UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JANAÍNA SCHWARZROCK

Adaptação dinâmica do número de threads em aplicações paralelas OpenMP para otimizar EDP em sistemas embarcados

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. Edison Pignaton de Freitas
Co-orientador: Prof. Dr. Antonio Carlos Schneider Beck Filho

Porto Alegre
2018

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Schwarzrock, Janaína

Adaptação dinâmica do número de threads em aplicações paralelas OpenMP para otimizar EDP em sistemas embarcados / Janaína Schwarzrock. – Porto Alegre: PPGC da UFRGS, 2018.

95 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Orientador: Edison Pignaton de Freitas; Coorientador: Antonio Carlos Schneider Beck Filho.

1. Adaptação dinâmica. 2. Otimização de EDP. 3. Aplicações paralelas OpenMP. 4. Sistemas embarcados. I. Freitas, Edison Pignaton de. II. Beck Filho, Antonio Carlos Schneider. III. Adaptação dinâmica do número de threads em aplicações paralelas OpenMP para otimizar EDP em sistemas embarcados.

Aos meus pais,
à minha amada irmã e
à minha pequena Doce de Leite.

AGRADECIMENTOS

Agradeço primeiramente aos Professores Edison Pignaton de Freitas e Antonio Carlos Schneider Beck Filho pela orientação no desenvolvimento deste trabalho. Agradeço ao Professor Philippe Olivier Alexandre Navaux pela oportunidade de participar do projeto HPC4E (com financiamento da União Européia/H2020 e RNP).

Agradeço aos meus pais pelo apoio. Agradeço à minha mãe por cuidar da minha pequena com tanto carinho e dedicação durante este período. Agradeço à minha irmã por conversar comigo nos momentos difíceis e pelas suas visitas que me fizeram aproveitar um pouquinho de Porto Alegre. Não posso deixar de agradecer à Rosa Maria, que me acolheu em sua casa e não me deixou desistir do mestrado.

Agradeço também aos meus colegas de laboratório, mesmo que eles tenham me deixado passar frio devido ao ar condicionado (mas eles são caras legais). Agradeço ao Maik Basso e ao Tiago Giacomelli Alves, do departamento de elétrica da UFRGS, que me ajudaram no desenvolvimento do circuito utilizado para medir o consumo de energia dos experimentos realizados neste trabalho.

RESUMO

Aplicações paralelas geralmente são executadas com o máximo número de *threads* de hardware disponíveis no sistema para maximizar o seu desempenho. Contudo, esta abordagem pode não ser a melhor escolha quando se busca eficiência energética e, em alguns casos, pode até mesmo degradar o desempenho. Desta maneira, o presente trabalho aplica a adaptação dinâmica do número de *threads* para otimizar o *Energy-Delay Product* (EDP) de aplicações paralelas OpenMP executadas em sistemas embarcados. Ao contrário de soluções anteriores, que focam em processadores de propósito geral (GPP, do inglês *General Purpose Processors*), o presente trabalho considera as características intrínsecas de sistemas embarcados, os quais geralmente possuem menos núcleos disponíveis, assim como apresentam diferenças significativas em relação à micro-arquitetura e à hierarquia de memória. Por meio de experimentos realizados em um sistema embarcado real com processador octa-core, este trabalho mostrou que a adaptação dinâmica do número de *threads* permite, em média, economizar 15,35% no consumo de energia com apenas 3,41% de perda de desempenho, gerando assim 12,47% de otimização de EDP em relação à configuração padrão (uso do máximo número de *threads* disponíveis no sistema). No melhor caso, a adaptação dinâmica foi capaz de economizar 26,97% em energia enquanto promoveu 25,74% de aumento no desempenho, resultando em 45,77% de melhora no EDP.

Palavras-chave: Adaptação dinâmica. otimização de EDP. aplicações paralelas OpenMP. sistemas embarcados.

Dynamic Adaptation of the number of threads for OpenMP applications in embedded systems to optimize EDP

ABSTRACT

Parallel applications usually execute using the maximum number of threads allowed by the available hardware at hand to maximize performance. However, this approach may not be the best when it comes to energy efficiency and may even lead to performance decrease in some particular cases. In this way, the present work proposes a new apporach for the dynamic adaptation of the number of threads to optimize Energy-Delay Product (EDP) of OpenMP applications when running on Embedded Systems. Differently from previous solutions, which focus on General Purpose Processors (GPP), the current one takes into account the intrinsic characteristics of embedded systems, which usually have a lower number of cores and significantly different characteristics concerning the micro-architecture and memory hierarchy when compared to GPPs. Through experiments on a real embedded system with an octa-core processor, this work demonstrates that adapting the number of threads at runtime saves energy, on average, by 15,35% with only 3,41% loss performance, improving the EDP by 12,47% over the default configuration (maximum number of threads available in the system). In the best case, the dynamic adaptation saves 26,97 % in energy while promoting a 25,74 % increase in performance, resulting in a 45,77 % improvement in EDP.

Keywords: dynamic adaptation, EDP optimization, OpenMP parallel applications, embedded systems.

LISTA DE ABREVIATURAS E SIGLAS

DCT	Dynamic Concurrency Throttling
DVFS	Dynamic voltage and frequency scaling
EDP	Energy-Delay Product
GPP	General Purpose Processors
IPC	Instruções por ciclo
RAM	Random Access Memory
SMT	Simultaneous Multithreading
SoC	System-on-Chip
TLP	Thread-level parallelism

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Processadores embarcados atuais possuem múltiplos núcleos de processamento. Para que uma aplicação possa fazer uso destes múltiplos núcleos, e desta forma acelerar sua execução, a aplicação precisa ser implementada para executar em paralelo, explorando o paralelismo no nível da *thread* (TLP, do inglês *Thread-level parallelism*). Espera-se que o desempenho de uma aplicação paralela, em termos de tempo de execução, seja linear ao aumento do número de *threads* utilizados na execução (PUSUKURI; GUPTA; BHUYAN, 2011; NAVAUX; ROSE; PILLA, 2011). Contudo, algumas aplicações não escalam tão bem. Nestes casos, aumentar o número dos núcleos utilizados para executar a aplicação pode aumentar o consumo de energia e até mesmo causar degradação do desempenho (SULEMAN; QURESHI; PATT, 2008; CHADHA; MAHLKE; NARAYANASAMY, 2012). Isto é uma consequência da necessidade de sincronização e comunicação entre as *threads* que pode gerar a saturação de recursos compartilhados (memórias compartilhadas, rede on-chip, largura de banda de memória) (CHADHA; MAHLKE; NARAYANASAMY, 2012; LORENZON; CERA; BECK, 2016).

Além do mais, embora desempenho seja um requisito primordial para diversas aplicações, eficiência energética não pode ser desconsiderada. A economia de energia é uma preocupação atual e importante, não só para processadores de alto desempenho, mas particularmente para sistemas embarcados, principalmente para aqueles que dependem de bateria como fonte de energia. Sendo assim, os requisitos de desempenho e eficiência energética devem ser considerados com equivalente grau de importância. O cenário ideal seria atingir o melhor desempenho com o menor consumo de energia. Contudo, em muitos casos é necessário optar por acelerar a aplicação e consumir mais energia ou o oposto (perder desempenho para economizar energia). Nestas situações, o produto da energia consumida e do tempo de execução (EDP, do inglês *Energy-Delay Product*) é uma métrica útil. EDP permite juntar as duas métricas em um único valor, de modo a se buscar um equilíbrio entre eficiência energética e desempenho.

Neste cenário, determinar o melhor número de *threads* a fim de minimizar o EDP da aplicação não é uma tarefa trivial. Especialmente quando a aplicação possui mais de uma região paralela, onde cada uma pode apresentar um comportamento diferente (BARI et al., 2016). Nestes casos, pode não haver um único valor (ou seja, um ótimo número de *threads*) que otimize toda a aplicação. Nesses casos, é necessário alterar o número de *threads* conforme a região paralela para obter o melhor resultado possível

(LORENZON; SOUZA; BECK, 2017). O melhor número de *threads* para executar uma aplicação paralela pode depender do tamanho da sua entrada. Então, diferentes entradas podem resultar em diferentes números ótimos de *threads*. Além disso, uma determinada região paralela pode receber mudança em sua carga de trabalho em tempo de execução. Por isso, após um determinado período de tempo, o número ideal de *threads* para executá-la também pode mudar.

A prática mais comum, do ponto de vista dos programadores, é executar a aplicação paralela com o máximo número de *threads* de hardware disponíveis no sistema (LEE et al., 2010; CHADHA; MAHLKE; NARAYANASAMY, 2012), ou seja, o máximo número de núcleos de processamento disponíveis). Utiliza-se o máximo número de *threads* pois espera-se maximizar o desempenho (PUSUKURI; GUPTA; BHUYAN, 2011). Essa estratégia pode não ser ideal para obter um melhor EDP para uma determinada aplicação. Portanto, uma solução seria analisar e adaptar o número de *threads* de cada região paralela da aplicação para atingir o objetivo desejado (neste caso, minimizar a métrica EDP). Realizar isso em tempo de execução permite abordar todos os aspectos necessários para encontrar a melhor combinação de número de *threads*. No entanto, a adaptação dinâmica produzirá uma sobrecarga que deve ser equilibrada para que os custos de encontrar o número ideal de *threads* não se sobreponha aos ganhos obtidos por ela.

Embora já existam muitos trabalhos nesta área, poucos deles endereçam o tema a processadores embarcados. Pesquisas que objetivam otimizar aplicações paralelas por adaptar o número de *threads* focam em processadores de propósito geral (GPP, do inglês *General Purpose Processors*) ou servidores, os quais geralmente são mais simples e feitos para baixo consumo energético. Diante disto, este trabalho traz o tema da adaptação do número de *threads* para processadores embarcados. A proposta é realizar esta adaptação de forma dinâmica, permitindo que sejam considerados todos os aspectos que só são perceptíveis em tempo de execução. A seguir, apresenta-se o detalhamento dos objetivos deste trabalho, organizados em objetivos geral e específicos.

1.1 Objetivos

1.1.1 Objetivo geral

Otimizar o produto do consumo de energia e tempo de execução (EDP) de aplicações paralelas OpenMP, implementadas em linguagem C ou C++, durante a sua execução

em processadores embarcados, por meio da adaptação dinâmica do números de *threads*.

1.1.2 Objetivos específicos

- Apresentar exploração de espaço de projeto, em termos de tempo de execução, consumo de energia e EDP, de aplicações paralelas executadas em sistemas embarcados;
- Propor e implementar uma biblioteca que em tempo de execução encontre e ajuste o número de *threads* utilizado para executar cada região paralela de uma aplicação OpenMP, com o objetivo de minimizar EDP;
- Avaliar a adaptação realizada em tempo de execução pela biblioteca proposta em termos de tempo de execução, consumo de energia e EDP;
- Descrever benefícios e limitações da biblioteca proposta;

1.2 Estrutura do documento

Como já mencionado, este trabalho propõe a adaptação dinâmica do número de *threads* utilizadas na execução de uma aplicação paralela OpenMP em sistemas embarcados com o objetivo de otimizar o EDP (produto de energia-tempo) resultante desta execução. Os conteúdos que são abordados em cada capítulo serão apresentados brevemente a seguir.

No Capítulo 2 são apresentados os conceitos que ajudam a compreender o presente trabalho e a motivação para o seu desenvolvimento. Discute-se sobre interfaces de programação paralela (Seção 2.1), dando destaque àquelas baseadas em memória compartilhada, a qual é o foco deste trabalho. Sendo que a otimização proposta destina-se a aplicações paralelizadas com OpenMP, alguns conceitos relacionados a esta interface de programação paralela também são apresentados nesta seção. Aborda-se conceitos de consumo de potência e energia (Seção 2.2) e como estes são impactados pelo número de *threads* utilizadas para executar uma aplicação paralela (Seção 2.3). Dados estes conceitos, aborda-se ainda nesta seção, a motivação pela qual este trabalho busca otimizar EDP ao invés de energia ou tempo.

No Capítulo 3, apresenta-se os trabalhos relacionados a este. Inicialmente apresenta-se uma visão geral sobre abordagens utilizadas para otimização energética em sistemas

embarcados e otimização em aplicações paralelas (Seção 3.1). Em sequencia, são descritas as principais técnicas e abordagens já propostas para realizar adaptação dinâmica do número de *threads* com objetivo de otimizar aplicações paralelas (Seção 3.2), bem como apresenta-se uma comparação destas com a proposta atual, destacando a sua contribuição (Seção 3.3).

No Capítulo 4, apresenta-se a metodologia aplicada nos experimentos realizados neste trabalho. Descreve-se as aplicações executadas (Seção 4.1) e a configuração do sistema embarcado utilizado (Seção 4.2), bem como, o circuito desenvolvido e utilizado para mensurar o consumo real de energia dos experimentos (Seção 4.3).

No Capítulo 5, apresenta-se o espaço de exploração de projeto existente, ou seja, mostra-se que aplicações paralelas executadas em sistemas embarcados também podem se beneficiar do ajuste do número de *threads*. Os resultados da exploração de espaço de projeto são apresentados e discutidos para cada aplicação em relação às métricas de tempo de execução, consumo de energia e EDP.

No Capítulo 6, a abordagem proposta para realizar a adaptação dinâmica do número de *threads* é descrita e avaliada. Na Seção 6.1 apresenta-se uma visão geral do funcionamento da adaptação do número de *threads* (Seção 6.1.1) e descreve-se como foi implementada a biblioteca para realizar o ajuste de maneira dinâmica (Seção 6.1.2). Para realizar a decisão sobre o número de *threads* que devem ser utilizadas na execução para otimizar EDP é necessário que, em tempo de execução, a aplicação esteja consciente do seu consumo de energia. Contudo, a grande maioria dos sistemas embarcados atuais não oferece uma maneira direta de obter esta informação. Por este motivo, a biblioteca precisa fazer uma estimativa do consumo energia. Esta estimativa também é descrita (Seção 6.1.3) e avaliada (Seção 6.2). Os resultados da adaptação dinâmica são apresentados e discutidos na Seção 6.3. Vale notar que, embora durante a execução da aplicação a biblioteca utilize o consumo de energia estimado, a validação dos resultados é baseada no consumo de energia real da aplicação.

Concluindo o trabalho, no Capítulo 7 são apresentadas as considerações finais e direções para trabalhos futuros.

2 BASE TEÓRICA

Nesta são apresentados alguns conceitos que ajudam no entendimento deste trabalho e motivam o seu desenvolvimento. São apresentados os conceitos relacionados às arquiteturas paralelas e modelos de programação paralela (Seção 2.1), bem como, apresenta-se brevemente a interface de programação paralela OpenMP, que é a interface alvo deste trabalho (Seção 2.1.1). São esclarecidos conceitos relacionados a consumo de potência e energia (Seção 2.2) e como estes são afetados pelo uso de múltiplos núcleos do sistema para processar uma aplicação (Seção 2.3).

2.1 Arquiteturas paralelas e modelos de programação paralela

As arquiteturas computacionais atuais possuem múltiplos núcleos de processamento, o que permite que uma aplicação possa ser executada em paralelo e assim acelerar a sua execução. Desta forma, ao invés da aplicação ser executada sequencialmente (por um único núcleo), as tarefas, as quais podem ser executadas de forma concorrente sem prejudicar o resultado da aplicação, são divididas entre os múltiplos núcleos do sistema. Esta divisão é realizada por meio da criação de *threads*, originando assim o termo paralelismo em nível de *thread* (TLP, do inglês *Thread-level parallelism*).

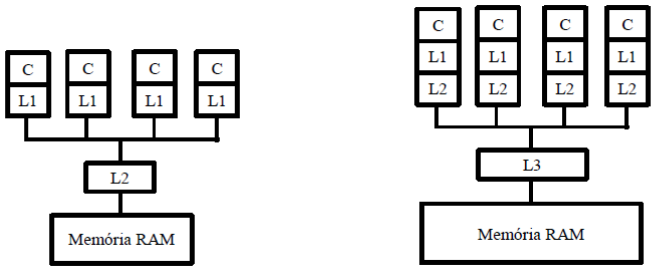
Cada *thread* é executada por um núcleo do sistema. Geralmente as *threads* precisam se comunicar, seja para trocar dados ou para sincronizar a execução. Esta comunicação pode ocorrer através de espaços de memória compartilhada ou via troca de mensagens. Modelos de programação paralela baseados em comunicação por memória compartilhada são geralmente mais eficientes nas arquiteturas em que os núcleos de processamento estão dentro de um mesmo nodo. Isso porque este modelo reduz a sobre carga de comunicação e faz uso da largura de banda de memória disponível dentro do nodo (LIVELY et al., 2008).

Por outro lado, modelos baseados em troca de mensagem (MPI, do inglês *Message Passing Interface*) são geralmente melhores para processadores que estão distantes, ou seja, são aplicados em sistemas com múltiplos nodos, onde não existe um espaço de endereçamento comum (RABENSEIFNER; HAGER; JOST, 2009). Neste modelo o trabalho é divido entre vários processos e estes se comunicam através de trocas de mensagens usando a rede de interconexão. Modelos baseados em troca de mensagem não serão abordados neste trabalho. Isso porque o objetivo deste está direcionado a sistemas

embarcados. Modelos de programação paralela baseados em comunicação por memória compartilhada são mais adequados neste caso, pois sistemas embarcados possuem arquitetura paralela de memória compartilhada.

Em relação às arquiteturas paralelas de memória compartilhada, vale notar que processadores embarcados diferem-se dos processadores de propósito geral (GPP, do inglês *General Purpose Processors*) tanto em hierarquia e tamanho das memórias quanto em frequência de processamento. Isso porque processadores embarcados são projetados para baixo consumo energético. A Figura 2.1 ilustra duas arquiteturas paralelas de memória compartilhada com diferentes hierarquias de memória.

Figura 2.1: Exemplos de arquiteturas paralelas com memória compartilhada



(a) ARM (b) Intel

Fonte: (LORENZON, 2014)

Na arquitetura dos processadores para sistemas embarcados da ARM (Figura 2.1a), os múltiplos núcleos compartilham memória cache de nível 2 (L2) e memória principal, sendo que cada núcleo tem apenas um nível de cache (L1). Já os processadores de propósito geral, como processadores Intel (Figura 2.1b), geralmente possuem L1 e L2 privadas para cada núcleo e mais um nível de cache (L3) que é compartilhada entre eles. Desta forma, enquanto em GPP a comunicação entre as *threads* ocorre no terceiro nível de cache, em processadores embarcados a comunicação ocorre no segundo nível de cache.

2.1.1 Open Multi-Processing (OpenMP)

Open Multi-Processing (OpenMP) é uma interface de programação paralela para memória compartilhada. OpenMP pode ser usado nas linguagens de programação C/C++ e FORTRAN (DAGUM; MENON, 1998). Este trabalho busca otimizar aplicações paralelizadas com esta interface, por isso, nesta seção serão abordados alguns conceitos desta interface necessários para o entendimento do trabalho. Devido a sua facilidade de uso, OpenMP é a interface de programação paralela baseada em memória compartilhada ge-

ralmente mais utilizada. Sendo este o motivo da escolha desta interface como foco para este trabalho.

Em OpenMP as aplicações são paralelizadas por meio da inserção de diretivas de compilação ($\#pragma$) no código da aplicação. Estas diretivas informam ao compilador que a região demarcada deve ser dividida em *threads* e executada em paralelo. A Figura 2.2 ilustra um laço de repetição paralelizado com OpenMP.

Figura 2.2: Exemplo de código paralelizado com OpenMP

```
#include <omp.h>
int main() {
    /* código sequencial  */

    omp_set_num_threads(8)
    #pragma omp parallel for
    for (i = 0; i < 10; i++)
        a[i] = 2 * i;

    /* código sequencial  */
}
```

Fonte: O Autor

A quantidade de *threads* que devem ser criadas para realizar as tarefas em paralelo podem ser informadas via atribuições de variável ambiente (OMP_NUM_THREADS) ou diretamente no código da aplicação por meio da função *omp_set_num_threads()* da biblioteca do OpenMP (como ilustrado na Figura 2.2). Por padrão, sempre que não informado o número de *threads*, usa-se a quantidade máxima de *threads* de hardware disponíveis no sistema (quantidade de núcleos de processamento). Usando esta função é possível determinar o número de *threads* em tempo de execução. Sendo que uma aplicação paralela pode ter uma ou mais regiões paralelas, usando a função *omp_set_num_threads()* pode-se realizar o ajuste fino do número de *threads*, ou seja, pode-se determinar um número de *threads* específico para cada região paralela.

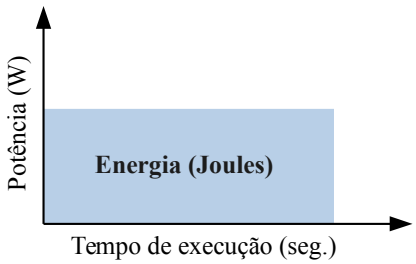
2.2 Consumo energético

Potência e energia, embora sejam relacionadas, são duas grandezas distintas. A potência indica a quantidade de energia consumida em uma unidade de tempo, sendo mensurada em Watts (joule por segundo). A potência é o produto da tensão (medida em Volts) e da corrente (medida em Ampères) que percorre um circuito elétrico. Enquanto a

potência refere-se à potência instantânea consumida, a energia é mensurada em $joules$ e é calculada integrando-se a potência instantânea dentro do intervalo de tempo de interesse (KEATING et al., 2007), conforme a Equação 2.1. Onde, P é a potência consumida no instante de tempo t. A Figura 2.3 ilustra a relação entre potência e energia.

$$Energia(joules) = \int P(t) * dt \tag{2.1}$$

Figura 2.3: Potência vs. Energia



Fonte: O Autor

A potência total dissipada por um circuito consiste de potência dinâmica e potência estática (potência de vazamento). A potência dinâmica é relativa à atividade de chaveamento, enquanto a potência estática refere-se à corrente de vazamento, ou seja, a corrente que flui pelos transistores mesmo sem chaveamento das portas lógicas. Com os projetos de circuitos cada vez menores, a potência estática vem se tornado cada vez mais representativa (KEATING et al., 2007).

Embora a potência dissipada seja um requisito importante (por questões térmicas, há um limite de potência no qual é considerado seguro para o sistema operar), o consumo de energia é o que determina o tempo que uma bateria irá durar. Como este trabalho é direcionado a sistemas embarcados, que em muitas vezes são dependentes de bateria, o interesse principal é no consumo de energia.

2.3 Consumo energético de aplicações paralelas

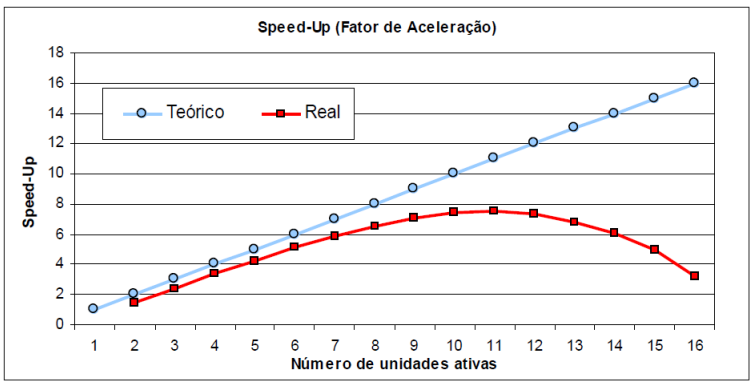
A potência consumida por um sistema varia dependendo do trabalho que está sendo realizado (KEATING et al., 2007). Quando uma aplicação é executada em paralelo, sua execução é acelerada devido à divisão do processamento entre os múltiplos núcleos do processador. Entretanto, mais núcleos trabalhando geram maior dissipação de

potência. Além disso, quando uma aplicação é paralelizada, mais instruções precisam ser executadas (instruções para criar/destruir *threads*, dividir o trabalho, etc.). Há também a necessidade de comunicação entre as *threads*, para sincronizar o trabalho e trocar informações. Para se comunicarem, as *threads* fazem uso das memórias compartilhadas (cache compartilhada e memória principal). Sendo que estas memórias estão mais distantes do processador, maior é a potência consumida e maior o tempo de acesso, aumentando o consumo de energia.

Uma aplicação executada em paralelo pode, então, consumir mais energia que sua versão sequencial. Contudo, o consumo geral de energia vai depender da aceleração proporcionada pela paralelização. Espera-se que a aceleração gerada pela paralelização seja linear ao aumento no número de *threads* utilizados na execução da aplicação. Entretanto, por consequência da sincronização entre as *threads* e saturação de recursos compartilhados (memórias caches compartilhadas, rede on-chip, largura de banda de memória) a escalabilidade das aplicações paralelas é limitada.

A Figura 2.4 ilustra a aceleração esperada e a real gerada pelo aumento no número de *threads*. Pode-se observar que até certa quantidade de *threads* há um ganho quase linear. Mas depois deste ponto o aumento no número de *threads* não gera ganhos, e pode chegar até mesmo a degradar o desempenho da aplicação. A limitação na escalabilidade de uma aplicação paralela pode ser causada por um ou mais fatores, que são relacionados às características de cada aplicação. Segundo Suleman, Qureshi e Patt (2008), as causas mais comuns de limitação são (i) a sincronização de dados e (ii) a saturação dos recursos compartilhados.

Figura 2.4: Aceleração de aplicações paralelas: esperado vs real



Fonte: (NAVAUX; ROSE; PILLA, 2011)

A saturação por sincronização de dados é causada quando há dependência de da-

dos, ou seja, quando as *threads* dependem do resultado gerado por outra *thread* e precisam trabalhar sobre um mesmo dado. Para garantir a coerência deste dado, é necessário assegurar a ordem de acesso, não permitindo que duas *threads* acessem ao mesmo tempo a mesma região de memória. Assim, uma *thread* precisa esperar a outra para ter acesso ao dado. Desta forma, quanto mais *threads* são usadas, maior será o tempo em que elas permanecem esperando em seções críticas, superando assim os benefícios da redução do tempo relativo à divisão do trabalho (SULEMAN; QURESHI; PATT, 2008). Com isto, as aplicações paralelas limitadas por sincronização de dados apresentam perda de desempenho quando executadas com muitas *threads* e consequentemente, aumenta o consumo de energia também.

Quando as aplicações não são limitadas por sincronização de dados, estas podem ser limitadas pela saturação dos recursos compartilhados. Por exemplo, se todas as *threads* estiverem utilizando a rede on-chip, estas serão limitadas por sua largura de banda. Assim, as aplicações paralelas limitadas pela saturação dos recursos compartilhados, após determinado número de *threads*, o aumento no número de *threads* não gera ganho em desempenho. Mas o aumento no número de núcleos trabalhando e aumento na quantidade de acessos às memórias compartilhadas irá gerar maior dissipação de potência, e a aplicação irá, consequentemente, consumir mais energia (SULEMAN; QURESHI; PATT, 2008; CHADHA; MAHLKE; NARAYANASAMY, 2012).

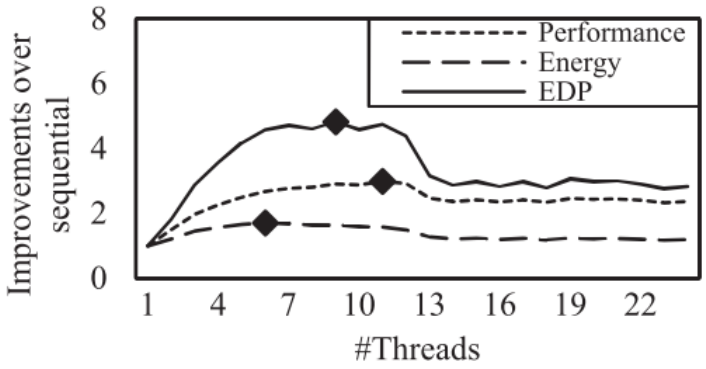
Vale notar que aplicações paralelas com ótima escalabilidade (fator de aceleração quase linear conforme o aumento do número de *threads*) podem ser mais eficientes energeticamente quando executadas com mais *threads*. Aplicações que escalam bem têm pouca comunicação, gerando pouco acesso às memórias compartilhadas (que tem alto custo de potência e tempo de acesso) e ocorre pouca ou nenhuma parada devido à dependência de dados. Desta forma, o desempenho aumenta e o consumo de energia não. Em alguns casos, o consumo de energia pode até ser reduzido, devido à redução do tempo de execução.

Como visto, até certo número de *threads* a paralelização pode trazer benefícios até mesmo para consumo de energia. Como exemplo, a Figura 2.5 representa os resultados obtidos da aplicação paralela LULESH 2.0 quando executada em um sistema x86 com 24 núcleos de processamento (LORENZON; SOUZA; BECK, 2017). Os resultados são apresentados como a melhora em relação à versão sequencial. Observa-se que o ótimo número de *threads* para obter a maior aceleração não é o número máximo de *threads* disponível no sistema (o melhor número de *threads* para cada métrica é representado por

♦).

LULESH 2.0 apresenta ganho de desempenho (*Performance*) até certa quantidade de *threads*. A partir disto, devido a problemas de escalabilidade inerentes da aplicação, à medida que o número de *threads* é aumentado o desempenho começa a degradar. Desta forma, neste exemplo, 12 *threads* é o número que oferece maior desempenho. Quando o foco é eficiência energética (*Energy*), pode-se observar que o melhor número de *threads* seria 6.

Figura 2.5: Aplicação LULESH 2.0 executada em um sistema x86 com 24 *cores*



Fonte: (LORENZON; SOUZA; BECK, 2017)

O melhor número de *threads* para atingir o maior desempenho nem sempre é o melhor para eficiência energética. Embora busca-se eficiência energética não é aceitável perder desempenho, até porque acelerar a execução é o motivo pelo qual uma aplicação é paralelizada. Nestes casos, *Energy-Delay Product* (EDP), representado na (Equação 2.2), é uma métrica útil. EDP permite juntar as duas métricas em um único valor, provendo um equilíbrio entre eficiência energética e desempenho. No exemplo da Figura 2.5, 8 é o melhor número de *threads* para obter o menor EDP.

$$EDP = Energia * Tempo \tag{2.2}$$

Encontrar o melhor número de *threads* para executar uma aplicação paralela, seja para otimizar desempenho, energia ou EDP, não é uma tarefa trivial. O número ideal de *threads* depende do sistema em que a aplicação está sendo executada e do tamanho do conjunto de dados (CHADHA; MAHLKE; NARAYANASAMY, 2012). Além disso, as aplicações podem apresentar variações na carga de trabalho durante a execução ou a situação do sistema também pode mudar (outras aplicações executando junto) (LORENZON; SOUZA; BECK, 2017; SRIDHARAN; GUPTA; SOHI, 2014). Desta forma, a identifi-

cação do melhor número de *threads* e o seu ajuste deve ser feito de forma dinâmica para poder considerar todas estas variáveis.

3 TRABALHOS RELACIONADOS

Este trabalho propõe a adaptação do número de *threads* em aplicações paralelas quando executadas em processadores *multicore* embarcados com objetivo de otimizar EDP, ou seja, obter um balanço entre otimização de eficiência energética e desempenho. Propõe-se que esta adaptação seja realizada de forma automática e dinâmica, podendo assim, considerar informações que só são acessíveis em tempo de execução. A técnica de otimização deste trabalho é realizada por software. Por este motivo, não serão discutidas técnicas de otimização que sejam majoritariamente baseadas em hardware.

Inicialmente na Seção 3.1 será apresentada uma visão geral sobre abordagens que buscam eficiência energética em sistemas embarcados e técnicas para otimização de aplicações paralelas. Na Seção 3.2 será apresentado o estado da arte na técnica de adaptação dinâmica do número de *threads* para otimização de aplicações paralelas. Para concluir esta seção, a contribuição deste trabalho será discutida comparando-o com o estado da arte (Seção 3.3).

3.1 Visão geral

Tratando-se de eficiência energética em sistemas embarcados, há uma gama de trabalhos nesta área e atualmente muitos são endereçados para processadores *multicore* heterogêneos. Esses trabalhos propõem políticas e mecanismos de escalonamento das tarefas (*threads*) nos diferentes núcleos disponíveis no processador. A otimização por meio do escalonamento das tarefas é diferente da técnica abordada no presente trabalho. No escalonamento trabalha-se com as *threads* já existentes, que geralmente são *threads* de diferentes aplicações rodando juntas (carga de trabalho *multithread*) buscando otimizar o sistema como um todo. Por outro lado, a proposta deste trabalho é a otimização da aplicação, ou seja, busca-se otimizar uma aplicação paralela ajustando o número de *threads* utilizadas para executá-la. Analisar o comportamento da técnica proposta neste trabalho quando outras aplicações estão executando junto no sistema, e assim competindo por recursos (o que influencia no desempenho da aplicação), é alvo de trabalhos futuros. Neste trabalho, conforme descrito no Capítulo 4, utiliza-se processador *multicore* homogêneo. A extensão da técnica para processadores com núcleos heterogêneos também é um trabalho futuro de interesse.

Para obter eficiência energética em aplicações paralelas, com relação à otimização

da aplicação, pesquisadores têm explorado oportunidades de economia de energia principalmente através do uso de técnicas de *Dynamic Voltage and Frequency Scaling* (DVFS) e *Dynamic Concurrency Throttling* (DCT). A técnica de DVFS refere-se ao ajuste da tensão de alimentação e frequência de clock do processador para reduzir o consumo de energia. DVFS é aplicado em aplicações paralelas baseadas em trocas de mensagens (MPI) com objetivo de aproveitar os períodos de ociosidade dos processadores, quando estão esperando pela resposta de outro processo, para reduzir a voltagem e frequência dos mesmos e desta forma não gastar potência desnecessariamente. Como exemplos destes trabalhos, pode-se citar Freeh et al. (2008), Rountree et al. (2009) e Bhalachandra et al. (2017). Já a técnica de DCT refere-se à limitação do paralelismo (número de *threads* que estão ativas na execução da aplicação) com objetivo de obter eficiência energética e/ou maior desempenho. DCT é aplicada em aplicações paralelas baseadas em memória compartilhada. Exemplos destes são os trabalhos de Curtis-Maury et al. (2006), Suleman, Qureshi e Patt (2008), Porterfield et al. (2013), Sridharan, Gupta e Sohi (2014) e Lorenzon, Souza e Beck (2017). DCT é a técnica aplicada no presente trabalho e, por isso, terá maior enfoque nesta seção que a técnica de DVFS.

Em aplicações paralelas de memória compartilhada, existem também os trabalhos que exploram o ajuste do número de *threads* (DCT) juntamente com o uso de DVFS, como é o caso dos trabalhos de Chadha, Mahlke e Narayanasamy (2012) e Alessi et al. (2015). A aplicação conjunta das técnicas de DVFS e DCT também é empregada em aplicações híbridas OpenMP/MPI por Li et al. (2013) e Marathe et al. (2015). Embora haja vários modelos de aplicações paralelas (troca de mensagem, memória compartilhada ou híbridas), o foco deste trabalho são aplicações paralelas de memória compartilhada, pois são executadas em um único processador *multicore* (como é o caso em sistemas embarcados). Sendo assim, a seguir serão abordados apenas trabalhos relacionados à otimização de aplicações paralelas de memória compartilhada.

Muitos pesquisadores têm mostrado que aplicações paralelas nem sempre atingem seu melhor desempenho quando são executadas com o máximo de *threads* de hardware disponíveis no processador. Dentre estes trabalhos, há também os que propõem técnicas para encontrar e ajustar de forma automática a melhor configuração para atingir algum objetivo (eficiência energética e/ou desempenho). As técnicas podem ser estáticas ou dinâmicas. Wang et al. (2015), por exemplo, usa um modelo baseado em árvore de decisão para prevê qual a melhor configuração de número de *threads* e modulação de frequência para uma aplicação paralela OpenMP. Contudo, para que esta configuração possa ser

escolhida, a aplicação precisa ser executada uma vez para coletar dados de densidade de acesso a memória. Desta forma, o trabalho de Wang et al. (2015) encontra o melhor número de *threads* usando informação de tempo de execução para tomada de decisão, mas não realiza a adaptação para a melhor configuração durante a execução da aplicação, podendo ser considerada uma técnica estática. As técnicas dinâmicas são aquelas que encontram e ajustam o número de *threads* durante a execução da aplicação, usando ou não informação prévia da aplicação. Pode-se citar como exemplo de propostas de adaptação dinâmica os trabalhos de Curtis-Maury et al. (2006), Suleman, Qureshi e Patt (2008), Lee et al. (2010), Pusukuri, Gupta e Bhuyan (2011), Chadha, Mahlke e Narayanasamy (2012), Porterfield et al. (2013), Sridharan, Gupta e Sohi (2014), Alessi et al. (2015), Bari et al. (2016) e Lorenzon, Souza e Beck (2017). Esses últimos serão detalhados a seguir, pois apresentam uma abordagem de adaptação dinâmica, assim como a proposta deste trabalho.

3.2 Estado da arte

Nesta seção são apresentados os trabalhos que objetivam maximizar desempenho e/ou reduzir consumo de energia de aplicações paralelas por meio da adaptação do dinâmica do número de *threads* utilizadas para executar uma aplicação paralela. Alguns destes trabalhos também aplicam outras técnicas em conjunto com a adaptação do número de *threads*. A descrição dos trabalhos segue em ordem cronológica.

3.2.1 Online power-performance adaptation of multithreaded programs

Curtis-Maury et al. (2006) propõem um *framework* para adaptação do número de *threads* de aplicações paralelas. O *framework* usa previsão de Instruções por ciclo (IPC) para decidir qual a melhor configuração de número de *threads*. A proposta objetiva melhorar o desempenho e reduzir o consumo de energia desativando *threads* (em processadores SMT, do inglês *Simultaneous multithreading*), cores em execução ou processadores inteiros. A proposta foca em aplicações OpenMP e é avaliada em um sistema real multi-SMT (Dell PowerEdge server) composto por 4 processadores Intel Xeon MP com hyper-threading, totalizando 8 *threads*.

Sendo que a proposta de Curtis-Maury et al. (2006) faz uso de um modelo de pre-

dição proposto pelos autores, não há necessidade de testar diferentes configurações para decidir qual número de *threads* é melhor. A técnica requer apenas duas execuções (recorrentes da mesma região) para prever o IPC para cada configuração e então ser possível decidir qual a melhor configuração. Para a predição são utilizados contadores de eventos de hardware, os quais são coletados e analisados por região paralela durante a execução da aplicação. No modelo de predição são consideradas as informações de instruções executadas por ciclo (IPC), taxa de acesso ao *bus*, taxa de *misses* na *cache* L2, taxa de *mispredicted branches*, taxa de *branch instructions* e a porcentagem de ciclos em que a *cache* de rastreamento do processador está no modo de entrega. Todos os eventos de hardware coletados são normalizados pelo tempo de duração em ciclos da região paralela. Para estimar o consumo de energia relativo a cada configuração é usada a informação de ciclos por instrução (CPI), e considera-se apenas consumo de potência estática.

A adaptação do número de *threads* é feita em tempo de execução e não requer conhecimento prévio das características da aplicação. A proposta tem uma etapa offline apenas para coletar os coeficientes do modelo de predição. Nesta etapa offline são usados dois *benchmarks* (que não são utilizados na avaliação): UA do NAS e MM5 – totalizando 248 regiões paralelas de treinamento. Na avaliação da proposta são usados 8 *benchmarks* do NAS. Nos experimentos Curtis-Maury et al. (2006) analisaram estratégias de adaptação com diferentes objetivos: (i) otimizar tempo de execução (que só usa a predição de IPC e não de consumo de energia), (ii) otimizar economia de energia, (iii) ED (do inglês *energy-delay*) e (iv) ED2 (do inglês *energy-delay-squared*).

Na avaliação da proposta, Curtis-Maury et al. (2006) apresentam resultados para as métricas de tempo de execução, consumo de energia, ED e ED2. Os resultados são comparados com a configuração padrão, ou seja, com o máximo número de *threads* possível (8 *threads*), a melhor configuração atribuída estaticamente (oráculo), adaptação com busca exaustiva feita em tempo de execução, e com um trabalho prévio dos mesmos autores em que usava-se um algoritmo *hill-climbing* ao invés do modelo de predição. A estratégia de adaptação centrada no desempenho, que usa as previsões de IPC, supera as demais abordagens adaptativas em todas as métricas (tempo de execução, energia, ED, ED2). Segundo Curtis-Maury et al. (2006), este fato deve-se ao erro na estimativa do consumo de energia (que é utilizada nas demais estratégias). A estratégia de adaptação baseada em predição de IPC apresenta em média uma redução 22% do tempo de execução quando comparada com a configuração padrão (8 *threads*) e consome em média 26% menos energia. A técnica proposta por Curtis-Maury et al. (2006) assume que uma região

paralela terá sempre o mesmo comportamento, ou seja, não considera-se a possibilidade de mudança de carga de trabalho da aplicação durante a execução.

3.2.2 Feedback-driven threading (FDT)

Suleman, Qureshi e Patt (2008) propõem um *framework* chamado *Feedback-driven threading* (FDT) que estima o número de *threads* que satura a performance de aplicações *multi-threads*. No artigo, Suleman, Qureshi e Patt (2008), apresentam a implementação do framework proposto para os dois maiores limitadores de performance em aplicações paralelas: (i) sincronização de dados, o qual os autores chamam de *Synchronization-Aware Threading* (SAT) e (ii) largura de banda, chamado de *Bandwidth-Aware Threading* (BAT). Os autores também apresentam a implementação da combinação entre SAT e BAT, em que o menor número de *threads* encontrado por ambos é atribuído como número ideal de *threads*.

Suleman, Qureshi e Patt (2008) descrevem o modelo analítico usado em cada implementação. Na implementação do SAT, o modelo usado para encontrar o ponto de saturação, causado pela sincronização, baseia-se no tempo gasto nas regiões que são consideradas críticas e no tempo da região paralela. Na implementação do BAT, para encontrar o ponto de saturação devido a limitação da largura de banda, calcula-se a utilização do *bus* como a relação entre o número de ciclos que o *bus* está ocupado e o total de ciclos. Segundo Suleman, Qureshi e Patt (2008), quando uma aplicação paralela é limitada por sincronização de dados, o uso de mais *threads* degrada o seu desempenho (tempo de execução) e consome mais potência, causando maior consumo de energia. Quando a aplicação é limitada pela largura de banda, o aumento no número de *threads* não reduz tempo de execução, mas aumenta a potência consumida. Desta forma, a proposta de Suleman, Qureshi e Patt (2008) objetiva reduzir o tempo de execução e a potência, e consequentemente o consumo de energia, das aplicações paralelas.

A técnica de Suleman, Qureshi e Patt (2008) controla dinamicamente e em tempo de execução o número de *threads* que deve ser usado para executar cada região paralela da aplicação. A técnica não requer informação prévia da aplicação e a adaptação é realizada em tempo de execução, por isso se adapta ao conjunto de entrada e a configuração do sistema. O FDT tem uma fase em que os autores chamam de treinamento, mas que ocorre durante a execução. Nesta fase, a região paralela é executada com apenas uma *thread* durante um curto período de tempo. A partir desta execução são coletadas informações

que são usadas para determinar qual o número de *threads* que satura o sistema. O restante da execução da região paralela é realizada com o número ideal de *threads* encontrado.

Suleman, Qureshi e Patt (2008) avaliaram sua proposta em um simulador x86 configurado com 32 cores. Na avaliação foram usados 12 *benchmarks* paralelizados com OpenMP: 4 limitados por sincronização de dados, 4 limitados por largura de banda e 4 que não tem nenhuma dessas limitações, ou seja, são escaláveis. Como resultado, a implementação combinada de SAT e BAT reduziu, em média, o tempo de execução em 17% e o consumo de potência em 59% comparado com a execução realizada com o número de *threads* igual ao número de cores, ou seja, 32 *threads*.

3.2.3 Thread tailor

Lee et al. (2010) propõem um compilador dinâmico, chamado *Thread tailor*, que objetiva otimizar o tempo de execução de aplicações paralelas. Apesar da adaptação do número de *threads* ser realizada dinamicamente e em tempo de execução, a técnica de Lee et al. (2010) requer uma fase offline. Nesta fase, a aplicação é executada para coletar estatísticas de número de *threads* e padrões de comunicação e sincronização. A partir destas informações é construído um grafo (específico para cada aplicação) que representa a comunicação entre as *threads*. Esse grafo é usado em tempo de execução para ajuda a decidir como agrupar as *threads* de forma a minimizar a comunicação e sincronização entre elas.

Em tempo de execução, o compilador dinâmico (Thread Tailor) proposto pelos autores coleta informações do estado atual do sistema para determinar quantos recursos estão disponíveis (ex.: número de núcleos do processador e espaço de cache disponível) e executar um algoritmo de particionamento do grafo de comunicação. Para determinar a largura de banda de memória utilizada por cada *thread* são utilizados contadores de performance de hardware. Segundo Lee et al. (2010), a proposta não requer a recompilação dos programas, pois ela atua no binário do executável da aplicação. Contudo, nos experimentos, as aplicações foram compiladas estaticamente com 128 *threads* para permitir que durante a execução o compilador dinâmico proposto pelos autores possa agrupá-las. A abordagem de Lee et al. (2010) defende a criação inicial de várias *threads* e a redução dinâmica para se ajustar as características da aplicação e situação atual do sistema.

Lee et al. (2010) usaram o conjunto de ferramentas *Low-Level Virtual Machine* (LLVM) para prototipar o compilador dinâmico proposto por eles. Os experimentos fo-

ram realizados em três sistemas diferentes: (i) Intel Core 2 6600 dual-core (2 threads de hardware), (ii) Intel Core 2 Q6600 quad-core (4 threads de hardware) e (iii) Intel Xeon E5520 dual quad-core sem e com SMT habilitado (8 e 16 threads de hardware). Os resultados foram comparados com (i) a execução usando o máximo de threads de hardware, com (ii) a técnica de Suleman, Qureshi e Patt (2008) e com (iii) a própria técnica proposta (avaliando a técnica com e sem analise de comunicação e sincronização). Em média, o compilado dinâmico proposto por Lee et al. (2010) obteve um ganho de 19% no tempo de execução.

3.2.4 Thread reinforcer

Pusukuri, Gupta e Bhuyan (2011) propõem um *framework*, chamado *Thread reinforcer*, para automaticamente selecionar o número adequado de *threads* para maximizar o desempenho de aplicações paralelas. Para encontrar o número ideal de *threads* o *framework* baseia-se em observações de fatores do sistema operacional: (i) contenção de bloqueio, (ii) taxa de migração de *thread*, (iii) taxa de troca de contexto voluntária e (iv) utilização do processador. O *framework* executa a aplicação em duas etapas. Na primeira etapa, a aplicação é executada diversas vezes por um curto período de tempo. Durante esta etapa o comportamento da aplicação é monitorado. Com base nestas obervações o *framework* encontra o número de *threads* apropriado. Após encontrado o número ideal de *threads*, a segunda etapa inicia. Na segunda etapa a aplicação é completamente reexecutada usando o número de *threads* encontrado na primeira etapa.

Pusukuri, Gupta e Bhuyan (2011) mostram que algumas aplicações apresentam melhor desempenho quando são executadas com mais *threads* que o número de *cores*. Na análise de escalabilidade as aplicações foram executadas com até 128 *threads*. Nos experimentos são executadas aplicações do PARSEC (para estudos e avaliação), SPEC OMP e PBZIP2 (na avaliação) em um sistema com 24 *cores* (Dell PowerEdge R905) rodando o sistema operacional Solaris.

A técnica proposta por Pusukuri, Gupta e Bhuyan (2011) usa informações dinâmicas, mas precisa executar a aplicação por um curto período de tempo primeiro, para poder encontrar o número ideal de *threads* e, então, executar toda a aplicação novamente com o número de *threads* encontrado. Pusukuri, Gupta e Bhuyan (2011) mostram nos resultados que o custo é baixo para fazer isso, pois as aplicações executam por um período mais longo que o tempo necessário para busca. Segundo Pusukuri, Gupta e Bhuyan (2011), as

aplicações que tem um período de inicialização maior serão prejudicadas. A técnica também não considera mudanças nas fases da aplicação (em aplicações que possuem regiões paralelas com comportamento diferentes).

3.2.5 When less is more (LIMO)

Chadha, Mahlke e Narayanasamy (2012) propõem o LIMO (do inglês *When less is more*), um sistema de tempo de execução que dinamicamente adapta o número de *threads* que estão executando a aplicação para maximizar desempenho e eficiência energética. LIMO monitora a escalabilidade da aplicação e a demanda por recursos compartilhados, para saber quando deve ser reduzido o número de *threads* ativas. Quando alguma *thread* pára, em uma função de sincronização, bloco de E/S ou quando é suspensa, o número de *threads* é reduzido. Para impedir a saturação da *cache* compartilhada, o sistema estima o tamanho do conjunto de trabalho da aplicação. Além do controle do número de *threads*, o sistema proposto por Chadha, Mahlke e Narayanasamy (2012) também aplica DVFS. Os núcleos do processador que não estão trabalhando são desabilitados e a frequência dos núcleos ativos é aumentada. Quando todos os núcleos estão ativos, o sistema os coloca em uma frequência mais baixa.

A proposta de Chadha, Mahlke e Narayanasamy (2012) assume suporte de sistema operacional, hardware e compilador. Para avaliação do sistema proposto, Chadha, Mahlke e Narayanasamy (2012) simulam um sistema x86 com 32 *cores* usando o simulador FeSi. O uso do simulador se deve ao fato da proposta requerer modificação de hardware. Nos experimentos foram utilizadas 8 aplicações do PARSEC e 2 aplicações do ALPBench. Como resultado as aplicações paralelas apresentaram, em média, melhora de 21% no desempenho e 2$\times$ na redução de consumo energético comparado com as aplicações executadas com a configuração padrão (32 *threads*).

3.2.6 Concurrency Throttling for Energy Reduction in OpenMP Programs

Porterfield et al. (2013) avaliaram o desempenho e o consumo de energia de catorze aplicações. Em quatro dessas aplicações a aceleração não escalou tão bem, mas o consumo de energia aumentou conforme o aumento no número de *threads* usadas para executar a aplicação. Diante destes resultados, Porterfield et al. (2013) propuseram um

sistema de tempo de execução que adapta automaticamente o número de *threads* usadas para executar uma aplicação paralela baseando-se em dados *online* coletados de contadores de performance de hardware.

O sistema proposto por Porterfield et al. (2013) limita o paralelismo em regiões do código em que a potência consumida e a utilização da largura de banda de memória são altas. Quando a biblioteca detecta um alto consumo de potência e grande utilização de largura de banda de memória, uma *flag* é ativada para informar que o número de *threads* deve ser reduzido na próxima oportunidade. A biblioteca RALP é utilizada para coletar as informações *online* referentes ao consumo de potência. Além de reduzir o número de *threads* utilizadas, a técnica proposta por Porterfield et al. (2013) aplica também o ajuste do ciclo de trabalho (*duty-cycle*) nos núcleos de processamento não utilizados.

Porterfield et al. (2013) implementaram a técnica proposta alterando a biblioteca de tempo de execução chamada *Qthreads*. Os autores modificam o escalonador desta biblioteca para considerar nas decisões de escalonamento o uso atual de potência e largura de banda de memória. A adaptação do número de *threads* usadas pode ser realizado pela biblioteca em qualquer ponto de inicialização de *threads* (início de um laço paralelo ou instanciação de tarefa).

Na avaliação da proposta, Porterfield et al. (2013) utilizam um sistema com 2 Intel Xeon E5-2680 (Sandybridge), totalizando 16 *threads* de hardware. Nas quatro aplicações vistas como não estaláveis durante os experimentos iniciais, a técnica proposta por Porterfield et al. (2013) proporcionou redução de potência e o uso geral de energia em até 3%. Nas demais aplicações, a técnica não detectou a necessidade de reduzir o número de *threads* e apresentou um pequeno custo de até 0.6%.

3.2.7 Varuna

Sridharan, Gupta e Sohi (2014) propuseram o Varuna, um sistema de tempo de execução que adapta o número de *threads* da aplicação para que esta seja executada com o melhor número de *threads* de acordo com a capacidade de recursos de hardware. A adaptação é feita dinamicamente e periodicamente, por isso o número de *threads* é adequado conforme as mudanças de carga de trabalho da aplicação e conforme as mudanças no sistema, como, por exemplo, outras aplicações executando junto no mesmo sistema.

A proposta de Sridharan, Gupta e Sohi (2014) funciona tanto para aplicações *multithreaded* (Pthreads) quanto para task-based (TBB). Além disso, não requer alteração no

sistema operacional, compilador ou código do programa. Para compilar a aplicação é usado o compilador GCC, apenas trocando "–lpthreads" ou "–ltbb" por "–lvaruna". Varuna usa tarefas virtuais (Vtasks) para abstrair as *threads* de hardware em tarefas lógicas, e assim conseguir gerenciar o paralelismo (reduzir ou aumentar o número de *threads* usados dependendo da necessidade).

Para que o Varuna possa encontrar o melhor número de *threads*, Sridharan, Gupta e Sohi (2014) propõem um modelo de escalabilidade que permite calcular o melhor número de *threads*. Para a entrada deste modelo, a parte paralela da aplicação é executada por um curto período de tempo (100ms) com apenas 1 *thread* (para ter um parâmetro de desempenho). Depois são feitas medidas em três outros pontos: 2 *threads*, N/2 e N *threads* (sendo N o número máximo de *threads* de hardware do sistema). Apesar de ser usado esse modelo, Sridharan, Gupta e Sohi (2014), comentam que às vezes é necessário fazer uma execução com quantidade de *threads* escolhida randomicamente, para que a escolha não caia em um ótimo local.

Para avaliar a técnica proposta, Sridharan, Gupta e Sohi (2014), usaram dois sistemas com diferentes configurações: (i) Opteron-8350 (AMD) com 4 *sockets* e total de 16 *threads* de hardware; e (ii) Xeon E5-2420 (Intel) com 2 *sockets* com SMT e total de 24 *threads* de hardware. Foram utilizados 12 *benchmarks*, de diferentes conjuntos de benchmarks. Foram realizados experimentos em ambiente isolado (só uma aplicação executando) e em ambiente com mais aplicações executando ao mesmo tempo.

Segundo Sridharan, Gupta e Sohi (2014), Varuna pode ser configurado para minimizar tempo de execução ou minimizar consumo de recurso (produto da média de *threads* de hardware usados pelo programa e do tempo de execução total), mas pode ser estendido para considerar outras métricas. Quando o objetivo é minimizar o tempo de execução, Varuna reduz o tempo em média em 15% e 33%, respectivamente, em ambiente isolado e ambiente com mais aplicações executando junto, quando comparado ao número de *threads* padrão. Nestes mesmos casos, o consumo de energia é reduzido em 31% e 32%, respectivamente.

3.2.8 OpenMPE

Alessi et al. (2015) propuseram uma extensão para o OpenMP, o qual chamam de OpenMPE. Nesta extensão é possível informar juntamente com os *pragmas* do OpenMP os objetivos que se deseja atingir (otimizar tempo de execução, reduzir potência, redu-

zir consumo de energia ou manter uma determinada qualidade de serviço). Devido às alterações nas anotações dos *pragmas*, a técnica proposta requer compilador específico. Na implementação do sistema de tempo de execução, Alessi et al. (2015) utilizam duas técnicas: DVFS e DCT. O sistema testa diferentes configurações selecionadas randomicamente, e usa um algoritmo *hill climbing* para encontrar a melhor configuração.

Nos experimentos, Alessi et al. (2015) utilizam dois sistemas diferentes para avaliar sua proposta: (i) uma placa de desenvolvimento com um processador ARM big.LITTLE (Cortex-A15 quad-core + Cortex-A7 quad-core) e (ii) um sistema de propósito geral com um processador Intel i7-3770k Ivy Bridge quad-core. Os dois sistemas oferecem contadores de hardware de consumo de energia. No processador Intel, o sistema utiliza a biblioteca RALP, e a placa de desenvolvimento utilizada (ODROID XU-E) é equipada com sensor de corrente e voltagem. Desta forma, o sistema é consciente do consumo energético real, ou seja, não há necessidade de fazer estimativa do consumo. Para a avaliação, os autores paralelizaram a aplicação *tmndec*, do conjunto de benchmarks MediaBench II.

O trabalho de Alessi et al. (2015) foi o único encontrado que aplica adaptação de número de *threads* em aplicações paralelas executadas em processador embarcado. Como resultado, o uso do DVFS apresentou grande economia de energia mas a adição da técnica de DCT (DVFS + DCT) não gerou ganhos à mais. Segundo Alessi et al. (2015), isto se deve ao fato de que o desempenho da aplicação utilizada nos experimentos escala quase linearmente com o aumento do número de *thread*.

3.2.9 ARCS

Bari et al. (2016) propuseram o *framework* ARCS, que objetiva melhorar o tempo de execução e reduzir o consumo energético de aplicações paralelas OpenMP atendendo a um requisito potência. Para atingir seu objetivo, em tempo de execução, o *framework* seleciona a melhor configuração de número de *threads*, política de escalonamento e o tamanho usado na divisão das iterações entre as *threads*. Por meio destas configurações, procura-se melhorar comportamento de cache e o balanceamento de carga (para reduzir o tempo em que as *threads* ficam em barreiras de sincronização).

O *framework* proposto por Bari et al. (2016), em tempo de execução avalia diferentes configurações até o valor convergir para um ótimo. A busca pela melhor configuração ocorre em um espaço de busca reduzido. O espaço de busca foi reduzido manualmente pelo autores. A configuração é realizada por região paralela (laços paralelos). Quando a

aplicação termina sua execução, o *framework* grava as configurações para serem utilizadas da próxima vez que a aplicação for executada. Para o funcionamento do framework, é utilizada uma extensão da biblioteca de tempo de execução do OpenMP, mas não há necessidade de alteração na aplicação.

A proposta de Bari et al. (2016) foca em aplicações de computação de alto desempenho (HPC, do inglês *High Performance Computing*). Para avaliação da proposta foram usados três aplicações: LULESH 2.0 e dois *benchmarks* do conjunto do NAS (SP e BT). Os experimentos foram realizados com cinco diferentes requisitos de limite de potência e duas cargas de trabalho diferentes, em dois sistemas: Intel Sandy Bridge (com 32 *hyper-threads*) e IBM POWER8 (160 *threads* de hardware). Comparando com a configuração padrão, o melhor resultado obtido foi de 40% de ganho em tempo de execução e 42% em economia de energia.

Embora proposta de Bari et al. (2016) trabalhe sobre um requisito de limite de potência, seu objetivo é prover otimização de desempenho e redução de consumo de energia. Sobre uma outra abordagem, o trabalho de Sensi, Torquati e Danelutto (2016) aplica diversas técnicas (ajuste do número de *threads*, frequência do clock do processador e localidade das *threads*) para atender requisitos do usuário, que podem ser limite de potência ou mínimo de performance. Sendo assim, o objetivo de Sensi, Torquati e Danelutto (2016) não é a otimização de desempenho e/ou energia, mas sim o atendimento aos requisitos. Ao contrário de Sensi, Torquati e Danelutto (2016), o presente trabalho busca otimização do consumo total de energia e desempenho da aplicação e sem trabalhar sobre um limite de potência ou requisito de desempenho.

3.2.10 LAANT

Lorenzon, Souza e Beck (2017) propõem o LAANT, uma biblioteca para automaticamente otimizar EDP (*Energy-Delay Product*) em aplicações OpenMP. Durante a execução de uma aplicação, a biblioteca encontra o número de *threads* ideal para cada região paralela e realiza a adaptação. LAANT não requer modificação de hardware ou compilador específico e não necessita de conhecimento prévio da aplicação para encontrar o melhor número de *threads*. O presente trabalho foi inspirado nesta proposta de Lorenzon, Souza e Beck (2017). Contudo, o trabalho de Lorenzon, Souza e Beck (2017) é voltado para processadores Intel, enquanto o presente trabalho destina-se a processadores ARM.

Para utilizar a biblioteca proposta por Lorenzon, Souza e Beck (2017), faz-se necessário apenas realizar anotações de código, usando as funções oferecidas pela biblioteca. As funções devem ser inseridas no início e ao final de cada região paralela. A função inserida antes é responsável por atribuir o número de *threads* que deve ser usado para executar a região paralela e por inicializar a contagem de tempo e consumo de energia da região. Após a execução da região paralela, para obter o tempo de execução a biblioteca usa a função *omp_get_wtime()* do OpenMP. A energia consumida é obtida diretamente de contadores de hardware, usando a biblioteca de RAPL (disponível nos processadores atuais de Intel), que provê energia e potência consumida pelo componentes.

Para encontrar o melhor número de *threads* a biblioteca proposta por Lorenzon, Souza e Beck (2017) usa uma heurística baseada em algoritmo *hill-climbing*. A cada ocorrência de uma região paralela, um novo passo é dado no algoritmo. Com base nas informações coletadas de tempo de execução e consumo de energia, o algoritmo decide qual será o número de *threads* que será utilizado para executar a região paralela na próxima iteração. Após encontrado o número de *threads* que oferece o melhor EDP, a região paralela será executada com este número. Periodicamente a biblioteca verifica se houve mudanças no comportamento da região paralela ou variação na carga de trabalho, e se necessário é realizando uma nova busca por outro número de *threads*.

Para avaliar a biblioteca proposta, Lorenzon, Souza e Beck (2017) utilizam nove *benchmarks*, já paralelizados com OpenMP, de diferentes conjuntos de *benchmarks*. Cada *benchmark* foi executado com dois conjuntos de dados de entrada diferentes. Os experimentos foram realizados em três processadores Intel *multicore* que suportam diferentes números de *threads* (8, 24 e 32 *threads*). Como resultado, a biblioteca proposta por Lorenzon, Souza e Beck (2017) consegui reduzir o EDP das aplicações em 29% em média comparado com a configuração padrão (quando a aplicação é executada com o máximo número de *threads* de hardware). Quando comparado com com o ajuste dinâmico oferecido pela biblioteca do OpenMP (que visa o melhor uso dos recursos do sistema), LAANT reduz o EDP em até 82%. Embora o trabalho apresente resultados para otimização de EDP, a biblioteca LAANT pode ser usada também para otimizar outra métrica como tempo ou energia.

3.3 Contribuição

Esta seção tem por objetivo destacar as principais semelhanças e diferenças do presente trabalho comparado com os trabalhos descritos na seção anterior. As principais características de cada um são apresentadas na Tabela 3.1. Todos os trabalhos realizam adaptação dinâmica do número de *threads*.

A primeira coluna da Tabela 3.1 (Sem profile a priori) indica que a técnica não necessita de nenhuma informação prévia da aplicação para encontrar o número ideal de *threads*. Embora possam ser consideradas técnicas dinâmicas, os trabalhos de Lee et al. (2010) e Pusukuri, Gupta e Bhuyan (2011) precisam de informações prévias da aplicação. A proposta de Lee et al. (2010) precisa que a aplicação seja executada uma vez para coletar informações relativas à comunicação e sincronização das *threads* (para construção do grafo de comunicação que é utilizado em tempo de execução). Já na abordagem de Pusukuri, Gupta e Bhuyan (2011) a aplicação é executada inicialmente durante um curto período de tempo, para coletar informações do sistema operacional, e depois ela é reexecutada por inteiro novamente. A proposta do presente trabalho, assim como a grande maioria dos demais trabalhos, não faz uso de informação a priori da aplicação, ou seja, encontram o número de *threads* ideal dinamicamente e sem nenhuma informação prévia da aplicação.

A segunda coluna (Adapt. contínua) indica os trabalhos que monitoram constantemente, ou periodicamente, o desempenho da aplicação, de forma a reagir às mudanças e buscar uma nova configuração de número de *threads* quando necessário. Assim, estes trabalhos se adaptam às variações na carga de trabalho da aplicação ou as mudanças ocorridas no sistema. Chadha, Mahlke e Narayanasamy (2012) não deixam claro se a abordagem proposta por eles faz ou não esta atividade de monitoramento. Atualmente, a proposta do presente trabalho, não monitora as regiões depois de encontrado o número de *threads*, ou seja, não reage às mudanças que possam ocorrer. O monitoramento periódico para identificar mudança na carga de trabalho e/ou na situação atual do sistema e assim poder reagir a isso (fazendo com que uma nova adaptação seja realizada) será desenvolvido como trabalho futuro.

Todos os trabalhos apresentados destinam-se a aplicações paralelas de memória compartilhada. Contudo, não são todas que se destinam a aplicações OpenMP. Na terceira coluna da Tabela 3.1 são indicados os trabalhos que também são direcionados a aplicações OpenMP, assim como este trabalho.

Tabela 3.1: Comparação com o estado da arte

	Sem profile a priori	**Adapt. contínua**	**OpenMP**	**Sistema**		**Ambiente real**	**#threads hardware**
				GPP/Servidor	**Embarcado**		
Curtis-Maury et al. (2006)	x		x	x		x	8
Suleman, Qureshi e Patt (2008)	x	x	x	x			32
Lee et al. (2010)		x		x		x	2, 4, 8 e 16
Pusukuri, Gupta e Bhuyan (2011)				x		x	24
Chadha, Mahlke e Narayanasamy (2012)	x	?		x			32
Porterfield et al. (2013)	x	x	x	x		x	16
Sridharan, Gupta e Sohi (2014)	x	x		x		x	16 e 24
Alessi et al. (2015)	x		x	x	x	x	8 e 4
Bari et al. (2016)	x		x	x		x	32, 160
Lorenzon, Souza e Beck (2017)	x	x	x	x		x	8, 24 e 32
A proposta deste trabalho	x		x		x	x	8

O grande diferencial deste trabalho para os demais é o tipo de sistema alvo (quarta e quinta coluna da Tabela 3.1). Enquanto a maioria dos trabalhos já existentes busca otimização de aplicações paralelas em processadores de propósito geral ou servidores, o presente trabalho destina-se a sistemas embarcados. Sistemas embarcados geralmente são mais simples, por serem projetados para baixo consumo energético. Embora o trabalho de Alessi et al. (2015) se destine também a sistemas embarcados, os ganhos apresentados com a técnica proposta refere-se à aplicação de DVFS. Alessi et al. (2015) não aplicaram somente a adaptação do número de *threads*. Quando aplicado DVFS juntamente com adaptação do número de *threads* os autores não observaram ganhos diferentes do uso apenas do DVFS devido à alta escalabilidade da aplicação utilizada na avaliação. Desta forma, o presente trabalho é o primeiro a mostrar espaço de otimização e adaptar dinamicamente o número de *threads* usados para executar aplicações paralelas em sistemas embarcados.

Este trabalho, assim como a maioria dos trabalhos relatados, realiza experimentos em sistemas reais, conforme indicado na sexta coluna (Ambiente real). Apenas os trabalhos de Suleman, Qureshi e Patt (2008) e Chadha, Mahlke e Narayanasamy (2012) utilizam simuladores para avaliação da abordagem proposta. A sétima coluna (#threads hardware) indica a quantidade de *threads* de hardware disponível nos sistemas utilizados nos trabalhos listados. Geralmente, processadores de propósito geral e servidores oferecem mais *threads* de hardware que sistemas embarcados, desta forma o espaço que existe para otimização é maior. Somente os trabalhos de Curtis-Maury et al. (2006), Lee et al. (2010), Alessi et al. (2015) e Lorenzon, Souza e Beck (2017) utilizam sistemas com 8 *threads* ou menos.

4 METODOLOGIA

Este capítulo apresenta a metodologia utilizada nos experimentos realizados neste trabalho. As aplicações utilizadas na exploração de espaço de projeto (Capítulo 5) e na avaliação da proposta da adaptação dinâmica do número de *threads* (Capítulo 6) são descritas na Seção 4.1. Os experimentos foram realizados conforme a configuração descrita na Seção 4.2.

Assim como grande parte dos sistemas embarcados, a placa de desenvolvimento utilizada nos experimentos deste trabalho (descrita na Seção 4.2) não possui sensor de corrente e voltagem. Desta forma, não existe uma forma direta de medir o consumo de energia da aplicação. Diante disto, para ser possível mensurar o consumo de energia das execuções realizadas nos experimentos, foi necessário o desenvolvimento de um circuito para coletar a corrente e tensão consumidas. O desenvolvimento deste circuito é descrito na Seção 4.3.

4.1 Benchmarks

Para os experimentos realizados neste trabalho foram utilizados oito *benchmarks* implementados nas linguagens de programação C/C++ e já paralelizados com OpenMP. Para a seleção dos mesmos, buscou-se aplicações de diversos domínios de aplicação e com diferentes comportamentos. Os *benchmarks* e o conjunto de entrada utilizados são apresentados na Tabela 4.1. A coluna *#regiões paralelas* refere-se à quantidade de regiões paralelas recorrentes que a aplicação possui e a coluna *Tempo execução* refere-se ao tempo médio quando executado no sistema descrito na Seção 4.2 com máximo número de *threads*. Estas aplicações foram escolhidas por serem paralelizadas com OpenMP e possuírem regiões paralelas recorrentes, ou seja, regiões paralelas que são executadas diversas vezes durante a execução da aplicação. Muitas destas aplicações já têm sido utilizadas em trabalhos anteriores que aplicam a técnica de adaptação do número de *threads*.

Foram utilizados dois kernels (CG e FT) e duas pseudo-aplicações (SP e LU) pertencentes ao conjunto de *benchmarks* paralelos do NASA Advanced Supercomputing (NAS) (BAILEY et al., 1991), três aplicações (HotSpot, StreamCluster, SRAD) do conjunto de *benchmark* Rodinia (CHE et al., 2009) e o algoritmo FFT (ARBENZ; PETERSEN, 2004). Os *benchmarks* CG, FT, SP e LU do NAS foram originalmente desenvolvidos na linguagem de programação Fortran (JIN; FRUMKIN; YAN, 1999), mas neste

Tabela 4.1: Aplicações e conjunto de entrada utilizados nos experimentos

Aplicação	**#regiões paralelas**	**Conjunto de entrada**	**Tempo de execução**
CG	3	Classe A (14000 linhas; 15 iterações)	3,26 seg.
FFT	3	22 iterações	11,35 seg.
FT	5	Classe A (256 x 256 x 128;6 iterações)	13,13 seg.
HotSpot	1	1024 x 1024; 2000 iterações	21,40 seg.
LU	2	Classe A (64 x 64 x 64; 250 iterações)	103,58 seg.
SP	9	Classe A (64 x 64 x 64; 400 iterações)	120,15 seg.
SRAD	2	2048 x 2048; 20 iterações	3,66 seg.
StreamCluster	2	65536 pontos de dados	46,25 seg.

Fonte: O Autor

trabalhado utilizou-se a versão na linguagem C, a qual foi traduzida por Seo, Jo e Lee (2011). Sendo assim, todas as aplicações utilizadas são em linguagem de programação C/C++ e paralelizados com OpenMP.

O conjunto de *benchmarks* do NAS é largamente utilizado na literatura pois endereçam muitos dos problemas associados com a avaliação de máquinas paralelas (BAILEY et al., 1991). O *benchmark* CG, por exemplo, contém acessos irregulares à memória, enquanto em FT, todas as *threads* precisam se comunicar com todas as outras. O conjunto de *benchmarks* Rodinia é composto por aplicações e kernels de diferentes domínios, tais como processamento de sinais, simulação de física, processamento de imagem, mineração de dados, álgebra linear, entre outros. Rodinia é projetado para sistemas heterogêneos, por isso oferece implementações em OpenMP, OpenCL e CUDA. A versão em OpenMP foi utilizada neste trabalho.

Os conjuntos de entradas dos *benchmarks* no NAS são organizados em classes. As classes A, B e C são consideradas entradas de problemas de teste padrão, sendo a classe A a menor delas. Considerando-se que estes conjuntos de entradas são utilizados para avaliação de arquiteturas paralelas de computação de alto desempenho, pode-se dizer que a classe A oferece um tamanho de entrada significativo para sistemas embarcados. O *benchmark* FT, por exemplo, gera erro e não pode ser executado com a classe B no sistema utilizado neste trabalho. Por estes motivos, optou-se por utilizar as entradas da classe A para os *benchmarks* do NAS (CG, FT, LU e SP). Seguiu-se esta mesma linha na escolha das entradas para os demais *benchmarks*. Para os *benchmarks* do Rodinia, utilizou-se as entradas padrões que vêm no *benchmark*. Apenas a quantidade de iterações do HotSpot e SRAD foram alteradas, respectivamente, para 2000 e 20 iterações. Para o algoritmo FFT, o valor de 22 pode ser considerado médio para sistemas embarcados, pois valor de

entrada 24 não executa, pois ocorre erro. Pode-se considerar, então, que os conjuntos de entradas utilizados nos experimentos são de tamanho médio a grande.

4.2 Configuração dos experimentos

Os experimentos foram realizados na placa de desenvolvimento NanoPi-M3 da FriendlyARM (FRIENDLYARM, 2017). Esta placa é equipada com um SoC (do inglês *System-on-Chip*) da Samsung (S5P6818) que possui processador ARM Cortex-A53 Octa-Core. As principais características da placa NanoPi-M3 são apresentadas na Tabela 4.2. Os oito núcleos de processamento estão organizados em dois *socktes*, mas dentro de um único chip.

Tabela 4.2: Principais características da placa NanoPi M3

Característica	**Valor**
SoC	Samsung S5P6818
Processador	ARM Cortex-A53
Arquitetura	ARMv8-A
Frequência	1.4GHz
# Sockets	2
# CPU Cores	8 (4 em cada Socket)
Cache L1 de instruções (privada)	8 x 32KB
Cache L1 de dados (privada)	8 x 32KB
Cache L2 (compartilhada)	1 MB (2 x 512KB)
Memória RAM	1GB

Fonte: O Autor

Como sistema operacional foi utilizado o Linux nanopim3 4.11.12-s5p6818, fornecido pela Armbian (ARMBIAN, 2017). Optou-se por utilizar o Linux fornecido pela Armbian pois o disponibilizado pela fornecedora da placa (FriendlyARM) rodava apenas em modo 32-bit, não sendo atualizado para a arquitetura ARMv8, a qual permite modo de 64-bit. Outro fator determinante na escolha foi a disponibilidade do arquivo DTB (do inglês *Device Tree Blob*), que descreve o layout de hardware do sistema para o *kernel* do Linux e é necessário para o correto funcionamento do PERF (usado para coletar contadores de hardware). Os eventos de hardware são utilizados neste trabalho para estimar o consumo de energia em tempo de execução (mais detalhes na Seção 6.1.3) e também para ajudar a compreender o comportamento do desempenho e consumo de energia das aplicações. O *kernel* do Linux fornecido pela Armbian apenas não estava configurado para

permitir o uso do PERF. Por isto foi necessário realizar alteração no *kernel*, habilitando as configurações CONFIG_PERF_EVENTS e CONFIG_HW_PERF_EVENTS.

As aplicações foram compiladas com GCC versão 5.4.0 utilizando *flag* de otimização -O2. A *flag* -O2 oferece um grau de otimização similar a *flag* de otimização -O3 ao mesmo tempo que se preocupa com o tamanho do programa. O fator determinante para a escolha da *flag* de otimização -O2 foi o fato da aplicação FT não rodar com 8 *threads* quando compilada com *flag* -O3 (a placa se desliga durante a execução, possivelmente devido ao aquecimento do sistema). Optou-se por não determinar localidade das *threads*, ou seja, não atribuir afinidade de *threads* à núcleos específicos. Os experimentos foram realizados com DVFS em modo *ondemand*. Cada experimento foi repetido 30 vezes (onde os resultados se estabilizaram). A média aritmética destas execuções será apresentada nos resultados.

Alguns experimentos também foram realizados com as aplicações compiladas com *flag* de otimização -O3 e com afinidade de *threads* atribuídas aos núcleos de processamento (configurando a variável de ambiente GOMP_CPU_AFFINITY com o valor *0,1,2,3,4,5,6,7*). Contudo, estes experimentos foram rodados poucas vezes (apenas 3 vezes). Para fins de documentação estes resultados foram adicionados no Apêndice A. Sendo assim, apenas os resultados das aplicações compiladas com *flag* de otimização -O2 e execução sem afinidade de *threads* aos núcleos serão apresentados nas seções a seguir, pois apresentaram os melhores resultados.

O tempo de execução que será apresentado nos resultados corresponde ao tempo informado pelo *benchmark*. Para algumas aplicações, este tempo não corresponde ao tempo total. Isso porque alguns *benchmarks* apresentam como resultado apenas o tempo do *kernel* da aplicação. Contudo, o consumo de energia relatado refere-se ao consumo energético total da aplicação (não apenas o *kernel*), devido à medida externa que é realizada por meio do circuito que será descrito na seção a seguir.

Para ajudar a compreender o desempenho e consumo energético das aplicações quando variado o número de *threads* utilizadas em sua execução, foram coletados os contadores de eventos de hardware durante os experimentos da exploração de espaço de projeto. Processadores ARM com arquitetura ARMv8 permitem que sejam coletados até 6 contadores simultaneamente. Os eventos de hardware que foram coletados e serão apresentados nos resultados do espaço de exploração de projeto (Capítulo 5) são apresentados na Tabela 4.3. Para coletar as informações dos contadores de hardware utilizou-se a biblioteca PAPI (do inglês *Performance Application Programming Interface*) (MUCCI et

al., 1999) versão 5.5.1. O PAPI permite o acesso aos contadores de eventos de hardware inserindo chamadas às funções no código fonte da aplicação. Internamente, o PAPI faz uso da ferramenta PERF do Linux.

Tabela 4.3: Contadores de eventos de hardware

Evento	**Contador de Hardware**
Instrução executada	INST_RETIRED
Acesso à cache L1 de instruções	L1I_CACHE_ACCESS
Acesso à cache L1 de dados	L1D_CACHE_ACCESS
Acesso à cache L2	L2D_CACHE_ACCESS
Acesso à memória principal	L2D_CACHE_REFILL
Ciclos	CPU_CYCLES

Fonte: O Autor

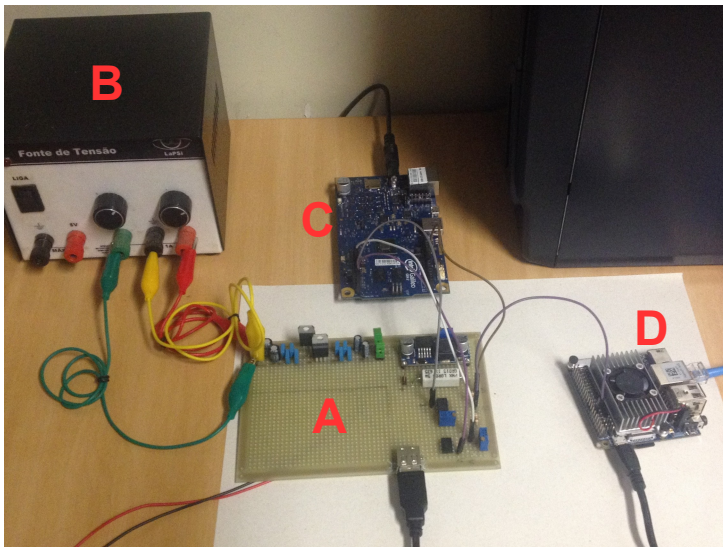
4.3 Circuito para medir o consumo de energia

Nesta seção é descrito o circuito utilizado para mensurar o consumo de energia dos experimentos realizados. Este circuito foi desenvolvido para adquirir amostras de tensão de alimentação e corrente que circula através de um resistor *shunt* para calcular a potência consumida nos experimentos pela placa NanoPi M3. A Figura 4.1 apresenta este circuito (A), conectado a fonte de tensão (B), uma placa Intel Galileo Gen2 (C) e a placa NanoPi-M3 (D). O circuito possui um resistor de 0,015 Ohms com precisão de 1% colocado em série com o circuito a ser medido.

Para leitura das amostras de corrente e tensão utilizou-se a placa Intel Galileo Gen2 (INTEL, 2017), que foi conectada através das portas de entradas analógicas A3 e A5 ao circuito a ser medido. A entrada A3 foi conectada entre o resistor e a fonte de alimentação, representando a queda de tensão sobre ele. O valor lido por A3 é utilizado para o cálculo da corrente. Para amplificar o sinal lido por esta porta utilizou-se o amplificador INA 114 AP. Para calibrar a fórmula para conversão dos dados lidos utilizou-se, no lugar da placa, um resistor com resistência conhecida (47 ohms). A entrada A5 foi conectada entre o circuito alvo e a fonte de alimentação, para receber a queda de tensão sobre o circuito. As leituras foram programadas para serem realizadas em intervalos de 20 milissegundos, obtendo-se assim 50 amostras por segundo.

Durante a execução dos experimentos são lidos os valores de corrente e tensão, pelas portas analógicas A3 e A5, e os valores lidos são gravados em cartão SD, para

Figura 4.1: Circuito para mensurar consumo de energia



Fonte: O Autor

posterior leitura e conversão dos dados (analógico para digital). Do consumo de potência medido foi descontado a potência de *idle* da placa (1,86 Watts). Isso porque usando uma forma de medida que é externa ao circuito, a potência calculada com base nos valores de tensão e corrente lidos refere-se ao consumo de potência de toda a placa.

A entrada digital 8 do Intel Galileo Gen2 foi conectada a um pino GPIO da placa NanoPi M3 para comunicar o início e finalização da execução de uma aplicação durante os experimentos, de forma a sincronizar a gravação do *log* das leituras de corrente e tensão. Posterior às execuções, os valores gravados em *log* foram convertidos de analógico para digital e a potência de cada amostra foi calculada usando os valores convertidos. As portas analógicas do Intel Galileo Gen2 oferecem resolução de 12 bits no conversor analógico digital (ARDUINO, 2017). Isto permitiu obter-se resolução de 0,73mV para corrente (usando referencia de 3V) e 1,22mV para os valores de tensão (com referencia de 5V). Para calcular o consumo geral de energia, aplicou-se integral nos valores de potência.

Embora o tempo de execução relatado por alguns *benchmarks* considere apenas o tempo de execução do *kernel* principal da aplicação (não considerando tempo de inicialização), a energia consumida que será relatada nos resultados considera o consumo de toda a execução da aplicação.

5 EXPLORAÇÃO DE ESPAÇO DE PROJETO

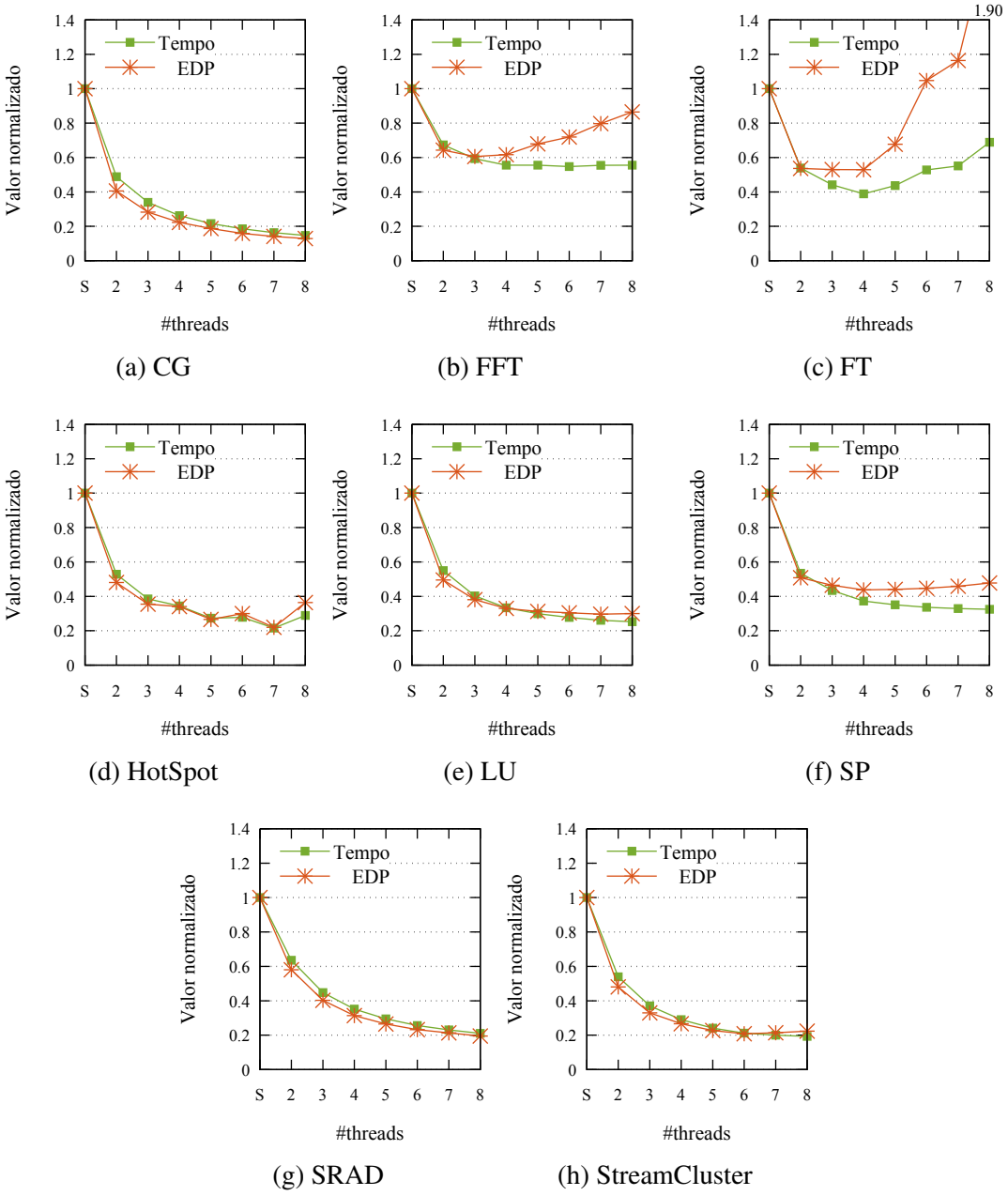
Trabalhos anteriores disponíveis na literatura mostram que as aplicações paralelas nem sempre atingem seu melhor desempenho ou eficiência energética fazendo uso do máximo número de *threads* possível, ou seja, utilizando todos os núcleos disponíveis no processador. Esta constatação foi baseada em aplicações paralelas quando executadas em processador de propósito geral (GPP, do inglês *General Purpose Processors*) ou servidor. Entretanto, neste capítulo, será apresentado o espaço de exploração de projeto existente quando as aplicações paralelas são executadas em um processador embarcado com 8 núcleos de processamento. Busca-se com isso, mostrar que por meio da limitação do número de *threads* utilizadas para executar uma aplicação paralela, em processador embarcado, é possível obter melhores resultados de eficiência energética, EDP e, em alguns casos, até mesmo desempenho.

Cada aplicação foi executada 30 vezes com cada número de *threads* possível (2 à 8 *threads*). A média aritmética destas execuções é apresentada nos gráficos normalizada pela média dos resultados obtidos pela versão sequencial de cada aplicação. A média dos resultados não normalizada e o desvio padrão estão documentados no Apêndice B.

Os resultados de tempo de execução e EDP de cada aplicação são apresentados na Figura 5.1 e os resultados de consumo de energia são apresentados na Figura 5.2. Os resultados de consumo de energia são apresentados em gráficos separados dos resultados de tempo de execução e EDP devido à diferença de escalas. Desta forma, facilita-se a visualização do impacto no aumento no número de *threads* em cada uma das métricas. Vale notar que quanto menor os valores nos gráficos (tempo de execução, consumo de energia e EDP), melhores são os resultados.

Para compreender os resultados de tempo de execução e consumo de energia de cada aplicação, foram também coletados os eventos de hardware das execuções com cada número de *threads*. Quando uma aplicação é paralelizada, mais instruções são necessárias para o gerenciamento das *threads*. Além disso, devido ao modelo de comunicação via memória compartilhada, o número de acessos ao último nível de cache e à memória principal tendem a aumentar. O aumento no número de instruções reflete na baixa aceleração gerada pela paralelização e consome mais energia dinâmica. Outro ponto importante é o aumento dos acessos às memórias compartilhadas. A potência consumida por estas memórias é maior, e por estarem localizadas mais distantes do processador, o tempo de acesso também é maior, impactando tanto no desempenho quanto na energia consumida.

Figura 5.1: Espaço de exploração de projeto: tempo de execução e EDP



(a) CG (b) FFT (c) FT

(d) HotSpot (e) LU (f) SP

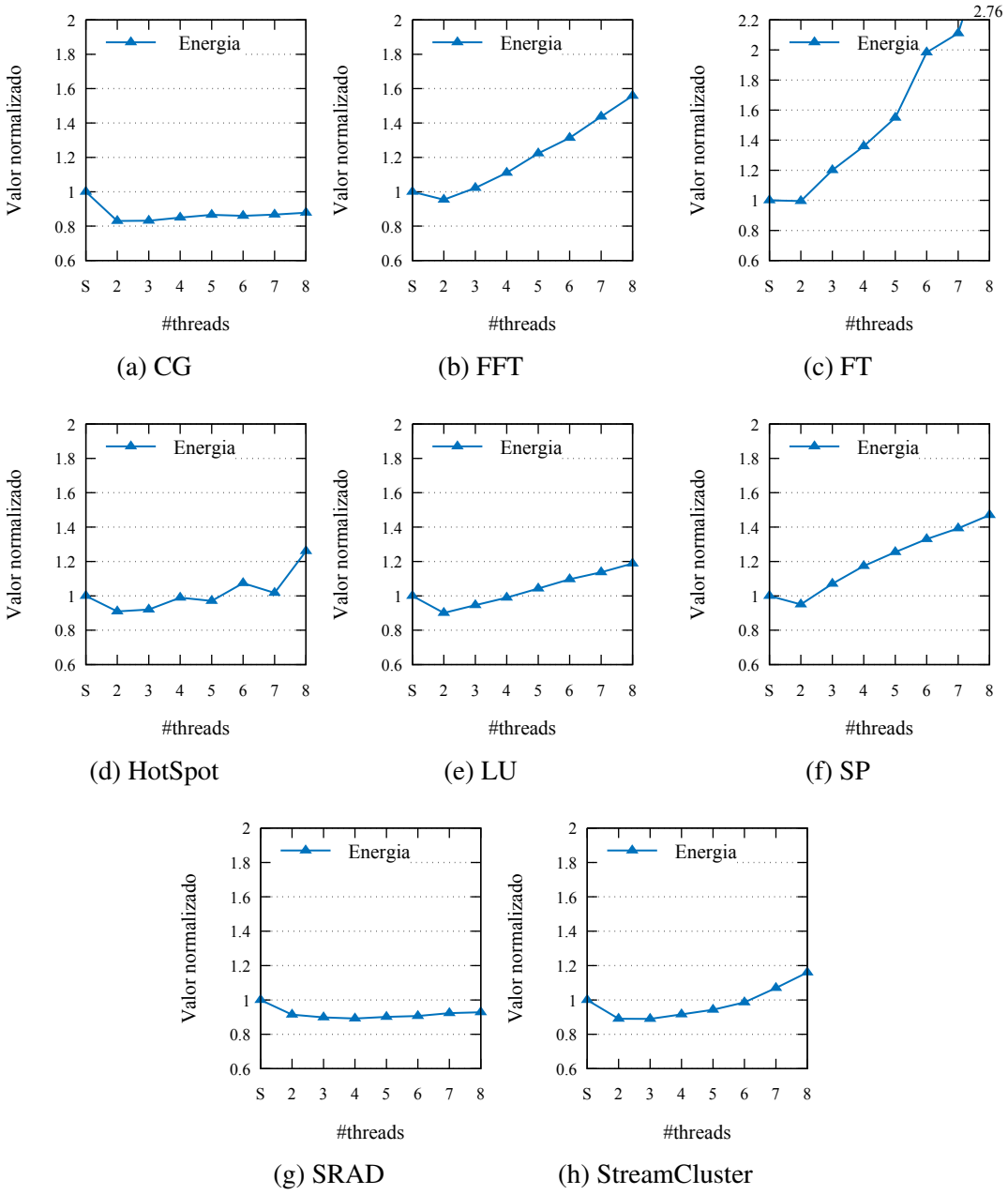
(g) SRAD (h) StreamCluster

Fonte: O Autor

Os eventos de hardware gerados por cada aplicação quando variado o número de *threads* utilizadas na execução são apresentados na Figura 5.3. São apresentadas as contagens de instruções executadas (Instr.), acessos às memórias *cache* L1 de instruções (Ac. L1 I) e L1 de dados (Ac. L1 D), *cache* L2 (Ac. L2) e memória principal (Ac. RAM).

O tempo de execução de uma aplicação é refletido não somente pela quantidade de instruções executadas mas também pela quantidade de instruções executadas por ciclo (IPC – instruções por ciclo). Quando uma aplicação é paralelizada, espera-se que a quantidade IPC aumente devido ao aumento no número de unidades de processamento

Figura 5.2: Espaço de exploração de projeto: consumo de energia

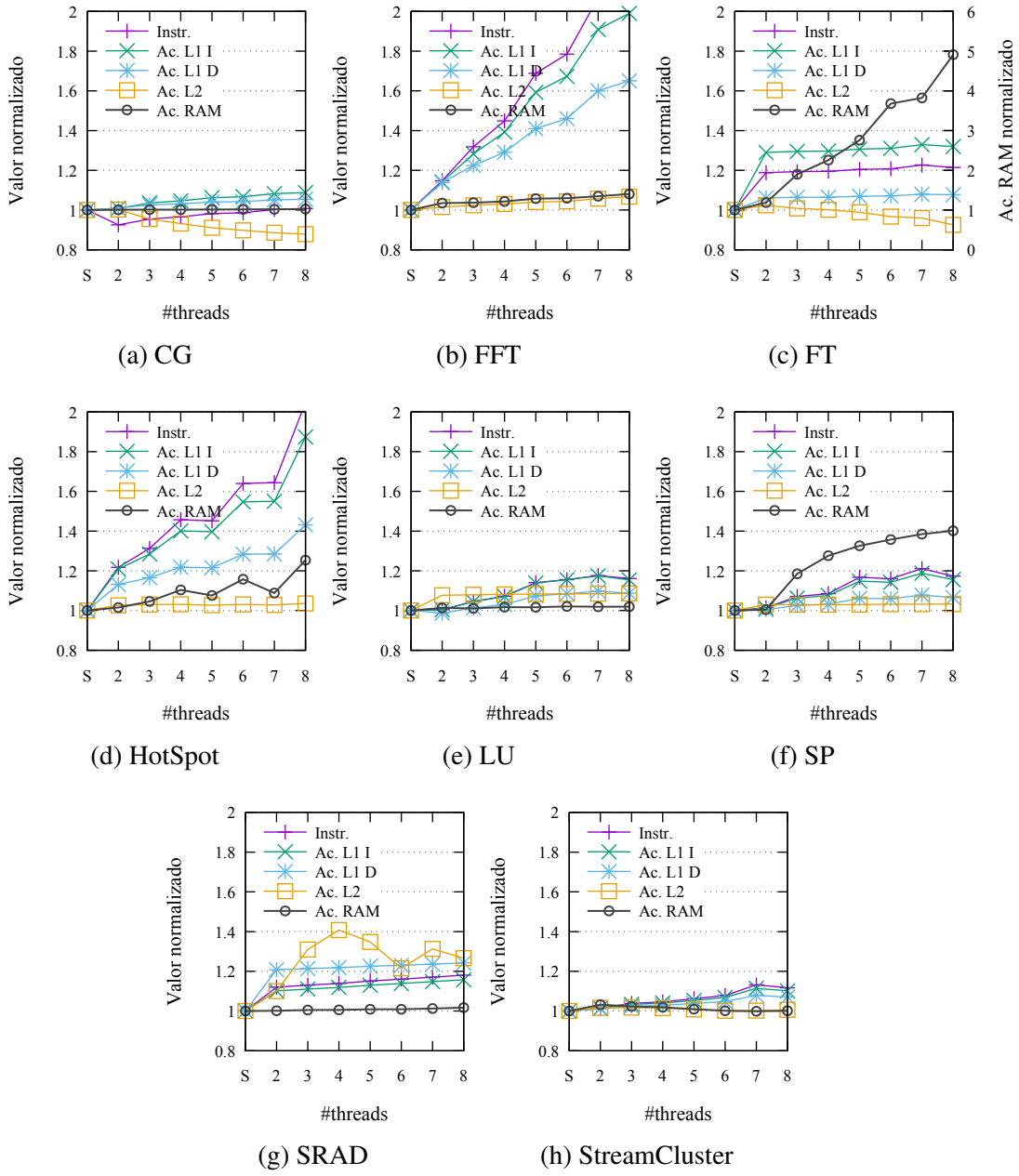


Fonte: O Autor

utilizadas. Entretanto, o aumento nas IPC nem sempre é linear ao aumento no número de *threads*. Além disso, o aumento nas IPC nem sempre significa maior aceleração. Enquanto na Figura 5.3 são apresentados o total de eventos gerados na execução, na Figura 5.4 apresenta-se as taxas, ou seja, a quantidade de eventos que ocorrem por ciclo. Apresenta-se a quantidade de IPC, a taxa de acesso à memória cache L2 (tx L2) e a taxa de acesso à memória principal (tx RAM).

A seguir os resultados de tempo de execução, consumo de energia e EDP de cada *benchmark* serão discutidos separadamente. O impacto das taxas e da quantidade de even-

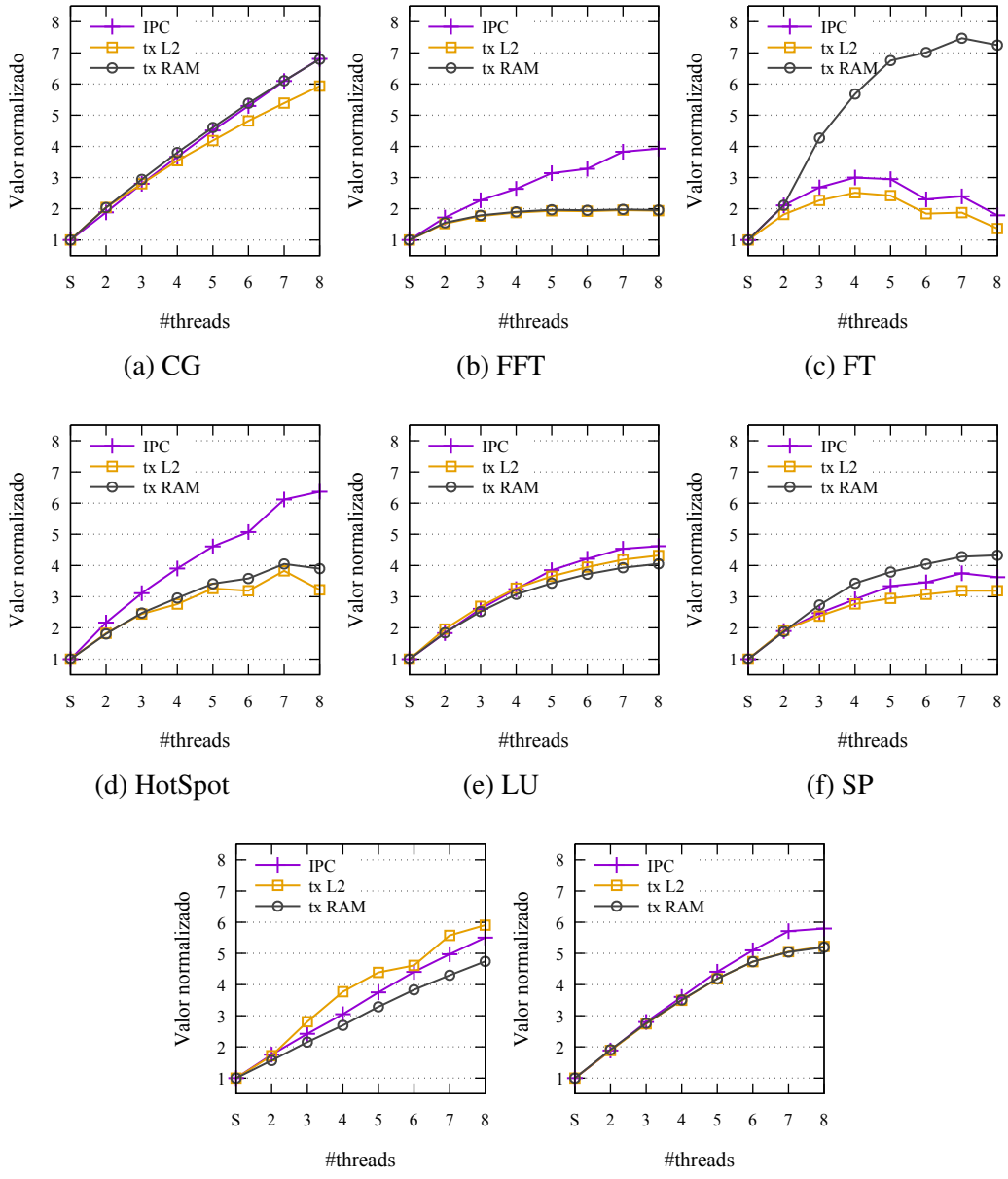
Figura 5.3: Espaço de exploração de projeto: eventos de hardware



Fonte: O Autor

tos gerados pela paralelização no desempenho e no consumo de energia de cada aplicação também é discutido. Segue-se uma apresentação pela ordem alfabética das aplicações. Para os três primeiros *benchmarks* (CG, FFT e FT) serão apresentados os gráficos da potência consumida ao longo do tempo de execução. Estes *benchmarks* apresentam três situações bem distintas resultadas pelo aumento no número de threads: (i) aumento na potência mas, como o tempo de execução é reduzido, gera-se economia de energia; (ii) aumento na potência mas sem melhora no desempenho, e assim, gasta-se energia desnecessariamente; (iii) além de aumentar a potência, o aumento no número de *threads* causa

Figura 5.4: Espaço de exploração de projeto: taxas de eventos de hardware



(a) CG (b) FFT (c) FT

(d) HotSpot (e) LU (f) SP

(g) SRAD (h) StreamCluster

Fonte: O Autor

degradação do desempenho e, consequentemente, tem-se um elevado consumo energético.

5.1 CG

O *benchmark* CG apresentou ótima escalabilidade (Figura 5.1a), fazendo com que o tempo de execução diminua conforme o aumento no número de *threads*. Observa-se

na Figura 5.4a que o aumento nas taxas de eventos do CG é quase linear ao aumento no número de *threads*, contribuindo assim para a redução no tempo de execução. CG executa aproximadamente 85% mais rápido com 8 *threads*, quando comparado com a versão sequencial.

Comparando as contagens dos eventos de hardware da execução paralela com a versão sequencial (Figura 5.3a), pode-se notar que a quantidade de acessos à memória principal é praticamente a mesma e a quantidade de acessos à memoria *cache* L2, que é compartilhada, diminui. Isso porque CG apresenta acesso irregular à memória, não havendo dependência de dados. Desta forma, cada núcleo trabalha mais com sua memória cache privada (*cache* L1), que apresenta menor custo de tempo de acesso e potência.

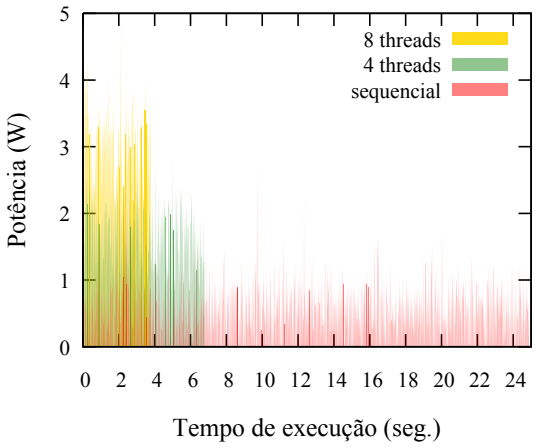
Esses fatores contribuem para que a versão paralela do *benchmark* CG apresente maior eficiência energética que a sequencial (Figura 5.2a). CG executado com 8 *threads* consome 12,18% menos energia que a versão sequencial. Como resultado, o EDP de CG com 8 *threads* é 87,11% menor que a versão sequencial (Figura 5.1a). Com isto, para o *benchmark* CG, 8 *threads* é o melhor número para otimizar tempo de execução e EDP. Entretanto, se o objetivo fosse exclusivamente reduzir o consumo de energia, 2 *threads* seria melhor, pois consumiria 5,49% menos energia que 8 *threads*.

Na Figura 5.5 são apresentadas as amostras de consumo de potência coletadas durante a execução da aplicação CG com 8 e 4 *threads* e sequencialmente. A versão sequencial de CG consome em média 0,47 Watts por segundo, enquanto a versão paralela consome 1,49 e 2,71 Watts, com 4 e 8 *threads*, respectivamente. Observa-se que embora a execução com 4 ou 8 *threads* gere maior consumo de potência, quando comparado com a versão sequencial, o tempo de execução diminui, promovendo, assim, maior eficiência energética (conforme observado na Figura 5.2a).

5.2 FFT

O *benchmark* FFT, ao contrário de CG, não apresenta redução no tempo de execução conforme o aumento no número de *threads* (Figura 5.1b). Quando FFT é executado com 2 *threads*, por exemplo, tem-se aceleração de 32,63% comparada com a versão sequencial da aplicação. Quando aumentado o número de *threads* para 4, o tempo de execução tem redução de apenas 17,57% comparado com a execução usando 2 *threads*. Seguindo assim, após 4 *threads* a aplicação não apresenta melhora no tempo de execução quando aumentado o número de *threads*. No entanto, mais núcleos trabalhando resultam

Figura 5.5: CG – potência consumida ao longo do tempo de execução



Fonte: O Autor

em maior potência consumida. Desta forma, ao aumentar o número de *threads* sem obter melhora no desempenho gera-se um consumo de energia desnecessário (Figura 5.2b). Quando FFT é executada com 8 *threads* o consumo de energia aumenta em 55,8% comparado com a execução sequencial. Este aumento no consumo de energia sem redução no tempo de execução faz com que o EDP também passe a aumentar (Figura 5.1b).

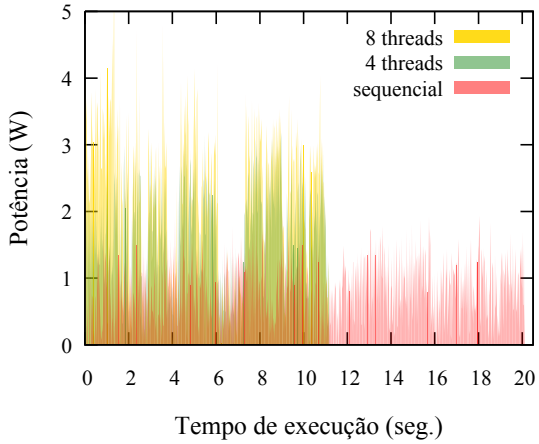
Pode-se observar que as taxas de eventos do FFT não apresentam aumento linear ao aumento no número de *threads* (Figura 5.4b). Além disso, a quantidade de instruções executadas e acessos ao primeiro nível de memória *cache* aumenta à medida que aumenta a quantidade de *threads*. Com 8 threads, por exemplo, são executadas 2,16 vezes mais instruções, comparado com a versão sequencial. A versão paralela de FFT também gera mais acessos às memórias compartilhadas (*cache* L2 e RAM). Com isto, o aumento no número de núcleos utilizados para executar a aplicação faz com que haja maior consumo de potência dinâmica (relacionada ao aumento das atividades) e maior potência estática (mais núcleos trabalhando sem reduzir o tempo de execução) gerando, assim, um elevado consumo de energia (Figura 5.2b).

Devido à limitação na escalabilidade, para a aplicação FFT o melhor número de *threads* para otimizar tempo de execução é 6 *threads* (45,25% mais rápido que a versão sequencial e 1,41% mais rápido que a configuração padrão de 8 *threads*). Se o objetivo fosse consumir menos energia, a melhor escolha seria a utilização de 2 *threads*. Contudo, a melhor opção para otimizar EDP na aplicação FFT é usar 3 *threads*. FFT executada com 3 *threads* apresenta EDP 29,92% menor que a execução com 8 *threads*.

A Figura 5.6 ilustra o impacto do aumento no número de *threads* na potência dis-

sipada pela aplicação FFT. Neste gráfico são apresentadas as amostras de consumo de potência coletadas durante a execução da aplicação FFT com 8 e 4 *threads* e sequencialmente. A versão sequencial de FFT consome em média 0,73 Watts por segundo, enquanto a versão paralela consome 1,45 e 2,03 Watts, com 4 e 8 *threads*, respectivamente. A potência dissipada com 8 *threads* é 40% maior que a com 4 *threads*. Entretanto, observa-se que o tempo de execução continua o mesmo, gerando um consumo de energia desnecessário.

Figura 5.6: FFT – potência consumida ao longo do tempo de execução



Fonte: O Autor

5.3 FT

O *benchmark* FT, assim como a aplicação FFT, também não apresenta boa escalabilidade. Até certo ponto, o aumento no número de *threads* proporciona aceleração no tempo de execução (Figura 5.1c). O tempo de execução de FT é 61,09% menor com 4 *threads*, quando comparado com versão sequencial. Contudo, depois deste ponto, o aumento no número de *threads* causa degradação do desempenho. Quando FT é executado com 8 *threads*, por exemplo, seu tempo de execução aumenta 76,95% comparado com a execução utilizando 4 *threads*.

Observando os eventos de hardware na Figura 5.3c, pode-se notar que o aumento no número de *threads* para executar o *benchmark* FT causa aumento significativo na quantidade de acessos à memória principal. FT executada com 8 threads, por exemplo, gera 4,9 vezes mais acessos à memória principal comparado com a versão sequencial. O au-

mento na quantidade de acessos à memória principal fazem com que aumento na taxa de acessos (Figura 5.4c) não gere melhora no desempenho da aplicação.

As quantidades IPC e taxa de acesso à cache L2 (Figura 5.4c) aumentam até 4 *threads*. Depois disso, as taxas começam a cair. Quanto mais acessos às memórias distantes, como a memória principal, maior é o tempo de espera fazendo com que diminua a quantidade de instruções que são executadas por ciclo. Além do elevado aumento nos acessos à memória principal, a paralelização de FT também requer que mais instruções sejam executadas (Figura 5.3c). Quanto mais instruções executadas e acessos ao sistema de memória, maior será o consumo de energia dinâmica.

Além do acesso à memória principal ser mais lento, esta também consome mais potência. Com isto, mais núcleos processando, sem ganhos em desempenho, e mais acesso à memória RAM fazem com que a energia consumida aumente consideravelmente (Figura 5.2c) conforme o aumento no número de *threads*. FT quando executado com 8 *threads*, por exemplo, consome 2,76 vezes mais energia que a versão sequencial.

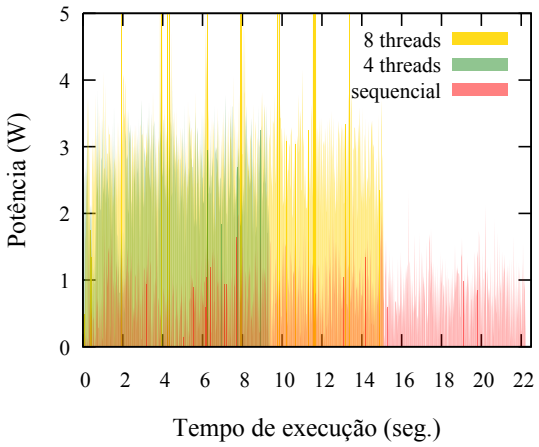
Em resumo, para o *benchmark* FT, 4 *threads* é o melhor número para otimizar tempo de execução e EDP (Figura 5.1c). FT executado com 4 *threads* apresenta 43,49% e 72,21% de redução do tempo de execução e EDP, respectivamente, comparado com a execução com o máximo número de *threads*. Se o objetivo fosse apenas reduzir o consumo de energia, a execução com 2 *threads* seria a melhor opção para FT, resultando em 64,01% de economia de energia comparada com a execução utilizando 8 *threads*.

Na Figura 5.7 são apresentadas as amostras de consumo de potência coletadas durante a execução do *benchmark* FT com 8 e 4 *threads* e sequencialmente. A versão sequencial de FT consome em média 0,70 Watts por segundo, enquanto a versão paralela consome 2,36 e 2,82 Watts, com 4 e 8 *threads*, respectivamente. Observa-se que a potência dissipada pelo uso de 8 *threads* não aumenta muito em relação ao uso de 4 *threads* (19,49%). Isto porque as taxas de eventos começam a cair. Contudo, mesmo que a potência não tenha um elevado aumento, o consumo de energia do uso de 8 *threads* é maior devido à degradação no desempenho.

5.4 HotSpot

O *benchmark* HotSpot apresenta desbalanceamento de carga, o que causa a variação observada no desempenho (Figura 5.1d) e no consumo de energia (Figura 5.2d) conforme o aumento no número de *threads*. HotSpot é executado 72,7% mais rápido com

Figura 5.7: FT – potência consumida ao longo do tempo de execução



Fonte: O Autor

5 *threads*, quando comparado com a versão sequencial. No entanto, quando HotSpot é executado com 6 *threads*, seu desempenho apresenta uma queda de 1,93% (comparado com o uso de 5 *threads*). Esta queda, no entanto, não se reflete na execução com 7 *threads*. HotSpot é executado 20,98% mais rápido com 7 *threads*, comparado com o uso de 5 *threads*. Ao aumentar o número de *threads* para 8, o desempenho retorna a cair. O consumo de energia também apresenta variação (Figura 5.2d).

Analisando a contagem de eventos de hardware do HotSpot (Figura 5.3d), as execuções com quantidade par de *threads* fazem mais acessos a memória principal comparadas com os números ímpares. A quantidade de instruções executadas também apresenta comportamento semelhante. Por exemplo, 4 *threads* executam mais instruções que o uso de 3 *threads*; 5 *threads* mantém a mesma quantidade de instruções executadas de 4 *threads*; 6 *threads* executa ainda mais instruções, enquanto 7 *threads* mantém a mesma quantidade de instruções executadas por 6 *threads*. Na sequencia, 8 *threads* aumenta ainda mais a quantidade de instruções executadas. Mesmo que exista esta variação no número de eventos, as taxas aumentam (Figura 5.3d). Desta forma, mesmo que 7 *threads* execute a mesma quantidade de instruções de 6 *threads*, as IPC aumentam, resultando assim na variação observada do desempenho.

Observa-se também neste *benchmark*, como os acessos à memória principal (Figura 5.3d) influenciam no desempenho e no consumo de energia da aplicação. Os acessos às memórias e as instruções extras executadas fazem com que o consumo de energia da aplicação aumente em relação a versão sequencial após 5 *threads*. Pode-se notar a semelhança no padrão dos eventos com o consumo de energia (Figura 5.2d) e no tempo de

execução (Figura 5.1d).

As IPC do HotSpot apresentam boa escalabilidade conforme o aumento no número de *threads* (Figura 5.4d). Mesmo havendo o aumento dos eventos (Figura 5.3d), o aumento nas taxas fazem com que o desempenho da aplicação melhore. Pode-se notar que o aumento na taxa de acesso à memória *cache* L2 e a redução no número de acessos à memoria principal fazem com que 7 *threads* apresente o melhor desempenho para este *benchmark*.

Sendo assim, HotSpot apresenta melhor desempenho quando executado com 7 *threads*, sendo este também o melhor número para otimizar EDP (Figura 5.1d). Quando HotSpot é executado com 7 *threads* o tempo de execução é reduzido em 25,37% e o EDP em 39,61%, comparado a execução padrão de 8 *threads*. Assim como para os *benchmarks* FFT e FT, se reduzir o consumo de energia fosse o objetivo, a execução com 2 *threads* seria a melhor escolha. HotSpot executado com 2 *threads* consome 27,81% menos energia que a execução com 8 *threads*.

5.5 LU

Para o *benchmark* LU, o máximo número de *threads* apresentou o melhor desempenho (Figura 5.1e). Contudo, a sua escalabilidade não é linear ao aumento no número de *threads*. LU executado com 2, 3 ou 4 *threads* consome menos energia que a versão sequencial (Figura 5.2e). Observa-se que 2 *threads* reduz o consumo de energia, mas a medida que aumenta-se no número de *threads* o consumo volta a aumentar. Após 5 *threads* a energia consumida já ultrapassa a versão sequencial.

O uso de 2 *threads* reduz o tempo de execução de LU em 45,07% comparada com a execução sequencial (Figura 5.1e). Mesmo que o consumo de potência aumente de 0,58 Watts para 0,95 Watts, a redução do tempo de execução faz com que o consumo de energia seja menor com 2 *threads*, resultando em 9,92% de economia de energia em relação a versão sequencial (Figura 5.2e).

A paralelização do *benchmark* LU, causa leve aumento nos acessos às memórias compartilhadas – aproximadamente 8% e 2%, respectivamente, nos acessos à *cache* L2 e memória RAM (Figura 5.3e). Observa-se que após 4 *threads* a quantidade de instruções executadas e acessos às memórias *cache* L1 (instruções e dados) passam a aumentar, contribuindo como o aumento no consumo de energia (Figura 5.2e).

Analisando as taxas de eventos do *benchmark* LU (Figura 5.4e), pode-se notar que

o uso de 2 *threads*, por exemplo, executa 1,82 vezes mais IPC que a versão sequencial. No entanto, esta aceleração não continua a medida que aumenta-se o número de *threads*. O uso de 5 *threads*, por exemplo, provê um aumento de 18,36% nas IPC em relação ao uso de 4 *threads* e aumenta em média 0,3 Watts consumidos por segundo. Enquanto que o uso de 8 *threads* aumenta apenas 20,68% nas IPC em relação ao uso de 5 *threads*, mas consome 0,71 Watts à mais por segundo.

Pode-se notar, então, que a taxa dos eventos não aumenta linearmente e, assim, não gera a aceleração esperada pela paralelização. O uso de 8 *threads*, por exemplo, reduz o tempo de execução de LU em apenas 3,08% comparado com a execução com 7 *threads* (Figura 5.1e). No entanto, o aumento no número de núcleos dissipa mais potência. Desta forma, o consumo de energia aumenta (Figura 5.2e), tanto estática quanto dinâmica (devido ao aumento nos eventos observados na Figura 5.3e).

Em resumo, embora LU não apresente escalabilidade linear, o máximo número de *threads* ainda oferece o melhor desempenho para esta aplicação. O tempo de execução de LU reduz 74,60% com o uso de 8 *threads*, quando comparado com a versão sequencial, mas consome 12,63% mais energia. Devido limitação na escalabilidade e ao aumento no consumo de energia, o melhor EDP é obtido com o uso de 7 *threads*, mas é apenas 1,23% melhor que o uso de 8 *threads*. Para economia de energia, 2 *threads* é o melhor número, apresentando 24,2% de redução comparado com a configuração padrão de 8 *threads*.

5.6 SP

O *benchmark* SP também apresentou melhor desempenho com 8 *threads* (Figura 5.1f). Contudo, a sua escalabilidade é ainda mais limitada que a observada no *benchmark* LU. O uso de 8 *threads* provê 67,53% de redução no tempo de execução em relação a versão sequencial. Contudo, observa-se que após 4 *threads* há pouca melhora no tempo conforme o aumento no número de *threads*. A redução do tempo de execução do uso de 8 *threads* em relação ao uso de 5 *threads*, por exemplo, é de apenas 7,45%. No entanto, a potência média dissipada aumenta 26,82%, resultando no aumento de 17,17% no consumo de energia, quando comparado a execução de 8 *threads* com 5 *threads* (Figura 5.2f). Com isto, a execução com o máximo número de *threads*, embora ofereça o melhor desempenho, causa 47,02% de aumento no consumo de energia em relação à execução sequencial.

Analisando a escalabilidade das taxas de eventos (Figura 5.4f), observa-se que SP

executado com 2 *threads* tem aumento de 89,74% nas IPC em relação a versão sequencial. Em conjunto a isto, não observa-se aumento significativos nos eventos (Figura 5.3f). Este aumento nas taxas sem aumento na quantidade de eventos resulta na aceleração da aplicação e redução no consumo de energia. Com isto, quando SP é executado com 2 *threads*, tem-se 4,97% de economia de energia em relação à versão sequencial, enquanto ainda reduz o tempo de execução em 46,54%.

No entanto, após 2 *threads*, o aumento no número de *threads* gera mais eventos (Figura 5.3f). Observa-se principalmente o aumento na quantidade de acessos a memória RAM, chegando à 40% quando comparado 8 *threads* com a versão sequencial. Este aumento impacta no consumo de energia (Figura 5.2f). Embora o uso de 2 *threads* tenha apresentado ótimas taxas de eventos, estas não continuam a escalar conforme o aumento no número de *threads* (Figura 5.4f). SP executado com 8 *threads*, por exemplo, executa apenas 24,25% mais IPC quando comparado com 4 *threads*, e apresenta queda de 3,27% nas IPC quando comparado com 7 *threads*.

Em resumo, devido à escalabilidade limitada, o melhor EDP para SP é resultado da execução com 4 *threads*, sendo 8,39% menor que o EDP resultado da execução com 8 *threads*. Embora 4 *threads* seja melhor para EDP, 8 *threads* ainda é o melhor número para desempenho. A execução com 4 *threads* é 14,73% mais lenta que a execução com 8 *threads*, mas oferece 20,15% de economia de energia. Se o objetivo fosse somente economizar energia, 2 *threads* seria o ideal, apresentando 35,36% e 19,05% menos consumo de energia quando comparado, respectivamente, com 8 e 4 *threads*.

5.7 SRAD

O *benchmark* SRAD, assim como o CG, apresenta o melhor desempenho e EDP com o uso de 8 *threads* (Figura 5.1g). Nota-se também que, assim como o *benchmark* CG, o consumo de energia da versão paralela é menor que o consumo da versão sequencial (Figura 5.2g). O uso de 8 *threads*, por exemplo, consome 7,15% menos energia que a versão sequencial, enquanto provê um aumento de 78,99% no desempenho, resultado na redução de 80,48% no EDP em relação a versão sequencial.

Analisando os eventos de hardware de SRAD (Figura 5.3g), observa-se que o número de instruções executadas aumenta na versão sequencial, mas este aumento não segue conforme o aumento no número de *threads*. Com o uso de 2 *threads*, são executadas 12,02% mais instruções que a execução sequencial. Contudo, o uso de 8 *threads*, gera

apenas 5,4% mais de instruções em relação ao uso de 2 *threads*. O acesso às *caches* L1 (instruções e dados) também segue este mesmo padrão das instruções executadas. Embora a quantidade de instruções executadas mantenha-se quase constante na versão paralela, as IPC (Figura 5.4g) aumentam de forma quase linear, sendo que 8 *threads* executa 5,5 vezes mais IPC que a versão sequencial.

O acesso à *cache* L2 é que apresenta maior variação (Figura 5.3g). Contudo, o aumento no número de *threads* também aumenta a taxa de acesso à esta, permitindo 5,9 vezes mais acessos por ciclo com 8 *threads*, comparando com a versão sequencial. Com relação aos acessos à memoria principal, observa-se que, 8 *threads* realiza apenas 1,72% mais acessos à memória RAM que a versão sequencial, enquanto a taxa de acesso aumenta quase linearmente (Figura 5.4g). Como as IPC e taxas de acessos às memórias compartilhadas aumentam conforme o aumento no número de *threads*, o tempo de execução de SRAD diminui, fazendo com que o consumo de energia estática diminua ao ponto de equiparar-se com o aumento na energia dinâmica. Desta forma, o consumo de energia deste *benchmark* não sofre muita variação com o aumento no número de *threads*.

Em resumo, para o *benchmark* SRAD, devido à sua boa escalabilidade, 8 *threads* além de ser o melhor número para desempenho, é também o melhor para EDP. Mesmo com pouca variação no consumo energético (considerando a versão paralela), 4 *threads* é a opção mais eficiente energeticamente, sendo 3,92% melhor que a configuração padrão (8 *threads*).

5.8 StreamCluster

O *benchmark* StreamCluster apresenta boa escalabilidade, tendo sua melhor performance com 8 *threads* (Figura 5.1h). StreamCluster é executado 80,71% mais rápido com 8 *threads*, quando comparado com a versão sequencial. Contudo, após 6 *threads*, o aumento no número de *threads* não produz redução significativa no tempo de execução. Quando StreamCluster é executado com 8 *threads*, o tempo de execução é apenas 8,83% menor que o tempo resultado da execução com 6 *threads*.

StreamCluster também apresenta economia de energia, em relação a versão sequencial, quando executado com menos de 6 *threads* (Figura 5.2h). Devido à escalabilidade quase linear até 3 *threads* nas taxas de eventos (Figura 5.4h) e o baixo aumento na quantidade de eventos (Figura 5.3h) pode-se observar a redução no consumo de energia. Com o uso de 3 *threads*, são processadas 2,8 vezes mais IPC que a versão sequencial, por

exemplo.

No entanto, após 3 *threads* a quantidade de instruções executadas e acessos às memórias *cache* L1 (instruções e dados) começam a aumentar (Figura 5.3h). Com 7 *threads*, por exemplo, são executadas 13% mais instruções que a versão sequencial. Em conjunto a isto, a aceleração quase linear que era observada até 3 *threads* passa a ser menor à medida que aumenta o número de *threads* (Figura 5.4h). Isto impacta no desempenho e no consumo de energia, que após 6 *threads* já é maior que o consumo da versão sequencial.

Mais núcleos processando dissipam maior quantidade de potência e, sem produzir o desempenho esperado, geram maior consumo de energia. Para executar o *benchmark* StreamCluster, o uso de 3 *threads*, por exemplo, consome 1,44 watts por segundo, enquanto 8 *threads* consome 3,62 watts. Desta forma, para o *benchmark* StreamCluster, o melhor número de *threads* para otimizar EDP é 6 *threads*, embora seja apenas 6,79% melhor que a configuração padrão de 8 *threads*. Quando o objetivo for otimizar eficiência energética, 3 *threads* é o melhor número, resultado em 23,26% e 9,69% de economia de energia, respectivamente, quando comparado com a execução com 8 e 6 *threads*.

5.9 Resumo do espaço de exploração de projeto

A Tabela 5.1 apresenta o melhor número de *threads* para cada uma das métricas: tempo de execução, consumo de energia e EDP. Apresenta-se também o benefício (coluna "ganho") que o uso do melhor número de *threads* pode gerar para a aplicação, em comparação com a configuração padrão do uso do máximo número de *threads* (8 *threads* neste trabalho). Para a métrica EDP apresenta-se, além do ganho em EDP, o impacto no consumo de energia e no tempo de execução.

A limitação do número de *threads* utilizados na execução das aplicações paralelas em sistemas embarcados pode até mesmo gerar ganhos em desempenho para algumas aplicações, com é o caso dos *benchmarks* FFT, FT e HotSpot (coluna "Tempo de execução"). Para a grande maioria das aplicações o uso do máximo número de *threads* oferece o melhor desempenho. Contudo, observa-se que, para todas as aplicações, o melhor número de *threads* para desempenho não é o melhor para economia de energia (coluna "Consumo de energia"). Por isto, faz-se o uso da métrica EDP.

EDP provê um balanço entre as métrica de consumo de energia e tempo de execução. O melhor número de *threads* para otimizar EDP provê economia de energia, com pequena perda de desempenho para os *benchmarks* FFT, LU, SP e StreamCluster. Para os

Tabela 5.1: Melhor número de *threads* para as diferentes métricas

	Tempo de execução		**Consumo de energia**		**EDP**	ganho em		
Benchmark	#threads	ganho	#threads	ganho	#threads	EDP	energia	tempo
CG	8		2	5,49%	**8**			
FFT	6	1,41%	2	38,77%	**3**	29,92%	34,38%	(-6,78%)
FT	4	43,49%	2	64,01%	**4**	72,21%	50,84%	43,49%
HotSpot	7	25,37%	2	27,81%	**7**	39,61%	19,27%	25,37%
LU	8		2	24,20%	**7**	1,23%	4,27%	(-3,18%)
SP	8		2	35,36%	**4**	8,39%	20,15%	(-14,73%)
SRAD	8		4	3,92%	**8**			
StreamCluster	8		3	23,26%	**6**	6,79%	15,03%	(-9,69%)

Fonte: O Autor

benchmarks FT e Hotspot, o melhor número de *threads* para EDP, além de proporcionar economia de energia, melhora também o desempenho. Sendo que os *benchmarks* CG e SRAD tem seu melhor EDP com 8 *threads*, não há economia de energia ou perda de desempenho.

Nos experimentos realizados e discutidos neste capítulo, as aplicações foram executadas com número de *threads* fixo, ou seja, o mesmo número de *threads* para todas as regiões paralelas. Entretanto, cada região paralela pode apresentar um comportamento diferente, e desta forma, o melhor número de *threads* pode variar de região para região. A adaptação dinâmica do número de *threads* proposta neste trabalho tenta encontrar o melhor número de *threads* específico para cada região paralela. A forma como a adaptação é realizada é descrita no capítulo a seguir.

6 ADAPTAÇÃO DINÂMICA DO NÚMERO DE THREADS

Baseado no estudo das possibilidades de otimização levantadas e apresentadas no capítulo anterior, desenvolveu-se uma biblioteca para realizar a adaptação do número de *threads* de aplicações paralelas OpenMP em tempo de execução, quando executadas em processadores ARM, com objetivo de otimizar EDP. Este capítulo destina-se a descrever esta biblioteca e apresentar os resultados obtidos com a sua utilização.

6.1 Biblioteca para adaptação

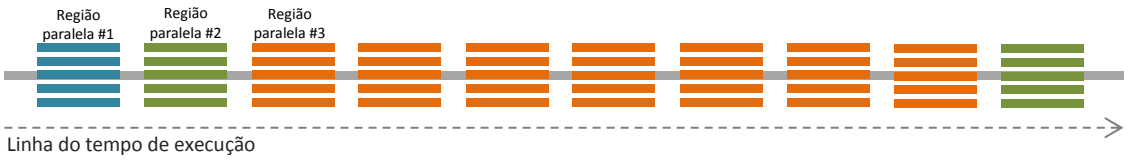
Nesta seção é apresentada a biblioteca proposta neste trabalho para adaptar o número de *threads* utilizados na execução de aplicações paralelas quando executadas em processador embarcado. Durante a execução da aplicação, sem qualquer informação prévia, é encontrado o número de *threads* que resulta no menor EDP, e a aplicação passa a ser executada com este número. Sendo que uma aplicação paralela pode conter mais de um trecho de código paralelo, é realizado o ajuste fino, ou seja, a adaptação do número de *threads* é realizada considerando cada região paralela individualmente, pois cada uma pode apresentar um comportamento diferente e assim o melhor número de *threads* também pode ser diferente. A adaptação proposta não requer modificação em hardware nem compilador específico.

6.1.1 Visão geral

Uma aplicação paralela pode conter uma ou mais regiões paralelas, ou seja, trechos de código que são executados em paralelo pelos múltiplos núcleos do processador. Durante a execução da aplicação, cada região paralela pode ocorrer uma única vez ou ser recorrente, ou seja, pode ser executada diversas vezes (como ilustrado na Figura 6.1). Esta recorrência se deve ao fato de que as aplicações podem conter iterações (laço de repetição externo) que fazem com que uma região paralela (ou várias) ocorra mais de uma vez durante a execução da aplicação. Aplicações que apresentam regiões paralelas recorrentes são o alvo deste trabalho.

A recorrência das regiões paralelas oferece a oportunidade de avaliar o uso de diferentes configurações e então poder determinar a melhor para atingir um objetivo. Esta

Figura 6.1: Linha do tempo de uma aplicação paralela com regiões paralelas recorrentes

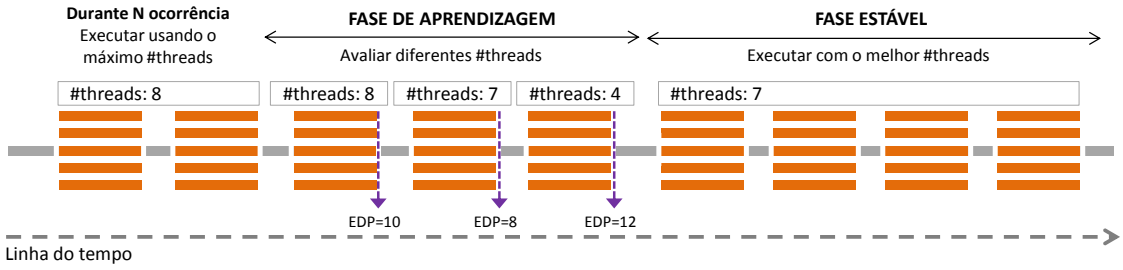


Fonte: O Autor

oportunidade já tem sido explorada em outros trabalhos, como descrito no Capítulo 3. Neste trabalho as regiões paralelas recorrentes são avaliadas com diferentes números de *threads* a fim de encontrar a configuração que resulta o menor EDP. Usando o produto entre consumo de energia e tempo de execução (EDP, do inglês *Energy-Delay Product*) é possível obter um balaço entre estas duas métricas (tempo e energia). Com isto, busca-se evitar uma grande perda de desempenho para economizar energia ou um elevado consumo energético para obter-se o melhor desempenho.

A adaptação do número de *threads* proposta neste trabalho é composta por duas fases: (i) a fase de "aprendizagem", onde são avaliados diferentes números de *threads* a fim de escolher a melhor configuração; e (ii) a fase "estável", onde a região paralela passa a ser executada com o número de *threads* encontrado como ótimo na fase de aprendizagem. A Figura 6.2 ilustra as fases da adaptação.

Figura 6.2: Ilustração das fases da adaptação dinâmica



Fonte: O Autor

É necessário que a região paralela ocorra diversas vezes pra que haja oportunidade de avaliar diferentes configurações. Por isto, uma região paralela só entra em fase de aprendizagem após N ocorrências. Isto ajuda a evitar que regiões paralelas não recorrentes ou com poucas recorrências sejam executadas com um menor número de *threads* que o padrão. Durante este período inicial, a região é executada com o máximo número de *threads* de hardware disponível no sistema. Neste trabalho utilizou-se $N = 2$, sendo que

uma região paralela entrará em fase de aprendizagem na sua terceira ocorrência.

Durante a fase de aprendizagem, a cada nova ocorrência a região paralela é executada com um número de *threads* diferente. O EDP resultante desta execução é calculado e utilizado para decidir qual será o próximo número de *threads* a ser avaliado ou atribuído como melhor número de *threads* (neste último caso, encerrando a fase de aprendizagem). A quantidade de ocorrências em que uma região paralela permanece na fase de aprendizagem varia para cada região. Isso porque a quantidade e a sequência de número de *threads* testados dependem dos resultados de EDP obtidos da região quando executada com um determinado número de *threads*. A fase de aprendizagem é explicada em mais detalhes na Seção 6.1.2. Assim que o melhor número de *threads* é encontrado, a fase de aprendizagem é finalizada. Nas próximas ocorrências a região paralela será executada o com o número de *threads* encontrado como sendo o melhor na fase de aprendizagem.

Vale notar que testar diferentes configurações em tempo de execução pode ter um alto custo em termos de desempenho e consumo energético para a aplicação. O custo refere-se à execução de uma ou mais ocorrências da região paralela com configurações não-ótimas. Dependendo do custo gerado pode levar algum tempo para que as demais execuções da região paralela realizadas com a configuração ótima consiga recuperar o custo e gerar benefícios.

Aplicações podem apresentar variação na carga de trabalho durante o tempo de execução, fazendo com que mude o melhor número de *threads* para executá-la. A inclusão de uma fase de monitoramento do desempenho das regiões paralelas, para identificar a necessidade de uma nova rodada da fase de aprendizagem, será desenvolvida como trabalho futuro (mais detalhes são descritos no Capítulo 7).

6.1.2 Implementação

A adaptação dinâmica do número de *threads* proposta neste trabalho não requer nenhuma modificação em hardware ou compilador, sendo apenas necessário fazer anotações de código na aplicação. Para isto, foi desenvolvida uma biblioteca em linguagem de programação C que encontra e adapta o número de threads com objetivo de otimizar (minimizar) o EDP da aplicação, quando esta é executada em sistemas embarcados com processador ARM. Esta biblioteca destina-se a aplicações paralelizadas com a interface de programação paralela OpenMP. Detalhes sobre a biblioteca implementada são descritos nesta seção.

O Algoritmo 6.1 ilustra o uso da biblioteca. O método $inicializa\_lib()$ (Linha 2), chamado no início do código da aplicação, serve para coletar informações referentes ao processador (número de cores e frequência máxima de operação) e inicializar a biblioteca do PAPI (interface para coletar contadores de hardware). Os contadores de eventos hardware são necessários para estimar a energia consumida em uma região paralela (mais detalhes na Seção 6.1.3). Ao final do programa, o método $finaliza\_lib()$ (Linha 10) é responsável por finalizar a biblioteca do PAPI.

Algoritmo 6.1: Uso da biblioteca em aplicações OpenMP

```
1  início
2      inicializa_lib()
3      iteração em 100
           /* código sequencial                    */
4          inicio_reg_paralela(0)
5          #pragma omp parallel
6          {
           /* código da região paralela            */
7          }
8          fim_reg_paralela(0)
9      fim
10     finaliza_lib()
11 fim
```

A ideia base é aproveitar a recorrência das regiões paralelas para avaliar diferentes números de *threads* a fim de escolher qual o melhor para otimizar EDP. Por isso, o alvo desta biblioteca são as regiões que estão dentro de iterações (Linha 3). Até mesmo as regiões não recorrentes devem ser marcadas. Isso garante que a região seja executada com o máximo número de *threads* (e não o atribuído previamente para a região anterior). Devido ao limite mínimo de ocorrências para iniciar a fase de aprendizagem, estas regiões não serão testadas com outros números de *threads*.

Cada região paralela deve ser envolvida pelos métodos $inicio\_reg\_paralela(id)$ e $fim\_reg\_paralela(id)$ determinando onde inicia e termina uma região paralela (respectivamente nas Linhas 4 e 8). Cada região paralela recebe um identificador (id). A biblioteca foi implementada para realizar o ajuste fino do número de *threads*, ou seja, o número de *threads* é adaptado por região paralela. Entretanto, pode-se usar os métodos para envolver mais de uma região paralela.

Os Algoritmos 6.2 e 6.3 representam respectivamente os métodos $inicio\_reg\_paralela(id)$ e $fim\_reg\_paralela(id)$. O método $inicio\_reg\_paralela(id)$ (Algoritmo 6.2), que é chamado antes de uma região paralela, serve para informar qual é o número de

threads que deve ser utilizado na execução. Para isto, utiliza-se a função $omp\_set\_num\_threads()$ da biblioteca do OpenMP (Linha 2). Cada região paralela tem o seu próprio número de *threads* (armazenado em um array e identificado pelo seu id). Todas as regiões iniciam com o número de *threads* igual ao máximo de *threads* de hardware disponíveis no sistema. O valor de $numThreads$ é ajustado durante a fase de aprendizagem. Quando a região já estiver na fase estável o valor de $numThreads$ corresponderá ao ótimo número de *threads* encontrado na fase de aprendizagem.

Algoritmo 6.2: Início da região paralela

```
   Entrada: id // identificador da região paralela
1  início
2      omp_set_num_threads(numThreads)
3      se (região paralela id não está na fase estável) então
4          Incrementa a quantidade de ocorrências da região paralela id
5          se (ocorrências de id > N) então
               // está na fase de aprendizagem
6              t_ini = omp_get_wtime()
7              Inicia contagem dos eventos hardware
8          fim
9      fim
10 fim
```

Enquanto a região paralela ainda não estiver na fase estável (Linha3), a quantidade de ocorrências da região paralela é contabilizada (Linha 4). Desta forma, identifica-se quando a região paralela está na fase de aprendizagem, ou seja, somente após N ocorrências (Linha 5). Neste trabalho utilizou-se N = 2, ou seja, apenas na terceira ocorrência a região entrará na fase de aprendizagem.

Algoritmo 6.3: Fim da região paralela

```
   Entrada: id // identificador da região paralela
1  início
2      se (região paralela id está na fase de aprendizagem) então
3          tempo = omp_get_wtime() − t_ini
4          Para a contagem dos eventos de hardware
5          Estima o consumo de energia
6          EDP = tempo ∗ energia estimada
7          algoritmo_decisao(id, EDP)
8      fim
9  fim
```

Durante a fase de aprendizagem, para calcular o EDP de uma região paralela quando executada com um determinado número de *threads* é preciso saber o tempo levado

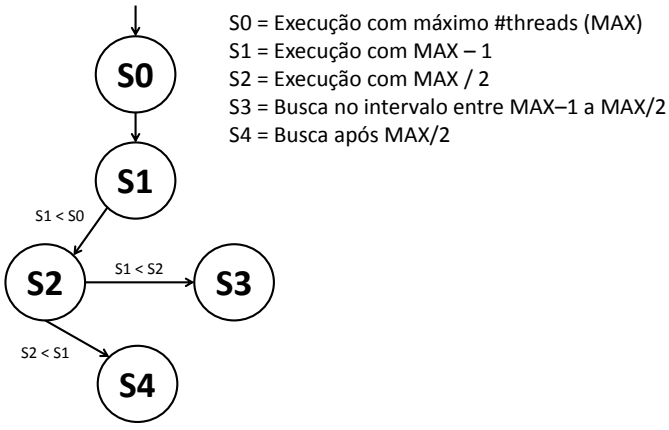
para a sua execução e a energia consumida. O tempo de execução é obtido com o uso da função $omp\_get\_wtime()$ da biblioteca do OpenMP (Linha 6 do Algoritmo 6.2 e Linha 3 do Algoritmo 6.3). Já o consumo de energia precisa ser estimado. Isso porque não há forma direta de saber, durante a execução, qual a quantidade real de energia consumida pela aplicação na maioria dos sistemas embarcados.

Para realizar a estimativa do consumo de energia utilizou-se um modelo baseado em contadores de eventos hardware, que será detalhado na Seção 6.1.3. A contagem dos eventos é iniciada antes da região paralela e finalizada após seu término, respectivamente na Linha 7 do Algoritmo 6.2 e Linha 4 do Algoritmo 6.3.

Observa-se que a coleta dos contadores de hardware só ocorre durante a fase de aprendizagem. Desta forma, mesmo que o custo para coleta seja baixo, a coleta destes dados durante todas as ocorrências da região poderia gerar um custo grande para a aplicação. No Algoritmo 6.3, uma vez tendo a contagem dos eventos de hardware, o consumo de energia é estimado (Linha 5) e o EDP da região paralela resultado da execução com o número de *threads* atual ($numThreads$) é calculado (Linha 6). Após, é decidido qual será o próximo número de *threads* usado para executar a região paralela em sua próxima ocorrência e qual será seu estado, ou seja, se continua na fase de aprendizagem ou se vai para a fase estável (Linha 7).

Para decidir qual o próximo número de *threads* que deve ser avaliado na próxima ocorrência de uma região paralela, e assim poder encontrar o melhor número de *threads* para otimizar EDP, utilizou-se uma heurística baseada nas técnicas de busca da Subida da Encosta (*hill climbing*) e busca em profundidade. A Figura 6.3 ilustra a máquina de estados utilizada pelo algoritmo de decisão para encontrar o melhor número de *threads*.

Figura 6.3: Máquina de estados para encontrar o melhor número de *threads*

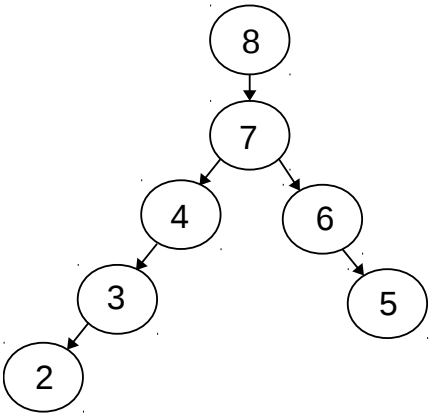


Fonte: O Autor

Quando uma região paralela entra em fase de aprendizagem, esta é executada com o máximo número de *threads* de hardware disponíveis no sistema, ou seja, 8 *threads* (estado S0). Na sua segunda ocorrência, em fase de aprendizagem, a região paralela será executada com 7 *threads*, ou seja, um número a menos que o máximo de *threads* (S1). Caso 8 *threads* tiver apresentado EDP menor que o resultado da execução com 7 *threads*, a região sai da fase de aprendizagem e entra na fase estável, sendo 8 o melhor número de *threads* encontrado para executar esta região paralela. Caso contrário, na próxima ocorrência, esta região paralela será executada com 4 *threads*, ou seja, metade do máximo número de *threads* (S2). Se 4 *threads* apresentar menor EDP que 7 *threads*, será realizado a busca nos próximos números abaixo de 4 *threads*, ou seja, 3 e 2 *threads* nas próximas ocorrências (S4). Se 4 for melhor que 3, 2 *threads* não chega a ser avaliada. Caso 7 *threads* for melhor que 4, a próxima ocorrência será avaliada com 6 *threads* (S3), e 5 *threads* será avaliada somente se 6 for melhor que 7 *threads*.

Conforme descrito, a Figura 6.4 ilustra o grafo de decisão para encontrar o melhor número de *threads*. Pode-se observar que no melhor caso, quando a aplicação apresenta uma boa escalabilidade, a fase de aprendizagem irá durar apenas 2 ocorrências (será avaliada a execução com 8 e com 7 *threads*). No caso médio 4 ocorrências e no pior caso, serão no máximo 5 ocorrências usadas para avaliação (casos em que a aplicação tem seu menor EDP com poucas *threads* – 2 ou 3; 5 ou 6).

Figura 6.4: Grafo de decisão para encontrar o melhor número de *threads*



Fonte: O Autor

6.1.3 Estimativa dinâmica do consumo de energia

Em tempo de execução a aplicação precisa estar ciente do seu consumo de energia para poder tomar decisão em relação à quantidade de *threads* que serão utilizadas para executar uma determinada região paralela. A maioria dos sistemas embarcados atuais não possui sensor de consumo de potência, assim como o sistema utilizado neste trabalho. Por isso, em tempo de execução é necessário realizar uma estimativa da energia consumida. A estimativa é realizada com base em contadores de hardware e segue a metodologia utilizada por Lorenzon, Cera e Beck (2016), assim como descrito nesta seção.

O consumo total de energia estimado (Et) considera a energia dinâmica, consumida pelas instruções executadas (E_{inst}) e pelos acessos aos diferentes níveis da hierarquia de memória (E_{mem}), e energia estática (E_{static}), com dado pela Equação 6.1.

$$Et = E_{inst} + E_{mem} + E_{static} \tag{6.1}$$

A energia consumida pelas instruções executadas (E_{inst}) é calculada pela Equação 6.2. Onde I_{exe} é o número de instruções executadas e E_{per_inst} é a quantidade de energia gasta em média por instrução.

$$E_{inst} = I_{exe} * E_{per\_inst} \tag{6.2}$$

A energia consumida pelo sistema de memória (E_{mem}) é calculada pela Equação 6.3. Onde, $(L1D_{acc} * E_{L1D_{acc}})$ é a energia gasta pelos acessos à memória cache L1 de dados, $(L1I_{acc} * E_{L1I_{acc}})$ é relacionado aos acessos à cache L1 de instruções, $(L2_{acc} * E_{L2_{acc}})$ é referente aos acessos feitos à cache L2 e $(RAM_{acc} * E_{RAM_{acc}})$ é a energia gasta pelos acessos à memória principal.

$$E_{mem} = (L1D_{acc} * E_{L1D_{acc}}) + (L1I_{acc} * E_{L1I_{acc}}) + (L2_{acc} * E_{L2_{acc}}) + (RAM_{acc} * E_{RAM_{acc}}) \tag{6.3}$$

O consumo de energia estática (E_{static}) de todos os componentes é dado pela Equação 6.4. A energia estática é calculada sobre o tempo de execução, o qual é obtido pelos ciclos ($\#Cycles$) dividido pela frequência de operação ($Freq$). S_{CPU}, S_{L1I}, S_{L1D}, S_{L2}, e S_{RAM} corresponde, respectivamente, a potência estática consumida pelo processador, cache L1 de instrução, cache L1 de dados, cache L2 e memória principal.

$$E_{static} = \left(\frac{\#Cycles}{Freq} \right) * (S_{CPU} + S_{L1I} + S_{L1D} + S_{L2} + S_{RAM}) \quad (6.4)$$

O número de instruções executadas, a quantidade de acessos às caches e memória principal, e ciclos são obtidos por contadores de hardware com o uso da biblioteca PAPI (MUCCI et al., 1999) (versão 5.5.1). Foram utilizados os mesmo contadores de hardware descritos na Tabela 4.3 na Seção 4.2.

A Tabela 6.1 apresenta o valor assumido para o consumo de potência estática de cada componente, o consumo de energia (em nano Joules) gasto em média por instruções executada, acesso aos diferentes níveis de cache e a memória principal. O consumo de potência estática foi duplicado do valor previamente utilizado por Lorenzon, Cera e Beck (2016). Lorenzon, Cera e Beck (2016) utilizavam processador com 4 núcleos, enquanto neste trabalho utiliza-se processador com 8 núcleos (como descrito na Seção 4.2).

Tabela 6.1: Consumo de energia assumido para cada evento e componente do sistema

Componente	**Potência estática**	**Energia por evento**
Processador	0,5 W	0,237 nJ (por instrução)
L1-D	0,0001 W	0,017 nJ (por acesso)
L1-I	0,0001 W	0,017 nJ (por acesso)
L2	0,0516 W	0,296 nJ (por acesso)
RAM	0,24 W	2,770 nJ (por acesso)

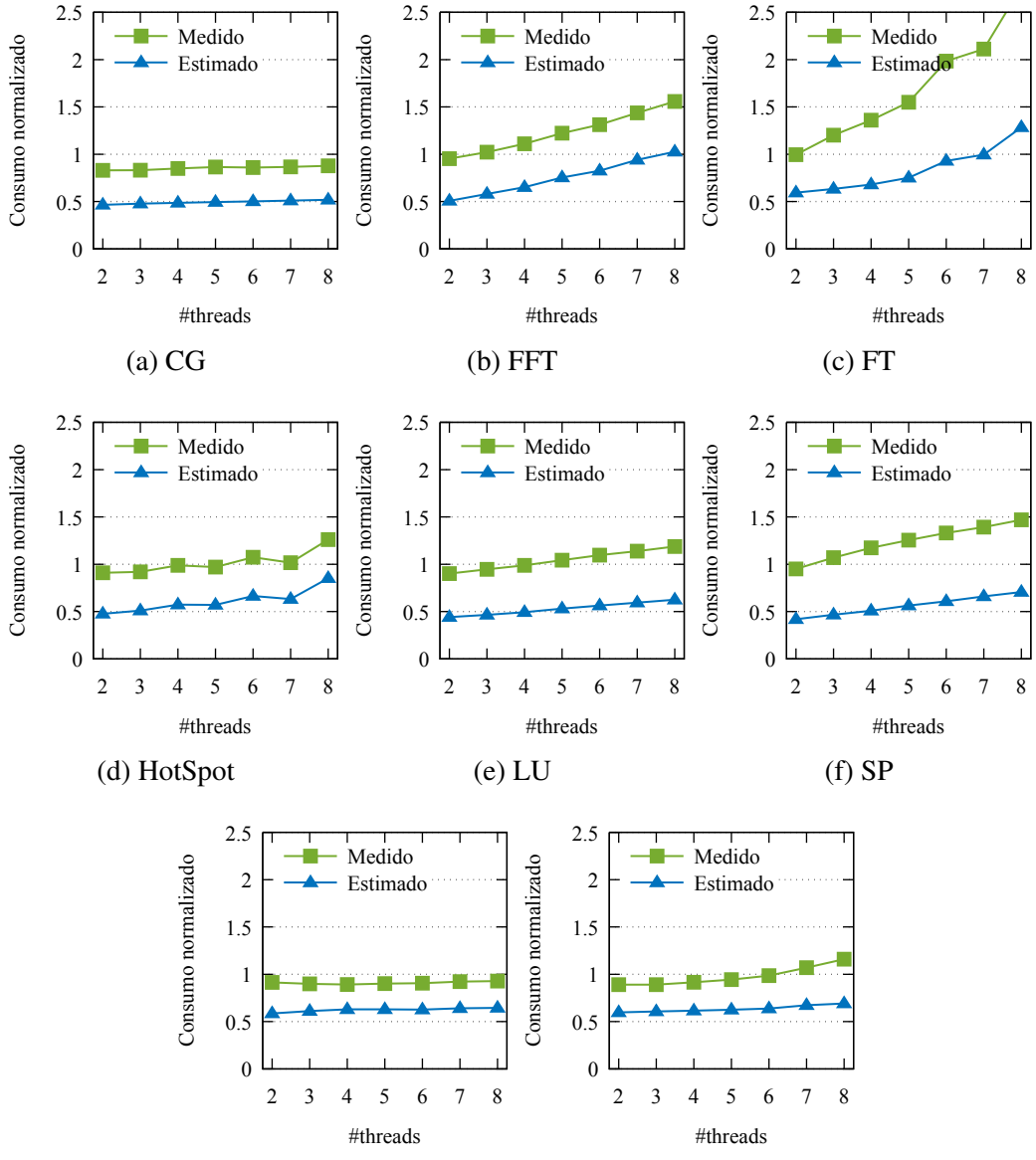
Fonte: O Autor

6.2 Avaliação da estimativa dinâmica do consumo de energia

Este trabalho objetiva minimizar o EDP das aplicações paralelas por meio da adaptação no número de *threads* utilizadas na execução da aplicação. Para realizar este ajuste é preciso saber, em tempo de execução, qual o tempo e o consumo de energia de uma determinada região paralela da aplicação, para então poder calcular o EDP e decidir qual o melhor número de *threads*. Como não há forma direta de saber a energia consumida, esta precisa ser estimada. Nesta seção é apresentada uma avaliação da estimativa de consumo de energia, descrita na Seção 6.1.3, utilizada na biblioteca proposta neste trabalho. Embora a estimativa seja utilizada em tempo de execução para cada região paralela separadamente, a avaliação feita nesta seção considera o consumo de toda a aplicação.

Para realizar a avaliação da estimativa do consumo de energia foram coletados contadores de hardware, listados na tabela 4.3, de cada aplicação quando executadas com diferentes números de *threads*, e aplicados na fórmula da estimativa do consumo de energia (descrita na seção anterior). Na Figura 6.5 são apresentados os gráficos de cada aplicação comparando o consumo medido com o resultado da estimativa. Os valores foram normalizados pelo consumo de energia medido da versão sequencial.

Figura 6.5: Comparação entre consumo de energia medido e estimado



Fonte: O Autor

A estimativa apresentou em média 42,75% de erro quando comparado com o consumo medido (menor erro de 29,38% e maior erro de 56,66%). A estimativa sempre apresentou-se menor que o consumo real. Embora haja uma grande diferença, é possí-

vel notar nos gráficos da Figura 6.5 que as proporções são parecidas, ou seja, a linha do consumo estimado segue o padrão da linha do consumo de energia medido. Este é o ponto importante para o objetivo deste trabalho. Isso porque esta estimativa é utilizada para decidir, durante a execução, qual o número de *threads* que resulta em menor EDP. Então, mesmo que haja uma grande diferença, o erro da estimativa será proporcional. Utilizar uma metodologia para obter estimativa mais acurada é um trabalho que poderá ser desenvolvido futuramente.

6.3 Resultados da adaptação dinâmica

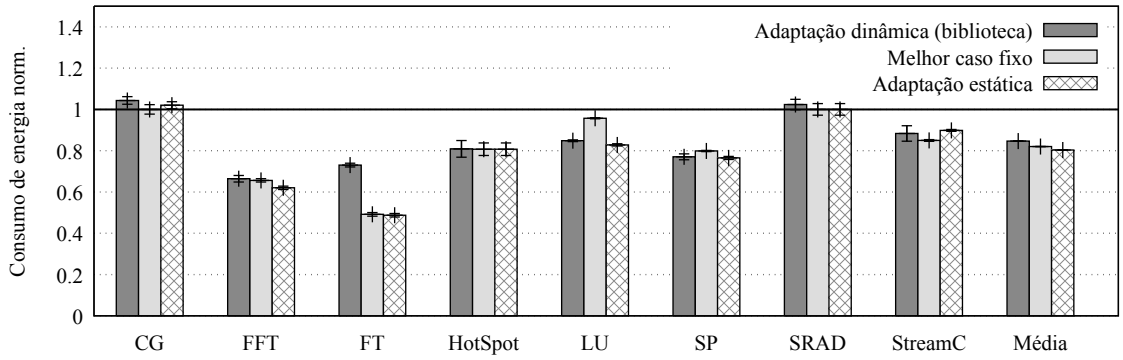
Para avaliar a biblioteca de adaptação dinâmica do número de *threads* utilizou-se a mesma metodologia descrita no Capítulo 4. Incluiu-se chamadas às funções da biblioteca em cada região paralela dos *benchmarks*, de forma a realizar o ajuste fino no número de *threads*. Os resultados de consumo de energia, tempo de execução e EDP são apresentados nas Figuras 6.6, 6.7 e 6.8, respectivamente.

Em geral, as aplicações paralelas são executadas com o número máximo de *threads* de hardware disponíveis no sistema. Diante disto, a linha de base, representada pela linha preta nos gráficos, refere-se aos resultados obtidos das execuções com 8 *threads* (máximo de *threads* disponível no sistema utilizado nos experimentos). Todos os resultados foram normalizados pela linha de base. Vale notar que, como os resultados referem-se ao tempo de execução, ao consumo de energia e EDP (produto do consumo de energia e tempo de execução), quanto menor for seu valor, melhor é o resultado.

Os resultados de "Adaptação dinâmica (biblioteca)" referem-se aos resultados obtidos com o uso da biblioteca de adaptação dinâmica. Os resultados de "melhor caso fixo" referem-se ao uso do número de *threads* encontrado como melhor para otimizar EDP durante a exploração do espaço de projeto (descrito no Capítulo 5), ou seja, quando usado um único número de *threads* para todas as regiões paralelas da aplicação. Já os resultados apresentados como "adaptação estática" referem-se à execução da aplicação com a combinação de número de *threads* encontrados pela biblioteca de adaptação dinâmica, mas atribuídos de forma fixa em cada região paralela. Com isso, pode-se avaliar o custo de aprendizagem em tempo de execução (testar diferentes configurações) e a acurácia na escolha da combinação de número de *threads*. Não foi realizada uma busca exaustiva para encontrar o melhor número de *threads* de cada região paralela de cada *benchmark*.

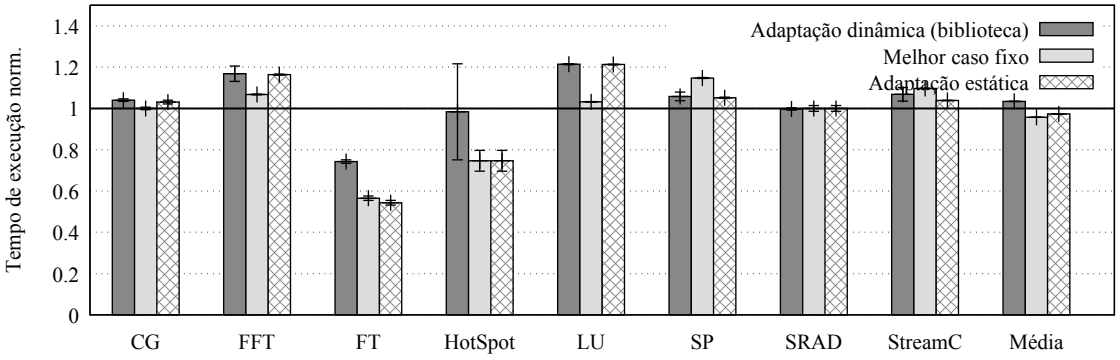
Na Tabela 6.2 é apresentado o número de *threads* encontrado pela adaptação di-

Figura 6.6: Resultados de consumo de energia normalizados pela configuração padrão



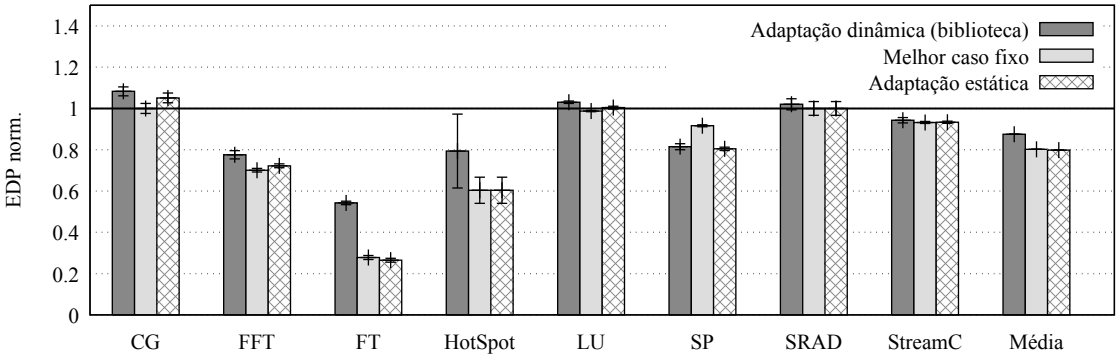
Fonte: O Autor

Figura 6.7: Resultados de tempo de execução normalizados pela configuração padrão



Fonte: O Autor

Figura 6.8: Resultados de EDP normalizados pela configuração padrão



Fonte: O Autor

nâmica para cada região paralela de cada *benchmark*. Cada aplicação foi executada 30 vezes com a adaptação dinâmica, e houveram variações na escolha do melhor número de *threads* para cada região. Nesta tabela são apresentados os número que foram escolhidos na maioria das vezes pela adaptação. HotSpot, por exemplo, é uma aplicação que apresentou grande desvio padrão nos resultados de tempo e EDP (Figuras 6.7 e 6.8), resultado pelas diferentes escolhas de melhor número de *threads*. Em doze das trinta execuções

a adaptação dinâmica encontrou 7 *threads* como sendo o melhor número para HotSpot. Contudo, nas outras execuções a adaptação dinâmica também encontrou 6, 4 ou 3 *threads* como melhor número. O *benchmark* HotSpot apresenta bastante variação no EDP, mesmo quando executado com o mesmo número de *threads*, fazendo com que a escolha feita pela adaptação não seja sempre a mesma. Pode-se observar na Tabela B.4 do Apêndice B, que HotSpot apresentou desvio padrão de até 11,98% em EDP nos experimentos da exploração de espaço de projeto.

A variação pode ter sido originada pela forma como as *threads* foram escalonadas. Sendo que não foi atribuído afinidade de *threads* aos núcleos de processamento, a cada execução o escalonador pode ter alocado as *threads* em locais diferentes e assim originado a variação. Vale notar que o processador utilizado nos experimentos é composto por 8 núcleos que, embora estejam no mesmo chip, estão organizados em 2 *sockets*. Assim, o local onde as *threads* são escalonadas irá influenciar no desempenho e consumo de energia da aplicação. Outro ponto que pode ter influenciado nesta variação é o DVFS em modo *ondemand*.

Tabela 6.2: Configuração do número de *threads* encontrado pela adaptação dinâmica

Benchmark	Número de *threads* encontrado para cada região paralela
CG	[8, 2, 2]
FFT	[2, 8, 2]
FT	[4, 4, 3, <=3, <=3]
HotSpot	7
LU	[8, 4]
SP	[7, 4, 4, 4, 4, 2, 2, 2, 2]
SRAD	[8, 8]
StreamCluster	[7, 6]

Fonte: O Autor

FT tem 5 regiões paralelas recorrentes mas, devido às poucas iterações geradas pelo conjunto de dados utilizado nos experimentos (apenas 6 iterações), somente para três regiões paralelas a adaptação dinâmica conseguiu encontrar o melhor número de *threads*. Isto porque estas três regiões são também chamadas fora da iteração, sendo repetidas mais que 6 vezes durante a execução da aplicação. Para as outras duas regiões paralelas a adaptação dinâmica apenas encontrou que 3 ou 2 *threads* poderiam ser o melhor número. Isso porque nas duas primeiras ocorrências as regiões paralelas não estão em fase de aprendizagem e após isso, nas demais ocorrências, foram executadas com 8, 7, 4 e 3 *threads*. 3 *threads* apresentou melhor resultado que 4 *threads*, e o próximo número a ser

avaliado seria 2 *threads*.

Nas seções a seguir, os resultados de consumo de energia, tempo de execução e EDP resultado da adaptação dinâmica serão discutidos. Inicialmente os resultados são comparados com os obtidos pela configuração padrão, do uso de 8 *threads*, na Seção 6.3.1. Em sequencia, na Seção 6.3.2, os resultados são discutidos em comparação com o "melhor caso fixo" e com a "adaptação estática" (uso da combinação de número de *threads* encontrada pela biblioteca, mas sem o custo da aprendizagem em tempo de execução).

6.3.1 Adaptação dinâmica vs. configuração padrão

Analisando os resultados da adaptação dinâmica em relação à configuração padrão, na qual a aplicação é executada com o máximo de *threads* de hardware disponíveis no sistema (representada pela linha de base), pode-se observar na Figura 6.6 que a adaptação resultou na economia de energia para grande maioria das aplicações. O resultado mais significativo foi apresentado no *benchmark* FFT, onde a adaptação dinâmica ofereceu 33,59% de economia de energia em relação à execução com o máximo número de *threads*. Apenas dois *benchmarks*, CG e SRAD, apresentaram pequeno aumento no consumo de energia. Em média, obteve-se com adaptação dinâmica, 15,35% de economia de energia.

Quando busca-se otimizar EDP, tem-se um balanço entre os objetivos de economia de energia e desempenho. Para a maioria da aplicações, é fato que a economia de energia gerada pela otimização desta métrica irá causar pequena perda de desempenho. Esta situação pode ser observada nos resultados de tempo de execução apresentados na Figura 6.7. Cinco aplicações apresentaram aumento no tempo de execução. A maioria destas aplicações apresentam seu melhor desempenho com 8 *threads* (Tabela 5.1). Desta forma, ao serem executadas com menor número de *threads*, o tempo de execução aumenta. Pode-se notar que, embora FFT tenha apresentado seu melhor desempenho com 6 *threads*, este foi apenas 1,41% melhor que o apresentado por 8 *threads* (Tabela 5.1). As duas aplicações que têm melhor desempenho quando executadas com menos *threads* que o padrão (FT e HotSpot), apresentaram redução no tempo de execução quando aplicada a adaptação dinâmica. O *benchmark* FT, por exemplo, teve 25,74% de melhora do desempenho. Em média, a adaptação dinâmica apresentou apenas 3,41% de perda de desempenho.

Com relação aos resultados de EDP (*Energy-Delay Product*) apresentados na Figura 6.8, a adaptação dinâmica conseguiu sua otimização em cinco das oito aplicações

avaliadas. O resultado mais significativo foi a redução de 45,77% no EDP do *benchmark* FT. Este resultado deve-se ao fato deste *benchmark* ter apresentado tanto economia de energia quanto ganho em desempenho com a adaptação dinâmica. Para os *benchmarks* FFT, HotSpot, SP e StreamCluster a adaptação dinâmica resultou na redução do EDP em 22,45%, 20,65%, 18,53% e 5,71%, respectivamente. Os *benchmarks* CG e SRAD, como visto na Seção 5, apresentam o melhor EDP com 8 *threads*. O *benchmark* LU, embora tem seu melhor EDP com 7 *threads*, este resultado é apenas 1,23% melhor que o obtido com 8 *threads*. Avaliar estas aplicações com outras configurações durante a execução tem custo, fazendo com que o EDP resultado da adaptação dinâmica seja maior que a execução com a configuração padrão. Em média, a adaptação dinâmica apresentou redução de 12,47% no EDP das aplicações.

Avaliar diferentes números de *threads* em tempo de execução pode gerar alto custo em termos de tempo e consumo de energia. Este custo refere-se à execução das regiões paralelas com número de *threads* não-ótimos. Pode levar um tempo para que este custo gerado pela aprendizagem seja amenizado e recuperado, fazendo com que a técnica da adaptação dinâmica traga benefícios para a aplicação. Sendo que não são avaliados todos os números de *threads* possíveis, a escolha do melhor número de *threads* pode cair em um ótimo local e, assim, escolher um solução quase-ótima. Somado à isto, há a questão da incerteza em relação ao consumo real de energia. Isso porque esta informação precisa ser estimada em tempo de execução. Desta forma, o erro nesta estimativa pode causar a escolha de uma configuração não-ótima, prejudicando os resultados obtidos pela adaptação do número de *threads*. Estas questões serão discutidas na seção a seguir.

6.3.2 Avaliação da combinação de número de threads encontrada pela biblioteca e avaliação do custo de aprendizagem

Nesta seção serão comparados os resultados de EDP (Figura 6.8) da adaptação dinâmica (uso da biblioteca), “melhor caso fixo” e “adaptação estática”. O “melhor caso fixo” refere-se à execução com o número de *threads* que resulta no menor EDP quando utilizado um único número de *threads* para toda a aplicação (Tabela 5.1), ou seja, todas as regiões paralelas são executadas com o mesmo número de *threads*. A “adaptação estática” refere-se ao uso da combinação de número de *threads* encontrada pela biblioteca (Tabela 6.2), mas atribuída no código de maneira estática, ou seja, sem o custo da aprendizagem em tempo de execução.

Comparando os resultados do "melhor caso fixo" com a "adaptação estática" busca-se discutir a escolha da combinação de número de *threads* feita pela biblioteca. Ao comparar os resultados da "adaptação dinâmica" (uso da biblioteca) com os resultados da "adaptação estática" será discutido o custo da aprendizagem em tempo de execução. A discussão apresentada nesta seção ajuda a entender o porquê, mesmo sendo o objetivo da biblioteca otimizar EDP, apenas 5 aplicações tiveram benefícios nesta métrica com o uso da adaptação dinâmica. Esta discussão pode ajudar trabalhos futuros a aprimorar a técnica. Os resultados de cada *benchmark* serão discutidos individualmente.

Para o *benchmark* CG, devido à sua ótima escalabilidade apresentada até 8 *threads*, este número é considerado o seu melhor caso fixo e corresponde ao mesmo valor da linha de base (máximo número de *threads*). CG possui 3 regiões paralelas recorrentes, e a biblioteca de adaptação dinâmica entrou 8 *threads* como sendo o melhor número para otimizar EDP em somente uma destas regiões paralelas. Para as outras duas regiões paralelas, 2 *threads* foi considerado o número ideal (Tabela 6.2). Contudo, quando CG é executado com estes números encontrados pela adaptação dinâmica atribuídos de forma estática em cada região, o resultado de EDP obtido não é menor que o melhor caso fixo (Figura 6.8).

Pode-se notar então que, para o *benchmark* CG, a biblioteca não foi capaz de encontrar o melhor número de *threads* para todas as regiões paralelas. Este erro na escolha da combinação de número de *threads* pode ter sido causado pela estimativa no consumo de energia. Quando CG é executado com a combinação de número de *threads* encontrada pela biblioteca, o EDP resultado é 5,11% maior que o melhor caso fixo (8 *threads* para todas as regiões). A adaptação dinâmica apresentou EDP 3,03% maior que a "adaptação estática", sendo este o custo de testar outras configurações em tempo de execução.

O *benchmark* FFT, como visto na Seção 5, apresentou menor EDP com 3 *threads* (Tabela 5.1), sendo este então considerado o seu "melhor caso fixo". FFT possui três regiões paralelas, e a biblioteca de adaptação dinâmica encontrou como melhor número 2 *threads* para duas das regiões paralelas e 8 *threads* para a outra (Tabela 6.2), sendo esta a configuração de número de *threads* utilizadas na execução da "adaptação estática". FFT com "adaptação estática" apresentou um resultado de EDP muito próximo do melhor caso fixo (Figura 6.8), mostrando que para este *benchmark* a adaptação dinâmica conseguiu encontrar uma configuração muito próxima da melhor. O "melhor caso fixo" apresentou EDP apenas 2,95% menor que a "adaptação estática". Para o *benchmark* FFT, a configuração de número de *threads* encontrada pela biblioteca atribuída de maneira está-

tica (adaptação estática) provê maior economia de energia que o melhor caso fixo (Figura 6.6), ao custo de maior perda de desempenho (Figura 6.7).

Ainda analisando os resultados de EDP do *benchmark* FFT (Figura 6.8), pode-se notar, ao comparar os resultados de EDP da "adaptação estática" com a adaptação dinâmica, que o custo da adaptação dinâmica para este *benchmark* é de apenas 7,38%. Este custo deve-se ao período inicial e a fase de aprendizagem, ou seja, momentos em que as regiões paralelas são executadas com número de *threads* não-ótimos.

O *benchmark* FT possui cinco regiões paralelas recorrentes. Para as duas primeiras regiões paralelas a biblioteca de adaptação dinâmica encontrou 4 *threads* como sendo o melhor número e para a terceira 3 *threads*. As duas últimas regiões paralelas não chegaram a concluir a fase de aprendizagem, devido às poucas iterações do conjunto de dados de entrada utilizado nos experimentos. Contudo, a adaptação dinâmica encontrou 3 *threads* como melhor número até então. Esta combinação de número de *threads* (apresentada na Tabela 6.2) foi utilizada na "adaptação estática". Já o "melhor caso fixo" de FT corresponde a execução com 4 *threads* (número de *threads* que apresentou menor EDP durante os experimentos realizado na exploração de espaço de projeto).

Para o *benchmark* FT, a configuração de número de *threads* encontrada pela biblioteca atribuída de maneira estática (adaptação estática) mostrou-se levemente mais eficiente na otimização de EDP que o melhor caso fixo (Figura 6.8). Quando FT é executada com a combinação encontrada pela biblioteca atribuída de forma fixa, tem-se uma otimização no EDP de 4,75% em comparação com o melhor caso fixo. Contudo, devido ao conjunto de entrada utilizado neste *benchmark* ter apenas 6 iterações, o custo de aprendizagem da adaptação dinâmica torna-se muito grande. Não há tempo do *benchmark* aproveitar os benefícios gerados pela seleção da melhor configuração. Desta forma, comparando o resultado da adaptação dinâmica com a "adaptação estática", pode-se notar que o custo de aprendizagem para este *benchmark* foi de 104,87% em EDP. Entretanto, que a tendência é que este custo de aprendizagem seja minimizado à medida que aumenta a quantidade de iterações das regiões paralelas. Mesmo apresentando este alto custo de aprendizagem, vale notar que a adaptação dinâmica gerou ganhos em EDP em relação à configuração padrão.

O *benchmark* HotSpot tem apenas uma região paralela, a qual apresenta o melhor EDP com 7 *threads*. A adaptação dinâmica foi capaz de encontrar este número na maioria das vezes. Desta forma, o "melhor caso fixo" e a "adaptação estática" correspondem aos resultados da execução deste *benchmark* com 7 *threads*. Embora nem sempre a melhor

configuração seja encontrada, pode-se observar pelo desvio padrão, que mesmo quando não encontrado o melhor número de *threads*, a adaptação dinâmica apresenta redução do EDP em relação a configuração padrão (Figura 6.8). Comparando o resultado da adaptação dinâmica com a "adaptação estática", tem-se que o custo médio da aprendizagem de 31,4%. Contudo, se considerado apenas as execuções nas quais a adaptação encontrou 7 como melhor número de *threads*, tem-se o custo de aprendizagem de apenas 2% em EDP.

O *benchmark* LU, como visto durante os experimentos da exploração de espaço de projeto, apresentou o melhor EDP com 7 *threads*. Sendo assim, este corresponde ao resultado de "melhor caso fixo". LU possui duas regiões paralelas recorrentes. A biblioteca de adaptação dinâmica encontrou um número de *threads* ideal para cada uma, conforme apresentado na Tabela 6.2. Quando este *benchmark* é executado com a configuração de número de *threads* encontrada pela adaptação dinâmica ("adaptação estática"), tem-se como resultado um EDP 1,68% maior que o melhor caso fixo e apenas 0,42% maior que a configuração padrão. Isto significa que a adaptação dinâmica conseguiu encontrar um de número de *threads* muito próximo do melhor para todas as regiões paralelas de LU. Vale notar também que a combinação encontrada pela adaptação dinâmica, gera uma economia de energia de 13,5% em comparação com o melhor caso fixo (Figura 6.6). Comparando o EDP da adaptação dinâmica com a "adaptação estática", observa-se que o custo de aprendizagem para este *benchmark* é de apenas 2,58% (Figura 6.8).

O *benchmark* SP, apresenta o menor EDP com 4 *threads*, quando considerado um único número de *threads* para todas as regiões paralelas, sendo este número o seu "melhor caso fixo". SP possui nove regiões paralelas e a biblioteca encontrou um número de *threads* para cada uma delas. Esta combinação, apresentada na Tabela 6.2, corresponde a "adaptação estática". Comparando os resultados de EDP (Figura 6.8) do "melhor caso fixo" com a "adaptação estática", pode-se observar que, por determinar um número de *threads* específico para cada região, a combinação encontrada pela biblioteca apresenta menor EDP que o melhor caso fixo, onde todas as regiões paralelas são executadas com 4 *threads*. Enquanto o melhor caso fixo reduz o EDP de SP em 8,39%, a "adaptação estática" reduz em 19,57%, quando comparados com a configuração padrão de 8 *threads*. Desta forma, a "adaptação estática" é 12,2% melhor que o "melhor caso fixo". Comparando os resultados da adaptação dinâmica com a "adaptação estática", pode-se notar que o custo de aprendizagem também é baixo para este *benchmark* (1,29% em EDP).

Para o *benchmark* SRAD, o "melhor caso fixo" e a "adaptação estática" correspondem a execução com 8 *threads*. Isso porque SRAD apresenta melhor EDP com 8

threads quando executado com um único número de *threads* (Tabela 5.1) e, apesar de SRAD ter duas regiões paralelas, a biblioteca de adaptação dinâmica encontrou 8 *threads* como sendo o melhor número para ambas (Tabela 6.2). Observa-se que a adaptação dinâmica apresentou EDP apenas 2,03% maior que a "adaptação estática" (Figura 6.8). Este baixo custo de aprendizagem deve-se ao fato da biblioteca avaliar apenas o uso de 8 e 7 *threads*, para então escolher 8 como melhor número. Quando o algoritmo de decisão verifica que 7 *threads* é pior para EDP que 8 *threads*, 8 *threads* é escolhido como melhor número e a fase de aprendizagem termina.

Para o *benchmark* StreamCluster, o qual tem duas regiões paralelas, a biblioteca também encontrou um número de *threads* ideal para cada uma. Os resultados da "adaptação estática" correspondem a execução deste *benchmark* com a configuração encontrada pela biblioteca, apresentada na Tabela 6.2 e atribuída de forma fixa em cada região paralela. Pode-se observar que a combinação de número de *threads* encontrada é bem próxima do "melhor caso fixo" (6 *threads*) encontrado durante a exploração de espaço de projeto. A "adaptação estática" apresenta diferença de apenas 0,12% em EDP comparado com o "melhor caso fixo", mostrando que para este *benchmark* a adaptação dinâmica também conseguiu encontrar uma configuração boa. O custo de aprendizagem do StreamCluster é de 1,04% em EDP (comparando os resultados da adaptação dinâmica com a "adaptação estática").

6.3.3 Discussão

A biblioteca de adaptação dinâmica do número de *threads* proposta neste trabalho aproveita a recorrência das regiões paralelas das aplicações para em tempo de execução avaliar diferentes números de *threads*, a fim de encontrar o número ideal para otimizar EDP. Executar as regiões paralelas com número de *threads* não-ótimos tem custo tanto em termos de desempenho quanto em consumo energético. Este custo depende do tamanho da região paralela, da quantidade de vezes que esta é repetida durante a execução da aplicação e da sua escalabilidade, ou seja, o prejuízo gerado por avaliar uma região paralela com número de *threads* menor que o ideal para ela. Embora exista o custo de aprendizagem, pode-se notar pelos resultados que, em geral, este apresentou-se baixo. Sendo alto apenas para o *benchmark* FT, o qual tem poucas iterações no conjunto de dados utilizados nos experimentos.

Mesmo com o custo de aprendizagem, o caso mais significativo de ganho com a

biblioteca foi na aplicação FT. Isso porque esta aplicação além obter ganhos em economia de energia, também obteve ganhos em desempenho. FT requer muita comunicação entre as *threads*, fazendo com que a partir de determinado número de *threads*, o tempo que as *threads* passam esperando para acessar a memória compartilhada ultrapasse os ganhos obtidos com a paralelização. Com isso, pode-se notar que aplicações com muita sincronização de dados são as que mais se beneficiam do uso da biblioteca, pois esta encontra um número de *threads* que provê também um melhor desempenho.

Um ponto importante a ser comentado é que a biblioteca nem sempre conseguiu encontrar a melhor combinação de número de *threads*. Para o *benchmark* CG, por exemplo, o aumento no EDP gerado em relação à configuração padrão foi gerado não somente pelo custo de aprendizagem, mas também pela seleção de um número de *threads* não-ótimo para algumas regiões paralelas. Contudo, sendo que o melhor número de *threads* para este *benchmark* seria 8 e o aumento no EDP foi baixo, pode-se dizer que a adaptação dinâmica foi capaz de encontrar um número de *threads* quase-ótimo. A escolha de um número de *threads* não-ótimo pode ter sido causada pelo algoritmo de busca utilizado, que devido a sua heurística baseada em *hill-climbing*, que pode cair em ótimos locais. Outro ponto importante é a estimativa no consumo de energia, que pode ter sido influenciada pela variação da frequência de processamento devido ao DVFS em modo *ondemand*.

Vale notar também que, embora alguns *benchmarks* tenham apresentado aumento no EDP, em média a adaptação dinâmica reduziu o EDP em 12,47%, quando comparado com a configuração padrão (Figura 6.8). A adaptação dinâmica também gerou, em média, 15,35% de economia em energia (Figura 6.6) com apenas 3,41% de perda de desempenho em relação a linha de base (Figura 6.7). Comparando a média de ganho em EDP resultado pelo "melhor caso fixo" com a "adaptação estática", onde tira-se o custo de aprendizagem da adaptação dinâmica, pode-se notar que estes são muito próximos. Enquanto o melhor caso fixo provê a redução média de 19,77% do EDP, a "adaptação estática" teve redução de 20,21%, comprovando que a adaptação dinâmica foi capaz de encontrar um número de *threads* ótimo ou quase-ótimo para cada região paralela.

7 CONSIDERAÇÕES FINAIS

Objetivando otimizar EDP (*Energy-Delay Product*) de aplicações paralelas OpenMP executadas em sistemas embarcados, este trabalho apresentou uma proposta de adaptação dinâmica do número de *threads*. Ao realizar uma revisão na literatura atual (Capítulo 3), pode-se constatar que, embora a técnica da adaptação dinâmica do número de *threads* já tenha sido bastante explorada por trabalhos anteriores para otimizar aplicações paralelas, pouco se fez em relação a sua aplicação em sistemas embarcados. O foco principal do estado da arte neste tema são os processadores de propósito geral ou servidores, os quais geralmente possuem maior quantidade de núcleos de processamento que sistemas embarcados, além serem diferentes em termos de micro-arquitetura, hierarquia e tamanho de memória e frequência de processamento.

Os sistemas embarcados atuais são compostos por múltiplos núcleos de processamento, permitindo que as aplicações possam fazer uso destes recursos para acelerar sua execução. Quando uma aplicação é paralelizada, seu consumo energético tende a aumentar, devido à maior potência dissipada pelo uso de mais núcleos de processamento e maior quantidade de acessos às memórias compartilhadas. Sendo que sistemas embarcados geralmente são dependentes de bateria, o consumo energético gerado pela paralelização das aplicações não pode ser negligenciado. Desta forma, otimizar a métrica de EDP pode ser uma solução útil, pois tem-se um balanço entre eficiência energética e desempenho.

A prática mais comum para executar uma aplicação paralela é utilizar o máximo número de *threads* disponíveis no sistema, ou seja, todos os núcleos do processador. Contudo, este trabalho mostrou através da exploração de espaço de projeto (Capítulo 5) que esta estratégia pode não ser a melhor para EDP. Isso porque, para a grande maioria das aplicações, usar o máximo número de *threads* pode causar prejuízos em termos energéticos. Além disso, existem também aplicações com escalabilidade limitada. Para estas aplicações, a limitação do número de *threads* pode trazer, não somente economia de energia como também, maior desempenho. Por meio destes resultados, pode-se verificar que aplicações paralelas quando executadas em sistemas embarcados, também podem se beneficiar da adaptação do número de *threads*.

Diante destas constatações, este trabalho propôs e implementou uma biblioteca para realizar a adaptação do número de *threads* de aplicações paralelas em sistemas embarcados com processador ARM, a fim de otimizar EDP (Capítulo 6). A biblioteca foi desenvolvida em linguagem de programação C, podendo ser utilizada para otimizar aplica-

ções C/C++ paralelizadas com a interface de programação paralela OpenMP. A biblioteca busca encontrar em tempo de execução, sem qualquer informação prévia da aplicação, o melhor número de *threads* para executar cada região paralela da aplicação. Uma aplicação paralela pode conter mais de um trecho de código paralelo, por isso, a biblioteca realiza o ajuste fino, ou seja, a adaptação do número de *threads* é realizada individualmente para cada região paralela. A adaptação automática e dinâmica permite considerar informações que só são acessíveis em tempo de execução como, por exemplo, o tamanho da entrada. Vale notar também que a biblioteca não requer modificação em hardware ou compilador específico.

7.1 Conclusões

A biblioteca proposta e desenvolvida neste trabalho foi avaliada em oito aplicações com diferentes características em um sistema embarcado com processador octa-core. Como resultado, a adaptação dinâmica do número de *threads* permitiu, em média, economizar 15,35% no consumo de energia com apenas 3,41% de perda de desempenho, gerando assim 12,47% de otimização de EDP em relação a configuração padrão (uso do máximo número de *threads* disponíveis no sistema). No melhor caso, a adaptação dinâmica foi capaz de economizar 26,97% em energia enquanto promoveu 25,74% de aumento no desempenho, resultando em 45,77% de melhora no EDP.

Na busca pela melhor configuração, a biblioteca aproveita a recorrência das regiões paralelas para testar em tempo de execução diferentes número de *threads*, até encontrar o número que ofereça o menor EDP. Contudo, quando uma região paralela é executada com número de *threads* não-ótimo, gera-se um custo tanto em tempo de execução quanto em energia. Este custo de aprendizagem, dependendo da aplicação pode ser alto, e precisa ser superado pelos ganhos gerados com a escolha da melhor configuração. Para isso, a região paralela precisa ocorrer diversas vezes, para que a adaptação gere efeitos positivos. Embora a adaptação dinâmica introduza o custo da aprendizagem em tempo de execução, vale notar que ela é mais vantajosa, pois permite que a escolha do número de *threads* seja adequada ao tamanho da entrada e às características do sistema (permite portabilidade).

Através da execução das aplicações com a configuração encontrada pela adaptação dinâmica atribuída de forma fixa no código de aplicação, avaliou-se o custo de testar diferentes configurações em tempo de execução, bem como a escolha do melhor número

de *threads*. Pode-se notar que a adaptação dinâmica foi capaz de encontrar, na maioria das vezes, uma solução ótima ou quase-ótima. Em conjunto a isto, o custo da aprendizagem também mostrou-se pequeno para a maioria das aplicações. Além disso, vale notar que a tendência é que quanto maior o programa, menor será o custo da aprendizagem, e com isto, maior serão os ganhos.

7.2 Trabalhos futuros

Para finalizar, nesta seção são discutidas questões que ficaram em aberto e pontos que precisam ser melhorados no trabalho proposto. Assim, busca-se apresentar direções para trabalhos que possam ser desenvolvidos futuramente.

O principal desafio encontrado, que ainda precisa ser superado, é a falta da informação sobre consumo real de energia durante a execução da aplicação. Como não há uma forma de ter este conhecimento em tempo de execução na grande maioria dos sistemas embarcados atuais, o consumo de energia precisa ser estimado. Devido à imprecisão na estimativa utilizada neste trabalho, a adaptação dinâmica nem sempre encontrou o melhor número de *threads* para todas as regiões paralelas das aplicações. Uma alternativa seria refinar os coeficientes da estimativa de consumo de energia já utilizada e fixar o DVFS. Isso porque a variação da frequência, devido ao modo *ondemand* utilizado nos experimentos, pode ter influenciado na estimativa. Outra possibilidade seria utilizar uma estimativa mais acurada a fim de garantir a qualidade na escolha do número de *threads* durante a adaptação dinâmica.

Entretanto, vale notar que a estimativa do consumo de energia precisa ser leve ao ponto de não incluir custos adicionais no tempo de execução da aplicação. Dependendo do processador utilizado, há também a limitação nos contadores de hardware que podem ser utilizados. Por exemplo, o processador utilizado neste trabalho permite coletar até 6 eventos simultaneamente, mas há também processadores que ofereçam apenas 4 contadores (como é o caso dos processadores ARM com arquitetura ARMv7). Uma sugestão para aprimoramento na estimativa do consumo de energia é aplicar a metologia proposta por Walker et al. (2017). Contudo, vale notar que ao encontrar uma fórmula com coeficientes específicos para um sistema, esta não apresentará a mesma acurácia quando utilizada em outro sistema. Sendo assim, haverá necessidade de ter uma estimativa específica para cada processador. Encontrar uma estimativa de consumo energético que seja acurada e ao mesmo tempo genérica é um desafio.

Aplicações paralelas podem apresentar variações na carga de trabalho durante a execução, fazendo com que o melhor número de *threads* para executar suas regiões paralelas mude. Situações externas, como outras aplicações executando junto e competindo pelos recursos do sistema, também podem alterar o número de *threads* que seria ideal para otimizar o EDP da aplicação. Contudo, a biblioteca desenvolvida neste trabalho realiza a adaptação no número de *threads* apenas uma vez, sem considerar estas possíveis mudanças. Desta forma, ao encontrar o melhor número de *threads*, este passa a ser utilizado até o final da execução.

Realizar monitoramento periódico, a fim de detectar alterações que justifiquem uma nova rodada da fase de aprendizagem, para então encontrar um novo número de *threads*, é outro ponto que precisa ser desenvolvido para aprimorar a biblioteca desenvolvida neste trabalho. Um cuidado que precisa ser considerado é o custo de monitorar a situação da aplicação. O monitoramento não pode ser constante e é preciso encontrar um intervalo ideal para realizá-lo, avaliando os custos e benefícios. Outra questão, é o critério que será utilizado para determinar se houve mudança e se realmente é preciso ser realizado uma nova busca por um novo melhor número de *threads*. Isto é um ponto importante visto há um custo gerado pela fase de aprendizagem, devido a necessidade de testar diferentes número de *threads*. Este custo leva um tempo até ser recuperado, e não pode exceder os benefícios gerados pela execução com o melhor número de *threads*, ou seja, é preciso encontrar um equilíbrio. Uma vez tendo esta característica implementada, é importante analisar o comportamento da técnica quando outras aplicações estão executando junto no mesmo sistema, e assim competindo por recursos.

Para que a adaptação do número de *threads* seja realizada, faz-se necessário que o código das aplicações paralelas sejam alterados, adicionando chamadas às funções da biblioteca em torno das regiões paralelas. Este trabalho foi realizado manualmente, mas pode ser automatizado. Se adicionado suporte de compilador, para demarcar as regiões paralelas identificando-as como recorrentes ou não recorrentes, além de não ter mais o trabalho manual, também não haveria necessidade da espera para entrar na fase de aprendizagem. A utilização de algoritmo baseado em *hill-climbing* para a busca do número ideal de *threads* permite reduzir a fase de aprendizagem da biblioteca, fazendo com que nem todas as possibilidades sejam avaliadas. Contudo, a busca pode terminar em um ótimo local. Por isto, avaliar outros algoritmos de busca é um trabalho que precisa ser realizado.

Neste trabalho as aplicações foram avaliadas apenas com um único conjunto de

entradas. Contudo, sabe-se que o tamanho do conjunto de entrada pode influenciar no melhor número de *threads* para executar uma aplicação paralela. Desta forma, avaliar a biblioteca proposta com outros conjuntos de entrada, além de outras aplicações, é um trabalho que precisa ser realizado. Os experimentos deste trabalho foram realizados com DVFS em modo *ondemand*, mas outros modos podem ser avaliados. Pode-se ainda, propor e implementar o ajuste dinâmico da voltagem e frequência, pela própria biblioteca, de acordo com as necessidades de cada aplicação. Neste trabalho utilizou-se processador *multicore* homogêneo, mas a extensão da técnica para considerar processadores com núcleos heterogêneos também é um trabalho futuro interessante, visto que estes já estão dominando o mercado de sistemas embarcados.

REFERÊNCIAS

ALESSI, F. et al. Application-level energy awareness for openmp. In: INTERNATIONAL WORKSHOP ON OPENMP (IWOMP), 2015, Aachen, Germany. **Proceedings...** [S.l.]: Springer, 2015. p. 219–232.

ARBENZ, P.; PETERSEN, W. **Introduction to Parallel Computing-A practical guide with examples in C**. [S.l.]: Oxford University Press, 2004.

ARDUINO. **Intel Galileo Gen2**. 2017. Disponível em: <https://www.arduino.cc/en/ArduinoCertified/IntelGalileoGen2>. Acesso em: 15 jan. 2018.

ARMBIAN. **Armbian: linux for ARM development boards**. 2017. Disponível em: <https://www.armbian.com/>. Acesso em: 21 set. 2017.

BAILEY, D. H. et al. The nas parallel benchmarks. **The International Journal of Supercomputing Applications**, Sage Publications Sage CA: Thousand Oaks, CA, v. 5, n. 3, p. 63–73, 1991.

BARI, M. A. S. et al. Arcs: Adaptive runtime configuration selection for power-constrained openmp applications. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING (CLUSTER), 2016, Taipei, Taiwan. **Proceedings...** [S.l.]: IEEE, 2016. p. 461–470.

BHALACHANDRA, S. et al. An adaptive core-specific runtime for energy efficiency. In: IEEE. **Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International**. [S.l.], 2017. p. 947–956.

CHADHA, G.; MAHLKE, S.; NARAYANASAMY, S. When less is more (limo): controlled parallelism forimproved efficiency. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURES AND SYNTHESIS FOR EMBEDDED SYSTEMS, 2012, Tampere, Finland. **Proceedings...** New York, NY, USA: ACM, 2012. p. 141–150.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION (IISWC), 2009. **Proceedings...** [S.l.]: IEEE, 2009. p. 44–54.

CURTIS-MAURY, M. et al. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In: ANNUAL INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 20., 2006, Queensland, Australia. **Proceedings...** New York, NY, USA: ACM, 2006. p. 157–166.

DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. **IEEE computational science and engineering**, IEEE, v. 5, n. 1, p. 46–55, 1998.

FREEH, V. W. et al. Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. **Journal of Parallel and Distributed Computing**, Elsevier, v. 68, n. 9, p. 1175–1185, 2008.

FRIENDLYARM. **NanoPi M3**. 2017. Disponível em: <http://wiki.friendlyarm.com/wiki/index.php/NanoPi_M3>. Acesso em: 26 dez. 2017.

INTEL. **Intel® Galileo Gen 2 Board**. 2017. Disponível em: <https://ark.intel.com/products/83137/Intel-Galileo-Gen-2-Board>. Acesso em: 15 jan. 2018.

JIN, H.-Q.; FRUMKIN, M.; YAN, J. The openmp implementation of nas parallel benchmarks and its performance. 1999.

KEATING, M. et al. Low power methodology manual: For system-on-chip design. Springer Publishing Company, Incorporated, 2007.

LEE, J. et al. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. **ACM SIGARCH Computer Architecture News**, ACM, v. 38, n. 3, p. 270–279, 2010.

LI, D. et al. Strategies for energy-efficient resource management of hybrid programming models. **IEEE Transactions on parallel and distributed Systems**, IEEE, v. 24, n. 1, p. 144–157, 2013.

LIVELY, C. W. et al. A methodology for developing high fidelity communication models for large-scale applications targeted on multicore systems. In: IEEE. **Computer Architecture and High Performance Computing, 2008. SBAC-PAD'08. 20th International Symposium on**. [S.l.], 2008. p. 55–62.

LORENZON, A. F. **Avaliação do desempenho e consumo energético de diferentes interfaces de programação paralela em sistemas embarcados e de propósito geral**. 2014. 166 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2014.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy. **Journal of Parallel and Distributed Computing**, Elsevier, v. 95, p. 107–123, 2016.

LORENZON, A. F.; SOUZA, J. D.; BECK, A. C. S. Laant: A library to automatically optimize edp for openmp applications. In: DESIGN, AUTOMATION & TEST IN EUROPE CONFERENCE & EXHIBITION (DATE), 2017. **Proceedings...** [S.l.]: IEEE, 2017. p. 1229–1232.

MARATHE, A. et al. A run-time system for power-constrained hpc applications. In: SPRINGER. **International Conference on High Performance Computing**. [S.l.], 2015. p. 394–408.

MUCCI, P. J. et al. Papi: A portable interface to hardware performance counters. In: **Proceedings of the department of defense HPCMP users group conference**. [S.l.: s.n.], 1999. v. 710.

NAVAUX, P. O.; ROSE, C. A. D.; PILLA, L. L. Fundamentos das arquiteturas para processamento paralelo e distribuído. **XI Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul-2011-Porto Alegre, RS**, p. 22–59, 2011.

PORTERFIELD, A. K. et al. Power measurement and concurrency throttling for energy reduction in openmp programs. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM WORKSHOPS & PHD FORUM (IPDPSW), 27., 2013, Cambridge, MA, USA. **Proceedings...** [S.l.]: IEEE, 2013. p. 884–891.

PUSUKURI, K. K.; GUPTA, R.; BHUYAN, L. N. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In: IEEE INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION (IISWC), 2011, Austin, USA. **Proceedings...** [S.l.]: IEEE, 2011. p. 116–125.

RABENSEIFNER, R.; HAGER, G.; JOST, G. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In: IEEE. **Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on**. [S.l.], 2009. p. 427–436.

ROUNTREE, B. et al. Adagio: making dvs practical for complex hpc applications. In: ACM. **Proceedings of the 23rd international conference on Supercomputing**. [S.l.], 2009. p. 460–469.

SENSI, D. D.; TORQUATI, M.; DANELUTTO, M. A reconfiguration algorithm for power-aware parallel applications. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM, v. 13, n. 4, p. 43, 2016.

SEO, S.; JO, G.; LEE, J. Performance characterization of the nas parallel benchmarks in opencl. In: IEEE. **Workload Characterization (IISWC), 2011 IEEE International Symposium on**. [S.l.], 2011. p. 137–148.

SRIDHARAN, S.; GUPTA, G.; SOHI, G. S. Adaptive, efficient, parallel execution of parallel programs. **ACM SIGPLAN Notices**, ACM, v. 49, n. 6, p. 169–180, 2014.

SULEMAN, M. A.; QURESHI, M. K.; PATT, Y. N. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. **ACM SIGOPS Operating Systems Review**, ACM, v. 42, n. 2, p. 277–286, 2008.

WALKER, M. J. et al. Accurate and stable run-time power modeling for mobile and embedded cpus. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 36, n. 1, p. 106–119, 2017.

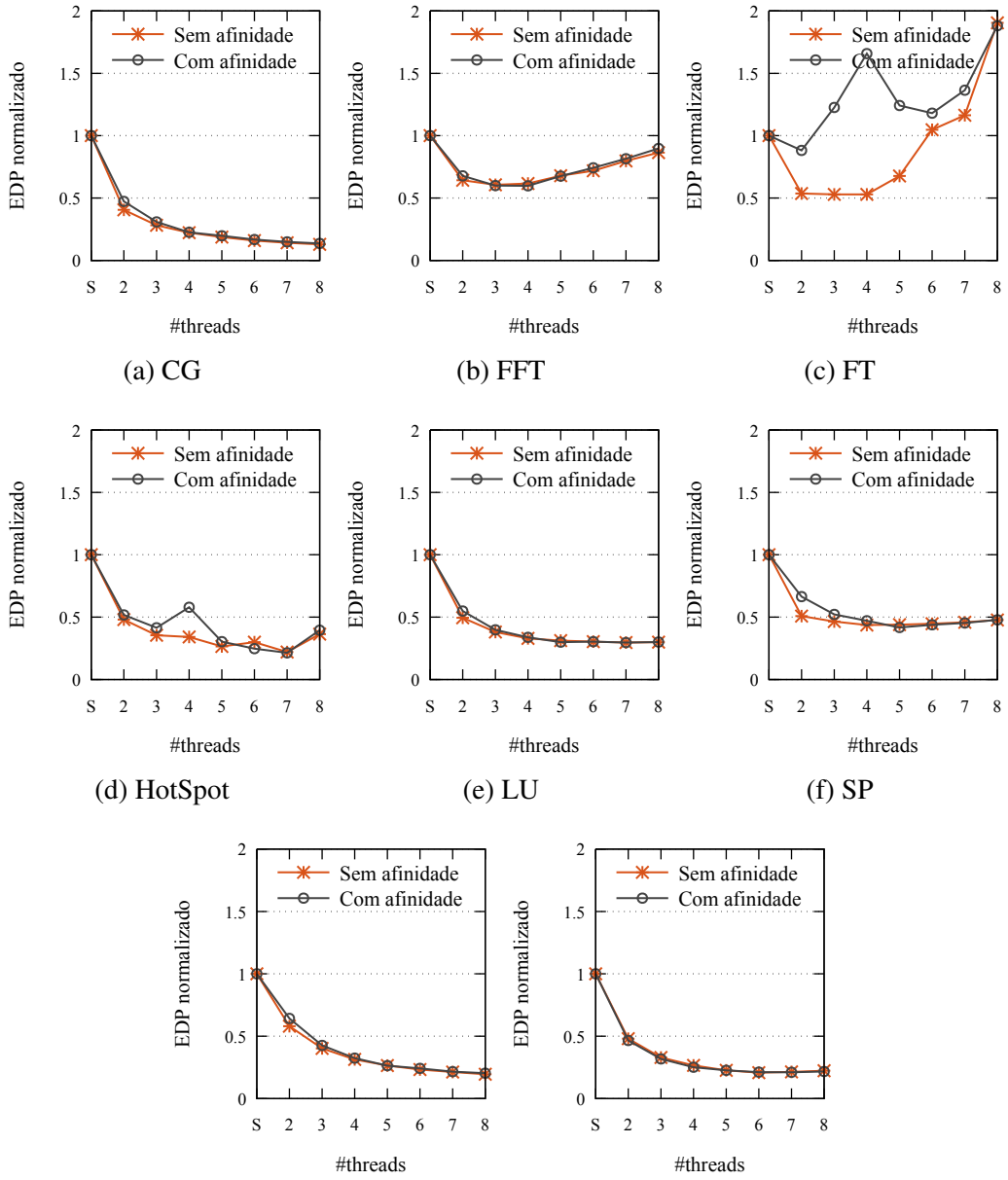
WANG, W. et al. Using per-loop cpu clock modulation for energy efficiency in openmp applications. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP), 44., 2015, Beijing, China. **Proceedings...** [S.l.]: IEEE, 2015. p. 629–638.

APÊNDICE A — RESULTADOS ADICIONAIS

A Figura A.1 apresenta a comparação dos resultados de EDP das execuções sem e com atribuição de afinidade. A afinidade de *threads* aos núcleos de processamento foi atribuída por meio da configuração da variável de ambiente GOMP_CPU_AFFINITY com o valor *0,1,2,3,4,5,6,7*. As aplicações foram compiladas com *flag* de otimização -O2. Os valores foram normalizados pela execução sequencial. Comparando os resultados da execução com a configuração padrão (máximo número de *threads*), não há diferença significativa entre a execução sem ou com atribuição de afinidade (Figura A.1). Entretanto, para algumas aplicações, as execuções realizadas com menor quantidade de *threads* sem afinidade apresentam maior margem para otimização (Figura A.1c).

Na Figura A.2 é apresentada a comparação entre os resultados obtidos com as aplicações compiladas com *flag* de otimização -O2 e -O3. Estes experimentos foram rodados com atribuição de afinidade das *threads* aos núcleos. Os valores foram normalizados pela execução sequencial da aplicação compilada com -O2. Pode-se notar que as *flags* de otimização -O2 e -O3 também não apresentam grandes diferenças, e não há uma *flag* que apresente melhores resultados para todas as aplicações (Figura A.2). A *flag* -O2, por exemplo, é melhor para a aplicação FFT, enquanto a *flag* -O3 é melhor para SP. O fator determinante para a escolha da *flag* -O2 foi a aplicação FT. FT não pode ser executada com 8 *threads* quando compilada com -O3. O sistema se desliga após um período de execução da aplicação não determinado, ou seja, a cada execução o sistema se desliga em um ponto diferente da execução. Acredita-se que isso seja uma medida de segurança devido ao aquecimento gerado no sistema.

Figura A.1: Comparação entre execuções sem e com atribuição de afinidade de threads aos núcleos. Aplicações compiladas com flag -O2.



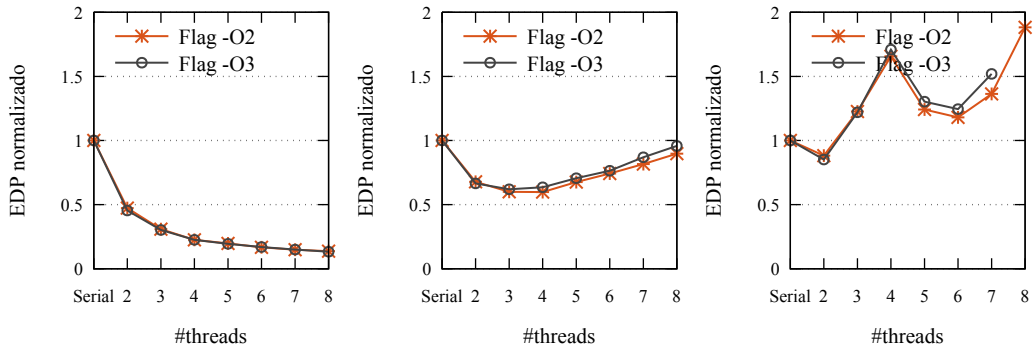
(a) CG (b) FFT (c) FT

(d) HotSpot (e) LU (f) SP

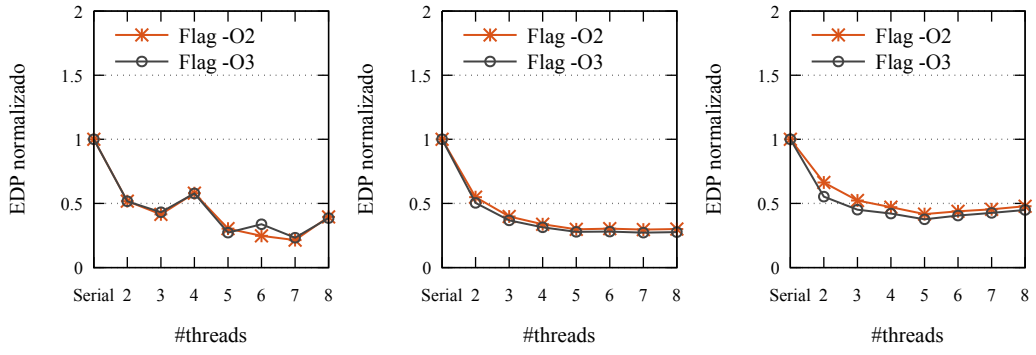
(g) SRAD (h) StreamCluster

Fonte: O autor

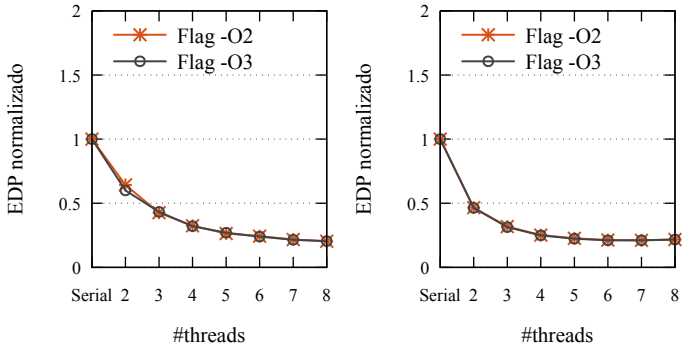
Figura A.2: Comparação entre as execuções das aplicações compiladas com flag -O2 e -O3. Experimentos com atribuição de afinidade.



(a) CG (b) FFT (c) FT



(d) HotSpot (e) LU (f) SP



(g) SRAD (h) StreamCluster

APÊNDICE B — RESULTADO MÉDIO E DESVIO PADRÃO DA EXPLORAÇÃO DE ESPAÇO DE PROJETO

Nas Tabelas B.1 a B.8 são apresentados os resultados médios de tempo de execução (em segundos), consumo de energia (em Joules) e EDP, juntamente com o desvio padrão, de cada aplicação quando executadas sequencialmente (S) e com diferentes número de *threads* (2 a 8 *threads*).

Tabela B.1: CG - Resultados de tempo de execução, consumo de energia e EDP com porcentagem de desvio padrão

Número de *threads*	**Tempo (seg)**	**Energia (J)**	**EDP**
S	22,22 (±0,62%)	11,82 (±3,06%)	262,82 (±3,28%)
2	10,86 (±0,65%)	9,81 (±2,59%)	106,51 (±2,69%)
3	7,54 (±0,27%)	9,83 (±1,93%)	74,17 (±2,00%)
4	5,83 (±0,33%)	10,04 (±1,58%)	58,53 (±1,59%)
5	4,81 (±0,32%)	10,23 (±1,40%)	49,24 (±1,46%)
6	4,12 (±0,49%)	10,16 (±1,61%)	41,8 (±1,74%)
7	3,63 (±0,68%)	10,25 (±1,23%)	37,22 (±1,54%)
8	3,26 (±0,55%)	10,38 (±2,28%)	33,87 (±2,42%)

Fonte: O Autor

Tabela B.2: FFT - Resultados de tempo de execução, consumo de energia e EDP com porcentagem de desvio padrão

Número de *threads*	**Tempo (seg)**	**Energia (J)**	**EDP**
S	20,44 (±0,20%)	14,75 (±2,84%)	301,55 (±2,82%)
2	13,77 (±0,13%)	14,07 (±1,70%)	193,79 (±1,73%)
3	12,12 (±0,12%)	15,08 (±1,29%)	182,73 (±1,27%)
4	11,35 (±0,08%)	16,38 (±1,55%)	185,86 (±1,54%)
5	11,35 (±0,19%)	18,04 (±1,37%)	204,85 (±1,43%)
6	11,19 (±0,13%)	19,37 (±1,52%)	216,84 (±1,59%)
7	11,34 (±0,30%)	21,2 (±1,31%)	240,44 (±1,37%)
8	11,35 (±0,30%)	22,98 (±1,64%)	260,73 (±1,84%)

Fonte: O Autor

Tabela B.3: FT - Resultados de tempo de execução, consumo de energia e EDP com porcentagem de desvio padrão

Número de *threads*	**Tempo (seg)**	**Energia (J)**	**EDP**
S	19,07 (±0,69%)	15,54 (±3,19%)	296,36 (±3,63%)
2	10,27 (±0,45%)	15,47 (±1,92%)	158,99 (±2,11%)
3	8,41 (±1,48%)	18,66 (±1,96%)	157,01 (±3,12%)
4	7,42 (±1,91%)	21,13 (±1,82%)	156,8 (±3,43%)
5	8,33 (±1,05%)	24,08 (±1,39%)	200,62 (±2,25%)
6	10,07 (±1,86%)	30,83 (±1,61%)	310,52 (±3,15%)
7	10,51 (±1,53%)	32,81 (±1,46%)	344,99 (±2,81%)
8	13,13 (±0,38%)	42,98 (±1,18%)	564,27 (±1,29%)

Fonte: O Autor

Tabela B.4: HotSpot - Resultados de tempo de execução, consumo de energia e EDP com porcentagem de desvio padrão

Número de *threads*	**Tempo (seg)**	**Energia (J)**	**EDP**
S	74,03 (±0,34%)	35,67 (±1,46%)	2641,18 (±1,57%)
2	39,13 (±0,77%)	32,45 (±1,88%)	1.269,73 (±2,39%)
3	28,51 (±3,47%)	32,83 (±1,82%)	936,31 (±5,03%)
4	25,48 (±5,30%)	35,29 (±2,57%)	900,12 (±7,68%)
5	20,21 (±7,34%)	34,61 (±3,57%)	701,25 (±10,83%)
6	20,6 (±8,08%)	38,30 (±4,17%)	791,64 (±11,98%)
7	15,97 (±6,73%)	36,29 (±3,77%)	581,11 (±10,48%)
8	21,4 (±1,98%)	44,95 (±1,40%)	962,24 (±3,11%)

Fonte: O Autor

Tabela B.5: LU - Resultados de tempo de execução, consumo de energia e EDP com porcentagem de desvio padrão

Número de *threads*	**Tempo (seg)**	**Energia (J)**	**EDP**
S	410,5 (±0,25%)	236,56 (±0,54%)	97108,81 (±0,59%)
2	225,48 (±0,14%)	213,09 (±1,07%)	48.046,38 (±1,10%)
3	165,24 (±0,16%)	223,79 (±0,68%)	36.977,71 (±0,75%)
4	136,85 (±0,24%)	234,00 (±1,64%)	32.024,52 (±1,88%)
5	122,85 (±0,15%)	246,74 (±0,48%)	30.312,32 (±0,56%)
6	113,86 (±0,07%)	259,33 (±0,36%)	29.528,26 (±0,37%)
7	106,87 (±0,12%)	269,12 (±0,26%)	28.759,39 (±0,34%)
8	103,58 (±0,07%)	281,13 (±0,36%)	29.118,75 (±0,39%)

Fonte: O Autor

Tabela B.6: SP - Resultados de tempo de execução, consumo de energia e EDP com porcentagem de desvio padrão

Número de *threads*	**Tempo (seg)**	**Energia (J)**	**EDP**
S	370,06 (±0,24%)	226,94 (±0,64%)	83983,21 (±0,75%)
2	197,84 (±0,44%)	215,67 (±1,53%)	42.669,65 (±1,93%)
3	160,6 (±0,32%)	242,86 (±0,50%)	39.003,92 (±0,72%)
4	137,85 (±0,20%)	266,41 (±0,38%)	36.724,74 (±0,53%)
5	129,82 (±0,45%)	284,76 (±0,43%)	36.966,62 (±0,64%)
6	124,37 (±0,11%)	301,9 (±0,94%)	37.548,17 (±0,97%)
7	121,91 (±0,07%)	316,07 (±0,36%)	38.533,18 (±0,39%)
8	120,15 (±0,06%)	333,65 (±0,34%)	40.088,61 (±0,36%)

Fonte: O Autor

Tabela B.7: SRAD - Resultados de tempo de execução, consumo de energia e EDP com porcentagem de desvio padrão

Número de *threads*	**Tempo (seg)**	**Energia (J)**	**EDP**
S	17,42 (±0,52%)	7,69 (±2,26%)	133,99 (±2,33%)
2	11,07 (±0,80%)	7,03 (±2,33%)	77,77 (±2,61%)
3	7,80 (±0,60%)	6,91 (±2,88%)	53,90 (±3,24%)
4	6,13 (±0,35%)	6,86 (±2,63%)	42,07 (±2,69%)
5	5,13 (±0,73%)	6,93 (±1,72%)	35,54 (±1,93%)
6	4,47 (±0,90%)	6,97 (±1,83%)	31,16 (±2,19%)
7	4,02 (±0,79%)	7,10 (±2,60%)	28,55 (±2,81%)
8	3,66 (±1,35%)	7,14 (±2,76%)	26,16 (±3,31%)

Fonte: O Autor

Tabela B.8: StreamCluster - Resultados de tempo de execução, consumo de energia e EDP com porcentagem de desvio padrão

Número de *threads*	**Tempo (seg)**	**Energia (J)**	**EDP**
S	239,78 (±0,10%)	143,1 (±3,81%)	34312,85 (±3,80%)
2	129,42 (±0,14%)	127,39 (±0,68%)	16.487,65 (±0,70%)
3	88,93 (±0,13%)	127,36 (±0,50%)	11.325,89 (±0,55%)
4	69,66 (±0,12%)	131,09 (±0,45%)	9.131,55 (±0,53%)
5	57,77 (±0,09%)	134,95 (±0,45%)	7.796,74 (±0,50%)
6	50,73 (±0,04%)	141,03 (±0,51%)	7.154,07 (±0,51%)
7	48,04 (±0,05%)	153,09 (±0,75%)	7.354,99 (±0,77%)
8	46,25 (±0,12%)	165,97 (±0,61%)	7.675,19 (±0,66%)

Fonte: O Autor