é facilmente paralelizável e `intruder`, `intruder+` e `yada` que tem alta contenção), o *speedup* aumenta com o aumento do número de núcleos executando com exceção para os casos de execução com 8 núcleos para uma grande parte das políticas de TM analisadas.

Este comportamento espalhado por quase todos os *benchmarks* indica mais do que um possível desajuste entre políticas de TM estudadas e a execução com 8 núcleos. O fato de as execuções com 8 núcleos em geral apresentarem decréscimo de *speedup* indica que a infraestrutura simulada está sendo saturada com esta quantidade de núcleos executando. Suspeita-se que esta saturação esteja acontecendo no barramento AMBA-AHB, que parece

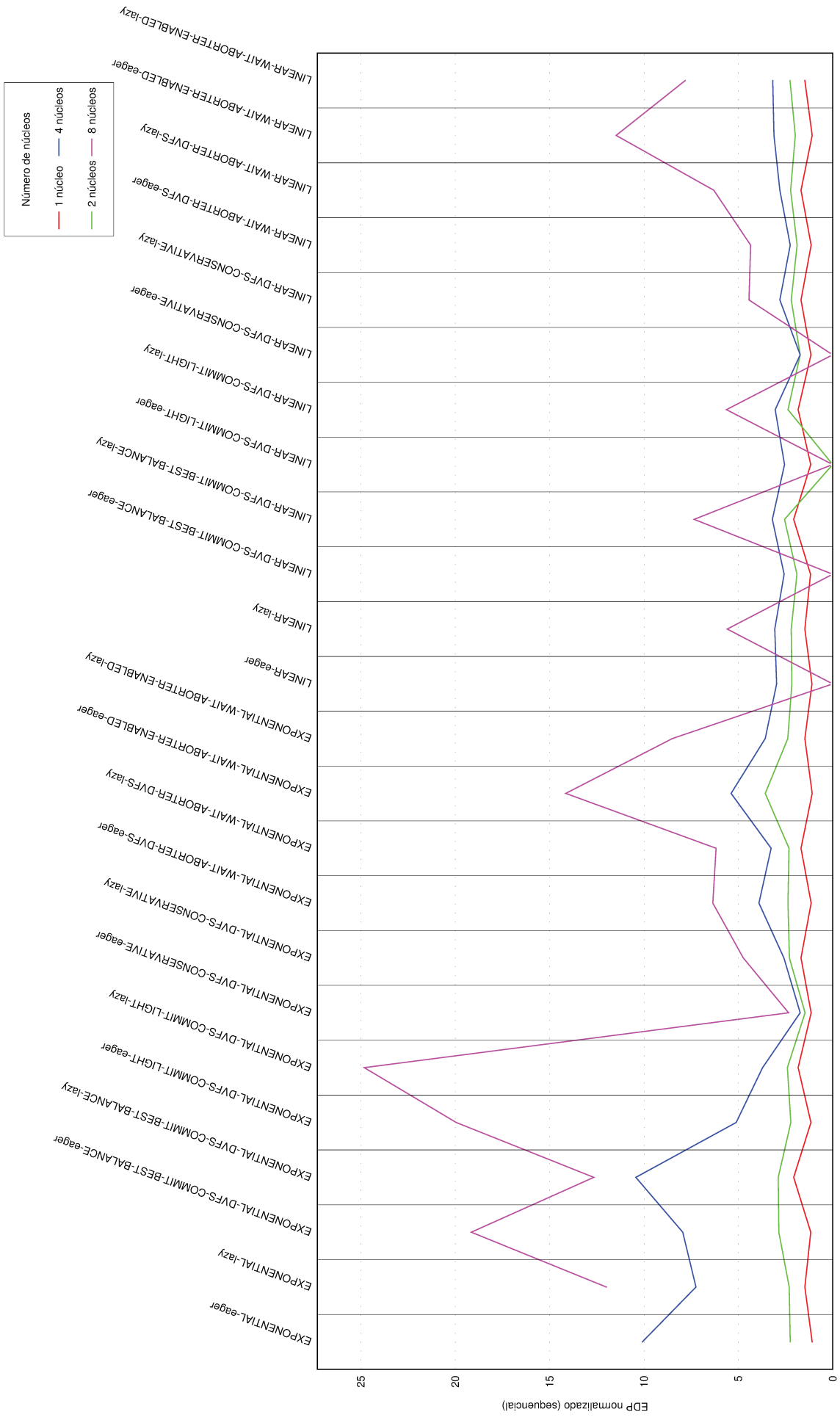


Figura 6.30: EDP normalizado em relação à execução sequencial — yada.

Tabela 6.2: Relação entre aumento de núcleos executando e *speedup*.

Benchmark	***Speedup* aumenta**
`bayes`	Sim, com exceção para as execuções com 8 núcleos
`genome`	Sim, com exceção para quase todas as execuções com políticas `DVFS_CONSERVATIVE` com 8 núcleos
`genome+`	Sim, com exceção para as execuções com políticas `BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE` com 8 núcleos
`intruder`	Usualmente não
`intruder+`	Usualmente não
`kmeans-high`	Sim
`labyrinth`	Sim, com exceção para as execuções com 8 núcleos para a maior parte das políticas
`labyrinth+`	Sim, com exceção para as execuções com 8 núcleos
`ssca2`	Sim, com exceção para as execuções com 8 núcleos para cerca de metade das políticas
`vacation-high`	Sim, com exceção para as execuções com 8 núcleos para a maior parte das políticas que envolvem `BACKOFF_EXPONENTIAL` e para algumas políticas que envolvem `BACKOFF_LINEAR`
`vacation-low`	Sim, com exceção de algumas execuções com 8 núcleos
`yada`	Sim, com exceção das execuções com 8 núcleos e de algumas outras execuções com menos núcleos

não ser capaz de escalar a comunicação necessária para a execução com este número de núcleos. Infelizmente a plataforma MPARM não permite simulação com 16 núcleos. Se isto fosse possível seria fácil visualizar se esta quebra no aumento do *speedup* era devido à saturação da plataforma simulada ou devido a algum outro fator. No entanto, devido às características do barramento e da plataforma simulada e devido à forma como as aplicações escalaram com menos núcleos executando, suspeita-se que seja esta a causa da quebra no aumento do *speedup* para grande parte das execuções com 8 núcleos. Esta conclusão, aliás, é corroborada por ao menos um outro estudo [69], o qual mostra que o barramento AMBA-AHB nitidamente não escala em uma plataforma MPARM com 8 processadores executando aplicativos que geram saturação do canal de interconexão (cenário similar ao explorado neste trabalho com as aplicações paralelas gerando contenção no acesso a dados).

Já na tabela 6.3 é listado o comportamento de cada um dos *benchmarks* analisados neste trabalho em relação ao EDP. A coluna “EDP diminui” indica se o EDP diminui com

o aumento do número de núcleos executando conforme seria esperado. Neste momento, não está sendo analisado se o EDP é menor que um, o que indicaria que a execução paralela é mais eficiente que a sequencial. Está sendo analisado simplesmente a tendência de diminuição do EDP com o aumento do número de núcleos executando. Pela tabela é possível ver uma piora em relação aos resultados mostrados na tabela 6.2 para o *speedup*. O EDP, em geral, diminui com o aumento do número de núcleos executando com exceção para os casos de execução com 8 núcleos. No entanto, para o EDP, este comportamento acontece com quase a totalidade das políticas estudadas quando executando com 8 núcleos. Além disto, o *benchmark* `labyrinth` que tinha um comportamento dentro do esperado para algumas políticas em relação ao *speedup* tem um desempenho muito ruim em relação ao EDP.

Tabela 6.3: Relação entre aumento de núcleos executando e EDP

Benchmark	**EDP diminui**
`bayes`	Sim, com exceção para as execuções com 8 núcleos
`genome`	Sim, com exceção para a maior parte das execuções com 8 núcleos
`genome+`	Sim, com exceção de algumas execuções com 8 núcleos
`intruder`	Não
`intruder+`	Não
`kmeans-high`	Sim: as execuções com 4 núcleos possuem EDP numericamente menor para a maior parte dos casos, mas as execuções com 8 núcleos possuem EDP com baixa variação entre si e em relação aos menores valores obtidos nas execuções com 4 núcleos em todos os casos, o que não acontece nas execuções com 4 núcleos
`labyrinth`	Não, com exceção para algumas execuções com 2 núcleos e uma execução com 4 núcleos
`labyrinth+`	Sim, com exceção para as execuções com 8 núcleos
`ssca2`	Sim, com exceção para quase todas as execuções com 8 núcleos
`vacation-high`	Sim, com exceção para as execuções com 8 núcleos
`vacation-low`	Sim para a maior parte dos casos, com exceção para as execuções com 8 núcleos
`yada`	Não

Além do problema já comentado anteriormente sobre a limitação da plataforma simulada utilizada neste trabalho para as execuções com 8 núcleos que afetou diretamente o *speedup* nestas configurações, a piora em relação ao EDP deve-se também ao aumento considerável do consumo de energia nas execuções com 8 núcleos em relação às execuções

com 4 núcleos. O aumento do número de *threads* executando simultaneamente aumenta as chances de conflito, já que os acessos à memória são mais frequentes. Este aumento de possíveis conflitos, como se pode ver pelas análises nesta subseção, afeta o tempo de execução mas também tem um impacto considerável sobre o consumo de energia para a execução dos *benchmarks*, levando a uma piora ainda maior das medidas de EDP.

De uma forma geral, a tabela 6.4 sumariza as tendências observadas até agora nesta subseção, indicando com quantos núcleos tem-se o melhor *speedup* e o melhor EDP para a maior parte das políticas estudadas nesta dissertação, considerando-se o uso de TM para todos os *benchmarks*. Esta não é uma análise sobre qual é a melhor combinação de política de TM e quantidade de núcleos executando, mas sim uma tendência geral que pode ser obtida através da análise dos gráficos da seção 6.3. Pela tabela pode-se concluir que os melhores *speedups* estão bem divididos entre execuções em quatro e oito núcleos. Já os melhores EDP concentram-se nas execuções com 4 núcleos.

Tabela 6.4: *Speedup* e EDP em relação ao número de núcleos executando.

Benchmark	**Maior *speedup***	**Menor EDP**
`bayes`	4	4
`genome`	8	4
`genome+`	8	divido entre 4 e 8
`intruder`	não definido	1
`intruder+`	não definido	1
`kmeans-high`	8	4
`labyrinth`	4	2
`labyrinth+`	4	4
`ssca2`	dividido entre 4 e 8	4
`vacation-high`	dividido entre 4 e 8	4
`vacation-low`	8	4
`yada`	não definido	1

Portanto, na plataforma simulada usada neste estudo, sem saber muito sobre o *benchmark* a ser executado, as chances de se obter melhor *speedup* e melhor EDP ao se utilizar um sistema de TM para controlar o acesso aos dados compartilhados em uma execução paralelizada seriam maiores caso o *benchmark* fosse executado em 4 núcleos de acordo com a análise feita para este conjunto de *benchmarks* do STAMP usados neste trabalho.

Caso, ao invés de se olhar apenas as tendências de cada uma das curvas nos gráficos analisados na seção 6.3, fosse analisada a melhor combinação de quantidade de núcleos

executando e política de TM para cada *benchmark* do STAMP estudado, obter-se-ia a tabela 6.5. Nela pode-se encontrar não apenas a melhor configuração para *speedup* e EDP assim como o valor normalizado em relação à execução sequencial para cada entrada.

Tabela 6.5: Melhores *Speedup* e EDP.

Benchmark	**Maior *speedup***	**Menor EDP**
`bayes`	`BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager` 4 núcleos (3,11)	`BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager` 4 núcleos (0,31)
`genome`	`BACKOFF_LINEAR WAIT_ABORTER_DVFS Eager` 8 núcleos (1,72)	`BACKOFF_LINEAR Eager` 4 núcleos (1,16)
`genome+`	`BACKOFF_LINEAR Eager` 8 núcleos (2,01)	`BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager` 4 núcleos (0,97)
`intruder`	`BACKOFF_LINEAR Eager` 4 núcleos (0,42)	`BACKOFF_LINEAR Eager` 1 núcleo (10,53)
`intruder+`	`BACKOFF_LINEAR Eager` 4 núcleos (0,51)	`BACKOFF_LINEAR Eager` 2 núcleos (9,49)
`kmeans-high`	`BACKOFF_EXPONENTIAL DVFS_COMMIT_BEST_BALANCE Eager` 8 núcleos (4,13)	`BACKOFF_LINEAR WAIT_ABORTER_DVFS Lazy` 4 núcleos (0,28)
`labyrinth`	`BACKOFF_EXPONENTIAL WAIT_ABORTER_ENABLED Eager` 4 núcleos (1,48)	`BACKOFF_LINEAR Eager` 1 núcleo (1,02)
`labyrinth+`	`BACKOFF_LINEAR Eager` 4 núcleos (1,97)	`BACKOFF_LINEAR Eager` 4 núcleos (0,79)
`ssca2`	`BACKOFF_EXPONENTIAL WAIT_ABORTER_ENABLED Eager` 8 núcleos (0,66)	`BACKOFF_LINEAR Eager` 4 núcleos (7,58)
`vacation-high`	`BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager` 4 núcleos (0,79)	`BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager` 4 núcleos (4,37)
`vacation-low`	`BACKOFF_EXPONENTIAL WAIT_ABORTER_DVFS Eager` 8 núcleos (1,12)	`BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager` 4 núcleos (2,13)
`yada`	`BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager` 4 núcleos (1,14)	`BACKOFF_LINEAR Eager` 1 núcleo (1,09)

Algumas conclusões podem ser tiradas da análise da tabela 6.5. Para o *speedup*, `BACKOFF_LINEAR Eager` com 4 núcleos foi a combinação que entregou melhor resultado mais vezes. Ignorando a quantidade de núcleos executando, `BACKOFF_LINEAR Eager` e `BACKOFF_EXPONENTIAL WAIT_ABORTER_ENABLED Eager` foram as melhores. Já para o EDP, `BACKOFF_LINEAR Eager` entregou o melhor resultado mais vezes, seguida de `BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager`. No entanto, a vantagem da política `BACKOFF_LINEAR Eager` é demonstrada especialmente em *benchmarks* com alta conten-

ção e que não tem bom desempenho quando paralelizados. Isto acontece porque esta é a política que insere a menor sobrecarga na execução com detecção precoce de conflitos, portanto é natural que em um cenário em que a paralelização do código não ajuda, esta seja a política que dê melhores resultados. No entanto, em cenários de alta contenção as execuções sequenciais são mais vantajosas. A tabela 6.6 mostra o conteúdo da tabela 6.5 sem os valores normalizados para *speedup* e EDP, mas considerando se a execução sequencial não é a melhor opção para cada um dos *benchmarks*.

Tabela 6.6: Melhores *Speedup* e EDP incluindo o caso sequencial.

Benchmark	**Maior *speedup***	**Menor EDP**
bayes	BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager 4 núcleos	BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager 4 núcleos
genome	BACKOFF_LINEAR WAIT_ABORTER_DVFS Eager 8 núcleos	SEQUENTIAL
genome+	BACKOFF_LINEAR Eager 8 núcleos	BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager 4 núcleos
intruder	SEQUENTIAL	SEQUENTIAL
intruder+	SEQUENTIAL	SEQUENTIAL
kmeans-high	BACKOFF_EXPONENTIAL DVFS_COMMIT_BEST_BALANCE Eager 8 núcleos	BACKOFF_LINEAR WAIT_ABORTER_DVFS Lazy 4 núcleos
labyrinth	BACKOFF_EXPONENTIAL WAIT_ABORTER_ENABLED Eager 4 núcleos	SEQUENTIAL
labyrinth+	BACKOFF_LINEAR Eager 4 núcleos	BACKOFF_LINEAR Eager 4 núcleos
ssca2	SEQUENTIAL	SEQUENTIAL
vacation-high	SEQUENTIAL	SEQUENTIAL
vacation-low	BACKOFF_EXPONENTIAL WAIT_ABORTER_DVFS Eager 8 núcleos	SEQUENTIAL
yada	BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager 4 núcleos	SEQUENTIAL

A tabela 6.6, em princípio, mostra um cenário relativamente desanimador para STM. O caso sequencial é o mais rápido em um terço das execuções e tem o melhor EDP em dois terços das execuções. Isto indica que, quando se leva em consideração apenas os números obtidos pelas análises das execuções das simulações, STM não demonstra ser uma alternativa viável para paralelização de código. Isto, aliás, é defendido por outros autores [19], apesar de ser um assunto bem controverso [21].

Por outro lado, considerando-se que sistemas de TM podem ter implementações em

hardware e considerando que estas mesmas, na maior parte dos casos, como pode ser visto em [17], terão uma tendência de comportamento similar ao de uma STM porém com maior desempenho, pode-se utilizar os resultados obtidos para STM como forma de análise de tendências sobre o que ocorrerá quando as mesmas políticas forem implementadas em um sistema de HTM. Neste sentido, desconsiderando-se os resultados de *speedup* da tabela 6.6 para *benchmarks* com alta contenção (`intruder`, `intruder+` e `vacation-high`) e desconsiderando os resultados de EDP para *benchmarks* que indicam configurações com um ou dois núcleos como sendo melhores (`intruder`, `intruder+`, `labyrinth` e `yada`) — o que, no fim das contas, engloba os algoritmos que apresentam maior contenção e indica que paralelização não está trazendo grandes vantagens — tem-se o resultado da tabela 6.7.

Tabela 6.7: Melhores *Speedup* e EDP incluindo o caso sequencial e excluindo os casos de alta contenção.

Benchmark	**Maior *speedup***	**Menor EDP**
`bayes`	`BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager` 4 núcleos	`BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager` 4 núcleos
`genome`	`BACKOFF_LINEAR WAIT_ABORTER_DVFS Eager` 8 núcleos	`SEQUENTIAL`
`genome+`	`BACKOFF_LINEAR Eager` 8 núcleos	`BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager` 4 núcleos
`kmeans-high`	`BACKOFF_EXPONENTIAL DVFS_COMMIT_BEST_BALANCE Eager` 8 núcleos	`BACKOFF_LINEAR WAIT_ABORTER_DVFS Lazy` 4 núcleos
`labyrinth`	`BACKOFF_EXPONENTIAL WAIT_ABORTER_ENABLED Eager` 4 núcleos	DESCONSIDERADO
`labyrinth+`	`BACKOFF_LINEAR Eager` 4 núcleos	`BACKOFF_LINEAR Eager` 4 núcleos
`ssca2`	`SEQUENTIAL`	`SEQUENTIAL`
`vacation-high`	DESCONSIDERADO	`SEQUENTIAL`
`vacation-low`	`BACKOFF_EXPONENTIAL WAIT_ABORTER_DVFS Eager` 8 núcleos	`SEQUENTIAL`
`yada`	`BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Eager` 4 núcleos	DESCONSIDERADO

As políticas mais vantajosas para *speedup* na tabela 6.7 foram as que envolveram `BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager`, `WAIT_ABORTER Eager` com ou sem DVFS e `DVFS_COMMIT_BEST_BALANCE Eager`, além da política `BACKOFF_LINEAR Eager` que, como já visto, é a que inclui a menor sobrecarga sobre a execução com detecção

de conflitos precoce. Por outro lado, para o EDP, as políticas mais vantajosas foram aquelas que envolveram `DVFS_CONSERVATIVE Eager` e `WAIT_ABORTER Eager` com ou sem DVFS, além da política `BACKOFF_LINEAR Eager` e da execução sequencial, que se mostrou vantajosa em alguns casos. Portanto, os grupos de políticas vantajosas para *speedup* e EDP são bem similares. Assim, para a execução de um algoritmo na plataforma simulada estudada, sugere-se tentar a execução sequencial e a execução com STM com quatro ou oito núcleos destas políticas.

Aliás, a predominância das políticas *Eager* dentre as melhores políticas vai de encontro com as observações de contornos dente-de-serra visualizados nos gráficos de EDP da seção 6.3. Lá concluiu-se para vários *benchmarks* que as políticas *Eager*, em geral, tinham menor EDP que as políticas *Lazy*. Como as políticas *Eager* e *Lazy* foram plotadas alternadamente nos gráficos, isto deu uma aparência de dente-de-serra a algumas curvas.

Entretanto, a curva dente-de-serra nem sempre está presente, o que não indica que as políticas *Eager* sejam piores nestes casos. Na verdade as políticas *Eager* mostraram-se melhores para todos os *benchmarks* com exceção do `kmeans-high`. As curvas dente-de-serra, assim como outras características das curvas de EDP também podem mudar de comportamento à medida que se aumenta o número de núcleos executando ou à medida que se altera o tamanho da carga de trabalho sendo processada.

Os *benchmarks* `genome` e `genome+`, por exemplo, apresentam o contorno dente-de-serra para o EDP nas execuções com um, dois e quatro núcleos. No entanto, este contorno não é mais visível nas execuções com 8 núcleos. Este tipo de tendência indica que com mais núcleos executando e a maior concorrência pelo acesso aos dados as diferentes sobrecargas impostas pelas políticas *Eager* e *Lazy* passam a ser menos relevantes quando o algoritmo apresenta uma boa escalabilidade como é o caso do algoritmo `genome`. No caso de *benchmarks* facilmente paralelizáveis e de baixa contenção, como o `kmeans-high`, os contornos de dente-de-serra somem ainda mais rápido com o aumento do número de núcleos executando e já não são perceptíveis a partir da execução com 4 núcleos.

Já para *benchmarks* com alta contenção, como `intruder`, `intruder+`, `vacation-high` e `vacation-low`, o contorno de dente-de-serra é mais proeminente quanto maior o número de núcleos executando. Isto porque um maior número de *threads* simultâneas acaba gerando ainda mais contenção. Com isto, é um desperdício de recursos executar as transações até o fim para, só então, descobrir um conflito. Como um grande número de transações são abortadas neste tipo de algoritmo, quanto antes o conflito for detectado, menor será a sobrecarga imposta pelo sistema de TM. Por isto, nestes casos, as políticas *Eager* são ainda mais vantajosas que as *Lazy* quanto maior for o número de núcleos executando. O *benchmark* `yada` provavelmente se encaixaria nesta categoria, no entanto a falta de dados para muitas de suas execuções com 8 núcleos prejudica esta análise. Já o algoritmo `ssca2`, apesar de apresentar este mesmo comportamento, teoricamente é um

algoritmo de baixa contenção. Nos experimentos feitos neste trabalho, no entanto, seu comportamento sempre esteve mais próximo do apresentado pelos algoritmos de alta contenção, característica que também foi encontrada em outros trabalhos como, por exemplo, [13].

Por outro lado, alguns *benchmarks* que possuem alta contenção não apresentam os contornos dente-de-serra, como são os casos dos *benchmarks* `bayes`, `labyrinth` e `labyrinth+`.

É claro, também, que certos *benchmarks* possuem características peculiares, em relação ao EDP, para determinadas combinações de políticas de TM e número de núcleos executando, como os vales em algumas das políticas envolvendo `DVFS_COMMIT` nas execuções com 2 núcleos no `bayes` ou os picos nas políticas com `DVFS_CONSERVATIVE` nas execuções com 8 núcleos no `genome` e `genome+`. Estas características são, na maior parte das vezes, relativas a especificidades das interações entre o *benchmark* e a política sendo testada. No entanto, em vários gráficos é possível visualizar que, apesar das diversas curvas terem diferentes comportamentos, em geral, a região à esquerda, central e mais à direita das curvas tendem a possuir valores menores em relação ao seu entorno. Este comportamento pode ser visualizado de forma mais nítida em várias curvas dos gráficos gerados para os *benchmarks* `genome`, `genome+`, `labyrinth`, `labyrinth+`, `ssca2`, `vacation-high` e `vacation-low`. Estas regiões nos gráficos de EDP correspondem às políticas que combinam `BACKOFF_EXPONENTIAL` e `BACKOFF_LINEAR` com *Eager* e *Lazy* apenas e políticas envolvendo `WAIT_ABORTER` com e sem DVFS. Esta percepção que pode ser rapidamente extraída dos gráficos bate com os números coletados anteriormente nesta seção sobre quais são as melhores políticas em relação ao EDP, com exceção das políticas `BACKOFF_EXPONENTIAL Eager` e `BACKOFF_EXPONENTIAL Lazy` que, apesar de apresentarem estas características nos gráficos não figuram entre as mais vantajosas para o EDP — elas são vantajosas em relação à maioria das outras políticas, mas não em relação àquelas que geralmente apresentam o melhor valor de EDP.

Especificamente em relação às políticas `WAIT_ABORTER` com e sem DVFS, respectivamente `WAIT_ABORTER_DVFS` e `WAIT_ABORTER_ENABLED`, como visto no capítulo 3, elas são praticamente iguais, com a diferença de que a política com DVFS diminui a frequência de execução do núcleo durante a espera. É claro que esta diminuição da frequência pode gerar impactos negativos na execução pois ela influencia a ordem como as transações são executadas, mas seria intuitivo pensar que com uma menor frequência durante a espera, gastar-se-ia menos energia para estas políticas de uma forma geral. Esta percepção é confirmada para todos os *benchmarks* analisados neste trabalho, com exceção do `intruder` e `intruder+` com *backoff* exponencial e dos *benchmarks* `ssca2`, `vacation-high` e `vacation-low`.

Outro aspecto que pode ser concluído a partir das diversas análises de gráficos feitas na seção 6.3 é sobre como o EDP se comporta ao se variar a carga de trabalho para

um mesmo algoritmo. Os algoritmos `genome`, `intruder`, `labyrinth` e `vacation` foram executados, cada um, com duas cargas de trabalho. É possível ver facilmente através dos gráficos de EDP que quanto maior a carga de trabalho mais suaves são as curvas para os algoritmos `genome`, `intruder` e `labyrinth`, com picos menos elevados e vales menos profundos. Isto indica que a maior carga de trabalho distribui melhor o trabalho entre os diversos núcleos executando — mais dados na memória tendem a diminuir a chance de conflitos nos seus acessos — e distribui melhor quaisquer sobrecargas impostas pelo sistema de TM à execução. Da mesma forma, com o algoritmo `vacation`, quanto mais contenção era gerada pela carga de trabalho menos suaves eram as curvas.

Capítulo 7

Conclusões e trabalhos futuros

De meados da década de 1980, quando foi introduzido o processo de fabricação baseado em CMOS, até o início dos anos 2000, a evolução das arquiteturas de computadores era fortemente atrelada à redução do tamanho do transistor, com consequente redução do consumo de energia e aumento da frequência de operação do processador. No entanto, o processo de diminuição do tamanho dos transistores não poderia ser aplicado indefinidamente devido às limitações atômicas dos materiais envolvidos. Desta forma, por volta de 2004, verificou-se uma quebra na evolução dos processadores em relação à sua frequência de operação, terminando com o crescimento exponencial do desempenho de computadores sequenciais [66]. A maneira encontrada pelos fabricantes de processadores para superar as limitações físicas impostas pelo modelo atual de fabricação dos elementos do núcleo de processamento foi o investimento em processadores com vários núcleos e sistemas computacionais com um ou mais destes processadores nos últimos anos, inclusive em máquinas embarcadas ou consideradas de baixo poder de processamento.

No entanto, os sistemas computacionais com vários núcleos de processamento ampliaram significativamente a distância usualmente existente entre as tecnologias que os softwares sendo desenvolvidos conseguem explorar e aquilo que o hardware disponível consegue entregar. A possibilidade de se executar múltiplas *threads* simultaneamente fazendo diversos trabalhos, técnica conhecida como programação paralela, imprescindível para o melhor aproveitamento das diversas estruturas disponíveis em processadores atuais, aumenta a complexidade não apenas lógica na estruturação de programas, como também aumenta a complexidade de tarefas de análise e correção de problemas em códigos.

Apesar de o conceito de programação paralela não ser novo, com diversos modelos tendo sido estudados e propostos, diversas dificuldades relacionadas a este paradigma fizeram com que ele não se tornasse popular. Portanto, novos modelos de programação paralela têm surgido para tentar endereçar estes problemas, tornando mais fácil com que programadores explorem paralelismo nos sistemas computacionais atuais de forma mais

natural e intuitiva. Dentre estas alternativas, sistemas de Memórias Transacionais (TM) aparecem como uma sugestão promissora.

Muita pesquisa foi feita nos últimos anos em relação às TM, especificamente às STM, que foram o foco principal desta dissertação. A maior parte destas pesquisas, assim como em outras áreas da ciência da computação, é relacionada ao desempenho de processamento das TM, com pouca atenção dada a outras métricas importantes, como, por exemplo, o consumo energético e a relação deste com o desempenho de processamento.

Esta dissertação de mestrado explicou como sistemas de TM funcionam e fez uma avaliação sobre o consumo de energia, o desempenho de processamento e a relação e o balanço entre estas duas grandezas através da medida do *energy-delay product* (EDP) em uma STM configurada com diversas políticas de gerenciamento da TM e de utilização energética, sendo algumas destas combinações inéditas, estendendo estudos anteriores registrados em [12, 13].

7.1 Conclusões

Como já era de se esperar, não foi encontrada uma política de TM que fosse mais vantajosa em todos os casos ou mesmo cenários patológicos que prejudicavam a execução em todos os *benchmarks* analisados, desde que se utilizando as políticas de TM validadas como viáveis neste trabalho. Para cada aplicação com seu conjunto de dados alguns aspectos foram mais relevantes. Com uma análise mais horizontal, focada nestas características e não nas aplicações em si, é possível agrupar as aplicações com características semelhantes de forma que se tire algumas conclusões sobre quais configurações podem ou não ser mais vantajosas para determinados tipos de aplicações. Além disto, métricas diferentes levam a conclusões diferentes. Cenários que entregam um melhor *speedup* podem não ser os mesmos que entregam melhor EDP. Esta distinção deve ser levada em consideração quando se escolher a política e o número de núcleos executando a ser utilizado. Usar mais núcleos pode trazer um melhor *speedup*, mas o gasto de energia adicional associado ao aumento do número de núcleos executando pode prejudicar sobremaneira o balanço entre consumo de energia e tempo de execução a ponto de tornar esta nova configuração desfavorável para EDP. Para determinados casos, inclusive, a paralelização do algoritmo pode se mostrar não muito vantajosa. Nestes casos, a recomendação é continuar executando os algoritmos sequencialmente ou tentar alterar a forma como a paralelização do código ou do algoritmo foi implementada.

Espera-se, ao paralelizar um algoritmo, que o tempo de execução da aplicação diminua em relação à execução sequencial e que o gasto de energia se mantenha praticamente constante, em um cenário com consumo de energia constante. No entanto, como pode ser visto pelos gráficos da seção 6.3, isto nem sempre ocorre na prática. Muitas vezes,

inclusive, presencia-se comportamentos conflitantes, em que ao se aumentar o número de núcleos executando aumenta-se o consumo de energia — o que está dentro do esperado — mas diminui-se o *speedup* — ao contrário do que se esperaria nesta situação.

Este tipo de comportamento conflitante, inclusive, pode ser observado com frequência na tabela 6.2. Com exceção dos casos extremos (`kmeans-high` que é facilmente paralelizável e `intruder`, `intruder+` e `yada` que tem alta contenção), o *speedup* aumenta com o aumento do número de núcleos executando com exceção para os casos de execução com 8 núcleos para uma grande parte das políticas de TM analisadas.

Este comportamento espalhado por quase todos os *benchmarks* indica mais do que um possível desajuste entre políticas de TM estudadas e a execução com 8 núcleos. O fato de as execuções com 8 núcleos em geral apresentarem decréscimo de *speedup* indica que a infraestrutura simulada está sendo saturada com esta quantidade de núcleos executando.

Da mesma forma, pode-se fazer uma análise para o EDP similar à que foi feita para o *speedup*. No caso do EDP, espera-se que, ao paralelizar um algoritmo, novamente em um cenário de consumo de energia constante, esta grandeza diminua em relação à execução sequencial, já que, em uma situação ideal, sendo o EDP o produto entre energia gasta e tempo de execução, a energia seria praticamente constante e o tempo de execução seria dividido aproximadamente pelo número de núcleos executando. No entanto, novamente, como pode ser visto pelos gráficos da seção 6.3, isto nem sempre ocorre na prática. Muitas vezes, inclusive, presencia-se comportamentos conflitantes com o aumento do EDP ao se aumentar o número de núcleos executando, como pode ser visto na tabela 6.3.

Além do problema já comentado anteriormente sobre a limitação da plataforma simulada utilizada neste trabalho para as execuções com 8 núcleos que afetou diretamente o *speedup* nestas configurações, a piora em relação ao EDP deve-se também ao aumento considerável do consumo de energia nas execuções com 8 núcleos em relação às execuções com 4 núcleos. O aumento do número de *threads* executando simultaneamente aumenta as chances de conflito, já que os acessos à memória são mais frequentes. Este aumento de possíveis conflitos afeta o tempo de execução mas também tem um impacto considerável sobre o consumo de energia para a execução dos *benchmarks*, levando a uma piora ainda maior das medidas de EDP.

De uma forma geral, portanto, sem saber muito sobre o *benchmark* a ser executado, as chances de se obter melhor *speedup* e melhor EDP ao se utilizar um sistema de TM para controlar o acesso aos dados compartilhados em uma execução paralelizada na plataforma simulada usada neste estudo seriam maiores caso o *benchmark* fosse executado em 4 núcleos de acordo com a análise feita para o conjunto de *benchmarks* do STAMP usados neste trabalho.

Da mesma forma, em relação às políticas (tabela 6.5), é possível inferir que, para o *speedup*, `BACKOFF_LINEAR Eager` e `BACKOFF_EXPONENTIAL WAIT_ABORTER_ENABLED`

`Eager` foram as políticas que entregaram melhores resultados mais vezes. Já para o EDP, `BACKOFF_LINEAR Eager` entregou o melhor resultado mais vezes, seguida pela política `BACKOFF_LINEAR WAIT_ABORTER_ENABLED Eager`. No entanto, a vantagem da política `BACKOFF_LINEAR Eager` é demonstrada especialmente em *benchmarks* com alta contenção. Nestes cenários de alta contenção as execuções sequenciais são mais vantajosas, como pode ser visto na tabela 6.6. Esta tabela, aliás, mostra que o caso sequencial é o mais rápido em um terço das execuções e tem o melhor EDP em dois terços das execuções.

STM, portanto, mostrou-se desfavorável principalmente nos casos de aplicações com alta contenção (`intruder`, `vacation` e `yada`). Estas aplicações eventualmente teriam problemas com outros métodos de paralelização de código também, especialmente se não fosse tomado o devido cuidado na análise de onde o código deveria ser paralelizado — algo que, como foi visto, está intrinsecamente ligado à proposta de uso de TM, que deveria ser tão fácil de implementar quanto *locks* de granularidade alta. Por outro lado, para aplicações com menos contenção (`genome` e `kmeans`), o uso de TM foi vantajoso mesmo utilizando-se STM. Apesar de quase todas as políticas que demonstraram melhor desempenho serem *Eager*, para uma aplicação facilmente paralelizável como o `kmeans`, o melhor EDP ficou com uma política *Lazy*.

Além disto, considerando-se que sistemas de TM podem ter implementações em hardware e considerando que estas mesmas, na maior parte dos casos, como pode ser visto em [17], terão uma tendência de comportamento similar ao de uma STM porém com maior desempenho, pode-se utilizar os resultados obtidos para STM como forma de análise de tendências sobre o que ocorrerá quando as mesmas políticas forem implementadas em um sistema de HTM. Neste sentido, desconsiderando-se os resultados de *speedup* da tabela 6.6 para *benchmarks* com alta contenção (`intruder`, `intruder+` e `vacation-high`) e desconsiderando os resultados de EDP para *benchmarks* que indicam configurações com um ou dois núcleos como sendo melhores (`intruder`, `intruder+`, `labyrinth` e `yada`) — o que, no fim das contas, engloba os algoritmos que apresentam maior contenção e indica que paralelização não está trazendo grandes vantagens — conclui-se que as políticas mais vantajosas para *speedup* foram as que envolveram `BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager`, `WAIT_ABORTER Eager` com ou sem DVFS e `DVFS_COMMIT_BEST_BALANCE Eager`, além da política `BACKOFF_LINEAR Eager`. Já para o EDP, as políticas mais vantajosas foram aquelas que envolveram `DVFS_CONSERVATIVE Eager` e `WAIT_ABORTER Eager` com ou sem DVFS, além da política `BACKOFF_LINEAR Eager` e da execução sequencial, que se mostrou vantajosa em alguns casos. Assim, para a execução de um algoritmo na plataforma simulada estudada, sugere-se tentar a execução sequencial e a execução com STM com quatro ou oito núcleos das seguintes políticas:

- `BACKOFF_LINEAR Eager`

- `BACKOFF_EXPONENTIAL DVFS_CONSERVATIVE Eager`
- `BACKOFF_EXPONENTIAL Eager` e `BACKOFF_LINEAR Eager` combinadas com:
  - `WAIT_ABORTER_DVFS`
  - `WAIT_ABORTER_ENABLED`
  - `DVFS_COMMIT_BEST_BALANCE`

Com isto, diminui-se o espaço de busca de 97 execuções (execução sequencial mais 24 combinações de políticas descritas no capítulo 6 executadas em um, dois, quatro e oito núcleos) para apenas 17 execuções (execução sequencial mais oito combinações de políticas executadas em quatro e oito núcleos).

Caso se saiba que o algoritmo sendo executado seja facilmente paralelizado, sugere-se também a execução das políticas acima substituindo o controle de concorrência *Eager* pelo *Lazy*. Neste caso, no total, para cada nova aplicação, sugere-se a execução de um conjunto de 33 execuções ao invés das 97 sugeridas inicialmente para cada *benchmark*.

7.2 Trabalhos futuros

A principal limitação observada durante a execução deste trabalho esteve relacionada à plataforma de simulação MPARM utilizada. Esta plataforma, além de simular um processador de frequência de operação máxima de 200 MHz (baixa para os padrões atuais), possui limitações de escalabilidade de seu barramento interno que limitaram sobremaneira o desempenho das execuções em sistemas com 8 núcleos. O fato de ela não ter gerado dados fidedignos para a simulação `kmeans-low` também foi um problema.

Seria necessário, inicialmente, procurar uma nova plataforma onde não só fosse possível escalar as execuções para 8 núcleos, como também eventualmente expandir a gama de execuções para 16, 32 e, eventualmente, 64 núcleos. Isto seria possível, por exemplo, caso os *benchmarks* do STAMP fossem executados diretamente sobre processadores x86_64 ou Power, sem o uso de um simulador. Estes processadores que equipam computadores de maior porte, hoje em dia, possuem uma coleção de contadores de desempenho embutidos no processador que podem ajudar a coletar as informações necessárias sobre o tempo de execução e o consumo de energia de cada uma das operações básicas de TM nas aplicações, além de ser possível aplicar DVFS nestas arquiteturas.

A vantagem do MPARM é que ele é capaz de gerar dados sobre o consumo de energia do sistema como um todo e não apenas do processador. No caso do uso de computadores de maior porte, os contadores de desempenho dos processadores seriam capazes apenas de

obter os dados sobre o consumo de energia dos processadores. Para uma coleta mais completa de dados seria necessário a utilização de algum tipo de equipamento que monitorasse a energia consumida por toda a máquina.

Por outro lado, um outro ponto a ser atacado é o conjunto de *benchmarks* utilizado. Como foi dito na seção 2.2, seria bom fazer uma análise de cobertura do STAMP a exemplo do que foi feito para o SPEC CPU2006 em [67]. Mais ainda, seria bom revisitar os aplicativos propostos no STAMP, já que como foi visto neste trabalho eles apresentaram inúmeros problemas, como por exemplo não executarem corretamente devido a incorreções no código-fonte que precisaram ser corrigidas. Falta, ainda hoje, um bom conjunto de aplicativos escritos com o objetivo de se avaliar o uso de sistemas de TM.

Aliás, o fato de algumas arquiteturas de processadores estarem disponibilizando instruções para uso de TM em hardware talvez acelere um pouco o desenvolvimento de um bom *benchmark* nesta área. Estes processadores, já lançados comercialmente em *mainframes* e Power [31] pela IBM e para a plataforma x86_64 pela Intel [43], seriam uma boa alternativa para uso em estudos similares. Isto porque, além do uso dos contadores de desempenho já citados anteriormente nesta seção, seria possível, com estes processadores, fazer uma comparação direta entre o desempenho de uma HTM e uma STM. Isto quando eventualmente não fosse possível, através do uso de bibliotecas, definir políticas de TM sobre a implementação em hardware de TM, através da utilização de HyTM. Desta maneira seria possível comparar uma mesma política de TM totalmente controlada por software, controlada em parte por software e em parte por hardware e, eventualmente, totalmente controlada por hardware, dependendo da flexibilidade para definição de políticas providas por estas implementações de TM em processadores.

Com este tipo de comparação mais profunda entre como STM se comporta frente a sistemas de TM com implementação total ou parcial em hardware, seria possível validar de forma definitiva se as STM podem ser utilizadas para definição de tendências sobre a utilização de TM. O uso de STM, mesmo que não seja o mais eficiente na maior parte das vezes, pode ser extremamente vantajoso durante estudos de novas políticas de TM, já que o software é bem flexível e de fácil alteração, possibilitando uma análise de uma gama maior de possibilidades em um menor espaço de tempo, sem a necessidade de redesenho do hardware. Este tipo de comparação, apesar de possível com sistemas simulados como o que foi utilizado nesta dissertação, é extremamente trabalhosa e complicada. Seria possível utilizar dados de trabalhos e artigos diferentes para esta comparação, no entanto dificilmente dois trabalhos independentes utilizariam as mesmas políticas de TM em sistemas com STM e com HTM sob condições exatamente iguais de forma que fosse possível extrair uma comparação válida.

Outra vertente a ser explorada é a de políticas de TM. Uma política não implementada neste estudo mas que poderia ter potencial é a `DVFS_NON_ABORTER`. Esta política,

inspirada nas políticas de `WAIT_ABORTER` que tiveram bons resultados neste trabalho, iria, ao detectar uma transação que aborta muito, diminuir a frequência dos outros núcleos que estão executando. Assim, espera-se que os outros núcleos que estejam executando transações que, por ventura, estejam contingenciando com este núcleo que está abortando muito, demorem mais para executar e, assim, este núcleo poderá terminar sua transação com sucesso. Uma outra variação deste caso é detectar quais outros núcleos possuem transações que estejam contingenciando com a transação que aborta muito e diminuir a frequência de execução apenas destes núcleos. Outro trabalho que poderia ser feito nesta frente seria uma nova análise das políticas `DVFS_AGGRESSIVE` e `DVFS_ABORTER_ENABLED`. Como foi visto na subseção 5.1.9, muitas das execuções com estas políticas tiveram problemas. No entanto, como pode ser visto no apêndice D, todos os problemas, com exceção para as execuções do *benchmark* `intruder+`, ocorreram com combinações destas políticas com políticas *Eager*. Uma análise melhor da combinação destas políticas com as políticas *Lazy* poderia, eventualmente, mostrar algum resultado interessante. Ainda nesta vertente seria interessante explorar sistemas de TM adaptativos, que mudam a política sendo utilizada de acordo com o comportamento da aplicação em um determinado momento baseando-se em parâmetros coletados durante a execução, sem intervenções externas.

Uma outra questão que poderia ser explorada é um mecanismo de detecção de *livelock* no sistema de TM. Como foi visto, apesar de o sistema de TM ser capaz de eliminar muitos problemas típicos da paralelização de código, ele ainda pode sofrer com *livelock*. Aliás, a ocorrência de *livelock* foi um dos motivos que invalidaram políticas sugeridas durante este estudo. Prevenir a ocorrência destas situações poderia trazer ganhos de desempenho. Por exemplo, poder-se-ia explorar este detector de *livelock* para que ele serializasse a execução das *threads* conflitantes caso esta situação fosse detectada.

Um outro aspecto seria analisar o comportamento das mesmas aplicações analisadas neste trabalho quando implementando a paralelização através do uso de *locks*. Algo parecido com o que foi feito em [48], mas de acordo com as premissas deste trabalho: com comparações sendo feitas em relação a um mesmo referencial, a execução sequencial.

Outra limitação encarada durante este estudo que poderia ter uma solução explorada é a limitação no tamanho dos arquivos de saída *finegrained*. Algumas ações poderiam ser tomadas a este respeito:

- Diminuir o tamanho das cadeias de caracteres utilizadas nos arquivos *finegrained* para identificar as operações de TM. Por exemplo, ao invés de "`NONSTM`", utilizar apenas "`N`" e, ao invés de "`TXSTORE`", utilizar apenas "`S`".

- Verificar a possibilidade de alterar o simulador MPARM para que ele seja compilado em 64 bits. Isto também obrigaria o sistema operacional que executasse as simulações ser de 64 bits.

- Verificar a possibilidade de o simulador MPARM utilizar alguma outra biblioteca de entrada e saída de arquivos que não tenha problemas com o limite de arquivos em 2 GB.

- Utilizar uma nova plataforma para a execução dos *benchmarks*, como a utilização de computadores reais com processadores x86_64.

7.3 Publicações

Como mostrado na subseção 1.3.1, o trabalho desta dissertação está ligado a outros trabalhos publicados em conferências e escolas:

- *A Software Transactional Memory System for an Asymmetric Processor Architecture.*
  Felipe Goldstein, Alexandro Baldassin, Paulo Centoducatte, Rodolfo Azevedo e Leonardo A. G. Garcia.
  Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'08), pg 175–182, Outubro de 2008.
  Este artigo foi agraciado com o prêmio Júlio Salek Aude de melhor artigo entre todos os apresentados no SBAC-PAD'08.

- *Análise de desempenho e consumo de energia em Memórias Transacionais.*
  Leonardo A. G. Garcia, Alexandro Baldassin e Rodolfo Azevedo.
  Poster apresentado na III Escola Regional de Alto Desempenho de São Paulo (ERAD-SP 2012).

- *Energy-Performance Tradeoffs in Software Transactional Memory.*
  Alexandro Baldassin, João P. L. de Carvalho, Leonardo A. G. Garcia e Rodolfo Azevedo.
  Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2012).

Além disto, os resultados obtidos nesta dissertação com os resultados publicados no trabalho *Energy-Performance Tradeoffs in Software Transactional Memory* estão sendo combinados e comparados para serem publicados em uma revista. A comparação destes dois trabalhos levará a uma comparação entre duas STM: a TL2 [20], utilizada neste dissertação, e a TinySTM [24], utilizada no trabalho publicado no SBAC-PAD'2012. Esta comparação entre TL2 e TinySTM, que já foi feita em [24], será estendida com as políticas exploradas nesta dissertação e no trabalho *Energy-Performance Tradeoffs in Software Transactional Memory.*

Referências Bibliográficas

[1] Accellera systems initiative. `http://www.accellera.org/`.

[2] Fedora project. `http://fedoraproject.org/`.

[3] Green500. `http://green500.org`.

[4] Htcondor high throughput computing. `http://research.cs.wisc.edu/condor/`.

[5] Mparm. `http://www-micrel.deis.unibo.it/sitonew/research/mparm.html`.

[6] Rtems operating system. `http://www.rtems.com/`.

[7] Top500. `http://top500.org`.

[8] Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, April 2006.

[9] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[10] Sarita Vikram Adve. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, Madison, WI, USA, 1993.

[11] Vishal Aslot and Rudolf Eigenmann. Performance characteristics of the spec omp2001 benchmarks. *SIGARCH Comput. Archit. News*, 29(5):31–40, 2001.

[12] Alexandro Baldassin and Paulo Cesar Centoducatte (orientador). *Explorando Memória Transacional em Software nos Contextos de Arquiteturas Assimétricas, Jogos Computacionais e Consumo de Energia*. PhD thesis, Universidade Estadual de Campinas, Instituto de Computação, Campinas, 2009. Disponível em: `http://cutter.unicamp.br/document/?code=000768401`. Acesso em: 17 jun 2012.

[13] Alexandro Baldassin, Felipe Klein, Guido Araujo, Rodolfo Azevedo, and Paulo Centoducatte. Characterizing the energy consumption of software transactional memory. *IEEE Comput. Archit. Lett.*, 8(2):56–59, July 2009.

[14] Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, 2003.

[15] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. June 2005.

[16] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 56–69, New York, NY, USA, 2001. ACM.

[17] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[18] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. *SIGPLAN Not.*, 41(6):1–13, 2006.

[19] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, September 2008.

[20] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[21] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why stm can be more than a research toy. *Commun. ACM*, 54(4):70–77, April 2011.

[22] Robert Ennals. Efficient software transactional memory. Technical Report IRC-TR-05-051, Intel Research Cambridge Tech Report, January 2005.

[23] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, January 2006.

[24] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[25] Cesare Ferri, Amber Viescas, Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy efficient synchronization techniques for embedded architectures. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, GLSVLSI '08, pages 435–440, New York, NY, USA, 2008. ACM.

[26] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.

[27] Kourosh Gharachorloo. *Memory consistency models for shared-memory multiprocessors*. PhD thesis, Stanford, CA, USA, 1996.

[28] Felipe Goldstein, Alexandro Baldassin, Paulo Centoducatte, Rodolfo Azevedo, and Leonardo A. G. Garcia. A software transactional memory system for an asymmetric processor architecture. *Computer Architecture and High Performance Computing, Symposium on*, 0:175–182, 2008.

[29] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing*, pages 303–323. LNCS, Springer, September 2005.

[30] Rachid Guerraoui and Michał Kapałka. *Principles of Transactional Memory*. Morgan & Claypool, 2010.

[31] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *Micro, IEEE*, 32(2):48 –60, march-april 2012.

[32] Tim Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005.

[33] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

[34] Tim Harris and Keir Fraser. Revocable locks for non-blocking programming. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 72–82, New York, NY, USA, 2005. ACM.

[35] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[36] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006.

[37] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 4th edition, 2006.

[38] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[39] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[40] W. Daniel Hillis and Guy L. Steele, Jr. Update to "data parallel algorithms". *Commun. ACM*, 30(1):78–78, 1987.

[41] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[42] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8–11, 1994.

[43] Intel Corporation. *Intel® Architecture Instruction Set Extensions Programming Reference*, 319433-012a edition.

[44] Aamer Jaleel, Matthew Mattina, and Bruce Jacob. Last level cache (llc) performance of data mining workloads on a cmp — a case study of parallel bioinformatics workloads. In *Proc. 12th International Symposium on High Performance Computer Architecture (HPCA 2006)*, February 2006.

[45] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Advanced Supercomputing (NAS) Division, December 1999.

[46] Jed Scaramella John Humphreys. The impact of power and cooling on data center infrastructure. Technical Report Document #201722, IDC, 2006.

[47] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 1st edition, 2008.

[48] Felipe Klein, Alexandro Baldassin, Joao Moreira, Paulo Centoducatte, Sandro Rigo, and Rodolfo Azevedo. Stm versus lock-based systems: an energy consumption perspective. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '10, pages 431–436, New York, NY, USA, 2010. ACM.

[49] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[50] James R. Larus, Ravi Rajwar, and Tim Harris. *Transactional Memory*. Morgan & Claypool, second edition, 2010.

[51] Chunho Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 330–335, December 1997.

[52] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[53] Yossi Lev and Jan-Willem Maessen. Toward a safer interaction with transactional memory by tracking object visibility. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*. San Diego, CA, October 2005.

[54] Yossi Lev and Mark Moir. Debugging with transactional memory. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. June 2006.

[55] Mirko Loghi, Massimo Poncino, and Luca Benini. Cycle-accurate power analysis for multiprocessor systems-on-a-chip. In *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, GLSVLSI '04, pages 410–406, New York, NY, USA, 2004. ACM.

[56] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Not.*, 12(3):128–137, 1977.

[57] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. *SIGARCH Comput. Archit. News*, 34(2):53–65, 2006.

[58] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2011. Available: `http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html`.

[59] Gordon E. Moore. Cramming more components onto integrated circuits. *Electron*, 38(8):114–117, April 1965.

[60] Gordon E. Moore. Progress in digital integrated electronics. *IEEE International Electron Devices Meeting (IEDM) Digest of Technical Papers*, pages 11–13, December 1975.

[61] T. Moreshet, R.I. Bahar, and M. Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, pages 331–334, 2005.

[62] Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy-aware microprocessor synchronization: Transactional memory vs. locks.

[63] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

[64] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 182–188, October 2006.

[65] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[66] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[67] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. *SIGARCH Comput. Archit. News*, 35(2):412–423, 2007.

[68] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[69] Martino Ruggiero, Federico Angiolini, Francesco Poletti, Davide Bertozzi, Luca Benini, and Roberto Zafalon. R.zafalon. scalability analysis of evolving soc interconnect protocols. In *In Int. Symp. on Systems-on-Chip*, pages 169–172, 2004.

[70] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[71] R. Schmidt and B. D. Notohardjono. High-end server low-temperature cooling. *IBM J. Res. Dev.*, 46(6):739–751, 2002.

[72] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[73] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[74] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 169–178, New York, NY, USA, 2008. ACM.

[75] Craig Zilles and Lee Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. June 2006.

Apêndice A

Arquivos de saída da execução da simulação

Como exemplo, serão mostrados neste capítulo os arquivos gerados durante uma execução do *benchmark* `bayes` com 2 núcleos no ambiente de simulação MPARM para uma política de TM `BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager`.

No apêndice C poderão ser encontrados os dados necessários para acessar todos os arquivos gerados pelas simulações no ambiente MPARM executadas neste trabalho.

A.1 `bayes-2c.condor_output`

Este é o arquivo com a saída padrão (`stdout`) gerada pela execução do aplicativo no nó HTCondor. Seu conteúdo pode ser visto no código A.1.

Código A.1: Arquivo `bayes-2c.condor_output`.

```
[laggarcia@cajarana] ~/Unicamp/master/thesis/work/mparm_tm/simulations/simulation-1202122257-cluster
    -284/stamp/bayes/simulation-1202122257/TL2_BACKOFF_LINEAR_DVFS_COMMIT_LIGHT-eager > cat bayes-2c.
    condor_output
========== COMMAND LINE ==========
Script: /home/laggarcia/Unicamp/master/thesis/work/mparm_tm/tm/scripts/run-benchmark
Number of arguments: 3
Arguments: -F -c2 -s"bayes-2c.txt"
Argument 1: -F
Argument 2: -c2
Argument 3: -s"bayes-2c.txt"
========== CONFIGURATION ==========
 Host: node07 (143.106.24.247)
 Current directory: /home/laggarcia/Unicamp/master/thesis/work/mparm_tm/tm/stamp/bayes/simulation
     -1202122257/TL2_BACKOFF_LINEAR_DVFS_COMMIT_LIGHT-eager
 # of cores: 2
 MPARM flags: '--it=4 -f -c2 --sfile=bayes-2c.txt --sfilelight=bayes-2c-finegrained.txt'
 Profiler flags: '--binary=linked2.o --input=trace0.tra --input=trace1.tra' (tracing = no)

========== STARTING SIMULATION ==========
Start: Sun Feb 26 20:21:52 BRT 2012
Command line: 'mpsim.x --it=4 -f -c2 --sfile=bayes-2c.txt --sfilelight=bayes-2c-finegrained.txt' --cmdl
    =""
argc = 1, argv[0] =
Uploaded Program Binary: TargetMem_1.mem
```

```
Uploaded Program Binary: TargetMem_2.mem
TxOnce profiling is DISABLED by default
[BACKOFF=LIN] [DVFS=DISABLED] [DVFSCOMMIT=LIGHT] [DVFSABORTER=DISABLED] [WAITABORTER=DISABLED] TL2
    system (EAGER) ready: GV= GV4
*** Profiling enabled: TL2 FULL ***
Random seed                  = 1
Number of vars               = 16
Number of records            = 1024
Max num parents              = 2
% chance of parent           = 20
Insert penalty               = 2
Max num edge learned / var = 2
Operation quality factor     = 1.0
Generating data... done.
Generating adtree... done.
Learning structure...    Processor 1 - begin_measurement @ 1713763370.0
Processor 0 - begin_measurement @ 1713763390.0
Processor 0 - end_measurement @ 2418337027.5
        Processor 1 - end_measurement @ 2590372290.0
done.
        Processor 1 shuts down
Learn score  = -8239.640625
Actual score = -8202.7412109
TL2 system shutdown:
  GCLOCK= 0x0000018a Start= 292 Aborts= 3
  Overflows: R= 0 W= 0 L= 0
TxShutdown profiling is DISABLED by default
Processor 0 shuts down


Simulation ended


SystemC: simulation stopped by user.
End: Sun Feb 26 21:49:38 BRT 2012
========== END OF SIMULATION ==========

Profiling not performed since tracing is disabled
========== REMOVING FLAG FILE ==========
Flag removed at Sun Feb 26 21:49:38 BRT 2012
========== FLAG FILE REMOVED ==========
```

A.2 bayes-2c.condor_error

Este é o arquivo com a saída padrão de erro (`stderr`) gerada pela execução do aplicativo no nó HTCondor. Seu conteúdo pode ser visto no código A.2.

Código A.2: Arquivo `bayes-2c.condor_error`.

```
[laggarcia@cajarana] ~/Unicamp/master/thesis/work/mparm_tm/simulations/simulation-1202122257-cluster
    -284/stamp/bayes/simulation-1202122257/TL2_BACKOFF_LINEAR_DVFS_COMMIT_LIGHT-eager > cat bayes-2c.
    condor_error

         SystemC 2.0.1 --- Sep 21 2011 12:15:29
    Copyright (c) 1996-2002 by all Contributors
                ALL RIGHTS RESERVED
```

A.3 bayes-2c.txt

Este é o arquivo com um resumo de estatísticas sobre a execução do programa no ambiente simulado. Seu conteúdo pode ser visto no código A.3.

Código A.3: Arquivo `bayes-2c.txt`.

```
[laggarcia@cajarana] ~/Unicamp/master/thesis/work/mparm_tm/simulations/simulation-1202122257-cluster
    -284/stamp/bayes/simulation-1202122257/TL2_BACKOFF_LINEAR_DVFS_COMMIT_LIGHT-eager > cat bayes-2c.
    txt
Statistics files: bayes-2c.txt and bayes-2c-finegrained.txt


Simulation executed with SWARM cores on AMBA AHB (interface model) interconnection
Simulation executed with 1 buses connected by 0 bridges
Simulation executed with 2 cores (2 masters including DMAs and smart memories)
6 slaves: 2 private, 1 shared, 1 semaphores, 1 interrupt,
          0 core-associated, 0 storage, 1 frequency scaling,
          0 smart memories, 0 FFT devices
          (core-associated off, frequency scaling on, smart memories off, DRAM controller off)
DMA controllers disabled
Scratchpad memories disabled
Instruction scratchpad memories disabled
Queue memories disabled
Advanced statistics on, Access traces off, TG traces off
Simulation executed without OCP interfacing (where applicable)
Snooping for cache coherence off, Master system clock period set to 5.00 ns
VCD waveforms off
Partitioned scratchpad analysis off, /dev/pts prompt skipped
Data cache of 4096 bytes, 4-way set associative, having 0 wait states
Cache write policy: write through
Instruction cache of 8192 bytes, 4-way set associative, having 0 wait states
Simulation executed with dynamic frequency scaling
Master clock dividers set to: 1 1
Interconnect clock dividers set to: 1
PLL delays (in master system clock cycles) set to: 100 100 100
Latencies: interrupts 1, memories 1 (initial) 1 (back-to-back)
Statistics collected since benchmark request
Simulation executed with quiet mode enabled


--------------------------------------------------------------------------------


Simulation executed: Sun Feb 26 20:21:52 2012
Elapsed time - overall simulation: 87:46 minutes
Total simulated master system cycles: 535589232 (2677946162 ns)
CPU cycles simulated per second: 203414.1
Elapsed time - processor 0 critical section: 30:07 minutes
Elapsed time - processor 1 critical section: 36:16 minutes


--------------------------------------------------------------------------------


------------------------
Interconnect statistics
------------------------
Overall exec time             = 175282912 master system cycles (876414560.00 ns)
1-CPU average exec time       = 157974915 master system cycles (789874576 ns)
Concurrent exec time          = 140666919 master system cycles (703334595.00 ns)
Bus busy                      = 56229412 master system cycles (39.97% of 140666919)
Bus transferring data         = 27169305 master system cycles (19.31% of 140666919, 48.32% of 56229412)
Bus Accesses                  = 26369856 (1667876 SR, 23354303 SW, 1347677 BR, 0 BW: 3015553 R,
    23354303 W)
Time (ns) to bus access (R)   = 52819350 over 3015553 accesses (max 55, avg 17.52, min 15)
Time (ns) to bus compl. (R)   = 123405190 over 3015553 accesses (max 95, avg 40.92, min 25)
Time (ns) to bus access (W)   = 429442445 over 23354303 accesses (max 55, avg 18.39, min 15)
Time (ns) to bus compl. (W)   = 662964520 over 23354303 accesses (max 65, avg 28.39, min 25)
Time (ns) to bus access (SR)  = 29226455 over 1667876 accesses (max 55, avg 17.52, min 15)
Time (ns) to bus compl. (SR)  = 45905215 over 1667876 accesses (max 65, avg 27.52, min 25)
Time (ns) to bus access (SW)  = 429442445 over 23354303 accesses (max 55, avg 18.39, min 15)
Time (ns) to bus compl. (SW)  = 662964520 over 23354303 accesses (max 65, avg 28.39, min 25)
Time (ns) to bus access (BR)  = 23592895 over 1347677 accesses (max 55, avg 17.51, min 15)
Time (ns) to bus compl. (BR)  = 77499975 over 1347677 accesses (max 95, avg 57.51, min 55)


--------------------------------------------------------------------------------


----------------
SWARM Processor 0
----------------
```

```
Direct Accesses                = 0 to DMA
Bus Accesses                   = 11753513 (731752 SR, 10420272 SW, 601489 BR, 0 BW: 1333241 R, 10420272
    W)
Time (ns) to bus access (R)    = 23629230 over 1333241 accesses (max 55, avg 17.72, min 15)
Time (ns) to bus compl. (R)    = 55006310 over 1333241 accesses (max 95, avg 41.26, min 25)
Time (ns) to bus access (W)    = 193913995 over 10420272 accesses (max 55, avg 18.61, min 15)
Time (ns) to bus compl. (W)    = 298109910 over 10420272 accesses (max 65, avg 28.61, min 25)
Time (ns) to bus access (SR)   = 13024120 over 731752 accesses (max 55, avg 17.80, min 15)
Time (ns) to bus compl. (SR)   = 20341640 over 731752 accesses (max 65, avg 27.80, min 25)
Time (ns) to bus access (BR)   = 10605110 over 601489 accesses (max 55, avg 17.63, min 15)
Time (ns) to bus compl. (BR)   = 34664670 over 601489 accesses (max 95, avg 57.63, min 55)
Time (ns) to bus access (SW)   = 193913995 over 10420272 accesses (max 55, avg 18.61, min 15)
Time (ns) to bus compl. (SW)   = 298109910 over 10420272 accesses (max 65, avg 28.61, min 25)
Time (ns) to bus access (tot)  = 217543225 over 11753513 accesses (max 55, avg 18.51, min 15)
Time (ns) to bus compl. (tot)  = 353116220 over 11753513 accesses (max 95, avg 30.04, min 25)
Wrapper overhead cycles        = 23507026
Total bus activity cycles      = 376623246 (bus completion + wrapper OH)
"Free" bus accesses            = 0 (0.00% of 11753513)
Idle cycles                    = 0
```

	Current setup	
	Ext Acc	Cycles
Private reads	601489*	74497168
Bus+Wrapper waits		14879559
Private writes	10203082	10203082
Bus+Wrapper waits		17035515
Shared reads	693660	1387320
Bus+Wrapper waits		13557467
Shared writes	199207	199207
Bus+Wrapper waits		199207
Semaphore reads	38092	76184
Bus+Wrapper waits		614433
Semaphore writes	26468	26468
Bus+Wrapper waits		26468
Interrupt writes	0	0
Bus+Wrapper waits		0
Frequency reads	0	0
Bus+Wrapper waits		0
Frequency writes	236	236
Bus+Wrapper waits		236
Internal reads		480
Internal writes		4341
SWARM total	11762234	86394486
Wait cycles total		46312885
Pipeline stalls		7879360
Overall total	11762234	140586731

```
---Cache performance---
* Read bursts due to 601489 cache misses out of 70888234 cacheable reads. Misses
also cost 3608934 int cycles to refill. All writes were write-through.
Reads are done by reading tag and data in parallel (so data reads happen
even on cache misses); write-throughs always involve a tag read followed,
only in case of hit, by a data word write.
D-Cache: 12761711 read hits; 1282 read misses (5128 single-word refills)
D-Cache: 10035190 write-through hits; 167892 write-through misses
D-Cache total: 22966075 tag reads, 1282 tag writes
               12762993 data reads, 1282 data line writes, 10035190 data word writes
D-Cache Miss Rate: 0.01%
I-Cache: 58126523 read hits; 600207 read misses (2400828 single-word refills)
I-Cache: 0 write-through hits; 0 write-through misses
I-Cache total: 58726730 tag reads, 600207 tag writes
               58726730 data reads, 600207 data line writes, 0 data word writes
I-Cache Miss Rate: 1.04%
```

```
-----------------
SWARM Processor 1
-----------------
Direct Accesses                 = 0 to DMA
Bus Accesses                    = 14616343 (936124 SR, 12934031 SW, 746188 BR, 0 BW: 1682312 R, 12934031
    W)
Time (ns) to bus access (R)     = 29190120 over 1682312 accesses (max 55, avg 17.35, min 15)
Time (ns) to bus compl. (R)     = 68398880 over 1682312 accesses (max 95, avg 40.66, min 25)
Time (ns) to bus access (W)     = 235528450 over 12934031 accesses (max 55, avg 18.21, min 15)
Time (ns) to bus compl. (W)     = 364854610 over 12934031 accesses (max 65, avg 28.21, min 25)
Time (ns) to bus access (SR)    = 16202335 over 936124 accesses (max 55, avg 17.31, min 15)
Time (ns) to bus compl. (SR)    = 25563575 over 936124 accesses (max 65, avg 27.31, min 25)
Time (ns) to bus access (BR)    = 12987785 over 746188 accesses (max 55, avg 17.41, min 15)
Time (ns) to bus compl. (BR)    = 42835305 over 746188 accesses (max 95, avg 57.41, min 55)
Time (ns) to bus access (SW)    = 235528450 over 12934031 accesses (max 55, avg 18.21, min 15)
Time (ns) to bus compl. (SW)    = 364854610 over 12934031 accesses (max 65, avg 28.21, min 25)
Time (ns) to bus access (tot)   = 264718570 over 14616343 accesses (max 55, avg 18.11, min 15)
Time (ns) to bus compl. (tot)   = 433253490 over 14616343 accesses (max 95, avg 29.64, min 25)
Wrapper overhead cycles         = 29232686
Total bus activity cycles       = 462486176 (bus completion + wrapper OH)
"Free" bus accesses             = 0 (0.00% of 14616343)
Idle cycles                     = 0
```

	Current setup	
	Ext Acc	Cycles
Private reads	746188*	92570030
Bus+Wrapper waits		18416390
Private writes	12653465	12653465
Bus+Wrapper waits		20922789
Shared reads	890994	1781988
Bus+Wrapper waits		17289974
Shared writes	258854	258854
Bus+Wrapper waits		258854
Semaphore reads	45130	90260
Bus+Wrapper waits		726205
Semaphore writes	33847	33847
Bus+Wrapper waits		33847
Interrupt writes	0	0
Bus+Wrapper waits		0
Frequency reads	0	0
Bus+Wrapper waits		0
Frequency writes	342	342
Bus+Wrapper waits		342
Internal reads		692
Internal writes		6133
SWARM total	14628820	107395611
Wait cycles total		57648401
Pipeline stalls		9813903
Overall total	14628820	174857915

```
---Cache performance---
* Read bursts due to 746188 cache misses out of 88092902 cacheable reads. Misses
also cost 4477128 int cycles to refill. All writes were write-through.
Reads are done by reading tag and data in parallel (so data reads happen
even on cache misses); write-throughs always involve a tag read followed,
only in case of hit, by a data word write.
D-Cache: 15864474 read hits; 1439 read misses (5756 single-word refills)
D-Cache: 12485109 write-through hits; 168356 write-through misses
D-Cache total: 28519378 tag reads, 1439 tag writes
               15865913 data reads, 1439 data line writes, 12485109 data word writes
D-Cache Miss Rate: 0.01%
I-Cache: 72228428 read hits; 744749 read misses (2978996 single-word refills)
I-Cache: 0 write-through hits; 0 write-through misses
I-Cache total: 72973177 tag reads, 744749 tag writes
               72973177 data reads, 744749 data line writes, 0 data word writes
I-Cache Miss Rate: 1.04%
```

```
--------------------------------------------------------------------------------


================================================================================
----------------
Power estimation
----------------

   Energy spent:
ARM 0:
    core:      6713387268.74 [pJ]
    icache:    8499371645.09 [pJ]
    dcache:    1734947080.47 [pJ]
    scratch:          0.00 [pJ]
    i-scratch:        0.00 [pJ]
ARM 1:
    core:      8347825269.03 [pJ]
    icache:    10561022079.91 [pJ]
    dcache:    2156259242.36 [pJ]
    scratch:          0.00 [pJ]
    i-scratch:        0.00 [pJ]
RAM 00:        2654692389.74 [pJ]
RAM 01:        3287037176.05 [pJ]
RAM 02:        550971259.94 [pJ]
RAM 03:           0.00 [pJ]
RAM 04:           0.00 [pJ]
RAM 05:           0.00 [pJ]
Bus 0:
    typ:       682729490.56 [pJ]
    max:              0.00 [pJ]
    min:       421526375.65 [pJ]
--------------------------------------
Partial sums:
   ARM cores:15061212537.77 [pJ]
    icaches:  19060393724.99 [pJ]
    dcaches:  3891206322.84 [pJ]
    scratches:        0.00 [pJ]
    i-scratches:      0.00 [pJ]
RAMs:          6492700825.73 [pJ]
DMAs:             0.00 [pJ]
Bridges:          0.00 [pJ]
Buses:
    typ:       682729490.56 [pJ]
    max:              0.00 [pJ]
    min:       421526375.65 [pJ]
--------------------------------------
--------------------------------------
Total:         45188242901.88 [pJ] typ
Total:         44505513411.32 [pJ] max
Total:         44927039786.97 [pJ] min
--------------------------------------------------------------------------------

   Power spent:
ARM 0:
    core:             9.54 [mW]
    icache:          12.07 [mW]
    dcache:           2.46 [mW]
    scratch:          0.00 [mW]
    i-scratch:        0.00 [mW]
ARM 1:
    core:             9.53 [mW]
    icache:          15.00 [mW]
    dcache:           3.06 [mW]
    scratch:          0.00 [mW]
    i-scratch:        0.00 [mW]
RAM 00:               3.04 [mW]
RAM 01:               3.76 [mW]
RAM 02:               0.63 [mW]
RAM 03:               0.00 [mW]
RAM 04:               0.00 [mW]
RAM 05:               0.00 [mW]
Bus 0:
    typ:              0.78 [mW]
    max:              0.00 [mW]
    min:              0.48 [mW]
================================================================================
RAM 0:  2403919 [reads]  10186140 [writes] 0 [stalls] 0 [noops]
RAM 1:  2978223 [reads]  12611354 [writes] 0 [stalls] 0 [noops]
```

```
RAM 2:  3019909 [reads]  456531 [writes] 0 [stalls] 0 [noops]
RAM 3:  83076 [reads]  59609 [writes] 0 [stalls] 0 [noops]
RAM 4:  0 [reads]  0 [writes] 0 [stalls] 0 [noops]
RAM 5:  0 [reads]  0 [writes] 0 [stalls] 0 [noops]
```

```
Cache Accesses:  tag_R tagW dataR datalW datawW dirtyR dirtyW bitR bitW
Instruction cache
CACHE 0  -  58726730 600207 58726730 600207 0 58726730 600207 0 0
CACHE 1  -  72973177 744749 72973177 744749 0 72973177 744749 0 0
Data cache
CACHE 0  -  22966075 1282 12762993 1282 10035190 22966075 1282 0 0
CACHE 1  -  28519378 1439 15865913 1439 12485109 28519378 1439 0 0
```

A.4 `bayes-2c-finegrained.txt`

Este é o arquivo com uma estatística detalhada sobre a execução do programa.

Os 12 campos de dados contidos nestes arquivos estão descritos no código A.4.

Código A.4: Campos do arquivo *finegrained*.

```
#
# Input file's expected format (12 fields separated by tab):
#
#  FIELD # : DESCRIPTION
#
#  0: <tag>
#  1: <proc>
#  2: <start time>
#  3: <start time + total crit time>
#  4: <accesses>
#  5: <total wait time>
#  6: <core energy>
#  7: <i$ energy>
#  8: <d$ energy>
#  9: <prv mem energy>
#  10: <shr mem energy>
#  11: <buses energy>
```

Por uma questão de espaço, apenas uma parte inicial e final deste arquivo são apresentadas no código A.5. Estes arquivos podem facilmente chegar a tamanhos que excedem 1 GB.

Código A.5: Parte inicial e final do arquivo `bayes-2c-finegrained.txt`.

```
[laggarcia@cajarana] ~ > head bayes-2c-finegrained.txt
NONSTM  0       1713763530.0    1713765745.0    31      660     18007.14        6233.83 1383.57 9710.30    0.00 1666.60
NONSTM  1       1713763510.0    1713765635.0    31      570     17353.74        6233.83 1085.77 10303.66          0.00 1666.60
TXNEWTHREAD     0       1713766545.0    1713768785.0    31      660     18263.45        6519.40 1480.90 9710.30    0.00 1701.03
TXNEWTHREAD     1       1713766595.0    1713768745.0    31      570     17610.04        6519.40 1183.11 10303.66          0.00 1701.03
NONSTM  0       1713766825.0    1713770415.0    46      970     29579.24        11896.91        2815.73 15300.95          0.00 2730.02
NONSTM  1       1713766875.0    1713770800.0    49      930     32347.96        11896.91        3089.65 15875.71          0.00 2899.77
TXINITTHREAD    0       1713768425.0    1716666895.0    82263   1645640 25483766.04     22109334.29     1968592.63      18543521.28
    22470.87        2409658.81
TXINITTHREAD    1       1713768900.0    1716668440.0    82279   1644860 25490543.19     22077343.53     1971522.60      18586403.94
    24566.23        2410385.54
NONSTM  0       1716663555.0    1723989175.0    161132  3110550 69141057.73     83933528.77     14822781.43     37225477.42     373141.70
        5909810.48
TXSTART 0       1721090945.0    1728420930.0    161188  3111620 69176936.71     83946662.87     14826239.49     37245240.97     373290.04
        5913250.44
[laggarcia@cajarana] ~ > tail bayes-2c-finegrained.txt
TXSTART 1       2590288845.0    3465983455.0    14615403        264703235       8347171239.32   10560571947.81  2156192830.70
    3286890585.24   550793729.45    682669835.00
NONSTM  1       2590290365.0    3465986360.0    14615425        264703585       8347183108.41   10560578753.16  2156194777.58
    3286897387.41   550795361.19    682670893.03
TXSTLOCAL       1       2590291855.0    3465988050.0    14615429        264703650       8347185102.73   10560581180.47  2156195531.50
    3286898289.85   550795509.53    682671047.16
NONSTM  1       2590292155.0    3465988980.0    14615441        264703850       8347190779.87   10560585225.86  2156197332.34
    3286901364.92   550796251.23    682671529.87
TXLOAD  1       2590292905.0    3465990275.0    14615446        264703930       8347195652.91   10560589937.70  2156198062.40
    3286901816.14   550797437.95    682671937.93
NONSTM  1       2590293555.0    3465991655.0    14615453        264704050       8347202411.37   10560595601.17  2156200544.58
    3286904130.91   550798327.99    682672482.45
TXCOMMIT        1       2590294390.0    3465995740.0    14615490        264704665       8347208815.30   10560611164.53  2156204753.19
    3286911124.82   550803081.07    682674910.88
NONSTM  1       2590297745.0    3466045115.0    14616060        264713960       8347600052.13   10560874067.51  2156240331.40
    3286990386.49   550913496.55    682709587.67
TXFREETHREAD    1       2590344305.0    3466117075.0    14616336        264718450       8347815518.55   10561019652.98  2156256434.05
    3287033022.53   550969924.88    682728635.72
NONSTM  1       2590369805.0    3466143730.0    14616343        264718570       8347825269.03   10561022079.91  2156259242.36
    3287037176.05   550971259.94    682729490.56
```

Apêndice B

Arquivos de saída do processamento dos arquivos *finegrained*

Como exemplo, serão detalhados neste capítulo os arquivos gerados pelo processamento do arquivo *finegrained* gerado durante uma execução do *benchmark* `bayes` com 2 núcleos no ambiente de simulação MPARM para uma política de TM `BACKOFF_LINEAR DVFS_COMMIT_LIGHT Eager`.

No apêndice C poderão ser encontrados os dados necessários para acessar todos os arquivos gerados pelo processamento dos arquivos *finegrained* gerados pelas simulações executadas no ambiente MPARM neste trabalho.

B.1 `bayes-2c-operations-output.txt`

Este é o arquivo com a saída padrão (`stdout`) da execução do *script* de processamento do arquivo *finegrained*. Este arquivo contem um resumo das operações de TM executadas e quanto de energia foi gasto por tipo de operação, além de outras informações bem resumidas sobre as operações executadas durante a simulação. Seu conteúdo pode ser visto no código B.1.

Código B.1: Arquivo `bayes-2c-operations-output.txt`.

```
[laggarcia@cajarana] ~/Unicamp/master/thesis/work/mparm_tm/simulations/simulation-1202122257-cluster
    -284/stamp/bayes/simulation-1202122257/TL2_BACKOFF_LINEAR_DVFS_COMMIT_LIGHT-eager > cat bayes-2c-
    operations-output.txt
==========================================================
>>>>>>>>>>>>      ATTENTION      <<<<<<<<<<<<
THIS VERSION OF THE SCRIPT IS INTENTED TO BE
USED ONLY AS A WORKAROUND FOR THE ISSUE RELATED
TO SIMULATIONS GENERATING LOG FILES EXCEEDING
THE 2GB FILE LIMIT SIZE
==========================================================
Processing /home/laggarcia/Unicamp/master/thesis/work/mparm_tm/simulations/simulation-1202122257-
    cluster-284/stamp/bayes/simulation-1202122257/TL2_BACKOFF_LINEAR_DVFS_COMMIT_LIGHT-eager//bayes-2c
    -finegrained.txt.bz2 ...
```

```
Creating /home/laggarcia/Unicamp/master/thesis/work/mparm_tm/simulations/simulation-1202122257-cluster
    -284/stamp/bayes/simulation-1202122257/TL2_BACKOFF_LINEAR_DVFS_COMMIT_LIGHT-eager//bayes-2c-
    operations.txt ...
Creating/updating file results-globals.dat ...
Creating/updating file results-primitives.dat ...
========================== SUMMARY ==========================
         TXFREE : count=        123 | ene_per_oper=          24.14 | cyc_per_oper=      259.4 | ene_total
              =          2969.02 [ 0.0%] | cyc_total=       31908 [ 0.0%]
            CAS : count=        572 | ene_per_oper=           9.92 | cyc_per_oper=       94.6 | ene_total
              =          5672.51 [ 0.0%] | cyc_total=       54115 [ 0.0%]
       ROLLBACK : count=          3 | ene_per_oper=     1155182.51 | cyc_per_oper= 8778652.3 | ene_total
              =       3465547.53 [ 7.7%] | cyc_total=    26335957 [ 8.3%]
         TXLOAD : count=       2330 | ene_per_oper=          13.90 | cyc_per_oper=      131.9 | ene_total
              =         32389.27 [ 0.1%] | cyc_total=      307264 [ 0.1%]
        TXSTART : count=        289 | ene_per_oper=          47.74 | cyc_per_oper=      482.6 | ene_total
              =         13797.26 [ 0.0%] | cyc_total=      139458 [ 0.0%]
        TXSTORE : count=       1324 | ene_per_oper=          19.43 | cyc_per_oper=      173.8 | ene_total
              =         25731.19 [ 0.1%] | cyc_total=      230177 [ 0.1%]
         NONSTM : count=       4577 | ene_per_oper=        9023.01 | cyc_per_oper=    62342.6 | ene_total
              =      41298308.27 [91.4%] | cyc_total=   285342002 [90.3%]
      TXSTLOCAL : count=        633 | ene_per_oper=           9.05 | cyc_per_oper=       75.8 | ene_total
              =          5730.85 [ 0.0%] | cyc_total=       47970 [ 0.0%]
     TXNEWTHREAD : count=          4 | ene_per_oper=           0.66 | cyc_per_oper=        5.0 | ene_total
              =             2.63 [ 0.0%] | cyc_total=          20 [ 0.0%]
    TXINITTHREAD : count=          4 | ene_per_oper=       68640.99 | cyc_per_oper=   573939.8 | ene_total
              =        274563.96 [ 0.6%] | cyc_total=     2295759 [ 0.7%]
      TXCOMMIT : count=        289 | ene_per_oper=         169.06 | cyc_per_oper=     3489.3 | ene_total
              =         48858.94 [ 0.1%] | cyc_total=     1008417 [ 0.3%]
       TXALLOC : count=        147 | ene_per_oper=          89.08 | cyc_per_oper=      953.8 | ene_total
              =         13094.87 [ 0.0%] | cyc_total=      140210 [ 0.0%]
    TXFREETHREAD : count=          4 | ene_per_oper=         394.15 | cyc_per_oper=     4134.2 | ene_total
              =          1576.60 [ 0.0%] | cyc_total=       16537 [ 0.0%]

num_cores         =              2
total_energy      =     45188242.90 nJ
total_power       =           51.60 mW
energy_per_core   =     22594121.45 nJ
power_per_core    =           28.60 mW
stm_ene_overhead  =            8.6%
rollback_energy   =      3465547.53 nJ [ 7.7%]
max_cycles        =       175154765
cycles_per_core   =     157974897.0
rollback_cycles   =        26335957 [ 8.3%] (might be greater than the maximum cycles)

started_trans     =             292
aborted_trans     =               3 [abort_rate= 1.0%]
committed_trans   =             289
loads             =            2330
stores            =            1324
storelocals       =             633
=============================================================
NOTE: clock_period=5ns and energy_unit=nJ
```

B.2 `bayes-2c-operations.txt`

Este é o arquivo que registra um resumo das informações mais importantes do arquivo *finegrained*. A descrição dos campos contidos neste arquivo se encontra na primeira linha do arquivo. Apesar de este arquivo ser um resumo das informações mais importantes contidas nos arquivos *finegrained*, eles também podem ter tamanhos consideráveis. O início deste arquivo pode ser visto no código B.2

Código B.2: Parte inicial do arquivo `bayes-2c-operations.txt`.

```
[laggarcia@cajarana] ~/Unicamp/master/thesis/work/mparm_tm/simulations/simulation-1202122257-cluster
    -284/stamp/bayes/simulation-1202122257/TL2_BACKOFF_LINEAR_DVFS_COMMIT_LIGHT-eager > head bayes-2c-
    operations.txt
```

```
Tag       Procid  Start_time       Cycles  Energy
NONSTM    0       342752706        443     37.00
NONSTM    1       342752702        425     34.98
TXNEWTHREAD       0       342753309       5       0.67
TXNEWTHREAD       1       342753319       5       0.64
NONSTM    0       342753365        270     24.65
NONSTM    1       342753375        355     27.76
TXINITTHREAD      0       342753685       578976  70474.85
TXINITTHREAD      1       342753780       579123  68065.43
NONSTM    0       343332711        885430  140865.63
```

B.3 `results-globals.dat`

Este arquivo contem, para cada um dos *benchmarks* em cada uma das configurações de TM e de número de núcleos utilizados durante a execução do aplicativo, os dados totalizados sobre a execução relativos ao consumo de energia e tempo gasto na execução. A descrição dos campos contidos neste arquivo se encontra na primeira linha do arquivo. O início deste arquivo pode ser visto no código B.3.

Código B.3: Parte inicial do arquivo `results-globals.dat`.

```
[laggarcia@cajarana] ~/Unicamp/master/thesis/work/mparm_tm/simulations/simulation-1202122257-cluster
    -284 > head results-globals.dat
Benchmark       Label   Num_cores       Total_energy    Total_power     Energy_per_core Power_per_core
      STM_energy_overhead       Max_cycles      Cycles_per_core Rollback_energy Rollback_cycles
    Started_trans   Aborted_trans   Abort_rate      Committed       Loads   Stores  Stlocals
genome  SEQUENTIAL      1       4495635.20      27.96   4495635.20      27.96   0.0     32155223
          32155223.0      0.00    0       0       0       0.0     0       0       0       0
genome+ SEQUENTIAL      1       14066094.39     27.30   14066094.39     27.30   0.0     103050500
          103050500.0     0.00    0       0       0       0.0     0       0       0       0
intruder        SEQUENTIAL      1       4496780.26      27.33   4496780.26      27.33   0.0
    32903404        32903404.0      0.00    0       0       0       0.0     0       0       0       0
intruder+       SEQUENTIAL      1       20113469.35     27.23   20113469.35     27.23   0.0
    147754632       147754632.0     0.00    0       0       0       0.0     0       0       0       0
yada    SEQUENTIAL      1       353370083.22    32.70   353370083.22    32.70   0.0     2161586258
          2161586258.0    0.00    0       0       0       0.0     0       0       0       0
labyrinth       SEQUENTIAL      1       24590965.26     29.50   24590965.26     29.50   0.0
    166706200       166706200.0     0.00    0       0       0       0.0     0       0       0       0
labyrinth+      SEQUENTIAL      1       52510680.85     29.65   52510680.85     29.65   0.0
    354143979       354143979.0     0.00    0       0       0       0.0     0       0       0       0
ssca2   SEQUENTIAL      1       3590846.20      25.54   3590846.20      25.54   0.0     28120165
          28120165.0      0.00    0       0       0       0.0     0       0       0       0
vacation-low    SEQUENTIAL      1       1386399.59      32.21   1386399.59      32.21   0.0     8608426
     8608426.0       0.00    0       0       0       0.0     0       0       0       0
```

B.4 `results-primitives.dat`

Este arquivo contem, para cada um dos *benchmarks* em cada uma das configurações de TM e de número de núcleos utilizados durante a execução do aplicativo, os dados totalizados relativos ao consumo de energia e tempo gasto na execução de cada operação primitiva de TM. A descrição dos campos contidos neste arquivo se encontra na primeira linha do arquivo. O início deste arquivo pode ser visto no código B.4.

Código B.4: Parte inicial do arquivo `results-primitives.dat`.

```
[laggarcia@cajarana] ~/Unicamp/master/thesis/work/mparm_tm/simulations/simulation-1202122257-cluster
    -284 > head results-primitives.dat
Benchmark       Label   Num_cores       Tag     Count   Ene_per_oper    Cyc_per_oper    Power_per_oper
      Energy  Cycles
genome  SEQUENTIAL      1       ROLLBACK        0       0.00    0.00    0.00    0.00    0
genome  SEQUENTIAL      1       NONSTM  1       4495635.20      32155223.00     27.96   4495635.20
         32155223
genome+ SEQUENTIAL      1       ROLLBACK        0       0.00    0.00    0.00    0.00    0
genome+ SEQUENTIAL      1       NONSTM  1       14066094.39     103050500.00    27.30   14066094.39
         103050500
intruder        SEQUENTIAL      1       ROLLBACK        0       0.00    0.00    0.00    0.00    0
intruder        SEQUENTIAL      1       NONSTM  1       4496780.26      32903404.00     27.33
    4496780.26      32903404
intruder+       SEQUENTIAL      1       ROLLBACK        0       0.00    0.00    0.00    0.00    0
intruder+       SEQUENTIAL      1       NONSTM  1       20113469.35     147754632.00    27.23
    20113469.35     147754632
yada    SEQUENTIAL      1       ROLLBACK        0       0.00    0.00    0.00    0.00    0
```

Apêndice C

Código-fonte e arquivos gerados nas simulações e seus processamentos

Todo o código-fonte utilizado neste trabalho, assim como os arquivos gerados pelas simulações sobre o ambiente MPARM e os arquivos gerados pelo processamento dos arquivos *finegrained*, conforme descrito no capítulo 5, podem ser encontrados na página pessoal do autor deste trabalho no *site* do LSC em `http://lampiao.lsc.ic.unicamp.br/~laggarcia/`.

O único código-fonte que não foi disponibilizado é o código-fonte do MPARM, já que sua licença não permite esta distribuição. No entanto, versões mais novas do MPARM, assim como a implementação de ARM que o acompanha, chamada de Software ARM (SWARM), possuem licença GNU Public License (GPL) e podem ter seu código-fonte encontrado em [5].

Já o código-fonte do SystemC 2.0.1 está sob a licença *SystemC Open Source License 2.3*.

O STAMP e a TL2, por sua vez, estão sob licença *BSD License*.

Apêndice D

Arquivos *finegrained* com 2 GB na Simulação 1102121722 (267)

Neste apêndice estão listadas todas as simulações cujo arquivo *finegrained* atingiu o tamanho máximo de 2 GB na simulação 1102121722 (267) e, portanto, teve seu conteúdo sobrescrito. As informações sobre estas simulações podem ser encontradas na tabela D.1.

Tabela D.1: Simulações cujos arquivos *finegrained* atingiram o limite suportado de 2 GB.

Benchmark	**Número de núcleos simulados**	**Política de TM**
genome	4	BACKOFF_LINEAR DVFS_AGGRESSIVE Eager
genome	8	BACKOFF_LINEAR DVFS_AGGRESSIVE Eager
genome+	4	BACKOFF_LINEAR DVFS_AGGRESSIVE Eager
genome+	8	BACKOFF_LINEAR DVFS_AGGRESSIVE Eager
labyrinth+	8	BACKOFF_LINEAR DVFS_AGGRESSIVE Eager
vacation-low	2	BACKOFF_LINEAR DVFS_AGGRESSIVE Eager
vacation-low	4	BACKOFF_LINEAR DVFS_AGGRESSIVE Eager
vacation-low	8	BACKOFF_LINEAR DVFS_AGGRESSIVE Eager
vacation-high	8	BACKOFF_LINEAR DVFS_AGGRESSIVE Eager
genome	4	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
genome	8	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
genome+	4	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
genome+	8	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
yada	8	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
labyrinth	8	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
labyrinth+	8	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager

continua na próxima página...

...continuação da página anterior

Benchmark	**Número de núcleos simulados**	**Política de TM**
vacation-low	2	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
vacation-low	4	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
vacation-low	8	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
vacation-high	4	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
vacation-high	8	BACKOFF_EXPONENTIAL DVFS_AGGRESSIVE Eager
genome	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
genome	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
genome	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
genome	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
genome+	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
genome+	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
genome+	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
genome+	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
intruder	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
intruder	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
intruder	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
intruder	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
intruder+	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
intruder+	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
intruder+	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
intruder+	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
yada	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
yada	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
yada	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
yada	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
labyrinth	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
labyrinth	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
labyrinth	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
labyrinth	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
labyrinth+	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
labyrinth+	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
labyrinth+	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
labyrinth+	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
ssca2	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
ssca2	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
ssca2	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
ssca2	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
vacation-low	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
vacation-low	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager

continua na próxima página...

...continuação da página anterior

Benchmark	Número de núcleos simulados	Política de TM
vacation-low	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
vacation-low	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
vacation-high	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
vacation-high	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
vacation-high	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
vacation-high	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
bayes	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
bayes	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
bayes	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
bayes	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
kmeans-low	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
kmeans-low	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
kmeans-low	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
kmeans-low	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
kmeans-high	1	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
kmeans-high	2	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
kmeans-high	4	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
kmeans-high	8	BACKOFF_LINEAR DVFS_ABORTER_ENABLED Eager
genome	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
genome	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
genome	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
genome	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
genome+	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
genome+	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
genome+	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
genome+	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
intruder	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
intruder	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
intruder	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
intruder	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
intruder+	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
intruder+	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
intruder+	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
intruder+	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
yada	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
yada	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
yada	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
yada	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
labyrinth	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager

continua na próxima página...

...continuação da página anterior

Benchmark	**Número de núcleos simulados**	**Política de TM**
labyrinth	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
labyrinth	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
labyrinth	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
labyrinth+	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
labyrinth+	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
labyrinth+	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
labyrinth+	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
ssca2	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
ssca2	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
ssca2	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
ssca2	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
vacation-low	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
vacation-low	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
vacation-low	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
vacation-low	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
vacation-high	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
vacation-high	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
vacation-high	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
vacation-high	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
bayes	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
bayes	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
bayes	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
bayes	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
kmeans-low	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
kmeans-low	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
kmeans-low	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
kmeans-low	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
kmeans-high	1	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
kmeans-high	2	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
kmeans-high	4	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
kmeans-high	8	BACKOFF_EXPONENTIAL DVFS_ABORTER_ENABLED Eager
intruder+	8	BACKOFF_LINEAR DVFS_COMMIT_BEST_BALANCE Lazy
intruder+	8	BACKOFF_LINEAR DVFS_COMMIT_LIGHT Lazy