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

CHARLES CARDOSO DE OLIVEIRA

Odin: Online, Non-Intrusive and Self-Tuning DCT and DVFS to Optimize OpenMP Applications

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Antonio Carlos S. Beck

Porto Alegre
April 2019

CIP — CATALOGING-IN-PUBLICATION

Oliveira, Charles Cardoso de

Odin: Online, Non-Intrusive and Self-Tuning DCT and DVFS to Optimize OpenMP Applications / Charles Cardoso de Oliveira. – Porto Alegre: PPGC da UFRGS, 2019.

92 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: Antonio Carlos S. Beck.

1. Thread-level parallelism exploitation. 2. DVFS. 3. OpenMP. 4. Energy and performance optimization. 5. Run-time environments. I. Beck, Antonio Carlos S.. II. Título.

AGRADECIMENTOS

Primeiramente, agradeço a Deus por me ajudar a superar as dificuldades durante o desenvolvimento deste trabalho. Agradeço também ao apoio recebido pela família.

Agradeço ao professor Antonio Carlos Schneider Beck Filho pela orientação durante todo o mestrado. Agradeço ao Arthur Francisco Lorenzon e a Janaína Schwarzrock, por compartilharem da sua experiência na área para o desenvolvimento do trabalho.

Ao professor Luigi Carro, por suas ótimas aulas ministradas que contribuíram bastante para minha evolução na área. Ao professor Marcus Ritt, que auxiliou no desenvolvimento da técnica principal deste trabalho.

Finalmente, aos colegas de laboratório que sempre estiveram à disposição para esclarecer qualquer dúvida que surgisse.

ABSTRACT

Modern applications have pushed multithreaded processing to another level of performance and energy requirements. However, in most cases using the maximum number of available cores running at the highest possible operating frequency will not deliver the best Energy-Delay Product (EDP), since there are many aspects that prevent linear improvements when exploiting them. Moreover, the many parallel regions that compose an application may vary in behavior depending on characteristics that can be only known at run-time: input set, microarchitecture, and number of available cores. To solve this problem, we propose Odin: an online and lightweight self-tuning approach that optimizes OpenMP applications for EDP. While its dynamic nature makes it capable of adapting to the changing environment, it is totally transparent to both designer and end-user. Therefore, Odin does not need any source or binary code modifications, so potentially any dynamically linked parallel OpenMP executable file can be optimized with zero effort. By implementing different online strategies, we show that Odin can transparently improve EDP, on average, in 37.6% when compared to the regular OpenMP execution with DVFS set to ondemand. Additionally, we implement an alternative offline approach that uses a genetic algorithm for optimizing the parallel applications, showing that Odin can achieve similar results to it. Finally, we evaluate Odin's learning overhead and solution quality by comparing it to an exhaustive local search, which is the optimal configuration for each parallel region.

Keywords: Thread-level parallelism exploitation. DVFS. OpenMP. energy and performance optimization. runtime environments.

Odin: DCT e DVFS Online, Não-Intrusivo e Auto-Ajustável para otimizar aplicações OpenMP

RESUMO

Aplicações modernas têm levado o processamento paralelo a outro nível de requisitos em desempenho e energia. Entretanto, na maioria dos casos, usar o número máximo de núcleos disponíveis executando na maior frequência possível não oferecerá o melhor *Energy-Delay Product* (EDP), pois existem vários aspectos que impedem melhorias lineares ao explorá-los. Além disso, as várias regiões paralelas que compõem uma aplicação podem variar em comportamento dependendo de características que podem ser conhecidas apenas em tempo de execução: conjunto de entradas, microarquitetura e número de núcleos disponíveis. Para resolver esse problema, propomos Odin: uma abordagem de autoajuste online e leve que otimiza as aplicações OpenMP para EDP. Enquanto sua natureza dinâmica torna-o capaz de adaptar-se em um ambiente variante, ele também é totalmente transparente para ambos desenvolvedor e usuário final. Portanto, Odin não necessita de nenhuma modificação em código-fonte ou binário, logo potencialmente qualquer arquivo executável OpenMP que seja ligado dinamicamente pode ser otimizado sem nenhum esforço. Ao implementar diferentes estratégias online, nós mostramos que Odin pode de forma transparente melhorar o EDP, em média, em 37.6% quando comparado ao método regular de execução OpenMP com o DVFS configurado para *ondemand*. Adicionalmente, nós implementamos uma abordagem alternativa offline que usa um algoritmo genético para otimizar as aplicações paralelas, mostrando que Odin pode alcançar resultados similares a ela. Finalmente, nós avaliamos o custo de aprendizado e qualidade da solução de Odin comparando-o com uma busca local exaustiva, que é a configuração ótima para cada região paralela.

Palavras-chave: Exploração de paralelismo em nível de threads, DVFS, OpenMP, otimização de desempenho e energia, ambientes em tempo de execução.

LIST OF ABBREVIATIONS AND ACRONYMS

CMOS	Complementary Metal-Oxide-Semiconductor
DCT	Dynamic Concurrency Throttling
DVFS	Dynamic Voltage and Frequency Scaling
EDP	Energy-Delay Product
HPC	High-Performance Computing
ILP	Instruction-Level Parallelism
IoT	Internet of Things
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
RAPL	Running Average Power Limit
SMT	Simultaneous Multithreading
TLB	Translation Lookaside Buffer
TLP	Thread-Level Parallelism

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

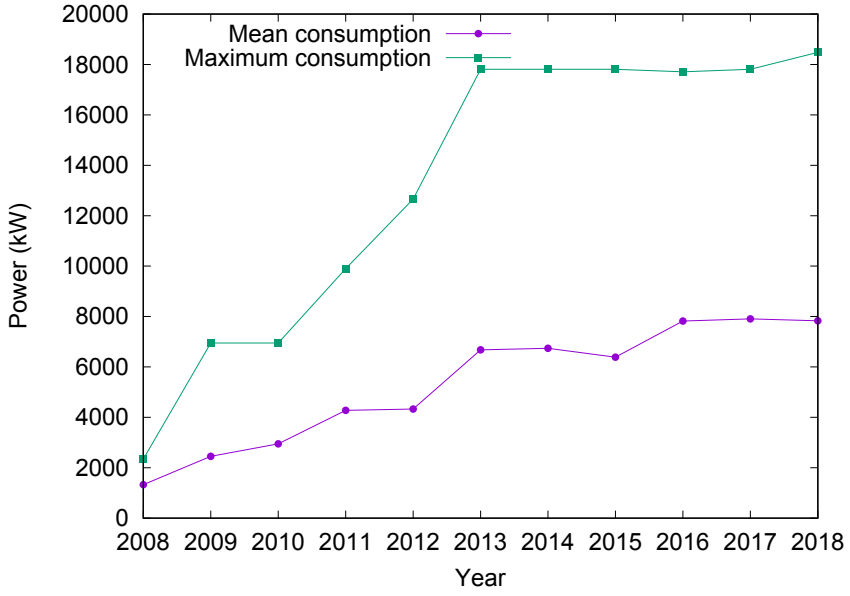
Demands for High-Performance Computing (HPC) systems have been growing due to technologies such as Cloud Computing, which has emerged as an essential computing paradigm, enabling ubiquitous and convenient on-demand access through the Internet to systems having configurable computing resources (KOBUSIŃSKA et al., 2018). Also, another notable technology trend that nowadays is gaining increasing attention is the Internet of Things (IoT). In IoT, intelligent embedded devices and sensors are interconnected in a dynamic and global network infrastructure (KOBUSIŃSKA et al., 2018). Besides that, devices such as smartphones and tablets complement this large embedded systems market.

For both fields, HPC and embedded devices, there are new applications that compute huge amounts of data, such as machine learning, smarter search mechanisms, big data in general, biomedical, and video and audio recognition, which have been pushing multithreaded processing to another level of performance requirements. However, power consumption is directly proportional to this increasing availability of data to compute.

Figure 1.1 shows the evolution of the power consumption (mean and maximum values) for the top 10 HPC systems in the last ten years based on the Top500 list (TOP500, 2019). As one can observe, on average, the power consumption had an increase of almost six times, while the system presenting the highest consumption grows up approximately eight times – exceeding the 18 MW mark – in the last decade. Therefore, the HPC systems are more likely to spend more funds on energy and cooling systems – as heating will increase from this large power dissipation – to maintain the services running. On top of that, a lot of current embedded systems are rapidly increasing in complexity, and hence, consuming more power. Since these systems are dependent on batteries, this rising complexity will decrease their battery lifetime.

Considering this scenario, the traditional way of executing parallel applications is using the maximum number of hardware threads available on the system at the maximum frequency or under the responsibility of the Operating System, which changes the frequency based on the CPU load. However, this conventional approach (i.e., the maximum number of threads and CPU frequency) does not always deliver the best performance and energy results. Therefore, this work uses two techniques for improving the outcome in performance and energy (expressed by Energy-Delay product, or EDP) of parallel applications: (*i*) **Dynamic Concurrency Throttling (DCT)**, which allows for changing the

Figure 1.1: The evolution of the power consumption based on the top 10 HPC systems of the Top500 list.



Source: The Author

number of threads of a parallel application at run-time; (*ii*) **Dynamic Voltage and Frequency Scaling (DVFS)** that dynamically adjusts the operating voltage and frequency of the CPU.

However, most applications are already deployed, which means that in many times the source code is not available for instrumentation and recompilation. Intel/ARM instruction set architectures (ISAs) have been showing that binary compatibility is mandatory, so one can reuse legacy code and maintain traditional programming paradigms and libraries. Hence, software transparency, which allows to optimize the target application without the need for recompiling it, plays a decisive role in the large adoption of any new solution. Thus, **we implement our framework to be transparent**, targeting the applications parallelized with OpenMP, which is a popular parallel programming interface.

1.1 Scalability and Variables Involved

As previously stated, not always the maximum number of available logical cores executing threads will provide the best outcome in energy and performance (represented by EDP), because many aspects prevent linear improvements as one increases the number of threads. Consider, for example, the Simultaneous Multithreading (SMT) technology, which permits more than one thread allocated to one physical core. The threads competing

for processor resources may cause a contention that degrades the overall performance (RAASCH; REINHARDT, 2003). Besides that, some applications make many memory requests and, as the off-chip bus has a limited bandwidth, the memory system can saturate as we increase the concurrency level (JOAO et al., 2012). Finally, a group of threads generally requires a synchronization point, which is a phase for data exchange that permits only one thread executing at the time, so increasing the number of threads may result in loss of benefits achieved in the parallel region (SULEMAN; QURESHI; PATT, 2008). We discuss in details each of these problems later.

Moreover, the current operating systems embed tools for power management, such as DVFS, which adapts the operating frequency and voltage at run-time according to the application at hand and has also been extensively exploited to improve energy (COCHRAN et al., 2011; LI; MARTINEZ, 2006; SENSI, 2016). A DVFS system takes advantage of the processor idleness (usually provoked by I/O operations or by memory requests) to achieve cubic power reduction, since voltage has quadratic influence in dynamic power. However, in the same way as the level of concurrency exploitation, using the maximum possible operating frequency will not always result in the highest EDP improvements, although it will very likely deliver the best performance. In this context, it is only natural to imagine that one may improve the EDP of a parallel application by tuning DCT and DVFS.

Besides that, one must also consider that parallel applications may comprise many parallel regions with different behaviors. Each of these regions may have an optimal number of threads that will deliver the best result in terms of performance or energy. Moreover, other aspects, such as the kind of executed instructions and amount of shared data may impact DVFS. Thus, considering that the system has **C** cores, **L** voltage/frequency levels, and **P** parallel regions for each program, we have $(C \times L)^P$ possibilities considering the global configuration targeting a specific non-functional requirement. Given the huge amount of possible solutions, this exponential behavior easily becomes impractical for an exhaustive search.

Based on this discussion, we can state two essential characteristics for optimizing parallel applications:

- **Adaptability:** related to the ability to adapt itself according to the application with respect to parameters that are known only at run-time, such as the number of available cores, system microarchitecture, and the input-set. In such cases, offline solutions will not suffice, since when the environment changes, the offline analysis

must be re-executed. In Chapter 5, we show a framework using a genetic algorithm that although the positive results, presents this limitation.

- **Transparency:** the lack of transparency comes from the necessity of transforming code, which can be manual or automatic (by using special languages or toolchains). This implies modifications in the source or binary codes, or requires the use of an Application Programming Interface (API) that is not conventional or widely used. However, as said before, many applications are already deployed, which means that in many times the source code is not available for instrumentation and recompilation. Most importantly, code annotation also involves the user, since it must analyze the source code and have particular knowledge of the application that must be optimized. Therefore, this task could only be performed by experienced programmers with full access to the code.

Finally, there are many solutions for DCT without DVFS and vice versa, but only a few that consider both. However, although some of them present adaptability, all of them lack software transparency.

1.2 Contributions

This dissertation makes the following contribution:

- Odin: a tool capable of automatically tuning, at run-time, the number of threads and DVFS for each parallel region of any OpenMP application. Because of its dynamic adaptability, Odin covers all cases discussed previously: it deals with the intrinsic characteristics of the application, the particularities of each parallel region, and can automatically adapt according to the microarchitecture, the number of available cores and the current input set, converging to either an optimal or a near optimal solution, and resulting in significant EDP gains. Odin was built on top of the original OpenMP library, it is completely transparent to both designer and end-user: given an OpenMP application binary, Odin runs on it without any code changes.

As a secondary contribution of this work, we also developed another framework to search for the optimal number of threads and CPU frequency level to execute each parallel region and that works at static time (prior execution). It is also automatic and transparent and built on top of the original OpenMP library. It implements an optimization algorithm based on a genetic algorithm that optimizes the entire application EDP. As an advantage,

this framework can search in a broader space of exploration because of its static essence. It also presents no costs w.r.t. to the learning overhead at runtime. For that, it has more opportunities to find near optimal solutions. On the other hand, it cannot benefit from many aspects that only dynamic strategies are offered.

1.3 Organization of this Dissertation

Chapter 2 depicts the theoretical concepts necessary to understand this dissertation. Firstly, we show notions of energy and power consumption. Then. how one can use the DVFS – a dynamic power management technique – to reduce the power consumed by a circuit. After that, we demonstrate a notion of parallel programming and how it works on the OpenMP framework. Finally, we study the scalability of parallel applications when applying DCT and DVFS.

Chapter 3 discusses the related work. We divide the chapter into three sections. First, we present those works that focus only on the optimization of the number of threads. Second, we show the researches that use only DVFS for optimization. Finally, we show studies that use both techniques to optimize parallel applications. Besides that, we highlight our contributions compared to the related work.

Chapter 4 shows the formal definition of the problem of optimizing parallel applications. Besides that, we give a short description of each algorithm that we use for our results comparison, which we divide between offline and online strategies.

Chapter 5 presents an alternative method, at static time, using a genetic algorithm to optimize parallel applications applying DVFS and DCT. First, we give the necessary background to understand the genetic algorithms, such as the main elements and operators. After, we show how we adapt it to our problem and the implementation bound to OpenMP. Finally, we show results comparing our tool to the usual way of executing parallel applications.

Chapter 6 presents the main contribution of this dissertation: Odin. Odin is a tool capable of automatically tuning the DCT and DVFS of an OpenMP application at run-time maintaining the software transparency. Besides that, we compare it to other techniques such as the framework using the genetic algorithm.

Chapter 7 presents the final considerations of this dissertation. It also discusses some points of improvement to our approach and promising future works.

2 FUNDAMENTAL CONCEPTS

This chapter presents the theoretical concepts necessary to understand the remainder of this work. First, we show the definition of energy and how it is related to power and time. Also, we explain what are the dynamic and static power consumption in an integrated circuit. Then, we introduce the Energy-Delay Product (EDP) metric, a formula that unifies performance and energy in a unique value. Secondly, the theory about parallel programming, models for exchange information between threads or processes, and the OpenMP interface. Finally, we dedicate a section to show how parallel application scales for both the number of threads and CPU frequency level.

2.1 Energy and Power Consumption

As mentioned in Chapter 1, the power consumption of the systems, both HPC and embedded devices, is growing up. Hence, it may imply more energy spent as can be seen in Equation 2.1 that gives the energy, in joules, consumed by a circuit.

$$Energy = \int P(t) \times dt \tag{2.1}$$

In Equation 2.1, *P(t)* is the power consumed at the instant of time *t* that is accumulated during a time interval. The **static** and **dynamic** power are the primary sources of power consumption in a Complementary Metal-Oxide-Semiconductor (CMOS) integrated circuit (KAXIRAS; MARTONOSI, 2008). We discuss each of them in the following subsections.

2.1.1 Dynamic Power

The power consumed mainly by the charge and discharge of the load capacitance when transistors switch is defined as dynamic power and given by Equation 2.2.

$$P_{dynamic} = C \times V^2 \times A \times f \tag{2.2}$$

Capacitance (*C*): shortly, aggregate the load capacitance and depends on both the capacitance of its transistors and the capacitance of its wires. Thus, the circuit designer

has a high influence on this component. For example, making smaller processor cores on-chip instead of a big one monolithic processor is likely to reduce wire lengths considerably, since most wires will interconnect units within a single core.

Supply Voltage (V): the main power source of the integrated circuit. Because of its quadratic influence on dynamic power, this presents excellent opportunities for power-aware design.

Activity Factor (A): the activity factor is a fraction between 0 and 1 that refers to how often wires transition from *high* to *low* and *low* to *high*. The clock signal, for example, has the activity factor of 1 as it is always switching between *low* and *high*.

Clock Frequency (f): has a direct impact on dynamic power. Besides that, the clock frequency maintains influence on the supply voltage because higher frequencies may require a higher supply voltage to the correct operation of the circuit. Therefore, combined with supply voltage it has a cubic impact on power consumption.

2.1.2 Static Power

Dynamic power dissipation still represents the predominant factor in CMOS power consumption, but leakage energy has been increasingly prominent in recent technologies. Static power consumption is due to the imperfect essence of transistors that permits leakage currents (Equation 2.3). Thus, the integrated circuit is always consuming power even when it is not switching (KAXIRAS; MARTONOSI, 2008).

$$P_{static} = V \times I_{leak} \tag{2.3}$$

In Equation 2.3, V is the supply voltage and I_{leak} is the leakage current.

2.1.3 Energy-Delay Product

At the same time that users want to maximize the performance of their applications, there are demands to reduce energy consumption. Embedded systems, for example, benefit from this by increasing the battery lifetime. While one can use DCT to optimize performance and energy in parallel applications, the DVFS aims to reduce power consumption and possibly decrease the energy spent – as energy depends on the execution time, this is not always achievable. Furthermore, to reach the maximum performance

using DVFS, we need to hold the frequency on the maximum level.

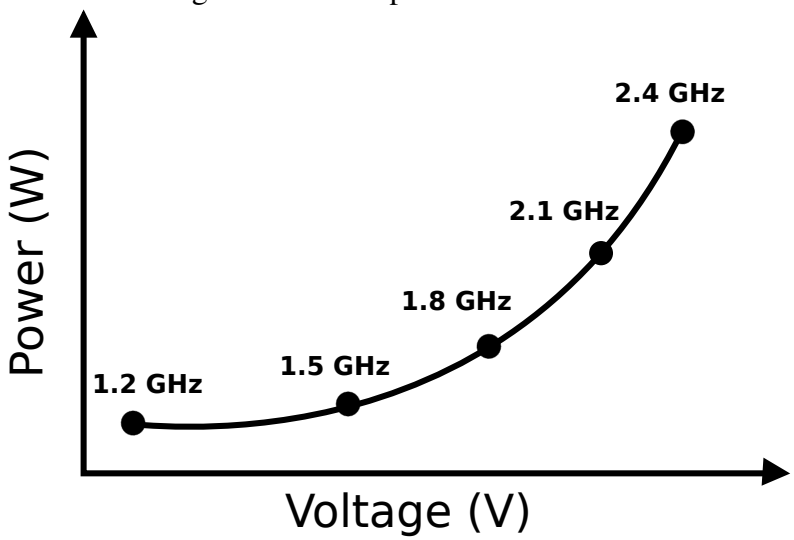
Thus, to optimize energy without a high degrading of the performance and vice versa, we use the Energy-Delay Product (EDP) as a metric for our work. The EDP is a metric that unifies energy and performance (delay) in a unique value as shown in equation 2.4 (GONZALEZ; HOROWITZ, 1996). Therefore, this metric transforms a multi-objective problem in a single objective form, helping the optimization procedure. Also, there are other metrics such as ED^2P, which give to performance more influence. Our tool can use it with minimal effort, but we focus our results only for EDP.

$$EDP = energy \times delay \tag{2.4}$$

2.1.4 Dynamic Voltage and Frequency Scaling for Power Management

Dynamic voltage and frequency scaling is a widely used technique for reducing power/energy consumption. The frequency at which the circuit operates determines the voltage required for stable operation, so decreasing it we can also reduce the voltage. Figure 2.1 depicts an example of a DVFS system that a processor can use, where each point refers to an operating pair of voltage/frequency and the reduction of power consumption. Thus, a DVFS system can take advantage of the processor idleness to achieve cubic dynamic power reduction, since voltage has quadratic influence in dynamic power as shown in Equation 2.2, without performance impact.

Figure 2.1: Example of DVFS levels



Source: The Author

According to Kaxiras and Martonosi (2008), there are three major levels of processor slackness in which DVFS decisions can be made:

1. System-level based on system slack: at this level, the idleness of the whole system determines the DVFS choices. In many cases, it considers the use of the CPU load to make subsequent decisions, as the Linux Operating System does through the *ondemand* governor (PALLIPADI; STARIKOVSKIY, 2006).
2. Program- or program-phase-level based on instruction slack: the decisions are based on program phases behavior, for example, a DVFS system can exploit phases that present long-latency memory operations.
3. Hardware-level based on hardware slack: finally, there is an approach that goes below the program level, right to the hardware. It tries to exploit slack hidden in hardware operation.

In this dissertation, we focus on the program phase granularity (level 2), in which we choose a DVFS setting for each parallel region of the application, aiming for the EDP optimization of the entire execution. Therefore, we consider the exploitation of operating frequencies in a lower granularity than the approach using DVFS decisions level 1 without hardware modification (level 3), thus being possible to implement on most current processors.

Besides, although our system has the possibility of changing the frequency level for each physical core individually, in this work we are considering the frequency switching for the entire package. Considering that we need a system call (syscall) to change the CPU operating frequency at the application level for each core, which is an expensive task, we also implemented a special governor using a kernel module to reduce the overhead for the frequency switching process. It comprises a single syscall that changes the operating frequency of the whole package. With the module, we have diminished the cost from about 1 ms to approximately 100 μs for each frequency changing (considering a processor with 24 cores). The user can insert or remove the governor dynamically on Linux, with no need to recompile the kernel. Therefore, the transparency for the application to be optimized is maintained.

2.2 Parallel Programming

Parallel programming is the process of dividing a set of tasks to be executed concurrently and, therefore, reducing the execution time of an application. It focuses on exploiting Thread-Level Parallelism (TLP) that is a high-level alternative way to exploit parallelism, in contrast with Instruction-Level Parallelism (ILP) that has significant limitations for some applications (JOUPPI; WALL, 1989).

The popularization of multicores in both desktops and embedded devices make the parallel programming a requirement to ensure high performance for taking advantage of the hardware resources. For example, we see parallel computing from mainstream applications such as a web browser and a text editor to scientific applications – e.g., fluid dynamics simulation.

Parallel applications require a method to exchange information between the threads or processes in run-time. There are two main models to implement it:

- **Shared-memory:** this model assumes that programs will be executed on one or more processors that share the available memory address space. Shared-memory programs are typically performed by multiple independent threads; the threads share data but may also have some additional, private data. Shared-memory approaches to parallel programming must provide, in addition to a normal range of instructions, a means for starting up threads, assigning work to them, and coordinating their accesses to shared data, including ensuring that certain operations are executed by only one thread at a time (CHAPMAN; JOST; PAS, 2007). OpenMP and PThreads are parallel programming interfaces that implement a shared-memory model.
- **Message-passing:** this model assumes that programs will be executed by one or more processes, each of which has its own private address space. Message-passing approaches must provide a mechanism to initiate and manage the participating processes, along with operations for sending and receiving messages, and possibly for performing specialized operations across data distributed among the different processes (CHAPMAN; JOST; PAS, 2007). The Message Passing Interface (MPI) is an example of standard that uses this model.

While PThreads and MPI present a set of routines libraries in which the programmer has to designate details of the parallel execution, OpenMP takes a form of additional

Figure 2.2: Example of parallel program using OpenMP.

```
#include <stdio.h>
#include <omp.h>

int main()
{
    int i;
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

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

    /* Sequential region */

    return 0;
}
```

Source: The Author

instructions to the compiler, which is expected to use them to generate the parallel code. Therefore, the use of OpenMP is usually more straightforward for the programmer.

2.2.1 OpenMP

OpenMP is a parallel programming interface for shared memory in C/C++ and FORTRAN that uses shared memory for communication between threads. It permits the user to parallelize its code only by using compilation directives. These directives inform the compiler of the regions for parallel execution and OpenMP takes care of the low-level steps, for example, thread creation and synchronization. Therefore, as already mentioned, it usually requires less effort to extract parallelism when compared to other APIs (e.g., PThreads and MPI), making it more appealing to software developers (S. et al., 2011). Figure 2.2 shows an example of a simple array processing using the directive *omp parallel for* to parallelizing the loop.

It supports the so-called fork-join programming model. Under this method, the process starts as a single thread, just like the sequential program. Whenever a thread finds an OpenMP parallel construct while it is executing the application, it creates a team of threads (this is the fork), becomes the master of the team, and cooperates with the other members of the team to execute the code dynamically enclosed by the construct. At the end of the construct, only the original thread, or master of the team, continues; all others

terminate (this is the join). The parallel region is the piece of code enclosed by a parallel construct (CHAPMAN; JOST; PAS, 2007).

Furthermore, OpenMP provides three ways for exploiting parallelism: parallel loops, sections, and tasks. Sections and tasks are only used in very particular cases: when the programmer must distribute the workload between threads in a similar way as PThreads, and when the application uses recursion (e.g. in sort algorithms). On the other hand, parallel loops are used to parallelize applications that work on multidimensional data structures (e.g arrays or grids), so the loop iterations (*for*) can be split into multithread executions. Therefore, parallel loops are by far the most used approach and all popular OpenMP benchmarks are implemented this way.

2.3 Scalability of Parallel Applications

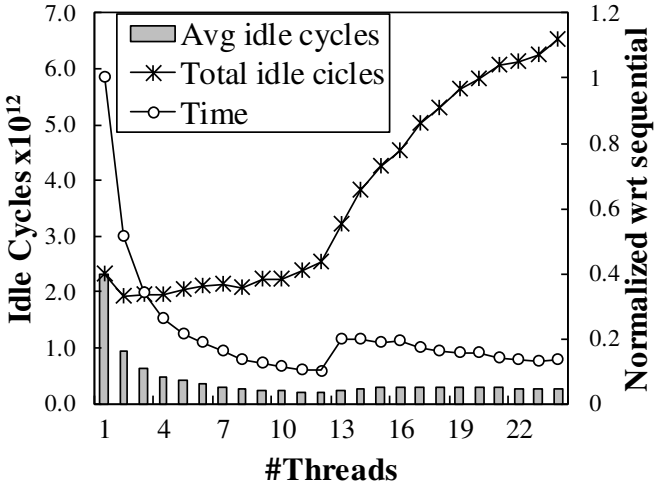
In this section, we discuss how parallel applications scale under the number of threads and CPU frequency – although the latter is not exclusive to parallel applications, the parallelism can contribute to the lack of scalability. First, we show some bottlenecks that prevent or even worsen the outcomes of parallel applications when one increases the number of threads. Next, we explain that some applications will barely improve the performance as one increases the CPU frequency, so we can set a low DVFS setting to reduce energy consumption, and consequently improve the EDP result.

2.3.1 Number of Threads

Not always selecting the maximum available number of cores to execute a parallel application will deliver the best results, since many aspects prevent linear improvements as one increases the number of threads. We discuss them in the following paragraphs.

Issue-width saturation: the SMT technology permits that two threads run simultaneously into the same physical core sharing functional units. However, the use of SMT when executing applications that present high Instruction-Level Parallelism (ILP) can lead to more competition for resources resulting in functional unit contention, consequently degrading performance (RAASCH; REINHARDT, 2003). Figure 2.3 presents the performance speedup relative to the sequential version and the number of idle cycles (average, represented by the bars, and total) as we increase the number of threads for the

Figure 2.3: Issue-width saturation



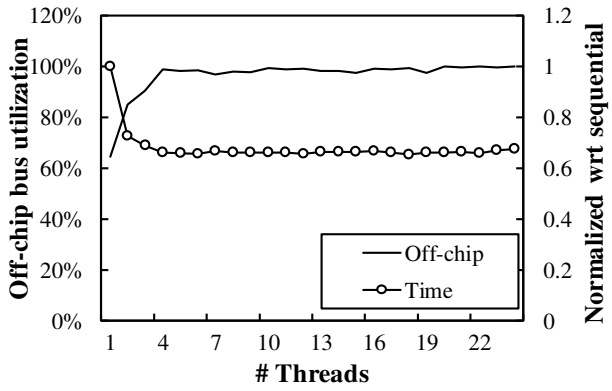
Source: (LORENZON et al., 2018)

HotSpot (HS) application. As the benchmark executes with 13 threads or more, some threads will share the same physical core because the SMT is activated. After enabling the SMT, the average number of idle cycles increases by a small amount or stays constant, while the total number of idle cycles significantly increases, which prevents improvements in performance.

Off-chip bus saturation: when dealing with a massive amount of data, the application will end up being dependent on the main memory as the private caches will not have sufficient storage space. The problem is that the off-chip bus bandwidth is limited compared with the number of cores. Therefore a higher number of threads will increase the memory requests that can lead to saturation of this resource (JOAO et al., 2012). Figure 2.4 shows the Fast-Fourier Transform (FFT) execution as an example. As the number of threads increases, the execution time diminishes until the off-chip bus becomes completely saturated (100% of utilization). After this point (4 threads), increasing the concurrency will not improve the performance because the bus is slowly processing all the requested data.

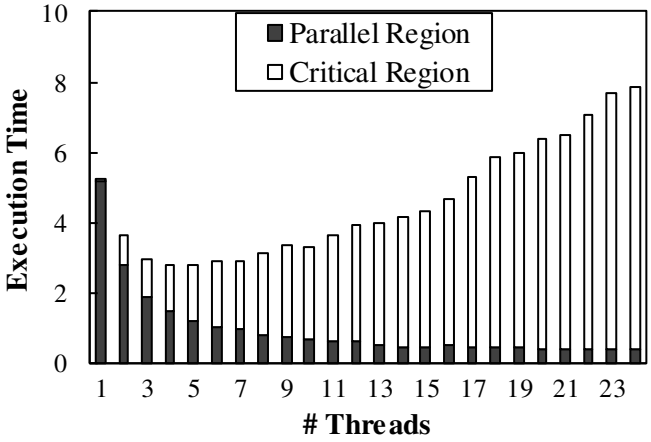
Data-synchronization saturation: the threads in a parallel application sometimes need to communicate with each other. To ensure data synchronization and integrity is necessary a critical section at the end of a parallel region. Critical sections allow only one thread execution at a time, i.e., each thread runs sequentially in this stage. Therefore, the higher is the number of threads more time will need to execute the critical section that may end up in loss of benefits achieved by the parallel phase (SULEMAN; QURESHI; PATT, 2008). Figure 2.5 presents the execution time broken on the critical and parallel

Figure 2.4: Off-chip bus saturation



Source: (LORENZON et al., 2018)

Figure 2.5: Data-synchronization saturation



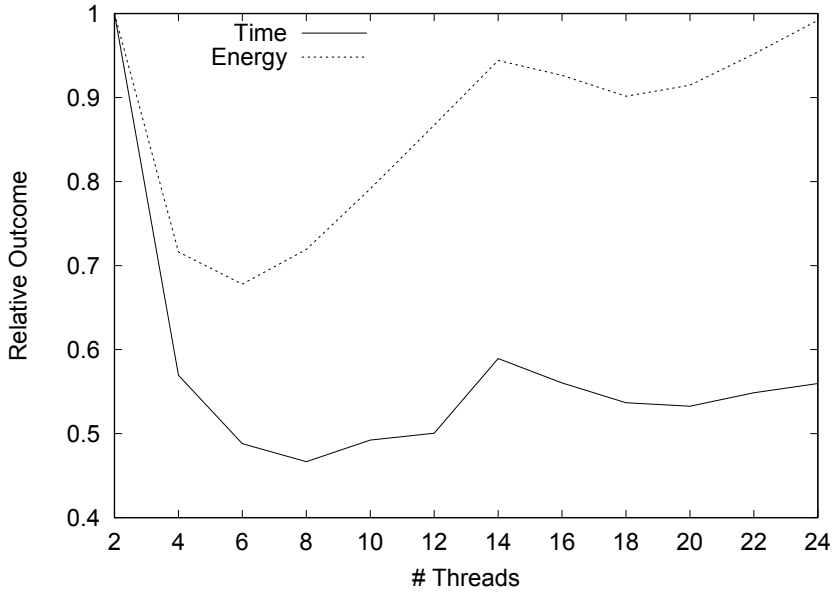
Source: (LORENZON et al., 2018)

region for the benchmark N-Body (NB) for each DCT configuration. After four threads, the critical phase exceeds the parallel region, so the performance begins to worsen.

Besides that, the best DCT configuration to optimize energy or performance can be different as there is a trade-off between achieving higher speed splitting the workload in more threads and extra power consumption caused by more resources working.

Figure 2.6 shows outcomes for time and energy normalized to the minimum parallelism (two) from one parallel region of the Block tri-diagonal solver (BT) benchmark running on a 24 cores system (24 hardware threads/12 physical cores). First, we can see that the region has poor scaling that will give unsatisfactory results running on the maximum number of threads. Besides that, while six threads ensure the optimum energy point, we need eight threads for the best result in performance.

Figure 2.6: Impact of the number of threads on the performance and energy.



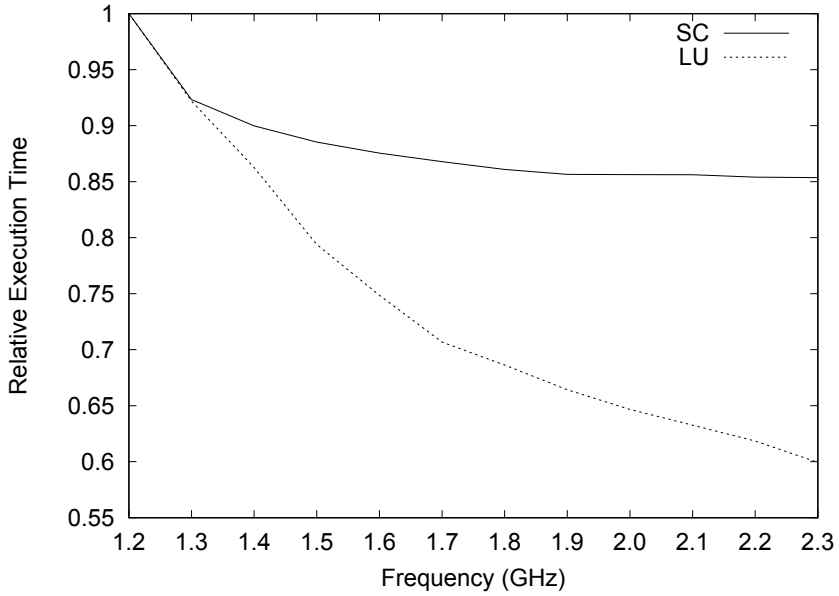
Source: The Author

2.3.2 CPU Frequency

As already described in Section 2.1, a DVFS system can take advantage of processor idleness and reduce the frequency level to decrease power consumption. For example, when an application makes many memory requests and has to wait for data from DRAM. In this case, increasing the processor frequency will not linearly improve the application performance.

Figure 2.7 shows the impact of the frequency scaling in the execution time of two applications running in a system with 24 threads and 12 frequency levels. The x-axis represents each available DVFS configuration on the processor, and the y-axis gives the execution time relative to the lowest frequency. While the StreamCluster (SC) that is a memory-bound application, shows minimal performance improvement after a certain frequency level, the Lower-Upper gauss-seidel solver (LU), a CPU-bound benchmark, maintains a consistent decrease of execution time till the maximum frequency. Thus, applications like SC permits that we use DVFS to apply a low frequency reducing power consumption with minimal performance impact consequently optimizing the application EDP.

Figure 2.7: Impact of the frequency on the execution time.



Source: The Author

3 RELATED WORK

We discuss the related work following the same order as listed in Table 3.1. We classify them into three categories: techniques that optimize the number of threads only; those that apply DVFS; and approaches that consider both. Works in each category are organized according to their adaptability:

- **Offline:** these works use an algorithm to learn the optimum or the near optimum configuration offline. After defining the best settings, the application loads it during the execution time. The drawbacks of this strategy are the need for training always that a characteristic of application or system changes, and the long learning time.
- **Hybrid:** these tools divide the adaptation process into two steps: offline and online phases. The former uses an algorithm with example cases, such as linear regression, to build a training a model. The latter uses the training model with dynamic data for online adaptation. Besides the long time needed for preparing the model, it also needs different training on each specific system. Moreover, in the same way as before, the cases used to train the model may miss some critical characteristics that can only appear at run-time (e.g., application input set and CPU load).
- **Online:** for these cases, the algorithm learns the configuration at run-time. They use some samples for training, find the best outcome and apply it to the next iterations. As to the disadvantages, there is an overhead caused during the learning step. Therefore, it needs many repetitions for a given target parallel region to overcome its cost.

Finally, we also use **transparency** for classifying the works. It is the capacity of adapting the application without code modification or instrumentation, that is, given the application binary, the tool can optimize it.

3.1 Adaptation of the Number of Threads

Thread Reinforcer (TR) (PUSUKURI; GUPTA; BHUYAN, 2011) works offline and involves the execution of the application binary multiple times for a short period with a different number of threads to find the appropriate configuration. Then, the application is fully re-executed with the number of threads previously defined. By executing the application binary already compiled, TR is a particular case that keeps binary compatibility.

Some approaches present some degree of adaptability, such as (JUNG et al., 2005), (CURTIS-MAURY et al., 2006), (CURTIS-MAURY et al., 2008), and (RAMAN et al., 2012). They are capable of changing the number of threads at runtime, although they need some previous offline analysis, instrumentation code, or a special compiler. Jung et al. (2005) present performance models for generating adaptive parallel code for SMT architectures. At compile-time, the preprocessor inserts performance estimation using a formula based on iterations, assignments, floating point operations, and function calls. Also, it injects instrumentation code using machine specific parameters for online analysis. At run-time, the master thread uses the analysis performed at compilation time to dynamically estimate whether it should enable the SMT in the core or not.

Curtis-Maury et al. (2006) propose a framework that has an off-line phase in which data from hardware event counters are collected to profile the parallel execution through a linear model to predict IPC. These trained models – one for each configuration target – are used at runtime to adapt the number of threads per processor and how many processors to use in each parallel region aiming to improve performance and energy consumption. Likewise, (CURTIS-MAURY et al., 2008) dynamically changes the number of threads and affinity of parallel regions that were identified by the programmer by using multivariate linear regression, which is trained off-line with performance counters. Also, the framework needs instrumentation in the target application.

In (RAMAN et al., 2012), the authors developed a particular compiler, Nona, that identifies parallelizable regions on a sequential code and applies multiple types of parallelism to each region. Also, it inserts profiling hooks for a run-time system to monitor its performance. The Parcae run-time system includes the Decima monitor and the Morta executor. The former can distinguish the time a task spends doing real computing and the overhead for communication. Finally, the latter can use the information from Decima to find the optimal or the near optimal parallelism configuration for the execution environment focusing on objectives such as minimize total execution time and energy consumption.

The following strategies that are online. In (SULEMAN; QURESHI; PATT, 2008), (PORTERFIELD et al., 2013), (LEE et al., 2010), (SRIDHARAN; GUPTA; SOHI, 2013), (SRIDHARAN; GUPTA; SOHI, 2014), (SHAFIK et al., 2015a), (LORENZON; SOUZA; BECK, 2017), and (LORENZON et al., 2018), the process of adjusting the concurrency level is totally at run-time.

Suleman et al. propose the FDT framework (SULEMAN; QURESHI; PATT,

2008). After code transformation of an application implemented with OpenMP, FDT samples portions of the application at run-time to estimate the application behavior, which is used to find the number of threads that saturates the parallel region performance so that the next iterations will run with fewer threads than the saturation point. During the training phase, it focuses on two performance bottlenecks: data-synchronization and bus bandwidth.

Porterfield et al. (2013) propose an adaptive run-time system that automatically adjusts the concurrency level based on online measurements of system resource usage. The framework extends Qthreads (a general purpose parallel library that is designed to support lightweight threading (WHEELER; MURPHY; THAIN, 2008)) and combines it with the running average power limit (RAPL) interface (HäHNEL et al., 2012) in the Intel architecture to build a scheduler that automates dynamic concurrency throttling. Thus, it will limit the level of parallelism in regions of code where power consumption is high and contention for a shared resource limits execution performance.

In Thread Tailor (LEE et al., 2010), a static software tool chain creates as many threads as can potentially be used by the architecture. At runtime, a Just-in-Time compiler takes a quick snapshot of the system state to determine how many free resources are available (e.g., number of available cores or free cache space) to apply thread throttling, generating code for that and redirecting the calls to increase system efficiency.

ParallelismDial (PD) (SRIDHARAN; GUPTA; SOHI, 2013) is a model that optimizes a program's execution efficiency by dynamically and continuously adapting the application parallelism to the execution conditions. To dynamically adjust software parallelism, PD firstly assesses the efficiency of the system using a proposed metric, Joules per instruction, that manifests effects in both instruction rate and energy expended. Also, it detects contention, variations in the program, and changes in the available resources. After these changes, it seeks to find and move the execution to the optimum degree of parallelism employing a heuristic based on the hill-climbing search algorithm. Finally, for continuous adaptation, PD periodically repeats the mentioned steps.

In (SRIDHARAN; GUPTA; SOHI, 2014), PD was extended to Varuna, which comprises an analytical engine which continuously monitors changes in the system using hardware performance counters to determine the optimum degree of parallelism; and a manager that regulates the execution to match the degree of parallelism previously defined. PD and Varuna comprise a monitor system that intercepts thread and task creation from PThreads, TBB, and Prometheus libraries, and create a pool of tasks to optimize

their degree of parallelism, creating a large number of fine-grained tasks, requiring more effort from the programmer.

Shafik et al. (2015a) propose an energy minimization model for OpenMP programs that involves code annotations that must be inserted in the code with specific performance requirements; which will be used by the run-time system to minimize energy.

LAANT (LORENZON; SOUZA; BECK, 2017) is a library that automatically adjusts the number of threads for optimizing the EDP of OpenMP applications. The code must be modified by the programmer to include additional function calls in each parallel region of interest. Besides that, it uses a finite state machine to implement a heuristic based on a hill-climbing algorithm.

All of these aforementioned works need code recompilation. Considering the case of LAANT, it was extended to Aurora (LORENZON et al., 2018), which can adapt the degree of TLP exploitation of OpenMP applications transparently. In order to achieve such level of transparency, the authors have implemented the search algorithm used by Aurora inside of the OpenMP library (libgomp), which is dynamically linked to the application at runtime. Therefore, any application can benefit from Aurora without modifications in the source code or recompilation.

3.2 CPU Frequency Level

Rossi et al. (2015) propose an offline approach using a multiple linear regression model based on DVFS and CPU usage that estimates the power consumption. It is implemented for different DVFS policies: performance (frequency is always at the maximum level), ondemand (frequency is adjusted according to the workload behavior), and powersaving (frequency is always at the minimum level). Thus, the programmer can use the predicted values to select the DVFS policy that provides the lowest power consumption.

In (HOTTA et al., 2006), the authors propose PowerWatch, a power-performance optimization model that adapts the processor frequency at run-time but relies on an off-line phase. In the approach, a parallel application is split into several regions by the programmer. Then, each region is executed with different processor frequencies during the off-line phase. Finally, the optimization algorithm determines the best processor operating frequency for each region and re-run the application with such frequency values.

Another hybrid approach is the Pack & Cap (COCHRAN et al., 2011), which manages the CPU voltage-frequency setting and the use of thread affinity (but do not

perform Thread Throttling) to optimize performance within a power budget. It consists of an offline phase where a large volume of data (performance, energy, temperature) are collected to train a multinomial logistic regression (MLR) classifier. Then, at runtime, the MLR classifier selects the optimal or the near optimal configuration to execute the rest of the application.

Next, we discuss run-time approaches, which do not need off-line analysis but need specific compilers/tools to enable the online adaptation.

DEP+BURST (AKRAM; SARTOR; EECKHOUT, 2016) is an online DVFS performance predictor to manage multithreaded applications that run on top of the Java virtual machine. It presents two key components, DEP and BURST. The former handles synchronization and inter-thread dependencies. It decomposes the execution time of a multithreaded application into epochs based on its synchronization activity identified by intercepting specific system calls used in multithread libraries. The latter identifies store operations that are on the application's critical path and predicts their impact on performance across frequency settings.

In (WU et al., 2006), the authors propose a dynamic compiler system that is a runtime software that compiles, changes and optimizes a program's instruction sequences. First, the framework selects frequently executed and long-running code regions to optimization such as loops and functions. After, it decides whether applying DVFS is beneficial for the candidate regions and determines the appropriate DVFS setting using hardware feedback information. Finally, for the regions where DVFS is useful, the tool inserts instructions at the entry point to start DVFS and at the exit point to restore the default configuration aiming to reduce energy consumption with a little performance impact.

Finally, we present online and transparent mechanisms to optimize the DVFS settings.

Hsu and Feng (2005) propose an automatic power-aware run-time system that adapts the CPU operating frequency to reduce energy consumption with minimal performance slowdown. The algorithm is an interval based scheduling that makes decisions at the beginning of time intervals of the same length, for example, every one second. The authors propose a model based on MIPS (Millions of Instruction Per Second) that associates the intensity of off-chip accesses to correlate the CPU frequency impact on the execution time.

Rizvandi et al. (2010) propose a maximum-minimum-frequency DVFS algorithm (MMF-DVFS). It uses a linear combination of the maximum and minimum processor

frequencies to reduce the energy consumption with minimal impact on the system's performance.

Ge et al. (2007) present the CPU Management Infra-Structure for Energy Reduction (CPU-MISER), a run-time DVFS scheduler for multicore-based power aware clusters. It consists of a monitor that collects performance events from the application using hardware counters and predicts the application's workload. Based on the predicted value, the DVFS scheduler determine the CPU frequency for the rest of the application.

In (MIFTAKHUTDINOV, 2014), the authors propose a performance predictor to control the CPU frequency level at runtime. The model measures the workload characteristics for each parallel region and estimates the performance at different CPU frequency levels. Then, when the region is re-executed, the CPU frequency is set to the level that offers the best performance.

Chen et al. (2016) also propose a model with the same purpose, but to predict the best CPU frequency level and voltage for multicore embedded systems aiming to reduce the energy consumption. In the approach, the user must define a given performance loss factor so the model can reduce the energy consumption accordingly.

3.3 CPU Frequency Level and Number of Threads

The aforementioned works apply either DVFS or Threads. More complete solutions that consider both are discussed here.

An offline approach is proposed by De Sensi (SENSI, 2016). It predicts the number of threads and CPU frequency level that offers the best performance and energy consumption for parallel applications. The idea is to execute the program using few configurations and then, predict the behavior of the other settings through multiple linear regression.

Hybrid (Offline+Online) approaches include (LI; MARTINEZ, 2006) and (LI et al., 2010). The former is divided into three phases: (*i*) the application is executed once for every combination (thread number and DVFS level), and energy and performance are collected; (*ii*) different optimization mechanisms are simulated with Matlab to find the combination that delivers the best result in energy under given performance restrictions (*ii*) At run-time, the approach uses the best combination found in phase *ii* to optimize the execution. In (LI et al., 2010), the authors propose a library for hybrid MPI/OpenMP applications by selecting the appropriate number of threads and CPU frequency to execute

each OpenMP region. The library has an off-line phase to train a model that will be used at runtime for each OpenMP region. The user has to instrument the applications with functions calls around each OpenMP region and selected MPI operations.

Finally, there are fully online approaches that are worth mentioning. They are (ALESSI et al., 2015), (SENSI; TORQUATI; DANELUTTO, 2016), (MARATHE et al., 2015), and (CHADHA; MAHLKE; NARAYANASAMY, 2012). We discuss them in the following paragraphs.

OpenMPE (ALESSI et al., 2015) is an extension designed for energy management of OpenMP applications, in which the programmers insert new directives in OpenMP code to indicate potential regions to save energy. For that, it requires a particular compiler and run-time system (from the Insieme project (ALESSI et al., 2015)) to recognize the directives at compilation time and apply the energy management during the application execution.

Nornir (SENSI; TORQUATI; DANELUTTO, 2016) is a runtime system that monitors the application execution and adjusts the resources configurations (DVFS, number of threads, and thread placement) in order to satisfy either performance or power consumption requirements. To use Nornir, the user has to install a system to manage the features provided by the OS (e.g. DVFS management and energy profiling), and instrument the parallel programming framework with Nornir functions.

Marathe et al. (2015) propose *Conductor*, a run-time system that dynamically selects the ideal number of threads and DVFS state to improve performance under a power constraint for hybrid applications (MPI + OpenMP). First the application is monitored in order to gauge its representative behavior; and then, a local search algorithm is applied to find and select the configuration to reduce power with minimal impact on execution time. For that, Conductor needs code modifications to insert functions.

LIMO (CHADHA; MAHLKE; NARAYANASAMY, 2012) is a dynamic system that monitors the application at run-time, being able to adapt the number of threads and DVFS accordingly. LIMO monitors the threads' progress aiming to disable cores when the thread is not making forward progress (e.g., in a synchronization function, blocking I/O call or is suspended) and, therefore, diminishing the power consumption. After disabling some cores, there is a space to increase the frequency on the active ones without exceeding a power budget. Although the work can make an online adaptation, this solution requires hardware modifications and special compiler support to determine the working set size of a thread, as well as additional OS support.

Table 3.1: Comparison of Odin with the related work

Proposal	Knobs		Adaptability			Transparency			APIs
	TLP	DVFS	Offline	Hybrid	Online	No special compilers/ tools	No programmer Influence	Binary Compatibility	
(PUSUKURI; GUPTA; BHUYAN, 2011)	x		x				x	x	OpenMP, PThreads
(CURTIS-MAURY et al., 2008)	x			x					OpenMP
(JUNG et al., 2005)	x			x			x		OpenMP-FORTRAN
(CURTIS-MAURY et al., 2006)	x			x			x		OpenMP-FORTRAN
(RAMAN et al., 2012)	x			x			x		Sequential Code
(LEE et al., 2010)	x				x				PThreads, MPI
(SRIDHARAN; GUPTA; SOHI, 2013)	x				x	x			TBB, Prometheus
(SRIDHARAN; GUPTA; SOHI, 2014)	x				x	x			PThreads, TBB
(SHAFIK et al., 2015a)	x				x	x			OpenMP
(SHAFIK et al., 2015b)	x				x	x			OpenMP
(LORENZON; SOUZA; BECK, 2017)	x				x	x			OpenMP
(SULEMAN; QURESHI; PATT, 2008)	x				x		x		OpenMP
(PORTERFIELD et al., 2013)	x				x		x		OpenMP
(LORENZON et al., 2018)	x				x	x	x	x	OpenMP
(ROSSI et al., 2015)		x	x				x	x	OpenMP, PThreads
(HOTTA et al., 2006)		x		x					OpenMP-FORTRAN
(COCHRAN et al., 2011)		x		x		x	x	x	OpenMP, PThreads
(AKRAM; SARTOR; EECKHOUT, 2016)		x			x				Java Applications
(WU et al., 2006)		x			x		x		Sequential code
(HSU; FENG, 2005)		x			x		x	x	Sequential, MPI
(RIZVANDI et al., 2010)		x			x	x	x	x	Any
(GE et al., 2007)		x			x	x	x	x	OpenMP-FORTRAN
(MIFTAKHUTDINOV, 2014)		x			x	x	x	x	OpenMP, PThreads
(CHEN et al., 2016)		x			x	x	x	x	OpenMP, PThreads
(SENSI, 2016)	x	x	x				x	x	OpenMP, PThreads
(LI; MARTINEZ, 2006)	x	x		x		x			OpenMP
(LI et al., 2010)	x	x		x		x			MPI+OpenMP
(ALESSI et al., 2015)	x	x			x				OpenMP
(SENSI; TORQUATI; DANELUTTO, 2016)	x	x			x				OpenMP, PThreads
(MARATHE et al., 2015)	x	x			x	x			MPI+OpenMP
(CHADHA; MAHLKE; NARAYANASAMY, 2012)	x	x			x		x		OpenMP, PThreads
Odin	x	x			**x**	**x**	**x**	**x**	**OpenMP**

Source: The Author

3.4 Our Contributions

Table 3.1 compares Odin to previous works. The column *knobs* indicates whether the approach optimizes TLP, DVFS, or both. The column *adaptability* indicates when the optimization is performed. *Offline* approaches only predict the behavior of a given application and do not perform any sort of run-time adaptation. *Online* mechanisms adapt the application behavior at run-time without any off-line phase. *Hybrid* approaches adapt at run-time but rely on some sort of off-line analysis. The column *no special compiler/tools*

indicates the approaches that do not need any specific compiler or tool (i.e., use different tools from the traditional programming frameworks. The column *no programmer influence* contains the approaches that do not demand any changes in the source code by the software developer. Techniques with *Binary compatibility* can be used without any need for code recompilation at all: the existent binary code as is can be optimized. The column *API* shows the parallel libraries supported by each referred work.

As we will show in this work, only approaches that consider both Threads and Operating Frequency are capable of delivering the best results in EDP. As depicted in Table 3.1, none of them covers all the needed characteristics so it could be considered completely transparent and adaptive. On the other hand, Odin performs DCT and tunes the DVFS as the application executes with minimal overhead, without the need for any sort of off-line analysis. Besides being capable of adapting to the system and application at hand, it works with any C/C++ compiler and OpenMP. It means that the software developer does not need to make any changes in the source code or even recompile it. To enable Odin, the user only has set one environment variable in the Linux OS. Because of this high level of transparency, Odin is limited to OpenMP applications.

4 PROBLEM DEFINITION AND SEARCH STRATEGIES

In this chapter, we present a formal definition of the problem of optimizing parallel applications, which we use as motivation to develop the searches strategies. Then, we describe each search strategy that we use for results comparison.

4.1 Problem Definition

A parallel application is composed of a set $\mathcal{P} = \{p_1, \ldots, p_k\}$ of k parallel regions and a set $\mathcal{S} = \{s_1, \ldots, s_m\}$ of m sequential regions. Therefore, it has $k + m$ regions in total. As the sequential regions run with one thread, they will not be considered while searching for the best solution. Each parallel region $i \in \mathcal{P}$ executes with a configuration $c_i = (t, f)$, which is a pair composed of a number of threads $t \in \mathcal{T} = \{1, \ldots, n\}$, where n is the maximum number of hardware threads, and a CPU operating frequency level $f \in \mathcal{F} = \{f_1, \ldots, f_q\}$, where f_1 is the lowest and f_q the highest possible frequency level. Therefore, a complete configuration of all k parallel regions is given by $\mathcal{C} = (c_1, \ldots, c_k)$. We denote by $\mathcal{O}$ the set of all possible $(nq)^k$ configurations. Let further be $edp(\mathcal{C})$ the EDP of the application when run with configuration $\mathcal{C}$. Then the objective is to find the configuration

$$\mathcal{C}^* = \arg\min_{\mathcal{C} \in \mathcal{O}} edp(\mathcal{C}) \tag{4.1}$$

that minimizes the EDP of the entire application execution.

In order to see how large the design space is, consider an application with four parallel regions running on a system that supports 24 threads and has 13 different CPU frequency levels. An exhaustive search would need to test $(24 \times 13)^4$ configurations, which is clearly impractical. With that in mind, we have developed and implemented different search strategies. Each search can be done either offline, prior to the program execution, or online (during its execution). Offline strategies can test many configurations without incurring any overhead when executing the application. Because of that, their training time can take several hours. However, they need to be re-executed whenever any system parameter changes (e.g., the input size or the microarchitecture). On the other hand, online strategies have the advantage that they can quickly adapt to these changes. Nonetheless, such strategies will add extra overhead to the program execution, so the tradeoff between the learning time and quality of solution increases in importance and

is key to reach a satisfactory solution. Odin features an online strategy. To assess its effectiveness, we compare it to other online and offline strategies. We detail each one in the next subsections.

4.2 Search Strategies

4.2.1 Offline Strategies

4.2.1.1 Optimal configuration for each individual region (OPT_CEIR)

It considers both the number of threads and operating frequency. However, due to the large design space, an exhaustive search considering all parallel regions is impractical. Therefore, this strategy performs an exhaustive local search for the best configuration c that will optimize the EDP of each parallel region individually.

Therefore, given a system with n threads supported by hardware and q frequency levels, each parallel region has nq possible configurations $c = (t, f)$. Let Δ be the set of all possible c. Given that $edp_i(c)$ defines the EDP of the individual parallel region i when it runs with the configuration c, this search strategy finds, for each $i \in P$, the configuration

$$c_i^* = \arg\min_{c \in \Delta} edp_i(c) \tag{4.2}$$

that minimizes its EDP.

We can define the global configuration $\mathcal{B} = (c_1^*, \ldots, c_k^*)$, which is composed of the best configuration for each individual of all k parallel regions. Then, the EDP $edp(\mathcal{B})$ is given by the execution of the application with configuration $\mathcal{B}$. Note that the set $\mathcal{B}$ not necessarily will be the best global solution. The transition from one parallel region to another, which is not taken into account by this strategy, can generate some overhead that creates a dependency between local configurations when switching from one operating frequency level to another, or when creating or destroying threads.

4.2.1.2 Genetic Algorithm – Globally Near Optimal Configurations (Static_GA)

A genetic algorithm is a meta-heuristic based on natural selection. A GA evolves a population of individuals which, by recombination and mutation, allows fitter individuals

to create offspring with a higher probability, and removes less fit individuals from the population. In our application, an individual represents a global candidate solution $\mathcal{C} \in \mathcal{O}$, and a solution is fitter when its EDP is lower. In this way the GA performs a global search that seeks to minimize the EDP of the entire application. After several generations, the GA defines the best chromosome

$$\mathcal{C}^* = \underset{\mathcal{C} \in O' \subset \mathcal{O}}{\arg\min}\, edp(\mathcal{C}) \tag{4.3}$$

of the current generation O', that is, the one that minimizes the application's EDP.

This strategy is one of the contributions of this dissertation, so we dedicate a chapter (see Chapter 5) to present a detailed discussion.

4.2.2 Online Strategies

4.2.2.1 Only DCT (Aurora)

Proposed by Lorenzon et al. (2018), Aurora optimizes each parallel region by adapting only the number of threads while using the *ondemand* governor of the OS for DVFS. Let $\mathcal{CT}^* = (t_1^*, \ldots, t_k^*)$ be the configuration of the number of threads that minimizes the EDP of each parallel region $i \in \mathcal{P}$. To find a $t_i^* \in \mathcal{CT}^*$ for each $i \in \mathcal{P}$, this strategy only searches in a subset of $\mathcal{T}$, defined here as X_i. The set $X_i \subset \mathcal{T}$ is dynamically defined by the search algorithm, as follows: for each parallel region i, the search exponentially increases the number of threads (i.e., $2, 4, 8, 16, \ldots$) while there are potential improvements, defining an interval of threads that reduces the size of the space exploration. Then, the algorithm performs a hill-climbing based algorithm with lateral movements in this interval, which is done by testing a neighboring configuration (number of threads) at another point in the search space that has not yet been tested. When the search ends, there is a set $X_i \subset \mathcal{T}$ of every x used to run i, and the best number of threads t_i^* is the $x \in X_i$ that minimizes $edp_i(x)$.

4.2.2.2 Odin – DCT + DVFS using Fibonacci Search (Odin)

The main work of this dissertation. This online strategy is inspired by the Fibonacci search of (KIEFER, 1953) for continuous domains, which repeatedly bisects the search interval unevenly to minimize the number of tests. It optimizes DCT at the max-

imum CPU frequency level. After, using the best number of threads, it searches for the optimum DVFS setting.

Formally, for a parallel region i, Odin first finds the number of threads t_i^* such that $t_i^* \in X_i$ and for all $x \in X_i : edp_i(t_i^*, q) \leq edp_i(x, q)$, where q is the highest frequency level. After that, it finds the frequency level f_i^* such that $f_i^* \in Y_i$ and for all $y \in Y_i : edp(t_i^*, f_i^*) \leq edp(t_i^*, y)$, where $Y_i \subset \mathcal{F}$ is the set of frequency levels visited by the search.

We dedicate Chapter 6 to show a detailed discussion about Odin and its search algorithm. Also, we implement two variants of Odin, which use the same Fibonacci search. Next, we describe the details of the derived strategies:

- Odin variant 1 – only DVFS (Fib_DVFS): this strategy applies the Fibonacci search only to the CPU frequency level, without DCT. Therefore, for a parallel region i, Fib_DVFS finds the best frequency level f_i^* such that $f_i^* \in Y_i$ and for all $y \in Y_i : edp_i(n, f_i^*) \leq edp_i(n, y)$, where $Y_i \subset \mathcal{F}$ is the set of frequency levels visited by the search and n is the maximum number of threads.
- Odin variant 2 – DCT + DVFS with a reduced DCT search space (Odin_pruned): in this approach, we implement a version of Odin with a reduced DCT search space. This variant eliminates from the search all the configurations using SMT, except the maximum level. We discuss our motivations for the exclusion of these settings later in Chapter 6. Formally, Odin_pruned searches t_i^* in a subset $X_i \subset \mathcal{T}^R = \{1, \ldots, n/2, n\}$.

Before we reach Odin, we developed and evaluated other two search strategies based on hill-climbing that employ DCT and DVFS, which we call *#Threads + DVFS (T+F)* and *DVFS + #Threads (F+T)*. Appendix A describes and shows the results for each of them.

5 GENETIC ALGORITHM

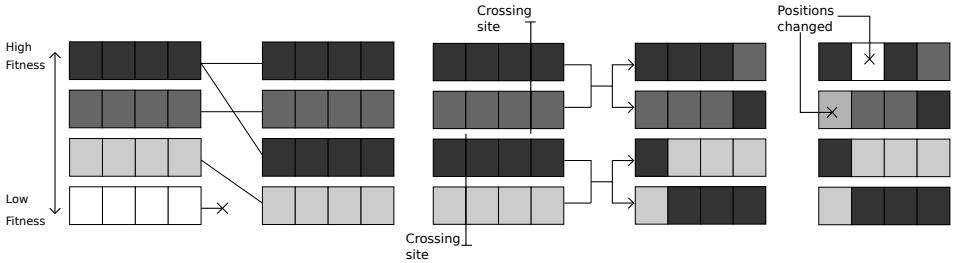
To assess the potentials gains achieved by tuning DVFS and DCT of the parallel regions of an application, we implemented a framework that uses a genetic algorithm to optimize the target statically. In contrast with the primary tool of this dissertation, presented in Section 6, which improves the EDP of each parallel region individually (at run-time), the GA works by searching a near optimal set of configurations for the whole application (statically). In any case, the aim is to optimize the EDP of the entire execution.

By executing eight well-known benchmarks on a real multicore system, we show that our framework achieves significant gains in EDP in four of them. On average, represented by the geometric mean, our approach reduces the EDP by 20.4%. In the best case, our framework improves the EDP by up to 58.3%. The remainder of this Chapter is organized as follows. In Section 5.1, we present the theory about the GA, discussing each operator and his role on the algorithm. Section 5.2 shows how we adapt it for our specific problem. Section 5.3 explains our implementation and the steps necessary to execute the framework. The methodology followed to evaluate our approach is described in Section 5.4, while the results are discussed in Section 5.5. Finally, Section 5.6 draws the final considerations for this strategy.

5.1 Background

A genetic algorithm is a search metaheuristic based on natural selection and genetics. It uses a concept of population – a set of individual solutions (chromosomes) – that can evolve to a near optimal solution through generations (GOLDBERG, 1989). There are three basic operators used to process the population: selection, crossover, and

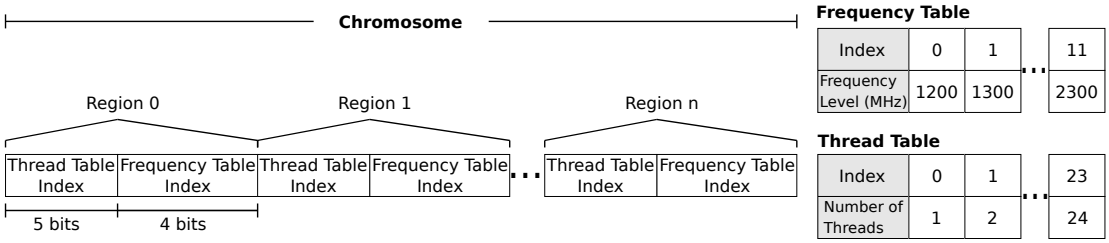
Figure 5.1: The operators of the Genetic Algorithm.



(a) Selection (b) Crossover (c) Mutation

Source: The Author

Figure 5.2: Representation of the chromosome used in the algorithm.



Source: The Author

mutation. They are illustrated in Figure 5.1 and discussed below.

The selection operator assigns probabilities to each chromosome based on a fitness function (also known as evaluation function), with the criterion that elements with better fitness will more likely to move forward (Figure 5.1a). Considering these probabilities for each chromosome, the GA will randomly select some of them for the next stage. Therefore, it is expected that a high proportion of better chromosomes will move forward, while a great portion of the worst chromosomes will be eliminated.

In the next stage, the crossover operator uses these selected chromosomes for information exchange. It randomly selects pairs and crosses sites to build new individuals (Figure 5.1b). The main idea of this step is that the chromosomes chosen during the previous step will exchange characteristics that can be recombined to generate even better chromosomes. As a secondary effect, one can also see this process as a means to increase the search space covered by the GA, because it creates new possible solutions for evaluation.

Finally, the Figure 5.1c illustrates the mutation operator. It operates at each available chromosome, performing a bit-by-bit flipping according to a defined probability (which is generally very small). Mutation usually prevents the search from being stuck in a region of a local optimum.

5.2 Our approach

In our approach, we modeled the chromosome to represent the global solution. For instance, let us consider that the program is composed of six different parallel regions. In this case, the chromosome would have twelve pieces of information (two knobs for each parallel region: CPU frequency level and the number of threads). Each knob is represented by index values (Thread Table Index and Frequency Table Index, as can be observed in Figure 5.2). These index values are used to access the Thread Table and

Frequency table, respectively, which hold the actual values for the number of threads and operating frequency. In the example in Figure 5.2, 32 different values are possible for the number of threads, and 16 values for the DVFS. The size of each field in the chromosome will change according to the number of possible variations in the number of threads and operating frequency given by the system at hand. The Thread Table is necessary because not always the number of tested threads will be continuous (i.e.: one could try testing the number of threads considering a list that follows an exponential order, so the indexing would be necessary).

We used the smallest possible amount of bits for the binary representation of the indexes (they have the minimal possible size to fit each table), which enables a more efficient crossing site during the crossover inside the field[1], ensuring that the GA will have more coding similarities to exploit (GOLDBERG, 1989). Therefore, the selection operators work on the aforementioned indexes, and the algorithm uses the tables to mount the configuration to execute the application and get its results.

In this work, the optimization objective is defined to minimize the EDP for each application. Thus we use the following equation for the fitness function (which the GA tries to maximize):

$$Fitness(chromosome) = Pop_{max} - EDP(chromosome) \tag{5.1}$$

In Equation (5.1), Pop_{max} is the maximum EDP value found in a population from a current generation. Consequently, the chromosomes with lower EDPs will produce the higher fitness, which will have higher probabilities to pass through generations.

To configure the parameters of our GA, we started using values with magnitude orders that have better results in similar problems from the literature (e.g.: a high chance for crossover and a low probability for mutation) (GOLDBERG, 1989) and refined them experimentally. For our experiments, we started with a random population with a fixed size of 30 to 40 individuals (depending on the application). The probability for the crossover and the mutation to happen is of 0.9 and 0.001, respectively.

[1] Assuring that the new value will not exceed the table range.

5.3 Implementation

To optimize an OpenMP application using our framework, the user has to replace the original OpenMP *libgomp* with our particular implementation, which adds support for changing the number of threads and CPU frequency dynamically. With that, it is not necessary to instrument or recompile the application - so it is totally transparent for the user. After this, the user should run our GA, which is currently implemented in Python, using the OpenMP application as input parameter, and it will search for the optimum settings as described earlier. For each configuration, the script runs the target application three times and uses the results average to filter any variation.

As said in Chapter 2, although our system has the possibility of changing the frequency level for each physical core individually, in this work we are considering the frequency switching for the entire package. The user can insert or remove our governor dynamically on Linux, with no need to recompile the kernel. Therefore, the transparency for the application to be optimized is maintained.

5.4 Methodology

5.4.1 Benchmarks

Eight applications written in C language and already parallelized with OpenMP from different well known benchmark suites were chosen:

Three applications from different domains:

- *Breadth-First Search* (BFS): is a traversing or searching algorithm for tree or graph data structures. It starts from a selected node (i.e., source or starting node) and traverses the graph exploring the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level (CORMEN et al., 2009). We use the implementation from the GAP benchmark suite (BEAMER; ASANOVIĆ; PATTERSON, 2015) and consider the following input parameters: 2^{20} vertexes and 2000 iterations.
- *Fast-Fourier Transform* (FFT): an algorithm that computes the discrete Fourier transform of a sequence of elements. It samples a signal over a period of time and divides it into its frequency components. This algorithm is widely used in many areas, such as engineering, science, and mathematics (PETERSEN; ARBENZ, 2004).

- *STREAM*: a synthetic benchmark used to measure the sustainable memory bandwidth through mathematical operations performed over long vectors (MCCALPIN, 1995).

Five kernels from the NAS Parallel Benchmark. It is a suite originally developed by NASA that comprises applications derived from computational fluid dynamic (BAILEY et al., 1991). As the original version of NAS is written in FORTRAN, we consider the OpenMP-C version developed by Seo et al. 2011 (SEO; JO; LEE, 2011). The kernels used from the NAS Parallel Benchmark were:

- *Discrete 3D fast Fourier transform* (FT) computes a 3-D partial differential equation by using FFTs. We consider the class B as the input set.
- *Multi-grid on a sequence of meshes* (MG) performs a defined number of conjugate gradient iterations in order to approximate the solution z to a certain specified $n \times n$ linear system of equations $Az = x$. We executed this kernel with the class C as the input set.
- *Lower-upper gauss-seidel solver* (LU), *Scalar penta-diagonal solver* (SP), and *Block tri-diagonal solver* (BT): each one implements a different method to compute a synthetic nonlinear system of partial differential equations. LU employs a symmetric successive over-relaxation numerical scheme to solve a regular-space, block(5×5) lower and upper triangular system; and SP and BT are used to solve multiple independent systems of non-diagonally dominant through scalar pentadiagonal and block tridiagonal equations, respectively. The three kernels were executed with the input parameters from the class B.

5.4.2 Execution Environment

The experiments were performed on a real multicore system able to concurrently execute up to 24 threads and 12 distinct CPU frequency levels (Table 5.1). The system uses the Ubuntu Operating System with kernel v. 4.15.0. All the benchmarks were compiled using GCC v. 5.4.0 with the -O3 optimization flag. The execution time and energy consumption of each benchmark were obtained through the *omp_get_wtime()* function (from OpenMP) and directly from the hardware counters through the Intel Running Average Power Limit (RAPL) (HäHNEL et al., 2012), respectively. We consider the total energy consumption as the sum of the energy spent by the DRAM modules and core

Table 5.1: Main characteristics of the system

Processor	Intel Xeon E5-2630
# Cores	2x6
# Threads	24
CPU Freq. range	1.2 - 2.3 GHz (12 levels)
L1 Cache	12x32 KB
L2 Cache	12x256 KB
L3 Cache	30 MB
RAM	32 GB

Source: The Author

domains (CPU and cache memories).

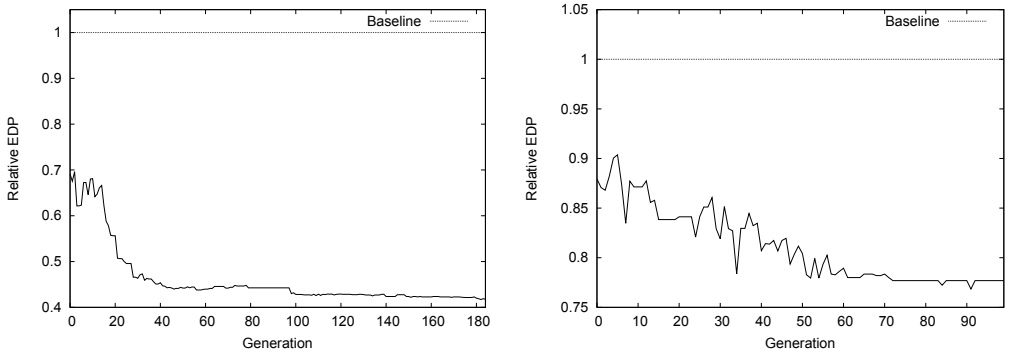
Our approach is compared with the regular way of executing parallel applications in multicore systems (LEE et al., 2010): the application executes with the maximum number of threads available in the system and the CPU frequency is set to adjust according to the workload application, using *ondemand* as DVFS governor. We used this configuration as the **baseline** when comparing to our genetic algorithm.

5.5 Experimental Results

Figures 5.3 and 5.4 show the EDP results for each benchmark. The *x axis* represents a given generation of our GA approach, while the *y axis* gives the relative EDP of the best element (in terms of EDP) of that generation. The EDP in the *y axis* is normalized to the EDP achieved by the **baseline** configuration (represented by the line fixed at 1 in the y axis), so lower is better. Therefore, when the relative EDP is lower than 1, it means that it is better than the baseline.

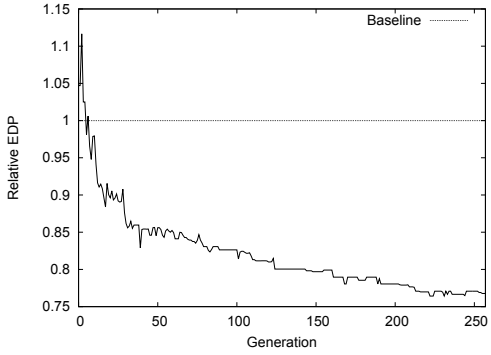
Some applications already start with a configuration found by the GA that has better EDP than the baseline, and the difference between them increases over the generations. That is the case of SP and STREAM benchmarks, as shown in Figures 5.3a and 5.3b, respectively. By finding a better solution at the end of all the generations, the GA reduced the EDP of SP and STREAM benchmarks by 58.3% and 23.2%, respectively, comparing to the baseline. On the other hand, for MG and FT benchmarks, the GA starts with a solution that is worse than the baseline. However, as the generations evolve, our solution gets better than the baseline (for the MG) or very close to it (FT). In the MG benchmark (Figure 5.3c), the genetic algorithm takes six generations to reach the same

Figure 5.3: The GA results from each benchmark normalized to the baseline EDP.

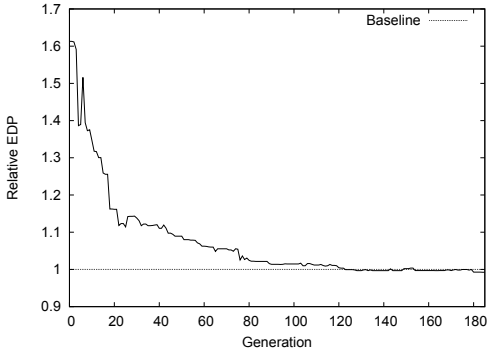


(a) SP

(b) STREAM



(c) MG



(d) FT

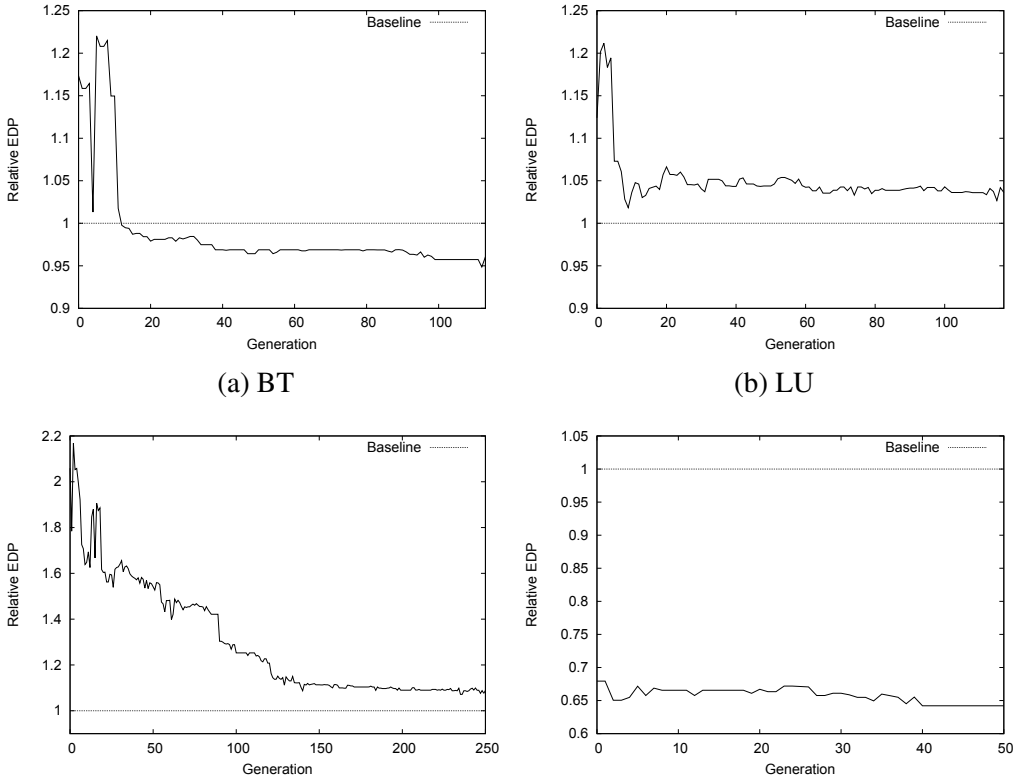
Source: The Author

EDP as the baseline, and ends with a reduction of 23.6% in EDP.

One can note that the optimization curves of the genetic algorithm are not strictly descending, that is, during the evolution process of the chromosome, the GA can reach valleys and peaks before achieving a near optimal result. This can be seen for the STREAM benchmark, as shown in Figure 5.3b. This situation happens due to the broader space of exploration ensured by the GA, i.e., even some solutions with no good results are allowed to survive because they can carry good features that may be useful in the crossover operation. The mutation also helps the increasing in the exploration by doing some random changes in the chromosomes. Therefore, the strategies employed by the GA can diminish chances of being stuck in local minima.

In some cases, our algorithm cannot find a better configuration than the original baseline: the resultant configurations for the LU and BFS are worse then the baseline by 1.8% and 7.1%, respectively (see Figure 5.4b and 5.4c). In such cases, as the framework is executed before application deployment, the user should choose to keep the original baseline for future executions. The baseline versions that are better than the ones found by the GA usually belong to a specific group of parallel applications that are well balanced

Figure 5.4: The GA results from each benchmark normalized to the baseline EDP.



(a) BT (b) LU

(c) BFS (d) FFT

Source: The Author

and CPU-bound, so executing them with the maximum possible number of threads with an operating frequency set to the maximum will usually deliver the best EDP.

Moreover, some configurations found by the GA show what knob has more potential for scaling (i.e. in some cases changing the number of threads will influence more the results than DVFS and vice versa). Table 5.2 shows the best pair of the number of threads and the CPU frequency level discovered by the genetic algorithm for each parallel region for all the benchmarks. The STREAM, for instance, has a better result with a high number of threads in almost all regions, but operating at a low/medium level of frequency. This benchmark makes a significant number of memory accesses, so the time to wait for memory requests allows good savings in energy consumption with a little performance impact while reducing the processor frequency. Another case, such as the best configuration for the benchmark SP (see Table II), reveals that only a few regions have an ideal number of threads that approaches the maximum, and that is why the optimized version of this application is much better than the baseline (that, by default, always operates with the maximum number of threads).

Finally, Figure 5.5 summarizes the results for each benchmark, showing the best

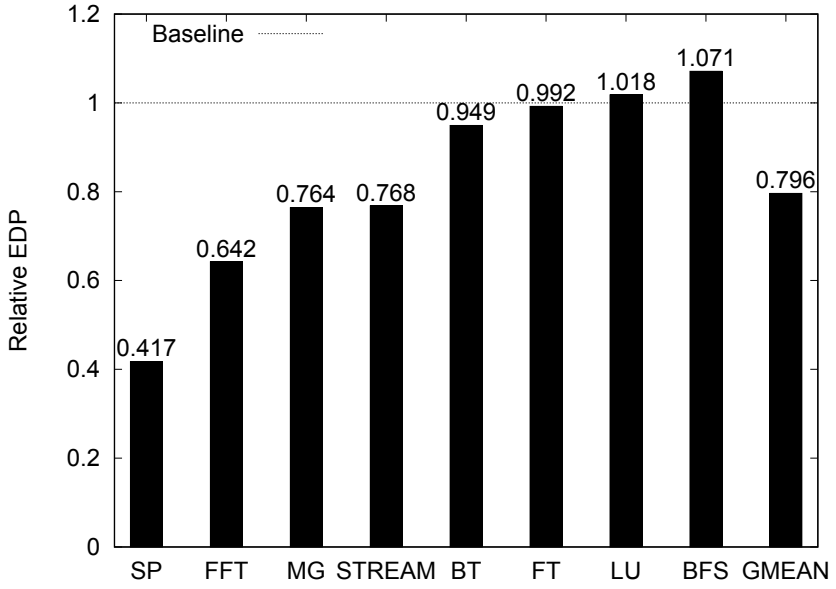
Table 5.2: Best pair of the number of threads and CPU frequency level (MHz) found by our GA for each parallel region.

Benchmark	Configuration
STREAM	(24 - 1800), (9 - 2300), (22 - 1300), (23 - 1700), (20 - 1400), (23 - 1300)
SP	(18 - 1300), (23 - 2100), (10 - 1500), (8 - 1300), (12 - 2300), (9 - 2000), (12 - 2300), (11 - 1900), (12 - 2300), (10 - 1200), (10 - 2200), (18 - 2300), (24 - 1800), (20 - 2300)
MG	(10 - 1600), (24 - 2300), (12 - 2300), (19 - 1300), (24 - 2200), (19 - 2100), (24 - 1400), (24 - 1700), (24 - 1600), (24 - 1200), (1 - 1200)
FT	(24 - 2100), (24 - 2300), (23 - 2300), (24 - 2300), (24 - 2300), (24 - 1900), (24 - 1400), (2 - 1200), (8 - 2200)
BT	(16 - 2000), (15 - 2200), (8 - 2000), (23 - 2300), (24 - 2300), (22 - 2300), (7 - 1700), (17 - 1800), (15 - 2000), (12 - 2300)
LU	(4 -1800), (13 - 2000), (22 - 2300), (14 - 2000), (12 - 2300), (24 - 2000), (22 - 2300), (20 - 2300), (19 - 1600), (5 - 1900)
BFS	(16 - 2300), (9 - 2000), (23 - 1400), (15 - 2200), (15 - 1200), (24 - 1700), (15 - 1200), (20 - 1800), (5 - 1600), (24 - 2000), (22 - 2000), (24 - 1400), (23 - 1600), (22 - 2300), (22 - 2200), (24 - 2300), (24 - 2300), (24 - 2300)
FFT	(23 - 1700), (5 - 2300), (11 - 1700)

Source: The Author

results found by our algorithm. Compared to the baseline and considering the geometric mean of the entire benchmark set, our approach reduced the EDP by 20.4%. In the best case, our framework improved the EDP by up to 58.3% (SP benchmark - Figure 5.3a). Four out of the eight tested benchmarks presented significant improvements. Three of them were very similar to the original baseline, while the BFS did not converge as expected by using the GA. As already mentioned, as this space exploration is done statically at design time, the user may choose to keep the baseline as the main version for future executions.

Figure 5.5: Best results found by our algorithm for each benchmark and the geometric mean (GMEAN) normalized to the baseline EDP.



Source: The Author

5.6 Discussion

We have shown that our static framework can optimize most of the tested benchmarks, improving the EDP of OpenMP applications by up to 58.3% when compared to the regular way that parallel applications are executed. Besides that, we have seen that while some benchmarks obtain the best results mainly by tuning the number of threads, others have the frequency scaling as the most critical knob to get better results, thus showing that the two combined techniques have more potential of optimization compared with using one of them alone.

6 ODIN: ONLINE, NON-INTRUSIVE AND SELF-TUNING DCT AND DVFS TO OPTIMIZE OPENMP APPLICATIONS

In Chapter 5, we showed how one can use a genetic algorithm to tune the number of threads and DVFS to optimize the EDP of OpenMP applications. However, although one can achieve positive results, it has some limitations, such as:

- **Long training time:** the framework works by collecting the total EDP result of the application, i.e., each test requires the complete execution of the target. As the algorithm needs many samples to converge to a near optimal configuration, the training time can take several hours. For example, an application with an execution time of 30 seconds can take from two to three days to complete the training.
- **Lack of adaptation:** the parallel applications may have different optimal configurations depending on many parameters, such as the application input set, the processor architecture, memory system etc. Thus, a new training would be necessary at every time that one of these parameters change.

In order to address these limitations, we developed Odin. Odin is an online approach capable of automatically tuning the DCT and DVFS of an OpenMP application and maintaining the software transparency. Odin optimizes at the execution time, using different algorithms to decrease the overhead of the training process and allowing the adaptation according to the online feedback from the application.

6.1 Odin Integration to OpenMP

As described in Chapter 2, OpenMP provides three ways for exploiting parallelism: parallel loops, sections, and tasks. Parallel loops is by far the most used approach, and all popular OpenMP benchmarks are implemented in this way. Therefore, for now, Odin works to optimize parallel loops and does not influence in any way the code that uses sections or tasks.

Odin was incorporated into the *libgomp*, a GNU Multi-Processing Run-time Library responsible for all OpenMP functionalities. It is dynamically linked to applications, so modifications in its code are completely transparent to user applications. To use Odin, the user simply has to replace the original OpenMP *libgomp* with Odin's *libgomp* (which also includes the original OpenMP functions). When the environment variable

OMP_ODIN is set in the Linux OS, the runtime system of Odin is used instead of the regular OpenMP functions. Otherwise, application executes with the original OpenMP. There is no need to make any modifications in the OS (package installation, kernel re-compilation).

When a regular OpenMP application with parallel loops (CHAPMAN; JOST; PAS, 2007) starts executing, the *initialize_env*() of *libgomp* function is called. It is responsible for initializing all the environment variables used by OpenMP during the execution. When the program reaches the directive *#pragma omp parallel* (used to indicate a parallel region), *gomp_resolve_num_threads*() is called to create and define the number of threads. At the end of the parallel region, the *gomp_parallel_end*() joins the threads and finalizes the region environment. When the execution ends, *team_destructor()* finalizes the OpenMP environment.

Given that, Odin comprises four functions that were incorporated into the aforementioned original *libgomp* functions:

- ***initOdin*()** initializes the necessary data structures, libraries, and variables used to control the search algorithm. This function was implemented inside the original *initialize_env*().
- ***odinResolveThreadsDVFS*()** sets the number of threads and DVFS to be used for the current parallel region based on the current state of the search algorithm. It also initializes the counters for collecting data from the execution environment of the current parallel region. It was implemented inside the *gomp_parallel_start*(), and replaces the original *gomp_resolve_num_threads*() function.
- ***odinEndParallelRegion*()** is implemented in the *gomp_parallel_end*() function, and executed after the execution of the current parallel region to get its EDP. Execution time is extracted by the *omp_get_wtime()* function, provided by OpenMP, while the energy is obtained directly from the hardware counters present in modern processors. In the case of Intel processors, the Running Average Power Limit (RAPL) library is used (HäHNEL et al., 2012), while the Application Power Management library could be used for AMD processors (HACKENBERG et al., 2013). Besides that, as we focus only on parallel regions for simplicity, this function changes the frequency to the maximum level for execution of a following sequential region. *odinEndParallelRegion*() also performs one step of the search algorithm if it has not converged yet, defining the pair threads/operating frequency that will be used for next iteration of the current parallel region.

- ***OdinDestructEnv*()** was implemented in the *team_destructor*() OpenMP function. It concludes and destroys the Odin environment at the end of execution.

Finally, the Algorithm 1 depicts the modifications made in the original *libgomp* source code using these mentioned functions. One can observe that we always check whether Odin is enabled (set by the environment variable) before calling any of our library procedures, therefore, preserving the original OpenMP features.

Algorithm 1 Libgomp functions modified by Odin

```
function INITIALIZE_ENV(void)
    Initialization of the OpenMP environment
    if Odin is enabled then
        odinInit()
    end if
end function

function GOMP_PARALLEL(*fn, *data, num_threads)
    region_ptr ← fn address region
    if Odin is enabled then
        num_threads ← odinResolveThreadsDVFS(region_ptr)
    else
        num_threads ← gomp_resolve_num_threads(num_threaads, 0)
    end if
    gomp_team_start(fn, data, num_threads, flags,
                    gomp_new_team(num_threads))
    Execute parallel region
end function

function GOMP_PARALLEL_END(void)
    if Odin is enabled then
        odinEndParallelRegion()
    end if
    Finalize region environment
    gomp_team_end()
end function

function TEAM_DESTRUCTOR(void)
    if Odin is enabled then
        odinDestructEnv()
    end if
    pthread_key_delete(gomp_thread_destructor)
end function
```

6.2 Optimization Strategy

Our online search strategy seeks to optimize each parallel region individually, evaluating different possible configurations at runtime. Once the online strategy converges to its best configuration, it will be used for the remaining executions of the corresponding parallel region. The main challenge is to reach a satisfactory solution without incurring in a high learning overhead. Because of its dynamic nature, it is not possible to consider other parallel regions as one parallel region is being optimized (i.e. the online optimization is always local with respect to the current parallel region). As this strategy searches for the best configuration as the application executes, the computation done during the learning phase is not wasted (i.e., it is used by the application), which helps reducing its costs.

As an example, let us consider a parallel region that is repeated by **N** times. Then, the total EDP (EDP_{tot}) of the region is given by Equation 6.1.

$$EDP_{tot} = \sum_{j=1}^{N} e_j \times \sum_{j=1}^{N} t_j \tag{6.1}$$

Where e and t is the energy and time, respectively, spent during one iteration. However, it is expected that the parallel region will present the same behavior (time and energy) during all the following iterations under the same configuration (DCT and DVFS). So we can rewrite the Equation 6.1 as:

$$EDP_{tot} = N \times e \times N \times t = N^2 \times e \times t = N^2 \times edp \tag{6.2}$$

Where *edp* is the EDP for one iteration of the parallel region. Therefore, we focus on finding, during the first iterations, the configuration of DCT and DVFS that minimizes *edp*, so we can use it for subsequent iterations to optimize the whole parallel region.

6.2.1 The search algorithm

The search algorithm was designed based on the following assumptions. They were derived from the general empirical behaviour of the EDP as a function of the number of threads and the frequency.

1. The EDP tends to be a convex function in the number of threads and frequency

when it ignores local noise, which can be caused by a slightly different behavior for an odd or even number of threads.

2. The number of threads plays a stronger influence on the EDP than the CPU frequency. This behavior happens because when tuning the number of threads, one can (*i*) decrease power – since fewer cores would be active – and (*ii*) reduce the execution time by avoiding bottlenecks at the same time. On the other hand, in the case of frequency scaling, we are always trading execution time for power reduction or vice-versa (in different proportions depending on the case).

Considering this scenario, the online strategy that is implemented in Odin was inspired by the Fibonacci search for continuous domains (KIEFER, 1953), which repeatedly bisects the search interval unevenly to minimize the number of tests. We adapted the search for a discrete domain as shown in Algorithm 2.

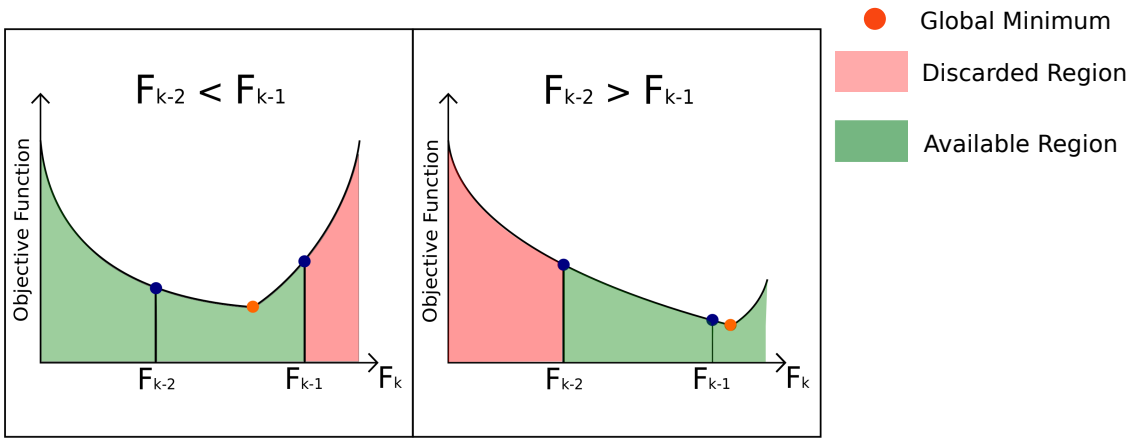
The algorithm starts by selecting the smallest Fibonacci number, F_k, which is greater or equal the size of our search space. For example, if it has 16 possible configurations, considering the Fibonacci sequence that is [1, 1, 2, 3, 5, 8, 13, 21, ...], $F_k = 21$, $F_{k-1} = 13$, $F_{k-2} = 8$ and so on – although we bind the search on a Fibonacci number that can be greater than our search space, the algorithm implementation is aware of the real boundary and will not exceed it. After that, it bisects the search interval using F_{k-1} and F_{k-2} as points for comparison. Figure 6.1 shows an example of the first algorithm movement where the orange mark represents the optimum value for the objective function. In the left side, if $f(F_{k-2})$ is smaller than $f(F_{k-1})$, the minimum point of the objective function must be in the left of F_{k-1}, so the search drops the range represented in the red region. The same logic applies to the right side, now $f(F_{k-2})$ is greater than $f(F_{k-1})$, and the algorithm should discard the red area. After that, the new comparison points, F_{k-1} and F_{k-2}, are adjusted based on the new range. Also, the space pruning is done as one configuration is always used to the next comparison and the algorithm continues till search space is finished. Therefore, for an interval of length $n = F_k$ the search needs at most k probes, such that the number of probes is bounded by $\lceil \log_\varphi(\sqrt{5}n + 1/2) \rceil = \Theta(\log_\varphi n)$, where $\varphi = (1 + \sqrt{5})/2 \approx 1.62$ is the golden ratio.

Besides that, Fibonacci search attends the Assumption 1 because for convex functions it will find the optimal value. With respect to the Assumption 2, we first apply the search on the thread space using the maximum frequency level. Then, maintaining the best number of threads found, we execute a similar search for the best frequency level.

Algorithm 2 Fibonacci search.

Require: A search interval $[l, u]$, an objective function f.
  Let F_k be the length of interval $[l, u]$.
  while $k > 3$ **do**
    if $f(u - F_{k-1}) < f(l + F_{k-1} - 1)$ **then**
      $u := l + F_{k-1} - 1$
    else
      $l := u - F_{k-1} + 1$
    end if
    $k := k - 1$
  end while
  if $f(l) < f(u)$ **then**
    return l
  else
    return u
  end if

Figure 6.1: Example of Fibonacci algorithm pruning the search space.



Source: The Author

6.3 Sample Selection

One of the most critical elements on our algorithm is the quality of the selected samples of each configuration responsible for guiding the search. Although we can statistically benefit from a large number of samples, this may negatively affect the final results, since we implement an online strategy for training.

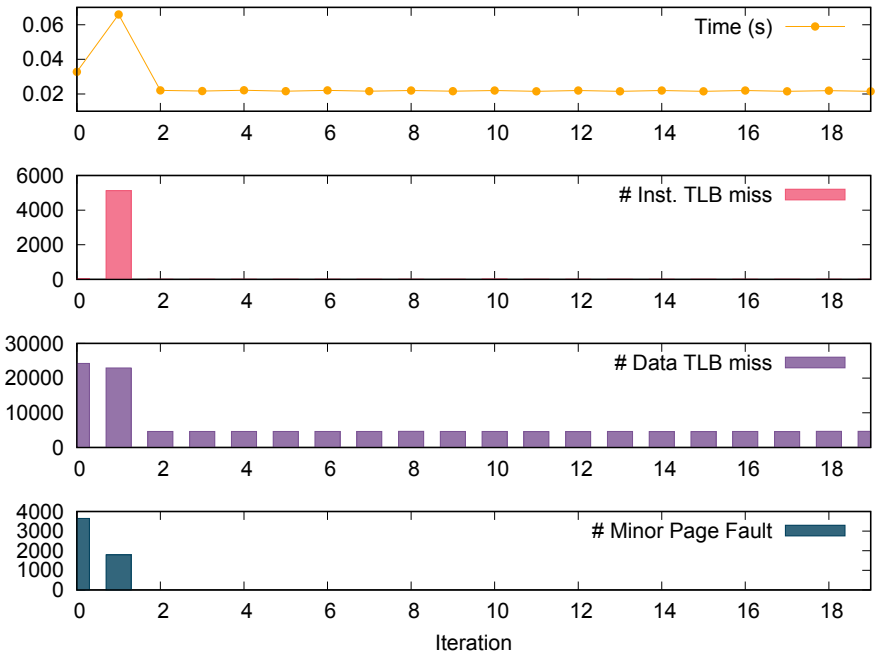
In this section, we focus on events that can add substantial noise on the sample results and how we can minimize their impact on our algorithm. Thus, we do not discuss some noises such as those caused by cache memories that are negligible and will be present on almost every sample, making the comparisons fair by this criterion.

6.3.1 Virtual Memory

All modern current operating systems implement virtual memory. Historically, there are two significant motivations for virtual memory: (*i*) to allow efficient and safe sharing of physical memory among multiple programs; and (*ii*) to remove the programming burdens of a small and limited amount of main memory (PATTERSON; HENNESSY, 2013). Thus, the processor uses a Memory Management Unit (MMU) to translate virtual to physical addresses. In order to help with the translation speed, the processor has a cache of page table entries called Translation Lookaside Buffer (TLB). It keeps the most recent accessed virtual pages for fast translation into physical pages. Therefore, when a TLB miss occurs, the processor has to search for the page in more distant memories (e.g., main memory) impairing the performance.

Considering this scenario, sometimes the Operating System can stall the running process to handle interruptions or exceptions, and its overhead can pollute the sample used by the search algorithm during the training time. Something usual in Linux is when a process requests memory through a system call, for example, when calling the *malloc* function. In this case, the kernel does not immediately allocate the physical memory, instead, it just gives an address for virtual memory. Thus, the first time when the program writes to these virtual pages raises the page faults exceptions, then at that point the kernel has to find pages from physical memory (in case of minor page faults) through an interrupt handler. Beyond the noise caused by the page faults, the kernel interruption may pollute the data and instruction TLB because it can overwrite the previous information in these caches, increasing its impact.

Figure 6.2: The impact of the virtual memory events on the execution time of the JA benchmark (32 Threads).



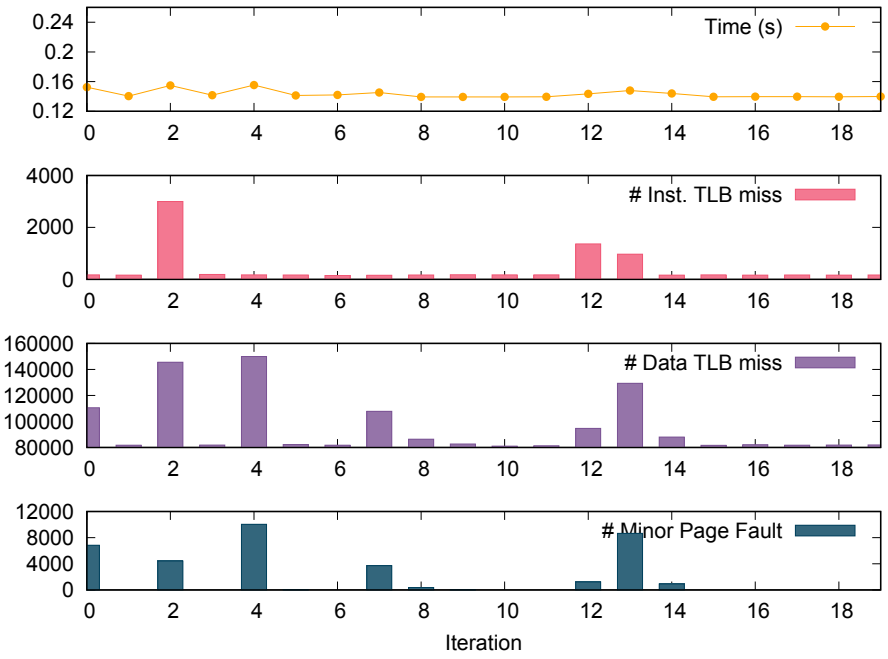
Source: The Author

Figure 6.2 shows the impact of virtual memory events on the execution time of a parallel region of the Jacobi (JA) benchmark running with 32 threads[1]. One can observe that the first iteration takes a little more time than after convergence, because it produces many minor page faults and misses in data TLB – also, in the first time, the processor may still be bringing data to cache memories. In the second iteration, the instruction TLB goes from a few tens to thousands of misses making the execution time up to 3x worse than after convergence. This benchmark presents this behavior independently of the number of threads and, therefore, during the training algorithm, the configuration tested on the second iteration will show a very inaccurate result that may guide the search to the wrong side.

There are cases where the online search by itself is responsible for increasing the events related to virtual memory when changing the DCT level during the training process. Figure 6.3 shows the execution time and virtual memory events for 16 threads when running a parallel region from the Scalar Penta-diagonal solver (SP) benchmark. Although the region presents some iterations with a thousand of instruction TLB misses, the influence on the execution time is only about 10%. Next, Figure 6.4 shows the related events when one changes from a configuration of 32 in the first iteration to 16 threads

[1]We present the TLB results for only one thread, but the quantity for the others exhibits the same order of magnitude.

Figure 6.3: The impact of the virtual memory events on the execution time of the SP benchmark (16 Threads).



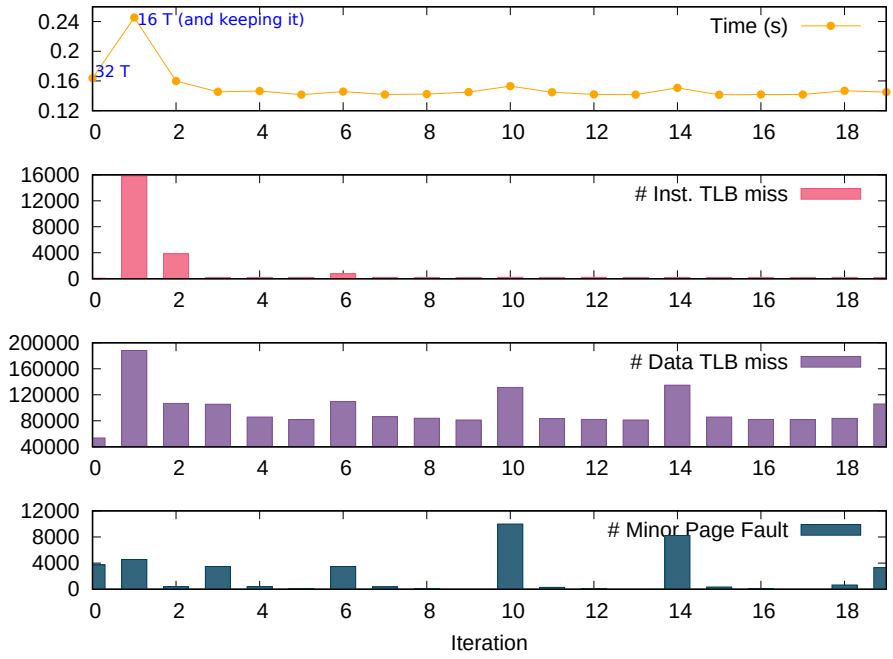
Source: The Author

in the second iteration. The decrease in the number of threads increases the number of instructions for each thread, so the amount of misses on the instruction TLB grows and raises in 50% the execution time of the region.

Therefore, we can expect some situations where our algorithm can collect a sample contaminated by noise exposed by these events. Because they are isolated events and their effect in the execution time of the region is dependent on the system (e.g.: distinct architectures will have different memory latency, TLB configurations etc), we decided to modify our algorithm during the search to discard these samples and repeat the tested configuration. Empirically, we see that the instruction TLB miss is the event that impacts the execution time of a parallel region the most. In most cases, this event tends to be insignificant for our benchmarks, making it a reliable criterion for the detection of a polluted sample. Therefore, we choose to set a threshold for the instruction TLB miss event, so we can discard a sample when this limit is reached and not consider it for the learning process.

To determine this threshold, we collected all the training points for the entire benchmark set, i.e., all the parallel regions of every benchmark when running our main algorithm. Then, we used a simple arithmetic mean to cluster the data between discarded and accepted values. Figure 6.5 shows the number of instruction TLB misses for every tested configuration when running all the benchmarks – the *x-axis* is used only to show

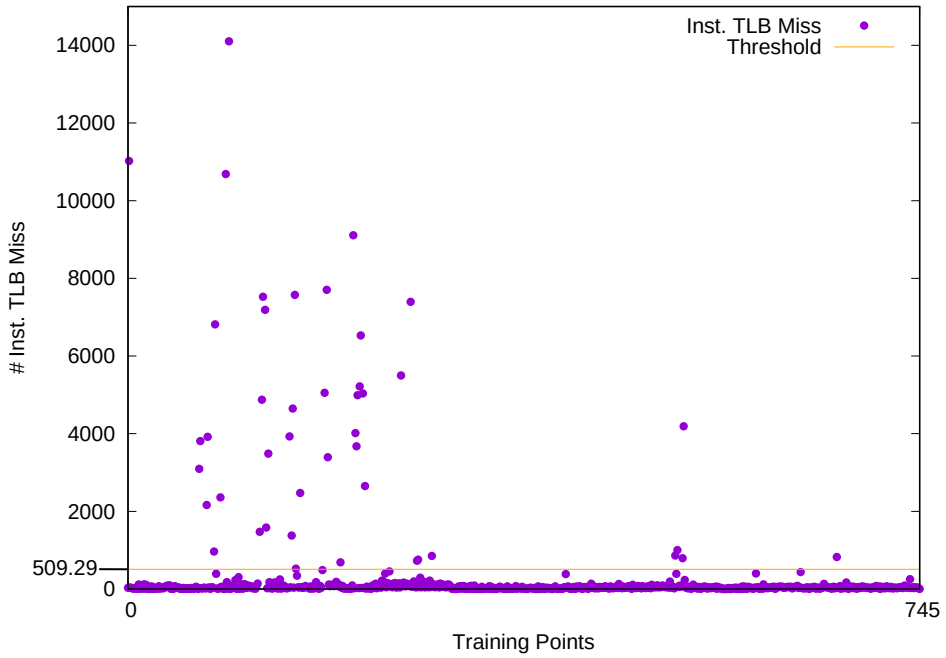
Figure 6.4: The impact of the virtual memory events on the execution time of the SP benchmark (32 to 16 threads).



Source: The Author

the number of training points, and its order is irrelevant. The arithmetic mean calculated is 509.29 and only 6.6% of points are above this threshold; also, we empirically confirm that the number of instruction TLB misses diminishes as we repeat a tested configuration.

Figure 6.5: The threshold for the number of instruction TLB misses.



Source: The Author

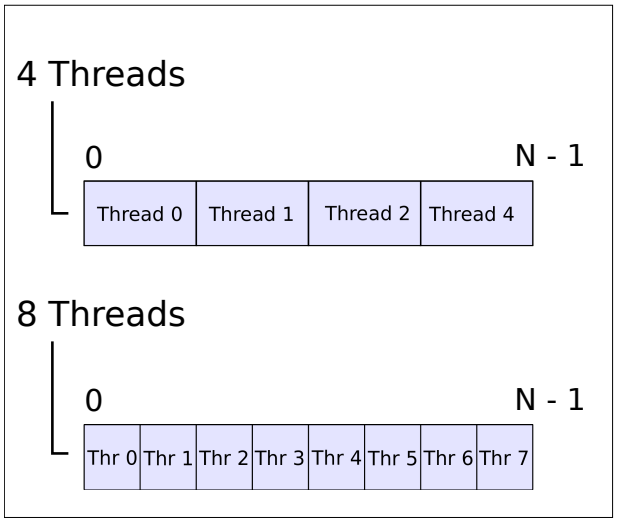
6.3.2 Memory Locality

During the DCT phase, the application data is moving across the memory system as the OpenMP framework is dividing the loop between threads. Figure 6.6 shows how the OpenMP static scheduler assigns blocks of iterations to each thread considering four and eight levels of concurrency – although the OpenMP has other schedulers, we focus only on static for the sake of simplicity.

Most of the modern systems have Non-Uniform Memory Access (NUMA). A NUMA architecture means that the latency to access a data or instruction in memory depends on its locality relative to the processor. Figure 6.7 presents an example of a NUMA system. There are two processors, each of them with two physical cores (four logical cores using SMT), with private and shared caches, and the main memory. The memory requests are served by the local or remote NUMA node and the latency of each request will change depending on the data location.

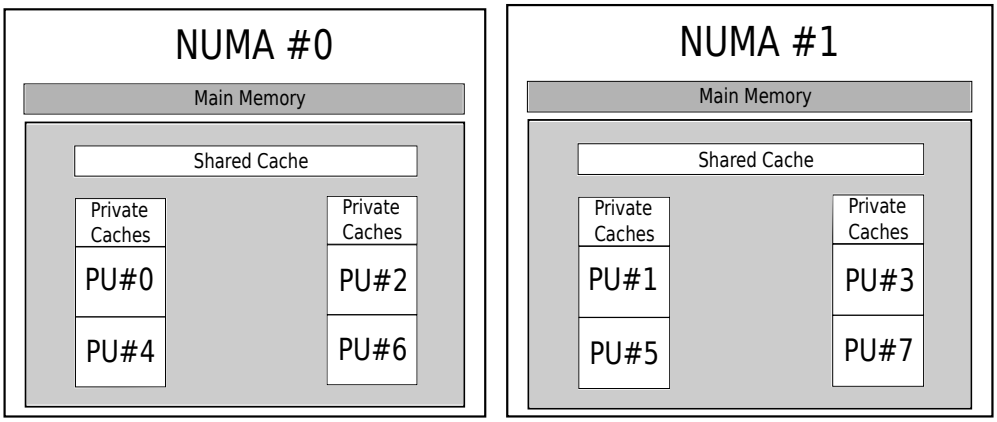
In order to assess how it can affect the algorithm samples during the training, let us consider an example of changing from eight to four threads while processing data from an array. If all the cores are free and we have not set a different affinity for OpenMP, the operating system will first fill with processes/threads the physical cores without using SMT, aiming to optimize performance. Then, it will keep dividing the workload and sending the threads to different NUMA nodes to maintain balance in memory division. Therefore, when dividing the loop for eight threads, one can see on the left side of Figure 6.8 that the first block of iterations will be assigned to NUMA #0, the second block will be

Figure 6.6: OpenMP static scheduler assigning block of iterations to each thread.



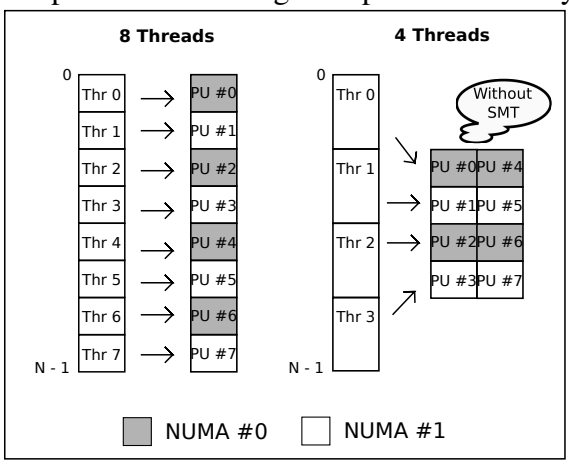
Source: The Author

Figure 6.7: Example of a NUMA system with two nodes.



Source: The Author

Figure 6.8: Example of OpenMP scheduling on top of a NUMA system with two nodes.

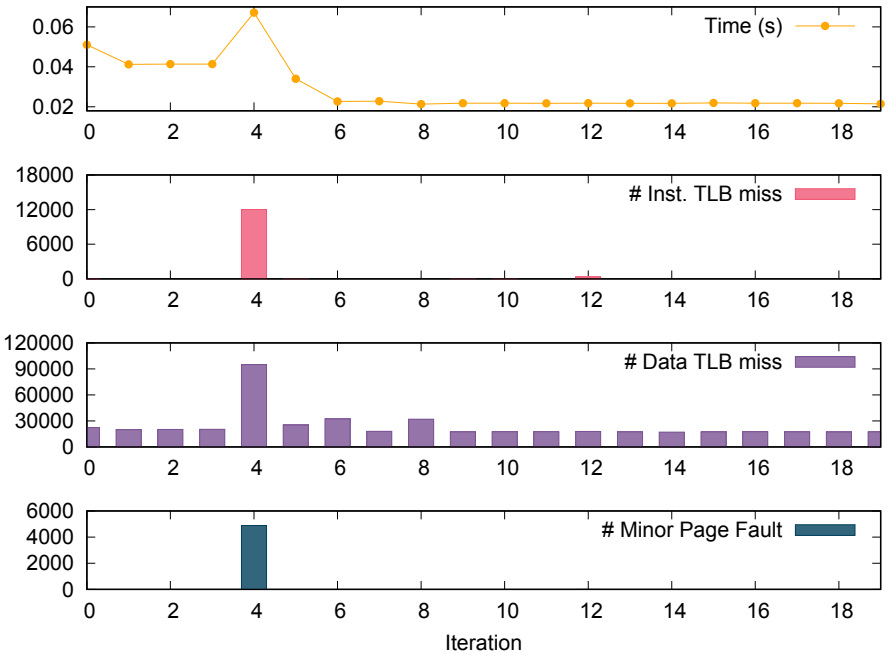


Source: The Author

allocated in NUMA #1, and so on. Also, we have to consider that each block of iterations commonly is used to process an array or matrix so that we can refer to it as a block of data. When the algorithm changes the concurrency level to four threads (right side of Figure 6.8), the data block of each thread will be two times larger than before, i.e., one data block for one thread is now equivalent to two contiguous data blocks in the previous DCT level (with eight threads). In this scenario, the data block processed before by Thread 1 that belongs to NUMA #1, now is processed by Thread 0, which is running on PU #0 under the NUMA #0. Thus, in the current DCT level (4 threads), one-half of the data processed by Thread 0 is allocated on a remote memory. This same pattern repeats to the other blocks.

Figure 6.9 shows the execution time, instruction TLB misses, data TLB misses, and minor page faults for a parallel region from the STREAM (ST) benchmark changing from 32 (omitted in the figure) to 16 threads. It is the same situation as the previous ex-

Figure 6.9: Example of memory locality noise in the ST benchmark.



Source: The Author

ample, but now in a real system with two processors, 16 physical cores including SMT, and 2 NUMA nodes. The ST is a synthetic benchmark that performs many memory requests through several operations on long arrays. After DCT, one can see that it takes seven iterations to converge to a stable value. Moreover, we see that in the fifth iteration the number of minor page faults rises, as the instruction and data TLB misses. This happens because how the Operating System handles that: it automatically balances the pages between NUMA nodes. Therefore, the virtual memory events show when the pages are invalidated and allocated in another node. Consequently, the first iterations are suffering from the long latency of the remote memory, which is a large sample window where the search algorithm could move to the wrong side.

Although we can adjust the affinity for this specific case, i.e., change the way that the OpenMP places the threads on each core, thus guaranteeing that the data are not dispersed between the NUMA nodes, maintaining this control during the whole training for each level of concurrency transition is challenging and is out of scope of this work. Also, whether two or more regions are working on the same memory space, there is a possibility of interference between them.

Therefore, to attenuate the impact of this noise in our algorithm, we deal with it indirectly by implementing the Odin_pruned (see Chapter 4), which reduces the DCT search space. The motivation for excluding the mentioned settings is that they produce

an imbalance on the running threads, i.e., some will run using the whole physical core, without SMT; while others will share it. However, the OpenMP synchronization primitive is a barrier, and hence, the slow threads always delay the fast ones. Therefore, even when in some cases these configurations consume less power for using fewer resources, it is unlikely that they will spend less time than every balanced setting. Thus, when we put the EDP in function of power, the formula is

$$EDP = P \times t^2 \tag{6.3}$$

Where *P* is the consumed power and *t* is the execution time. Hence, it gives us the quadratic importance of the execution time.

6.4 Methodology

6.4.1 Benchmarks

Ten applications written in C/C++ language and already parallelized with OpenMP from different well known benchmark suites were chosen. We have considered small and medium input sizes. The execution time for the medium input size is between 3 and 4 times the small input size. For some benchmarks, the medium input size can use even 4 times more memory. Finally, we detail each benchmark in the following paragraphs.

Five kernels from the NAS Parallel Benchmark. It is a suite originally developed by NASA that comprises applications derived from computational fluid dynamic (BAILEY et al., 1991). As the original version of NAS is written in FORTRAN, we consider the OpenMP-C version developed by Seo et al. 2001 (SEO; JO; LEE, 2011). The kernels used from the NAS Parallel Benchmark were:

- *Conjugate Gradient* (CG) estimates an eigenvalue of a symmetric positive sparse matrix. The core of CG is a solution of a sparse system of linear equations by iterations of the conjugate gradient method.
- *Unstructured Adaptive* (UA) deals with a solution of a stylized heat transfer problem in a cubic domain, discretized on an adaptively refined, unstructured mesh (FENG et al., 2004).
- *Lower-upper gauss-seidel solver* (LU), *Scalar penta-diagonal solver* (SP), and *Block tri-diagonal solver* (BT): each one implements a different method to compute a syn-

thetic nonlinear system of partial differential equations. LU employs a symmetric successive over-relaxation numerical scheme to solve a regular-space, block(5×5) lower and upper triangular system; and SP and BT are used to solve multiple independent systems of non-diagonally dominant through scalar pentadiagonal and block tridiagonal equations, respectively.

Four applications from different domains:

- *Fast-Fourier Transform* (FFT): an algorithm that computes the discrete Fourier transform of a sequence of elements. It samples a signal over a period of time and divides it into its frequency components. This algorithm is widely used in many areas, such as engineering, science, and mathematics (PETERSEN; ARBENZ, 2004).
- *STREAM* (ST): a synthetic benchmark used to measure the sustainable memory bandwidth through mathematical operations performed over long vectors (MCCALPIN, 1995).
- *Jacobi* (JA) method iteration computes the solutions of a diagonally dominant system of linear equations (QUINN, 2004).
- *n-body* (NB) computes a simulation of a dynamical system of particles (BHATT et al., 1992).

One application from the Rodinia benchmark (CHE et al., 2009):

- *Stream Cluster* (SC) solves the online clustering problem. For a stream of input points, it finds a pre-determined number of medians so that each point is assigned to its nearest center. The sum of squared distances metric measures the quality of the clustering.

6.4.2 Execution Environment

The experiments were performed on real multicore systems (see Table 6.1). The system uses the Ubuntu Operating System with kernel v. 4.15.0. All the benchmarks were compiled using GCC/G++ v. 6.3.0 with the -O3 optimization flag. The execution time and energy consumption of each benchmark were obtained through the *omp_get_wtime()* function (from OpenMP) and directly from the hardware counters through the Intel Running Average Power Limit (RAPL) (HäHNEL et al., 2012), respectively. We consider the total energy consumption as the sum of the energy spent by the DRAM modules and

core domains (CPU and cache memories). Also, the collected results take into account the whole application execution, i.e., we consider both sequential and parallel regions.

Table 6.1: Main characteristics of the systems

	Intel Xeon	
Processor	E5-2630	E5-2640
# Cores	2x6	2x8
# Threads	24	32
CPU Freq. range	1.2 - 2.3 GHz (12 levels)	1.2 - 2.0 GHz (9 levels)
L1 Cache	12x32 KB	16x32 KB
L2 Cache	12x256 KB	16x256 KB
L3 Cache	30 MB	40 MB
RAM	32 GB	64 GB

Source: The Author

We define as the **baseline** the regular way of executing parallel applications in multicore systems (LEE et al., 2010): the application executes with the maximum number of threads available in the system and the CPU frequency is set to adjust according to the workload application, using *ondemand* as DVFS governor.

6.5 Experimental Results

In this section, we show the results related to the strategies presented in Chapter 4, both online and offline approaches. For *#Threads + DVFS (T+F)* and *DVFS + #Threads (F+T)* strategies, we present the results in Appendix A.

First, we discuss the costs of training for each benchmark optimized for every online strategy running on the 24 and 32 cores systems. Then, compare all approaches, including the static algorithms.

6.5.1 Online Learning Overhead

Tables 6.2 and 6.3 show the normalized learning overhead – the percentage of the execution time that the algorithm spent searching for the optimum configuration – of each benchmark and the geometric mean (GMEAN), with the small (S) and medium (M) inputs, for the 24 and 32 cores system, respectively.

One can observe in both Tables 6.2 and 6.3 that for all strategies, the CG bench-

Table 6.2: Learning time (%) of each benchmark for the online strategies – 24 cores system

	Aurora		**Fib_DVFS**		**Odin**		**Odin_pruned**	
	S	M	S	M	S	M	S	M
FFT	0.019	0.006	0.028	0.002	0.019	0.006	0.035	0.004
NB	0.097	0.003	0.012	1.9e-4	0.040	0.003	0.046	0.002
SC	0.503	0.259	0.265	0.161	0.583	0.286	0.501	0.275
JA	0.933	0.490	0.672	0.353	0.846	0.527	1.029	0.544
LU	6.008	4.806	3.332	4.339	5.385	8.362	5.385	8.798
BT	6.073	6.013	4.008	5.476	6.873	8.684	7.134	7.579
CG	17.264	17.314	11.536	11.671	21.486	22.293	19.463	24.182
SP	2.966	2.691	1.700	1.990	3.827	3.583	3.605	3.328
UA	1.454	1.379	0.692	0.743	1.551	1.345	1.424	1.392
ST	0.784	0.640	0.685	0.671	1.103	1.223	1.086	1.511
GMEAN	1.063	0.554	0.612	0.300	1.069	0.694	1.132	0.650

Source: The Author

Table 6.3: Learning time (%) of each benchmark for the online strategies – 32 cores system

	Aurora		**Fib_DVFS**		**Odin**		**Odin_pruned**	
	S	M	S	M	S	M	S	M
FFT	0.033	0.006	0.001	3.9e-4	0.032	0.008	0.024	0.005
NB	0.101	0.005	0.003	0.003	0.096	0.006	0.083	0.005
SC	0.621	0.280	0.267	0.133	0.696	0.382	0.535	0.251
JA	1.474	0.790	0.591	0.304	1.110	0.565	1.129	0.575
LU	8.478	5.730	2.785	3.557	5.427	7.311	3.755	7.460
BT	9.169	9.407	3.215	4.117	6.534	8.311	5.780	6.180
CG	25.257	24.580	13.922	12.278	21.442	27.208	22.496	15.893
SP	4.700	3.670	1.792	1.639	4.346	3.818	3.005	3.666
UA	2.072	2.063	0.624	0.586	1.626	1.472	1.230	1.277
ST	1.225	0.825	0.650	0.681	1.218	1.151	1.355	1.289
GMEAN	1.532	0.748	0.366	0.298	1.316	0.802	1.114	0.661

Source: The Author

mark is the only one that presents an overhead higher than 10%. This behavior happens because the CG benchmark has only 75 repetitions, while the other benchmarks repeat hundreds or even thousands of times the same parallel region. Besides that, while the input size is related to the number of iterations for some benchmarks (e.g., JA and FFT), it is associated with the size of each parallel region for other benchmarks (e.g., CG and ST). Therefore, one can observe a reduction on the overhead for JA and FFT benchmarks when the input increases, as one can see in Tables 6.2 and 6.3 for the medium input size.

Finally, the impact of the learning overhead is minimal in most of the benchmarks considering strategies, input size, and systems. Furthermore, as discussed before, the application uses the computation during the learning phase, which helps to reduce the training cost. Thus, we can assume that the configuration found by each algorithm is the main factor in the quality of the final EDP result.

6.5.2 EDP Comparison

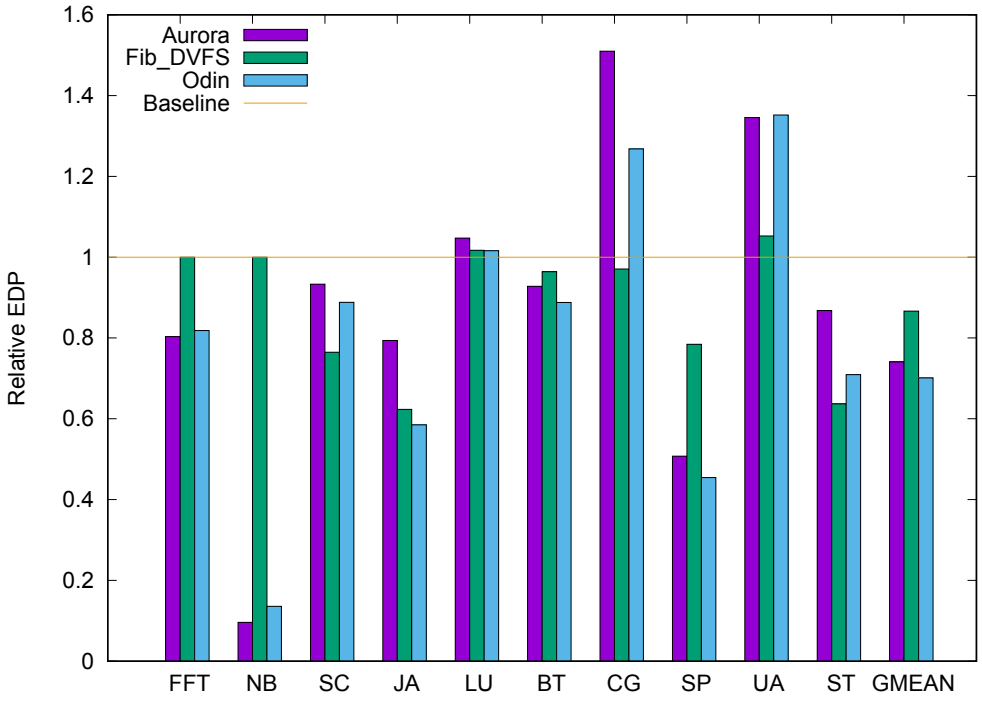
In this section, we discuss the EDP results. We first analyze the strategies that use only one knob (DCT or DVFS) for optimization: Aurora and Fib_DVFS. After that, we examine the results related to Odin, which tunes the two knobs. Next, we dedicate a section to analyze the Odin_pruned (Odin with a pruned search space) compared to the baseline, Aurora, and Odin. Finally, we compare the best dynamic strategy (Odin_pruned) to the static algorithms, Static_GA and OPT_CEIR.

6.5.2.1 Strategies using only one knob vs Odin

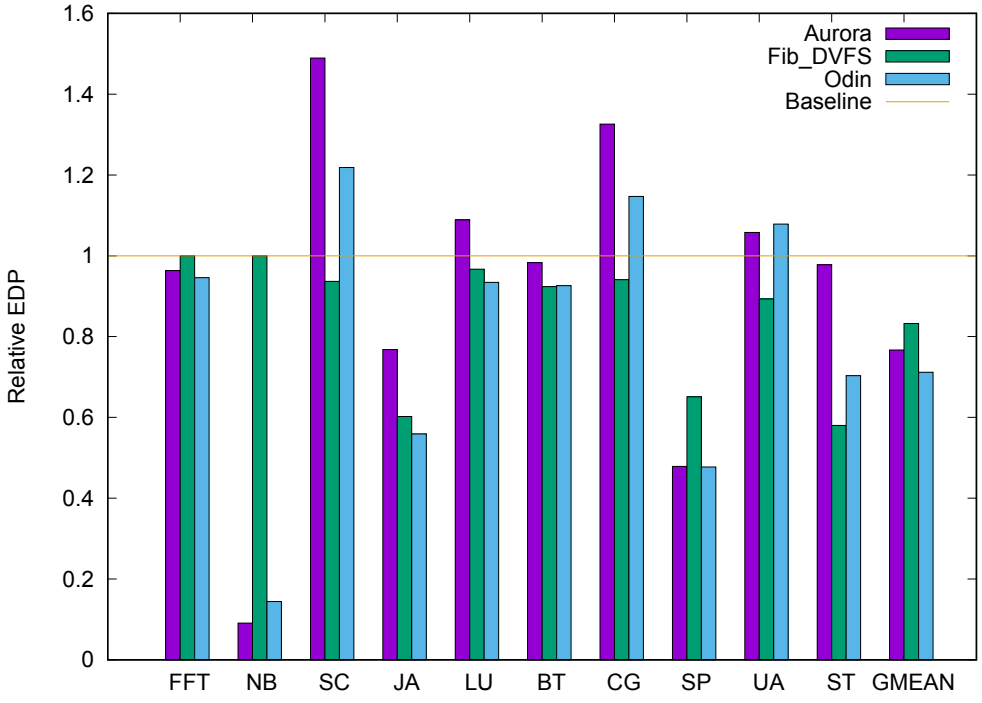
Figures 6.10 and 6.11 present the results for the EDP normalized to the baseline of the entire benchmark set and the geometric mean (GMEAN) using the small and medium input sets when running on the 24 cores and 32 cores machine, respectively. The strategies under comparison are Aurora, Fib_DVFS, and Odin (the original version without the pruning on DCT search). Values below than 1.0 represent an EDP better than the baseline.

Let us first discuss the results achieved by Aurora and Fib_DVFS running on the 24-core system (Figure 6.10). For both input sets, small and medium, on average represented by the GMEAN, the strategy of employing DCT presents more potential to enhance EDP in comparison to the one using only DVFS. On average, Aurora improves the EDP of the baseline by 25.9% and 23.3% for the small and medium inputs, respectively. Besides that, some benchmarks such as NB and SP depict the substantial impact of DCT on the final result, presenting 90.1% and 52.1% of EDP improvements, respectively, for the medium input. Also, the high enhancement on the NB benchmark result, compared to all the other applications, is due to the short execution time of the parallel region. Therefore, when using the maximum number of threads, the critical region time exceeds it in up to 8x. In the case of Fib_DVFS, the GMEAN outcomes are 13.3% and 16.7% better than the baseline, for small and medium input, respectively. It presents better results for JA

Figure 6.10: The results from each benchmark normalized to the baseline EDP (24 logical cores).



(a) SMALL INPUT

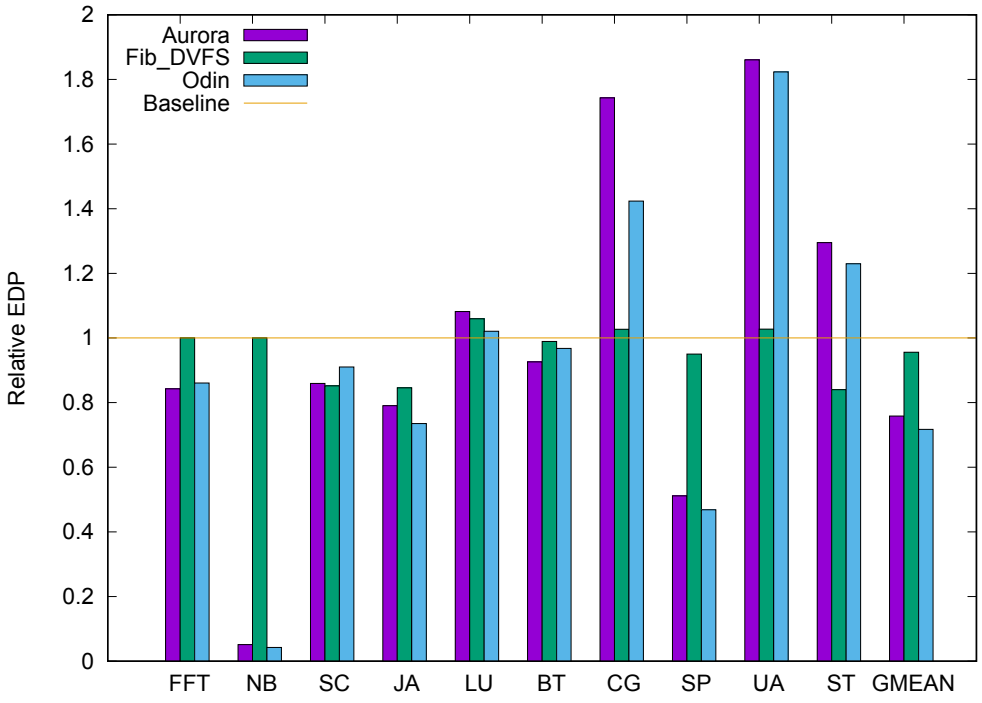


(b) MEDIUM INPUT

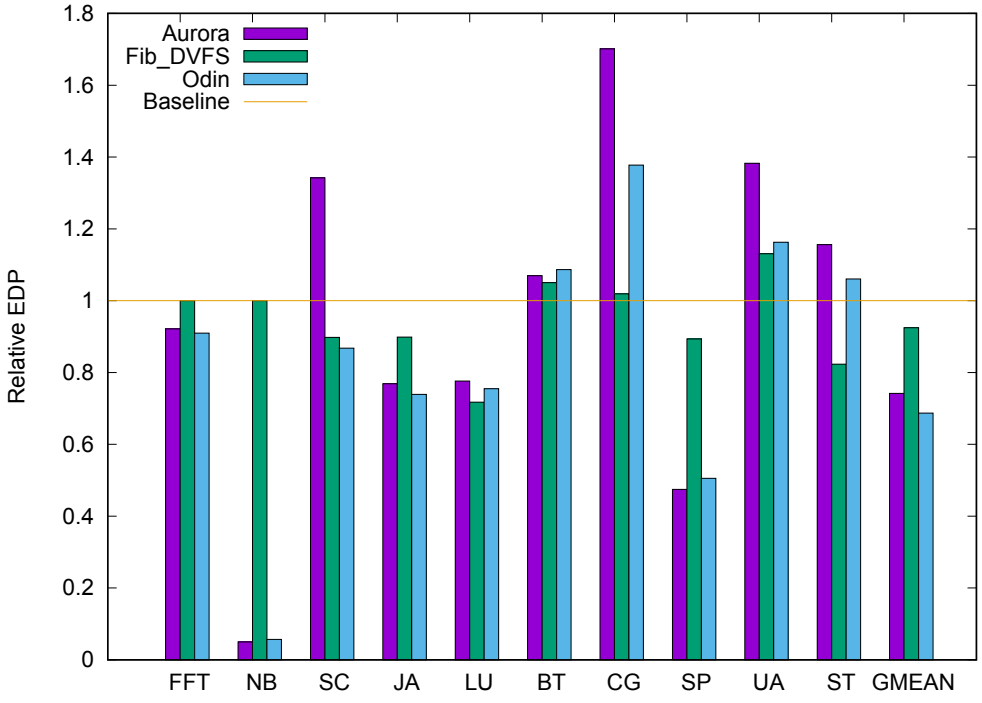
Source: The Author

and ST benchmarks (39.8% and 42.0% of EDP improvements) compared to the baseline using the medium input. These applications are memory-bound, i.e., they proportionally

Figure 6.11: The results from each benchmark normalized to the baseline EDP (32 logical cores).



(a) SMALL INPUT



(b) MEDIUM INPUT

Source: The Author

make many memory requests in comparison to the time wasted executing the fetched data, which guarantees an excellent opportunity for CPU frequency scaling with little impact

on performance.

Now, we point the essential details for these strategies running on the 32 cores system (shown in Figure 6.11). Let us start with Aurora. On average, the EDP results compared to the baseline are similar to the 24-core machine, being 24.1% and 25.8% better for the small and medium input, respectively. However, with the expansion of the search space, there is a significant increase in the EDP results of some benchmarks when we compare to the baseline. For example, we see that the CG and UA benchmarks for the small input – they are already worse than the baseline in the 24 cores machine – increase their EDP outcomes relative to the baseline by 23.3% and 51.5%, respectively. For the Fib_DVFS, on average, the improvements in EDP results compared to the baseline are 4.4% and 7.5% for small and medium input, respectively. There is a decrease in EDP gains observed in the 32 cores machine compared to the 24 cores system because the former has a reduced range on the supported DFVS settings in comparison to the latter.

Besides that, when we compare the overall benchmarks results using DCT (this also includes Odin, which we discuss next) to DVFS techniques, one can see that algorithms that use DCT are more prone to worse the final result. It is due to the noise caused by the DCT, which prevents the algorithm from finding the optimum configuration, as we have shown in Section 6.3. In the case of Fib_DVFS, when this strategy is worse than the baseline, the exceeded value is minimal compared to the approaches employing DCT.

As one can observe, running on the 24-core machine (see Figure 6.10), on average, Odin presents better results than the other strategies. Considering the GMEAN, one can see an improvement of 29.9% and 28.8% in the EDP when compared to the baseline, for small and medium input, respectively. The first thing to notice is that the gains of each knob are not accumulative because both techniques can solve the same bottlenecks. Consider, for example, the JA benchmark using the medium input. Although the Aurora and Fib_DVFS achieve 33.2% and 39.8% in EDP reduction, respectively, Odin that employs the two knobs is only 44.1% better than baseline. In this case, when Odin uses DCT first, it diminishes the number of threads and alleviates the simultaneous memory requests received by the off-chip memory. Thus, when the algorithm uses DVFS, the slice of time caused by the processor waiting for the memory system is small and, hence, it signifies a minor impact on the final result.

Finally, running on the 32 cores machine, on average, Odin presents EDP gains of 28.2% and 31.3% compared to the baseline, for small and medium input, respectively (shown in Figure 6.11). Therefore, for both systems and size inputs, Odin achieves a

better outcome in EDP gains represented by the GMEAN.

Therefore, considering all the benchmarks, one can see the importance of using the two knobs (DCT and DVFS) to achieve the best improvements in EDP results. As some benchmarks are more prone to be optimized by DCT and others present better results when applying DVFS, we can cover the most applications employing the two dynamic techniques. Besides, there are benchmarks such as JA that show better results when optimized by Odin on all systems and input sizes in comparison to approaches using only one knob (see Figures 6.10 and 6.11).

6.5.2.2 Odin pruned

Figures 6.12 and 6.13 show the results for Odin_pruned running on the 24 cores and 32 cores machines, respectively. In these results, we keep the same baseline as before and maintain the two best strategies, Aurora and Odin, for comparison.

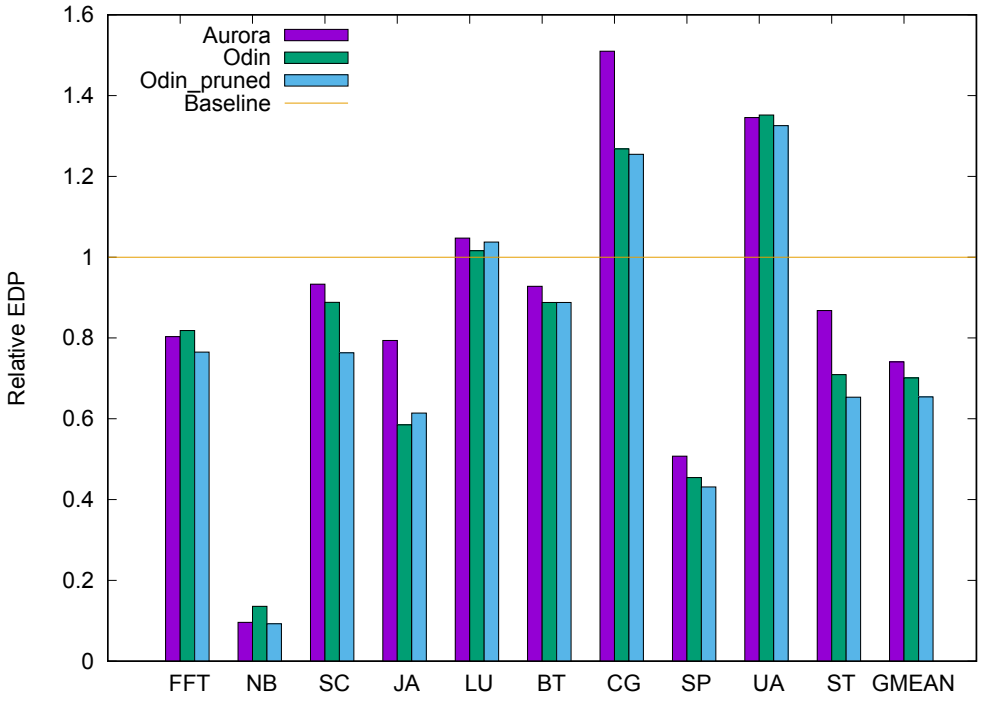
We start analyzing the results for the 24 cores machine (see Figure 6.12). On average, for both small and medium inputs, Odin_pruned improves the EDP outcomes compared to the other strategies. Considering the small input (6.12a), the Odin_pruned GMEAN is 34.6%, 8.7%, and 4.7% better than baseline, Aurora and Odin, respectively. When using the medium input (6.12b), Odin_pruned presents GMEAN 33.2%, 9.9%, and 4.4% better than baseline, Aurora and Odin, respectively. Also, we see that both ST and SC, memory-bound applications, which may present the memory locality noise detailed in section 6.3, show results improvements for both input sizes in comparison to Odin that searches on the whole space.

Finally, Figure 6.13 presents the results for the strategies running on the 32 cores system. The best outcome, on average, is for the medium input that shows 37.6%, 11.8%, and 6.3% of EDP improvement compared to the baseline, Aurora, and Odin, respectively. Also, thus like the other strategies employing DCT, Odin_pruned is prone to miss the optimal configuration that ends in high damage in the outcome of benchmarks such as CG and UA using the small input.

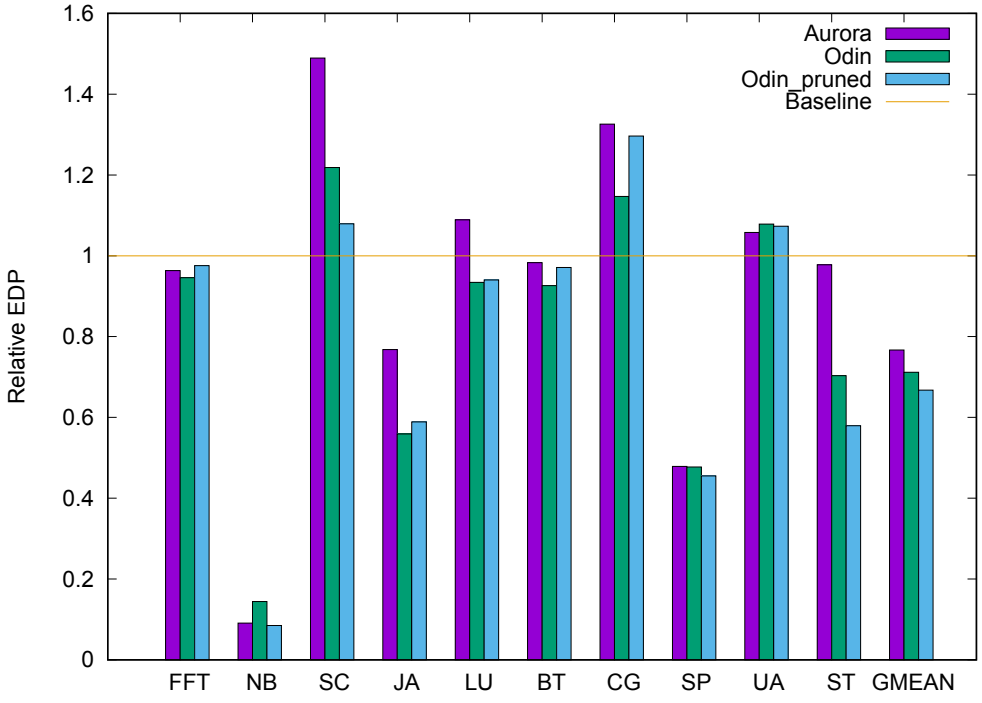
6.5.2.3 Dynamic x Static strategies

In this section, we compare both static strategies, Static_GA and OPT_CEIR, to the best of the given dynamic approaches, Odin_pruned. Figure 6.14 shows the EDP results for the shared benchmark set and geometric mean (GMEAN) to the Static_GA,

Figure 6.12: The results from each benchmark normalized to the baseline EDP (24 logical cores).



(a) SMALL INPUT

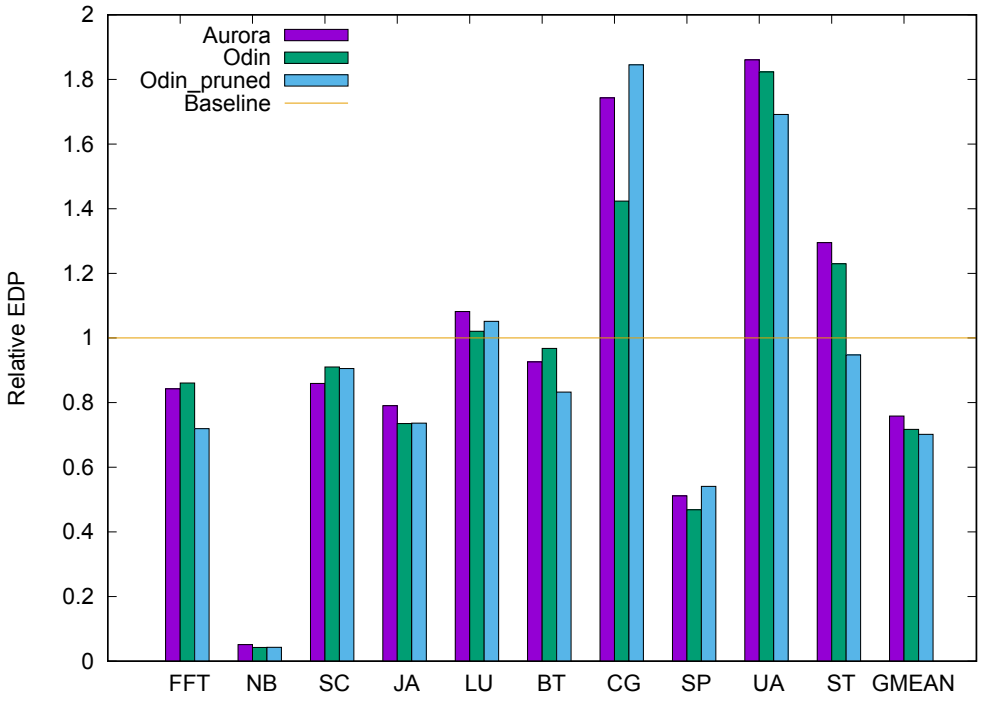


(b) MEDIUM INPUT

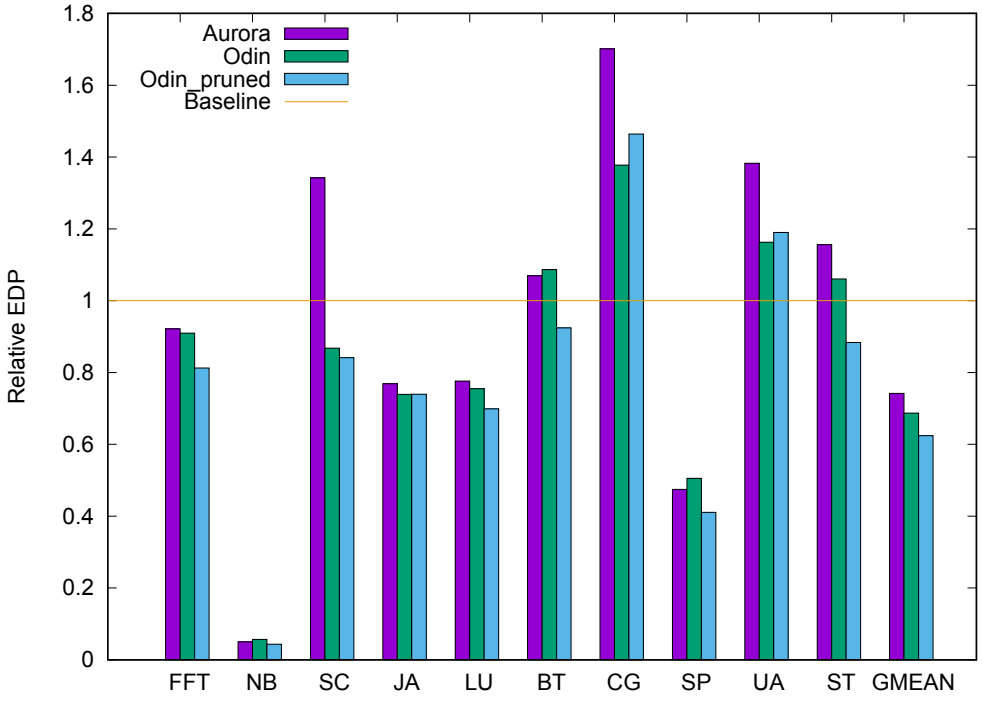
Source: The Author

OPT_CEIR, and Odin_pruned algorithms relative to the same baseline as before. The outcomes are for the small input set running on the 24 cores machine.

Figure 6.13: The results from each benchmark normalized to the baseline EDP (32 logical cores).



(a) SMALL INPUT

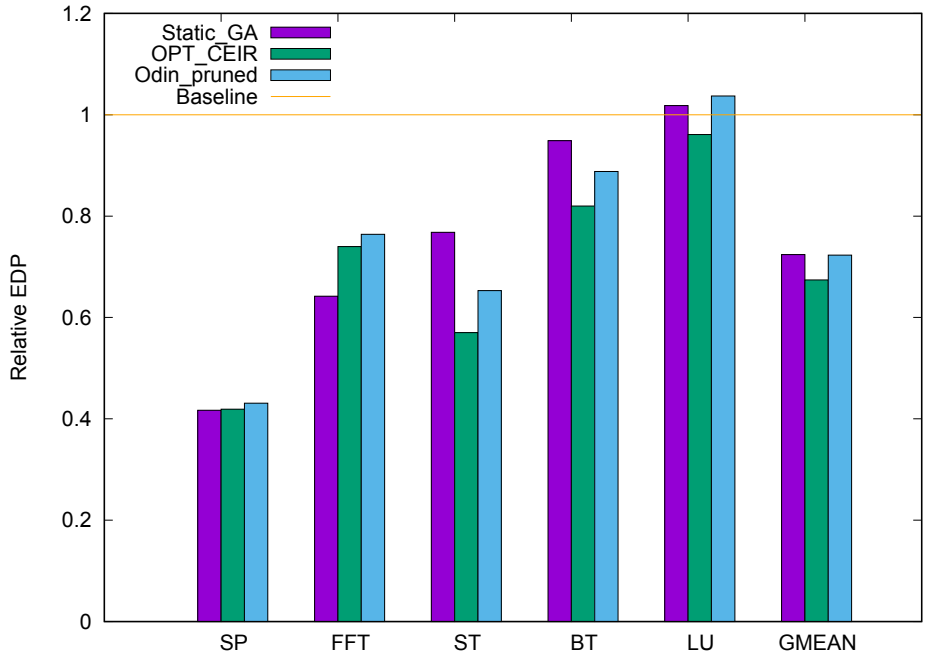


(b) MEDIUM INPUT

Source: The Author

On average, represented by the GMEAN, the OPT_CEIR, Odin_pruned, and Static_GA approaches achieve 32.6%, 27.7%, and 27.6% of EDP improvements relative

Figure 6.14: The results for Dynamic and Static strategies normalized to the baseline EDP.



Source: The Author

to the baseline, respectively. As expected, OPT_CEIR always has better results than Odin_pruned since they work by optimizing each parallel region independently, but OPT_-CEIR is the optimum configuration without the learning overhead. Comparing OPT_CEIR and Static_GA, one can see that only on the FFT benchmark that the latter have a significant gain relative to the former, which is an improvement of 9.8% in EDP.

Now, we analyze Odin_pruned in comparison to the Static_GA. One can see an equilibrium between the outcomes of the two approaches, which is sustained when we look at the small difference of only 0.1% of GMEAN results. The FFT and ST are the benchmarks that present a significant difference between the two methods. While the former is 12.2% better in EDP when executed by the Static_GA compared to the Odin_pruned, the latter shows 11.5% of EDP improvements when applying the Odin_pruned in comparison to the Static_GA. Therefore, Odin_pruned shows similar results compared to a robust different strategy, Static_GA, and when compared to the OPT_CEIR, it is only 4.8% worse, which is a small difference because Odin_pruned have the overhead of the online training.

7 FINAL CONSIDERATIONS

We have presented Odin, an approach capable of automatically finding, at runtime, the optimal or the near optimal number of threads and CPU operating frequency for each parallel region. It is completely transparent to both designer and end user: given an OpenMP application binary, Odin optimizes it without any code changes, transformation or recompilation, by simply setting an environment variable in Linux OS. We have shown that Odin can optimize distinct OpenMP applications.

Furthermore, to see the potentials of gains achieved by tuning DVFS and DCT using a different approach, we have implemented a static framework, which also maintains the software transparency, that uses a genetic algorithm to optimize OpenMP applications. In contrast with the main tool of this dissertation that performs by improving the EDP of each parallel region, the GA works by searching a near optimal set of configurations that fit the whole application. To validate our static approach, we used eight benchmarks. Thus, we have shown that our proposed strategy can optimize most of the tested benchmarks, improving the EDP of OpenMP applications by up to 58.3% when compared to the regular way that parallel applications are executed.

To validate Odin, first, we have compared it to our baseline (the maximum number of threads using *Ondemand* as DVFS governor) and other online approaches, employing only one knob: Fib_DVFS applying DVFS and Aurora using DCT. We have shown gains of 31.3%, 23.8%, and 5.5%, on average, when compared to the baseline, Fib_DVFS, and Aurora, respectively. Next, we have presented results for Odin_pruned, a variant of Odin with a reduced DCT search space. It shows improvements of 37.6%, 11.8%, and 6.3%, on average, in comparison to the baseline, Aurora, and Odin, respectively. Additionally, we compared our best strategy, Odin_pruned, to both offline approaches. In comparison to OPT_CEIR – the Optimal Configuration for Each Individual Region –, Odin_pruned is only 4.8% worse, on average, which is a small difference considering that Odin_pruned has an online learning overhead. Finally, when we compare Odin_pruned to the Static_GA, on average, the difference is only 0.1%, showing similar results for different optimization approaches.

7.1 Future Work

In this section, we discuss potential points of improvement to our approach and promising future works.

First, it is desirable better handling of the noise during the online training. To detect an anomaly on samples during the learning step, we set a static threshold for the instruction miss TLB, so we can discard samples that exceed it. Although this strategy will work on most processors with similar TLB sizes, a better alternative can turn it adaptive or even employ a method to filter a sample contaminated by noise. Besides, one can also implement a direct alternative to diminish the noise impact caused by memory locality in a NUMA architecture. The possible ways are:

- To control the affinity during the training step, trying to minimize the data spreading between the NUMA nodes.
- Working with the Linux automatic NUMA balancing, which has configuration parameters accessible through the *sysfs* interface.

Also, one can extend our strategy to use thread affinity as another knob for optimization. In a complex system including NUMA architecture and processors with SMT, the choice of placement of the threads may improve the results in some benchmarks. For example, some applications have a parallel region with small execution time compared to the critical section, so an extended part of the work is for exchanging information. Therefore, these applications may benefit from an arrangement of threads in cores close to each other.

Finally, as we have stated, the embedded systems represent a significant target for parallel applications. On top of that, a lot of current embedded systems are rapidly increasing in complexity, increasing the power consumption and probably wasting more energy. However, our results are concerned only to HPC systems. Therefore, it is necessary to adapt and test our strategy in these devices.

REFERENCES

AKRAM, S.; SARTOR, J. B.; EECKHOUT, L. DVFS performance prediction for managed multithreaded applications. In: **ISPASS**. [S.l.: s.n.], 2016. p. 12–23.

ALESSI, F. et al. Application-level energy awareness for OpenMP. In: ____. **OpenMP: Heterogenous Execution and Data Movements: 11th IWOMP**. Cham: Springer, 2015. p. 219–232. ISBN 978-3-319-24595-9.

BAILEY, D. H. et al. The nas parallel benchmarks—summary and preliminary results. In: **ACM/IEEE Conf. on Supercomputing**. NY, USA: ACM, 1991. p. 158–165. ISBN 0-89791-459-7.

BEAMER, S.; ASANOVIĆ, K.; PATTERSON, D. The gap benchmark suite. **arXiv preprint arXiv:1508.03619**, 2015.

BHATT, S. et al. Abstractions for parallel N-body simulations. In: **Scalable HPC Conf.** [S.l.: s.n.], 1992. p. 38–45.

CHADHA, G.; MAHLKE, S.; NARAYANASAMY, S. When less is more (limo): controlled parallelism forimproved efficiency. In: ACM. **Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems**. [S.l.], 2012. p. 141–150.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)**. [S.l.]: The MIT Press, 2007. ISBN 0262533022, 9780262533027.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: IEEE. **Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on**. [S.l.], 2009. p. 44–54.

CHEN, Y. L. et al. Performance and energy efficient dynamic voltage and frequency scaling scheme for multicore embedded system. In: **IEEE ICCE**. [S.l.: s.n.], 2016. p. 58–59.

COCHRAN, R. et al. Pack & cap: Adaptive DVFS and thread packing under power caps. In: **IEEE/ACM MICRO**. USA: [s.n.], 2011. p. 175–185. ISBN 978-1-4503-1053-6.

CORMEN, T. H. et al. **Introduction to Algorithms, Third Edition**. 3rd. ed. [S.l.]: The MIT Press, 2009. ISBN 0262033844, 9780262033848.

CURTIS-MAURY, M. et al. Prediction-based power-performance adaptation of multithreaded scientific codes. **IEEE Trans. Parallel Distrib. Syst.**, v. 19, n. 10, p. 1396–1410, 2008.

CURTIS-MAURY, M. et al. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In: **Int. CS**. [S.l.: s.n.], 2006. p. 157–166.

FENG, H. et al. Unstructured adaptive (ua) nas parallel benchmark. version 1.0. 2004.

GE, R. et al. CPU MISER: A performance-directed, run-time system for power-aware clusters. In: **ICPP**. [S.l.: s.n.], 2007. p. 18–18. ISSN 0190-3918.

GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization and Machine Learning**. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675.

GONZALEZ, R.; HOROWITZ, M. Energy dissipation in general purpose microprocessors. **IEEE Journal of solid-state circuits**, IEEE, v. 31, n. 9, p. 1277–1284, 1996.

HACKENBERG, D. et al. Power measurement techniques on standard compute nodes: A quantitative comparison. In: **IEEE ISPASS**. [S.l.: s.n.], 2013. p. 194–204.

HäHNEL, M. et al. Measuring energy consumption for short code paths using rapl. **SIGMETRICS Perform. Eval. Rev.**, ACM, NY, USA, v. 40, n. 3, p. 13–17, 2012. ISSN 0163-5999.

HOTTA, Y. et al. Profile-based optimization of power performance by using dynamic voltage scaling on a pc cluster. In: **IEEE IPDPS**. [S.l.: s.n.], 2006.

HSU, C. hsing; FENG, W. chun. A power-aware run-time system for high-performance computing. In: **ACM/IEEE CS**. [S.l.: s.n.], 2005. p. 1–1.

JOAO, J. A. et al. Bottleneck identification and scheduling in multithreaded applications. In: **ASPLOS**. NY, USA: ACM, 2012. p. 223–234. ISBN 978-1-4503-0759-8.

JOUPPI, N. P.; WALL, D. W. **Available instruction-level parallelism for superscalar and superpipelined machines**. [S.l.]: ACM, 1989.

JUNG, C. et al. Adaptive execution techniques for SMT multiprocessor architectures. In: **ACM Symp. on Principles and Practice of Parallel Programming**. USA: [s.n.], 2005. p. 236–246. ISBN 1-59593-080-9.

KAXIRAS, S.; MARTONOSI, M. Computer architecture techniques for power-efficiency. **Synthesis Lectures on Computer Architecture**, Morgan & Claypool Publishers, v. 3, n. 1, p. 1–207, 2008.

KIEFER, J. Sequential minimax search for a maximum. **Proc. American Mathematical Society**, p. 502–506, 1953.

KOBUSIŃSKA, A. et al. **Emerging trends, issues and challenges in Internet of Things, Big Data and cloud computing**. [S.l.]: Elsevier, 2018.

LEE, J. et al. Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. **SIGARCH Comput. Archit. News**, ACM, NY, USA, v. 38, n. 3, p. 270–279, 2010. ISSN 0163-5964.

LI, D. et al. Hybrid MPI/OpenMP power-aware computing. In: **IEEE IPDPS**. [S.l.: s.n.], 2010. p. 1–12. ISSN 1530-2075.

LI, J.; MARTINEZ, J. F. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In: **HPCA**. [S.l.: s.n.], 2006. p. 77–87.

LORENZON, A. F. et al. Aurora: Seamless optimization of OpenMP applications. **IEEE Transactions on Parallel and Distributed Systems**, p. 1–1, 2018. ISSN 1045-9219.

LORENZON, A. F.; SOUZA, J. D.; BECK, A. C. S. LAANT: A library to automatically optimize EDP for OpenMP applications. In: **DATE**. [S.l.: s.n.], 2017. p. 1229–1232.

MARATHE, A. et al. A run-time system for power-constrained hpc applications. In: KUNKEL, J. M.; LUDWIG, T. (Ed.). **High Performance Computing**. Cham: Springer, 2015. p. 394–408. ISBN 978-3-319-20119-1.

MCCALPIN, J. D. Memory bandwidth and machine balance in current high performance computers. **IEEE Computer Society Technical Committee on Computer Architecture Newsletter**, p. 19–25, 1995.

MIFTAKHUTDINOV, R. R. **Performance Prediction for Dynamic Voltage and Frequency Scaling**. Thesis (PhD) — The University of Texas, 2014.

PALLIPADI, V.; STARIKOVSKIY, A. The ondemand governor. In: SN. **Proceedings of the Linux Symposium**. [S.l.], 2006. v. 2, n. 00216, p. 215–230.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design MIPS Edition: The Hardware/Software Interface**. [S.l.]: Newnes, 2013.

PETERSEN, W.; ARBENZ, P. **Introduction to Parallel Computing : A practical guide with examples in C**. [S.l.]: OUP Oxford, 2004. (Oxford Texts in Applied and Engineering Mathematics). ISBN 9780191513619.

PORTERFIELD, A. K. et al. Power measurement and concurrency throttling for energy reduction in OpenMP programs. In: **IEEE IPDPS**. [S.l.: s.n.], 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 ISWC**. USA: [s.n.], 2011. p. 116–125. ISBN 978-1-4577-2063-5.

QUINN, M. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Higher Education, 2004. ISBN 9780072822564.

RAASCH, S. E.; REINHARDT, S. K. The impact of resource partitioning on SMT processors. In: **PACT**. [S.l.: s.n.], 2003. p. 15–25. ISSN 1089-795X.

RAMAN, A. et al. Parcae: A system for flexible parallel execution. In: **ACM SIGPLAN PLDI**. NY, USA: ACM, 2012. p. 133–144. ISBN 978-1-4503-1205-9.

RIZVANDI, N. B. et al. Linear combinations of DVFS-enabled processor frequencies to modify the energy-aware scheduling algorithms. In: **CCGRID**. [S.l.: s.n.], 2010. p. 388–397.

ROSSI, F. D. et al. Modeling power consumption for DVFS policies. In: **IEEE ISCAS**. [S.l.: s.n.], 2015. p. 1879–1882. ISSN 0271-4302.

S., T. C. D. et al. Comparison of parallel programming models for multicore architectures. In: **IEEE IPDPS**. [S.l.: s.n.], 2011. p. 1675–1682. ISSN 1530-2075.

SENSI, D. D. Predicting performance and power consumption of parallel applications. In: **PDP**. [S.l.: s.n.], 2016. p. 200–207.

SENSI, D. D.; TORQUATI, M.; DANELUTTO, M. A reconfiguration algorithm for power-aware parallel applications. **TACO**, v. 13, n. 4, p. 43:1–43:25, 2016.

SEO, S.; JO, G.; LEE, J. Performance characterization of the nas parallel benchmarks in opencl. In: **IEEE Int. Symp. on Workload Characterization**. [S.l.: s.n.], 2011. p. 137–148.

SHAFIK, R. A. et al. Adaptive energy minimization of openmp parallel applications on many-core systems. In: **WPPRTMTMA**. NY, USA: ACM, 2015. p. 19–24. ISBN 978-1-4503-3343-6.

SHAFIK, R. A. et al. Thermal-aware adaptive energy minimization of OpenMP parallel applications. 2015.

SRIDHARAN, S.; GUPTA, G.; SOHI, G. S. Holistic run-time parallelism management for time and energy efficiency. In: **Int. Conf. on Supercomputing**. NY, USA: ACM, 2013. p. 337–348. ISBN 978-1-4503-2130-3.

SRIDHARAN, S.; GUPTA, G.; SOHI, G. S. Adaptive, efficient, parallel execution of parallel programs. In: **ACM SIGPLAN PLDI**. NY, USA: ACM, 2014. p. 169–180.

SULEMAN, M. A.; QURESHI, M. K.; PATT, Y. N. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. **SIGARCH Comput. Archit. News**, ACM, NY, USA, v. 36, n. 1, p. 277–286, 2008. ISSN 0163-5964.

TOP500. 2019. <https://www.top500.org>. Accessed: 2019-01-17.

WHEELER, K. B.; MURPHY, R. C.; THAIN, D. Qthreads: An API for programming with millions of lightweight threads. In: **IEEE IPDPS**. [S.l.: s.n.], 2008. p. 1–8. ISSN 1530-2075.

WU, Q. et al. Dynamic-compiler-driven control for microprocessor energy and performance. **IEEE Micro**, v. 26, n. 1, p. 119–129, Jan 2006. ISSN 0272-1732.

APPENDIX A — OTHER STRATEGIES

Before we use Fibonacci for searching, we have implemented other search strategies based on Aurora (LORENZON et al., 2018). Here, we show the descriptions and results for them.

A.1 Description: #Threads+DVFS (T+F) and DVFS+#Threads (F+T)

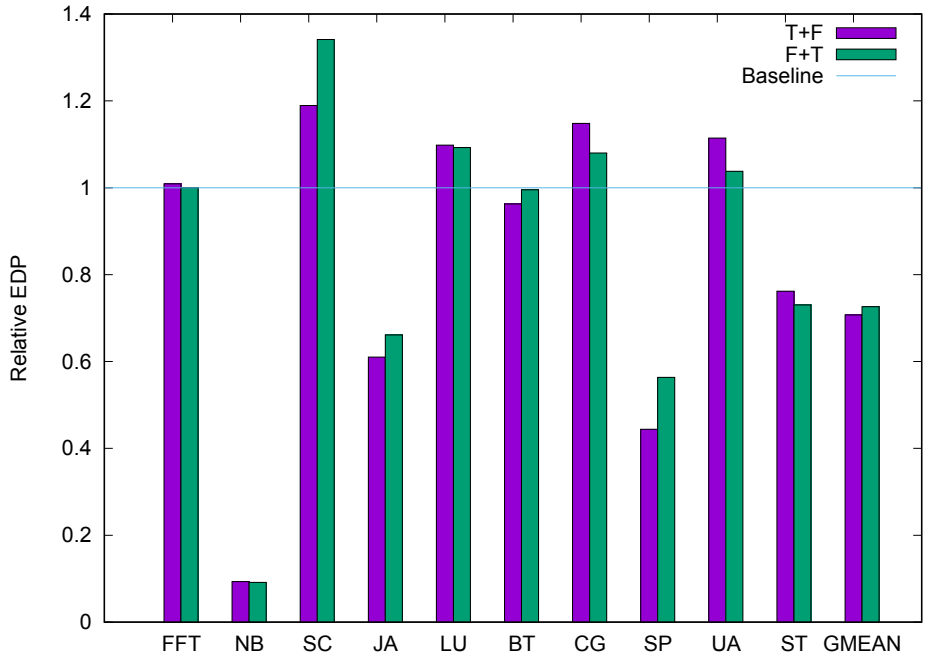
T+F seeks for the best number of threads with the operating frequency set to the maximum possible value and, once the number of threads is found, it tests different values for the operating frequency with that fixed number of threads. It uses the same hill-climbing based search strategy employed in Aurora, which we describe in Chapter 4, for both threads and operating frequency searches. F+T is the same as the previous, but works in the inverse order, i.e. it first optimizes the operating frequency with the maximum number of threads, and then searches for the number of threads using the operating frequency previously found.

So for a parallel region i, T+F first finds the number of threads t_i^* such that $t_i^* \in X_i$ and for all $x \in X_i : edp_i(t_i^*, q) \leq edp_i(x, q)$, where q is the highest frequency level. After that, it finds the frequency level f_i^* such that $f_i^* \in Y_i$ and for all $y \in Y_i : edp(t_i^*, f_i^*) \leq edp(t_i^*, y)$, where Y_i is the set of frequency levels visited by the search. In the same way, for a parallel region i, F+T first finds the best frequency level f_i^* such that $f_i^* \in Y_i$ and for all $y \in Y_i : edp_i(n, f_i^*) \leq edp_i(n, y)$, where n is the maximum number of threads. After that, it finds the number of threads t_i^* such that $t_i^* \in X_i$ and for all $x \in X_i : edp(t_i^*, f_i^*) \leq edp(x, f_i^*)$.

A.2 Results

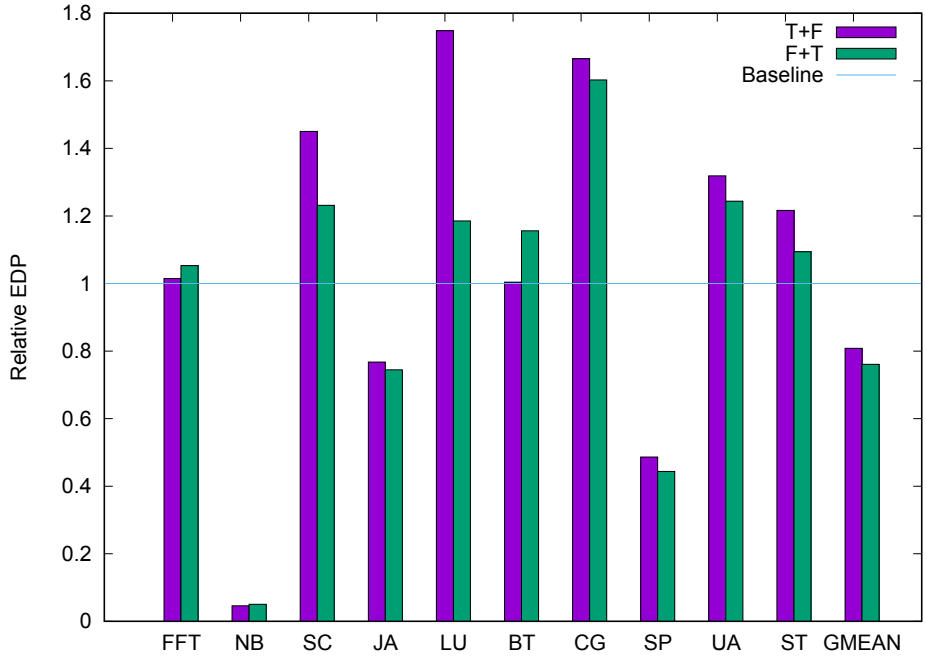
Figures A.1 and A.2 show the results normalized to the same baseline as before (maximum number of threads available and *ondemand* as DVFS governor) for the entire benchmark set and geometric mean (GMEAN) with the medium input size running on 24 and 32 cores systems, respectively.

Figure A.1: The results from each benchmark normalized to the baseline EDP (24 logical cores).



Source: The Author

Figure A.2: The results from each benchmark normalized to the baseline EDP (32 logical cores).



Source: The Author