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

ARTHUR FRANCISCO LORENZON

Aurora: Seamless Optimization of OpenMP Applications

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

Advisor: Prof. Dr. Antonio Carlos Schneider
Beck

Porto Alegre
June 2018

CIP — CATALOGING-IN-PUBLICATION

Lorenzon, Arthur Francisco

Aurora: Seamless Optimization of OpenMP Applications / Arthur Francisco Lorenzon. – Porto Alegre: PPGC da UFRGS, 2018.

183 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: Antonio Carlos Schneider Beck.

1. Parallel computing. 2. Energy and performance optimization. 3. Software tuning. 4. OpenMP. I. Beck, Antonio Carlos Schneider. II. Título.

To the memory of Márcia Cristina Cera.

Thank you so much for everything you have done for me during your life!

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus pela saúde, coragem e persistência para alcançar os objetivos.

Ao Prof. "Caco", que, mostrou ser mais que um orientador de mestrado e doutorado. Se tornou um amigo e orientador de decisões importantes da minha vida. Sou e serei muito grato por todos os teus ensinamentos, puxões de orelha, discussões e momentos em que pude aprender contigo. Seus ensinamentos serão passados adiante. Obrigado por apostar em um guri lá de "cacimbinhas".

À Prof. Márcia Cristina Cera, que, durante sua passagem contribuiu e foi muito importante para minha formação pessoal e profisisonal. Obrigado por ter me ensinado o caminho e ter me dito "vai" quando hesitei. Seu legado será passado adiante.

Ao Prof. Stephan Wong, pelos ensinamentos durante os dois períodos em que fui aluno de doutorado visitante na TU-Delft. Estendo os cumprimentos aos demais amigos que lá fiz e companheiros de futebol.

Ao corpo docente e quadro de funcionários do Instituto de Informática da UFRGS, em especial aos professores Luigi Carro, Flávio Rech, Gabriel Nazar e Paolo Rech.

Ás agências financiadoras CAPES e CNPq, pelo apoio financeiro.

Ao Prof. e amigo Fábio Rossi, que foi o responsável por instigar a vontade e desejo de um jovem de cidade pequena se tornar Dr. em Computação. Talvez tu não compreenda a grandeza de seu gesto e a gratidão que eu tenho por tu ter feito parte deste processo.

Aos colegas do Laboratório de Sistemas Embarcados que essa longa caminhada me possibilitou encontrar. Em especial, os que foram importantes para minha formação: Anderson, Charles, Geraldo, Jeckson, Leonardo, Marcelo, Paulo, Pedro, Tiago e Thiago. Estendo meus cumprimentos à todos os demais integrantes do laboratório.

Aos amigos que compartilharam diversos momentos de minha caminhada, e que sempre me ajudaram independente da situação: Ademir, Brandon, Cadu, Henrique, Jaline, Jean, Juliana, Junior, Sander e Thiarles. Agradeço ainda os irmãos que a trajetória acadêmica me deu: Adriano, Matheus e Uillian.

Por fim, mas o mais importante, à minha familía: Osvaldo, Senilda, Allyne, Jeferson, Matheus e Sônia. Só tenho a agradecer por todo o apoio incondicional. Por, terem me ensinado desde criança que o estudo é o maior bem que podemos carregar. Peço desculpas por não estar presente em vários momentos e datas especiais.

ABSTRACT

Efficiently exploiting thread-level parallelism has been challenging for software developers. As many parallel applications do not scale with the number of cores, blindly increasing the number of threads may not produce the best results in performance or energy. However, the task of rightly choosing the ideal amount of threads is not straightforward: many variables are involved (e.g. off-chip bus saturation and overhead of data-synchronization), which will change according to different aspects of the system at hand (e.g., input set, micro-architecture) and even during execution.

To address this complex scenario, this thesis presents Aurora. It is capable of automatically finding, at run-time and with minimum overhead, the optimal number of threads for each parallel region of the application and re-adapt in cases the behavior of a region changes during execution. Aurora works with OpenMP and is completely transparent to both designer and end-user: given an OpenMP application binary, Aurora optimizes it without any code transformation or recompilation. By executing fifteen well-known benchmarks on four multi-core processors, Aurora improves the trade-off between performance and energy by up to: 98% over the standard OpenMP execution; 86% over the built-in feature of OpenMP that dynamically adjusts the number of threads; and 91% over a feedback-driven threading emulation.

Keywords: Parallel computing. energy and performance optimization. software tuning. OpenMP.

Aurora: Otimização Transparente de Aplicações OpenMP

RESUMO

A exploração eficiente do paralelismo no nível de threads tem sido um desafio para os desenvolvedores de *softwares*. Como muitas aplicações não escalam com o número de núcleos, aumentar cegamente o número de threads pode não produzir os melhores resultados em desempenho ou energia. No entanto, a tarefa de escolher corretamente o número ideal de threads não é simples: muitas variáveis estão envolvidas (por exemplo, saturação do barramento off-chip e sobrecarga de sincronização de dados), que mudam de acordo com diferentes aspectos do sistema (por exemplo, conjunto de entrada, micro-arquitetura) e mesmo durante a execução da aplicação.

Para abordar esse complexo cenário, esta tese apresenta Aurora. Ela é capaz de encontrar automaticamente, em tempo de execução e com o mínimo de sobrecarga, o número ideal de threads para cada região paralela da aplicação e se readaptar nos casos em que o comportamento de uma região muda durante a execução. Aurora trabalha com o OpenMP e é completamente transparente tanto para o programador quanto para o usuário final: dado um binário de uma aplicação OpenMP, Aurora o otimiza sem nenhuma transformação ou recompilação de código. Através da execução de quinze benchmarks conhecidos em quatro processadores multi-core, mostramos que Aurora melhora o trade-off entre desempenho e energia em até: 98% sobre a execução padrão do OpenMP; 86% sobre o recurso interno do OpenMP que ajusta dinamicamente o número de threads; e 91% quando comparado a uma emulação do feedback-driven threading.

Palavras-chave: Computação paralela, otimização de desempenho e energia, *software tuning*, OpenMP.

LIST OF ABBREVIATIONS AND ACRONYMS

API Application Programming Interface.

APM Application Power Management.

BNT Best Number of Threads.

BT Block Tri-diagonal solver.

CG Conjugate Gradient.

CMOS Complementary Metal–Oxide–Semiconductor.

DSE Design Space Exploration.

DVFS Dynamic Voltage and Frequency Scaling.

ECC Error Correction Code.

EDP Energy-Delay Product.

FDT Feedback-driven threading.

FFT fast Fourier transform.

FIFO First-In-First-Out.

FSM Finite State Machine.

FT discrete 3D fast Fourier Transform.

FU Functional Unit.

GCC GNU Compiler Collection.

GPP General-Purpose Processor.

HC High Communication.

HPC High-Performance Computing.

HPCG High Performance Conjugate Gradient.

HPL High Performance LINPACK.

HS Hotspot.

ICC Intel C++ Compiler.

ILP Instruction-Level Parallelism.

ISA Instruction Set Architecture.

JA Jacobi.

LC Low Communication.

LU Lower-Upper Gauss-Seidel solver.

MG Multi-Grid on a sequence of meshes.

MPI Message Passing Interface.

NB N-body.

NoC Network-on-Chip.

OpenMP Open Multi-Processing.

PAPI Performance Application Programming Interface.

PARSEC Princeton Application Repository for Shared-Memory Computers.

PCM Performance Counter Monitor.

PO Poisson.

PPI Parallel Programming Interface.

PThreads POSIX Threads.

RAPL Running Average Power Limit.

SC Streamcluster.

SMT Simultaneous Multithreading.

SP Scalar Penta-diagonal solver.

SPEC Standard Performance Evaluation Corporation.

ST STREAM.

TBB Threading Building Block.

TDP Thermal Design Power.

TLP Thread-Level Paralellism.

UA Unstructured Adaptive mesh.

UPC Unified Parallel C.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

With the increasing complexity of parallel applications, which require more computing power, energy consumption has became an important issue. On the one hand, the power consumption of High-Performance Computing (HPC) systems in 2020 will require 200 MW, according to the Advanced Scientific Computing Research from the U.S. Department of Energy [1]. On the other hand, general-purpose processors are being pulled back by the limits of the Thermal Design Power (TDP), while most of the embedded devices are mobile and heavily dependent on battery (e.g., smartphones, tablets, etc.). Therefore, the primary objective when designing and executing parallel applications is not to merely improve performance but to do so with minimal impact on energy consumption.

Performance improvements can be achieved by exploiting Instruction-Level Parallelism (ILP) or Thread-Level Paralellism (TLP). In the former, independent instructions of a single program are simultaneously executed, usually on a superscalar processor, as long as there are functional units available. However, typical instruction streams have only a limited amount of parallelism (WALL, 1991), resulting in considerable efforts to design a micro-architecture that will bring only marginal performance gains with very significant area/power overhead. Even if one considers a perfect processor, ILP exploitation will reach an upper bound (OLUKOTUN; HAMMOND, 2005).

Hence, to continue increasing performance and to provide a better use of the extra available transistors, modern designs have started to exploit TLP more aggressively. In such case, multiple processors simultaneously execute parts of the same program, exchanging data at runtime through shared variables or message passing. In the former, all threads share the same memory region, while in the latter, each process has its private memory, and the communication occurs by send/receive primitives (even though they are also implemented using a shared memory context when the data exchange is done intra-chip (CHANDRAMOWLISHWARAN; KNOBE; VUDUC, 2010)). Therefore, regardless of the processor or communication model, data exchange is usually done through memory regions that are more distant from the processor (e.g., L3 cache and main memory) and have higher delay and power consumption when compared to memories that are closer to it (e.g., register, L1, and L2 caches).

Even though execution time shall decrease because of TLP exploitation, energy will not necessarily follow the same trend, since many other variables are involved:

[1] http://science.energy.gov/ascr/research/scidac/exascale-challenges/

- Memories that are more distant from the processor will be more accessed for synchronization and data exchange, increasing energy related to dynamic power (which increases as there is more activity in the circuitry).

- An application that was parallelized will usually execute more instructions than its sequential counterpart. Moreover, even considering an ideal scenario (where processors are put on standby with no power consumption), the sum of the execution times of all threads executing on all cores tends to be be larger than if the application was sequentially executed on only one core. In consequence, the resulting energy from static power (directly proportional to how long each hardware component is turned on) consumed by the cores will also be larger. There are few exceptions to this rule, such as non-deterministic algorithms, where an execution of a parallel application may execute fewer instructions than its sequential counterpart.

- The memory system (which involves caches and main memory) will be turned on for a shorter time (the total execution time of the applications), which will decrease the energy resultant from the static power.

Therefore, cores tend to consume more energy from both dynamic and static power; while memories will usually spend more dynamic power (and hence energy), but also tends to save static power, which is very significant (VOGELSANG, 2010). On top of that, neither performance nor energy improvements resultant from TLP exploitation are linear, and sometimes they do not scale as the number of threads increases, which means that in many cases the maximum number of threads will not offer the best results. There are several reasons for this lack of scalability: instruction issue-width saturation, which affects the performance of CPU-Bound applications that are running on Simultaneous Multithreading (SMT) architectures; off-chip bus saturation, which limits the performance of applications whose threads operate on large amounts of data; data synchronization, which limits the scalability of applications that have many synchronization points; and concurrent shared memory accesses, which affects the applications with high communication demands among the threads (LEVY et al., 1996; RAASCH; REINHARDT, 2003; SULEMAN; QURESHI; PATT, 2008; JOAO et al., 2012; SUBRAMANIAN et al., 2013). They all will be discussed in more details later in this thesis.

Considering the aforementioned scenario, choosing the right number of threads to a given application will offer opportunities to improve performance and increase the energy efficiency. However, such task is extremely difficult: besides the huge number of

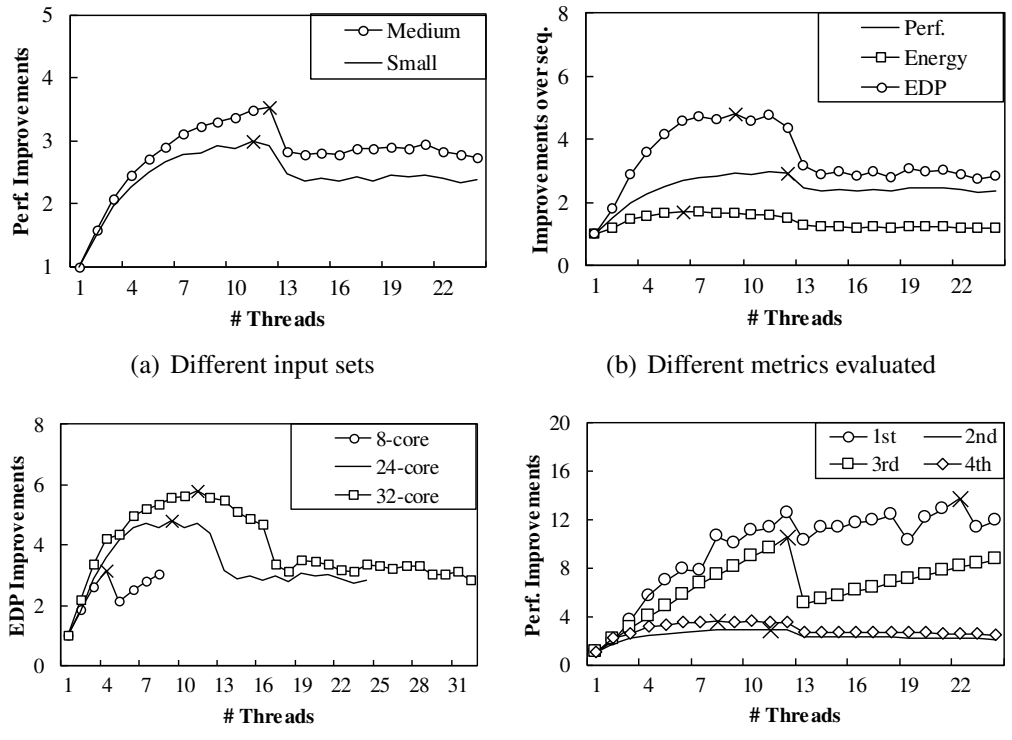
variables involved, many of them will change according to different aspects of the system at hand and are only possible to be defined at runtime, such as:

- Input set: As shown in Figure 1.1(a), different levels of performance improvements for the LULESH benchmark (KARLIN; KEASLER; NEELY, 2013) (also used as examples in the next two items) over its single-threaded version are reached with a different number of threads (x-axis). However, these levels vary according to the input set (small or medium). While the best number of threads is 12 for the medium input set, the ideal number for the small set is 11.

- Metric evaluated: As Figure 1.1(b) shows, the best performance is reached with 12 threads, while 6 threads bring the lowest energy consumption, and 9 presents the best trade-off between both metrics (represented by the Energy-Delay Product (EDP)).

- Processor architecture: Figure 1.1(c) shows that the best EDP improvements of the parallel application on a 32-core system are when it executes with 11 threads. However, the best choice for a 24-core system is 9 threads.

Figure 1.1: Appropriate number of threads (x-axis) considering the improvements over sequential version (y-axis)



(a) Different input sets

(b) Different metrics evaluated

(c) Different multicore processors

(d) Different parallel regions

Source: The Author

- Parallel regions: many applications are divided into several parallel regions, in which each of these regions may have a distinct ideal number of threads, since their behavior may vary as the application executes. As an example, Figure 1.1(d) shows the behavior of four parallel regions from the Poisson Equation benchmark (QUINN, 2004) when running on a 24-core system. One can note that each parallel region is better executed with a different number of threads.

1.1 Contributions

Considering the scenario discussed in the previous section, this thesis makes the following contributions:

- Conduct a comprehensive study of the opportunities for parallel computing regarding the most popular Parallel Programming Interfaces (PPIs) (POSIX Threads (PThreads), Open Multi-Processing (OpenMP), and Message Passing Interface (MPI)) and platforms of embedded and general-purpose processors. In such study, comparisons regarding performance, energy consumption, EDP, and the influence of the static power of the processor on the total energy consumption are discussed.

- Develop a library to automatically adapt the number of threads for OpenMP applications. Such library has the goal to improve the execution of applications implemented with OpenMP regarding different metrics, such as performance, energy consumption, and EDP.

- Through mathematical correlation, present the bottlenecks that affect the scalability of OpenMP applications. That is, discuss the causes related to hardware and software that explain why selecting the maximum number of available threads will not necessarily lead to the best possible result.

- Incorporate the developed library into the OpenMP Application Programming Interface (API) library (*libgomp*) to provide an approach that improves OpenMP applications with no modifications in the source code nor code recompilation.

1.2 Organization of this thesis

This thesis is organized as follows:

Chapter 2 presents the fundamental concepts used in this work. First, an overview about parallel computing and multicore architectures is presented. Second, a study regarding the issues that affect the scalability of parallel applications is described. Then, the design space exploration regarding the estimation and adaptation of the behavior of a parallel execution is discussed. Finally, the metrics of interest that will be used to compare and evaluate our mechanisms are discussed.

Chapter 3 discusses the related work. It is divided into three subsections: First, the studies that explore the opportunities of parallelism exploitation on embedded and general-purpose processors are discussed. In addition, this section discusses the studies that compare the parallel programming interfaces used in this thesis. Second, the studies that performs the optimization of parallel applications regarding the adaptation of the execution environment are discussed. Finally, the contributions of this thesis are compared to the related work.

In Chapter 4, the study regarding the parallelism exploitation of different communication models and platforms (embedded and General-Purpose Processors (GPPs)) is presented. First, the comparison of different PPIs considering different evaluation metrics, such as performance, energy consumption, and energy-delay product, is presented. Also, this section discusses the influence of the static power of the processor in the total energy consumption. Finally, we present the importance of proposing a mechanism for the OpenMP parallel programming interface.

Chapter 5 presents the main contributions of this thesis: the mechanisms to automatically and transparently improve OpenMP applications. The two proposed approaches are presented: first, the library to automatically improve OpenMP applications, with code transformation and recompilation; and second, the transparent mechanism that optimize OpenMP applications without user influence. Then, the two mechanisms are evaluated and compared to different well-know techniques from the related work.

Chapter 6 concludes the conducted work in this thesis. It also discusses some of the most promising future works envisioned at this time.

2 FUNDAMENTAL CONCEPTS

This chapter presents the fundamental concepts for understanding the remaining of the work. First, the concepts of parallel programming, as well as the communication models and parallel programming interfaces used in this work are presented. Also, this section highlights the main characteristics of the multicore architectures regarding the parallelism exploitation. Second, the issues that affect the scalability of parallel applications are discussed. Then, the techniques used to address the design and space exploration of parallel computing are presented. Finally, the metrics that will be used for comparison and validation of our methodology are presented.

2.1 Parallel Computing in Software

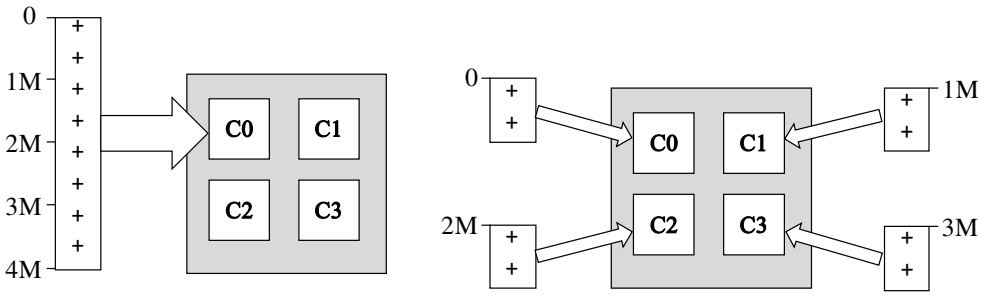
Parallel programming is defined as the division of tasks of an application that can be executed concurrently aiming to reduce their total execution time (RAUBER; RüNGER, 2013). It has been widely used in the development of scientific applications that require a large computing power, such as weather forecasting calculations, DNA sequences and genome calculation. In addition, with the popularization of multicore architectures, the general-purpose applications (graphic editors, web servers, etc.) have also taken advantage of parallel programming.

The main goal of parallel computing is to use multiple processing units for solving problems in less time (FOSTER, 1995). The key for parallel computing is the possibility to exploit concurrency on a given application by decomposing a problem into subproblems that can be executed at the same time. As a simple example, suppose that part of an application involves computing the summation of a large set of values. In a sequential execution, all the values are added together in only one core, sequentially, as depicted in Figure 2.1(a). On the other hand, with the parallel computing, the data set can be partitioned, and the summations computed simultaneously, each on a different processor (C0, C1, C2, and C3, in Figure 2.1(b)). Then, the partial sums are combined to get the final answer.

2.1.1 Communication Models

Parallel computing exploits the use of multiple processing units to execute parts of the same program simultaneously. Thus, there is cooperation between the processors that

Figure 2.1: Example of parallel computing



(a) Sequential execution (b) Parallel execution in four cores

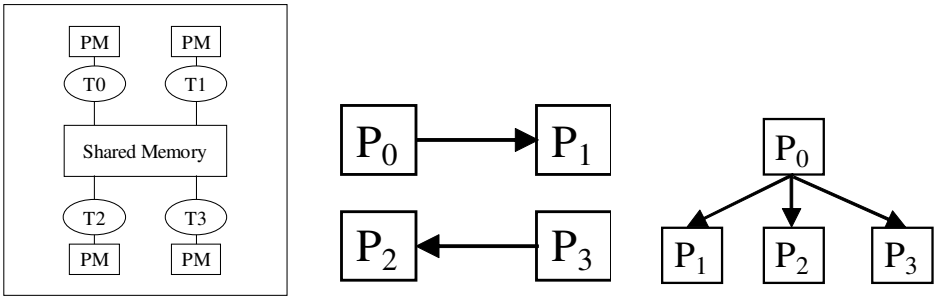
Source: The Author

execute concurrently. However, for this cooperation to occur, processors should exchange information at runtime. In multicore processors, this can be done through shared variables or message passing (RAUBER; RüNGER, 2013):

Shared variable is based on the existence of an address space in the memory that can be accessed by all processors. It is widely used when parallelism is exploited at the level of the thread since they share the same memory address space. In this model (Figure 2.6), the threads can have private variables (the thread has exclusive access) and shared variables (all the threads have access). When the threads need to exchange information between them, they use shared variables located in memory regions that are accessed by all threads (shared memory). Each parallel programming interface provides synchronization operations to control the access to shared variables, avoiding race conditions.

Message Passing is used in environments where memory space is distributed and/or where processes do not share the same memory address space. Therefore, communication occurs using send/receive operations which can be point-to-point or collective ones. In the first (Figure 2.2(b)), data exchange is done between pairs of processes. In the latter, more than two processes are communicating (Figure 2.2(c)).

Figure 2.2: Communication Models



(a) Shared Variable (b) Message Passing: P2P (c) Message Passing: Collective

Source: The Author

2.1.2 Parallel Programming Interfaces

The development of applications that can exploit the full potential parallelism of multiprocessor architectures depends on many specific aspects of their organization, including the size, structure and hierarchy of the memory. Operating Systems provide transparency concerning the allocation and scheduling of different processes across the various cores. However, when it comes to TLP exploitation, which involves the division of the application into threads or processes, the responsibility is of the programmer. Therefore, Parallel Programming Interface (PPI)s make the extraction of the parallelism easier, fast, and less error prone. Several parallel programming interfaces are used nowadays, in which the most common are OpenMP, PThreads, MPI, Threading Building Block (TBB), Cilk plus, Charm, among others.

OpenMP is a PPI for shared memory in C/C++ and FORTRAN that consists of a set of compiler directives, library functions, and environment variables (CHAPMAN; JOST; PAS, 2007). Parallelism is exploited through the insertion of directives in the sequential code that inform the compiler how and which parts of the code should be executed in parallel. The synchronization can be implicit (implied barrier at the end of a parallel region), or explicit (synchronization constructs) to the programmer. By default, whenever there is a synchronization point, OpenMP threads enter in a hybrid state (Spin-lock and Sleep), i.e., they access the shared memory repeatedly until the number of spins of the busy-wait loop is achieved (Spin-lock); and then, they enter into a sleep state until the end of synchronization (CHAPMAN; JOST; PAS, 2007). The amount of time that each thread waits actively before waiting passively without consuming CPU power may vary according to the wait policy that gives the number of spins of the busy-wait loop (e.g., the standard value when *omp wait policy* is set to being active is 30 billion iterations) (OPENMP, 2013).

PThreads is a standard PPI for C/C++, where functions allow fine adjustment in the grain size of the workload. Thus, the creation/termination of the threads, the workload distribution and the control of execution are defined by the programmer (BUTENHOF, 1997). PThreads synchronization is done by blocking threads with mutexes, which are inserted in the code by the programmer. In this case, threads lose the processor and wait on standby until the end of the synchronization, when they are rescheduled for execution (TANENBAUM, 2007).

Cilk Plus is integrated with a C/C++ compiler and extends the language with the addition of keywords by the programmer indicating where parallelism is allowed. Cilk

Plus enables programmers to concentrate on structuring programs to expose parallelism and exploit locality. Thus, runtime system has the responsibility of scheduling the computation to run efficiently on a given platform. In addition, it takes care of details like load balancing, synchronization, and communication protocols (MCCOOL; REINDERS; ROBISON, 2012).

TBB is a library that supports parallelism based on a tasking model and can be used with any compiler ISO C++. TBB requires the use of function objects to specify blocks of code to run in parallel, which relies on templates and generic programming. The synchronization between threads is done by mutual exclusion, in which the threads in this state perform busy-waiting until the end of synchronization (MCCOOL; REINDERS; ROBISON, 2012).

MPI is a standard message passing library for C/C++ and FORTRAN that implements optimization to provide communication in shared memory environments (GROPP; LUSK; SKJELLUM, 1999). MPI is like PThreads regarding the explicit exploitation of parallelism. Currently, it is divided into three norms. In MPI-1, all processes are created at the beginning of the execution and the number of processes does not change throughout program execution. In MPI-2, processes are created at runtime, and the number of processes can change during the execution. In MPI-3, the updates include the extension of collective operations to include nonblocking versions and extensions to the one-sided operations. Communication between MPI processes occurs through send/receive operations (point-to-point or collective ones), which are likewise explicitly handled by the programmers. When MPI programs are executed on shared memory architectures, message transmissions can be done as shared memory accesses, in which messages are broken into fragments that are pushed and popped in First-In-First-Out (FIFO) queues of each MPI process (CHANDRAMOWLISHWARAN; KNOBE; VUDUC, 2010)(BUONO et al., 2014).

2.1.3 Multicore Architectures

Multicore architectures have multiple processing units (cores) and a memory system that enables communication between the cores. Each core is an independent logical processor with its resources, such as functional units, pipeline execution, registers, among others. The memory system consists of private memories that are closer to the processor and only accessible by a single processor; and shared memories, that are more distant

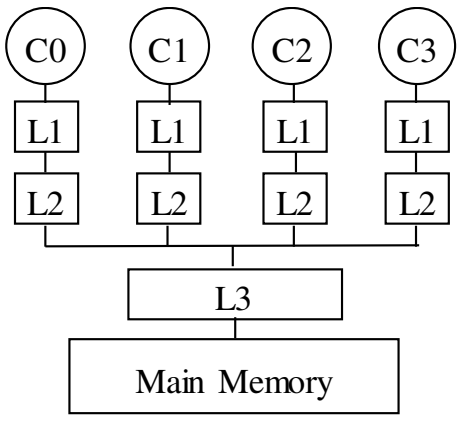
from the processor and can be accessed by multiple processors (HENNESSY; PATTERSON, 2003). Figure 2.3 shows an example of a multicore architecture with four cores (C0, C1, C2, and C3) and its private (L1 and L2 caches) and shared memories (L3 cache and main memory).

Multicore processors can exploit TLP. In this case, multiple processors simultaneously execute parts of the same program, exchanging data at runtime through shared variables or message passing (as discussed in Section 2.1). Regardless of the processor or communication model, data exchange is done through load/store instructions in shared memory regions. As Figure 2.3 shows, these regions are more distant from the processor (e.g., L3 cache and main memory), and have a higher delay and power consumption when compared to memories that are closer to it (e.g., register, L1, and L2 caches) (KORTHIKANTI; AGHA, 2010).

Among the challenges faced in the design of multicore architectures, one of the most important is related to the data access on parallel applications. When a private data is accessed, its location is migrated to the private cache of a core, since no other processor will use the same variable. On the other hand, a shared data is replicated in multiple caches, since other processors can access it to communicate. Therefore, while sharing data improves concurrency between multiple processors, it also introduces the cache coherence problem: when a processor writes on any shared data, the information stored in other caches become invalid. To solve this problem, cache coherence protocols are used.

Cache coherence protocols are classified into two classes: directory based and snooping (PATTERSON; HENNESSY, 2013). In the former, a centralized directory maintains the state of each block in different caches. When an entry will be modified,

Figure 2.3: Basic structure of a multicore architecture with four cores



Source: The Author

the directory is responsible for either updating or invalidating the other caches with that entry. In the snooping protocol, rather than keeping the state of sharing block in a single directory, each cache that has a copy of the data can track the sharing status of the block. Thus, all the processors observe memory operations and take proper action to update or invalidate the local cache content if needed.

Cache blocks are classified into states, which the number of states depends on the protocol. For instance, directory based and snooping protocols are simple three-state protocols in which each block is classified into modified, shared, and invalid (they are often called as MSI - modified, shared, and invalid - protocol). When a cache block is in the modified state, it has been updated in the private cache, and cannot be in any other cache. The shared state indicates that the block in the private cache is potentially shared, and the cache block is invalid when a block contains no valid data. Based on the MSI protocol, extensions have been created by adding additional states. There are two common extensions: MESI, which adds the state "*exclusive*" to the MSI to indicate when a cache block is resident only in a single cache but is clean; and MOESI, which adds the "*state-owned*" to the MESI protocol to indicate that the associated block is owned by a particular cache and out-of-date in memory (HENNESSY; PATTERSON, 2003).

When developing parallel applications, the software developer does not need to know about all details of cache coherence. However, knowing how the data exchange is performed at the hardware level can help the programmer to make better decisions during the development of parallel applications.

2.2 Scalability of Parallel Applications

Many works have associated the fact that selecting the maximum number of available threads (the common choice for most software developers (LEE et al., 2010)) will not necessarily lead to the best possible performance. The causes are related to hardware or software: saturation of functional units in SMT processors (LEVY et al., 1996; RAASCH; REINHARDT, 2003), off-chip bus saturation (SULEMAN; QURESHI; PATT, 2008; SUBRAMANIAN et al., 2013), overhead of data synchronization among threads (SULEMAN; QURESHI; PATT, 2008; LEE et al., 2010; JOAO et al., 2012), and number of shared memory accesses (SUBRAMANIAN et al., 2013).

To measure (through correlation) their real influence, we have executed 4 benchmarks from our set (and used them as examples for the next subsections) on a 12-core

Table 2.1: Pearson correlation between the scalability issues and each benchmark

		Hotspot	**FFT**	**MG**	**N-body**
Small Input	***Issue-width saturation***	**-0.91**	-0.71	-0.81	-0.82
	Off-chip bus saturation	-0.51	**-0.98**	-0.76	0.46
	Shared memory accesses	-0.52	-0.43	**-0.90**	0.80
	Data-Synchronization	-0.54	-0.50	-0.59	**0.97**
Medium Input	***Issue-width saturation***	**-0.92**	-0.71	-0.79	-0.78
	Off-chip bus saturation	-0.52	**-0.97**	-0.88	0.39
	Shared memory accesses	-0.54	-0.75	**-0.96**	0.81
	Data-Synchronization	-0.64	-0.53	-0.78	**0.96**

Source: The Author

machine with SMT support. Each one of them has one limiting characteristic that stands out (i.e. it is the main reason for the application's lack of scalability), as shown in Table 2.1. The benchmark Hotspot (HS) saturates the issue-width; fast Fourier transform (FFT), the off-chip bus; MG, the shared memory accesses; and N-body (NB) saturates data-synchronization. To analyze each of the scalability issues, we considered the Pearson Correlation (BENESTY et al., 2009). It takes a range of values from +1 to -1: the stronger the "r" linear association between two variables, the closer the value will be to either +1 or -1 ($r \geq 0.9$ or $r \leq -0.9$ mean a very strong correlation), depending on whether the association is directly proportional or inversely proportional.

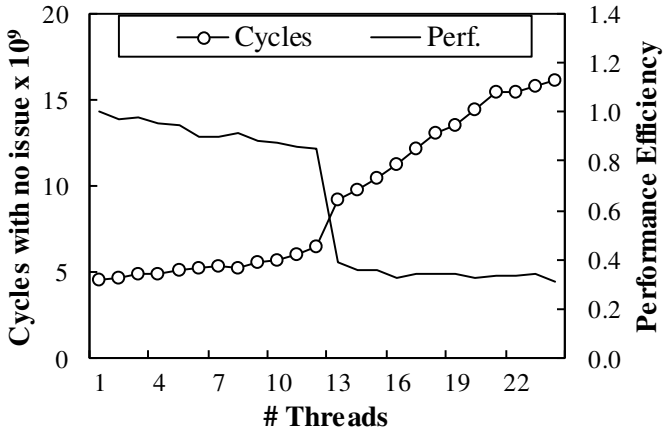
We briefly discuss these bottlenecks next and revisit them in Section 5.2 to show how the approach proposed in this thesis deals with them.

2.2.1 Issue-width Saturation

By allowing many threads to run simultaneously on a core, the probability of having more independent instructions, and thus filling the Functional Units (FUs), greatly increases. Nonetheless, although SMT can potentially maximize the functional unit usage and increase the performance of applications with low ILP, it can lead to the opposite behavior if an individual thread presents enough ILP to issue instructions to all or most of the core's FUs. Then, mapping an additional thread to the same core may lead to resource conflicts and functional unit contention, degrading performance.

Figure 2.4 shows the performance improvements and the number of idle cycles (i.e. cycles without instruction issued) for the Hotspot application. When increasing the number of threads from 12 to 13, two threads will be mapped to the same physical core,

Figure 2.4: Scalability Behavior: Issue-width saturation



Source: The Author

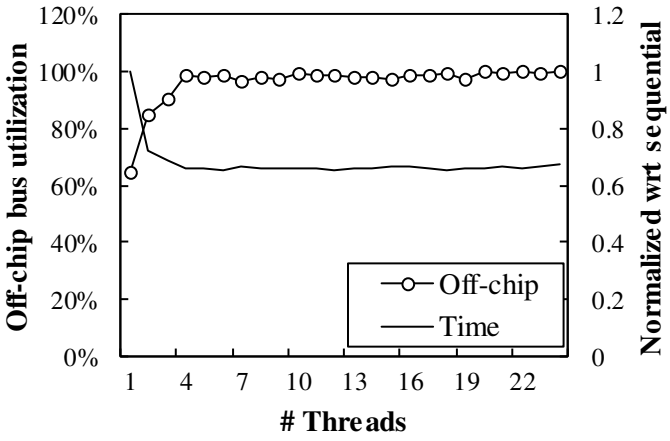
so SMT usage starts. Since the application presents high ILP, performance degrades for both two threads on that core (the number of idle cycles increases), becoming the new critical path of that parallel region, as both threads will delay the execution of the entire region. As a result, overall performance decreases.

2.2.2 Off-chip Bus Saturation

Many parallel applications operate on huge amounts of data that are private to each thread and have to be constantly fetched from the main memory. In this scenario, the off-chip bus that connects memory and processor plays a decisive role in thread scalability: as each thread computes on different data blocks, the demand for off-chip bus increases with the number of threads. However, the bus bandwidth is limited by the number of I/O pins, which does not increase according to the number of cores (HAM et al., 2013). Therefore, when the off-chip bus saturates, no further improvements are achieved by increasing the number of threads (SULEMAN; QURESHI; PATT, 2008).

In the FFT execution, as the number of threads increases, execution time and energy consumption reduce until the off-chip bus becomes completely saturated (100% of utilization), as shown in Figure 2.5. In this example, from this point on (4 threads), increasing the number of threads does not provide performance improvements, as the bus cannot deliver all the requested data. There might be an increase in energy consumption as well since many hardware components will stay active while the cores are not being properly fed with data.

Figure 2.5: Scalability Behavior: Off-chip saturation



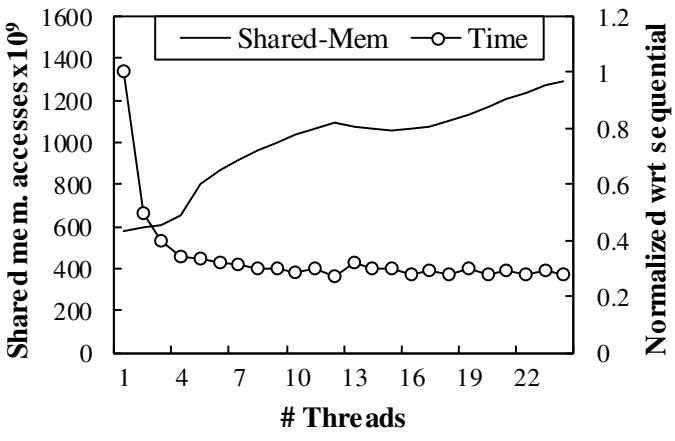
Source: The Author

2.2.3 Shared Memory Accesses

Threads communicate by accessing data that are located in shared memory regions, which are usually more distant from the processor (e.g., L3 cache and main memory) and have a higher delay and power consumption when compared to memories that are closer to the processor (e.g., registers, L1, and L2 caches). Therefore, the number of shared memory accesses may also become a bottleneck.

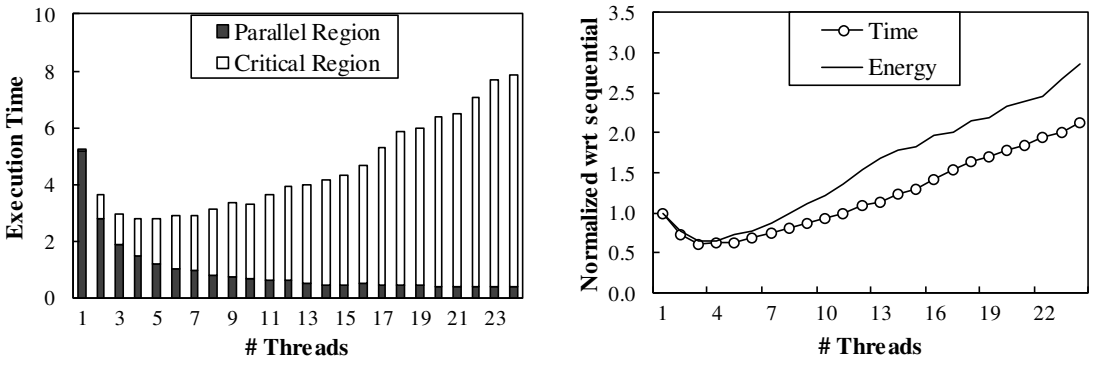
Figure 2.6 presents this behavior for the MG benchmark. The number of accesses to the L3 cache (the only cache level shared among the cores in the target processor) is shown in the primary y-axis, while the secondary y-axis shows the execution time normalized with respect to the sequential execution. As one can note, when the application

Figure 2.6: Scalability Behavior: Shared memory accesses



Source: The Author

Figure 2.7: Scalability Behavior: Data-synchronization



(a) Critical section behavior (b) Perf./Energy degradation

Source: The Author

executes with more than four threads, the performance is highly influenced by the increased number of L3 cache accesses.

2.2.4 Data-Synchronization

Synchronization operations are used to ensure data integrity during the execution of a parallel application. In this case, critical sections are implemented to guarantee that only one thread will execute a given region of code at once, and therefore data will correctly synchronize. In this way, all code inside a critical section must be executed sequentially. Therefore, when the number of threads increases, more threads must be serialized inside the critical sections, also increasing the synchronization time (Figure 2.7(a)) and potentially affecting the execution time and energy consumption of the whole application.

Figure 2.7(b) shows this behavior for the n-body benchmark: while it executes with 4 threads or less, the performance gains within the parallel region reduces the execution time and energy consumption, even if the time spent in the critical region increases (Figure 2.7(a)). However, from this point on, the time the threads spend synchronizing overcomes the speedup achieved in the parallel region.

2.3 Design Space Exploration

The Design Space Exploration (DSE) is used to tune the configurable parameters, and it generally consists of a multi-objective optimization problem. The DSE problem consists of exploring a large design space consisting of several parameters that must be

Figure 2.8: Design space exploration process

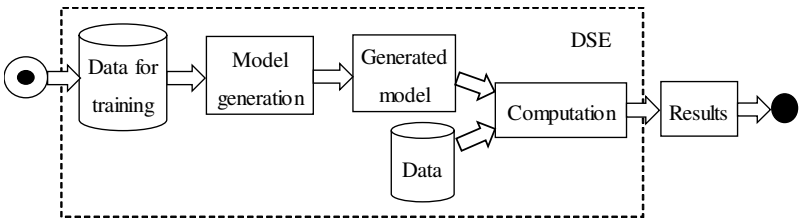
Source: The Author

tuned to find the best trade-offs in terms of the selected figures of merit, also called as metrics, such as energy, performance, EDP, etc. (PALERMO; SILVANO; ZACCARIA, 2008). Figure 2.8 shows a schematic illustration of the DSE process. Firstly, the input data that should be evaluated are provided to the DSE. These values are calculated through any classification model, such as neural network, linear regression, mathematical model, among other models. Then, the output of the DSE phase are the results containing the best trade-offs between the values and metrics.

The DSE phase can be performed in different moments of the application execution. In this work, we consider that it can be performed using offline or online information provided by the processor architecture; and with or with no adaptation of the parallel application at runtime:

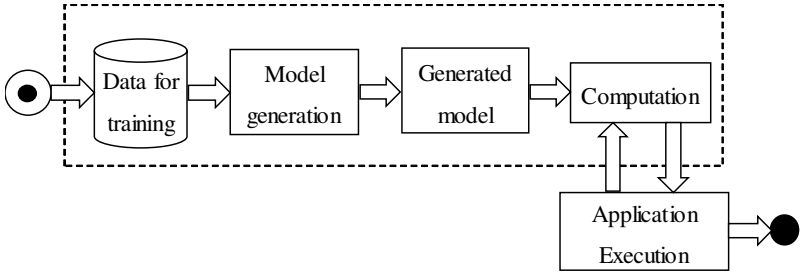
Offline information with no runtime decision and adaptation: In this approach, the DSE is fully realized before the application execution. It comprises prediction models which use a variety of statistical models to analyze current and historical values of the target architecture and applications to make predictions about the future. However, the data obtained by the prediction are used only to decide the best configuration to run an application. Therefore, there is no decision-making and adaptation of the parallel application at runtime. A predictive model consists of four basic steps, as Figure 2.9 shows. In the first step, data from the architecture and applications are collected to generate the model. Then, a statistical model is formulated by applying some method (i.e., linear regression or a neural network) over the collected data. After, predictions are made for the new input data. In the last step, additional data are used to validate the model.

Figure 2.9: Offline information with no runtime decision and adaptation



Source: The Author

Figure 2.10: Offline information with runtime decision and adaptation



Source: The Author

Offline information with runtime decision and adaptation: This approach differs from the last one in the sense that the information obtained from the prediction model are used to take decisions and adapt the parallel application at runtime. Figure 2.10 illustrates this approach, which works as follows: (i) the model is created using static information provided from the architecture and applications; (ii) during the application execution, data related to its behavior are extracted; (iii) then, these data are used as input to the previously model created in (i); (iv) finally, the result of this computation is used to adapt the execution environment. The main limitation of this approach is the need of rerun the statistical model whenever there are changes in the execution environment, such as application characteristics, microarchitecture, and metric evaluated.

Online information with no runtime decision and adaptation: The approaches of this class are those that consider the current behavior of the microarchitecture processor when the application is being compiled. Thus, the model verifies characteristics of the microarchitecture and the application to help the compiler for generating optimized code for such environment. However, there is no adaptation of the parallel application at runtime.

Online information with runtime decision and adaptation: Differently from the last approach, in this class, the models consider information obtained at runtime to make decisions and adjust the application execution. In this case, different characteristics of the application that are only known at runtime, such as the length of the input are considered. Moreover, the adaptation using dynamic information is essential for applications with variable behavior, in which the workload changes constantly; and with many parallel regions, in which each of them has different behavior.

2.4 Metrics of Interest

2.4.1 Performance

Computer performance is defined as the total time required for the computer to complete a task, including disk and memory accesses, operating system overhead, CPU execution, and so on. It is measured in different ways, such as execution time, elapsed time, response time, wall clock time, or throughput (i.e., the number of tasks completed per unit time) (HENNESSY; PATTERSON, 2003).

The total execution time of an application is classified according to the resource usage: CPU, user, and system time. The first is the total amount of time that the CPU was used for processing instructions of a program or operating system. The second is the CPU time spent in the program itself. The latter is related to the CPU time spent in the operating system performing tasks on behalf of the program.

In parallel computing, another metric called speedup ratio is used to compare different applications and computer systems. It is defined in the Equation 2.1, and shows how much a parallel application is faster than its sequential counterpart. Considering only the performance, the greater the speedup, the better is the parallel implementation.

$$Speedup = \frac{Seq_{time}}{Par_{time}} \tag{2.1}$$

2.4.2 Power and Energy Consumption

Two main components constitute the power used by a Complementary Metal–Oxide–Semiconductor (CMOS) integrated circuit: dynamic and static (KAXIRAS; MARTONOSI, 2008). The first is the power consumed while the inputs are active, with capacitance charging and discharging, being directly proportional to the circuit switching activity, given by Equation 2.2.

$$P_{dynamic} = CV^2Af \tag{2.2}$$

Capacitance (C) depends on the wire lengths of on-chips structures. The designers in several ways can influence this metric. For instance, building two smaller cores on-chip, rather than one large, is likely to reduce average wire lengths, since most wires will interconnect units within a single core.

Supply Voltage (*V* or *Vdd*) is the main voltage to power the integrated circuit. Because of its direct quadratic influence on dynamic power, supply voltage has a higher importance on power-aware design.

Activity factor (*A*) refers to the ratio between the number of switching gates in a clock period over the total number of gates (OKLOBDZIJA; KRISHNAMURTHY, 2006).

Clock frequency (*f*) has a fundamental impact on power dissipation because it indirectly influences supply voltage: higher clock frequencies will require a higher supply voltage. Thus, the combined portion of supply voltage and clock frequency in the dynamic power equation has a cubic impact on total power dissipation.

While dynamic power dissipation represents the predominant factor in CMOS power consumption, static power has been increasingly prominent in recent technologies (KAXIRAS; MARTONOSI, 2008). The static power essentially consists of the power used when the hardware component is turned on and is determined by Equation 2.3, where the supply voltage is *V*; and the total current flowing through the device is I_{static}.

$$P_{static} = I_{static} x V \tag{2.3}$$

Energy, in joules, is the integral of total power consumed (P) over the time (T), given by Equation 2.4.

$$Energy = \int_0^T P_i \tag{2.4}$$

Nowadays, the energy consumption of a computing system can be estimated or measured. The former makes use of models to estimate the energy by using hardware- or OS-provided metrics. There are two essential steps to carry out this task: the selection of the model input parameters (e.g., CPU load, memory usage, and disk utilization) and the identification of a tool to train and test it. Although estimation models have several features, they can be inaccurate, and their scope is determined by the underlying system architecture (BODE, 2013; LI et al., 2013).

The energy measurement of a computing system can be done by external or internal tools. In the first, an external power meter is used to measure the energy consumption of anything plugged into it. The most commons external measurement devices are Watt's Up Pro Power Metter[1] and the PowerMon 2 (BEDARD et al., 2010). On the other hand,

[1] Available online at: https://www.wattsupmeters.com/secure/index.php

to get energy consumption through internal tools, the measurements require direct low-level hardware reads and a special library to perform these reads. The special library varies according to the processor architecture. For Intel processors, the Running Average Power Limit (RAPL) interface provides a set of hardware counters to get energy and power consumption, available since the SandyBridge microarchitecture (ROTEM et al., 2012). As for AMD processors, the energy measurements occurs via Application Power Management (APM), introduced in the Buldozer family (HACKENBERG et al., 2013a). As the authors show in (HäHNEL et al., 2012), (HACKENBERG et al., 2013b), and (VENKATESH; KANDALLA; PANDA, 2013), RAPL from Intel and APM from AMD are widely used to provide energy consumption with good accuracy.

2.4.3 Energy-Delay Product

When performance is being evaluated, the designer aims to reduce the execution time without any concerns on energy consumption. The same is true when the goal is to reduce energy consumption. However, according to their niche, companies of general-purpose processors may give more importance to performance, while the embedded ones to energy. In this case, the energy-delay product metric proposed by Gonzalez and Horowitz (1996) may be useful since it correlates performance (delay) and energy into a unique value, as shown in Equation 2.5. EDP offers equal weight to either energy or performance degradation. If either energy or delay increase, the EDP will increase. Thus, lower EDP values are desirable.

$$EDP = EnergyxTime \tag{2.5}$$

This metric is widely used to evaluate different environments, such as in (BLEM; MENON; SANKARALINGAM, 2013) and (TIWARI et al., 2015), since it allows, in a unique value, to analyze the relationship between energy and performance. For example, let us consider two scenarios: (*i*) an application spends 10 Joules of energy and executes in 100 seconds; (*ii*) an application executes in 50 seconds, but spends 40 Joules of energy. The scenario (*i*) has EDP of 1000 while the second one has EDP of 2000. This shows that, although the first scenario *i* is twice slower than scenario *ii*, it has the best EDP.

Considering the original EDP proposal, the authors (BLEM; MENON; SANKARALINGAM, 2013) have suggested an alternative metric by adding an exponent x on delay ($EDP = Energy{*}Delay^{x}$). In this way, it is possible to change the weight of delay

(performance) towards energy, which would reflect the importance given to performance considering the application field.

2.4.4 Resource Efficiency

Resource efficiency is here defined as the best possible use of the processors, considering the sequential execution of the application on a single core as a baseline. For example, a program that is split into four threads/processes may be faster and less efficient than one split into only two threads/processes. In this case, each of the four processors was used less (e.g., they spent more time waiting for synchronization) during program execution than each of the two processors. This can be extrapolated to energy or EDP.

The resource efficiency is given by Equation 2.6, where R_{Seq} corresponds to the resource usage of the processors when executing the sequential version of the application. R_{Par} is the same as the previous, but for each thread of the parallel version, and *NT* is the number of the threads/processes that are executing the application.

$$R_{ef} = \frac{R_{Seq}}{\sum_{i=0}^{NT} R_{Par}} \tag{2.6}$$

3 RELATED WORK

In this Chapter, the related work comprising this thesis will be discussed. As already mentioned in Chapter 1, the main contribution of this thesis is an automatic and transparent approach for improving OpenMP applications regarding different metrics. For that, two main steps were followed:

1. A comprehensive study of the opportunities for parallel computing regarding the parallel programming interfaces widely used nowadays. Section 3.1 presents the related work regarding this study.

2. A study of the approaches used to improve the performance, energy, or EDP of parallel applications. Thus, the related work regarding such approaches is discussed in Section 3.2.

Finally, the approach proposed in this thesis is compared to the related work in Section 3.2.3, highlighting its main contributions.

3.1 Possibility of Parallel Computing Exploitation

Here, the most representative works that evaluate performance and/or energy consumption of embedded and general-purpose processors are discussed. They are listed in chronological order.

3.1.1 Parallel Computing on Embedded and General-Purpose Processors

3.1.1.1 Performance Evaluation

A few number of works have evaluated the performance of parallel computing in embedded and general-purpose systems. A comparison between a single-core and a dual-core AMD Opteron processor is presented in (PASE; ECKL, 2005). The authors measure the performance of both processors with the High Performance LINPACK (HPL) benchmark and show that the dual-core processor is up to 60% faster than the single-core. They also evaluated the memory latency and throughput, showing that the dual-core has a throughput of 10% greater than single-core processor.

The work developed in (HANAWA et al., 2009) compares the performance of three Symmetric Multiprocessing cores for embedded systems with a General-Purpose Processor (GPP). Different sequential and OpenMP applications from MiBench suite, MediaBench, and NAS Parallel Benchmarks were used. The results show that embedded processors have larger synchronization cost and slower memory performance than GPPs to improve the synchronization performance.

The authors in (OU et al., 2012) analyzed both energy- and cost-efficiency of clusters based on ARM and x86 workstations. The results show that the performance per Watt ratio of the ARM cluster against the Intel workstation varies according to the benchmark. As an example, for database operations (update, insert, delete, and full table scan), this ratio is up to 9.5 times while it is only 1.21 times in video transcoding application. The authors concluded that, although more ARM processors are necessary to provide similar performance as an Intel workstation, ARM-based data centers are advantageous in computationally lightweight applications.

The work developed in (FASIKU et al., 2014) presents a performance comparison between an AMD and Intel dual-core processors. The Standard Performance Evaluation Corporation (SPEC) CPU2006 benchmark suite was used to measure the performance of both the processors. Experimental results show that the Intel dual-core is about 6.62% faster than the AMD processor. Moreover, the authors have compared the throughput of both processors, in which it was 1.06 times higher in the Intel processor. According to the authors, Intel was better because it has faster core-to-core communication, dynamic cache sharing between cores and smaller size of L2 cache when compared to the AMD processor.

3.1.1.2 Energy Consumption Evaluation

Considering only sequential benchmarks running on a single processor, the authors in (BLEM; MENON; SANKARALINGAM, 2013) compared the impact of different microarchitectures and Instruction Set Architecture (ISA)s on performance, power, energy, and EDP for both general-purpose and embedded systems. A significant number of experiments were performed, evaluating the number of executed instructions and processor cycles, average instruction length, the amount of memory accesses, and execution time. The authors demonstrated that there is a significant difference between processors implementing ISAs: while the Cortex-A9 is more energy-efficiency, the Intel Core i7 is

the most efficient when it comes to execution time. These results, however, are mainly dependent on the microarchitecture than the ISA itself.

The authors in (STANISIC et al., 2013) present a performance and energy comparison between x86 systems and embedded platforms. They show that an HPC system using ARM Dual Cortex-A9 processors require less energy than the one composed of Intel Xeon X5550 processor to run the LINPACK, CoreMark, and StockFish benchmarks. In the same way, the authors in (RAJOVIC et al., 2013) show that although performance and energy consumption suggest that mobile processors are becoming ready for HPC, there are limitations which should be addressed. These limitations include lack of Error Correction Code (ECC) protection, slow interconnection, 32-bit address space, and inferior thermal package.

The authors in (JARUS et al., 2013) evaluate the performance and energy of HPC platforms based on Intel, AMD, and ARM processors. They used different benchmark suites, such as CoreMark, LINPACK, and High-Performance LINPACK. The results show that when the execution time matters, it is always better to choose performance-efficient CPUs, such as Intel Xeon E7 or Core i7 processors. On the other hand, the energy consumption of low-power processors (Intel Atom, AMD Fusion, and ARM Cortex-A9) is, in some cases, up to 12 times lower than that of the evaluated high-performance processors.

In (RAJOVIC et al., 2014), the authors present Tibidabo, an experimental HPC cluster built with ARM processors. Simulations show that a logical cluster of 16 ARM Cortex-A15 cores would increase the energy efficiency of the original cluster composed of Intel Corte i7 processors by 8.7 times. The authors in (PADOIN et al., 2014) compared the performance and energy of low-power and high-performance processors considering parallel benchmarks. They show that, in most cases, the ARM big.LITTLE presents better performance/energy trade-off compared to Intel Sandy Bridge-EP to execute the same applications.

An examination of the effects of TLP exploitation on mobile applications is presented in (FELLOWS, 2014). They compared the ARM Cortex-A15 with the Intel Core-i5 processor, using performance and EDP as metrics. They consider parallel benchmarks written with OpenMP and Intel TBB. By assessing memory-bound benchmarks, they have shown that Intel is four times better in performance and almost two times better in EDP. For CPU-Bound benchmarks, this difference decreases: Intel has the performance of fewer than three times better, and EDP of about 1.5 times better than the ARM.

The work in (DUKAN; KOVARI; KATONA, 2014) evaluates different CPUs to determine the strength of the current Mini PC market conditions: an ARM-based processor, an Intel Celeron, and an AMD processor. The results show that the ARM processors are preferred when energy consumption is considered; even though performance is slightly lower than the x86/64 rivals. Comparing only the x86/64 architecture processors, Intel consumed less energy than the AMD processor; however, their performance is very similar. The results also show that Intel is slightly faster in floating point calculations when compared to the other two processors.

The authors in (RAJOVIC et al., 2016) present the first HPC system built with commodities (SoCs, memories, and network interfaces) from the embedded domain, called the Mont-Blanc prototype. This project has the objective of build a large HPC system based on embedded technology, to provide an alternative HPC system with low energy consumption. The system is composed of 1080 nodes made of Samsung Exynos 5250 SoCs. The authors compare the Mont-Blanc prototype with MareNostrum III, a contemporary supercomputer. The experiments show that when running MPI applications, the Mont-Blanc prototype spends 9% less energy than the MareNostrum III, with a slowdown of 3.5 times. Moreover, the authors show that when both platforms target the same execution time, the Mont-Blanc prototype can reduce the energy consumption by up to 12.5%.

3.1.2 Parallel Computing with different Parallel Programming Interfaces

3.1.2.1 Performance Evaluation

The authors in (MALLÓN et al., 2009) evaluate the performance of MPI, Unified Parallel C (UPC), and OpenMP in multicore architectures through a subset of the NAS Parallel Benchmark. Executions were performed in an environment with hybrid memory (distributed and shared memory) and only shared memory. The results show that for the hybrid memory environment, MPI has the best overall performance compared to the UPC due to the better use of the cache memory. When considering the shared memory environment, although communication between the OpenMP threads occurs through shared variables, MPI and UPC performed better in some cases, due to the better use of the cache memory.

The work developed in (WAHLÉN, 2010) shows that the performance of a parallel application is highly influenced by its characteristics and the PPI used. The authors

discuss the performance of PThreads, OpenMP, and Cilk++ when executing iterative and recursive applications on a dual Quad-Core AMD Opteron. For iterative applications, PThreads got better speedup over its sequential counterpart (6.05 with 8 threads), while OpenMP and Cilk++ had similar results (5.60 and 5.79 respectively). On the other hand, for recursive applications, Cilk++ was better.

Considering only embedded processors, the authors in (LEE et al., 2011) evaluate TLP exploitation on a Dual-core ARM Cortex-A9. They consider two parallel programs implemented with OpenMP. Results show speedups of up to 111% with two threads over the sequential version. A comparative study of OpenMP, PThreads, and Cilk++ through the parallelization of a method for determining similarities between histograms is presented in (LORENZON et al., 2011). Executions were performed on an Intel Quad-core processor and showed that PThreads obtained the best performance, due to the fine-tuning during the distribution of workload.

The authors in (CAO et al., 2013) analyze the performance scalability of OpenMP, Intel TBB, and Cilk++. Fifteen parallel applications from different domains were executed on a 32-core system. Results show that most of the OpenMP applications didn't scale linearly with 32 threads. Among the reasons for the poor performance, the authors state that the scheduling strategy provided by the runtime system can negatively impact the performance.

The authors in (HUA; YANG, 2013) present a performance analysis of OpenMP and MPI through the execution of the Newton's iteration, a nonlinear algorithm. The authors used the Intel VTune Performance Analyzer tools, Intel Thread Checker and Intel Thread Profiler to analyze the behavior of these parallel programming interfaces. The results show that the use of OpenMP for shared memory systems can lead to good performance results, while the MPI has a favorable performance for parallel executions in distributed memory systems.

Salehian et al. (SALEHIAN; LIU; YAN, 2017) present a performance evaluation of different threading programming models. The study considers the following programming models: OpenMP, Intel Cilk Plus, and C++11. Ten benchmarks were implemented and classified into two different classes: simple computation kernels and applications from the Rodinia benchmark (CHE et al., 2009). The experiments were performed on a dual-processor Intel Xeon E5-2699v3 with a total of 36 cores. The authors conclude that the execution time is different for each programming model because of the strategies of workload balancing and the overhead of the scheduling and loop distribution. Moreover, the experiments showed that work-stealing has better performance for data-parallelism;

while tasking outperforms data-parallelism when there is dependency in different parallel loop phases.

3.1.2.2 Energy Evaluation

The study performed in (LIVELY et al., 2011) explores the energy and performance of different scientific applications. The experiment is focused on comparing MPI with hybrid programming (MPI/OpenMP) through two applications from the NAS Parallel benchmarks. The results show that for the execution of up to 16 cores, MPI applications obtained the lowest energy consumption and the best performance gains. As for the execution in 32 cores, the hybrid implementation (MPI/OpenMP) achieved better energy and performance gains. The authors also evaluated the use of CPU Frequency Scaling. In such case, the scenario that showed the lowest energy consumption was not the same that provided the best performance.

The authors in (BALLADINI et al., 2011) analyze the influence of OpenMP and MPI on the energy consumption and the behavior of systems at different clock frequencies of CPUs. Four applications from the NAS Parallel Benchmark, written using OpenMP and MPI were executed on a dual socket Intel Xeon dual core. The results show that the execution time, energy efficiency, and maximum power spent depend not only on the kind of application but also on its implementation in a programming model.

The performance and energy consumption of OpenMP implementations are studied in (ZECENA et al., 2012). Three traditional sorting algorithms are considered, such as odd-even, shell sort, and quick sort. The first two algorithms were implemented iteratively with parallel loops. On the other hand, the quicksort algorithm, which has the recursion as its main characteristic, was written with parallel tasks. The experiments were performed on a Quad-Core AMD Opteron processor. The results show that the quicksort achieved the best results of performance and energy savings. Additionally, the authors found that energy savings can be better when the granularity of the work to be assigned to each thread is correctly selected.

The authors in (PORTERFIELD et al., 2013) analyze factors that can influence the energy consumption of OpenMP applications compiled with Intel C++ Compiler (ICC) and GNU Compiler Collection (GCC). Through hardware performance counters present in the microarchitecture Intel Sandybridge, the authors measured the energy consumption of a variety of OpenMP programs. The evaluation revealed variations in energy consumption depending on the algorithm, its compiler optimization level, the number of threads

and the chip temperature. In most applications, the increasing in the number of threads allowed better performance, but with a substantial increase in energy consumption.

The study performed in (WANG; SCHMIDL; MÜLLER, 2015) investigates the energy consumption of parallel applications implemented with OpenMP on the Intel Haswell processor. The authors consider two benchmark classes: compute-bound, which is CPU-intensive; and memory-bound, represented by the Stream application. Considering that OpenMP applications can have imbalanced load, the authors evaluate different runtime configurations of *OMP_WAIT_POLICE* (i.e. time spent on a busy-waiting state): passive and active waiting. The results show that when the threads are put into a sleep state (passive waiting) instead of the active waiting state, it is possible to save energy without losing performance.

The authors in (LIMA et al., 2017) explores the performance and energy consumption of the OpenMP runtime system when executing on a NUMA platform. The study considers three different kernels from the dense linear algebra: Cholesky, LU, and QR matrix factorization. The authors compared the following runtime systems: LibGOMP, LibOMP, LibKOMP, and XKAAPI; on a machine composed of four NUMA nodes with a total of 96 cores. The results show that small algorithmic and runtime improvements may allow performance gains and energy reductions, when compared to a baseline without improvements.

3.1.3 Discussion

Table 3.1 presents a comparison of the environment and metrics evaluated in the extensive study performed in this thesis to the contributions discussed in this section. The internal rows of the "Execution Environment" detail the number of variables that the authors are considering in their work. For example, the row "distinct microarchitectures" presents the number of microarchitectures which the authors are evaluating. As it can be observed in the "TLP Exploitation" row, our work targets mainly multicore processors involving Embedded and General-Purpose Systems, rather than HPC systems, as it has already been extensively done in (OU et al., 2012; JARUS et al., 2013; STANISIC et al., 2013; RAJOVIC et al., 2013; FELLOWS, 2014; RAJOVIC et al., 2014; DUKAN; KOVARI; KATONA, 2014; PADOIN et al., 2014).

Few works investigate multicore for embedded systems, and they do so in a very limited environment and/or restricted number of metrics. In (HANAWA et al., 2009),

Table 3.1: Comparison of our contributions with the related work

Work	Execution environment					Evaluated metrics				
	Diff. comm. models-PPIs	*Diff. micro.*	*ES*	*GPPs*	*TLP*	*Perf.*	*Energy*	*EDP*	*Impact of delay on the EDP*	*Influence of static power of the proc.*
(HANAWA et al., 2009)			3		4	x				
(LEE et al., 2011)			1		2	x				
(BALLADINI et al., 2011)	2-2			1	4	x	x			
(OU et al., 2012)		2	1	1	4	x	x			
(PORTERFIELD et al., 2013)	2-2			1	16	x	x			
(BLEM; MENON; SANKARALINGAM, 2013)		4	3	1	1	x	x	x	x	
(JARUS et al., 2013) (STANISIC et al., 2013) (RAJOVIC et al., 2013) (RAJOVIC et al., 2014)		2	1	1	>=16	x	x			
(DUKAN; KOVARI; KATONA, 2014)		3	3		4	x	x			
(FELLOWS, 2014)		2	1	1	4	x	x	x		
(PADOIN et al., 2014)		2	2	1	8	x	x			
(WANG; SCHMIDL; MÜLLER, 2015)				1	36	x	x			
(RAJOVIC et al., 2016)			1	1	>=96	x	x			
(LIMA et al., 2017)				1	96	x	x			
(SALEHIAN; LIU; YAN, 2017)				1	36	x				
This Thesis - Chapter 4	**2-4**	**5**	**3**	**2**	**8**	**x**	**x**	**x**	**x**	**x**

Source: The Author

(LEE et al., 2011), and (SALEHIAN; LIU; YAN, 2017), only performance is evaluated; while in (BALLADINI et al., 2011) and (PORTERFIELD et al., 2013), communication models are studied on GPPs only. Even though the authors in (BLEM; MENON; SANKARALINGAM, 2013) perform an extensive evaluation of performance, energy, and EDP and consider different microarchitectures and ISAs, they evaluate only sequential applications.

Therefore, as it can be observed in Table 3.1, the study developed and presented in Chapter 4 extends the works above by looking for optimal points considering performance, energy consumption (with various levels of static power) and EDP; and executing a large range of applications with various communication models on a number of processors from different families and organizations.

3.2 Performance and Energy Optimization of Parallel Applications

In this section, the most representative approaches used to improve the performance and energy of parallel applications by defining the ideal number of threads are discussed considering the following:

1. **Adaptability**, with respect to the number of threads (i.e., when adaptation happens and whether it is continuous or not).

2. **Transparency**, which involves the need for special tools or compilers, programmer influence and/or changes in the source or binary codes.

First, Subsection 3.2.1 presents the studies that comprise the approaches where the definition of the ideal number of threads to execute a parallel application is performed before the application execution, that is, the approaches that use prediction models. In this case, there is no adaptation of the parallel execution at runtime. Then, Subsection 3.2.2 discusses the studies that perform adaptation at runtime and/or are transparent to the programmer/end user. Finally, Subsection 3.2.3 highlights the contributions of this thesis.

3.2.1 Approaches with No Runtime Adaptation and No Transparency to the User

In 2001 and 2002, Taylor et al. (TAYLOR et al., 2001; TAYLOR et al., 2002) propose a model that uses analytical models to predict the performance of three parallel applications from the NAS Parallel Benchmark (BT, LU, and SP). The model consists of performance coupling, which quantifies the interaction between adjacent kernels in a parallel application, giving more accuracy to the model. Results were validated on a machine with 80 processors, showing that the higher the performance coupling, the better the model accuracy.

In (IPEK et al., 2005), the authors have refined and adapted a multilayer neural network to predict performance results for the parallel application SMG2000 on two different high-performance platforms (IBM BlueGene and Intel Itanium 2). The proposed model predicts performance within 5% - 7% error across a large multidimensional parameter space. However, there is a large overhead due to the time required to gather each data point in the training set. An extended version of the work was published in (SINGH et al., 2007), where two benchmarks were added: a semi-coarsening multigrid solver and LINPACK.

To reduce the overhead of the performance prediction model, the authors in (YANG; MA; MUELLER, 2005) propose an approach that uses partial executions of an application to predict its behavior. The idea is to predict the overall execution time of a large-scale application through the execution of a short test drive of the application. Two benchmarks from the ASCI Purple suite were used to validate the model on ten different multicore platforms. The results show that the proposed approach can predict the performance with an accuracy of up to 97% or higher. Moreover, in the best case, it adds an overhead of only 1% on the total execution time.

The authors in (BARNES et al., 2008) explore the use of multivariate regression to predict the performance of a larger number of processors through training data obtained from a smaller number of processors. They propose three techniques: one that applies a multivariate regression over the execution time from the training step to predict the performance of a larger number of processors; and other two techniques that refine this approach by using a pre-processor information to handling computation and communication separately. The proposed model was validated running seven benchmarks from the NAS Parallel Benchmark, and Sweep3d on the Atlas cluster with 1152 four-way AMD Opteron nodes. Results show a prediction error between 6.2% and 17.3%.

A framework for the automatic construction of performance skeletons to predict performance on distributed environments is presented in (SODHI; SUBHLOK; XU, 2008). The approach captures the execution behavior of an application and automatically generates a skeleton of a program that reflects its entirely behavior. The approach was validated through the execution of six applications from the NAS Parallel Benchmarks on a cluster of 10 Intel Xeon dual CPU. Results show that the automatically generated performance skeletons can predict application performance with an average error around 8%.

The work developed in (SHARKAWI et al., 2009) proposes a methodology to predict the performance of HPC applications running on different architectures. The method uses data obtained from the executions on the base machine to predict the performance of other four systems (IBM JS22, p570, x3550 and x3650). Basically, benchmarks are executed on the base machine to get application performance metrics, which are correlated with data from the target platform through a genetic algorithm. After that, a model is generated to make the performance projection on the target platform. The model was validated through the execution of the SPEC CFP2006 benchmark suite on the base machine and the target platforms. Results show an average error of 7.2% when the performance is predicted to the same system where the data were collected, and an average error of up 12.8% for different ISAs.

The authors in (BHATTACHARJEE; MARTONOSI, 2009) propose a thread criticality predictor for parallel applications. The idea is to predict the slowest thread of an application, disable this thread or adjust the frequency of the processor, and then redistribute the workload among the remaining threads. The approach was validated on an ARM-based-in-order (32 cores) simulator running benchmarks from the SPLASH-2 and PARSEC suite. Results show that the approach can improve the performance of up to

31.8% and reduce the energy by 15% on average, compared to the execution without the approach.

Artificial neural networks (ANNs) were used in (TIWARI et al., 2012) to predict the power and energy usage for memory and CPU when executing certain HPC computational kernels. These ANNs are trained using empirical data gathered on the target architecture. The approach was validated running three distinct computational kernels (matrix multiplication, stencil, and LU factorization) on an Intel Xeon E5530 (which has 2 quad-core processors). The results show that, once the networks are trained, they can predict the performance, power and energy consumption for the CPU and memory with a maximum error of 5.5%.

The work developed in (BASMADJIAN; MEER, 2012) presents a methodology for estimating the dynamic power consumption of multi-core processors. The proposed mathematical model considers different components, such as chip-level (frequency and voltage), inter-die communication (active cores and dies involved in the communication/computation), and die-level (cores and off-chip caches). The authors also consider the impact of DVFS on the energy consumption. Two synthetic benchmarks were executed on Intel and AMD multicore processors to validate the approach. Results show that the model provides an accuracy within a maximum error of 5% when predicting the energy consumption of a parallel application.

The authors in (CABRERA et al., 2013) propose an analytical model to predict the energy consumption for the High-Performance LINPACK running on HPC systems. The proposed approach is based on the performance model presented in (CHOU et al., 2007). The authors added to this model new parameters regarding the energy required to perform communication and computation. The energy model was validated on a cluster with 24 nodes, each one containing an Intel Xeon dual-core. Results show that the model predicted the energy with 1% of error in the best case. However, it achieved an error of 67% in the worst case.

In (SONG; BARKER; KERBYSON, 2013), the authors present a unified performance and power model for the Nek-Bone mini-application using a combination of empirical analysis and micro-benchmarking. The approach considers the impact of computation and communication, and quantitatively predict their impact on both performance and energy consumption. The model was validated on a cluster with 64 nodes, each one with a dual-socket AMD Opteron processor. The results show that the model provides performance and energy prediction with a maximum error of 5% when predicting the behavior for up to 1024 cores.

Considering HPC applications, a model for software estimation of power consumption in an HPC environment is proposed in (WITKOWSKI et al., 2013). The authors use multivariate linear regression to find out the hardware data with high correlation with power consumption and to build the model. Several benchmarks were used to train and validate the model, such as Abinit, NAMD, Intel LINPACK, HMMER, among others, on three distinct machines: a dual-core Intel Xeon 3.0 GHz; a quad-core Intel Xeon 2.33 GHz; and a dual-core AMD Opteron 2.2 GHz. Results show that the proposed model can predict the power consumption of HPC applications with a maximum error of up to 7.88%.

An energy prediction mechanism for OpenMP applications using a Random Forest Modeling (RFM) approach in compilers is proposed in (BENEDICT et al., 2015). The approach is expressed in five steps: (*1*) an analyzer entity does an initial analysis of OpenMP applications regarding parallelism and code regions. (*2*) The optimizer entity finds the optimal energy solution for the identified code regions considering performance concerns. (*3*) The optimizer entity prepares a list of the best configurations and submits them to the prediction mechanism. (*4*) The energy consumption and performance for each configuration are predicted by using the RFM. (*5*) Finally, the predicted results are sent to the optimizer entity, where it would provide the best solution. The proposed approach was validated running different OpenMP applications (NAS benchmarks, matrix multiplications, n-body simulations, and stencil applications) on four Intel Xeon E5-4560, each offering 8 cores. From the experiments, the authors observed that RFM predicted the applications almost accurately with R2 (coefficient of determination) of 0.998 in the best case (the closer to 1, the better), and 0.814 in the worst case.

DwarfCode, a performance prediction for hybrid applications, is proposed in (ZHANG; CHENG; SUBHLOK, 2016). It uses computation and communication traces to predict the performance of MPI-OpenMP and MPI-ACC applications. DwarfCode captures these traces and generates a shorter benchmark of the entire application which mimics its behavior. Then, this shorter benchmark is executed on the target platform to predict the application's performance. The model was validated running the NAS parallel benchmarks on three clusters, each with a different number of nodes. Results show that the approach can predict the performance of MPI applications with an error rate lower than 10% for computing and communication-intensive applications.

A prediction framework that matches executions signatures for performance predictions of HPC applications using a single small-scale application execution is proposed in (JAYAKUMAR; MURALI; VADHIYAR, 2015). The framework extracts execution

signatures of applications and performs automatic phase identification of different application phases. Then, these signatures are matched with the execution profiles of reference kernels stored in a database and used to predict the performance of the application phases during execution time. To validate the prediction framework, three large-scale real scientific applications (GTC, Sweep3d, and SMG2000) were executed on an 800-core heterogeneous cluster and a 3600-core cluster. The results achieved show that the proposed framework can predict the energy consumption with errors in the range 0.4-18.7%.

A technique to predict the number of threads and DVFS level that offers the best performance and energy consumption for parallel applications is proposed in (SENSI, 2016). The idea is to execute the program using few configurations and then, predict the behavior of the other settings through multiple linear regression. The proposed technique was validated by executing the PARSEC benchmark on a machine with 24 cores and 13 possible CPU frequency levels. The results show that performance and power consumption can be predicted with an average of 96% of accuracy by executing only 1% of the total possible configurations.

3.2.2 Approaches with Runtime Adaptation and/or Transparency to the User

Thread Reinforcer (PUSUKURI; GUPTA; BHUYAN, 2011) is an example of work that presents a certain level of transparency to the user, but cannot adjust the number of threads dynamically, at run-time. It consists of a framework that runs in two steps: (*i*) the application binary is executed multiple times with a different number of threads for a short period (e.g., 100 ms), while Thread Reinforcer searches for the appropriate configuration. (*ii*) Once this configuration is found, the application is fully re-executed with the number of threads defined in the first step. By executing the application binary already compiled, Thread Reinforcer is a particular case that keeps binary compatibility. However, it works well only for applications that have a short initialization period - thus introducing a small overhead -, and it considers that all parallel regions of an application have the same behavior.

The approaches proposed in (JUNG et al., 2005) and (SULEMAN; QURESHI; PATT, 2008) already present some adaptability through the definition of the number of threads at runtime. (JUNG et al., 2005) present performance models for generating adaptive parallel code for SMT architectures. In their work, an analysis is applied during compile time to filter parallel loops in OpenMP in which the overhead from the thread

management (creation/termination, workload distribution, and synchronization) is higher than its own workload. Then, at run-time, the master thread uses the compilation time analysis to dynamically estimate whether it should use the SMT feature of the processor or not. This approach is also dependent on a compiler and can only be applied for SMT processors.

Suleman et al. propose the feedback-driven threading (FDT) framework (SULEMAN; QURESHI; PATT, 2008), which can adapt the number of threads considering contention for locks and memory bandwidth. The framework consists of a specific compiler that samples a portion of parallel regions of an application implemented with OpenMP, insert instructions at the entry and exit of the critical section, and executes it sequentially to analyze synchronization and communication points. It then uses this analysis to estimate the optimal number of threads for the given parallel region. The application is executed with the estimated number of threads and cannot re-adapt at runtime. Furthermore, FDT considers that all threads are homogeneous and ignores fundamental hardware characteristics that are highly correlated to the parallel application behavior: FDT assumes that bandwidth requirement increases linearly with the number of threads, ignoring cache contention and data-sharing between the threads. Moreover, FDT does not consider the effects of the SMT feature (discussed in Section 2.2), by assuming that only one thread executes per core.

More adaptive solutions, which consider run-time and continuous adaptation, include (CURTIS-MAURY et al., 2006; CURTIS-MAURY et al., 2008; LEE et al., 2010; CHADHA; MAHLKE; NARAYANASAMY, 2012; RAMAN et al., 2012; PORTERFIELD et al., 2013; ALESSI et al., 2015). However, these solutions either rely on hardware/OS support, or special compiler and need for recompilation or a previous off-line analysis, as discussed next.

In (CURTIS-MAURY et al., 2006), Curtis-Maury et al. propose a framework for nearly optimal on-line adaptation of multithreaded code for low-power and high-performance execution. The approach has an off-line phase in which data from hardware counters are collected, and profiles of parallel execution are analyzed. Then, at runtime, the framework uses the information obtained in the off-line phase to adapt the number of threads and achieve optimal performance or energy consumption. A solution proposed by Curtis-Maury et al. is ACTOR (CURTIS-MAURY et al., 2008), a system that dynamically changes the number of threads to improve energy efficiency. ACTOR is divided into three steps: (i) artificial neural networks (ANNs) are trained off-line to model the relationship between performance counter events and the resulting performance with a different

number of threads; (ii) at runtime, the derived ANN models are used to predict the performance of parallel regions that were previously identified by the programmer with special function calls from the ACTOR library; (iii) the parallel regions are executed with the predicted number of threads. Although the number of threads is predicted at runtime in (CURTIS-MAURY et al., 2006) and (CURTIS-MAURY et al., 2008), an off-line phase is required before the execution of each parallel application. Therefore, if either the input set or processor is changed, the off-line analysis must be re-executed, which significantly increases the total execution time of the entire framework.

Thread Tailor (LEE et al., 2010) is a framework that dynamically adjusts the number of threads to optimize system efficiencies, such as cache and memory space. The approach works as follows: (*i*) programmers create a parallel application that uses a high number of threads; (*ii*) the binary created is profiled off-line to collect statistics regarding the number of threads, communication, and synchronization to form a communication graph; (*iii*) at runtime, a dynamic compiler takes a quick snapshot of the system state to determine how many free resources are available and to decide the optimum number of threads; (*iv*) based on that information, the dynamic compiler generates code for the new threads, intercepts future calls to thread creations, and redirects them to the new threads. However, as Thread Tailor works for PThreads and MPI, it requires huge effort from the programmer to develop a parallel application that is able to use a high number of threads/processes, since the developer must explicitly implement thread/process management (creation/termination), workload distribution, synchronization and communication points between threads/processes.

LIMO (CHADHA; MAHLKE; NARAYANASAMY, 2012) is a dynamic system that monitors the application at run-time, being able to adapt the execution accordingly. However, this solution requires hardware modifications to determine the working set size of a thread, as well as additional operating system support for detecting threads that block due to busy-wait (spin loop). Consequently, it cannot be applied to any existent commercial microarchitecture. Also, LIMO relies on compiler support to insert special system calls and to modify loop bodies. Therefore, applications need to be recompiled to take advantage of LIMO functionalities.

Parcae (RAMAN et al., 2012) is a framework that comprises a compiler and runtime system to optimize the overall system performance. The compiler (Nona) identifies parallelizable regions in a sequential program and applies multiple parallelizing transforms (data-parallel with critical sections and a pipeline transform) to each region. When the application is executing, the Parcae run-time system (Decima monitor and the

Morta executor) monitors the program performance and system events to determine the best configuration for the parallel application. However, Parcae relies on system support (compiler, monitor, and executor) to modify sequential applications at compilation and execution time. Therefore, if there are any changes in the environment (input set or microarchitecture), the application need to be recompiled.

Porterfield et al., (PORTERFIELD et al., 2013) propose an adaptive run-time system that automatically adjusts the number of threads based on on-line measurements of system resource usage. The approach extends Qthreads (a parallel library (WHEELER; MURPHY; THAIN, 2008)) to be used with MAESTRO, a dynamic runtime library for power and concurrency adaptation of parallel applications (PORTERFIELD; FOWLER; NEYER, 2008). However, it is dependent on ROSE source-to-source compiler (QUINLAN; LIAO, 2011) to obtain OpenMP directives and map the functions and data structures to the Qthreads library. OpenMPE (ALESSI et al., 2015) is an extension designed for energy management of OpenMP applications, in which the programmers expose energy saving opportunities through the insertion of directives in OpenMP codes. However, it works only for the Insieme compiler and runtime system from the Insieme Project (JORDAN et al., 2012).

Shafik et al. (SHAFIK et al., 2015a) propose an adaptive and scalable energy minimization model for OpenMP programs, which comprises two steps: (*i*) code annotations are inserted by the programmer in the sequential and parallel parts of the code to enable energy minimization with specified performance requirements; (*ii*) the runtime system reads these performance requirements and uses this information to guide the energy minimization. The same method, but aiming to improve lifetime reliability through balanced thermal controls while meeting a given power budget, was presented in (SHAFIK et al., 2015b). All of these works need code recompilation.

More transparent approaches, which do not need support from special compilers, include (SRIDHARAN; GUPTA; SOHI, 2013), (SRIDHARAN; GUPTA; SOHI, 2014). In (SRIDHARAN; GUPTA; SOHI, 2013), ParallelismDial (PD), a model that automatically tunes a program's performance to the underlying system is proposed. It monitors the system efficiency, regulates the degree of parallelism, and continuously adapts the execution through a heuristic to an optimum point of operation. The heuristic used to find the best degree of parallelism is based on the hill-climbing search algorithm, which works as follows: (*i*) the parallel region runs with only one thread to establish a sequential measure; (*ii*) the same region is executed with three degrees of parallelism (low, medium,

and high); (*iii*) the search is refined to the best interval and continues until the optimum point be reached.

In (SRIDHARAN; GUPTA; SOHI, 2014), ParallelismDial was extended to Varuna system. It comprises two components: (*i*) an analytical engine which continuously monitors changes in the system using hardware performance counters, models the execution behavior, and determines the optimum degree of parallelism; and (*ii*) a manager that automatically regulates the execution to match the degree of parallelism determined by the analytical engine. 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. However, to do so efficiently, PD and Varuna create a large number of fine-grained tasks. Consequently, it requires more effort from the programmer, that is, the programmer is required to create as many threads as possible, each one with the lowest possible workload. Because of this intrinsic characteristic, PD and Varuna focus on recursive applications that are mostly concentrated on big-data. Besides that, they cannot optimize OpenMP applications due to limitations of the system (virtual tasks) used to control parallelism (ADYA et al., 2002).

3.2.3 Context of this Thesis

Table 3.2 compares the approaches proposed in this thesis (LAANT and Aurora) to previous works. The column *run-time adaptation* indicates the approaches able to select the ideal number of threads as the parallel application executes. However, once the technique converged to a given number of threads, this number will not change anymore. On the other hand, *continuous adaptation* refers to those works that can readjust the number of threads during application execution according to variations in its workload or environment system. The column *no special compiler/tools* presents the approaches that do not need any specific compiler or tool to generate special parallel code (i.e. use different tools from the traditional programming framework that usually involves a C/C++ compiler and a parallel API). The column *no programmer influence* contains the approaches that do not demand any changes in the source code by the software developer. *Binary compatibility* refers to the techniques that can be used without any need for code recompilation at all: the existent binary code as is can take advantage of the approach. The column *PPI* shows the parallel libraries supported by each referred work. Finally, *Diverse metrics* refers to works that can optimize more than one metric.

Table 3.2: Comparison of LAANT and Aurora with the related work

Proposal	**Adaptability**		**Transparency**			**PPIs**	**Diverse Metrics**
	Run-time	*Continuous*	*No special compilers/tools*	*No programmer Influence*	*Binary Compatibility*		
Thread Reinf. (PUSUKURI; GUPTA; BHUYAN, 2011)				x	x	OMP, PT	
(JUNG et al., 2005)	x			x		OMP-FT	
FDT (SULEMAN; QURESHI; PATT, 2008)	x			x		OMP	x
(CURTIS-MAURY et al., 2006)	x	x		x		OMP-FT	x
LIMO (CHADHA; MAHLKE; NARAYANASAMY, 2012)	x	x		x		OMP, PT	x
Parcae (RAMAN et al., 2012)	x	x		x		Seq.	
Porterfield et al. (PORTERFIELD et al., 2013)	x	x		x		OMP	
ACTOR (CURTIS-MAURY et al., 2008)	x	x				OMP	x
Thread Tailor (LEE et al., 2010)	x	x				PT, MPI	
OpenMPE (ALESSI et al., 2015)	x	x				OMP	x
(SHAFIK et al., 2015a) (SHAFIK et al., 2015b)	x	x	x			OMP	
ParallelismDial (SRIDHARAN; GUPTA; SOHI, 2013)	x	x	x			TBB, PM	
Varuna (SRIDHARAN; GUPTA; SOHI, 2014)	x	x	x			PT, TBB	x
LAANT - Chapter 5.1	**x**	**x**	**x**			**OMP**	**x**
Aurora - Chapter 5.2	**x**	**x**	**x**	**x**	**x**	**OMP**	**x**

PT: PThreads; OMP: OpenMP; Seq: Sequential Code; PM: Prometheus; FT: FORTRAN

Source: The Author

LAANT comprises the first effort to develop a mechanism that is entirely transparent to the user and adaptive: a library that provides adaptation at run-time and continuously of each parallel region with no need of special compilers and tools. Different from the mechanisms developed by Shafik et al. (SHAFIK et al., 2015a)(SHAFIK et al., 2015b), LAANT is able to optimize each parallel region for different metrics, such as performance, energy, and EDP. Because it is a library, LAANT relies on programmer influence and it does not present binary compatibility, which means that the application must be recompiled in order to receive the LAANT functions. LAANT is presented in Section 5.1.

As depicted in Table 3.2, no approach discussed covers all the needed characteristics so it could be considered completely transparent and adaptive. On the other hand, Aurora can find the ideal number of threads as the application is being executed, and continuously adjusts the number of threads if there are changes in the workload or the environment system. Also, it works with any C/C++ compiler and OpenMP and can target its optimization mechanism to different metrics and systems: embedded, GPPs, and HPC. More important, the software developer does not need to make any changes in the source code that is already parallelized with OpenMP or even recompile it. Any existent binary code that uses OpenMP can take advantage from Aurora. For that, the programmer only has to enable Aurora and its optimization metric through the use of one environment variable in the Linux Operating System. However, because of this high level of transparency, Aurora is limited to OpenMP applications. Section 5.2 presents Aurora and its validation.

4 POSSIBILITY OF PARALLEL COMPUTING EXPLOITATION

Thread-level parallelism exploitation is being widely used to make the best use of hardware resources and improve performance. However, as discussed in Chapter 1, energy consumption has become an important issue. Therefore, the objective when designing parallel applications is not to simply improve performance but to do so with a minimal impact on energy consumption. In order to speed up the development process of parallel applications and make it as transparent as possible to the programmer, different PPIs are used (e.g. OpenMP, PThreads, or MPI). However, each one of these has different characteristics with respect to the management (i.e. creation and finalization of threads/processes), workload distribution, synchronization, and communication.

Considering the aforementioned scenario, the first step of this thesis performs a comprehensive study of the opportunities for parallel computing regarding the most common parallel programming interfaces that exploit parallelism through shared variables (OpenMP and PThreads) or message passing (MPI-1 and MPI-2). Fourteen applications, classified according to their communication demands, were parallelized and executed on different embedded and general-purpose processors. Several metrics were used to evaluate the parallel programming interfaces and multicore processors, such as performance, energy, EDP, and the influence of the processor's static power on the total energy consumption. Finally, we present a comparison between the parallel programming interfaces regarding their use to implement the popular benchmarks. Such comparison shows the importance of optimizing OpenMP applications and justifies its choice as the focus of this work.

The remainder of this Chapter is organized as follows. First, the methodology used is discussed, i.e., the multicore architectures, benchmark suite, communication models and parallel programming interfaces, setup, and how the energy consumption was calculated. Then, the results are discussed. Finally, we present the conclusions of this study, and discuss the importance of the OpenMP for the parallel computing in the last section.

4.1 Methodology

4.1.1 Benchmarks

In order to study the characteristics of each PPI regarding the thread/process management and synchronization/communication, fourteen parallel benchmarks were imple-

Table 4.1: Main characteristics of the benchmarks

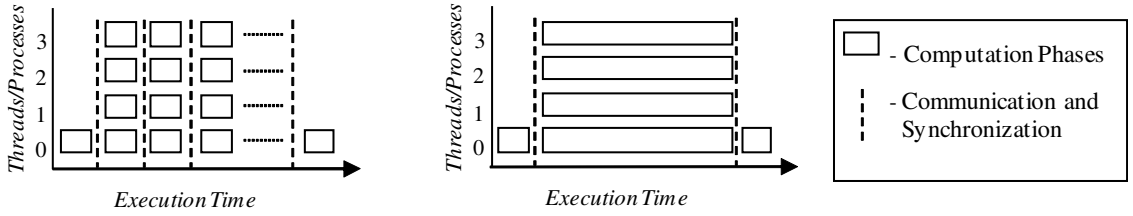
Benchmarks		Operations to exchange data (*total per n° of threads/processes*)				Input size
		2	3	4	8	
HC	***Game of Life***	414	621	1079	1625	4096 × 4096
	Gauss-Seidel	20004	20006	20008	20016	2048 × 2048
	Gram-Schmidt	3009277	4604284	6385952	12472634	2048 × 2048
	Jacobi	4004	6006	8008	16016	2048 × 2048
	Odd-even sort	300004	450006	600008	1200016	150000
	Turing ring	16000	24000	32000	64000	2048 × 2048
LC	***Calc. of the PI number***	4	6	8	16	4 billions
	DFT	4	6	8	16	32368
	Dijkstra	4	6	8	16	2048 × 2048
	Dot-product	4	6	8	16	15 billions
	Harmonic series	8	12	16	32	100000
	Integral-quadrature	4	6	8	16	1 billion
	Matrix multiplication	4	6	8	16	2048 × 2048
	Similarity of histograms	4	6	8	16	1920 × 1080

Source: The Author

mented and parallelized in C language and classified into two classes: High Communication (HC) and Low Communication (LC). For this, it was considered the amount of communication (i.e., data exchange), the synchronization operations needed to ensure data transfer correctness (mutex, barriers), and operations to create/finalize threads/processes. Table 4.1 quantifies the communication rate for each benchmark (it also shows their input sizes), considering 2, 3, 4 and 8 threads/processes, obtained by using the Intel Pin Tool (LUK et al., 2005). HC programs have several data dependencies that must be addressed at runtime to ensure correctness of the results. Consequently, they demand large amounts of communication among threads/processes, as it is shown in 4.1a. On the other hand, LC programs present little communication among threads/processes, because they are needed only to distribute the workload and to join the final result (as it is shown in Figure 4.1b)).

Since the way a parallel application is written may influence its behavior during execution, we have followed the guidelines indicated by (FOSTER, 1995), (BUTENHOF, 1997), (GROPP; LUSK; SKJELLUM, 1999), and (CHAPMAN; JOST; PAS, 2007). The OpenMP implementations were parallelized using parallel loops, splitting the number of loops iterations (*for*) among threads. As discussed in (CHAPMAN; JOST; PAS, 2007), this approach is ideal for applications that compute on uni- and bi- dimension structures, which is the case. Loop parallelism can be exploited by using different scheduling types that distribute the iterations to threads (static, guided, and dynamic) with different granularities (number of iterations assigned to each thread as the threads request them). As demonstrated in (LORENZON; CERA; BECK, 2015b), the static scheduler with coarse granularity presents the best results for the same benchmark set used in this study and,

Figure 4.1: Behavior of benchmarks



a) High Communication b) Low Communication

Source: The Author

therefore, this scheduling mechanism is used here. On the other hand, as indicated by (FOSTER, 1995), (BUTENHOF, 1997), and (GROPP; LUSK; SKJELLUM, 1999), the approach using parallel tasks was utilized in PThreads and MPI implementations. In such cases, the loops iterations were distributed based on the best workload balancing between threads/processes. Moreover, the communication between MPI processes was implemented by using non-blocking operations, to provide better performance, as showed in (HOEFLER; LUMSDAINE; REHM, 2007).

4.1.2 Multicore Architectures

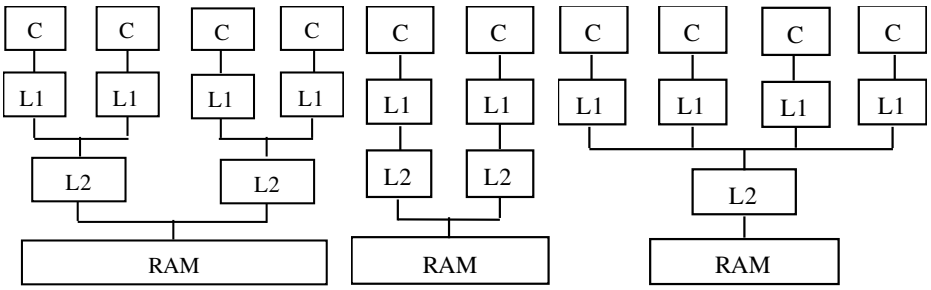
4.1.2.1 General-Purpose Processors

Core2Quad: The Intel Core2Quad is an implementation of the x86-64 ISA. In this study, the 45 nm Core2Quad Q8400 was used, which has 4 CPU cores running at 2.66 GHz, and a TDP of 95 W. It uses the Intel Core microarchitecture targeted mainly to desktop and server domains. It is a highly complex superscalar processor, which uses several techniques to improve ILP: memory disambiguation; speculative execution with advanced prefetchers; and a smart cache mechanism that provides flexible performance for both single and multithreaded applications[1]. As Figure 4.2(a) shows, the memory system is organized as follows: each core has a private 32 kB instruction and 32 kB data L1 caches. There are two L2 caches of 2 MB (4 MB in total), each of them shared between clusters of two cores. The platform has 4 GB of main memory, which is the only memory region accessible by all the cores.

Xeon: The Intel Xeon is also an x86-64 processor. The version used in this work is a 45 nm dual processor Xeon E5405. Each processor has 4 CPU cores (so there are 8 cores

[1]Available at: http://www.intel.com/technology/architecture/coremicro

Figure 4.2: Memory organization of each processor used in this work



(a) Intel Core2Quad and Xeon (b) Intel Atom (c) ARM Cortex-A9/A8

Source: The Author

in total), running at 2.0 GHz, with a TDP of 80 W. It also uses the Core microarchitecture; however, unlike Core2Quad, Xeon processor E5 family is designed for industry-leading performance and maximum energy efficiency, since it is widely employed in HPC systems. The memory organization is similar to the Core2Quad (Figure 4.2(a)): each core has a private 32 kB instruction and 32 kB data L1 caches. There are two L2 caches of 6 MB (12 MB in total), each of them shared between clusters of two cores. The platform has 8 GB of RAM, which is the only memory region accessible by all the cores.

4.1.2.2 Embedded Processors

Atom: The Intel Atom is also an x86-64 processor, but targeted to embedded systems. In this study, the 32 nm Atom N2600 was used, which has 2 CPU cores (4 threads by using Hyper-Threading support) running at 1.6 GHz, a TDP of 3.5 W. It uses the Saltwell microarchitecture, designed for portable devices with low power consumption. Since the main characteristic of x86 processors is the backward compatibility with the x86 instructions set, programs already compiled for these processors will run without changes on Atom[2]. The memory system is organized as illustrated in Figure 4.2(b): each core has 32 kB instruction and 24 kB data L1 caches, and a private 512 kB L2 cache. The platform has 2 GB of RAM, which is the memory shared by all the cores.

ARM: We consider the Cortex-A9 processor. ARM is the world's leading in the market of embedded processors. Designed around a dual-issue out-of-order superscalar, the Cortex-A family is optimized for low-power and high-performance applications[3]. The 40 nm ARM Cortex-A9 is a 32-bit processor, which implements the ARMv7 architecture with 4 CPU cores running at 1.2 GHz and TDP of 2.5 W. The memory system is organized

[2]Available at: http://www.intel.com/content/www/us/en/processors/atom/atom-processor.html
[3]Available at: http://www.arm.com/products/processors/cortex-a/index.php.

as illustrated in Figure 4.2(c): each core has a private 32 kB instruction and 32 kB data L1 caches. The L2 cache of 1 MB is shared among all cores, and the platform has 1 GB of RAM. Since the ISA and microarchitecture of the Cortex-A8 and Cortex-A9 are similar, we also investigate the behavior of A8 based on the results obtained in the A9. The version considered is a 65 nm Cortex-A8 which has an operating frequency of 1 GHz, a TDP of 1.8 W.

4.1.3 Execution Environment

The Performance Application Programming Interface (PAPI) (BROWNE et al., 2000) was used to evaluate the behavior of processor and memory system without the influence of the operating system (i.e., function calls, interruptions, etc.). By inserting functions in the code, PAPI allows the developer to obtain the data directly from the hardware counters present in modern processors. With these hardware counters, it is possible to gather the number of completed instructions, memory accesses (data/instructions), and the number of executed cycles to calculate performance and energy consumption.

The energy consumption was calculated using the data provided by the authors in (BLEM; MENON; SANKARALINGAM, 2013) (for the processors) and Cacti Tool [4] (for the memory systems), as shown in Table 4.2. To estimate the total energy consumption (*Et*), we have taken into account the energy consumed for the executed instructions (E_{inst}) , cache and main memory accesses (E_{mem}), and static energy (E_{static}), as given by

[4]Retrieved from http://www.cs.utah.edu/%7Erajeev/cacti6/

Table 4.2: Energy consumption for each component on each processor

	ARM		**Intel**		
	Cortex-A8	***Cortex-A9***	***Atom***	***Core2Quad***	***Xeon***
Processor - static power	0.17 W	0.25 W	0.484 W	4.39 W	3.696 W
L1-D static power	0.0005 W	0.0005 W	0.00026 W	0.0027 W	0.0027 W
L1-I static power	0.0005 W	0.0005 W	0.00032 W	0.0027 W	0.0027 W
L2 - static power	0.0258 W	0.0258 W	0.0096 W	0.0912 W	0.1758 W
RAM - static power	0.12 W	0.12 W	0.149 W	0.36 W	0.72 W
Energy per instruction	0.266 nJ	0.237 nJ	0.391 Nj	0.795 nJ	0.774 nJ
L1-D - energy/access	0.017 nJ	0.017 nJ	0.013 nJ	0.176 nJ	0.176 nJ
L1-I - energy/access	0.017 nJ	0.017 nJ	0.015 nJ	0.176 nJ	0.176 nJ
L2 - energy/access	0.296 nJ	0.296 nJ	0.117 nJ	1.870 nJ	3.093 nJ
RAM - energy/access	2.77 nJ	2.77 nJ	3.94 nJ	15.6 nJ	24.6 nJ

Source: The Author

Equation 4.1.

$$Et = E_{inst} + E_{mem} + E_{static} \tag{4.1}$$

To find the energy consumed by the instructions, Equation 4.2 was used, where I_{exe} is the number of executed instructions multiplied by the average energy spent by each one of them ($E_{perinst}$).

$$E_{inst} = I_{exe} * E_{perinst} \tag{4.2}$$

The energy consumption for the memory system was obtained with Equation 4.3, where ($L1DC_{acc} \times E_{L1DC}$) is the energy spent by accessing the L1 data cache memory; ($L1IC_{acc} \times E_{L1IC}$) is the same, but for the L1 instruction cache; ($L2_{acc} \times E_{L2}$) is for the L2 cache; and ($L2_{miss} \times E_{main}$) is the energy spent by the main memory accesses.

$$E_{mem} = (L1DC_{acc} \times E_{L1DC}) + (L1IC_{acc} \times E_{L1IC}) + (L2_{acc} \times E_{L2}) + (L2_{miss} \times E_{main}) \tag{4.3}$$

The static consumption of all components is given by Equation 4.4. As static power is consumed while the circuit is powered, it must be considered during all execution time: (*#Cycles*) of application divided by the operating frequency (*Freq*). We have considered the static consumption of the processor (S_{CPU}), L1 data (S_{L1DC}) and instruction (S_{L1IC}) caches, L2 cache (S_{L2}), and main memory (S_{MAIN}).

$$E_{static} = \left(\frac{\#Cycles}{Freq}\right) \times (S_{CPU} + S_{L1DC} + S_{L1IC} + S_{L2} + S_{MAIN}) \tag{4.4}$$

4.1.4 Setup

The results presented in the next section consider an average of ten executions, with a standard deviation of less than 1% for each benchmark. Their input sizes are described in Table 4.1. The programs were split into 2, 3, 4 and 8 threads/processes. Although most of the processors used in this work support only four threads, and are not commercially available in an 8-core configuration, it is possible to approximate the results by using the following approach: as an example, let us consider that we have two

threads executing on one core only. These threads have synchronization points and when one thread gets there, it must wait for the other one; and so on as long as there still are synchronization points. What it is done is to gather data of each thread executing on the core in between two synchronization points (which involves number of instructions, memory access, execution time etc.). This behavior would be the same as if the two threads would be executing on two different cores, since the cores are homogeneous (i.e.: have the same organization and, therefore, the same ILP exploitation capabilities). There may have context switches between both threads as they are executing, but they are not considered for the calculations (in the same way other services of the operating system are not considered). Therefore, at the end of execution, we have all the data of each thread for each part of code in between synchronization points. We can calculate the energy consumption because we have the number of executed instructions, memory accesses and so on; and we can infer the performance since we have the execution time of each part of code of each thread in between two synchronization points. For each part, we consider as execution time the one presented by the slowest thread (which simulates the behavior of one waiting for another if they were actually executing on two cores). This approach can be easily extrapolated to a larger number of threads.

The compiler used was the GCC-4.7.3 without optimization flags, to minimize the influence of the compiler on the PPIs. The following distributions were used: OpenMPI 1.6, OpenMP 3.0, and PThreads/POSIX.1-2008, running on the Linux Debian operating system.

It is important to highlight that the results discussed here are restricted to this chapter due to the following reasons:

- The benchmark set was developed and classified with the only purpose to evaluate each PPI regarding the thread/process management, workload distribution, and synchronization/communication;

- The versions of libraries, compilers, and tools used here have been updated since the experiments were performed;

- When this study was performed, we did not have access to processors that provide energy consumption directly from the hardware counters.

4.2 Results

4.2.1 Performance and Energy Consumption

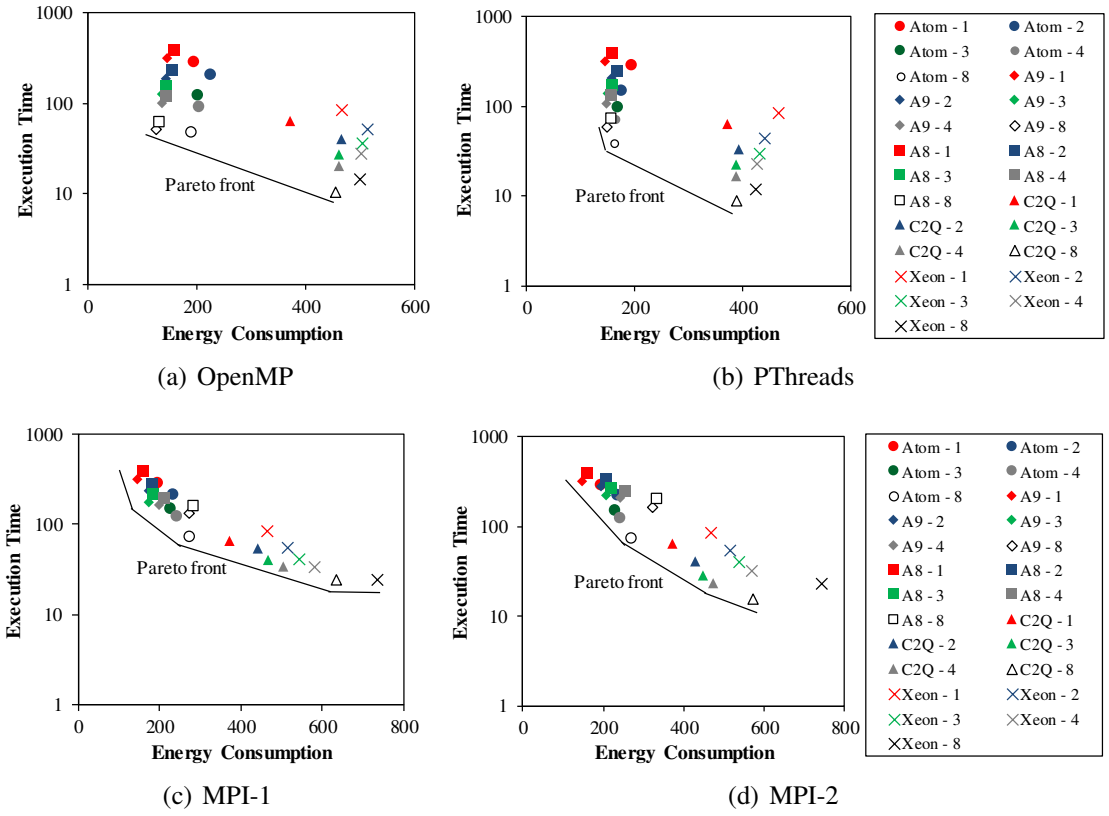
Figures 4.3 - 4.9 show the results of performance (in seconds) and energy (in Joules) of each processor and number of threads/processes ("1" means sequential execution) for the two benchmark classes (High and Low communication). Figures 4.3 and 4.7 show raw numbers, where the x-axis of each chart is the energy consumption, and the y-axis is the execution time. Figures 4.4 and 4.8 demonstrate the fraction of energy consumed by each hardware component with respect to the total energy. Static and dynamic (S and D) energy for the processor and memory are considered. Also, Figures 4.5 and 4.9 present the normalized performance and energy using the processor with the best results as the baseline. The results are discussed in detail in the next sub-sections, considering both classes of programs separately.

4.2.1.1 High-Communication Programs

Figure 4.3 shows the performance and energy consumption for each processor running a different number of threads/processes. Each chart analyzes a different parallel programming interface. Considering the performance, regardless the PPI used, all the processors performed better when exploiting a TLP of 8, and Core2Quad processor achieved the lowest execution time. Comparing the best case of each processor, Core2Quad is 4.32 times faster than Atom; 5.73 times faster than Cortex-A9; 6.87 times faster than Cortex-A8; and 1.34 times faster than Xeon. Considering only the embedded processors, Atom performed better, being 1.32 and 1.59 times faster than Cortex-A9 and A8, respectively.

When the energy consumption matters, embedded processors spend less energy than GPPs, and the A9 is the most efficient one. Considering the lowest energy consumption in each processor: A9 consumed 25% less energy than Atom; 8% less than A8; 61% less than Core2Quad; and 69% less energy than Xeon. In the most significant case, this difference is even greater: A9 consumed 55% less energy than Atom; 63% less than A8; 81% less than Core2Quad; and 84% less than Xeon. Moreover, the processors have different behaviors according to the PPI used: if the HC programs are parallelized using OpenMP, it is better to use the ARM Cortex-A9 exploiting a TLP of 8. In such case, the energy consumed is 35% lower than the best result in the Atom; and 5%, 64% and 73% lower than the A8, Core2Quad and Xeon, respectively. In another situation,

Figure 4.3: Performance (seconds) and energy consumption (joules) results for High-Communication programs



(a) OpenMP (b) PThreads

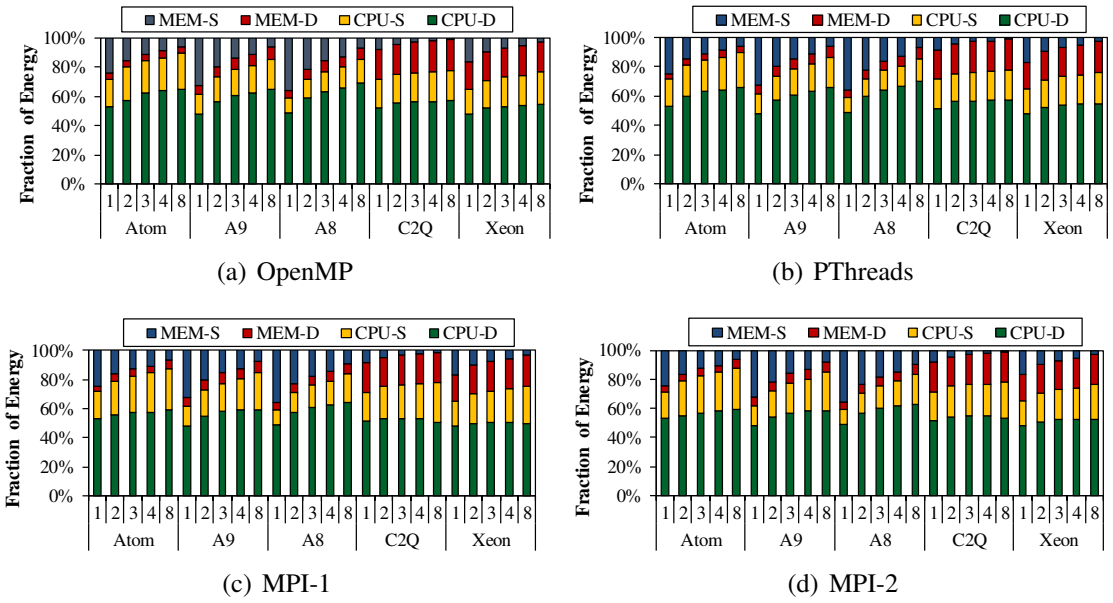
(c) MPI-1 (d) MPI-2

Source: The Author

when HC programs are parallelized using PThreads, MPI-1, or MPI-2, the lowest energy consumption is achieved by executing the sequential versions of the benchmarks on the Cortex-A9. Therefore, when it comes to energy and these interfaces, it is better to use one core only—even if there are more available.

In this application class, in which there are many accesses to the shared memory because of data exchange, the processor's performance and energy are highly influenced by the communication model (Figure 4.3). For shared variables (OpenMP and PThreads), there are significant performance improvements, even though it does not increase in the same ratio as the TLP exploitation increases (i.e. when the number of threads is equal to 2, the execution time of a parallel version is greater than the half of its sequential version, and so on). In addition, parallel applications have similar energy consumption when one compares to their sequential counterparts in most cases. On the other hand, when using message passing (MPI-1 and MPI-2), even though there are performance gains, execution time decreases at a slower rate as the TLP increases, when compared to applications implemented using OpenMP and PThreads. The performance gains are limited by the excessive number of send/receive operations performed by communication, becoming

Figure 4.4: Fraction of energy consumed by each hardware component (MEM: Memory; CPU: Processor; D: Dynamic; S: Static) for HC applications



(a) OpenMP (b) PThreads

(c) MPI-1 (d) MPI-2

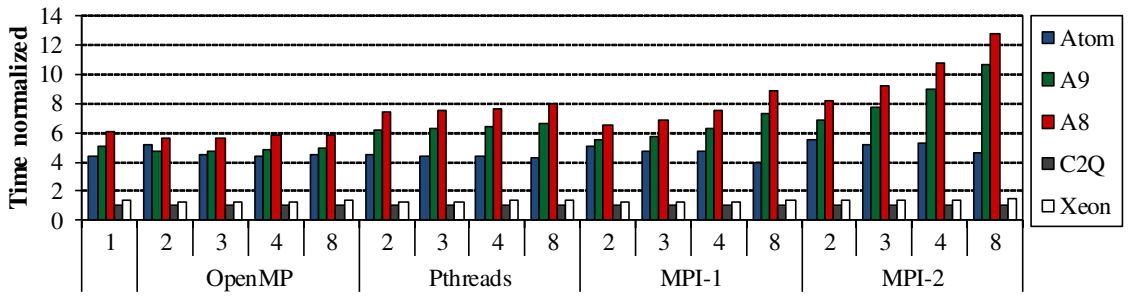
Source: The Author

a bottleneck. As a result of this poor performance improvements, energy consumption increases, compared to the sequential version, in all cases.

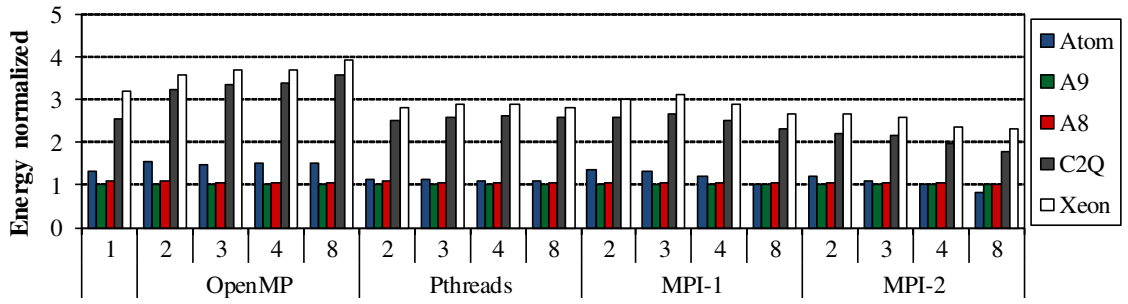
As there is no optimal combination of processor and number of threads/processes that offer at the same time the best performance with the lowest energy consumption, one must choose which metric is the most significant. In this way, the Pareto front is used in the charts. As Figure 4.3 shows, it varies according to the PPI: in the OpenMP, there is only one combination offering the lowest energy consumption (Cortex-A9 executing 8 threads) and one with the best performance (Core2Quad, also running 8 threads). When other PPIs are used, the number of combinations is greater than three. Another interesting fact is that while we have few points when it comes to shared memory based PPIs (OpenMP and PThreads), the Pareto Front is composed of several points when it comes MPI (Message Passing), increasing the complexity of finding the best trade-off in energy and performance.

Moreover, there are cases in which it is possible to reduce the energy consumption maintaining similar performance when embedded processors are chosen instead of GPPs. In the most significant case, it is possible to save 76% in energy by executing OpenMP HC programs on the Cortex-A9 with 8 threads instead of on the Xeon with 2 threads. On the other hand, if one chooses general-purpose instead of embedded processors aiming to reduce execution time, there is no single option available that will not result in huge increases in energy consumption. For instance, executing PThreads HC Applications with

Figure 4.5: Results normalized to Core2Quad (performance) and A9 (energy) - HC Programs



(a) Performance normalized to Core2Quad



(b) Energy consumption normalized to A9

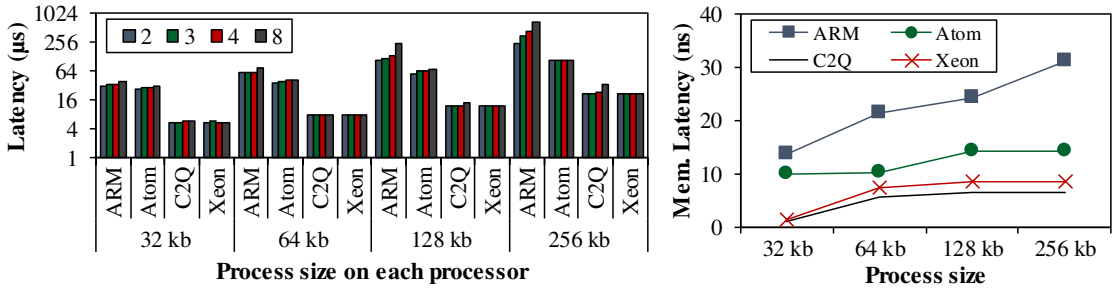
Source: The Author

8 threads on the Core2Quad instead of their OpenMP versions on the Cortex-A9 reduces execution time by 83%. However, it will increase the energy consumption by a factor of 3 times (304%).

In order to discuss how the processor and memory system influence each communication model and how they synchronize, let us first consider the programs that exchange data through shared variables. In OpenMP (Figure 4.4(a)), threads come into a busy-waiting state, accessing the shared memory repeatedly until the end of synchronization (CHAPMAN; JOST; PAS, 2007). This synchronization mechanism does not incur significant performance overhead, so all processors have similar behavior as TLP exploitation increases (as can be seen in Figure 4.5(a), the performance gap between the processors remains similar).

When it comes to energy, however, only in ARM processors the energy is reduced. For instance, while Cortex-A9 executing 8 threads saved almost 15% of energy and performed 6.15 times better than its sequential counterpart; on the Core2Quad, the energy increased 19% with similar performance improvements. This is because the energy consumed due to the extra executed instructions and accesses to the shared memory for the busy-waiting during synchronization have less influence in the ARM processors

Figure 4.6: Overhead to execute context switching on each processor



(a) Time to execute context switching

(b) Memory system latency (average) for each process size during context switching

Source: The Author

than in the Intel ones (Figure 4.5(a)). While in the ARM processors these accesses were performed in the L2 cache, in the Intel processors they occurred in the main memory.

For PThreads, (Figure 4.3(b)) the context switching imposed by the mutex influenced more the performance in ARM processors than Intel ones. As more TLP is exploited, the performance gap between these two processors increases (Figure 4.5(a)). In order to understand this behavior, lmbench (a suite to measure system performance) (MCVOY; STAELIN, 1996) was used to measure the impact of context switching on each processor. Figure 4.6(a) shows the latency of each context switching (logarithmic scale) considering processes with different parameters (which influences execution time, data size, etc), and level of TLP exploitation. One can note that context switching (saving and restoring the contents of the register file etc.) was slower on the ARM processors in all cases. This happens because the average latency to access the memory system is greater on the ARM than Intel processors, as shown in Figure 4.6(b). On the other hand, as PThreads access less the memory system during synchronization, the energy difference between all the processors remains almost the same as TLP exploitation increases (Figure 4.5(b)). This means that for HC programs parallelized using PThreads, a more robust processor is the best choice, since it provides considerable performance improvements at the same price in the energy consumption. For instance, when TLP exploitation increases from 1 to 8, the performance difference between Core2Quad and Cortex-A9 increases 33% (4.88 to 6.52 times), while the energy gap remains the same.

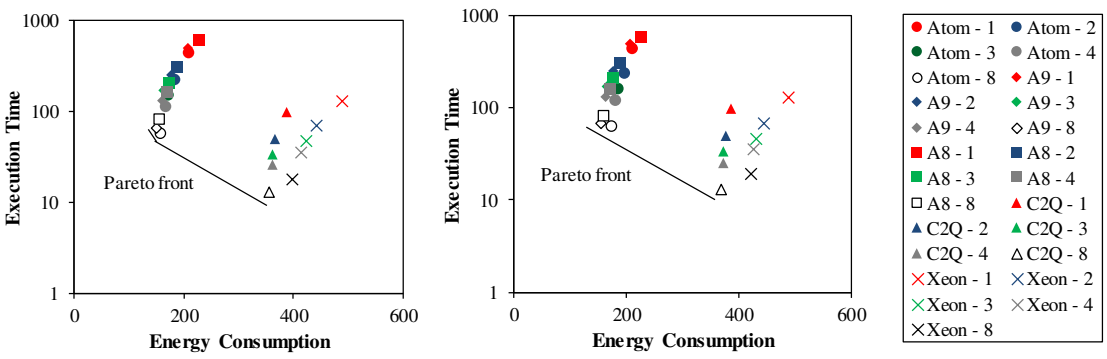
In MPI-1 and MPI-2, the amount of send/receive operations performed by each processor to exchange data impacted in different ways the performance and energy consumption. Intel processors performed better than ARM ones, but spending more energy in most cases. As the number of processes increases, the performance gains are lower in ARM processors, increasing the performance difference between them and Intel ones

(Figure 4.5(a)), and influencing the energy consumption. In such cases, as more TLP is exploited, the energy difference between ARM and Intel decreases (Figure 4.5(b)); and in the execution of 8 processes Atom got to a point where it consumed less energy than ARM processors. This scenario worsens when the processors are executing MPI-2 (Figure 4.3(d)), where the performance gains as TLP exploitation increases is even lower in ARM processors. The reason for this is that dynamic process creation adds an overhead in the runtime in terms of executed instructions, mainly due to the communication using intercoms, which affects more ARM processors than Intel (CERA et al., 2006).

4.2.1.2 Low-Communication Programs

For LC programs, the performance and energy consumption for each communication model are very similar. In this way, results are separated only by communication model: shared variables and message passing (Figure 4.7). As the applications are more CPU-bound, the impact of characteristics of each communication model on the memory system is reduced, highlighting the importance of the microarchitecture and operating frequency. In most cases, the overall performance increases in a similar ratio as more TLP is exploited (i.e., when the TLP exploitation is equal to 2, the execution time of parallel version is almost the half of sequential time, and so on). However, when the number of threads/processes is 8, performance gains are impacted by the overhead of managing the parallelization (e.g., creation/termination of threads or processes), which is greater in message passing implementations, since the cost to manage processes is greater than threads (TANENBAUM, 2007).

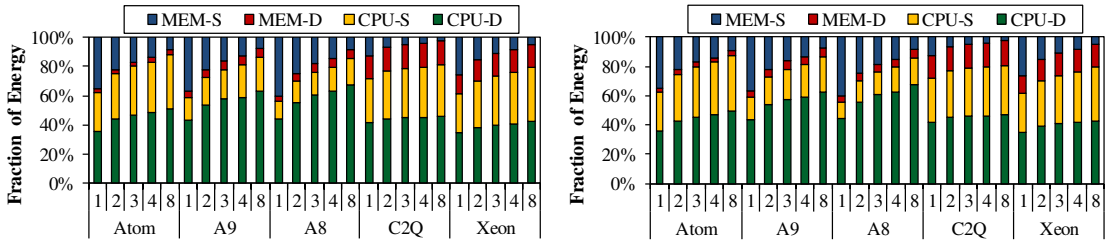
Figure 4.7: Performance (seconds) and energy consumption (joules) results for Low-Communication programs



(a) Shared Variables (b) Message Passing

Source: The Author

Figure 4.8: Fraction of energy consumed by each component - LC Applications (MEM:Memory; CPU: Processor; D: Dynamic; S: Static)



(a) Shared Variables (b) Message Passing

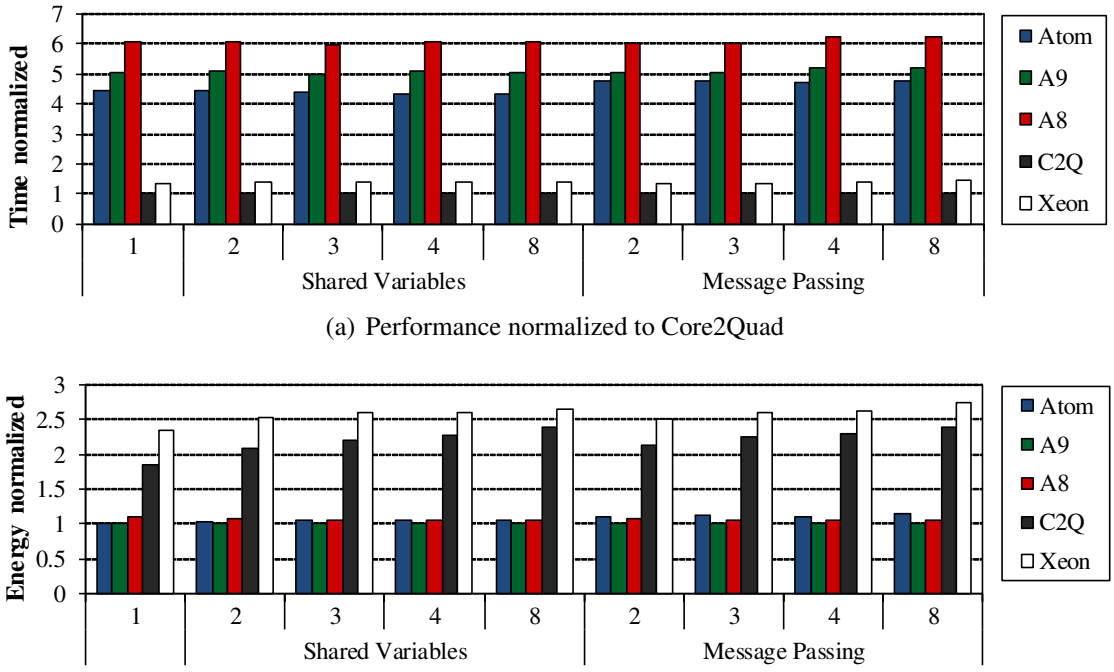
Source: The Author

All the processors perform better when they are running 8 threads/processes, and the Core2Quad continues offering the lowest execution time. Considering the best result of each processor, the performance difference between Intel processors is similar as observed for HC programs (Core2Quad is 1.37 times faster than Xeon; 4.32 times than Atom), while the performance gap between Intel and ARM diminishes in almost 13%. For instance, the difference between Core2Quad and Cortex-A9 decreases from 5.73 to 5.04 times; and from 6.87 to 6.04 times in relation to the Cortex-A8.

Unlike the HC programs, the higher the TLP exploited, the lower will be the energy consumption for all the processors regardless of the communication model. In this way, all the processors consumed less energy when executing 8 threads/processes, and in the overall Cortex-A9 is the best choice. When one compares embedded and general-purpose processors, the energy difference between them increases as more TLP is exploited (Figure 4.9(b)). When the number of threads increases, the memory system is more stressed and, therefore, spends more energy in Intel processors. As this class of applications has lower communication rate than the HC programs, it happens in a smaller proportion. Also, the performance difference between general-purpose and embedded processors decreases in almost 10% compared to the HC programs (e.g., 69% to 63% in the gap between A9 and Xeon).

In cases where the developer is looking for the best trade-off between energy and performance, there is no optimal choice. The same happens to HC programs (even though with more points and variations). As Figure 4.7(a) shows, the Pareto front consists of three points in the results for shared variables. Two of them are the best choice for energy (Cortex-A9 with 8 threads) and performance (Core2Quad with 8 threads/processes). The other one (Atom running 8 threads) is the point that improves performance over the best choice in energy with minimal impact on it. On the other hand, if the designer aims to

Figure 4.9: Results normalized to Core2Quad (performance) and A9 (energy) - LC Programs



(a) Performance normalized to Core2Quad

(b) Energy normalized to A9

Source: The Author

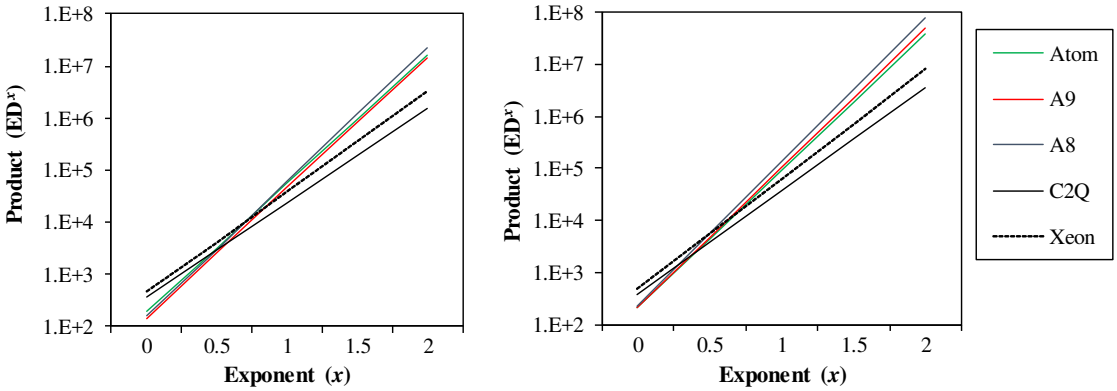
reduce the energy consumption maintaining similar execution time to the best possible, there is no satisfactory option available. For message passing (Figure 4.7(b)), the Pareto front consists of only two points: one is the best energy possible (Cortex-A9); while the other is the lowest execution time (Core2Quad). This means that for this communication model, no option can improve a metric without causing a major impact on another. For instance, if the programmer wants to improve performance with minimal impact on energy, it will reduce the execution time by only 8%, increasing energy by a factor of 15%.

There are cases in which it is possible to use embedded instead of general-purpose processors to reduce the total energy consumption with little performance degradation. In the most significant case, energy can be reduced by 70% with minimal influence on performance, if a given LC program exploits a TLP of 4 or 8 executing on any embedded multicore rather than executing on the Core2Quad and Xeon with 1, 2, or 3 threads/processes, regardless of the communication model used.

4.2.2 Energy-Delay Product

As shown in the previous section, there is no optimal combination of processor and number of threads/processes that offer at the same time the best performance with the

Figure 4.10: Impact of exponent, *x*, on product ED^x - sequential execution



(a) High Communication (b) Low Communication

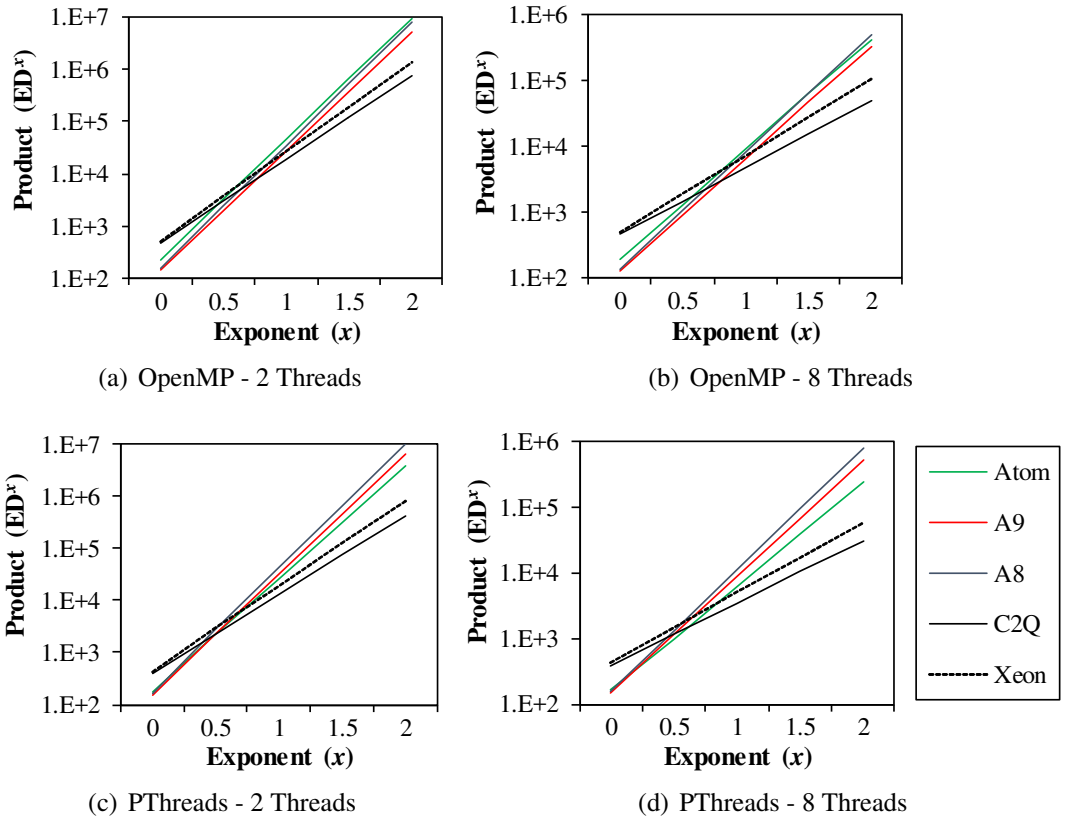
Source: The Author

lowest energy consumption. Moreover, according to their niche, companies of general-purpose processors give more importance to performance, while the embedded ones to energy. In this case, the EDP may be useful since it correlates both metrics into a unique value. By adding an exponent *x* on delay ($EDP = Energy \times Delay^x$), as the authors in (BLEM; MENON; SANKARALINGAM, 2013) have already done (but considering only sequential applications), it is possible to change the weight of delay (performance) towards energy, which would reflect the importance given to performance considering the application field.

Figures 4.10 to 4.13 show the EDP for each processor as the importance of the delay is changed. The y-axis is the product of ED^x as the exponent (x) increases in the x-axis. Figure 4.10 shows the results of the sequential executions, while Figures 4.11, 4.12 and 4.13 present the most representatives results for the parallel versions (2 and 8 threads/processes). Following the same methodology as before, HC programs are separated by PPI, while LC programs are separated by the geometric mean of the PPIs in each communication model. In overall, when both energy and performance are weighted equally (i.e., when $x = 1$), Core2Quad is the best choice (note that lower is better). Moreover, the difference between GPPs and embedded processors increases as the importance of performance towards energy increases (i.e., when the value of x increases). This reinforces the idea that GPPs are more focused on performance rather than energy, corroborating the authors research in (BLEM; MENON; SANKARALINGAM, 2013).

Let us discuss the results for the sequential versions (Figure 4.10). For HC programs (Figure 4.10(a)), Cortex-A9 provides the best ED^xP until $x = 0.6$. After that, Core2Quad outperforms all the processors. On the other hand, for LC programs (Figure

Figure 4.11: Impact of exponent, x, on product ED^x of HC programs implemented with shared variables



(a) OpenMP - 2 Threads

(b) OpenMP - 8 Threads

(c) PThreads - 2 Threads

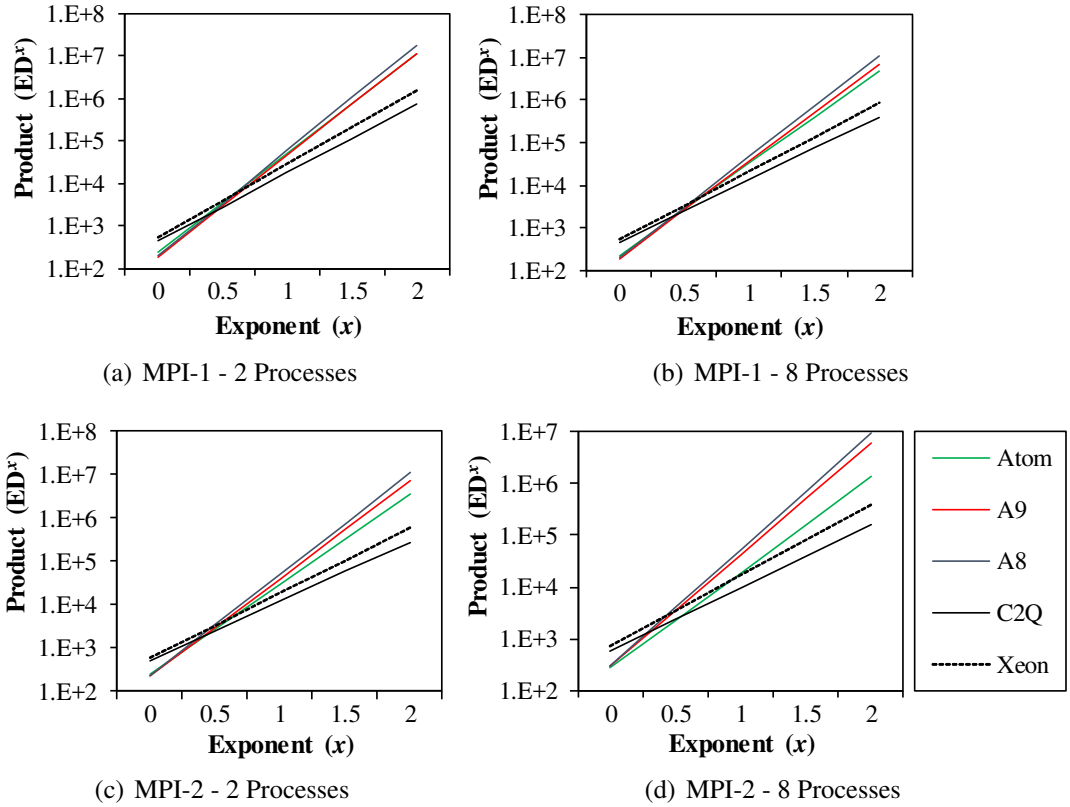
(d) PThreads - 8 Threads

Source: The Author

4.10(b)), the Cortex-A9 provides the best ED^xP until $x = 0.1$, while Atom is better when x is greater than 0.1 and lower than 0.41. After that, Core2Quad outperforms all the processors. Therefore, the Core2Quad is the best choice even in a significant part where energy is more important than performance ($0.41 < x < 0.99$). Comparing only the embedded processors, in programs where memory system is more accessed (HC programs), the ARM A9 processor has better ED^xP than the Intel Atom for any value of x. On the other hand, when the applications use more the processor rather than memory (LC programs), Atom is the best choice in most cases.

As for the parallel versions (Figures 4.11, 4.12 and 4.13), in all cases they achieved better ED^xP than their sequential counterparts, regardless the number of threads/processes and communication model used. Let us first consider the results when the processors are executing HC programs using shared variables. In OpenMP implementations (Figures 4.11(a) and 4.11(b)), Cortex-A9 has better ED^xP than the other embedded processors, no matter the value of x. In addition, as the number of threads increases, the more important must be the performance (i.e. higher values for x) so the GPPs can present better EDP than the embedded ones (see Table 4.3). For PThreads implementations, the behavior is

Figure 4.12: Impact of exponent, x, on product ED^x of HC programs implemented with message passing



(a) MPI-1 - 2 Processes

(b) MPI-1 - 8 Processes

(c) MPI-2 - 2 Processes

(d) MPI-2 - 8 Processes

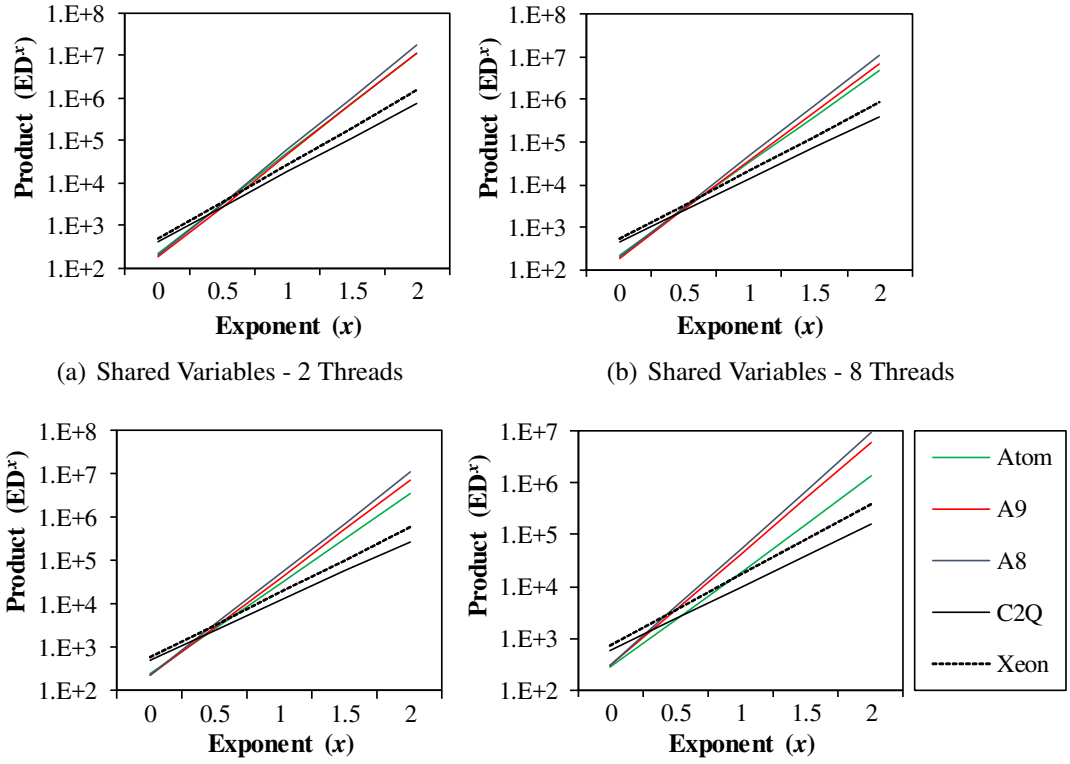
Source: The Author

different (Figures 4.11(c) and 4.11(d)): Cortex-A9 has the best EDP only when $x < 0.36$ and $x < 0.19$ for 2 and 8 threads respectively. After that, Atom is better until $x = 0.55$ and $x = 0.61$, for 2 and 8 threads, respectively. When x is greater than these values, Core2Quad outperforms all the processors.

Figure 4.12 shows the results when HC programs are implemented with message passing. Let us first discuss the MPI-1 results, where the GPPs outperforms embedded ones at a very similar value of x as the one presented in PThreads. Considering embedded processors only, the one that offers the best ED^xP changes as the number of threads increase, regardless the importance of x. In the execution of 2 processes, Cortex-A9 has the best ED^xP, while with 8 processes, Atom is the best choice. The reason for that has already discussed in Section 3.2.3: as more TLP is exploited, the performance loss and the increases in the energy consumption are more significant in ARM processors than in the Intel ones.

When it comes to the LC programs (Figure 4.13), Core2Quad continues offering the best ED^xP in most cases (mainly when performance and energy have the same weight). Comparing only the embedded processors: when they communicate through

Figure 4.13: Impact of exponent, x, on product ED^x of LC programs



(a) Shared Variables - 2 Threads

(b) Shared Variables - 8 Threads

(c) Message Passing - 2 Processes

(d) Message Passing - 8 Processes

Source: The Author

shared variables, Atom processor has better ED^xP than ARM when $x > 0.38$ and $x > 0.47$ for 2 and 8 threads respectively. On the other hand, for the results using message passing, Cortex-A9 has the best ED^xP in the execution with 2 processes regardless of the performance importance. When TLP exploitation increases to 8, Atom once more outperforms Cortex-A9 for $x > 1.35$. Therefore, there are specific scenarios where the best choice is one processor or another. When the general-purpose processors are compared, Core2Quad has better ED^xP than Xeon in all cases.

Table 4.3 shows the intersection points to figure out which is the best processor in between the intervals of x considering the charts of Figures 4.10, 4.11, 4.12 and 4.13. In overall, when performance is the most important parameter ($x > 1$), it is true that GPP is always the best choice. However, as already discussed, looking at the other side (energy), it depends on how much energy matters for the designer.

4.2.3 Influence of Static Power Consumption of Processor

In this section, we present a study regarding the influence of the static power on the total energy consumption of different multicore processors. First, we briefly discuss what

Table 4.3: Intervals of *x* where each processor is better on the ED^xP, when energy is the most important

		TLP	*Embedded Processors*			*GPPs*	
			Atom	*Cortex-A9*	*Cortex-A8*	*Core2Quad*	*Xeon*
HC		1	–	0.0 - 0.60	–	>0.60	–
LC		1	0.10 - 0.41	0.0 - 0.10	–	>0.41	–
HC Shared Variables Figure	OMP	2	–	0.0 - 0.77	–	>0.7	–
		8	–	0.0 - 0.81	–	>0.81	–
	PT	2	0.36 - 0.55	0.0 - 0.36	–	>0.55	–
		8	0.19 - 0.61	0.0 - 0.19	–	>0.61	–
HC Message Passing Figure	MPI-1	2	–	0.0 - 0.56	–	>0.56	–
		8	0.0 - 0.61	–	–	>0.61	–
	MPI-2	2	–	0.0 - 0.42	–	>0.42	–
		8	0.0 - 0.49	–	–	>0.49	–
LC Figure	SV	2	0.37 - 0.48	–	–	>0.49	–
		8	0.48 - 0.56	0.0 - 0.48	–	>0.56	–
	MP	2	–	0.0 - 0.42	–	>0.42	–
		8	–	0.0 - 0.49	–	>0.49	–

Source: The Author

static power is and how it can affect the energy consumption of parallel applications. Next, the methodology used in this experiment is presented, followed by a discussion about the results achieved.

As already discussed in Section 2.4.2, there are two main components that constitute the power used by a CMOS integrated circuit: dynamic and static. The former is the power consumed while the inputs are active, with capacitance charging and discharging, being directly proportional to the circuit switching activity. The static power derives from the length of the transistor channel as well as the doping level and gate thickness. For instance, increasing doping levels allows higher on current for faster transitions but cause greater leakage. Therefore, companies can tune the circuits during the manufacturing process to be faster and consume more static power or vice-versa (NOSE; SAKURAI, 2000). In some cases, the static power in the processor may represent up to 40% of the total energy consumption (NOSE; SAKURAI, 2000), (KONTORINIS et al., 2009) , and (ESMAEILZADEH et al., 2012).

TLP exploitation in multicore systems affects dynamic and static power consumption in different ways. The former will most likely increase as the number of threads increase, since additional memory accesses and executed instructions are necessary for synchronization and data exchange. On the other hand, memory will consume less static power because it will be powered for a shorter period because of overall performance improvements. However, since parallelization is not perfect, some threads distributed

Table 4.4: Respective energy consumed per instruction and static power when changing the importance of static power of processor

		10%	**20%**	**30%**	**40%**
Atom	*Static Power* (W)	0.242	0.484	0.726	0.968
	Energy per Instruction (nJ)	0.448	0.391	0.335	0.276
Cortex-A9	*Static Power* (W)	0.125	0.250	0.375	0.500
	Energy per Instruction (nJ)	0.291	0.237	0.183	0.129
Cortex-A8	*Static Power* (W)	0.085	0.170	0.255	0.340
	Energy per Instruction (nJ)	0.338	0.266	0.195	0.124
Core2Quad	*Static Power* (W)	2.195	4.390	6.585	8.780
	Energy per Instruction (nJ)	1.267	1.126	0.985	0.845
Xeon	*Static Power* (W)	1.848	3.696	5.544	7.392
	Energy per Instruction (nJ)	1.419	1.261	1.103	0.946

Source: The Author

over the processors will take longer to execute than others. In such cases, the sum of all amounts of static power consumed by all the processors will be larger than its sequential counterpart.

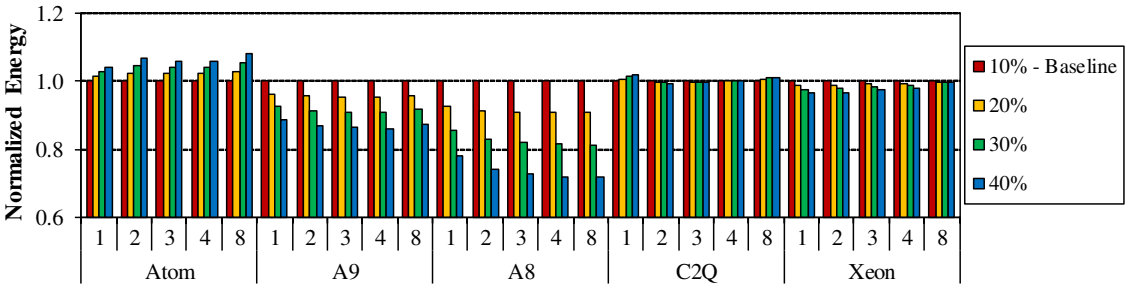
Considering the aforementioned scenario, this section aims to investigate the influence of the static power consumption of the processor on parallel applications in multicore systems. We consider four different proportions of static power in respect to the total power consumption of the processor obtained from (BLEM; MENON; SANKARALINGAM, 2013) and CACTI 5.1 [5]: 10%, 20%, 30%, and 40%. Table 4.4 shows the static power and the energy consumption per instruction when different ratios of static/dynamic power are considered. When the proportion of static power increases in respect to the total power consumption of the processor, dynamic (energy per executed instruction) will decrease in the same amount; therefore, total energy consumption will always be the same. This analysis involves power in the core only: the ratio of static/dynamic power consumption of the memory system is not changed.

The results consider the geometric mean of each communication model, since the behavior is very similar between the interfaces that implement them (standard deviation lower than 1%). Figures 4.14 and 4.15 show the impact of static power for each communication model on each processor in HC and LC programs, respectively. All the charts consider the results when the static power of the processor is fixed to 10% as baseline, and show the impact on the total energy consumption when it is changed to 20%, 30%, and 40%. Therefore, values lower than "1" mean that there are energy savings.

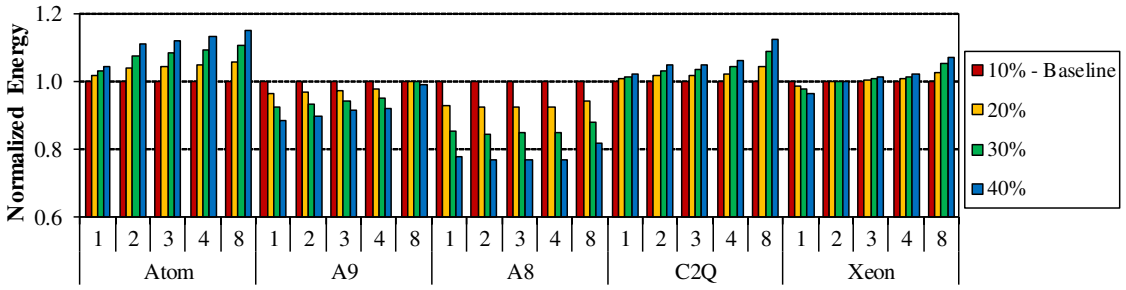
In overall, the architecture of the processors influences how the static power impacts the total energy consumption. In Intel processors, increasing the importance of

[5] Available at: http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html

Figure 4.14: Impact on the total energy consumption when the static power of processor varies from 10% - HC Programs



(a) Shared Variables



(b) Message Passing

Source: The Author

static power will also increase energy consumption, while one can observe the opposite behavior for ARM processors. The amount of TLP also changes the variation ratio: the more TLP is exploited, the more significant the impact when changing the amount of static power on the total energy consumption.As the parallelization is not perfect, the sum of the static power consumed by all cores is larger than if it was sequentially executed. It means that the static power consumed by the processors starts to be more important as more TLP is exploited.

Let us first discuss the results of the Intel processors executing HC programs (Figure 4.14). In such cases, the effect of changing the proportion of static power is negligible in most cases. To better understand that, let us consider the Table 4.5 and 4.6. The former presents the number of executed instructions by core per second. To compare only the behavior of each PPI on each processor, Table 4.6 depicts the number of instructions executed per second in the parallel version by its sequential counterpart, the bigger the result, the closer it is to the behavior of its sequential version, meaning that the processor will be executing more instructions instead of waiting for sync and data exchange.

When doing this calculation, we can note that the LC programs have bigger values than HC programs – which means that, even though they execute less instructions per second (Table 4.5) because of the kind of application, their parallel versions proportionally

Table 4.5: Number of executed instructions by core per second

Comm. Model	TLP	HPC Programs					LC Programs				
		Atom	A9	A8	C2Q	Xeon	Atom	A9	A8	C2Q	Xeon
Shared Variables	2	837	899	749	4018	3286	432	744	620	1916	1441
	3	875	893	743	3969	3197	427	747	623	1880	1428
	4	887	882	735	3924	3136	432	718	598	1849	1421
	8	835	840	700	3770	2973	431	717	598	1838	1394
Message Passing	2	720	807	672	3376	2945	410	745	621	1955	1525
	3	696	754	628	3347	2842	405	738	615	1932	1502
	4	671	729	607	3262	2780	407	708	590	1911	1462
	8	640	599	499	2759	2440	404	702	584	1892	1365
Sequential		884	905	754	3625	3342	419	733	611	1936	1541

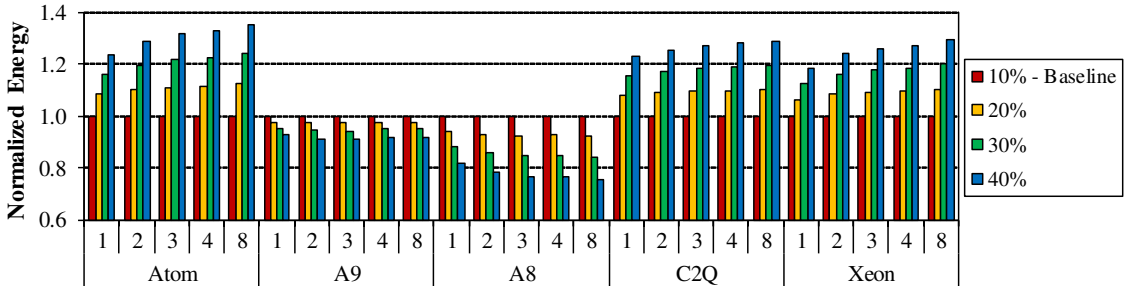
Source: The Author

execute more instructions per second than the HC applications; which shows that they spend less time waiting for data exchange or sync. This can be observed for the message passing in Table 4.5 and 4.6: the higher the amount of executed processes, the higher the load imbalance, and the smaller is the number of executed instructions per second. In this case, static power plays an important role. When it comes to the ARM processors executing HC programs (Figure 4.14), the results show that in all cases, increasing static power of the processor reduces the total energy consumption. The reason for this is that the reduction in the dynamic power consumption is greater than the increase provided by the change in the value of the static power in the processor.

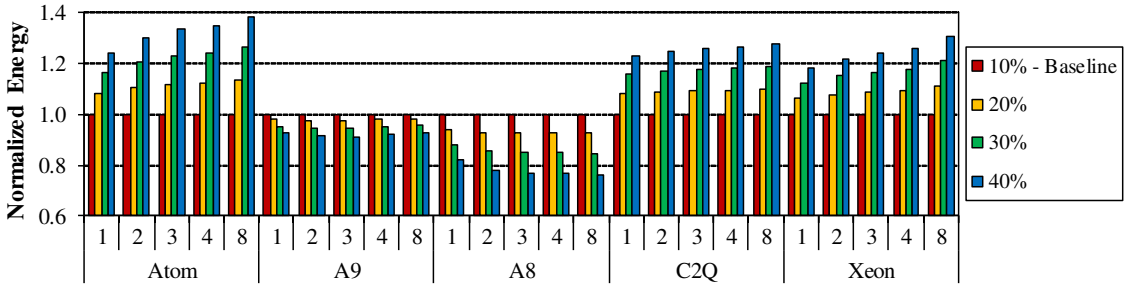
For the LC programs (Figure 4.15), the impact of changing the amount of static power is greater than the observed for the HC programs. In addition, the same behavior is observed regardless the communication model used. Considering the Intel processors, the higher the TLP exploitation, the greater the impact of increasing the static power of the processor. In the sequential version, when the static power of the processor is changed from 10% to 40%, the total energy consumption increases almost 24% on both Atom and Core2Quad, and 18% in the Xeon processor; while in the execution with 8 threads/processes, this difference increases even more: 35% and 38% in the Atom, when using shared variables and message passing respectively; 28% and 30% in the Core2Quad and Xeon respectively, regardless the communication model. As for ARM processors, which have a high number of executed instructions per second (see Table 4.5), changing the static power of the processor from 10% to 40% results in energy savings in all cases: almost 8% in the Cortex-A9, and 24% in the Cortex-A8.

Analyzing the whole scenario, Intel and ARM processors have different behaviors when the proportion of static power is changed in respect to the total power consumption. In the former, regardless of the kind of application and the communication model used,

Figure 4.15: Impact on the total energy consumption when the static power of processor varies from 10% - LC Programs



(a) Shared Variables



(b) Message Passing

Source: The Author

Table 4.6: The proportion of the number of executed instructions by core per second in the parallel versions regarding its sequential version

Comm. Model	TLP	HPC Programs					LC Programs				
		Atom	A9	A8	C2Q	Xeon	Atom	A9	A8	C2Q	Xeon
Shared Variables	2	0.95	0.99	0.99	1.11	0.98	1.03	1.02	1.01	0.99	0.94
	3	0.99	0.99	0.99	1.09	0.96	1.02	1.02	1.02	0.97	0.93
	4	1.00	0.97	0.97	1.08	0.94	1.03	0.98	0.98	0.96	0.92
	8	0.94	0.93	0.93	1.04	0.89	1.03	0.98	0.98	0.95	0.90
	AVG	0.97	0.97	0.97	1.08	0.94	1.03	1.00	1.00	0.97	0.92
Message Passing	2	0.81	0.89	0.89	0.93	0.88	0.98	1.02	1.02	1.01	0.99
	3	0.79	0.83	0.83	0.92	0.85	0.97	1.01	1.01	1.00	0.97
	4	0.76	0.80	0.80	0.90	0.83	0.97	0.97	0.97	0.99	0.95
	8	0.72	0.66	0.66	0.76	0.73	0.96	0.96	0.96	0.98	0.89
	AVG	0.77	0.79	0.79	0.88	0.82	0.97	0.99	0.99	0.99	0.95
Sequential		1	1	1	1	1	1	1	1	1	1

Source: The Author

keeping static power of the processor as low as possible saves energy in most cases, even though at different levels. On the other hand, for ARM processors, the higher the static power, the greater the reduction in energy consumption.

4.3 Discussion

This Chapter performed a static exploration for optimal combinations of processors, communication models, and TLP exploitation to reach the best results in performance, energy, and EDP. A great number of variables were considered: 5 multicore processors with different microarchitectures and ISAs; 14 parallel benchmarks classified according to the communication rate; four parallel programming interfaces classified into two classes of communication models; different levels of TLP exploitation; and four different levels of static power of the processor. We demonstrated that even though there are combinations with the best performance and the lowest energy consumption, there is no single one that offers the best result for both at the same time. However, we found some significant results, summarized next.

Let us first discuss performance and energy (Section 4.2.1), in which the most robust processor (Core2Quad) achieved the lowest execution time, while the embedded processor Cortex-A9 consumed less energy in all cases. For HC applications, the PPIs matter: PThreads has shown to be the best choice for all Intel processors (GPP or embedded), since it provides considerable performance improvements over the others at the same price of energy consumption as the sequential version. On the other hand, when exploiting parallel loops, OpenMP is better for ARM processors, since the impact of the busy-waiting mechanism is lower on these processors than on the Intel ones. In overall, MPI is the worst choice for all the processors, presenting poor scalability: as TLP exploitation increases, performance gains are limited by its message based communication; and energy consumption increases when compared to its sequential version. It was expected that MPI would perform worse than OpenMP and PThreads in HC applications on shared memory environments. This behavior happens because each communication between MPI processes involves an additional cost related to the construction/deconstruction of the message as well the message transmission.

There are different situations when analyzing the Pareto front for all the cases. In OpenMP applications, it contains only two points: the best result for performance (Core2Quad running 8 threads) and the best for energy consumption (Cortex-A9, also executing 8 threads). There is no option that will not influence considerably a metric to improve another. For the other PPIs, there are more points to be explored, and the impact on a metric to improve another is minimal. For instance, in MPI-1 applications with 8 processes, it is possible to reduce the energy consumption in 15% without impact on performance by changing processors (Core2Quad instead of Xeon).

The scenario is different for LC benchmarks. For those, what matters is the communication model rather than a specific PPI. Since they are more CPU-bound, how the processor can exploit ILP and its operating frequency gain in importance. Regardless of the PPI, performance increases and energy reduces as the TLP increases, resulting in better EDP. Therefore, even though these applications scale better than HC ones, the design space is more restricted, offering less opportunities for optimization. The Pareto front has fewer points and alternatives to optimize a metric with minimal impact on another; and the differences between Intel and ARM processors are subtler.

When it comes to ED^xP (energy-delay^{x}product, depicted in Section 4.2.2), in all cases (no matter the processor or PPI used) the parallel versions were better than their sequential counterparts, if one considers that performance has the same weight as energy ($x = 1$); and the difference in EDP between a parallel version and its sequential counterpart increases as more importance is given to performance. The Core2Quad processor has better ED^xP in this case, regardless of the communication model used or the number of threads/processes.

In general, GPPs are always the best choice when targeting performance only. However, looking at the other side (energy), it depends on how much energy matters to the designer. For instance, in HC programs using PThreads, three processors have the best ED^xP according to the importance of energy: Cortex-A9 for $x < 0.36$; Atom for $0.36 < x < 0.55$; Core2Quad for larger values of *x*. In some scenarios, Core2Quad is the best choice even if energy is more important ($x < 1$). However, as the number of threads increase, more importance to performance must be given (the *x* value must get closer to 1) so the Core2Quad still presents the best ED^xP.

The PPIs influence EDP in different aspects. For OpenMP, energy consumption in the memory system is very important, because of the busy-waiting. For PThreads, on the other hand, a more robust processor will decrease context switching time. For the MPI versions of the applications, as more threads execute, EDP in general worsens for ARM processors and improves for Intel ones, since the impact of the communication on the former is more evident.

In Section 4.2.3, we demonstrated that processors present different behaviors when tuning the values of energy resultant from the static and dynamic power of the processor. For Intel processors, by keeping the static power of the processor as low as possible, more energy will be saved. In the most significant case, it is possible to save 38% of energy if the hardware designer keeps the static power at 10% instead of 40%. On the other hand, the opposite happens for the ARM processors, where the higher the static

power, the lower the total energy. For instance, it is possible to save 28% of energy if the static power represents 40% of the total energy, instead of 10%. The number of executed threads also influences results: as more TLP is exploited, more impact it has on tuning the static power. These results are directly related to how long the processor spends time synchronizing and communicating. Therefore, HC applications are more susceptible to changes in static power.

4.4 The Importance of Improving OpenMP Applications

The previous sections showed that there is an extensive design space exploration in parallel computing regarding the choice of the configuration of PPIs and number of threads that offer the best result for a given metric. It reinforces the needed for developing a mechanism to optimize the process of finding the best configuration for executing parallel applications. However, the development of a single mechanism that comprises all the PPIs discussed in this section (OpenMP, PThreads, and MPI) is not feasible due to the way the parallelism is exploited in each one of them (thread and process management, workload distribution, communication model, etc.). Therefore, when proposing a novel mechanism, it needs to reach as many commercial and scientific parallel applications as possible.

In this context, we chose to develop a mechanism that improves the parallel execution behavior for applications implemented with OpenMP. The following reasons were taken into account:

- OpenMP consists of a set of compiler directives, library functions, and environment variables that eases the developer burden of creating and managing threads in code. Therefore, extracting parallelism using OpenMP usually requires less effort when compared to other PPIs, making it more appealing to software developers (S. et al., 2011)(AJKUNIC et al., 2012).

- OpenMP is widely used, and there are many parallel benchmarks implemented with it, but more importantly, applications that comprise different niches and areas (Table 4.7): As an example, NAS Parallel Benchmark, SPEComp, Linpack, Parboil, Rodinia, and Princeton Application Repository for Shared-Memory Computers (PARSEC) are used to compare different processor architectures and systems. In the same way, hydrobench, OpenLB, GROMACS, and LULESH are used to simulate a wide variety of science and engineering problems.

Table 4.7: Parallel benchmarks widely used

Benchmarks	**OpenMP**	**PThreads**	**MPI**
NAS Parallel Benchmark (BAILEY et al., 1991)	x		x
SPEComp2001 and 2012 (ASLOT et al., 2001)	x		
PARSEC (BIENIA et al., 2008)	x	x	
OpenLB (HEUVELINE; LATT, 2007)	x		x
Linpack (DONGARRA, 1988)			x
Lonestar (KULKARNI et al., 2009)		x	
LULESH (KARLIN; KEASLER; NEELY, 2013)	x		
HydroBench (available at https://github.com/HydroBench/Hydro)	x		x
Rodinia (CHE et al., 2009)	x		
Parboil (STRATTON et al., 2012)	x		
Parmibench (IQBAL; LIANG; GRAHN, 2010)		x	
GROMACS (ABRAHAM et al., 2015)	x		x

Source: The Author

- OpenMP is more suitable for mechanisms in which the goal is to provide the maximum transparency as possible to the programmer, since the whole process of thread management and workload distribution are done by functions implemented in the OpenMP library, in opposite to other PPIs (e.g., PThreads, MPI, TBB).

- In addition to these reasons, the comprehensive study presented in this Chapter has shown that OpenMP has remarkable results on performance, energy, and EDP regardless of the processor microarchitecture and benchmarks class when compared to the other PPIs. As an example, while PThreads has good behavior on Intel processors and poor performance on ARM processors due to synchronization mechanism, OpenMP has good results in both processors. This behavior can be observed on Figure 4.3.

5 OPTIMIZATION OF OPENMP APPLICATIONS

In this chapter, the main contributions of this thesis are presented, discussed, and validated. The approaches proposed in this thesis are divided into two sections:

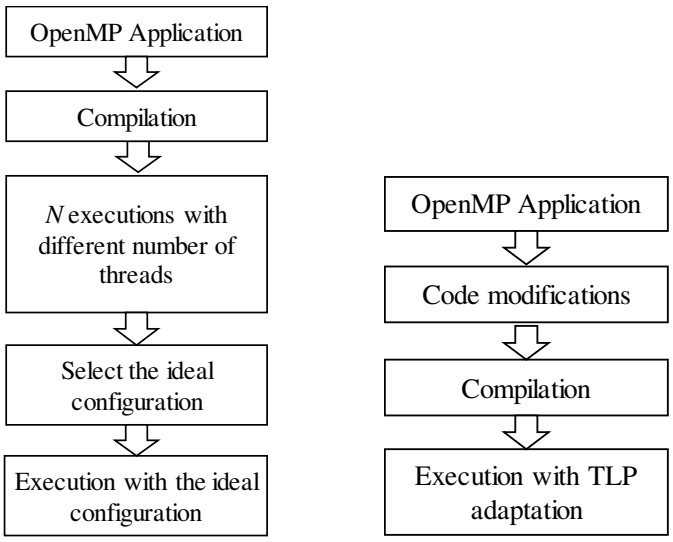
- Section 5.1 presents LAANT, a library to automatically adapt the number of threads of OpenMP applications. It is capable of finding at run-time the ideal number of threads for each parallel region of the application, learning the best number of threads as the application executes, and resulting in significant improvements in EDP with an almost negligible overhead. The proposed process is completely automatic to the developer. It can be applied to any parallel application developed with the OpenMP interface and compiled with GCC or G++ compilers, by annotating code in the parallel regions, which were already identified by OpenMP directives. LAANT can optimize the parallel regions for different metrics, such as EDP, performance, energy consumption, among others. This was the first effort of this thesis to develop a mechanism that is completely transparent to the user.

- Section 5.2 describes Aurora, an extension of LAANT. In the same way, it is a mechanism that automatically finds, at runtime and according to a given metric defined a priori by the user, the ideal number of threads for each parallel region of any OpenMP application. Moreover, Aurora can re-adapt at the event of a change in the behavior of a particular parallel region during program execution. Aurora was built on top of the original OpenMP library, being completely transparent to both designer and end-user: given an OpenMP application binary, Aurora runs on it without any code changes. Therefore, existent OpenMP applications do not need to be annotated, recompiled or pass through any code transformation. Such transparency is achieved by redirecting the calls originally targeted for the dynamically linked OpenMP library to Aurora. This re-targeting is configured by setting an environment variable in the Operating System.

5.1 LAANT: A Library to Automatically Adjust the Number of Threads for OpenMP Applications

Figure 5.1(a) shows the usual way of finding the best number of threads to run a parallel application. The source code is compiled and executed N times with a different

Figure 5.1: Adaptation of OpenMP Applications



(a) Brute force (b) LAANT

Source: The Author

number of threads, where N is the number of available cores in the processor microarchitecture. After this period, the best configuration is selected, and the next executions will be performed with the configuration found in this step. For example, if the processor has 16 cores, the application would be executed 16 times (from 1 to 16 threads) to find the number of threads that offer the best result. However, if there is any change in the application behavior (e.g., input set size) or the execution environment, the executions must be performed again. Moreover, whether we consider that the applications may have different parallel regions with distinct behaviors, the DSE exponentially increases.

Therefore, in order to cope with the challenge of selecting the best number of threads to execute an OpenMP application, we developed LAANT. It is a library to automatically adjust the number of threads for OpenMP applications. The overall organization of LAANT is depicted in Figure 5.1(b). Given an OpenMP application, functions of LAANT are inserted before and after parallel regions already identified by the programmer. Then, the application is compiled, and at runtime, LAANT automatically finds the best number of threads to execute each parallel region.

The remainder of the Section is organized as follows. Section 5.1.1 discusses the details regarding the LAANT implementation and its use in OpenMP applications. Second, Section 5.1.2 presents the methodology used to evaluate LAANT. It also discusses the results obtained when comparing LAANT to the OpenMP dynamic feature (that dynamically changes the number of threads as the application executes) and the standard

way that OpenMP applications are executed (that uses the maximum number of threads along with all the execution). Finally, Section 5.1.3 draws the final considerations.

5.1.1 LAANT Implementation

LAANT uses a heuristic based on a hill-climbing algorithm to find the optimal number of threads for parallel regions of OpenMP applications. Each one of these regions can be optimized for different metrics, such as performance, energy consumption, EDP, resource usage, among others. In order to get information for calculating each metric, LAANT uses functions provided by the OpenMP and processors. For instance, to obtain the execution time of each parallel region, LAANT uses the $omp\_get\_wtime()$ function, provided by the OpenMP. On the other hand, energy is obtained directly from hardware counters present in modern processors. In the case of Intel processors, the RAPL library (HäHNEL et al., 2012) is used to get energy and power consumption of the CPU-level components. As for AMD processors, another library could be used: Application Power Management (HACKENBERG et al., 2013a). LAANT is divided into two parts: the first one contains the functions provided to the developer, and the second part is the heuristic to find the ideal number of threads for parallel regions.

5.1.1.1 Using LAANT on OpenMP Applications

LAANT consists of three main functions: *initLaant*, *startKernel*, and *endKernel*. These are inserted either manually or by a script (provided by LAANT) into any OpenMP application, whose the parallel regions were already identified by $\#pragmas$, as shown in line 6 of Figure 5.2.

- The *initLaant* function initializes the structures and variables used to control the hill-climbing algorithm, and the libraries used to collect information from parallel regions behavior. Thus, it is inserted at the beginning of the main function.
- The *startKernel* function sets the number of threads that executes each parallel region based on the current state of the hill-climbing algorithm. It is done through the $omp\_set\_num\_threads$ function, provided by the OpenMP. Also, *startKernel* initializes the counters for execution time, energy, and EDP of the parallel region. This function is inserted before a parallel region, as shown in line 5 from Figure 5.2.

Figure 5.2: Using LAANT on OpenMP Applications

```
1.  int main ( ){
2.     initLaant();
3.     for(int iter=0; i<100; iter++){
4.           /* Sequential region */
5.           startKernel();
6.           #pragma omp parallel
7.           {
8.              /* Parallel region */
9.            }
10.          endKernel();
11.    }
12. }
```

Source: The Author

- Finally, the *endKernel* function is inserted after the parallel region to get its execution time, energy, and the EDP. With this information, it performs one step of the hill-climbing algorithm to find the number of threads that will execute this parallel region in the next iteration.

The functions *startKernel* and *endKernel* run until the ideal number of threads is found. Once found, they run periodically to verify changes in the behavior of parallel region, or when there are variations in its workload.

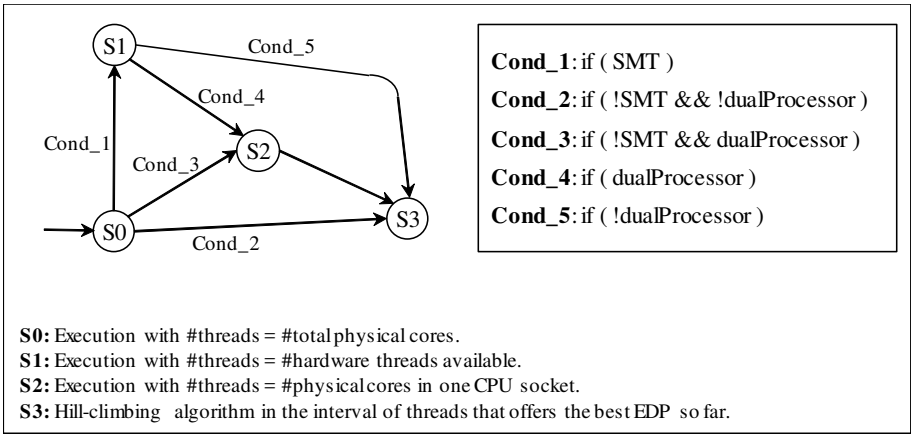
5.1.1.2 Automatically Adjusting the Number of Threads

LAANT uses a simple Finite State Machine (FSM) to implement a heuristic based on a hill-climbing algorithm. The heuristic is divided into two phases. The main states of the FSM are described in Figure 5.3:

- The $S0$ state comprises the execution of the parallel region with the number of threads equal to the number of physical cores of the platform. If the platform has SMT technology or it is a dual-processor machine, then the next state will be $S1$ or $S2$. Otherwise, the hill-climbing algorithm will be performed in the $S3$ state, starting at the number of threads executed in this state ($S0$).

- If the platform has SMT technology, then in the $S1$ state, the parallel region run with the number of hardware threads available.

Figure 5.3: Main states of the FSM



Source: The Author

- If the machine is a dual-processor, then in the $S2$ state, the parallel region is executed with the number of threads that matches the number of physical cores of one CPU socket.

These decisions are made based on previous knowledge: several executions and experiments have been done on different platforms, with distinct number of physical cores, CPU sockets and maximum number of supported threads. This information is part of the LAANT and used only internally, and does not need to be updated by the designer or user. As an example, let us consider a dual processor with six physical cores on each CPU socket, and with SMT technology, totalizing 24 hardware threads available. In the $S0$ state, it will execute 12 threads (#physical cores). In the $S1$ state, it will run 24 threads (#hardware threads available). As for in the $S2$ state, the parallel region will be executed with 6 threads (#physical cores in one CPU socket). Once these executions are done, in the $S3$ state the hill-climbing algorithm starts its execution in the interval of threads defined as follows:

- If the application has a high degree of TLP exploitation, the number of threads that offer the best result will be closer to the maximum number of threads available (LEE et al., 2010). Then, the hill-climbing algorithm will be guided to a number close to this. In such example, the search will focus on the interval between 12 and 24 threads.

- If the application has a low degree of TLP exploitation due to data synchronization and communication between threads, the best number of threads will be closer to the minimum number of threads (SULEMAN; QURESHI; PATT, 2008). Thus, the

Table 5.1: Main characteristics of the benchmarks

Characteristics	Benchmarks
Changes in the workload of the parallel region at runtime	Fast Fourier Transform (FFT)
More than one parallel region, each one with a different behavior	Block tri-diagonal solver (BT), Lower-upper gauss-seidel solver (LU), Poisson equation (PO), LULESH 2.0
One of more parallel regions with the same behavior, and with no changes in the workload at runtime	Hotspot, Scalar penta diagonal solver (SP)
High degree of TLP exploitation	Conjugate Gradient (CG), Discrete 3D Fast Fourier Transform (FT)

Source: The Author

hill-climbing algorithm will be applied to a number close to this minimum, wherein the example the search will focus on the number near to 6 threads.

- Finally, if the application has a medium degree of TLP exploitation, the search algorithm will operate between the minimum and the maximum number of threads. In the example with 24 cores, the search will be performed on the number of threads near to 12.

In order to avoid minimal locals and plateaus during the search, and so wrongly converging to an incorrect number of threads, the hill-climbing algorithm uses lateral movement. Once the number of threads that offers the best result (i.e., performance, energy, or EDP) is found, the parallel region will execute with this number until the periodical executions to verify changes in the behavior of parallel region, or when there are variations in its workload.

5.1.2 Evaluation and Discussion

5.1.2.1 Methodology

Nine applications already parallelized with OpenMP written in C and C++ from assorted benchmark suites and domains were chosen. They were divided into four categories regarding the behavior characteristics of the parallel regions present on each application, as shown in Table 5.1.

Table 5.2: Main characteristics of the processors

	Core i7	**Xeon E5-2630**	**Xeon E5-2650**
Microarchitecture	Skylake	Sandy Bridge	Sandy Bridge
Frequency	3.4 GHz	2.3 GHz	2.0 GHz
#CPU Sockets	1	2	2
#Total Cores	4	12	16
#Hardware Threads	8	24	32
Linux Kernel	4.4.0-21	3.13.0-85	3.16.0-70

Source: The Author

The experiments were performed on three different multicore processors, each one able to execute a different number of threads, as shown in Table 5.2. Also, each core can execute one or two threads in hardware each. The benchmarks were executed with two different input sets: B and C for the NAS applications; and small and medium for the other benchmarks. The applications were compiled with GCC and G++ 5.3, using the optimization flag $-O2$ and $-O3$ (as $-O2$ had better results, they are presented). The OpenMP distribution used was version 4.0. The results presented in the next session are an average of ten executions with a standard deviation lower than 0.5%.

When designing parallel applications, the usual is to execute with the maximum number of hardware threads available in the system (LEE et al., 2010). Therefore, we use this scenario as the comparison **Baseline**. We also ran the same benchmarks with an OpenMP dynamic feature ($OMP\_Dynamic$), which dynamically adjusts the number of threads of a parallel region as the parallel region is being executed (CHAPMAN; JOST; PAS, 2007), aiming to make the best use of system resources, such as memory and processor. This feature is enabled by using the environment variable $OMP\_DYNAMIC$ or through the insertion of the $omp\_set\_dynamic()$ in the source code (CHAPMAN; JOST; PAS, 2007). LAANT, on the other hand, executes its adaptation algorithm after the parallel region and targets different optimization metrics. In this section, we present the behavior of LAANT when the EDP is configured as optimization metric.

Figure 5.4 presents EDP results for LAANT and $OMP\_Dynamic$ for each application, along with their geometric mean (gmean) for the entire benchmark set on the three multicore systems. EDP is normalized considering the baseline explained in the previous section (represented by the black line). Table 5.3 depicts the Best Number of Threads (BNT) found by LAANT that offers the best EDP to execute each parallel regions of each application, represented by the numbers inside the parenthesis. For instance, the application Conjugate Gradient (CG) has two parallel regions, in which – for the Input Set B – the optimal number of threads for the first region is eight and for the second region is

three. In all cases, LAANT found the best number of threads to execute each parallel region. This was validated by comparing the numbers found by LAANT with an exhaustive search using data from the execution of each parallel region with all possible numbers of threads (from 1 to the maximum number of cores). The results are discussed in the next subsections, considering the **Baseline** and the $OMP\_Dynamic$ separately.

5.1.2.2 LAANT versus Baseline

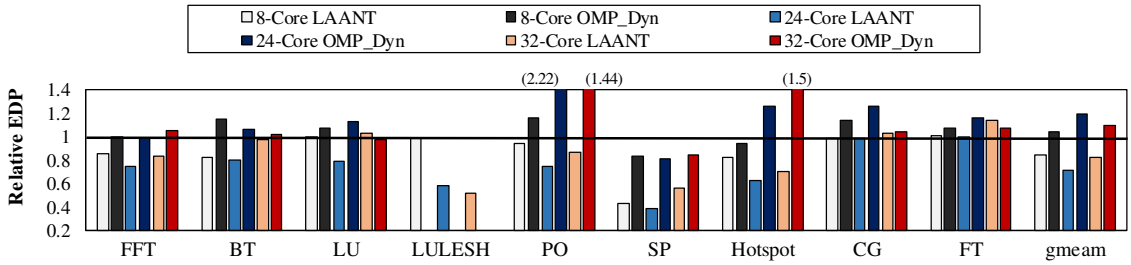
Comparing to the **Baseline** and considering the geometric mean (gmean) of the entire benchmark set, LAANT can reduce the EDP in up to 29% on the 24-core system running the small input set (Figure 5.4(a)). The smallest gains are achieved in the 32-core system running the medium input set (Figure 5.4(b)) –, and, even in this case, LAANT presents 15% of EDP reductions.

LAANT automatically detects changes in the workload of the parallel region and correctly finds the appropriate number of threads to run this region. For instance, when the workload of the FFT application changes at runtime, the number of threads that offers the best EDP also changes. Let us consider the FFT execution with a medium input set on the 24-core system (Table 5.3). On its first parallel region execution, the best EDP is initially achieved by executing 20 threads. However, as the workload of this parallel region increases, the synchronization and communication between threads also increase, saturating the shared resources (SULEMAN; QURESHI; PATT, 2008). Therefore, the ideal number of threads end up reducing to 12, and then 6, when it finally stabilizes (as shown in Table 5.3). LAANT detects these workload changes and adjusts the number of threads, which provides 31% of EDP reductions.

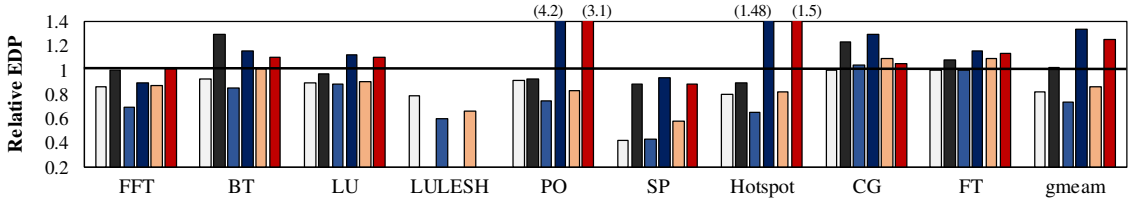
In applications with more than one parallel region, LAANT finds the ideal number of threads to execute each one of them. For instance, let us consider the execution of LULESH with the small input set on the 32-core system. LULESH has three parallel regions (Table 5.3), in which the first region is better executed with 14 threads. However, the regions 2 and 3 present the best EDP with 10 and 16 threads, respectively. By finding these numbers at runtime, LAANT reduced the EDP by 49%. The same behavior happens with Block Tri-diagonal solver (BT), Poisson, and Lower-Upper Gauss-Seidel solver (LU) applications, but at different EDP improvement ratios.

Table 5.3 also presents the percentage of time and energy (α and β, respectively) that LAANT spent in the search for the best number of threads relative to its total time and energy – this we call the overhead for using LAANT. The lowest overhead occurs in

Figure 5.4: Relative EDP of LAANT and OMP_Dynamic compared to the baseline (black line)



(a) Benchmark execution with the **small input**



(b) Benchmarks execution with the **medium input**

Source: The Author

applications that possess only one parallel region, such as Scalar Penta-diagonal solver (SP) and Hotspot. For example, the overhead for the Hotspot application on the 24-core system was of only 0.1% of the total time and energy. This happens because the search algorithm used to find the best number of threads (12) can do so at only four steps: by running with 12, 24, 6 and 10 threads. Besides the low overhead, LAANT also provided huge EDP reductions, such as 61% when executing the SP with the small input set on the 24-core system.

The highest overhead presented by LAANT was in applications which the best EDP is achieved with either the maximum number of threads or a number near it, such as discrete 3D fast Fourier Transform (FT) and CG (Table 5.3). For the FT application with the medium input set (C) on the 32-core system, LAANT increased the execution time by 8.1% and energy by 1.6%. This happens because the execution of each parallel region with a lower number of threads during the search results in increased execution time and energy consumption for the whole application. For instance, to find that 32 threads provided the best EDP for FT, the parallel regions were executed with 16, 32, 8, 24, 28, 30, and 31 threads. Although this is the worst case for LAANT, this unnecessary overhead can be reduced by enhancing the search algorithm to minimize the training overhead on applications of high degree of TLP.

5.1.2.3 LAANT versus OMP_Dynamic

LAANT outperforms the OpenMP dynamic feature in the great majority of cases (Figure 5.4). On average of the entire benchmark set with the medium input size, LAANT has 21% of EDP gains on the 8-core system; 44% on the 24-core system; and 32% on the 32-core system.

While LAANT has EDP improvements for applications in which the parallel regions are executed many times, the OpenMP dynamic feature has its worst results, increasing the gap between both. The applications with this behavior are SP, Hotspot and Poisson, wherein each parallel region is executed more than a hundred thousand times (e.g., 156000 times in Poisson). For example, LAANT has EDP gains of up to 82% when compared to the $OMP\_Dynamic$ on the execution of Poisson on the 24-core system (Figure 5.4(b)). This happens because when the number of threads is changed by the $OMP\_Dynamic$, the workload must be redistributed to all threads again since it is performed as the parallel region executes. Therefore, the greater the number of parallel regions, the greater the overhead for redistributing the workload among the threads inside these regions. This will increase the difference in EDP between $OMP\_Dynamic$ and LAANT, since the latter does not suffer from this overhead. Also because of that, if one compares $OMP\_Dynamic$ with the **baseline** (OpenMP without dynamic adaptation), in some cases the average EDP increases.

A significant characteristic of the $OMP\_Dynamic$ was observed when running the LULESH application, in which it was unable to terminate the execution correctly. In such application, the variables inside the parallel region are allocated based on the number of threads that initialize those regions. Therefore, as the OpenMP dynamic feature changes the number of threads during the execution of the parallel region, it will allow a thread to access an address that was not allocated, causing an error in the execution. Besides correctly finishing the LULESH execution, LAANT reached huge EDP gains (up to 49%) over the baseline regardless the processor used.

Finally, the only cases when $OMP\_Dynamic$ had similar behavior as LAANT are with applications (FT, CG) where the ideal number of threads is near to the maximum. This is because the overhead of the $OMP\_Dynamic$ is reduced, since the behavior of the parallel regions of those applications have negligible changes during execution.

Table 5.3: Best number of threads (BNT) to execute each parallel region and LAANT overhead

Bench.	Input Set	8-Core System					24-Core System					32-Core System				
		BNT	***Time***	α-%	***Energy***	β-%	***BNT***	***Time***	α-%	***Energy***	β-%	***BNT***	***Time***	α-%	***Energy***	β-%
FFT	***S***	6,3,2	47.6s	0.9	1620J	1.5	20,12,6	39.9s	0.7	2743J	1.7	32,12,10	40.7s	0.4	3693J	0.7
	M	6,3,2	96.6s	0.7	3307J	1.6	20,12,6	72.6s	0.7	5423J	1.7	32,12,10	82,5s	0.5	7164J	0.8
BT	***S***	8,8, 2,2,8	44.4s	0.2	2504J	0.1	24,24, 4,4,24	28.5s	0.5	3424J	0.2	28,32, 16,8,23	21.8s	0.64	3544J	0.20
	M	8,8, 2,2,8	188.7s	0.1	10892J	0.1	24,4, 4,4,24	121.3s	0,5	14256J	0.2	32,32, 32,32,32	81.9s	2.1	14367J	0.2
LU	***S***	8,8	34.8s	0.5	2210J	0.4	12,24	20.8s	0.4	2614J	0.1	16,32	17.1s	3.6	2712J	0.1
	M	4,4	256.8s	1.3	12465J	1.2	12,24	83.8s	0.7	10797J	0.3	16,32	56.2s	0.8	9853J	0.3
LULESH	***S***	8,4,8	9.7s	0.3	635J	0.23	10,6,6	15.3s	0.2	1507J	0.2	14,10,16	13.7s	0.4	1687J	0.3
	M	4,4,8	77.3s	0,1	5327J	0.2	11,8,6	92.9s	0.3	9580J	0.2	15,12,16	77.9s	0.3	10143J	0.3
PO	***S***	4,4	19.0s	0.3	982J	0.1	12,24	13.0s	0.4	1676J	0.2	16,32	10.7s	0.4	1536J	0.3
	M	4,4	73.8s	0.3	4563J	0.2	12,24	31.4s	0.3	3825J	0.2	16,16	25.3s	0.4	3722J	0.3
SP	***S***	2	50.9s	0.1	2600J	0.2	10	34.3s	0.2	3787J	0.1	16	19.6s	0.2	3658J	0.1
	M	2	253.5s	0.1	11874J	0.2	6	194.1s	0.3	15614J	0.2	9	108.6s	0.4	16134J	0.1
HS	***S***	3	20.8s	0.10	1281J	0.04	12	17.3s	0.1	1938	0.1	16	13.4s	0.1	1804J	0.1
	M	3	51.3s	0.2	3196J	0.1	12	44.1s	0.1	4916J	0.1	16	36.1s	0.1	4867J	0.1
CG	***S***	8,3	15.9s	0.2	897J	0.1	24,23	11.9s	2.1	1445J	1.0	32,32	8.3s	1.9	1518J	0.9
	M	8,7	43.7s	0.4	2440J	0.2	24,23	30.8s	5.1	3791J	1.4	32,32	22.6s	6.0	4089J	3.0
FT	***S***	2,4,8	10.1s	5.2	559J	1.0	24,24,23	6.5s	5.5	845J	1.5	32,32, 32	5.6s	9.7	848J	3.4
	M	2,8,7	43.8s	0.6	2599J	0.1	24,24,24	29.1s	5.8	3862J	1.3	32,32,32	19.9s	8.1	3763J	1.6
Geometric Mean				0.3		0.2			0.5		0.3			0.6		0.3

Source: The Author

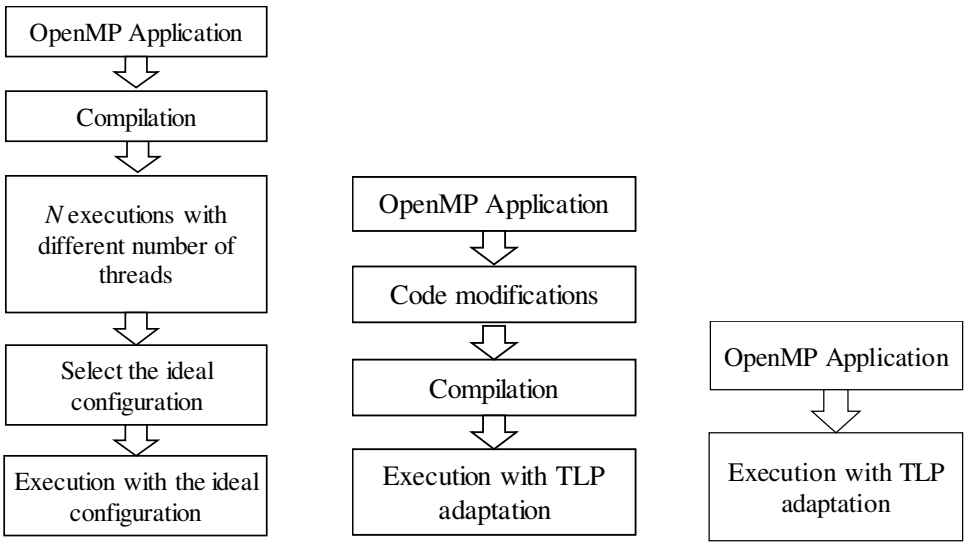
5.1.3 Discussion

This section presented LAANT, a library capable of automatically adjusting the number of threads to optimize the application for a given metric. Specifically, we defined LAANT to use EDP as optimization metric since it correlates energy and performance in a unique value. LAANT applies a hill climbing algorithm for training while the application is running, which results in an almost negligible overhead for most of the applications. We show that LAANT can reduce the EDP in 29%, on average, when compared to the standard way that OpenMP applications are executed; and reduces the EDP in up to 82% when compared to the dynamic feature of OpenMP is active.

5.2 Aurora: Seamless Optimization of OpenMP Applications

In the previous Section, a library to automatically adjust the number of threads for OpenMP applications was presented. Although LAANT provides positive results, it has some limitations, such as the need of code annotation and recompilation (Figure 5.5). In this way, LAANT was extended to Aurora: given an OpenMP application already compiled, Aurora is capable of adapt the number of threads without the need for any modifications in the source code nor code recompilation, as shown in Figure 5.5(c).

Figure 5.5: Adaptation of OpenMP Applications



(a) Brute force (b) LAANT (c) Aurora

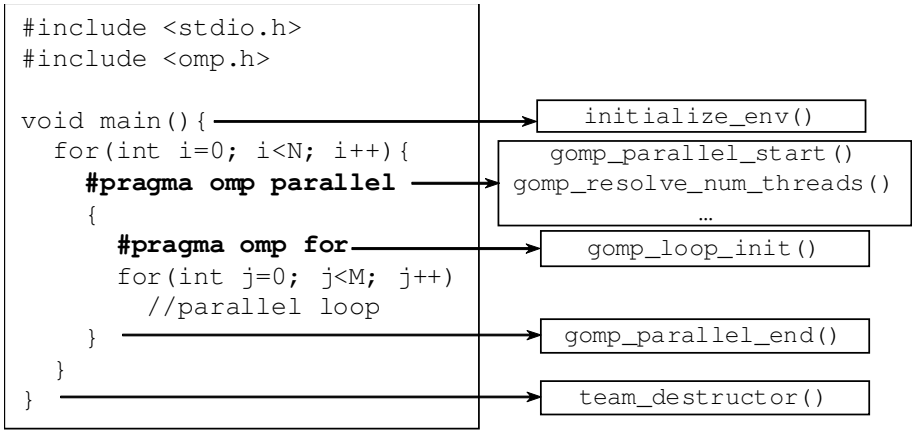
Source: The Author

5.2.1 Integration to OpenMP

As already described in Chapter 2, parallelism in OpenMP is exploited through the insertion of directives in the sequential code that inform the compiler how and which parts of the application should be executed in parallel (CHAPMAN; JOST; PAS, 2007). OpenMP provides three ways for exploiting parallelism: parallel loops, sections, and tasks. Parallel sections and tasks are only used in very particular cases: when the programmer must distribute the workload over the threads in a similar way as PThreads, and when the application uses recursion (i.e., sort algorithms), respectively. On the other hand, parallel loops are used to parallelize applications that work on multidimensional data structures (i.e., array, grid, etc.), so the loop iterations (*for*) can be split into multithread executions. Therefore, parallel loops are by far the most used approach (all the aforementioned benchmark sets are implemented in this way). For now, Aurora works to optimize parallel loops and does not influence in any way other OpenMP applications that are parallelized using sections or tasks.

All functionalities provided by OpenMP are implemented into the *libgomp*, a GNU Offloading and Multi-Processing Run-time Library. This library is dynamically linked to applications that use OpenMP, so any modifications in its code are completely transparent to user applications. Aurora was incorporated into this library. In order to better understand how Aurora works, let us first consider Figure 5.6, which illustrates the regular way for parallelizing an iterative application with parallel loops (CHAPMAN; JOST; PAS,

Figure 5.6: OpenMP execution environment with the respective *libgomp* functions



Source: The Author

2007) and the respective main functions implemented by *libgomp*. When the program starts executing, the *initialize_env*() function is called, which is responsible for initializing all the environment variables used by OpenMP during the application execution. When the program reaches the directive *#pragma omp parallel* (used to indicate a parallel region), functions to create and define the number of threads (*gomp_resolve_num_threads*()) are called. Within the parallel region, the directive *#pragma omp for* indicates the loop that must be parallelized. At the end of the parallel region, the function *gomp_parallel_end*() joins the threads and finalizes the parallel region environment. Finally, when the application ends, *team_destructor()* concludes the entire OpenMP environment.

Aurora functionalities were split into four functions (discussed in details next). They were incorporated into the *libgomp* functions previously mentioned. Algorithm 1 depicts the modifications done in the source code of each function in order to support Aurora functions. *libgomp* also has another function called *gomp_loop_init*(), which was not modified as its job is to distribute the workload between the already defined threads.

auroraInitEnv*()** is responsible for recognizing the Aurora optimization target defined by the environment variable (OMP_AURORA) and for initializing the necessary data structures, libraries, and variables used to control the search algorithm (described in Section 5.2.2). The pseudocode of this function can be seen in Algorithm 2. ***auroraInitEnv is called from the original *initialize_env*() only if Aurora optimization is enabled, as presented in Lines 3-7 in Algorithm 1. If OMP_AURORA is not defined, the OpenMP execution follows its standard behavior.

***auroraResolveNumThreads*()** sets the number of threads that execute each parallel region based on the current state of the search algorithm. Also, it initializes the

counters for collecting data from the execution environment of the current parallel region. Algorithm 3 depicts the pseudocode of this function: if the parallel region is a new region, the search algorithm will start the search from the initial state (*S0*) and with the number of threads defined either by the environment variable AURORA_START or by 2, that is the standard value used by Aurora. Also, if the search algorithm is in the *END* state, the best number of threads (*bnt*) found to execute a parallel region is returned. Otherwise, the actual number of threads (*ant*) is returned. ***auroraResolveNumThreads*** is called by the *gomp_parallel_start*()[1] when Aurora is active, replacing the original *gomp_resolve_-num_threads*() function, as depicted in Algorithm 1.

***auroraEndParallelRegion*()** is executed after the parallel region to get its execution time, energy, or EDP, depending on the optimization metric defined by the user. Ex-

[1] *GOMP_parallel_start* is also namely as *GOMP_parallel*

Algorithm 1 OpenMP functions that were modified to integrate Aurora optimization

```
1: function INITIALIZE_ENV(void)
2:     Initialization of OpenMP environment (variables, CPU affinity, wait policy, etc.)
3:     if OMP_AURORA is defined then
4:         aurora_metric ← get the value defined by the user in OMP_AURORA
5:         aurora_start_search ← get the value defined by the user in AURORA_START
6:         auroraInitEnv(aurora_metric, aurora_start_search)
7:     end if
8: end function

9: function GOMP_PARALLEL_START(*fn, *data, num_threads)
10:    ptrToRegion ← gets pointer to fn address region
11:    if Aurora is Enabled then
12:        num_threads ← auroraResolveNumThreads(ptrToRegion)
13:    else
14:        num_threads ← gomp_resolve_num_threads(num_threads, 0)
15:    end if
16:    gomp_team_start(fn, data, num_threads, 0, gomp_new_team(num_threads))
17: end function

18: function GOMP_PARALLEL_END(void)
19:    if OMP_AURORA is defined then
20:        auroraEndParallelRegion();
21:    end if
22:    finalize parallel region environment
23:    gomp_team_end()
24: end function

25: function TEAM_DESTRUCTOR(void)
26:    if OMP_AURORA is defined then
27:        auroradestructEnv();
28:    end if
29:    pthread_key_delete(gomp_thread_destructor)
30: end function
```

Source: The Author

Algorithm 2 Initialization of Aurora Environment

```
1: function AURORAINITENV(metric, startSearch)
2:     numCores ← get the total number of cores through sysconf
3:     threadStartSearch ← get the number of threads defined to start the search
4:     Initialize hardware counters to get the parallel region behavior
5:     for i in maxNumberOfParallelRegions do
6:         Initialize the variables used to monitor/control the parallel region i
7:         i.e., startSearch, metric, actualstate
8:     end for
9: end function
```

Source: The Author

ecution time is extracted by the *omp_get_wtime()* function, provided by OpenMP, while energy is obtained directly from the hardware counters present in modern processors. In the case of Intel processors, the RAPL library is used to get energy and power consumption of CPU-level components (HäHNEL et al., 2012), while the APM library is used for AMD processors (HACKENBERG et al., 2013a). Such functions and libraries were incorporated to Aurora, being totally transparent to the user. That is, there is no need to make any modifications in the Operating System (package installation, kernel recompilation, etc.) to use them.

Using either one of the objectives of execution time, energy, or EDP, *auroraEndParallelRegion*() performs one step of the search algorithm (which is explained in the next sub-section) and, according to this algorithm, it defines the number of threads that will be used for the execution of this parallel region in the next iteration. *auroraEndParallelRegion*() is implemented inside *gomp_parallel_end*() function, and it is called when Aurora is active, as depicted in Algorithm 1.

Algorithm 3 Setting up the number of threads

```
1: function AURORARESOLVENUMTHREADS(ptrToRegion)
2:     idR ← get the id of the parallel region from ptrToRegion
3:     if idR is a newRegion then
4:         auroraKernel[idR].state ← S0
5:     end if
6:     switch auroraKernel[idR].state do
7:         start monitoring the parallel region behavior
8:         case END
9:             return auroraKernel[idR].bnt
10:        case Default
11:            return auroraKernel[idR].ant
12: end function
```

Source: The Author

***auroraDestructEnv*()** concludes and destroys Aurora environment at the end of application execution, when Aurora is active (Algorithm 1. It was implemented inside *team_destructor*() OpenMP function.

To use Aurora, the user simply has to replace the original OpenMP *libgomp* with Aurora's *libgomp*. This new library includes all original OpenMP functionalities plus the new functions of Aurora. When the environment variable *OMP_AURORA* is set in the Linux Operating System, the thread management system of Aurora is used instead of the original OpenMP functions. This environment variable can be configured to the following values (and, therefore, optimization metrics): performance, energy, or EDP. If the variable is not set, Aurora will not influence the execution of that OpenMP application (i.e., the application executes with the original OpenMP functions). In this way, any existing binary code can benefit from Aurora without any modifications or need for recompilation.

5.2.2 Search Algorithm

The heuristic used by Aurora improves the one used by LAANT, and it is divided into two phases. The first one investigates the scalability of the parallel region and reduces the size of the space exploration, exponentially increasing the number of threads (i.e., 2, 4, 8, 16, ...) while there are potential improvements (cases S0, S1, and S2 in Algorithm 4, Figure 5.7, and Table 5.4). The second phase performs a based hill-climbing algorithm in the interval of threads defined in the first phase (cases S2, S3, and S4). As the search algorithm implemented by Aurora learns towards the best number of threads as the application executes, all the computation done during this step is not wasted (i.e., it is used for

Figure 5.7: States and transitions of the search algorithm

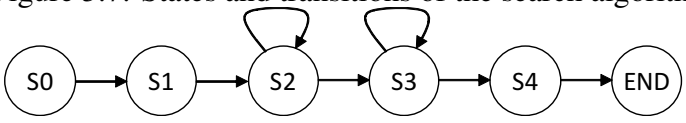


Table 5.4: States of the search algorithm

State	Operation
S0	Execution with the initial number of threads.
S1	Double the number of threads.
S2	Compare the results achieved in S0 and S1, and exponentially increases the number of threads while either there are improvements or when the max number of hardware threads is met. Then, state changes to S3.
S3	Search the ideal number of threads in the interval of candidates defined in S2. When there are only two candidates, state changes to S4.
S4	Define the best number of threads and performs lateral movement.
END	Aurora begins to monitor the behavior of the parallel region.

Algorithm 4 Search algorithm implemented by Aurora

```
1: function SEARCHALGORITHM()
2:     if state != END then
3:         result ← get time, energy, or EDP according to the target metric
4:         switch state do
5:             case S0:
6:                 lastNT ← actualNT ← threadStartSearch;
7:                 state ← S1;
8:             case S1:
9:                 bestResult ← result;
10:                bestNT ← actualNT;
11:                actualNT ← actualNT × 2;
12:                state ← S2;
13:            case S2:
14:                step ← lastNT/2;
15:                if result ≤ bestResult then
16:                    bestResult ← result;
17:                    bestNT ← actualNT;
18:                    if actualNT × 2 ≤ numCores then
19:                        lt ← actualNT;
20:                        actualNT ← bestNT × 2;
21:                    else
22:                        actualNT -= step;
23:                        state ← S3;
24:                    end if
25:                else
26:                    if bestNT == numCores/2 then
27:                        actualNT -= step;
28:                    else
29:                        actualNT += step;
30:                    end if
31:                    state ← S3;
32:                end if
33:            case S3:
34:                if result ≤ bestResult then
35:                    bestNT ← actualNT;
36:                    bestResult ← result;
37:                end if
38:                step ← step/2;
39:                actualNT += step;
40:                if step == 1 then
41:                    state ← S4;
42:                end if
43:            case S4:
44:                if result ≤ bestResult then
45:                    bestNT ← actualNT;
46:                end if
47:                Performs lateral movement to avoid minimum locals
48:                state ← END;
49:    else
50:        if workloadVariation == true then
51:            run Aurora search algorithm again
52:        end if
53:    end if
54: end function
```

the application), reducing the overhead of Aurora. Basically, the search algorithm works as follows (Algorithm 4):

The search starts by the *S0* state (*line 5*), where the initial number of threads and the actual number of threads is defined. Then, the parallel region is executed with the initial number of threads (e.g., 2 threads) and the state changes to *S1*. In *S1* (*line 8*), the best result so far is updated with the result obtained by the execution with the number of threads defined in *S0*, the number of threads is doubled, and state changes to *S2*. In

S2 (*line 13*), the measured metric (time, energy, or EDP) is evaluated, and the number of threads will continue to double either while the measured result keeps improving or until the execution reaches the maximum number of hardware threads available (*lines 15-25*). When the condition to stop increasing threads is met, the state changes to *S3*. Once in *S3* (*line 33*), Aurora knows the interval of potential candidates for the ideal number of threads, which is in the range between the last number of threads executed and the best number of threads found so far. Then, the algorithm starts the second phase.

To better understand the second phase, let us consider that the interval of potential candidates lies in the range of 8 and 16 threads. Then, the algorithm searches for the best number of threads in this range. It will start executing with 12 threads (the average amount between 8 and 16) and then compares to the best result so far to decide the next range (which will be between 8-12 or 12-16). This process is repeated until the best number of threads is found (*state S4*). At this point, to avoid minimal locals and plateaus during the search, and so wrongly converging to an incorrect number of threads, Aurora uses lateral movement (*line 47*). When Aurora converges to the best number of threads for a particular parallel region, it begins to monitor the behavior of such region. If there is any change in the workload, which in this work we consider a variation of 30%, the search algorithm starts its execution again.

5.2.3 Methodology

5.2.3.1 Benchmarks

In order to evaluate the Aurora behavior, some applications were added to the benchmark set used to evaluate LAANT (Section 5.1). In this way, Fifteen applications written in C/C++ and already parallelized with OpenMP from assorted benchmarks suites were classified according to the scalability issues discussed in Section 2.2. The benchmarks used were:

- Seven kernels from the NAS Parallel Benchmark (BAILEY et al., 1991). NAS is a suite originally developed by NASA that comprises applications derived from computational fluid dynamic: BT, CG, FT, LU, Multi-Grid on a sequence of meshes (MG), SP, and Unstructured Adaptive mesh (UA). As the original version of NAS is written in FORTRAN, we consider the OpenMP-C version developed by the authors in (SEO; JO; LEE, 2011).

Table 5.5: Pearson correlation between the scalability issues and each benchmark

		NB	FFT	ST	UA	JA	SP	HS	SC	PO	HPCG	MG	BT	LU	CG	FT
Small Input	***Issue-width sat.***	-0.82	-0.71	-0.56	**-0.92**	-0.80	-0.80	**-0.91**	-0.80	-0.84	-0.65	-0.81	-0.75	-0.87	**-0.91**	**-0.90**
	Off-chip bus sat.	0.46	**-0.98**	**-0.90**	-0.84	-0.57	-0.71	-0.51	-0.82	-0.56	**-0.94**	-0.76	-0.79	-0.80	-0.82	-0.68
	Shared mem. acc.	0.80	-0.43	-0.71	-0.78	0.52	-0.83	-0.52	**-0.91**	0.71	-0.86	**-0.90**	**-0.91**	**-0.96**	-0.85	-0.78
	Data-synchr.	**0.97**	-0.50	-0.61	-0.49	**0.92**	**0.95**	-0.54	-0.54	**0.94**	-0.24	-0.59	-0.64	-0.61	-0.61	-0.82
Medium Input	***Issue-width sat.***	-0.78	-0.71	-0.63	-0.73	-0.69	-0.82	**-0.92**	-0.76	-0.83	-0.74	-0.79	-0.73	**-0.90**	**-0.94**	**-0.91**
	Off-chip bus sat.	0.39	**-0.97**	**-0.95**	-0.85	**-0.90**	-0.62	-0.52	-0.86	-0.46	**-0.94**	-0.88	-0.79	-0.65	-0.82	-0.76
	Shared mem. acc.	0.81	-0.75	-0.73	**-0.94**	0.82	**-0.90**	-0.54	**-0.96**	**-0.94**	-0.86	**-0.96**	**-0.92**	0.09	0.70	-0.86
	Data-synchr.	**0.96**	-0.53	-0.38	-0.74	-0.48	-0.11	-0.64	-0.68	-0.67	-0.70	-0.78	-0.61	-0.18	-0.64	-0.77

Source: The Author

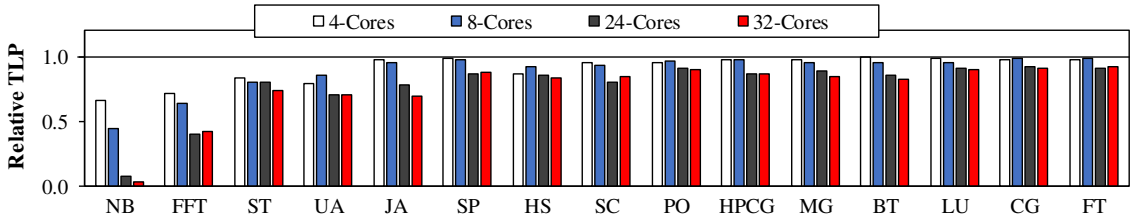
- Two applications from the Rodinia Benchmark Suite (CHE et al., 2009): Hotspot (HS), which iteratively solves a series of differential equations; and Streamcluster (SC), a dense linear algebra algorithm for data mining.

- Six applications from different domains: N-body (NB) - computes a simulation of a dynamical system of particles (BHATT et al., 1992); FFT - calculates the discrete Fourier transform of a given sequence (PETERSEN; ARBENZ, 2004); STREAM (ST) - measures sustainable memory bandwidth (MCCALPIN, 1995); Jacobi (JA) method iteration - computes the solutions of a diagonally dominant system of linear equations (QUINN, 2004). Poisson (PO) - computes an approximate solution to the Poisson equation in a rectangular region (QUINN, 2004); and the High Performance Conjugate Gradient (HPCG) benchmark, a stand-alone code that measures the performance of basic operations (i.e., sparse matrix multiplication, vector updates, etc.) (DONGARRA; HEROUX; LUSZCZEK, 2015).

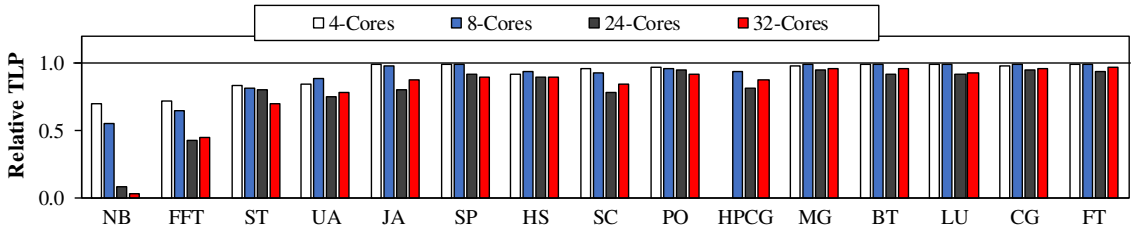
Two different input sets for each benchmark were considered: small and medium. Table 5.5 depicts the Pearson correlation between each scalability issue (discussed in Section 2.2) and the application behavior. As can be observed, the chosen applications do not scale for different reasons, according to Section 2.2. All the data used for the scalability analysis was obtained directly from hardware, as follows: the off-chip bus utilization was accessed using Intel Performance Counter Monitor (PCM) (WILLHALM; DEMENTIEV; FAY, 2017); the time that threads spend synchronizing was obtained from the Intel Parallel Studio; and the number of cycles with no instruction issue and the shared memory accesses were collected with PAPI (BROWNE et al., 2000);

As one can note in Figure 5.8, the chosen benchmarks also cover a wide range of different TLP behaviors (normalized with respect to the maximum number of threads in each processor), which varies from the NB (lowest TLP available, where only 10% of the execution is performed in parallel when the 32-core system is considered) to the FT

Figure 5.8: TLP Available for each benchmark - normalized wrt the maximum number of threads in each processor



(a) Small input set



(b) Medium input set

Source: The Author

benchmark (highest TLP available, in which more than 95% of the application is executed in parallel). We measured TLP as defined by the authors in (BLAKE et al., 2010): the average amount of concurrency exhibited by the program during its execution when at least one core is active, and it is expressed in Equation 5.1. c_i is the fraction of time that i cores are concurrently running different threads, n is the number of cores, and $1 - c_0$ is the non-idle time fraction. The closer this value is to 1.0 (normalized to the total number of cores available), the more TLP is available (BLAKE et al., 2010).

$$TLP = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0} \tag{5.1}$$

5.2.3.2 Execution Environment

The experiments were performed on four different multicore processors (Table 5.6). We used the Ubuntu Operating System with Kernel v. 4.4.0 in all the machines. The CPU frequency was configured to adjust according to the workload application, using ondemand as Dynamic Voltage and Frequency Scaling (DVFS) governor, which is the standard governor used in most Linux versions. We compiled the applications with gcc/g++ 6.3, using the optimization flag -O3, and the OpenMP distribution version 4.0. The results presented in the next session are the average of ten executions with a standard deviation lower than 0.5%.

Table 5.6: Main characteristics of each processor

	Intel Core		**Intel Xeon**	
	i5-4460	***i7-6700***	***E5-2630***	***E5-2640***
Microarch.	Haswell	Skylake	Sandy Bridge	Ivy Bridge
# Cores	4	4	2x6	2x8
# Threads	4	8	24	32
CPU Freq	3.2 GHz	3.4 GHz	2.3 GHz	2.0 GHz
L1 Cache	4x32 KB	4x32 KB	12x32 KB	16x32 KB
L2 Cache	4x256 KB	4x256 KB	12x256 KB	16x256 KB
L3 Cache	6 MB	8 MB	30 MB	40 MB
RAM	16 GB	32 GB	32 GB	64 GB

Source: The Author

We evaluated Aurora in four different scenarios:

- **Baseline**: the application executes with the maximum number of threads available in the system;

- **OMP_Dynamic**: a built-in feature of OpenMP that dynamically adjusts the number of threads of each parallel region, aiming to make the best use of system resources, such as memory and processor. This feature is enabled by using the environment variable OMP_DYNAMIC or through the insertion of the *omp_set_dynamic*() in the source code (CHAPMAN; JOST; PAS, 2007);

- **Feedback-driven threading (FDT)**: is the most popular framework and therefore is widely used for comparisons with new approaches. In it, the number of threads is defined based on the contention for locks and memory bandwidth (as discussed in Section 3.2). We have faithfully implemented FDT mechanism in C language and inserted their functions into the OpenMP codes, as defined by (SULEMAN; QURESHI; PATT, 2008).

- **Oracle solution**: the execution of each parallel region with the optimal number of threads for each metric, without the cost of the learning curve. The optimal number of threads was obtained through an exhaustive execution of each parallel region of each application with 1 to n threads, where n is the maximum number supported by hardware.

5.2.4 Results

5.2.4.1 Performance, Energy, and EDP

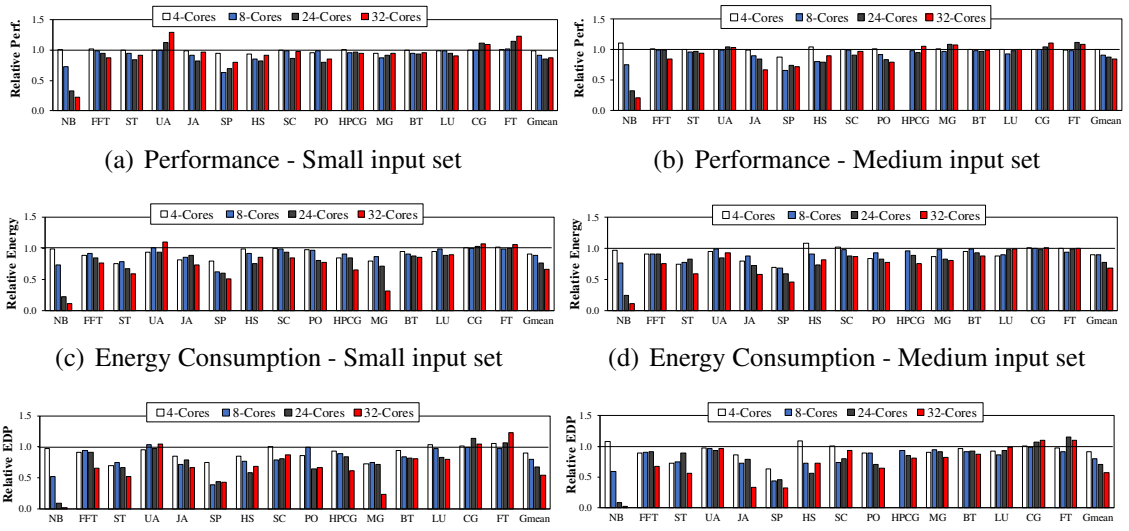
Figures 5.9, 5.14, and 5.15 present the results for the entire benchmark set, along with their geometric mean (Gmean) considering the four multicore systems. Figure 5.9 compares Aurora to the baseline (represented by the black line), while Figures 5.14 and 5.15 compare Aurora to OMP_Dynamic and FDT framework, respectively (also represented by a black line). Results are normalized according to the baseline, OMP_Dynamic, or FDT, and presented regarding performance, energy consumption, and EDP, depending on the optimization metric used by Aurora. As an example, Figure 5.9(c) shows the energy savings achieved by Aurora over the baseline when set to reduce the energy consumption. Moreover, Appendix A depicts the results for all the experiments discussed in this section.

Aurora versus Baseline: as observed in Figure 5.9, in most cases Aurora shows improvements regarding any metric. On the other hand, if one considers the geometric mean (Gmean bars in each figure) in any scenario, Aurora is always better (in very specific scenarios where the design space exploration is limited, it presents similar results as the baseline). Considering its best case compared to the baseline, the execution time was reduced by 16% with the medium input set executing on the 32-core system. The best scenario for energy consumption and EDP is the small input set on the 32-core system: energy is reduced by 34%, and EDP is improved by 47%. When considering the overall geometric mean (entire benchmark set and all processors), Aurora provided 10% of performance improvements, 20% of energy reductions, and 28% of EDP improvements.

Aurora versus OMP_Dynamic: considering the best case for each metric regarding the geometric mean (Gmean) in Figure 5.14, Aurora reduced the execution time by 26% (medium input on the 4-core machine), energy consumption by 24% (medium input on the 32-core system) and EDP by 38% (small input on the 4-core system). In the overall geometric mean, Aurora was 11% faster, saved 17% of energy, and improved EDP by 32%.

Aurora versus FDT: as observed in Figure 5.15, Aurora outperforms FDT in all cases regarding any metric and processor. In the best case of Aurora for each metric regarding the geometric mean (Gmean), Aurora reduced the execution time and energy consumption by 34%, and EDP by 56% (small input set on the 24-core machine). When

Figure 5.9: Aurora vs Baseline: lower than 1.0 means that Aurora is better than the baseline



(a) Performance - Small input set

(b) Performance - Medium input set

(c) Energy Consumption - Small input set

(d) Energy Consumption - Medium input set

(e) EDP - Small input set

(f) EDP - Medium input set

Source: The Author

considering the overall geometric mean, Aurora provided 26% of performance improvements, 25% of energy reductions, and 45% of EDP improvements.

Tables 5.7 depicts the number of threads that offers the best result to execute all parallel regions of each application for each optimization metric (performance, energy, and EDP), which we call this scenario as **Oracle**. As an example, let us consider the FFT application that has two parallel regions, in which when optimized for performance - for the medium input set (M) on the 8-core system – the optimal number of threads for the first region is eight and for the second region is two. One can note that depending on the input set, the optimal number of threads for each parallel region may vary, which is the case of the JA application when optimized for EDP running on the 32-core system. When changing the input set from small to medium, the workload of the parallel region changes, increasing its TLP. In this case, the number of threads that offer the best EDP for this region is now 6 instead of 15. In all cases, Aurora found the best number of threads.

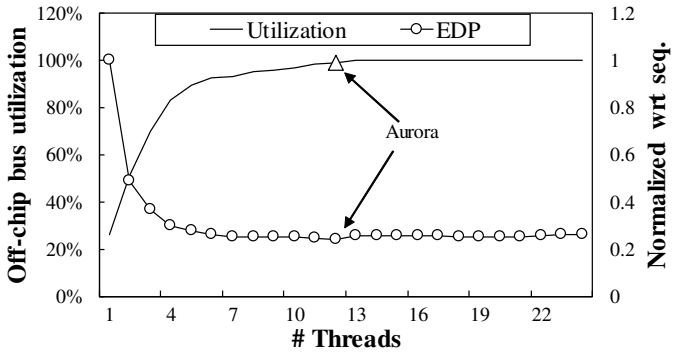
The same behavior is observed when the optimization metric changes, which is the case of the ST benchmark running on the 32-cores system. When Aurora is set to optimize performance, 12 threads offer the best result. On the other hand, the lowest energy consumption is reached with only four threads and the best EDP with six threads.

5.2.4.2 Handling Scalability

As a result of its run-time analysis, the search algorithm used by Aurora can detect the point in which the number of threads saturates any metric. As the first example, the off-chip bus saturation is considered (as discussed in Section 2.2). For instance, consider the execution of HPCG benchmark. This benchmark has four parallel regions that are mostly better executed with a different number of threads (Table 5.7). Let us discuss the behavior of the second parallel region considering the EDP, in which the optimal execution is with 12 threads, when executing with medium input on the 24-core system. Figure 5.10 shows that when this region is executed with more than 12 threads, the off-chip bus saturates (100% of utilization), and no further EDP improvements can be achieved by increasing the number of threads. By using its continuous monitoring and avoiding this saturation, Aurora was able to reduce the EDP of the whole application by 15% (Figure 5.9(f)). The very same behavior can be observed in FFT and ST (regardless of the input set) and JA for the medium input set (Table 5.5) but at different improvement ratios.

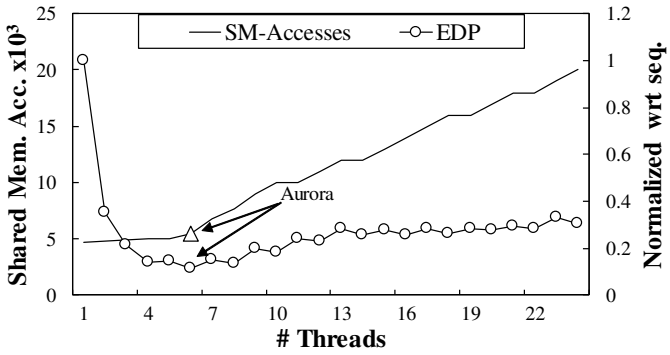
In applications with high communication demands among the threads, there is an optimal point in which the overhead imposed by the shared memory accesses does not overcome the gains achieved by the parallelism exploitation, as discussed in Section 2.2. Aurora detected this point for all benchmarks in this class: SC, MG, and BT; LU (with small input); PO, UA, and SP (with medium input) (Table 5.5). As an example, let us consider the SP benchmark running on the 24-core system. This application has thirteen parallel regions, in which each one is better executed with a different number of threads. Figure 5.11 shows that when the number of shared memory accesses performed by all threads in the first parallel region starts to increase (after six threads - primary y-axis), no further improvements in the EDP of the application can be achieved (secondary y-axis). As shown in the same Figure and in the Table 5.7, Aurora found the best number

Figure 5.10: Behavior of the 2^{nd} parallel region of HPCG



Source: The Author

Figure 5.11: Behavior of the 1^{st} parallel region of SP



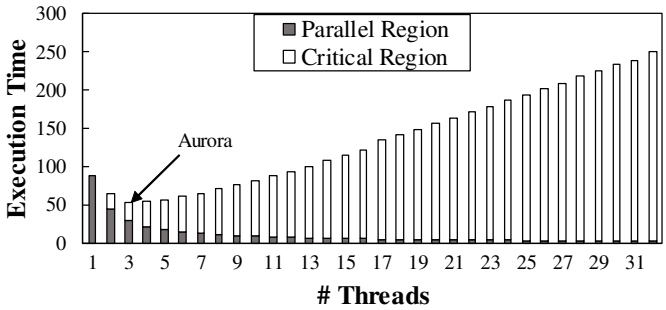
Source: The Author

of threads to execute this parallel region, providing EDP gains of 58% when set to this target (Figure 5.9(f)).

Aurora will similarly detect the point that the synchronization time overlaps the gains provided by TLP exploitation. This behavior can be observed in some benchmarks, such as n-body (NB) with small or medium input, Jacobi(JA) and SP with small input set (Table 5.5). In these benchmarks, the higher the number of threads, the greater the time spent synchronizing, which can worsen the application result, as already discussed in Section 2.2. As a specific example of this behavior, we consider the n-body benchmark with the medium input set executing on the 32-core system (Figure 5.12). When increasing the number of threads from 1 to 3, the performance is improved. However, from this point on, the time that the threads spend synchronizing overcomes the gains achieved by the parallelism exploitation (Figure 5.12), increasing the energy consumption and EDP of the whole application. As demonstrated in Table 5.7, by selecting the best number of threads and avoiding the increase in the number of threads, Aurora reduced the execution time by 79%, energy by 89%, and EDP by 98%.

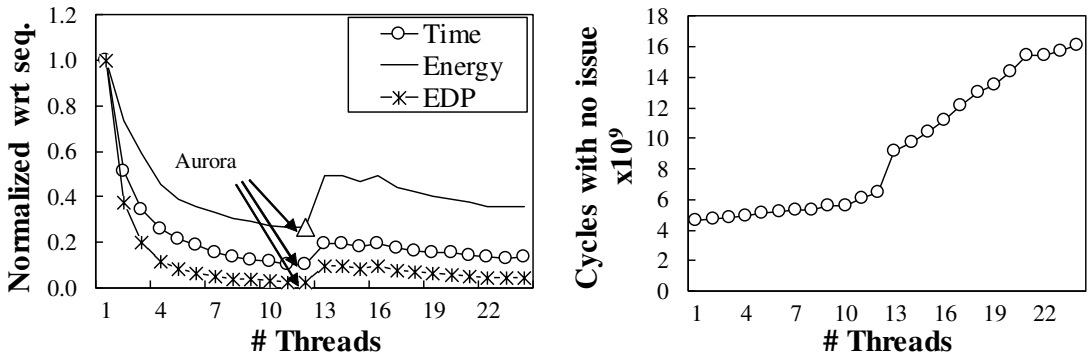
Aurora also converges to the best number of threads for applications that are negatively influenced by the issue-width saturation, such as Hotspot (HS), FT, and CG with

Figure 5.12: N-body execution on the 32-core system



Source: The Author

Figure 5.13: Hotspot behavior on the 24-core system



(a) Hotspot Behavior (b) Cycles with no issue

Source: The Author

any input set; and UA and PO with the small input (Table 5.5). To discuss this behavior, let us consider the Hotspot benchmark with the medium input set executing on the 24-core system, in which the number of threads that provide the best EDP is 12 (see Table 5.7). As Figure 5.13 shows, when increasing the number of threads from 12 to 13, the number of cycles that the threads spend without issuing any instruction abruptly increases (Figure 5.13(b)), decreasing performance and increasing energy consumption (Figure 5.13(a)). Once more, by avoiding the excessive increment in the number of threads, Aurora increased performance by 21% and reduced EDP and energy by 44% and 25%, respectively (Figure 5.9), when set to these targets.

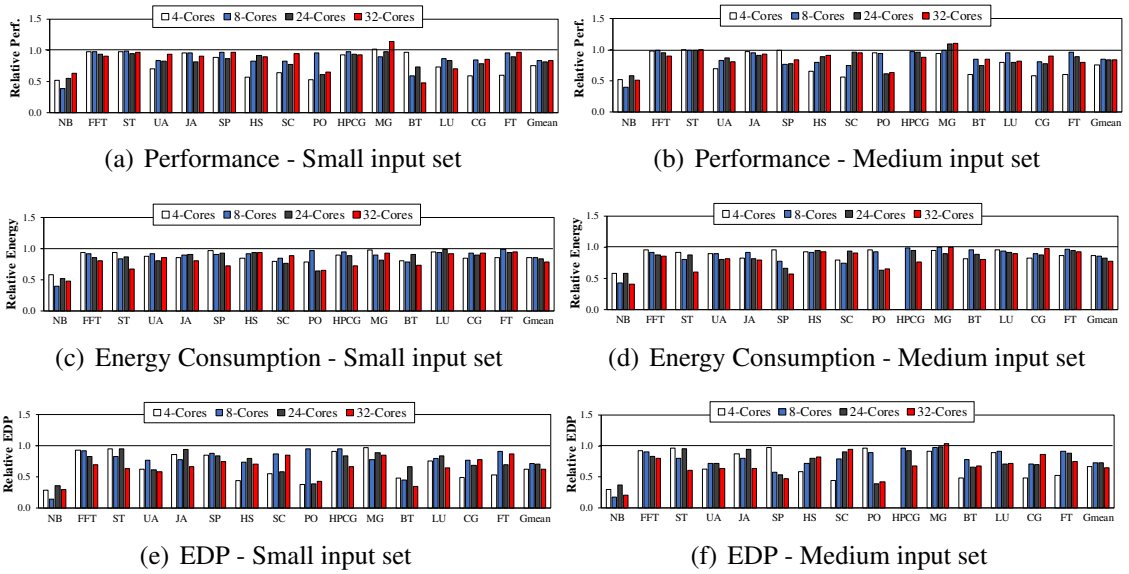
Finally, as discussed in Section 2.2, it is important to note that there are cases in which the characteristic that influences the thread scalability changes according to the input set (Table 5.5). As a specific example, let us consider JA application. When it is executed with the small input, the time that the threads spend synchronizing limits the application scalability, while in the medium input the operations in a larger amount of data saturate the off-chip bus.

5.2.4.3 Costs of the Learning Curve

Table 5.8 depicts (in percentage) how the results obtained by Aurora differs from the **Oracle** solution for each application in all scenarios (benchmarks, processors, and metrics). It also shows how the results achieved by OMP_Dynamic and FDT approaches differ from the **Oracle** regarding the geometric mean (gmean) of the entire benchmark set for each processor and metric.

Using this optimal solution, we can measure the cost of the learning curve, that is, the overhead of our technique, which is originated from two different situations: the

Figure 5.14: Aurora vs OMP_Dynamic: lower than 1.0 means that Aurora is better than OMP_Dynamic



(a) Performance - Small input set

(b) Performance - Medium input set

(c) Energy Consumption - Small input set

(d) Energy Consumption - Medium input set

(e) EDP - Small input set

(f) EDP - Medium input set

Source: The Author

execution of the search algorithm itself; and the execution of a given parallel region with a number of threads that is not the ideal, while the search algorithm is trying different possibilities to converge to the ideal number. Aurora showed high overheads in the following situations (Table 5.8):

- The best result is achieved with either the maximum number of threads or a number close to it, which is the case of the FT and CG benchmarks in the execution on the 24 and 32-core systems.

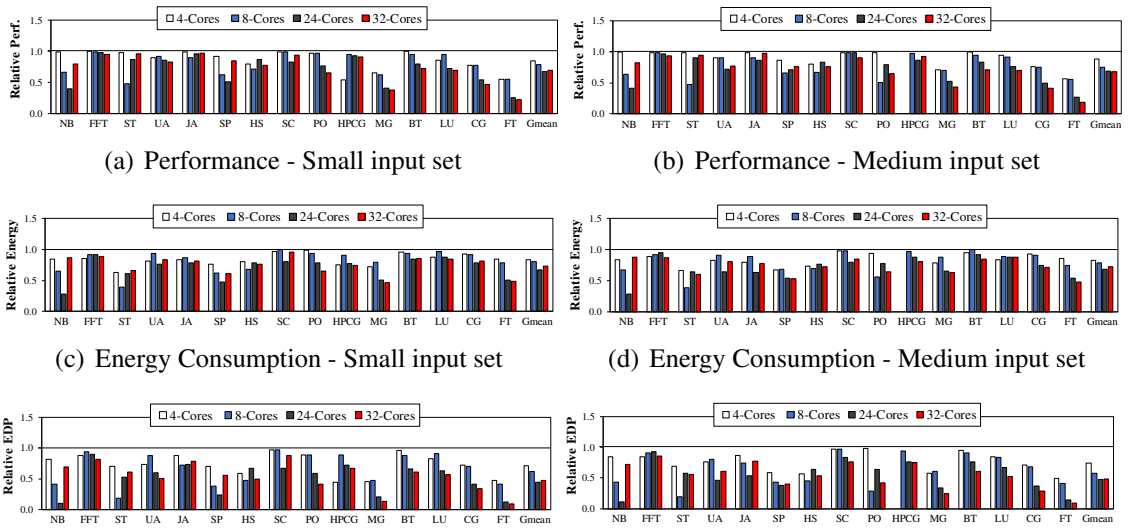
- The parallel region has a relatively small number of interactions but executes for a significant time, such as HPCG;

- Applications that have short execution time (i.e., less than 10 seconds), such as MG, in which the execution with the small input set by the Oracle solution takes only 1.45 seconds in the 32-core system.

- Applications with many parallel regions, in which most of them have a low workload, as in the UA benchmark. UA has 54 parallel regions, in which 44 of them take less than 0.5 seconds to execute regardless of the four target processors.

Besides, the higher the number of hardware threads available in the system, the greater the space exploration that the search algorithm must cover. However, even though

Figure 5.15: Aurora vs FDT: lower than 1.0 means that Aurora is better than FDT



(a) Performance - Small input set

(b) Performance - Medium input set

(c) Energy Consumption - Small input set

(d) Energy Consumption - Medium input set

(e) EDP - Small input set

(f) EDP - Medium input set

Source: The Author

the overhead increases, it does so in small rates, as can be observed when one compares the averages of the 24- and 32-core systems to the 4- and 8-core systems.

While Aurora defines the number of threads based on the parallel region behavior, OMP_Dynamic considers the last 15 minutes of execution to define the number of threads (CHAPMAN; JOST; PAS, 2007). It does not use any search algorithm nor considers each parallel region in particular. For this reason, OMP_Dynamic presented poor results, and, in many cases, it is worse than the baseline (average of the entire benchmark for performance and EDP). The advantage of not being called often (and therefore potentially decreasing overhead) does not overcome the fact that it is not able to get near to the optimal number of threads. Considering the geometric mean of the entire benchmark set, Aurora gets much closer to the results achieved by the Oracle solution than the OMP_Dynamic, as shown in Table 5.8.

As observed in Figure 5.15, only in particular cases, FDT can achieve similar results than Aurora. These applications are part of the scalability issues that the FDT proposes to solve, such as FFT (off-chip bus saturation) and JA (synchronization) when performance is considered. On the other hand, FDT does not deal with all scalability issues, and therefore, converges to the incorrect number of threads in many applications. Moreover, as observed in Table 5.8, FDT presents higher overhead for learning than Aurora (geometric mean of the entire benchmark set). This behavior arises due to the following reasons:

- As the training phase of FDT considers the execution of each parallel region in a single threaded mode until the standard deviation of the observed metrics (memory bandwidth usage and synchronization time) is stable, it leads to a higher overhead in applications that present medium and high degree of TLP. Because of this, in many cases FDT achieves worse results than the baseline and OMP_Dynamic.

- FDT considers that all threads are homogeneous and ignores fundamental hardware characteristics that are highly correlated to the parallel application behavior: FDT assumes that bandwidth requirement increases linearly with the number of threads, ignoring cache contention and data-sharing between the threads. Moreover, FDT does not consider the effects of the SMT feature (discussed in Section 2.2), by assuming that only one thread executes per core.

- FDT does not consider that each metric may have a different optimal number of threads. Instead, it assumes the same number of threads for all the evaluated metrics, which shows to be incorrect, for reasons discussed in the previous Sections.

5.2.5 Discussion

In this subsection, we have presented Aurora, a mechanism capable of automatically finding, at run-time, the optimal number of threads for each parallel region. Aurora is completely transparent to both designer and end user: given an OpenMP application binary, Aurora optimizes it without any code changes, transformation or recompilation, by simply setting an environment variable in Linux OS.

We have shown that Aurora can optimize distinct OpenMP applications regarding different metrics with an almost negligible overhead: in the most significant case, Aurora improves performance, energy, and EDP by up to 79%, 89%, and 98%, respectively, over the standard OpenMP execution; by up to 62%, 61%, and 86%, when compared to the built-in feature of OpenMP that dynamically adjusts the number of threads; and by up to 81%, 72%, and 91%, over the FDT.

Table 5.7: Number of threads found by the Oracle (exhaustive search)

		Performance				Energy				EDP			
		4-Core	*8-Core*	*24-Core*	*32-Core*	*4-Core*	*8-Core*	*24-Core*	*32-Core*	*4-Core*	*8-Core*	*24-Core*	*32-Core*
NB	***S***	3	4	4	3	3	3	4	3	3	3	4	3
	M	4	4	4	3	3	3	3	3	4	4	4	3
FFT	***S***	4,4	8,2	24,2	31,6	4,1	8,1	24,4	32,4	4,2	8,1	24,2	31,4
	M	4,4	8,2	24,6	32,14	4,1	8,1	24,2	32,4	4,2	8,2	24,2	32,4
ST	***S***	4	2	4	12	1	1	4	4	2	1	4	6
	M	4	2	4	6	1	1	4	4	2	1	4	6
UA	***S***	3,2,3,3	4,2,2,1	12,4,7,7	12,14,11,4	2,1,4,2	1,1,3,3	5,4,6,6	5,4,9,26	2,2,3,2	1,1,3,3	5,4,6,4	12,6,9,26
		2,4,1,3	2,4,1,2	5,10,2,4	13,15,10,14	2,2,2,3	2,8,2,3	5,24,1,3	7,7,3,8	2,4,2,3	2,2,2,3	5,24,2,3	7,7,4,8
		1,1,4,3	1,1,8,4	1,1,22,7	2,2,30,16	1,1,4,3	4,1,4,2	1,1,23,4	1,1,30,5	1,1,4,3	3,1,8,2	1,1,22,4	1,2,30,11
		1,4,4,4	1,4,4,4	1,12,12,6	2,31,16,11	2,3,4,2	4,3,3,2	1,24,4,6	3,32,10,7	2,4,4,2	4,4,3,2	1,24,4,6	3,32,10,11
		3,3,3,1	4,2,1,1	4,1,1,1	14,2,15,2	3,1,1,2	1,1,1,2	2,1,2,1	5,2,3,1	3,1,1,2	1,1,1,2	2,1,2,1	5,2,3,1
		3,4,1,1	1,4,1,1	1,7,1,1	11,16,2,2	2,2,1,1	4,3,1,1	1,6,1,1	3,6,1,4	2,2,1,1	4,4,1,1	2,1,6,1	3,6,1,4
		4,1,3,1	4,1,4,1	4,1,3,1	15,2,15,2	3,1,2,2	7,7,3,2	3,1,1,1	6,2,2,3	3,1,2,2	7,1,3,2	1,3,1,3	6,2,4,3
		3,1,4,4	2,1,2,8	3,1,6,24	15,3,9,32	3,1,1,4	5,1,1,8	1,1,6,24	6,1,6,32	3,1,2,4	5,1,1,8	1,1,1,6	6,1,6,32
		4,4,4,4	8,4,3,4	23,8,6,12	32,12,11,14	4,2,1,2	8,2,1,2	12,3,6,4	16,6,6,6	4,2,2,2	8,2,1,2	24,24,8,6	31,9,10,10
		4,4,3,4	8,8,2,8	12,24,5,12	32,32,12,30	4,4,1,4	4,8,2,1	11,24,3,12	32,32,5,32	4,4,2,4	8,8,2,8	6,11,24,3	32,32,8,30
		4,4,2,4	4,8,1,4	12,11,4,11	16,14,10,16	4,3,3,2	1,4,3,2	10,9,2,3	16,11,6,7	4,3,3,4	4,4,3,4	12,12,9,2	16,13,6,14
		2,4,4,3	2,2,8,2	4,6,12,2	4,6,32,10	2,1,4,1	1,1,8,1	4,2,12,2	4,5,32,2	1,2,4,2	1,1,8,1	11,4,6,12	4,5,32,4
		4,4,4,4	4,2,2,2	12,6,6,6	10,12,26,10	1,1,1,1	2,1,1,1	6,4,6,4	6,6,6,6	2,2,2,2	2,1,2,2	2,6,6,6	6,6,6,8
		4,4,4	4,8,4	12,24,12	12,32,16	1,4,4	1,8,1	6,24,12	10,16,7	2,4,4	1,8,4	6,24,6	10,32,16
	M	2,2,4,2	2,2,4,3	4,8,6,5	9,8,14,3	1,1,3,2	1,1,3,3	4,4,6,3	4,4,13,2	2,2,3,2	1,1,3,7	4,4,8,8	9,6,13,2
		3,4,2,4	4,8,2,2	9,12,4,4	16,8,16,16	1,4,1,2	3,8,2,2	8,8,2,4	16,16,5,4	3,4,1,2	3,8,2,2	8,8,2,4	16,5,5,4
		1,1,4,4	1,1,4,4	1,1,24,12	2,2,32,14	2,1,4,2	4,1,4,3	2,1,23,6	9,2,30,6	2,1,4,2	4,1,4,3	2,1,23,6	9,2,32,8
		1,4,4,4	1,4,4,4	3,24,12,11	2,32,14,16	1,3,2,2	1,2,3,2	2,24,10,6	5,30,8,8	1,4,4,2	1,4,4,2	2,10,6,4	5,32,13,11
		4,4,3,3	4,4,3,4	4,4,3,3	14,14,12,14	1,2,2,2	2,4,4,7	2,3,3,1	4,6,2,4	1,2,2,2	2,4,4,2	3,3,1,3	10,6,2,4
		4,4,1,1	4,4,1,1	3,9,1,1	14,16,2,3	2,4,1,1	4,4,1,1	3,6,1,2	9,11,1,1	2,4,1,1	4,4,1,1	7,1,2,6	9,11,1,1
		4,3,4,4	4,3,4,4	7,3,4,3	16,9,14,14	2,3,4,4	7,6,7,8	6,1,4,3	7,3,4,5	2,3,4,4	7,4,4,3	3,4,3,4	12,3,8,9
		3,2,4,4	4,2,2,8	3,1,12,24	14,3,12,32	2,3,1,4	4,5,1,8	4,2,4,24	6,2,6,32	2,3,2,4	4,5,1,8	2,4,24,24	6,2,6,32
		4,4,4,4	8,2,2,4	24,12,4,8	32,14,11,14	4,1,1,2	8,1,1,2	24,12,2,4	32,8,4,6	4,2,2,2	8,1,1,2	12,4,6,24	32,14,4,6
		4,4,3,4	8,8,3,8	24,24,4,24	16,32,13,32	4,4,2,4	4,8,2,8	11,24,2,24	13,32,4,32	4,4,2,4	8,8,2,8	24,4,24,12	16,32,3,32
		4,4,2,4	2,2,1,4	12,11,4,12	16,30,6,14	3,3,1,3	2,2,1,7	12,10,2,9	16,13,6,7	4,3,1,3	2,2,1,7	10,4,9,7	16,13,6,14
		2,4,4,3	2,5,8,4	7,22,24,4	12,13,32,12	1,1,4,2	1,1,4,2	7,2,12,2	5,4,15,3	2,2,4,2	1,2,8,2	4,24,4,6	6,6,32,6
		4,4,4,4	3,2,5,2	6,4,11,8	6,13,11,12	1,1,1,1	1,1,1,1	4,4,2,2	4,4,4,4	2,2,2,2	5,1,1,1	4,4,4,4	4,4,4,6
		4,4,4	3,8,4	22,24,18	13,32,27	1,4,2	1,8,7	4,24,4	4,32,30	2,4,4	1,8,7	24,20,8	32,30,16
JA	***S***	3	3	12	28	2	2	6	14	2	2	6	15
	M	3	2	10	12	2	2	4	6	2	2	4	6
SP	***S***	4,4,2,4	8,8,2,2	10,12,6,6	13,16,12,26	4,4,2,1	8,4,2,1	10,12,6,4	13,15,6,4	4,4,2,2	8,8,2,1	10,12,6,4	13,15,6,4
		4,2,3,2	4,2,4,2	12,6,12,6	15,32,15,32	3,1,3,1	3,1,3,1	10,4,10,4	13,4,13,4	3,2,3,2	4,1,4,1	12,6,12,6	15,4,15,4
		3,3,4,4	4,2,2,8	12,6,6,12	15,6,8,32	3,1,1,4	3,1,1,4	10,4,2,23	10,4,4,15	3,2,2,4	3,1,1,8	10,4,4,12	15,4,4,15
		4	4	10	32	2	2	4	6	4	2	4	6
	M	4,4,3,4	8,4,2,5	12,12,6,6	16,16,12,10	4,4,2,1	8,4,1,1	12,12,4,4	16,15,6,4	4,4,2,2	8,4,2,1	12,12,6,6	16,16,6,6
		2,2,2,2	2,2,2,5	6,20,6,20	10,32,10,14	2,1,2,1	2,1,2,1	6,4,6,4	8,4,8,4	2,2,2,2	2,2,2,2	6,6,6,6	8,4,8,4
		2,4,4,4	2,5,2,8	6,6,24,24	8,10,24,30	2,1,1,4	2,1,1,8	6,4,4,11 4	8,4,4,15 6	2,2,2,4	2,5,1,8	6,4,4,24 4	8,6,4,15 6
		4	4	24	32	2	4			4	4		
HS	***S***	4	4	12	16	4	4	10	16	3	4	12	16
	M	4	4	12	16	4	4	12	16	4	4	12	16
SC	***S***	4	8	23	9	4	8	7	9	4	8	7	9
	M	4	7	9	15	4	7	7	9	4	7	7	15
PO	***S***	4,4	8,8	12,7 12,7	29,16	2,4	7,7	12,5 12,7	29,7 16,6	2,4	7,7	12,5 12,7	29,8 16,11
		4,4	8,4		16,14	4,2	8,2			4,2	8,2		
	M	4,4	8,8	22,11	30,16	2,4	5,8	21,6 12,6	29,11	4,4	7,8	21,8 12,6	29,14
		4,3	7,2	12,8	16,16	2,1	3,1		16,6	2,1	3,1		16,14
HPCG	***S***	4,4,4,4	2,5,8,4	10,10,12,8	8,30,14,16	1,2,3,2	1,2,8,2	4,6,6,6	6,6,6,8	2,3,3,3	2,2,8,3	6,6,6,8	8,8,6,8
	M	—	2,4,8,4	6,12,12,20	8,32,16,22	—	1,3,3,2	4,8,6,6	4,8,8,6	—	2,3,8,2	4,12,8,6	6,8,10,8
MG	***S***	3,4,4,3	2,8,8,2	6,24,8,24	14,32,8,14	2,4,3,2	2,8,8,1	6,24,8,6	7,32,8,6	2,4,4,2	2,8,8,2	6,24,8,6	10,32,8,14
		3,4,2,4	4,8,2,3	4,24,6,8	12,1,8,10	3,3,2,2	1,6,2,2	5,8,6,6	5,8,6,8	3,4,2,3	1,6,2,3	5,24,6,8	5,8,8,8 8,6
		3,3	3,2	8,10	10,12	2,1	2,1	6,4	8,4	2,2	3,2	8,6	
	M	3,4,4,2	2,8,8,2	10,24,12,6	14,32,16,14	2,4,3,2	1,8,7,1	6,24,10,4	8,32,10,6	2,4,3,2	2,8,7,2	6,24,10,4	10,32,10,8
		4,4,4,4	3,6,3,4	6,12,12,10	14,16,16,16	2,3,2,2	1,6,2,2	4,10,4,6	6,8,6,8	3,3,2,3	1,6,2,3	4,10,6,6	10,10,6,8
		4,2	3,2	8,10	12,6	2,1	2,1	6,4	6,6	3,2	3,2	6,4	8,6
BT	***S***	4,4,2,4	4,8,2,8	12,12,6,23	15,15,10,28	4,4,2,4	4,8,2,8	12,10,6,22	15,13,6,28	4,4,2,4	4,8,2,8	12,12,6,22	15,15,8,28
		4,4,4,4	8,8,2,8	22,23,4,12	31,30,8,31	4,4,1,4	4,8,1,4	22,22,4,11	28,29,6,14	4,4,2,4	8,8,1,8	22,22,4,12	28,29,8,31
		4	4	14	28	2	1	6	30	2	2	6	30
	M	4,4,3,4	4,8,2,8	24,12,8,24	15,16,10,32	4,4,2,4	4,8,1,8	12,12,4,23	15,16,6,32	4,4,2,4	4,8,2,8	12,12,4,24	15,16,8,32
		4,4,4,4	8,8,2,8	24,24,12,24	32,32,32,30	4,4,1,4	8,8,1,8	23,23,4,11	32,32,4,15	4,4,2,4	8,8,1,8	23,23,6,24	32,32,4,30
		4	4	10	30	2	2	6	4	2	8	6	10
LU	***S***	4,4,4,2	4,8,8,2	10,23,12,6	13,32,15,12	3,4,4,2	3,8,4,1	8,23,10,6	12,15,10,8	4,4,4,2	3,8,4,2	12,24,12,6	12,28,15,12
		4,4,4,4	8,4,8,8	22,12,22,23	27,25,31,28	4,2,3,4	4,1,8,8	10,4,22,10	1,10,6,15	4,2,4,4	8,2,8,8	22,4,22,23	14,6,26,25
		4	4	8	12	4	3	8	9	4	3	8	9
	M	4,4,4,2	4,8,4,2	18,24,12,10	14,32,16,14	2,4,4,2	2,8,4,1	7,23,12,4	14,32,14,4	3,4,4,2	3,8,4,1	10,23,12,4	14,32,16,12
		4,4,4,4	8,4,4,8	12,10,24,23	14,23,32,32	4,2,2,4	4,2,2,8	10,6,24,23	10,6,32,32	4,2,3,4	4,2,3,8	12,10,24,23	14,9,32,32
		4	8	10	14	3	5	11	7	3	5	11	10
CG	***S***	4,4,4,4	8,8,8,4	24,24,4,3	32,32,10,16	4,4,4,4	8,8,8,4	24,12,13,15	16,16,3,13	4,4,4,4	8,8,8,4	24,24,13,15	32,16,7,13
		3,4,4	4,4,4	3,4,4	15,10,13	4,4,2	4,7,3	2,4,2	5,5,8	4,4,4	4,7,3	4,4,2	5,5,8
	M	4,4,4,4	8,8,4,4	24,24,6,7	31,32,16,16	4,4,4,4	8,8,4,4	24,24,2,5	32,14,4,5	4,4,4,4	8,8,4,4	24,24,2,5	31,32,4,5
		4,4,4	4,8,4	4,7,4	16,14,15	4,4,4	3,6,6	5,5,4	19,8,5	4,4,4	4,6,6	5,5,4	19,8,5
FT	***S***	2,4,4,4	2,8,8,4	6,24,24,24	8,32,32,32	2,4,4,4	2,8,8,4	6,24,24,24	8,32,32,32	2,4,4,4	2,8,8,4	6,24,24,24	8,32,32,32
		4,4,4,1	4,4,5,3	24,24,8,1	32,32,14,10	4,4,2,2	4,4,2,2	12,12,6,1	16,16,6,1	4,4,4,1	4,4,2,2	12,24,8,1	32,30,8,2
	M	2,4,4,4	2,8,8,7	6,24,24,24	8,32,32,32	2,4,4,4	2,8,8,4	6,24,24,24	8,32,32,32	2,4,4,4	2,8,8,7	6,24,24,24	8,32,32,32
		4,4,4,1	4,8,5,1	24,24,12,1	32,32,32,16	4,4,2,3	4,4,2,1	22,22,6,1	32,32,6,5	4,4,3,2	4,4,2,1	24,22,12,1	32,32,8,5

Source: The Author

Table 5.8: Learning overhead (%) for Aurora, FDT, and OMP_Dynamic

			Performance				**Energy**				**EDP**			
			4-Core	***8-Core***	***24-Core***	***32-Core***	***4-Core***	***8-Core***	***24-Core***	***32-Core***	***4-Core***	***8-Core***	***24-Core***	***32-Core***
Aurora	NB	S	1.2	1.4	0.1	2.3	1.0	2.5	0.1	0.1	0.1	0.1	0.1	0.1
		M	10.9	0.1	0.1	3.4	1.0	0.2	2.2	0.2	8.0	0.1	4.0	3.7
	FFT	S	1.6	1.9	2.0	4.9	7.1	3.8	6.0	7.0	8.8	5.8	3.5	4.8
		M	2.2	2.2	3.0	1.1	4.6	4.9	12.0	9.0	6.9	7.3	12.0	8.0
	ST	S	0.1	0.1	3.5	0.8	0.1	0.1	0.1	7.9	0.1	0.1	0.1	2.7
		M	2.2	0.1	0.1	0.4	0.1	2.5	0.1	0.1	0.1	0.1	0.1	0.1
	UA	S	0.9	2.8	29.3	39.7	13.4	8.7	13.2	38.2	14.4	11.8	46.3	52.0
		M	0.7	1.5	16.0	18.7	1.6	6.4	3.4	34.4	2.3	8.0	19.9	29.0
	JA	S	0.1	0.1	0.1	20.8	0.1	4.4	0.1	0.1	0.1	0.1	0.1	13.6
		M	0.1	0.1	0.1	1.2	0.1	7.3	0.1	9.6	3.6	0.1	0.1	10.7
	SP	S	0.7	2.4	8.0	36.6	2.2	2.4	6.5	24.3	2.9	0.7	15.0	27.0
		M	0.9	4.9	2.3	7.3	2.0	4.9	5.0	12.9	2.9	6.1	7.4	21.2
	HS	S	0.1	0.2	4.1	6.0	0.1	3.6	0.1	4.3	1.0	2.5	0.1	0.1
		M	4.6	2.3	0.9	6.3	3.2	8.0	1.8	0.1	4.4	8.3	0.1	3.6
	SC	S	0.1	0.1	3.5	15.3	0.1	0.1	2.6	32.0	0.5	0.1	3.3	9.9
		M	0.1	0.1	0.1	0.1	2.0	0.1	1.1	5.0	1.8	0.1	2.0	0.1
	PO	S	1.2	0.6	3.7	5.3	3.2	1.0	0.1	0.5	4.4	1.6	3.8	5.8
		M	2.0	1.2	2.4	4.0	3.1	4.7	0.1	1.7	5.2	5.9	2.5	1.6
	HPCG	S	0.1	1.6	0.1	5.69	3.5	3.0	3.5	20.5	3.5	4.6	3.6	27.2
		M	–	0.5	1.0	5.1	–	3.6	10.0	5.9	–	4.1	6.1	11.3
	MG	S	5.0	12.2	31.6	35.0	3.3	8.0	11.7	9.0	8.5	21.0	47.2	19.0
		M	7.0	7.9	15.0	19.0	2.1	4.8	9.9	19.0	9.3	13.0	18.2	23.0
	BT	S	0.7	1.0	3.5	10.2	0.6	1.0	3.5	6.9	1.3	2.0	5.8	17.8
		M	0.5	1.0	2.6	3.9	0.8	2.7	4.3	6.0	1.3	3.7	5.1	10.2
	LU	S	3.2	0.5	1.4	4.9	7.0	0.7	6.1	7.2	10.4	1.2	4.6	12.4
		M	0.3	0.3	2.2	1.3	1.1	0.6	5.6	4.2	1.3	0.9	8.0	5.5
	CG	S	1.4	1.1	18.6	21.1	1.1	0.5	14.8	15.6	2.6	1.7	8.4	40.0
		M	1.0	1.2	15.6	4.5	1.2	0.6	11.0	14.9	2.2	2.0	0.4	20.0
	FT	S	4.8	8.6	28.0	23.5	2.7	6.0	9.8	6.7	7.7	15.2	18.2	25.4
		M	4.2	6.2	22.7	9.4	1.7	4.0	5.2	6.6	6.0	10.5	15.5	10.1
	Gmean	S	**0.7**	**0.9**	**2.9**	**9.9**	**0.9**	**1.4**	**0.9**	**4.1**	**1.8**	**2.1**	**2.6**	**6.6**
		M	**1.0**	**0.7**	**2.4**	**3.1**	**0.9**	**2.0**	**1.6**	**3.0**	**2.8**	**1.9**	**2.5**	**5.4**
OMP_Dynamic - Gmean		S	**25.1**	**13.7**	**21.0**	**24.2**	**15.1**	**14.6**	**25.3**	**37.0**	**55.0**	**35.5**	**52.5**	**70.5**
		M	**19.3**	**12.9**	**14.0**	**17.1**	**13.8**	**15.1**	**30.3**	**34.7**	**43.8**	**30.7**	**42.9**	**62.5**
FDT - Gmean		S	**4.4**	**12.6**	**24.0**	**35.0**	**11.8**	**14.7**	**43.4**	**43.6**	**13.1**	**31.1**	**101.1**	**111.5**
		M	**8.6**	**22.0**	**22.7**	**33.5**	**18.2**	**24.7**	**45.6**	**43.6**	**34.0**	**56.5**	**106.8**	**109.5**

Source: The Author

6 CONCLUSIONS AND FUTURE WORK

In this Ph.D. thesis, a mechanism to automatically and transparently improve the execution of parallel applications was proposed, evaluated, and validated with respect to well-known techniques.

In order to achieve the main objective of this work, we first performed a static exploration for optimal combinations of processors, communication models, and TLP exploitation to reach the best results in performance, energy, and EDP, presented in Chapter 4. By considering a great number of variables (5 multicore processors; 14 parallel benchmarks; four parallel programming interfaces; different levels of TLP exploitation; and different levels of static power of the processor), we showed that there is no single combination of all the variables that offers the best result for performance and energy at the same time. However, we found significant results that demonstrate the large space of exploration and the needed for developing a mechanism to optimize parallel applications at runtime.

As the development of a single mechanism that comprises all the PPIs discussed in Chapter 4 is not feasible at this moment, due to the way the parallelism is exploited in each one of them, we chose to develop a mechanism for OpenMP applications due to the following reasons: parallelism exploitation with OpenMP usually requires less effort when compared to other PPIs, making it more appealing to software developers; OpenMP is widely used and there are many parallel benchmarks implemented with it; OpenMP is more suitable for approaches in which the goal is to provide the maximum transparency as possible to the programmer, as the whole process of thread management and workload distribution are done by functions implemented in the OpenMP library, in opposite to other PPIs (e.g., PThreads, MPI, TBB).

We first proposed LAANT, a library to automatically adapt the number of threads of OpenMP applications. It is capable of finding at run-time the ideal number of threads for each parallel region of the application. LAANT can be applied to any OpenMP application that exploits parallel loops, by simply annotating code in the parallel regions. Nine parallel applications from assorted benchmarks suites and domains, with different input sets, were chosen. LAANT was compared to the usual way that parallel applications are executed (**Baseline**) and with the OpenMP dynamic feature (*OMP_Dynamic*). By setting LAANT to optimize the EDP of such applications running on three distinct multicore architectures, we have shown that LAANT can reduce the EDP in 29%, on average when

compared to the **Baseline** and in up to 82% when compared to the *OMP_Dynamic*. Although LAANT achieves positive results, it relies on the need for code annotation and recompilation. That is, applications already compiled with OpenMP will not be benefited by LAANT. Besides that, LAANT is very useful in situations in which the replacement of the OpenMP library is not desirable (e.g., when a commercial solution is available, as Intel's libomps). Therefore, LAANT would come to the rescue i this cases by allowing the algorithm to build on the standard *omp_set_numthreads* operation.

In order to provide transparency and binary compatibility to the user, that is, alleviate the burden of the user's side, LAANT was extended to Aurora. Given an OpenMP application already compiled, Aurora is capable of adapt the number of threads without the need for any modifications in the source code nor code recompilation. Therefore, existent OpenMP applications do not need to be annotated, recompiled or pass through any code transformation. Such transparency is achieved by redirecting the calls originally targeted for the dynamically linked OpenMP library to Aurora. This retargeting is configured by simply setting an environment variable in the Operating System.

Aurora was validated executing fifteen well-known benchmarks on four distinct multicore processors. With DVFS set to *ondemand*, we compared Aurora to the execution with the maximum possible number of threads; to the OpenMP dynamic (*OMP_Dynamic*); and to the well know feedback-driven threading (FDT) framework. We showed gains of 79%, 89%, 98% (performance, energy, and EDP, respectively) for the maximum number of threads; 62%, 61%, and 86% for the OpenMP dynamic; and 81%, 72%, and 91% for FDT. To measure the cost of the learning curve to converge for the ideal number of threads brought by its dynamic adaptation, we also compared Aurora to the Oracle solution, which comprises the execution of each parallel region with the optimal number of threads without the cost of the learning curve. In this comparison, the average cost to optimize performance, energy, and EDP for the whole benchmark set and processors is of only 1.8%, 1.6%, and 2.9%, respectively.

6.1 Extending Aurora

This section discusses some of the most promising future works envisioned at this time.

6.1.1 Energy Optimization Techniques

Aurora can be extended in order to support energy optimization techniques, such as the adaptation of the processor frequency (through DVFS), and the number of active cores (through power gating). Both techniques are briefly described bellow.

6.1.1.1 Dynamic Voltage and Frequency Scaling

DVFS is a feature of the processor that allows software to adapt the clock frequency and operating voltage of a processor on the fly without requiring a reset. DVFS enables software to change System on Chip processing performance to attain low power consumption while meeting the performance requirements [1]. The main idea of the DVFS is dynamically scaling the supply voltage of the CPU for a given frequency, so that it operates at a minimum speed required by the specific task executed (SUEUR; HEISER, 2010). Therefore, this can yield a significant reduction in power consumption because of the V^2 relationship shown in the Equation 6.1.

$$P_{dynamic} = CV^2Af \tag{6.1}$$

Reducing the operating frequency reduces the processor performance and the power consumption per second. Also, when reducing the voltage, the leakage current from the CPU's transistors decreases, making the processor most energy-efficient resulting in further gains (ROSSI et al., 2015). However, determining the ideal frequency and voltage for a given point of execution is not a trivial task. To make the DVFS management as transparent as possible to the software developer, Operating Systems provide frameworks that allow each CPU core to have a min/max frequency, and a governor that governs it. Governors are kernels models that can drive CPU core frequency/voltage operating points. Currently, the available governors are:

- **Performance**: the frequency of the processor is always fixed at the maximum, even if the processor is underutilized.
- **Powersave**: the frequency of the processor is always fixed at the minimum allowable frequency.

[1] Available at: http://www.ti.com/lit/an/slva646/slva646.pdf

- **Userspace**: allows the user, or any userspace program running to set the CPU for a specific frequency.
- **Ondemand**: the frequency of the processor is adjusted as the workload behavior within the range of frequencies allowed.
- **Conservative**: the frequency of the processor is adjusted based on the workload but it is a bit more conservative than the ondemand, by adjusting the frequency gradually.

In addition, it is possible to set the processor frequency level manually, by editing the configurations of the CPU frequency driver[2] .

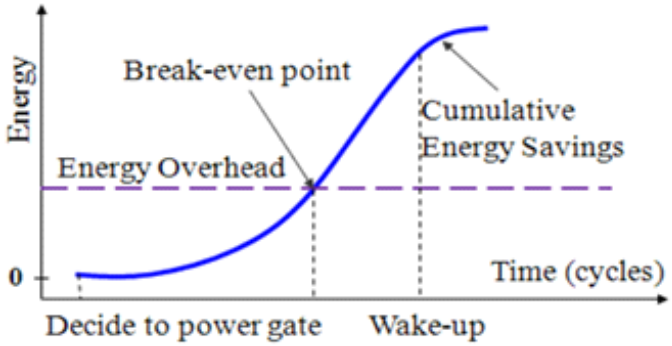
6.1.1.2 Power Gating

Power gating consists of selectively powering down certain blocks in the chip while keeping other blocks powered up. The goal of power gating is to minimize leakage current by temporarily switching power off to blocks that are not required in the current operating mode (KEATING et al., 2007). Power gating can be applied either at the unit-level, reducing the power consumption of unused core functional units or at the core-level, in which entire cores may be power gated (MADAN et al., 2011) (KAHNG et al., 2013). Currently, power gating is mainly used in multicore processors to switch off unused cores to reduce power consumption (OBORIL; TAHOORI, 2012).

Power gating requires the presence of a header "sleep" transistor that can set the supply voltage of the circuit to ground level during idle times. Power gating also requires a control logic that decides at which time power gate the circuit. Every time that the power gating is applied, an energy overhead cost occurs due to distributing the sleep signal to the header transistor before the circuit is turned off; and turning off the sleep signal and driving the voltage when the circuit is powered-on again. To cope with this additional cost, a break-even point must be considered, which is the point when the leakage energy savings equals the energy overhead of switching the circuit on and off. Figure 6.1 depicts an example of this scenario, in which the break-even point on the figure represents the point in time where the cumulative leakage energy savings equals to the energy overhead incurred by power gating. If, after the decision to power gate a unit, the unit stays idle for a time interval that is longer than the break-even point, then power gating saves energy. On the other hand, if the unit needs to be active again before the break-even point is reached, then power gating incurs an energy penalty (LUNGU et al., 2009).

[2]Description available at: https://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq-set.html

Figure 6.1: Break-even point



Source: (LUNGU et al., 2009)

6.1.1.3 Scenarios for Optimization

Considering the large space of exploration regarding the combination of the number of threads, processor frequency, and the active number of cores, Table 6.1 depicts the possible scenarios.

Scenario I: it comprises the adaptation of the number of threads at runtime, as already implemented by LAANT and Aurora. In such scenario, the processor frequency can be configured depending on the user requirements: *performance*, *powersave*, *ondemand*, *conservative*, and *userspace* (Section 6.1.1.1). Once configured, the processor frequency will not change at runtime. Also, this scenario does not consider the application of power gating on the unused cores.

Scenario II: this scenario comprises the adaptation of the processor frequency level at runtime. In this way, the application will start its execution with the default number of threads (previously defined by the programmer or the number of cores available on the platform) and this number will not change at runtime. The changes in the processor frequency level can be applied in different ways. For example, when memory-bound applications are being executed, the processor frequency can be reduced to save energy with no performance loss. In such case, it could be better to set the DVFS governor for power-saving (Section 6.1.1.1). On the other hand, when CPU-bound applications

Table 6.1: Dscription of the possible scenarios for optimization

Scenario	Description
I	Adaptation of the number of threads
II	DVFS-Only
III	Adaptation of the number of threads + DVFS
IV	Adaptation of the number of threads + power gating
V	Adaptation of the number of threads + DVFS + power gating

Source: The Author

are being executed, a higher processor frequency can be better, which can be done by setting the DVFS governor for performance (Section 6.1.1.1). There will also be cases of applications that may fluctuate their behavior during execution. As Figure 6.2 shows, such applications can have regions where it is best to set the processor frequency for the minimum value and regions where the maximum frequency is the best choice. In this case, setting the DVFS governor for ondemand (Section 6.1.1.1) may be the best choice.

Scenario III: it comprises the adaptation of the number of threads and the processor frequency level. In this case, different approaches to find the best combination that offers the best result can be used:

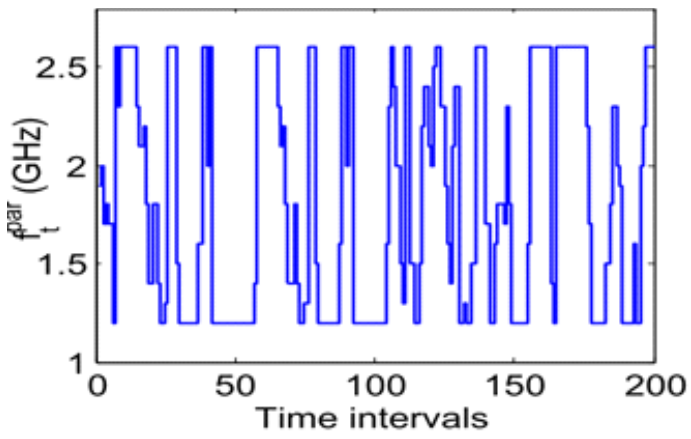
a Definition of the best number of threads first, and after, the definition of the best processor frequency;

b Definition of the processor frequency first, and then, the definition of the best number of threads; and

c Definition of the best number of threads and optimal point of processor frequency together.

The most common technique could be the use of the approaches *a* and *b*. However, they can lead to incorrect results. For instance, let us consider an eight-core processor with 10 levels of processor frequency. Using the approach (*a*), it may be that the best result found is the execution with 4 threads and, after that, the definition of the processor frequency at the maximum. However, the best configuration for the application is the use of 8 threads with the frequency set to the minimum. Therefore, to avoid this behavior, the

Figure 6.2: Processor frequency behavior when using the *ondemand* governor



Source: (SHAFIK et al., 2015a)

ideal technique to be implemented must be the approach (*c*), where the best point of the number of threads and processor frequency are found together.

Scenario IV: It comprises the adaptation of the number of threads together with the power gating technique. When an OpenMP application is running, it is possible to bind threads to specific cores (using OpenMP functions). Therefore, depending on the application behavior, the unused cores could be turned off to save energy. As an example, lets consider an application that has two parallel regions running on an eight-core processor: the first one is better executed with six threads while the latter executes better with only two threads. In this way, when the application is running with six threads on six cores, the other two cores can be turned off to save energy. The same scenario happens in the second parallel region, where the other six cores can be turned off. In this scenario, an important information must be considered before applying the power gating: the break-even point. As Figure 6.1 shows, it represents the point in time where the cumulative leakage energy savings equals to the energy incurred by power gating. If, after the decision to power gate a core, the core stays idle for a time interval that is longer than the break-even point, then power gating saves energy. On the other hand, if the core needs to be active again before the break-even point is reached, then power gating incurs in some energy penalty (LUNGU et al., 2009).

Scenario V: This last scenario considers the union of the three techniques presented previously. In this case, the mechanism is able to define the best number of threads, the optimal point of processor frequency, and apply power gating for the cores that are not running the application.

6.1.2 Support for Different PPIs and Heterogeneous Architectures

As early discussed in Chapter 3, the PPIs widely used nowadays correspond to OpenMP, PThreads, Cilk++, and Intel TBB for shared memory environments; MPI for distributed memory; and the PPIs that exploit parallelism in GPUs: CUDA, OpenACC, and OpenCL. The same Chapter also showed that homogeneous processors have different behavior regarding performance, energy, and EDP; and that this behavior changes according to the PPI used, the number of threads, and the application characteristic.

However, the DSE in heterogeneous systems is much broader than in the homogeneous architectures that use communication through shared memory. For example, a System on Chip that implements a Network-on-Chip (NoC) with DVFS support and runs

OpenMP applications may have better energy efficiency when compared to a heterogeneous architecture with high-performance computers and GPUs running with CUDA.

Considering this scenario, Aurora can be extended in order to investigate the DSE of all the possible solutions at runtime when different PPIs and heterogeneous architectures are being considered and choose the best configuration with respect to a given optimization metric. In this case, it is necessary to incorporate the following functionalities into Aurora:

- **Support for different PPIs**: Aurora should be able to deal with different communication models and process/thread management. For example, In the same way that Aurora supports communication through shared variables (OpenMP), it has to be able to deal with data exchange through message passing, without incurring in problems with the application execution.

- **Support for heterogeneous architectures**: Aurora should consider that different architectures will often require different application binaries, due to the particularities of each ISA and its optimization flags. For example, if the system has a GPP Intel Core *i*7 with one GPU and one Intel Co-Processor Xeon Phi, there are different optimization flags that can be used to each device. Therefore, Aurora should be able to deal with this situation.

6.2 Publications Regarding the Scope of this Thesis

In this subsection, the publications regarding the scope of this thesis and work product of cooperation will be highlighted:

Exploration of parallel computing opportunities:

Conference Papers:

- On the Influence of Static Power Consumption in Multicore Embedded Systems. In: *IEEE International Symposium on Circuits and Systems* - (LORENZON; CERA; BECK, 2015a)

- The Influence of Parallel Programming Interfaces on Multicore Embedded Systems. In: *IEEE Computer Society International Conference on Computers, Software & Applications* - (LORENZON et al., 2015a)

- Optimized Use of Parallel Programming Interfaces in Multithreaded Embedded Architectures. In: *IEEE Computer Society Annual Symposium on VLSI* - (LORENZON et al., 2015b)

Journal Articles:

- Performance and Energy Evaluation of Different Multi-Threading Interfaces in Embedded and General-Purpose Processors. In: *Journal of Signal Processing Systems* - (LORENZON; CERA; BECK, 2015c)

- Investigating Different General-Purpose and Embedded Multicores to Achieve Optimal Trade-offs between Performance and Energy. In: *Journal of Parallel and Distributed Computing* - (LORENZON; CERA; BECK, 2016)

Automatically adjusting OpenMP applications:

Conference Paper:

- LAANT: A Library to Automatically Optimize EDP for OpenMP Applications. In: Design, Automation and Test in Europe (DATE) - (LORENZON; SOUZA; BECK, 2017)

Joint Cooperation:

Conference Paper:

- Adaptive and Polymorphic VLIW Processor to Optimize Fault Tolerance, Energy Consumption, and Performance - In ACM International Conference on Computing Frontiers 2018

- Potential Gains in EDP by Dynamically Adapting the Number of Threads for OpenMP Applications in Embedded Systems - *Brazilian Symposium on Computing Systems Engineering*(SCHWARZROCK et al., 2017)

- Improving EDP in Multi-Core Embedded Systems through Multidimensional Frequency Scaling. Submitted to: IEEE International Symposium on Circuits and Systems - (MARQUES et al., 2017)

- How Programming Languages and Paradigms Affect Performance and Energy in Multithreaded Applications. In *Brazilian Symposium on Computing Systems Engineering* - (MAGALHÃES et al., 2016)

- Improved Dynamic Cache Sharing for Communicating Threads on a Runtime-Adaptable Processor - In *11th HiPEAC Workshop on Reconfigurable Computing (WRC2017)* (HOOZEMANS et al., 2017)

- The Impact of Virtual Machines on Embedded Systems. In *IEEE Computer Society International Conference on Computers, Software & Applications* - (SARTOR; LORENZON; BECK, 2015)

- A Novel Phase-based Low Overhead Fault Tolerance Approach for VLIW Processors. In *IEEE Computer Society Annual Symposium on VLSI* - (SARTOR et al., 2015)

- A Sparse VLIW Instruction Encoding Scheme Compatible with Generic Binaries. In International Conference on ReConFigurable Computing and FPGAs - (BRANDON et al., 2015)

Journal Articles:

- Under Review (Major review): How Object Oriented Programming and Virtual Machines Influence Parallel Applications. In *International Journal of High Performance Systems Architecture*

- Exploiting the Idle Hardware to Provide Low Overhead Fault Tolerance for VLIW Processors. In *ACM Journal on Emerging Technologies in Computer Systems* - (SARTOR et al., 2017)

REFERENCES

ABRAHAM, M. J. et al. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. **SoftwareX**, v. 1–2, p. 19 – 25, 2015. ISSN 2352-7110.

ADYA, A. et al. Cooperative task management without manual stack management. In: **Annual Conf. on USENIX**. CA, USA: USENIX Association, 2002. p. 289–302. ISBN 1-880446-00-6.

AJKUNIC, E. et al. A comparison of five parallel programming models for c++. In: **Int. Convention MIPRO**. [S.l.: s.n.], 2012. p. 1780–1784.

ALESSI, F. et al. Application-level energy awareness for openmp. In: SPRINGER. **International Workshop on OpenMP**. [S.l.], 2015. p. 219–232.

ASLOT, V. et al. Specomp: A new benchmark suite for measuring parallel computer performance. In: ____. **OpenMP Shared Memory Parallel Programming: Int. Workshop on OpenMP Applications and Tools**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 1–10. ISBN 978-3-540-44587-6.

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.

BALLADINI, J. et al. Impact of parallel programming models and cpus clock frequency on energy consumption of hpc systems. In: IEEE. **Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on**. [S.l.], 2011. p. 16–21.

BARNES, B. J. et al. A regression-based approach to scalability prediction. In: **Proceedings of the 22Nd Annual International Conference on Supercomputing**. New York, NY, USA: ACM, 2008. (ICS '08), p. 368–377. ISBN 978-1-60558-158-3.

BASMADJIAN, R.; MEER, H. de. Evaluating and modeling power consumption of multi-core processors. In: **2012 Third International Conference on Future Systems: Where Energy, Computing and Communication Meet (e-Energy)**. [S.l.: s.n.], 2012. p. 1–10.

BEDARD, D. et al. Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In: **Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)**. [S.l.: s.n.], 2010. p. 479–484. ISSN 1091-0050.

BENEDICT, S. et al. Energy prediction of openmp applications using random forest modeling approach. In: **2015 IEEE International Parallel and Distributed Processing Symposium Workshop**. [S.l.: s.n.], 2015. p. 1251–1260.

BENESTY, J. et al. Pearson correlation coefficient. In: ____. **Noise Reduction in Speech Processing**. Berlin, Heidelberg: Springer, 2009. p. 1–4. ISBN 978-3-642-00296-0.

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

BHATTACHARJEE, A.; MARTONOSI, M. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 37, n. 3, p. 290–301, jun. 2009. ISSN 0163-5964.

BIENIA, C. et al. The parsec benchmark suite: Characterization and architectural implications. In: **Int. Conf. on Parallel Architectures and Compilation Techniques**. NY, USA: ACM, 2008. p. 72–81. ISBN 978-1-60558-282-5.

BLAKE, G. et al. Evolution of thread-level parallelism in desktop applications. **SIGARCH Comput. Archit. News**, ACM, NY, USA, v. 38, n. 3, p. 302–313, 2010. ISSN 0163-5964.

BLEM, E.; MENON, J.; SANKARALINGAM, K. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In: **2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2013. p. 1–12. ISSN 1530-0897.

BODE, A. Energy to solution: A new mission for parallel computing. In: WOLF, F.; MOHR, B.; MEY, D. an (Ed.). **Euro-Par 2013 Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 1–2. ISBN 978-3-642-40047-6.

BRANDON, A. et al. A sparse vliw instruction encoding scheme compatible with generic binaries. In: IEEE. **ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on**. [S.l.], 2015. p. 1–7.

BROWNE, S. et al. A portable programming interface for performance evaluation on modern processors. **Int. J. High Perform. Comput. Appl.**, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 14, n. 3, p. 189–204, aug. 2000. ISSN 1094-3420.

BUONO, D. et al. Optimizing message-passing on multicore architectures using hardware multi-threading. In: **2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. [S.l.: s.n.], 2014. p. 262–270. ISSN 1066-6192.

BUTENHOF, D. R. **Programming with POSIX Threads**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-63392-2.

CABRERA, A. et al. Analytical modeling of the energy consumption for the high performance linpack. In: **2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. [S.l.: s.n.], 2013. p. 343–350. ISSN 1066-6192.

CAO, Y. et al. Performance analysis of current parallel programming models for many-core systems. In: IEEE. **Computer Science & Education (ICCSE), 2013 8th International Conference on**. [S.l.], 2013. p. 132–135.

CERA, M. et al. Improving the dynamic creation of processes in mpi-2. **Recent Advances in Parallel Virtual Machine and Message Passing Interface**, Springer, p. 247–255, 2006.

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.

CHANDRAMOWLISHWARAN, A.; KNOBE, K.; VUDUC, R. Performance evaluation of concurrent collections on high-performance multicore computing systems. In: **2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)**. [S.l.: s.n.], 2010. p. 1–12. ISSN 1530-2075.

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 Int. Symp. on Workload Characterization**. DC, USA: IEEE Computer Society, 2009. p. 44–54. ISBN 978-1-4244-5156-2.

CHOU, C.-Y. et al. An improved model for predicting hpl performance. In: CÉRIN, C.; LI, K.-C. (Ed.). **Advances in Grid and Pervasive Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 158–168. ISBN 978-3-540-72360-8.

CURTIS-MAURY, M. et al. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In: ACM. **Proceedings of the 20th annual international conference on Supercomputing**. [S.l.], 2006. p. 157–166.

CURTIS-MAURY, M. et al. Prediction models for multi-dimensional power-performance optimization on many cores. In: ACM. **Proceedings of the 17th international conference on Parallel architectures and compilation techniques**. [S.l.], 2008. p. 250–259.

DONGARRA, J.; HEROUX, M. A.; LUSZCZEK, P. **Hpcg benchmark**: A new metric for ranking high performance computing systems. In: . Knoxville, Tennessee: [s.n.], 2015. p. 1–11.

DONGARRA, J. J. The linpack benchmark: An explanation. In: **Proceedings of the 1st International Conference on Supercomputing**. New York, NY, USA: Springer-Verlag New York, Inc., 1988. p. 456–474. ISBN 0-387-18991-2.

DUKAN, P.; KOVARI, A.; KATONA, J. Low consumption and high performance intel, amd and arm based mini pcs. In: IEEE. **Computational Intelligence and Informatics (CINTI), 2014 IEEE 15th International Symposium on**. [S.l.], 2014. p. 127–131.

ESMAEILZADEH, H. et al. Power limitations and dark silicon challenge the future of multicore. **ACM Trans. Comput. Syst.**, ACM, New York, NY, USA, v. 30, n. 3, p. 11:1–11:27, aug. 2012. ISSN 0734-2071.

FASIKU, A. et al. Performance evaluation of multicore processors. **International Journal of Engineering and Technology**, v. 4, n. 1, p. 73–84, 2014.

FELLOWS, K. **A comparative study of the effects of parallelization on ARM and Intel based platforms**. Thesis (PhD), 2014. Available from Internet: <http://hdl.handle.net/2142/50710>.

FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201575949.

GONZALEZ, R.; HOROWITZ, M. Energy dissipation in general purpose microprocessors. **IEEE Journal of Solid-State Circuits**, v. 31, n. 9, p. 1277–1284, Sep 1996. ISSN 0018-9200.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface**. Cambridge, MA, USA: MIT Press, 1999. ISBN 0-262-57132-3.

HACKENBERG, D. et al. Power measurement techniques on standard compute nodes: A quantitative comparison. In: **IEEE Int. Symp. on Performance Analysis of Systems and Software**. [S.l.: s.n.], 2013. p. 194–204.

HACKENBERG, D. et al. Power measurement techniques on standard compute nodes: A quantitative comparison. In: **2013 IEEE International Symposium on Performance Analysis of Systems and Software (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.

HAM, T. J. et al. Disintegrated control for energy-efficient and heterogeneous memory systems. In: **IEEE Int. Symp. on High Performance Computer Architecture**. [S.l.: s.n.], 2013. p. 424–435. ISSN 1530-0897.

HANAWA, T. et al. Evaluation of multicore processors for embedded systems by parallel benchmark program using openmp. In: **Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism**. Berlin, Heidelberg: Springer-Verlag, 2009. (IWOMP '09), p. 15–27. ISBN 978-3-642-02284-5.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 3. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 1558607242.

HEUVELINE, V.; LATT, J. The openlb project: An open source and object oriented implementation of lattice boltzmann methods. **Int. Journal of Modern Physics C**, v. 18, n. 4, p. 627–634, 2007.

HOEFLER, T.; LUMSDAINE, A.; REHM, W. Implementation and performance analysis of non-blocking collective operations for mpi. In: **Proceedings of the 2007 ACM/IEEE Conference on Supercomputing**. New York, NY, USA: ACM, 2007. (SC '07), p. 52:1–52:10. ISBN 978-1-59593-764-3.

HOOZEMANS, J. et al. Improved dynamic cache sharing for communicating threads on a runtime-adaptable processor. In: **HiPEAC Workshop on Reconfigurable Computing (WRC2017)**. [S.l.: s.n.], 2017. p. 1–4.

HUA, S.; YANG, Z. Comparison and analysis of parallel computing performance using openmp and mpi. **Open Automation and Control Systems Journal**, v. 5, n. 1, 2013.

IPEK, E. et al. An approach to performance prediction for parallel applications. In: **Proceedings of the 11th International Euro-Par Conference on Parallel Processing**. Berlin, Heidelberg: Springer-Verlag, 2005. (Euro-Par'05), p. 196–205. ISBN 3-540-28700-0, 978-3-540-28700-1.

IQBAL, S. M. Z.; LIANG, Y.; GRAHN, H. Parmibench - an open-source benchmark for embedded multiprocessor systems. **IEEE Computer Architecture Letters**, v. 9, n. 2, p. 45–48, Feb 2010. ISSN 1556-6056.

JARUS, M. et al. Performance evaluation and energy efficiency of high-density hpc platforms based on intel, amd and arm processors. In: SPRINGER. **European Conference on Energy Efficiency in Large Scale Distributed Systems**. [S.l.], 2013. p. 182–200.

JAYAKUMAR, A.; MURALI, P.; VADHIYAR, S. Matching application signatures for performance predictions using a single execution. In: **2015 IEEE International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2015. p. 1161–1170. ISSN 1530-2075.

JOAO, J. A. et al. Bottleneck identification and scheduling in multithreaded applications. In: **Int. Conf. on Architectural Support for Programming Languages and Operating Systems**. NY, USA: ACM, 2012. p. 223–234. ISBN 978-1-4503-0759-8.

JORDAN, H. et al. A multi-objective auto-tuning framework for parallel codes. In: IEEE. **Int. Conf. for High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2012. p. 1–12.

JUNG, C. et al. Adaptive execution techniques for smt multiprocessor architectures. In: ACM. **Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming**. [S.l.], 2005. p. 236–246.

KAHNG, A. B. et al. Many-core token-based adaptive power gating. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 32, n. 8, p. 1288–1292, Aug 2013. ISSN 0278-0070.

KARLIN, I.; KEASLER, J.; NEELY, R. **LULESH 2.0**: Updates and changes. In: . [S.l.: s.n.], 2013. p. 1–9.

KAXIRAS, S.; MARTONOSI, M. **Computer Architecture Techniques for Power-Efficiency**. 1st. ed. [S.l.]: Morgan and Claypool Publishers, 2008. ISBN 1598292080, 9781598292084.

KEATING, M. et al. **Low Power Methodology Manual: For System-on-Chip Design**. [S.l.]: Springer Publishing Company, Incorporated, 2007. ISBN 0387718184, 9780387718187.

KONTORINIS, V. et al. Reducing peak power with a table-driven adaptive processor core. In: **Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture**. New York, NY, USA: ACM, 2009. (MICRO 42), p. 189–200. ISBN 978-1-60558-798-1.

KORTHIKANTI, V. A.; AGHA, G. Towards optimizing energy costs of algorithms for shared memory architectures. In: **SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010**. [S.l.: s.n.], 2010. p. 157–165.

KULKARNI, M. et al. Lonestar: A suite of parallel irregular programs. In: **2009 IEEE International Symposium on Performance Analysis of Systems and Software**. [S.l.: s.n.], 2009. p. 65–76.

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

LEE, K. M. et al. Openmp parallel programming using dual-core embedded system. In: IEEE. **Control, Automation and Systems (ICCAS), 2011 11th International Conference on**. [S.l.], 2011. p. 762–766.

LEVY, H. M. et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In: **Int. Symp. on Computer Architecture**. [S.l.: s.n.], 1996. p. 191–191. ISSN 1063-6897.

LI, D. et al. Strategies for energy-efficient resource management of hybrid programming models. **IEEE Transactions on Parallel and Distributed Systems**, v. 24, n. 1, p. 144–157, Jan 2013. ISSN 1045-9219.

LIMA, J. V. F. et al. Performance and energy analysis of openmp runtime systems with dense linear algebra algorithms. In: **2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)**. [S.l.: s.n.], 2017. p. 7–12.

LIVELY, C. et al. Energy and performance characteristics of different parallel implementations of scientific applications on multicore systems. **The International Journal of High Performance Computing Applications**, SAGE Publications Sage UK: London, England, v. 25, n. 3, p. 342–350, 2011.

LORENZON, A. et al. Análise de interfaces de programação paralela na determinação de similaridades de histogramas. **XII WSCAD-WIC, IEEE Computer Society**, 2011.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. On the influence of static power consumption in multicore embedded systems. In: IEEE. **Circuits and Systems (ISCAS), 2015 IEEE International Symposium on**. [S.l.], 2015. p. 1374–1377.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Performance and energy evaluation of different multi-threading interfaces in embedded and general purpose systems. **J. Signal Process. Syst.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 80, n. 3, p. 295–307, sep. 2015. ISSN 1939-8018.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Performance and energy evaluation of different multi-threading interfaces in embedded and general purpose systems. **Journal of Signal Processing Systems**, Springer, v. 80, n. 3, p. 295–307, 2015.

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

LORENZON, A. F. et al. The influence of parallel programming interfaces on multicore embedded systems. In: IEEE. **Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual**. [S.l.], 2015. v. 2, p. 617–625.

LORENZON, A. F. et al. Optimized use of parallel programming interfaces in multithreaded embedded architectures. In: IEEE. **VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on**. [S.l.], 2015. p. 410–415.

LORENZON, A. F.; SOUZA, J. D.; BECK, A. C. S. Laant: A library to automatically optimize edp for openmp applications. In: IEEE. **2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2017. p. 1229–1232.

LUK, C.-K. et al. Pin: Building customized program analysis tools with dynamic instrumentation. In: **Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2005. (PLDI '05), p. 190–200. ISBN 1-59593-056-6.

LUNGU, A. et al. Dynamic power gating with quality guarantees. In: **Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design**. New York, NY, USA: ACM, 2009. (ISLPED '09), p. 377–382. ISBN 978-1-60558-684-7.

MADAN, N. et al. A case for guarded power gating for multi-core processors. In: **Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 2011. (HPCA '11), p. 291–300. ISBN 978-1-4244-9432-3.

MAGALHÃES, G. G. et al. How programming languages and paradigms affect performance and energy in multithreaded applications. In: IEEE. **Computing Systems Engineering (SBESC), 2016 VI Brazilian Symposium on**. [S.l.], 2016. p. 71–78.

MALLÓN, D. A. et al. Performance evaluation of mpi, upc and openmp on multicore architectures. In: SPRINGER. **European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting**. [S.l.], 2009. p. 174–184.

MARQUES, W. dos S. et al. Improving edp in multi-core embedded systems through multidimensional frequency scaling. In: IEEE. **Circuits and Systems (ISCAS), 2017 IEEE International Symposium on**. [S.l.], 2017. p. 1–4.

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.

MCCOOL, M.; REINDERS, J.; ROBISON, A. **Structured Parallel Programming: Patterns for Efficient Computation**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN 9780123914439, 9780124159938.

MCVOY, L.; STAELIN, C. Lmbench: Portable tools for performance analysis. In: **Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference**. Berkeley, CA, USA: USENIX Association, 1996. (ATEC '96), p. 23–23.

NOSE, K.; SAKURAI, T. Optimization of vdd and vth for low-power and high speed applications. In: **Proceedings of the 2000 Asia and South Pacific Design Automation Conference**. New York, NY, USA: ACM, 2000. (ASP-DAC '00), p. 469–474. ISBN 0-7803-5974-7.

OBORIL, F.; TAHOORI, M. B. Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level. In: **IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)**. [S.l.: s.n.], 2012. p. 1–12. ISSN 1530-0889.

OKLOBDZIJA, V. G.; KRISHNAMURTHY, R. K. **High-Performance Energy-Efficient Microprocessor Design (Series on Integrated Circuits and Systems)**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN 0387285946.

OLUKOTUN, K.; HAMMOND, L. The future of microprocessors. **Queue**, ACM, New York, NY, USA, v. 3, n. 7, p. 26–29, sep. 2005. ISSN 1542-7730.

OPENMP, A. **OpenMP 4.0**: specification. In: . [S.l.: s.n.], 2013.

OU, Z. et al. Energy- and cost-efficiency analysis of arm-based clusters. In: **2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)**. [S.l.: s.n.], 2012. p. 115–123.

PADOIN, E. L. et al. Performance/energy trade-off in scientific computing: the case of arm big. little and intel sandy bridge. **IET Computers & Digital Techniques**, IET, v. 9, n. 1, p. 27–35, 2014.

PALERMO, G.; SILVANO, C.; ZACCARIA, V. An efficient design space exploration methodology for on-chip multiprocessors subject to application-specific constraints. In: **2008 Symposium on Application Specific Processors**. [S.l.: s.n.], 2008. p. 75–82.

PASE, D. M.; ECKL, M. A. A comparison of single-core and dual-core opteron processor performance for hpc. **IBM xSeries Performance Development and Analysis**, 2005.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design, Fifth Edition: The Hardware/Software Interface**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 0124077269, 9780124077263.

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.; FOWLER, R.; NEYER, M. Maestro: Dynamic runtime power and concurrency adaptation. In: **Workshop on Managed Many-Core Systems (MMCS)**. [S.l.: s.n.], 2008. p. 1–8.

PORTERFIELD, A. K. et al. Power measurement and concurrency throttling for energy reduction in openmp programs. In: IEEE. **Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International**. [S.l.], 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. **Workload Characterization (IISWC), 2011 IEEE International Symposium on**. [S.l.], 2011. p. 116–125.

QUINLAN, D.; LIAO, C. The rose source-to-source compiler infrastructure. In: **Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011**. [S.l.: s.n.], 2011.

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: **Int. Conf. on Parallel Architectures and Compilation Techniques**. [S.l.: s.n.], 2003. p. 15–25. ISSN 1089-795X.

RAJOVIC, N. et al. Supercomputing with commodity cpus: Are mobile socs ready for hpc? In: ACM. **Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2013. p. 40.

RAJOVIC, N. et al. The mont-blanc prototype: An alternative approach for hpc systems. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. Piscataway, NJ, USA: IEEE Press, 2016. (SC '16), p. 38:1–38:12. ISBN 978-1-4673-8815-3.

RAJOVIC, N. et al. Tibidabo: Making the case for an arm-based hpc system. **Future Generation Computer Systems**, Elsevier, v. 36, p. 322–334, 2014.

RAMAN, A. et al. Parcae: A system for flexible parallel execution. In: **ACM SIGPLAN Conf. on Programming Language Design and Implementation**. NY, USA: ACM, 2012. (PLDI '12), p. 133–144. ISBN 978-1-4503-1205-9.

RAUBER, T.; RüNGER, G. **Parallel Programming: For Multicore and Cluster Systems**. 2nd. ed. [S.l.]: Springer Publishing Company, Incorporated, 2013. ISBN 3642378005, 9783642378003.

ROSSI, F. D. et al. Modeling power consumption for dvfs policies. In: **2015 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.: s.n.], 2015. p. 1879–1882. ISSN 0271-4302.

ROTEM, E. et al. Power-management architecture of the intel microarchitecture code-named sandy bridge. **IEEE Micro**, v. 32, n. 2, p. 20–27, March 2012. ISSN 0272-1732.

S., T. C. D. et al. Comparison of parallel programming models for multicore architectures. In: **IEEE Int. Symp. on Parallel and Distributed Processing Workshops and Phd Forum**. [S.l.: s.n.], 2011. p. 1675–1682. ISSN 1530-2075.

SALEHIAN, S.; LIU, J.; YAN, Y. Comparison of threading programming models. In: **2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [S.l.: s.n.], 2017. p. 766–774.

SARTOR, A. L.; LORENZON, A. F.; BECK, A. C. The impact of virtual machines on embedded systems. In: IEEE. **Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual**. [S.l.], 2015. v. 2, p. 626–631.

SARTOR, A. L. et al. A novel phase-based low overhead fault tolerance approach for vliw processors. In: IEEE. **VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on**. [S.l.], 2015. p. 485–490.

SARTOR, A. L. et al. Exploiting idle hardware to provide low overhead fault tolerance for vliw processors. **ACM Journal on Emerging Technologies in Computing Systems (JETC)**, ACM, v. 13, n. 2, p. 13, 2017.

SCHWARZROCK, J. et al. Potential gains in edp by dynamically adapting the number of threads for openmp applications in embedded systems. In: **2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.: s.n.], 2017. p. 79–85.

SENSI, D. D. Predicting performance and power consumption of parallel applications. In: **2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)**. [S.l.: s.n.], 2016. p. 200–207.

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: ACM. **Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures**. [S.l.], 2015. p. 19–24.

SHAFIK, R. A. et al. **Thermal-aware adaptive energy minimization of openMP parallel applications**. In: . [S.l.: s.n.], 2015.

SHARKAWI, S. et al. Performance projection of hpc applications using spec cfp2006 benchmarks. In: **2009 IEEE International Symposium on Parallel Distributed Processing**. [S.l.: s.n.], 2009. p. 1–12. ISSN 1530-2075.

SINGH, K. et al. Predicting parallel application performance via machine learning approaches: Research articles. **Concurr. Comput. : Pract. Exper.**, John Wiley and Sons Ltd., Chichester, UK, v. 19, n. 17, p. 2219–2235, dec. 2007. ISSN 1532-0626.

SODHI, S.; SUBHLOK, J.; XU, Q. Performance prediction with skeletons. **Cluster Computing**, Kluwer Academic Publishers, Hingham, MA, USA, v. 11, n. 2, p. 151–165, jun. 2008. ISSN 1386-7857.

SONG, S. L.; BARKER, K.; KERBYSON, D. Unified performance and power modeling of scientific workloads. In: **Proceedings of the 1st International Workshop on Energy Efficient Supercomputing**. New York, NY, USA: ACM, 2013. (E2SC '13), p. 4:1–4:8. ISBN 978-1-4503-2504-2.

SRIDHARAN, S.; GUPTA, G.; SOHI, G. S. Holistic run-time parallelism management for time and energy efficiency. In: ACM. **Proceedings of the 27th international ACM conference on International conference on supercomputing**. [S.l.], 2013. p. 337–348.

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

STANISIC, L. et al. Performance analysis of hpc applications on low-power embedded platforms. In: EDA CONSORTIUM. **Proceedings of the conference on design, automation and test in Europe**. [S.l.], 2013. p. 475–480.

STRATTON, J. et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. **Center for Reliable and High-Performance Computing**, 2012.

SUBRAMANIAN, L. et al. Mise: Providing performance predictability and improving fairness in shared main memory systems. In: **IEEE Int. Symp. on High Performance Computer Architecture**. [S.l.: s.n.], 2013. p. 639–650. ISSN 1530-0897.

SUEUR, E. L.; HEISER, G. Dynamic voltage and frequency scaling: The laws of diminishing returns. In: **Proceedings of the 2010 International Conference on Power Aware Computing and Systems**. Berkeley, CA, USA: USENIX Association, 2010. (HotPower'10), p. 1–8.

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, New York, NY, USA, v. 36, n. 1, p. 277–286, mar. 2008. ISSN 0163-5964.

TANENBAUM, A. S. **Modern Operating Systems**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

TAYLOR, V. et al. Using kernel couplings to predict parallel application performance. In: **Proceedings 11th IEEE International Symposium on High Performance Distributed Computing**. [S.l.: s.n.], 2002. p. 125–134. ISSN 1082-8907.

TAYLOR, V. et al. Prophesy: automating the modeling process. In: **Proceedings Third Annual International Workshop on Active Middleware Services**. [S.l.: s.n.], 2001. p. 3–11.

TIWARI, A. et al. Performance and energy efficiency analysis of 64-bit arm using gamess. In: **Proceedings of the 2Nd International Workshop on Hardware-Software Co-Design for High Performance Computing**. New York, NY, USA: ACM, 2015. (Co-HPC '15), p. 8:1–8:10. ISBN 978-1-4503-3992-6.

TIWARI, A. et al. Modeling power and energy usage of hpc kernels. In: **2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum**. [S.l.: s.n.], 2012. p. 990–998.

VENKATESH, A.; KANDALLA, K.; PANDA, D. K. Evaluation of energy characteristics of mpi communication primitives with rapl. In: **2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum**. [S.l.: s.n.], 2013. p. 938–945.

VOGELSANG, T. Understanding the energy consumption of dynamic random access memories. In: **Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 2010. (MICRO '43), p. 363–374. ISBN 978-0-7695-4299-7.

WAHLÉN, N. **A comparison of different parallel programming models for multicore processors**. 2010.

WALL, D. W. Limits of instruction-level parallelism. In: **Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: ACM, 1991. (ASPLOS IV), p. 176–188. ISBN 0-89791-380-9.

WANG, B.; SCHMIDL, D.; MÜLLER, M. S. Evaluating the energy consumption of openmp applications on haswell processors. In: TERBOVEN, C. et al. (Ed.). **OpenMP: Heterogenous Execution and Data Movements**. Cham: Springer International Publishing, 2015. p. 233–246. ISBN 978-3-319-24595-9.

WHEELER, K. B.; MURPHY, R. C.; THAIN, D. Qthreads: An api for programming with millions of lightweight threads. In: **IEEE Int. Symp. on Parallel and Distributed Processing**. [S.l.: s.n.], 2008. p. 1–8. ISSN 1530-2075.

WILLHALM, T.; DEMENTIEV, R.; FAY, P. **Intel Performance Counter Monitor**: A better way to measure cpu utilization. In: . [S.l.: s.n.], 2017.

WITKOWSKI, M. et al. Practical power consumption estimation for real life hpc applications. **Future Gener. Comput. Syst.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 29, n. 1, p. 208–217, jan. 2013. ISSN 0167-739X.

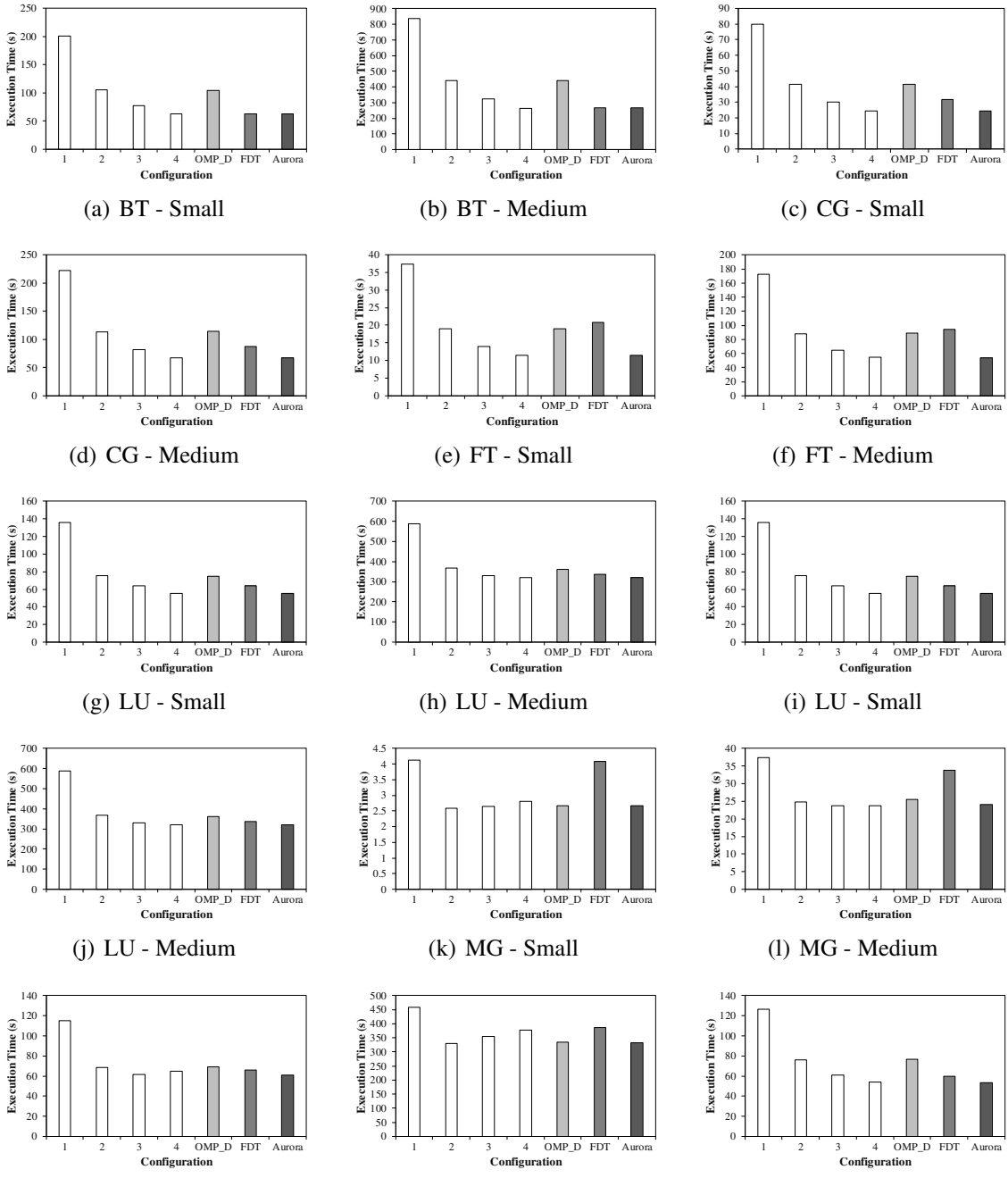
YANG, L. T.; MA, X.; MUELLER, F. Cross-platform performance prediction of parallel applications using partial execution. In: **Proceedings of the 2005 ACM/IEEE Conference on Supercomputing**. Washington, DC, USA: IEEE Computer Society, 2005. (SC '05), p. 40–. ISBN 1-59593-061-2.

ZECENA, I. et al. Energy consumption analysis of parallel sorting algorithms running on multicore systems. In: IEEE. **Green Computing Conference (IGCC), 2012 International**. [S.l.], 2012. p. 1–6.

ZHANG, W.; CHENG, A. M. K.; SUBHLOK, J. Dwarfcode: A performance prediction tool for parallel applications. **IEEE Transactions on Computers**, v. 65, n. 2, p. 495–507, Feb 2016. ISSN 0018-9340.

APPENDIX A — RESULTS OF AURORA EXECUTION

Figure A.1: Performance - 4-Core System



(a) BT - Small (b) BT - Medium (c) CG - Small

(d) CG - Medium (e) FT - Small (f) FT - Medium

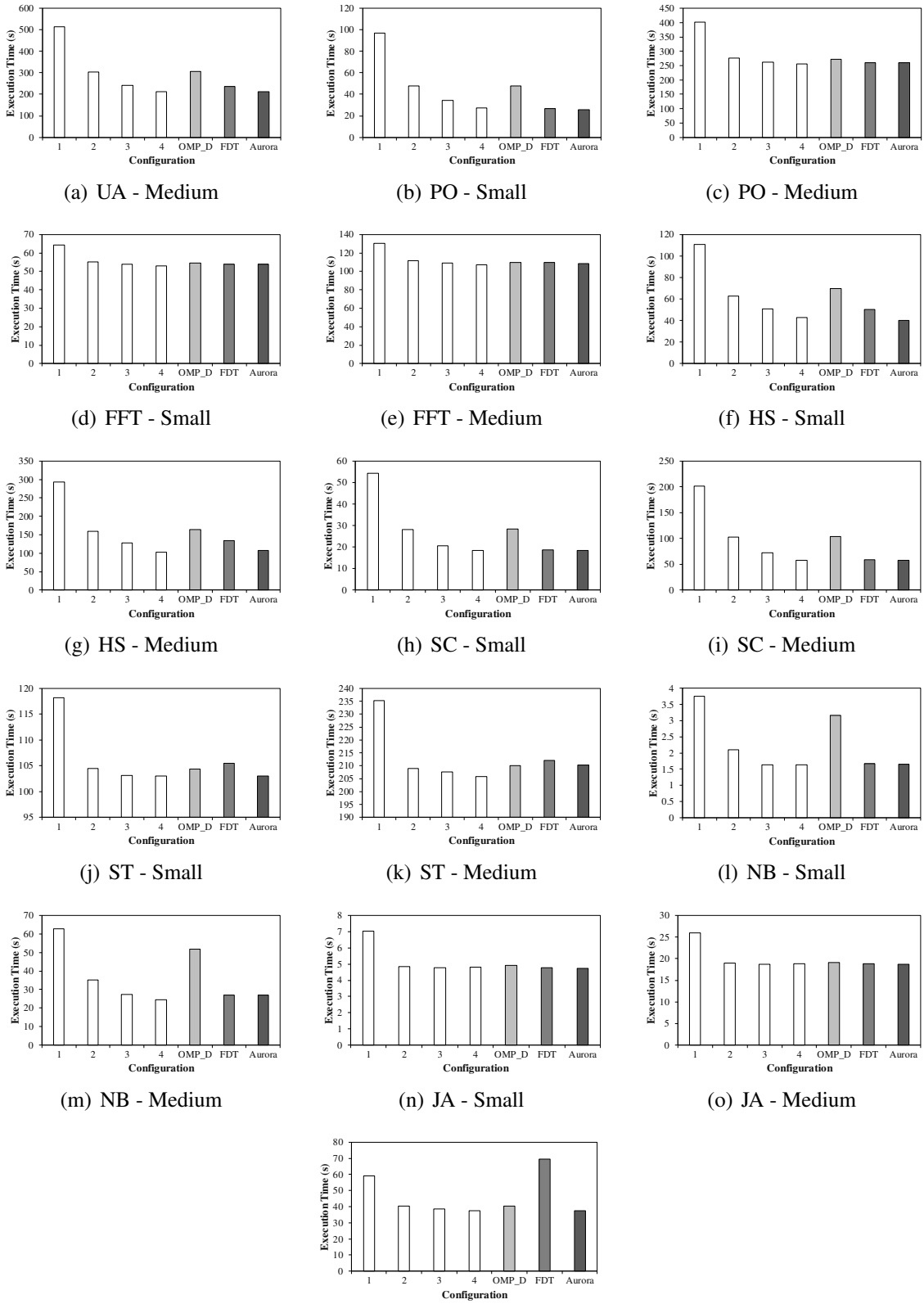
(g) LU - Small (h) LU - Medium (i) LU - Small

(j) LU - Medium (k) MG - Small (l) MG - Medium

(m) SP - Small (n) SP - Medium (o) UA - Small

Source: The Author

Figure A.2: Performance - 4-Core System (Continuation)



(a) UA - Medium (b) PO - Small (c) PO - Medium

(d) FFT - Small (e) FFT - Medium (f) HS - Small

(g) HS - Medium (h) SC - Small (i) SC - Medium

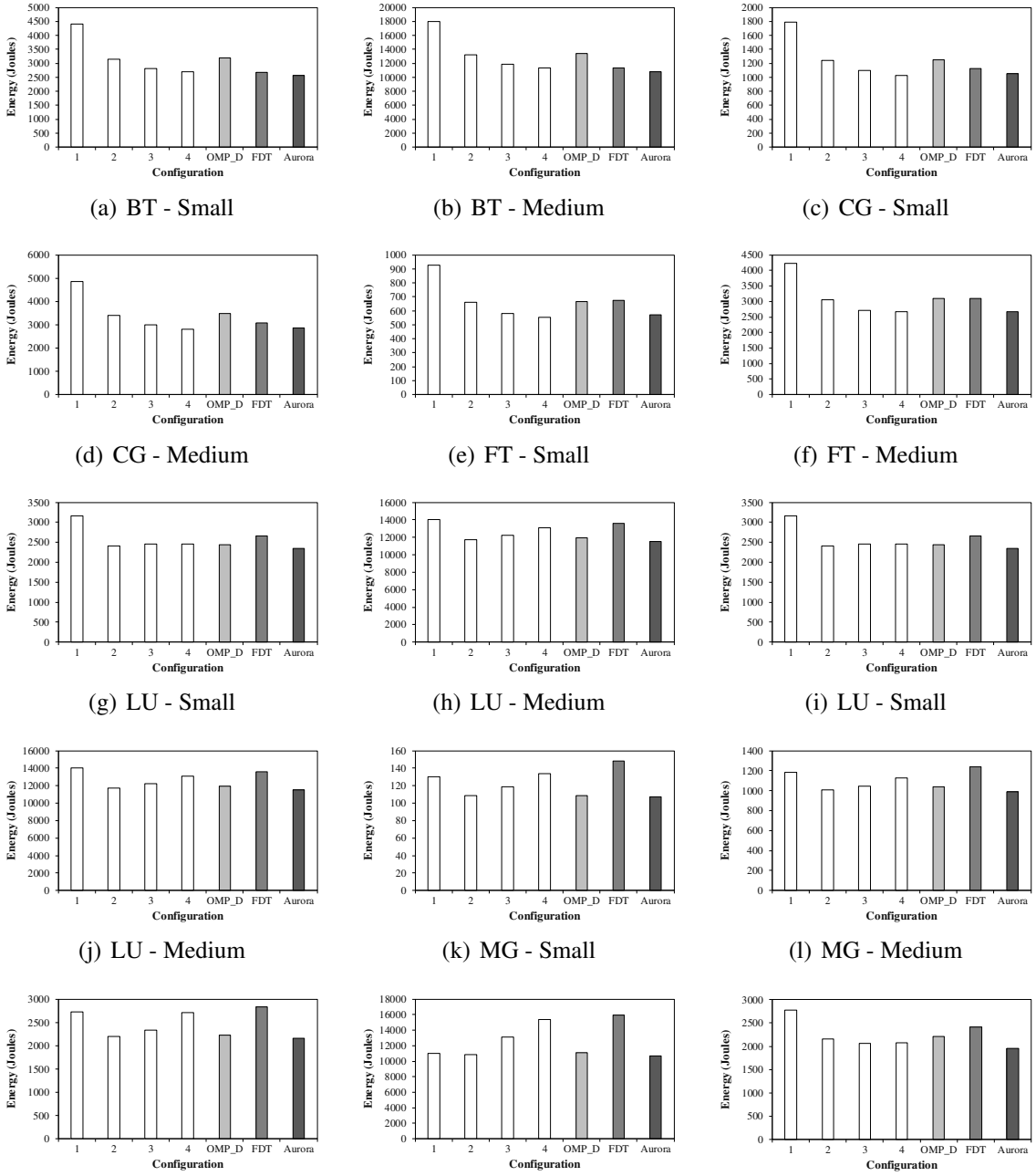
(j) ST - Small (k) ST - Medium (l) NB - Small

(m) NB - Medium (n) JA - Small (o) JA - Medium

(p) HPCG - Small

Source: The Author

Figure A.3: Energy Consumption - 4-Core System



(a) BT - Small (b) BT - Medium (c) CG - Small

(d) CG - Medium (e) FT - Small (f) FT - Medium

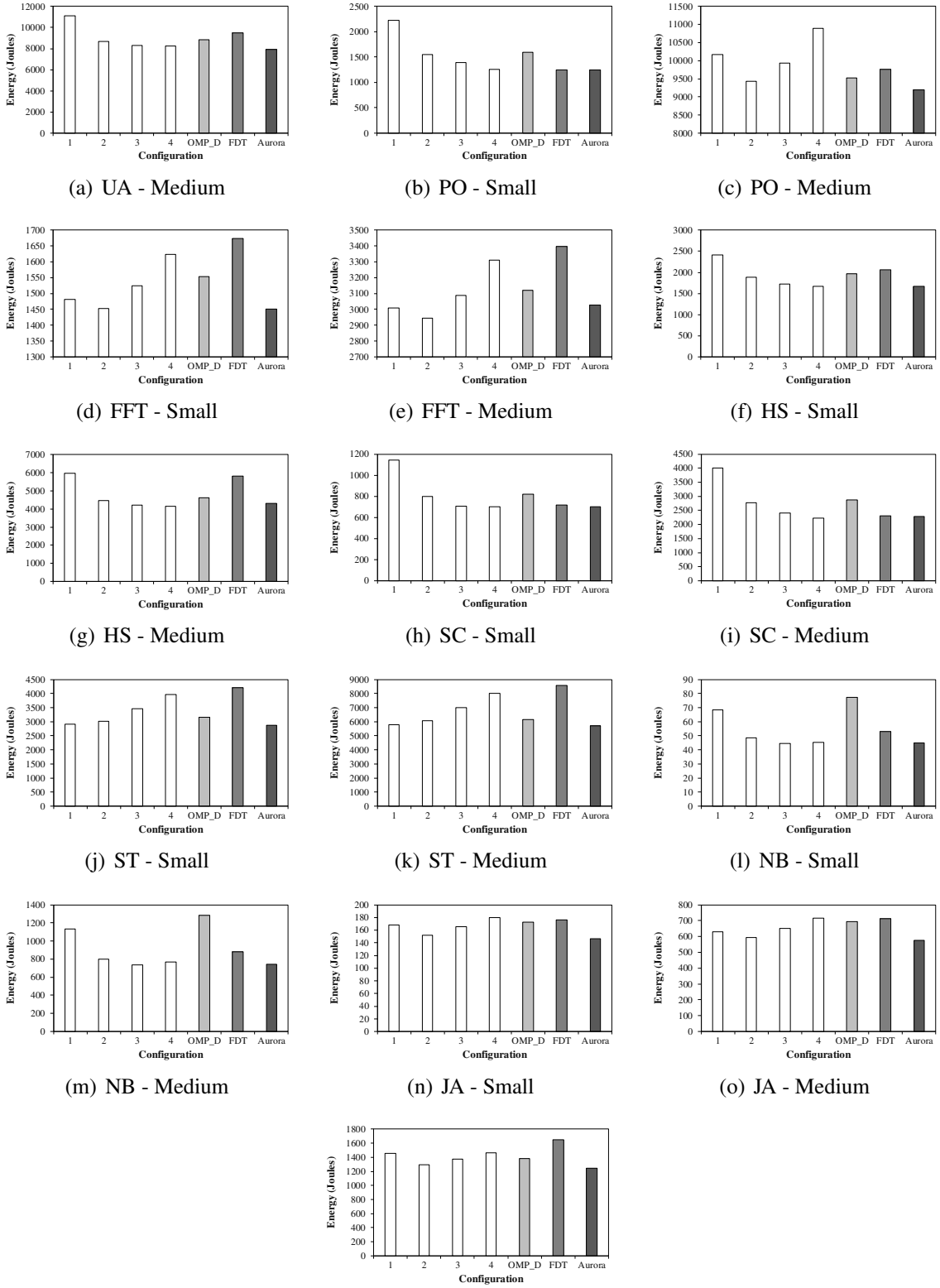
(g) LU - Small (h) LU - Medium (i) LU - Small

(j) LU - Medium (k) MG - Small (l) MG - Medium

(m) SP - Small (n) SP - Medium (o) UA - Small

Source: The Author

Figure A.4: Energy Consumption - 4-Core System (Continuation)



(a) UA - Medium (b) PO - Small (c) PO - Medium

(d) FFT - Small (e) FFT - Medium (f) HS - Small

(g) HS - Medium (h) SC - Small (i) SC - Medium

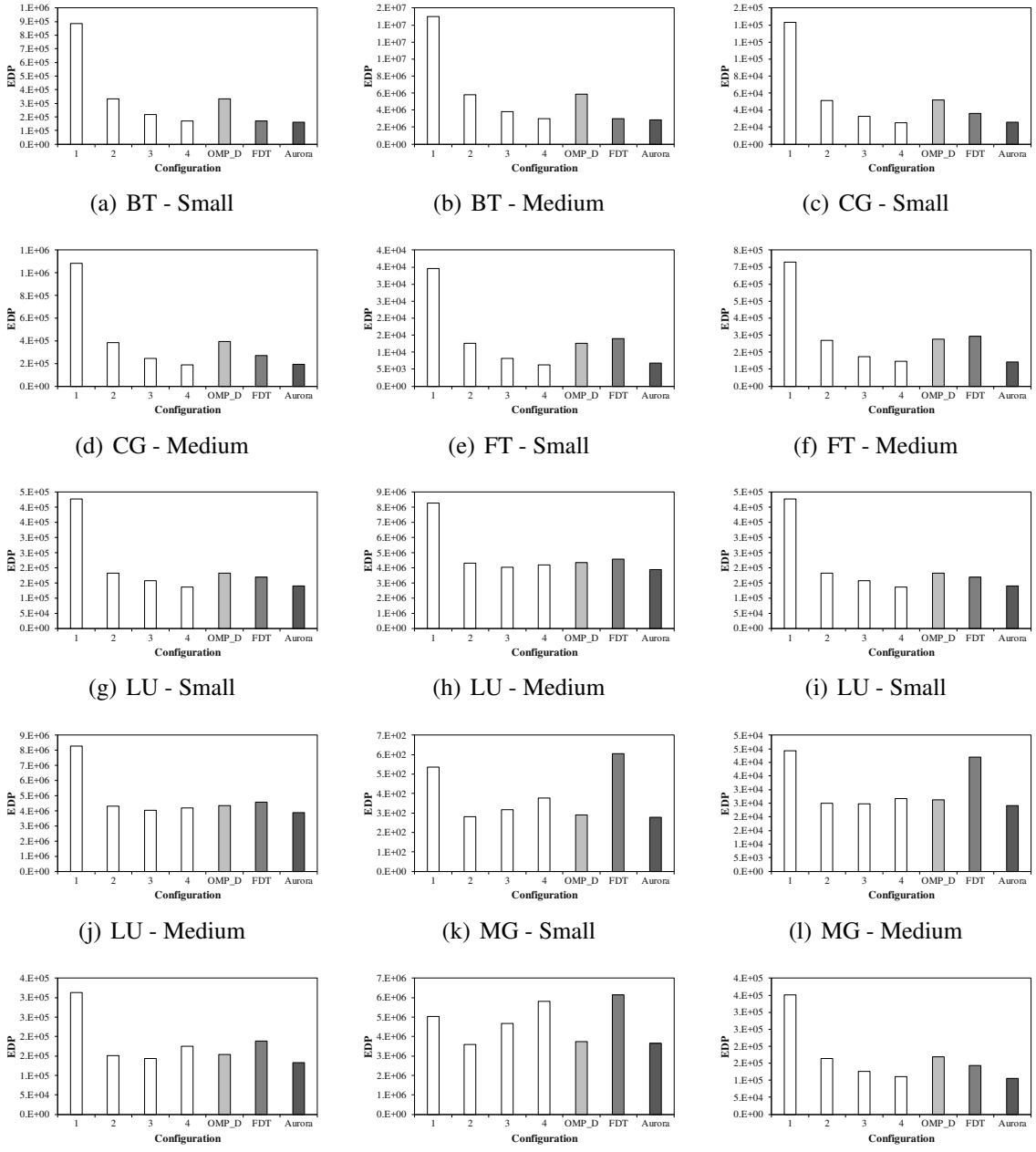
(j) ST - Small (k) ST - Medium (l) NB - Small

(m) NB - Medium (n) JA - Small (o) JA - Medium

(p) HPCG - Small

Source: The Author

Figure A.5: EDP - 4-Core System



(a) BT - Small (b) BT - Medium (c) CG - Small

(d) CG - Medium (e) FT - Small (f) FT - Medium

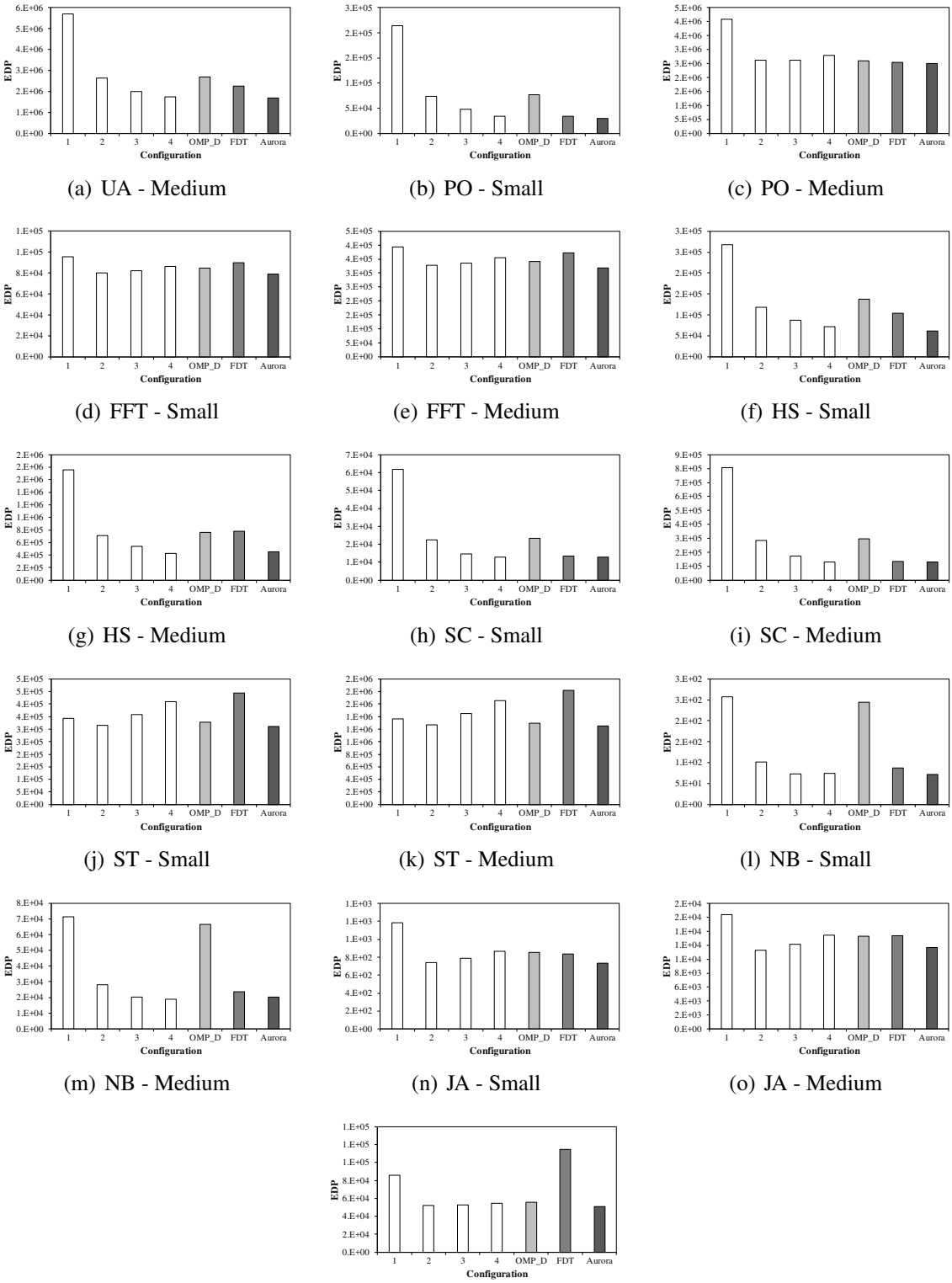
(g) LU - Small (h) LU - Medium (i) LU - Small

(j) LU - Medium (k) MG - Small (l) MG - Medium

(m) SP - Small (n) SP - Medium (o) UA - Small

Source: The Author

Figure A.6: EDP - 4-Core System (Continuation)



(a) UA - Medium (b) PO - Small (c) PO - Medium

(d) FFT - Small (e) FFT - Medium (f) HS - Small

(g) HS - Medium (h) SC - Small (i) SC - Medium

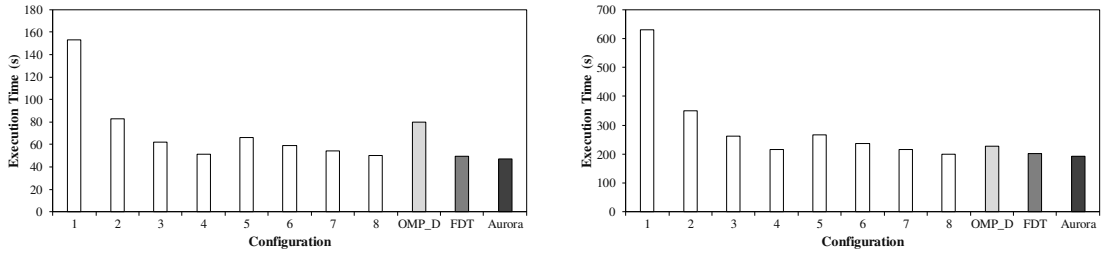
(j) ST - Small (k) ST - Medium (l) NB - Small

(m) NB - Medium (n) JA - Small (o) JA - Medium

(p) HPCG - Small

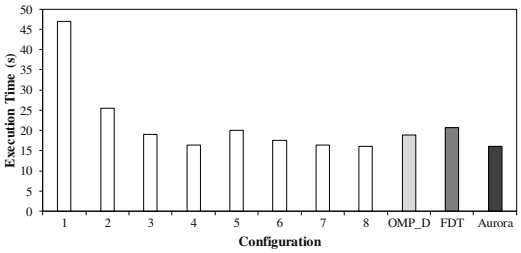
Source: The Author

Figure A.7: Performance - 8-Core System

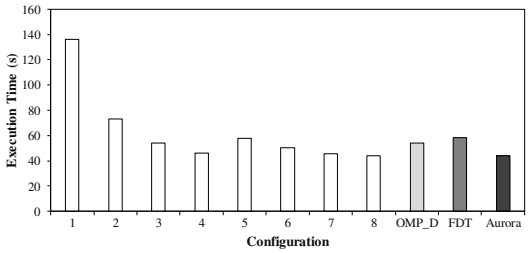


(a) BT - Small

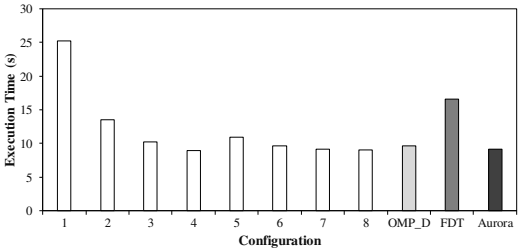
(b) BT - Medium



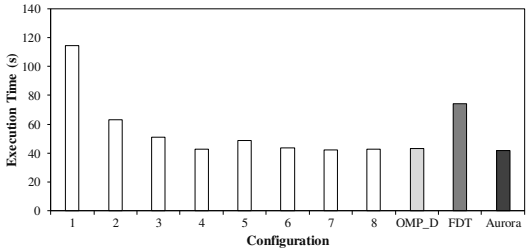
(c) CG - Small



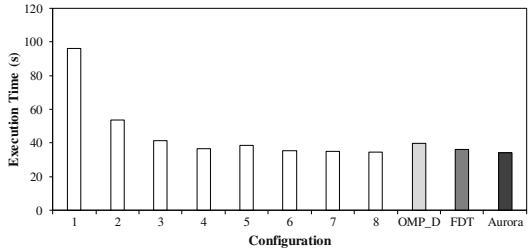
(d) CG - Medium



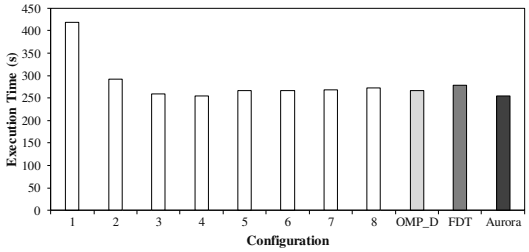
(e) FT - Small



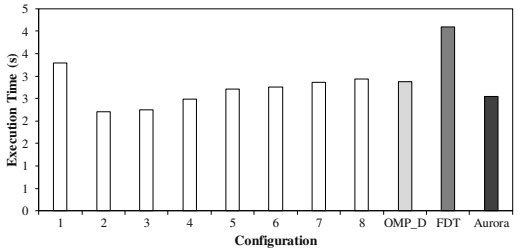
(f) FT - Medium



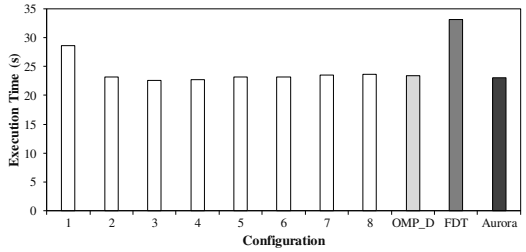
(g) LU - Small



(h) LU - Medium



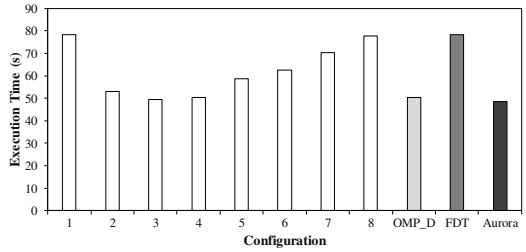
(i) MG - Small



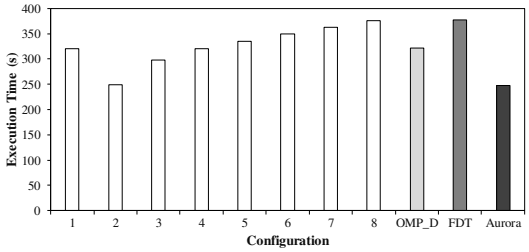
(j) MG - Medium

Source: The Author

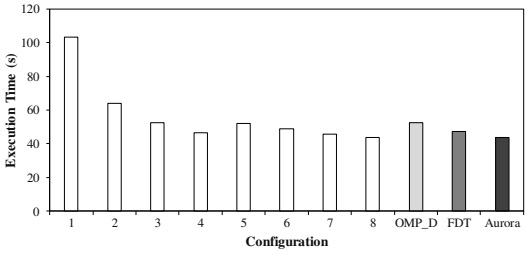
Figure A.8: Performance - 8-Core System (Continuation)



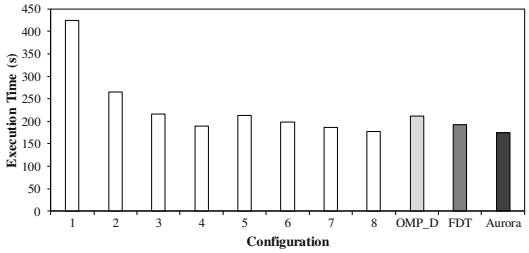
(a) SP - Small



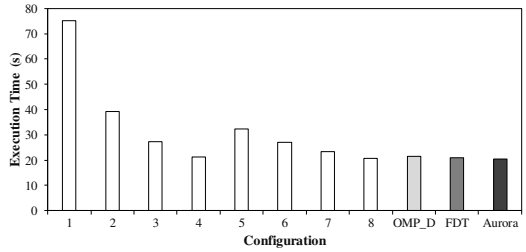
(b) SP - Medium



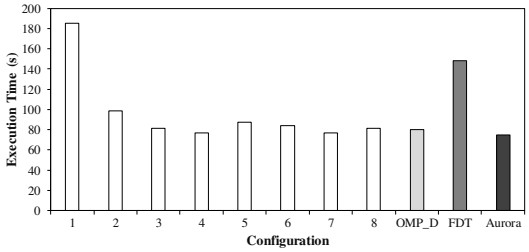
(c) UA - Small



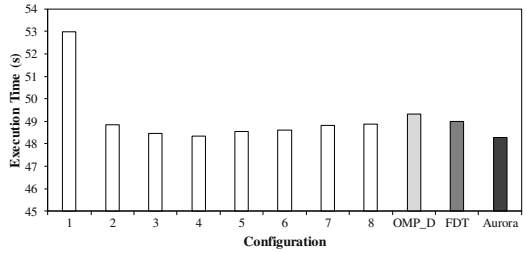
(d) UA - Medium



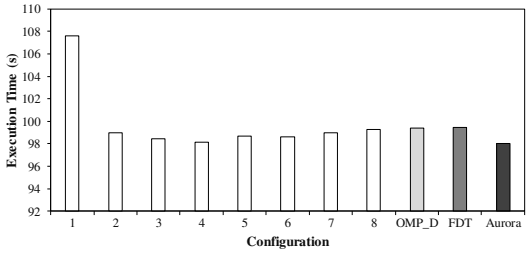
(e) PO - Small



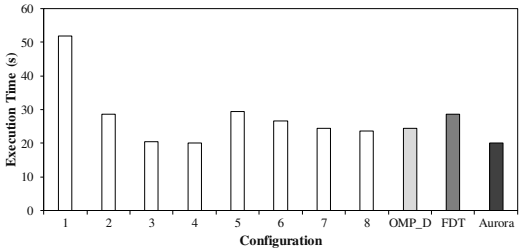
(f) PO - Medium



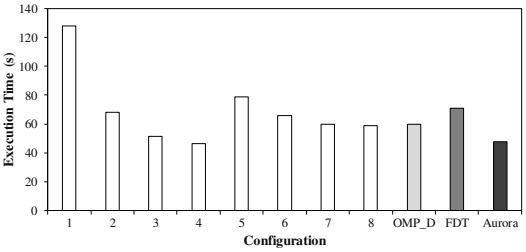
(g) FFT - Small



(h) FFT - Medium



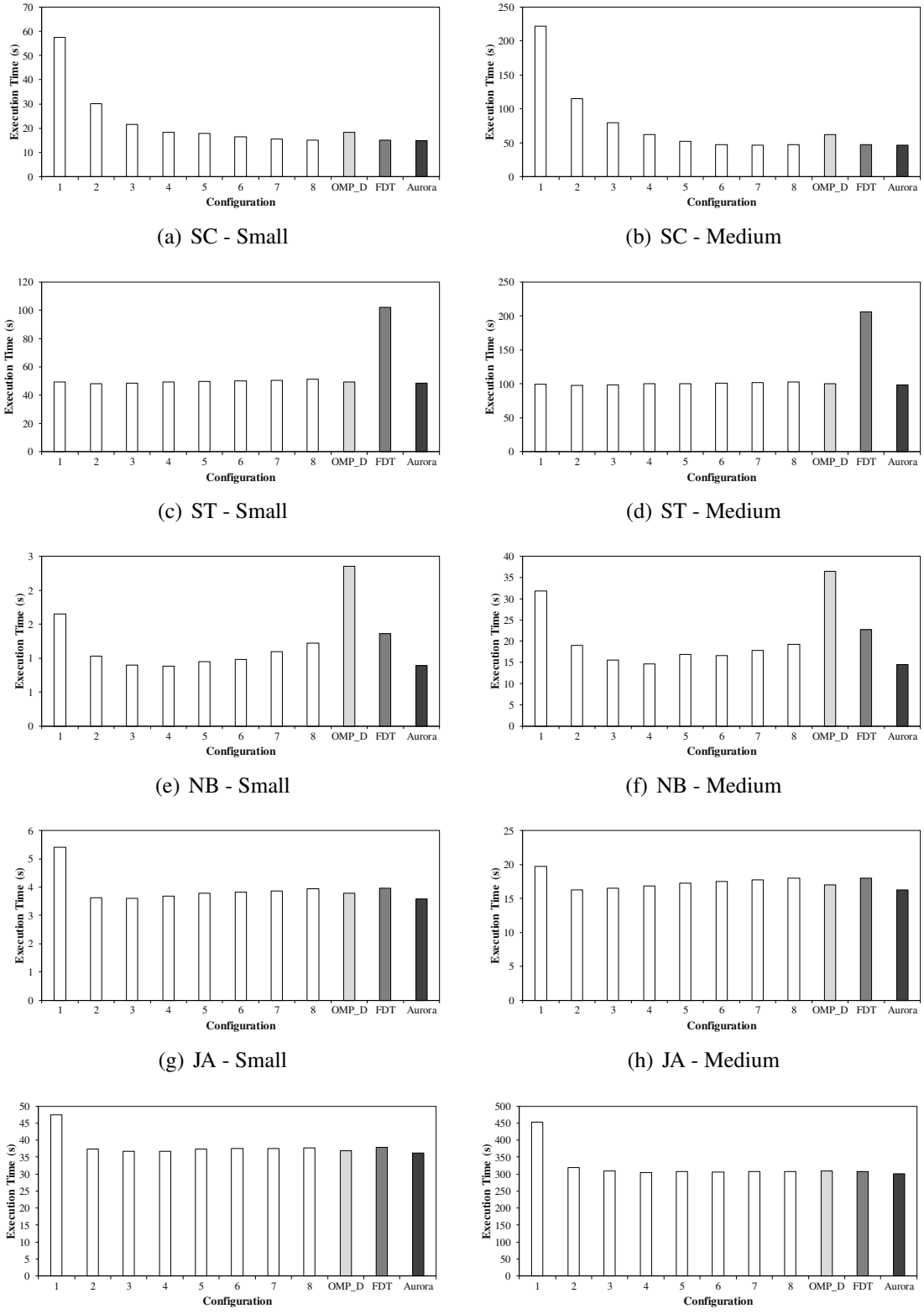
(i) HS - Small



(j) HS - Medium

Source: The Author

Figure A.9: Performance - 8-Core System (Continuation)



(a) SC - Small

(b) SC - Medium

(c) ST - Small

(d) ST - Medium

(e) NB - Small

(f) NB - Medium

(g) JA - Small

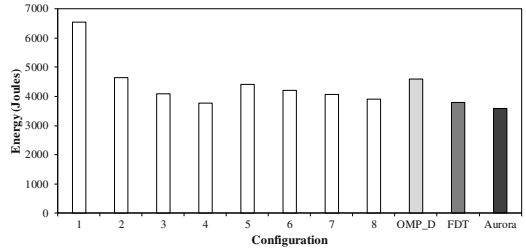
(h) JA - Medium

(i) HPCG - Small

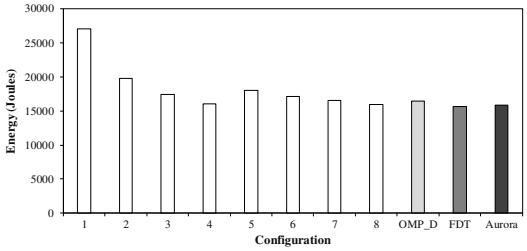
(j) HPCG - Medium

Source: The Author

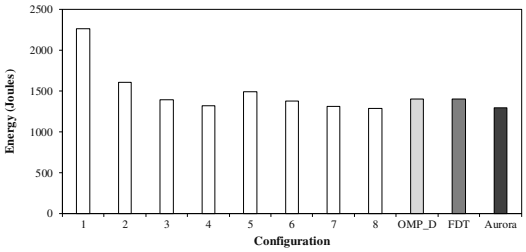
Figure A.10: Energy Consumption - 8-Core System



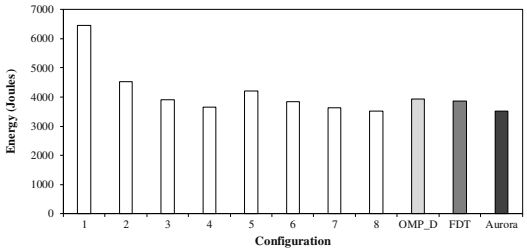
(a) BT - Small



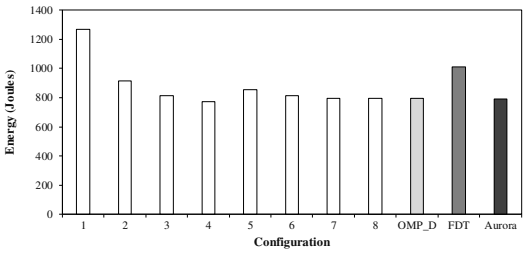
(b) BT - Medium



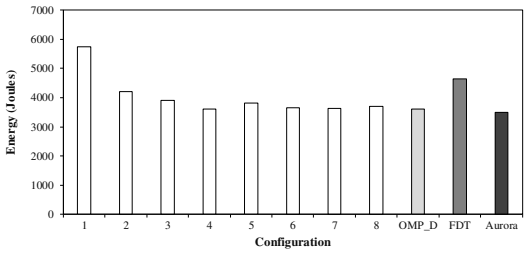
(c) CG - Small



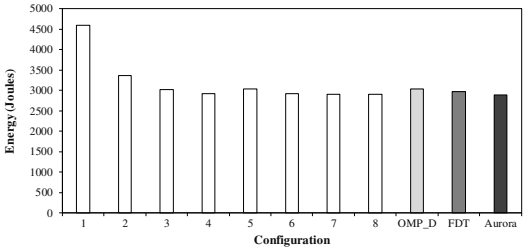
(d) CG - Medium



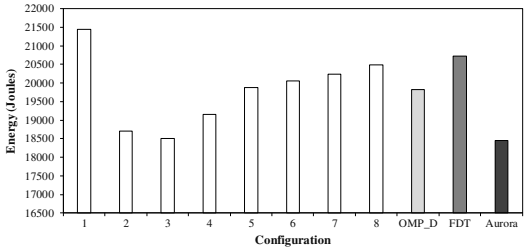
(e) FT - Small



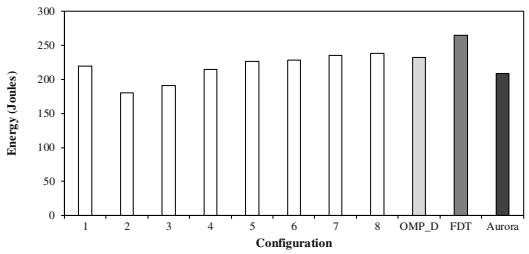
(f) FT - Medium



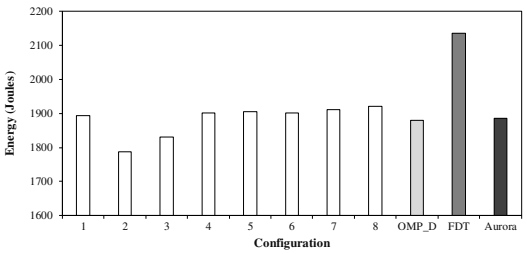
(g) LU - Small



(h) LU - Medium



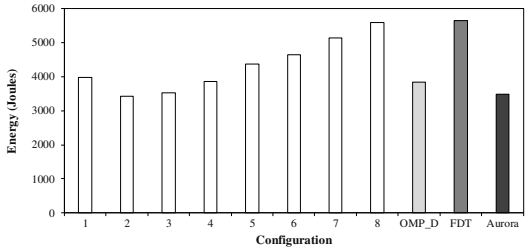
(i) MG - Small



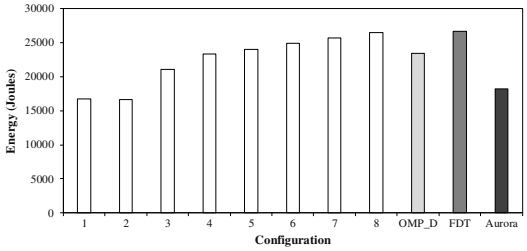
(j) MG - Medium

Source: The Author

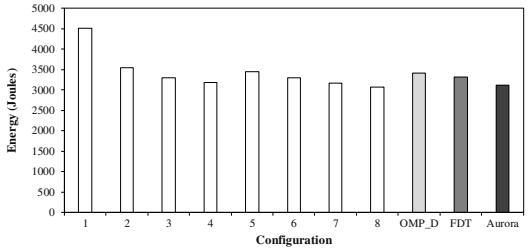
Figure A.11: Energy Consumption - 8-Core System (Continuation)



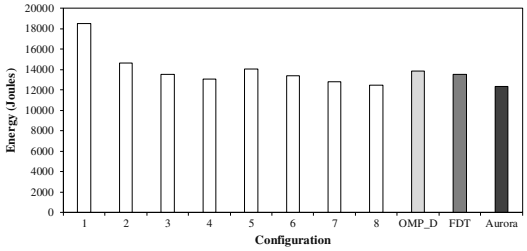
(a) SP - Small



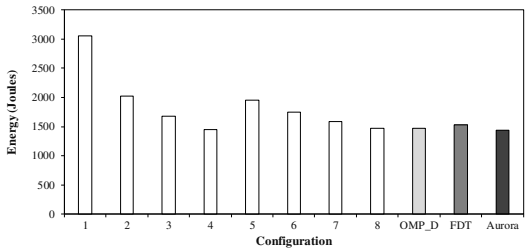
(b) SP - Medium



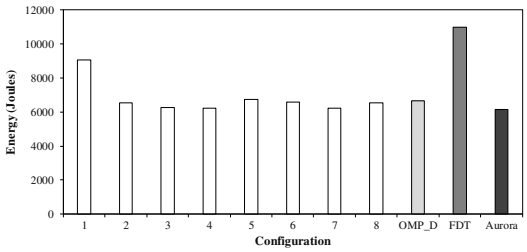
(c) UA - Small



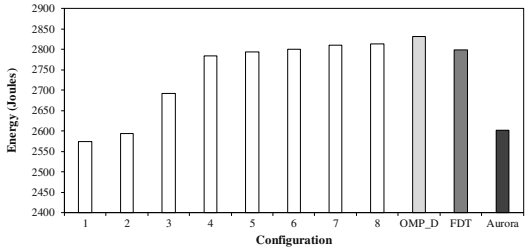
(d) UA - Medium



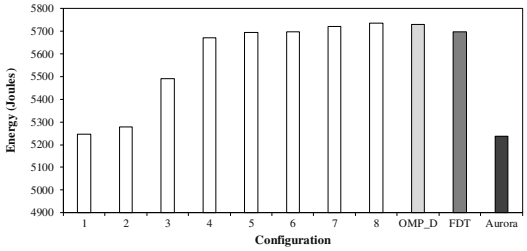
(e) PO - Small



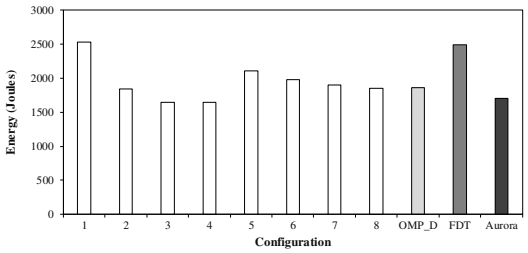
(f) PO - Medium



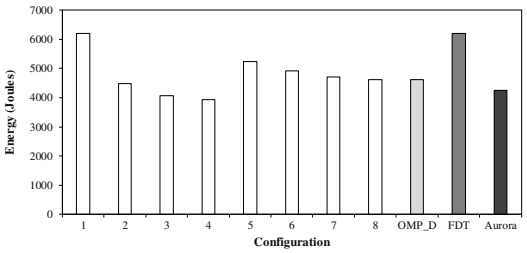
(g) FFT - Small



(h) FFT - Medium



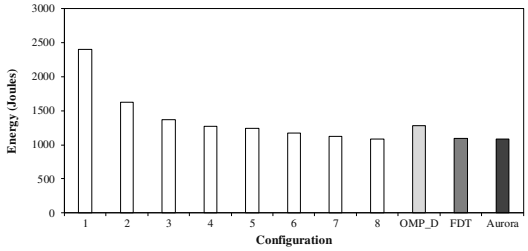
(i) HS - Small



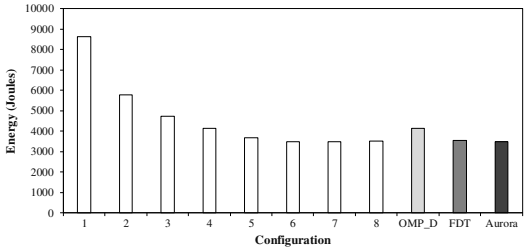
(j) HS - Medium

Source: The Author

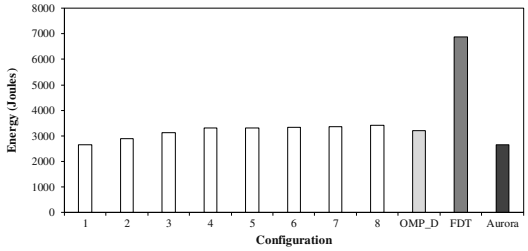
Figure A.12: Energy Consumption - 8-Core System (Continuation)



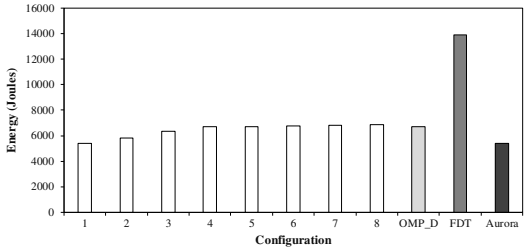
(a) SC - Small



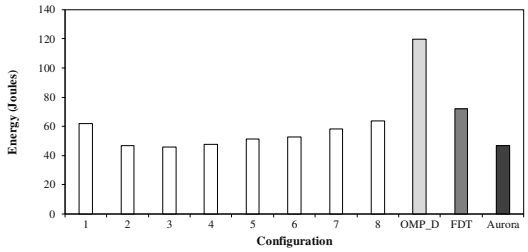
(b) SC - Medium



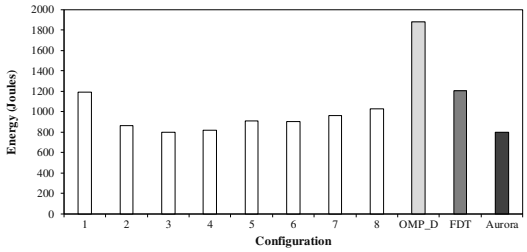
(c) ST - Small



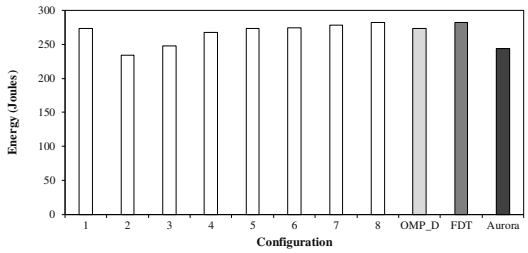
(d) ST - Medium



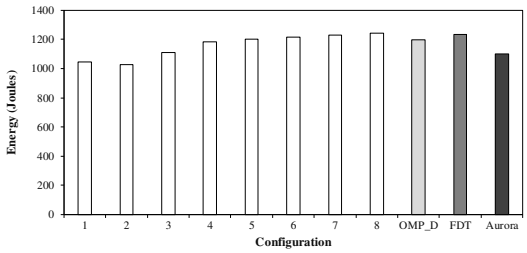
(e) NB - Small



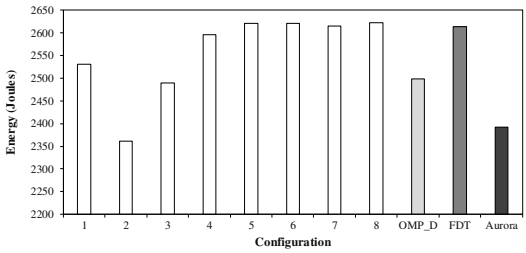
(f) NB - Medium



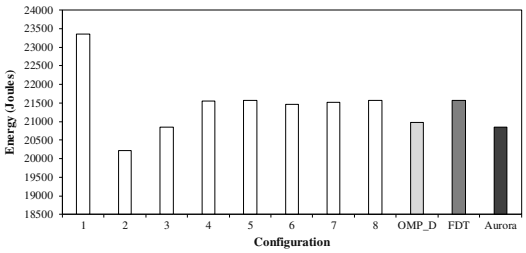
(g) JA - Small



(h) JA - Medium



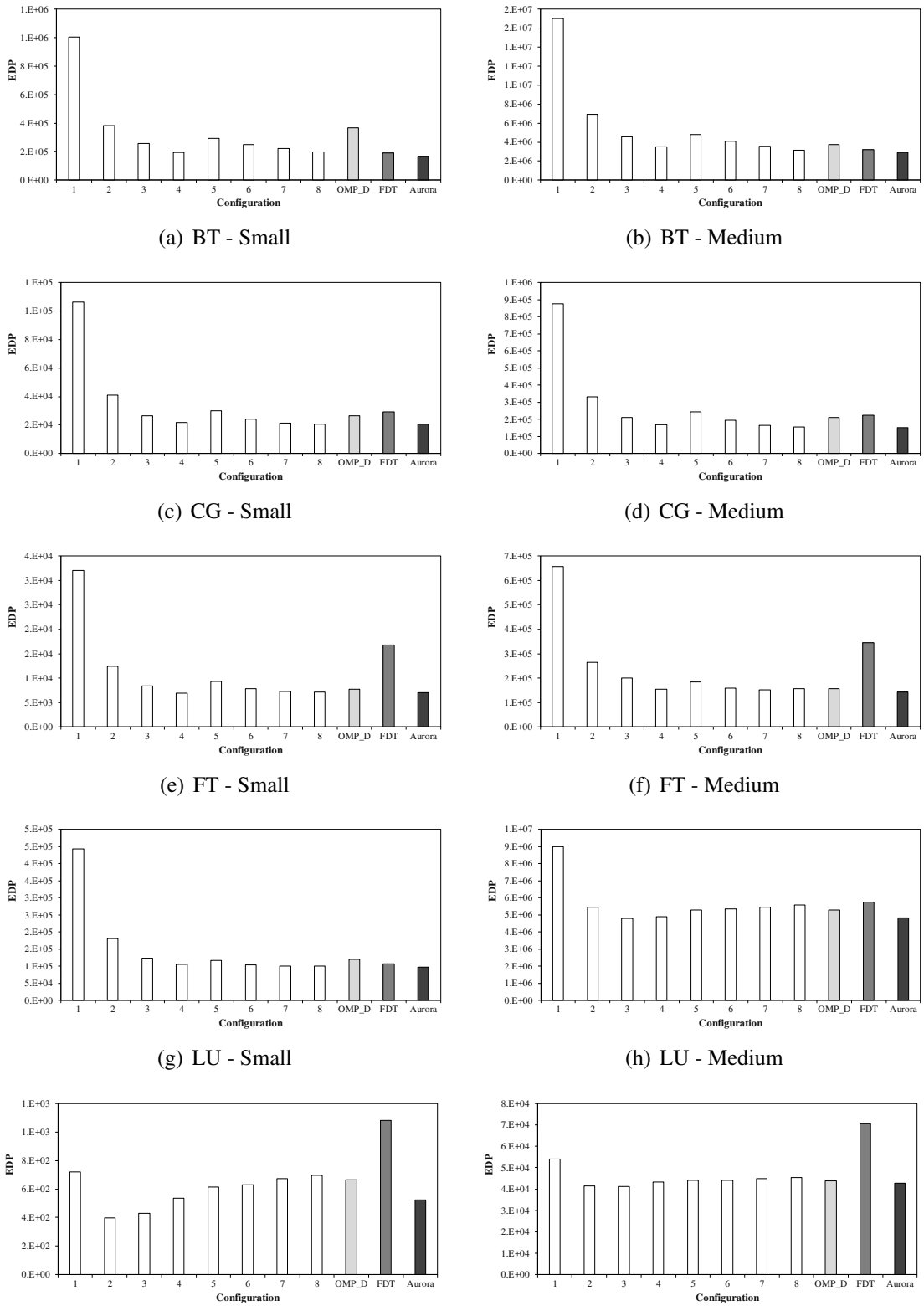
(i) HPCG - Small



(j) HPCG - Medium

Source: The Author

Figure A.13: EDP - 8-Core System



(a) BT - Small

(b) BT - Medium

(c) CG - Small

(d) CG - Medium

(e) FT - Small

(f) FT - Medium

(g) LU - Small

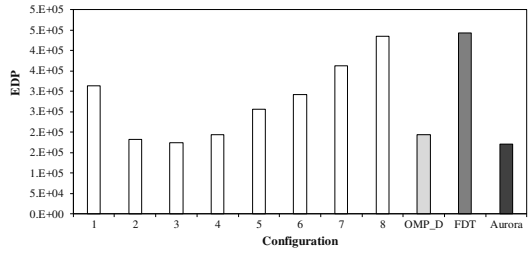
(h) LU - Medium

(i) MG - Small

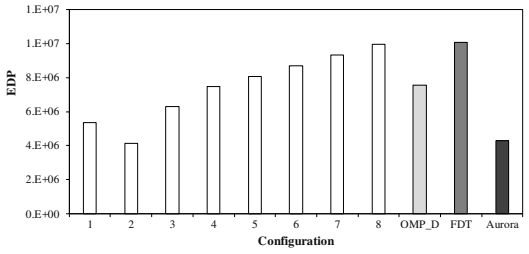
(j) MG - Medium

Source: The Author

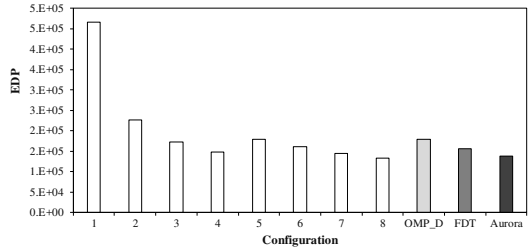
Figure A.14: EDP - 8-Core System (Continuation)



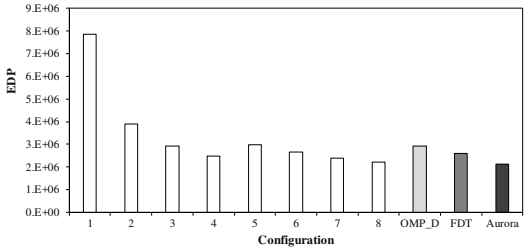
(a) SP - Small



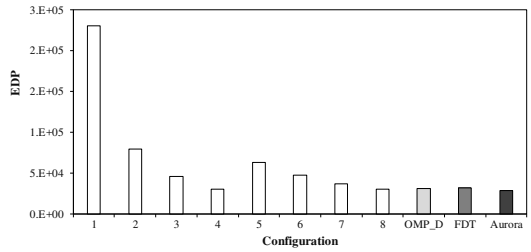
(b) SP - Medium



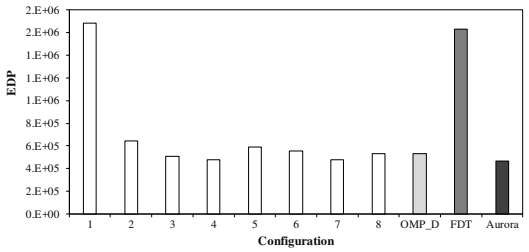
(c) UA - Small



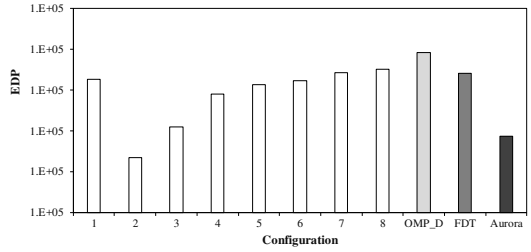
(d) UA - Medium



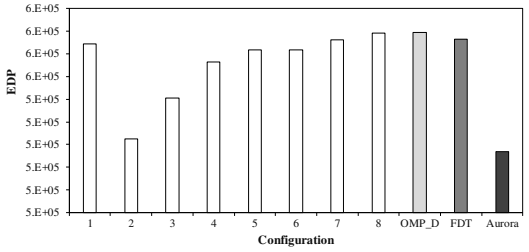
(e) PO - Small



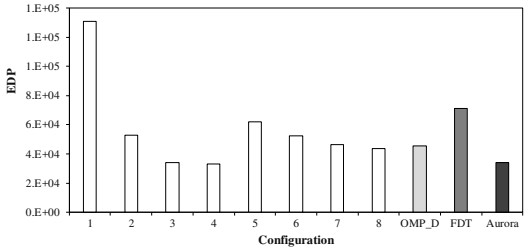
(f) PO - Medium



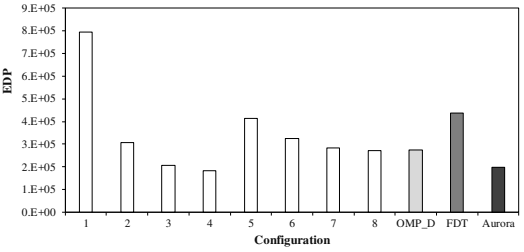
(g) FFT - Small



(h) FFT - Medium



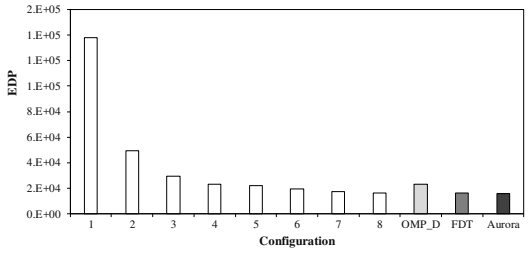
(i) HS - Small



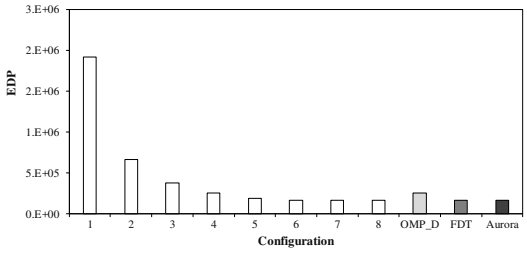
(j) HS - Medium

Source: The Author

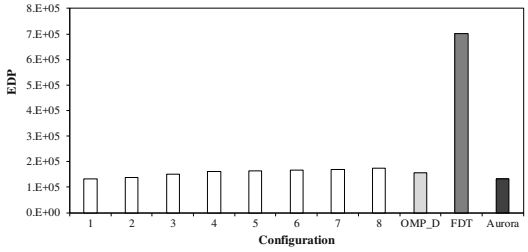
Figure A.15: EDP - 8-Core System (Continuation)



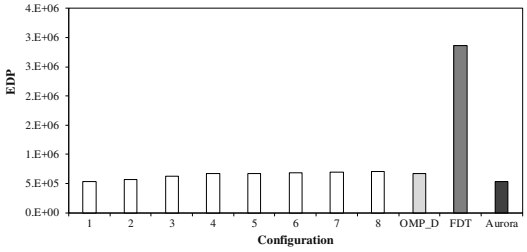
(a) SC - Small



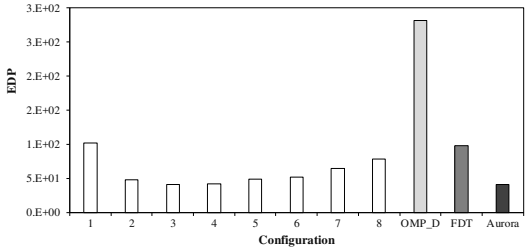
(b) SC - Medium



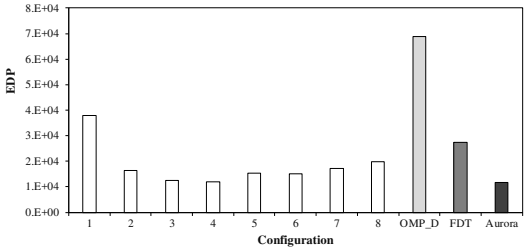
(c) ST - Small



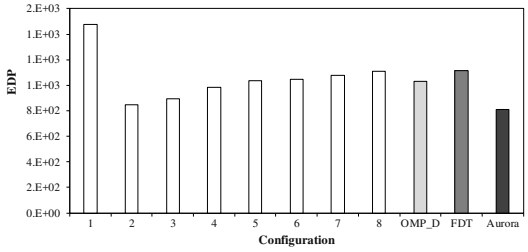
(d) ST - Medium



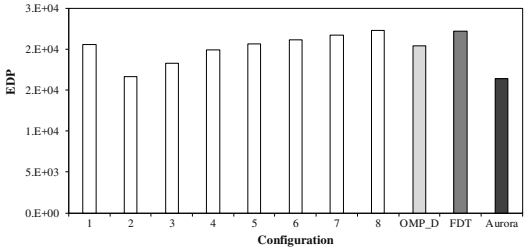
(e) NB - Small



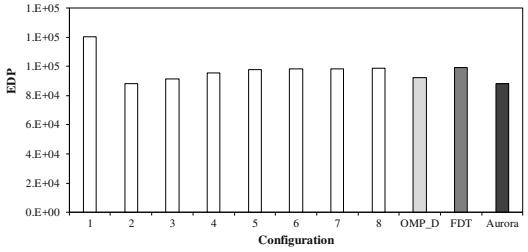
(f) NB - Medium



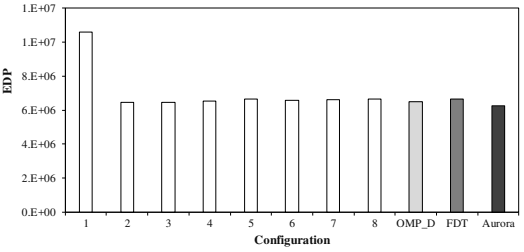
(g) JA - Small



(h) JA - Medium



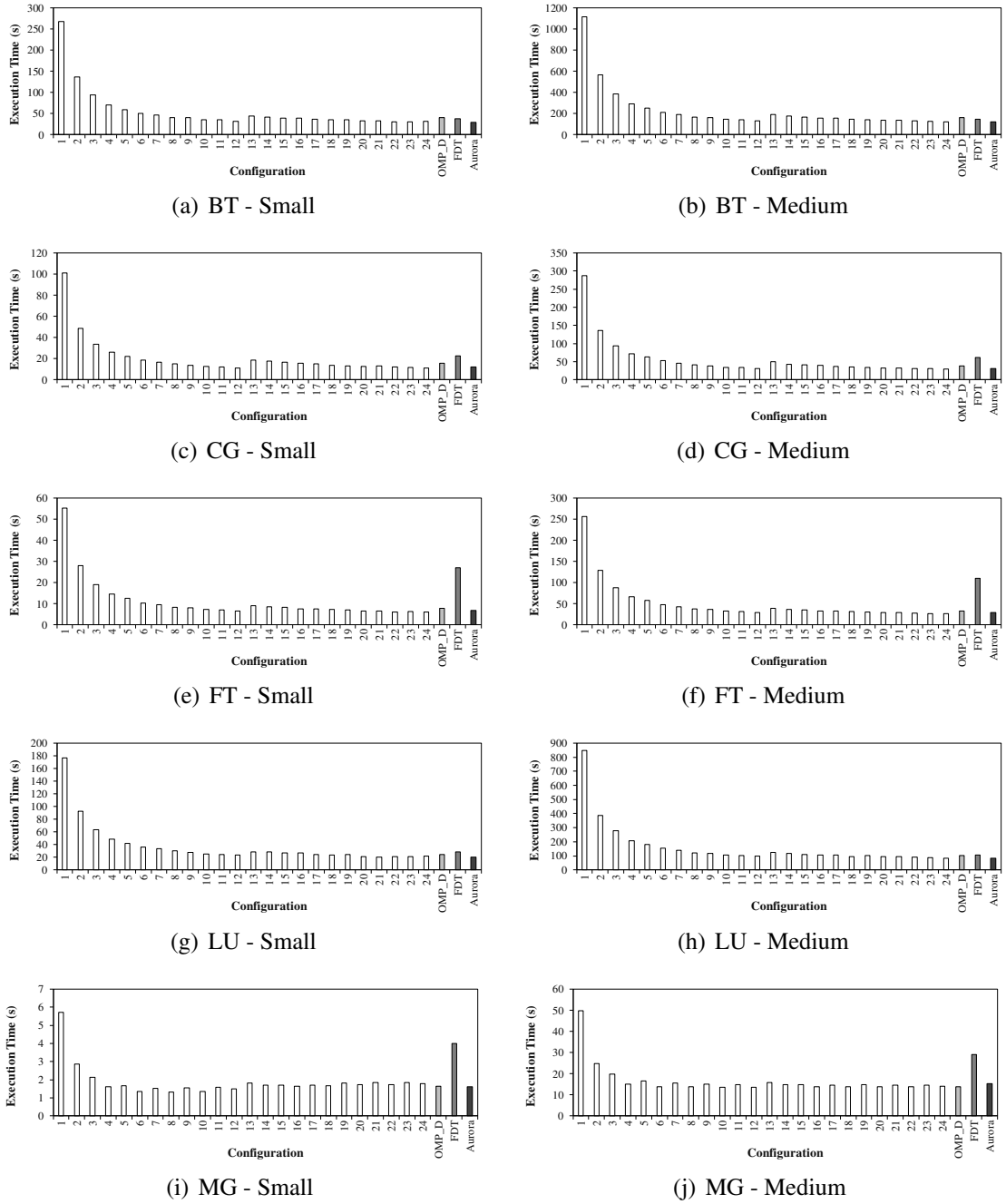
(i) HPCG - Small



(j) HPCG - Medium

Source: The Author

Figure A.16: Performance - 24-Core System



(a) BT - Small (b) BT - Medium

(c) CG - Small (d) CG - Medium

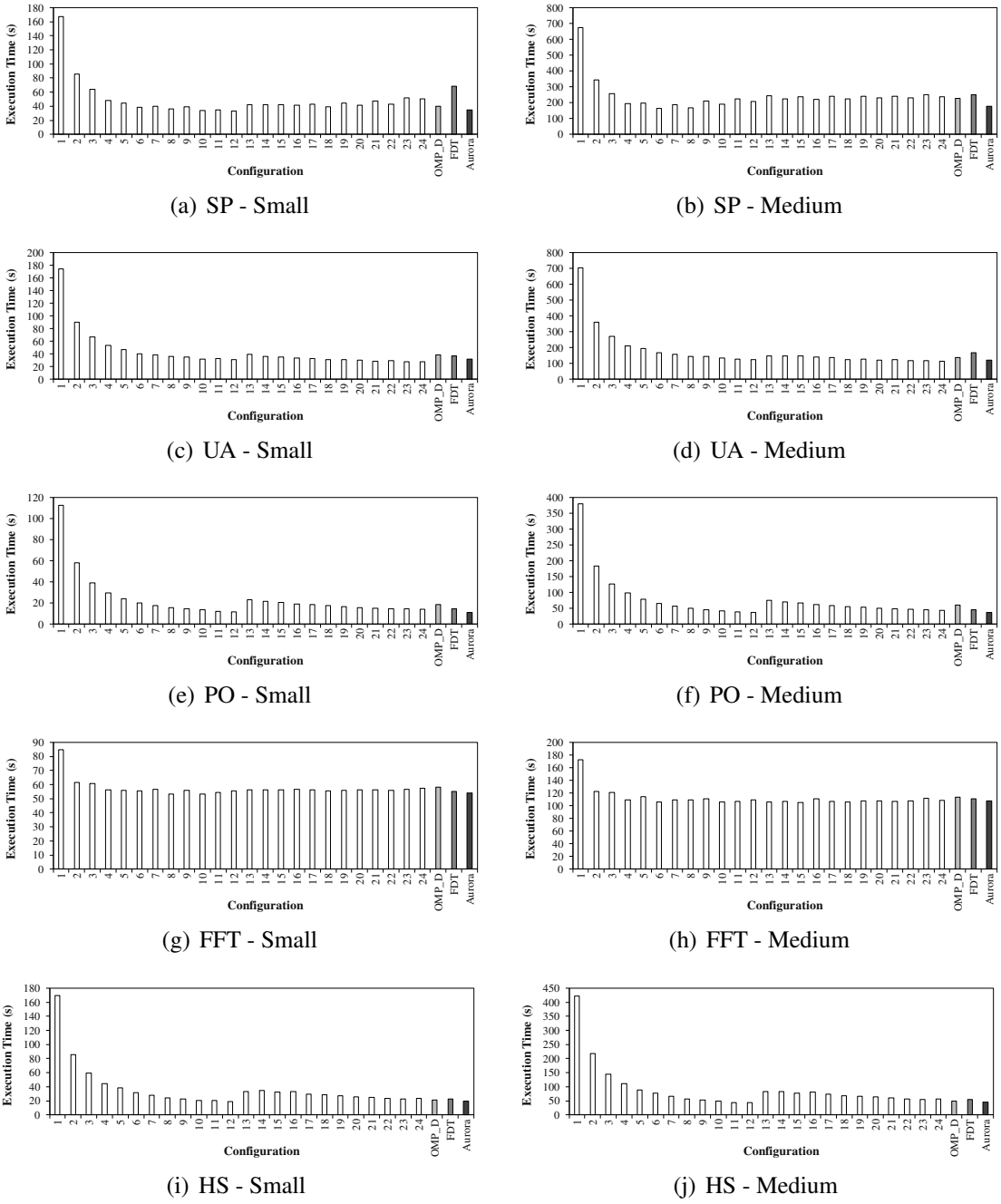
(e) FT - Small (f) FT - Medium

(g) LU - Small (h) LU - Medium

(i) MG - Small (j) MG - Medium

Source: The Author

Figure A.17: Performance - 24-Core System (Continuation)



(a) SP - Small (b) SP - Medium

(c) UA - Small (d) UA - Medium

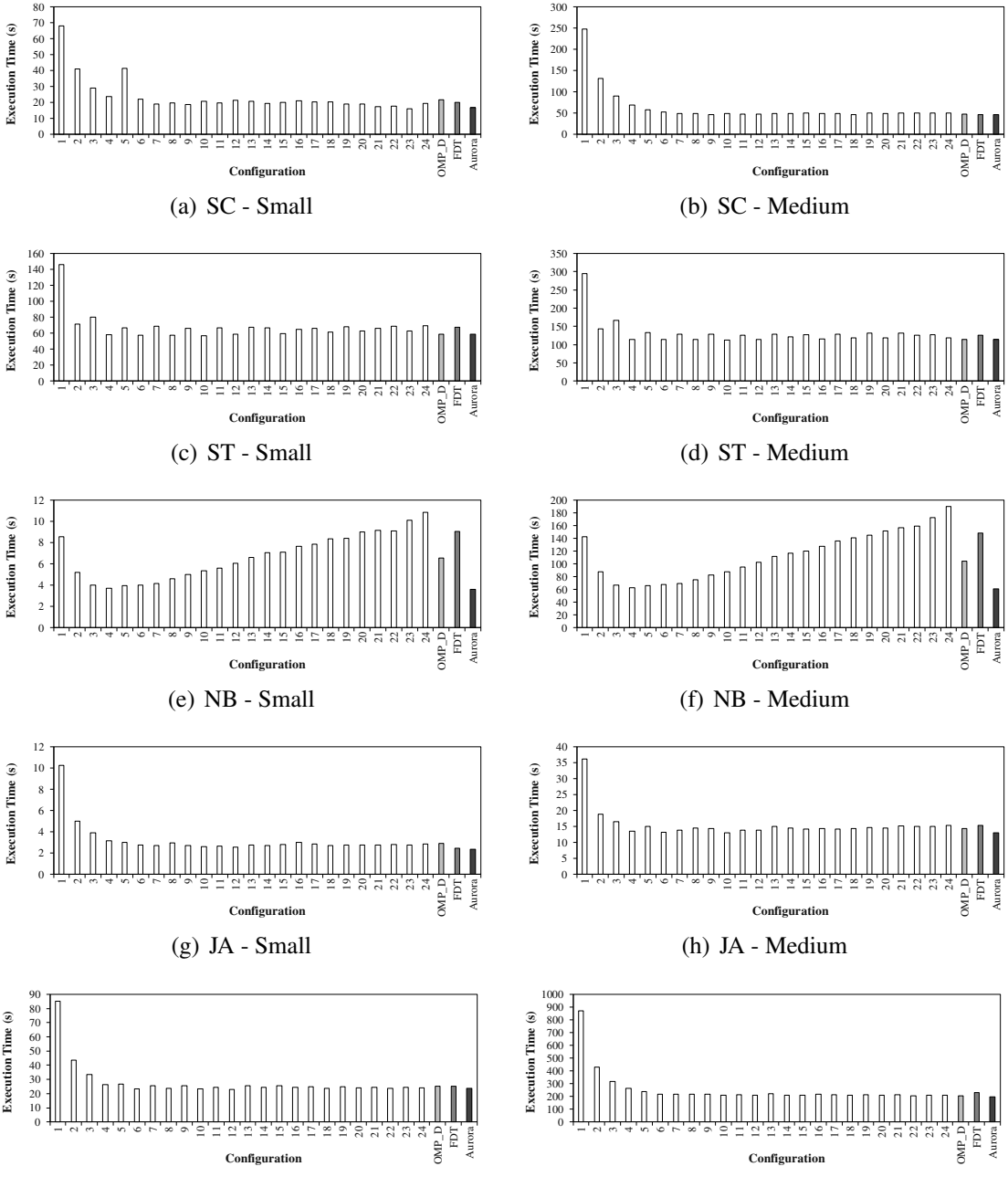
(e) PO - Small (f) PO - Medium

(g) FFT - Small (h) FFT - Medium

(i) HS - Small (j) HS - Medium

Source: The Author

Figure A.18: Performance - 24-Core System (Continuation)



(a) SC - Small (b) SC - Medium

(c) ST - Small (d) ST - Medium

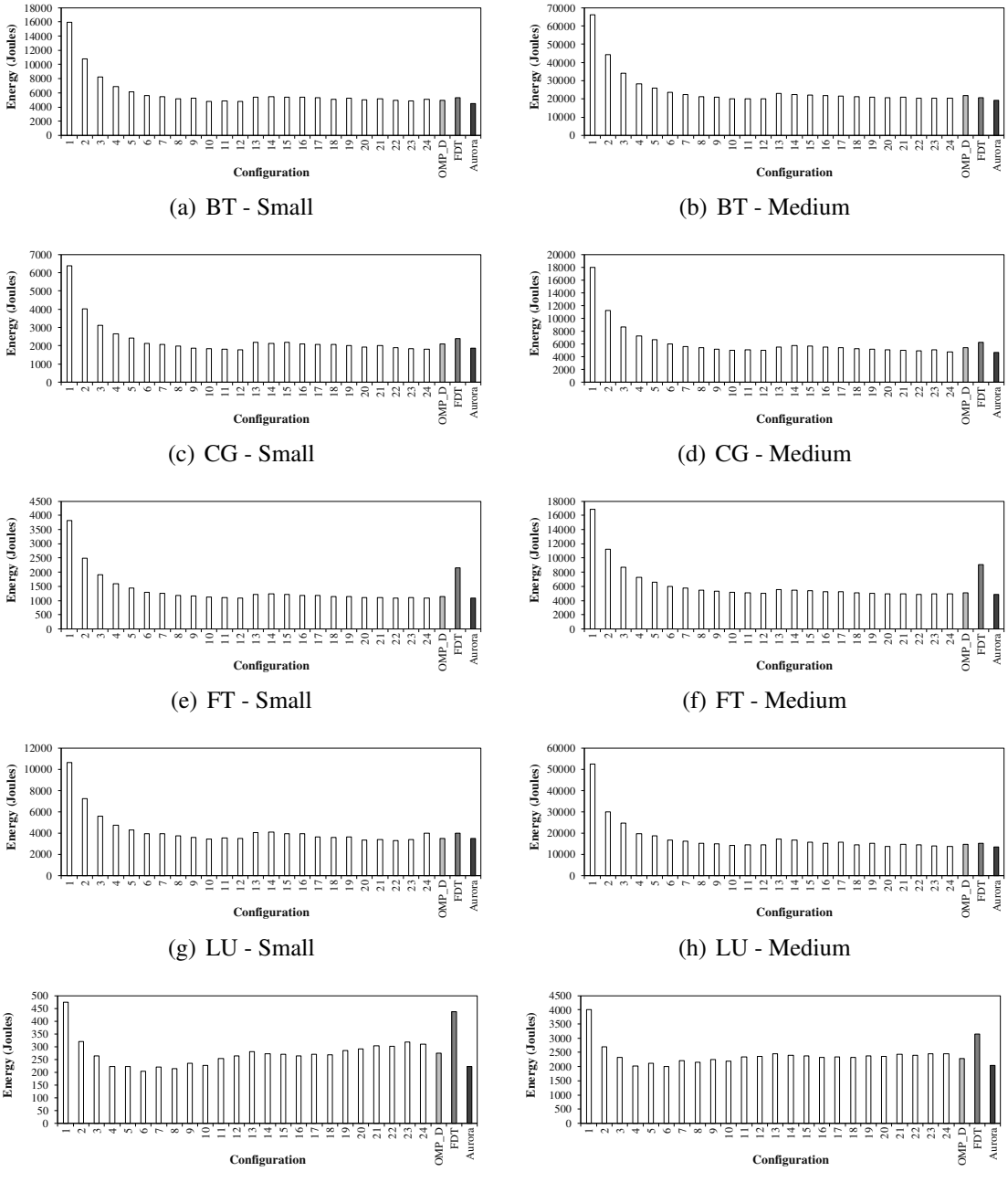
(e) NB - Small (f) NB - Medium

(g) JA - Small (h) JA - Medium

(i) HPCG - Small (j) HPCG - Medium

Source: The Author

Figure A.19: Energy Consumption - 24-Core System



(a) BT - Small (b) BT - Medium

(c) CG - Small (d) CG - Medium

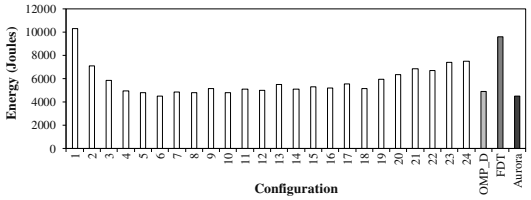
(e) FT - Small (f) FT - Medium

(g) LU - Small (h) LU - Medium

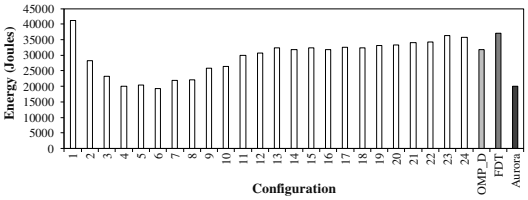
(i) MG - Small (j) MG - Medium

Source: The Author

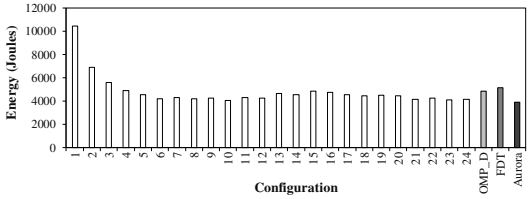
Figure A.20: Energy Consumption - 24-Core System (Continuation)



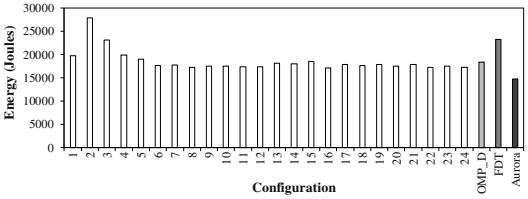
(a) SP - Small



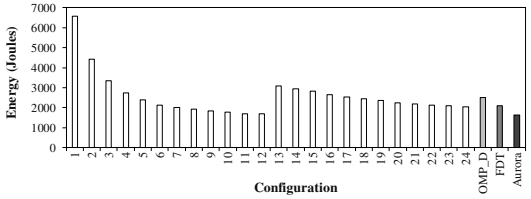
(b) SP - Medium



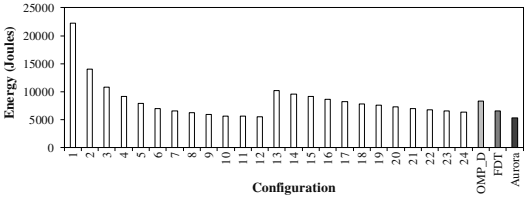
(c) UA - Small



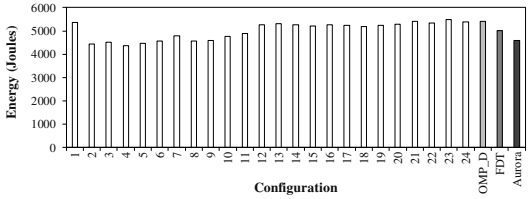
(d) UA - Medium



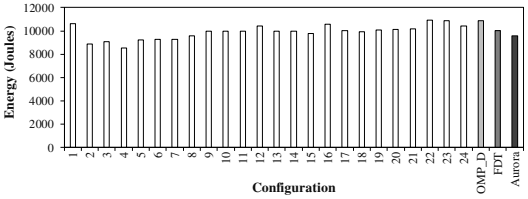
(e) PO - Small



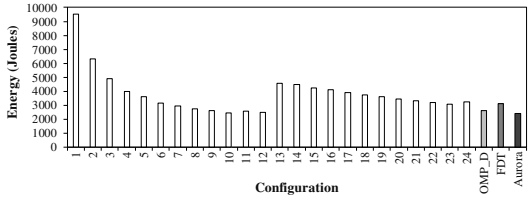
(f) PO - Medium



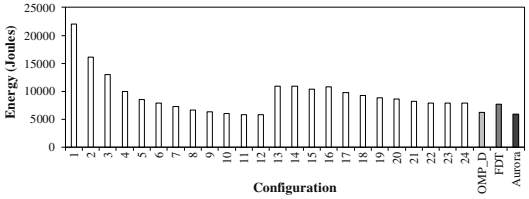
(g) FFT - Small



(h) FFT - Medium



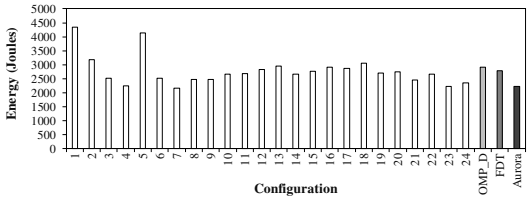
(i) HS - Small



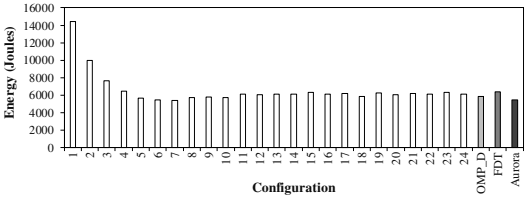
(j) HS - Medium

Source: The Author

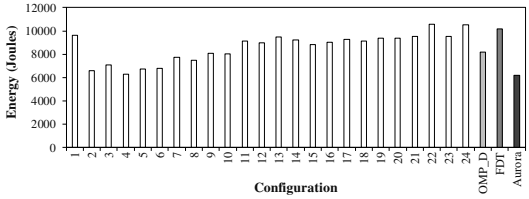
Figure A.21: Energy Consumption - 24-Core System (Continuation)



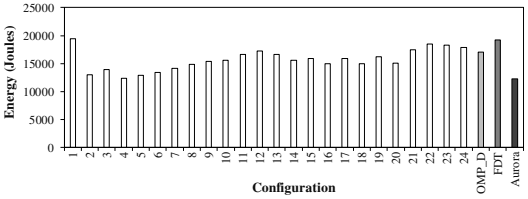
(a) SC - Small



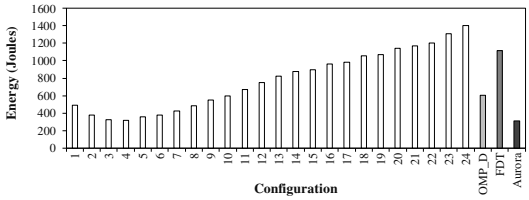
(b) SC - Medium



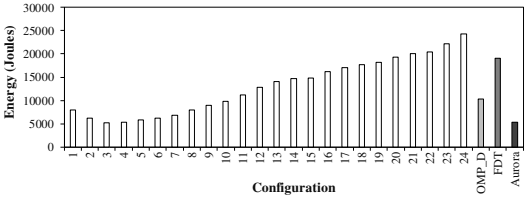
(c) ST - Small



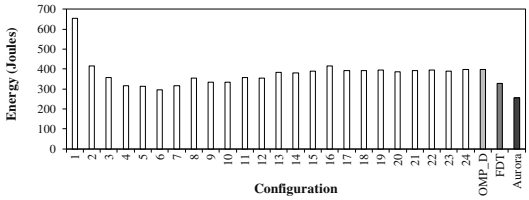
(d) ST - Medium



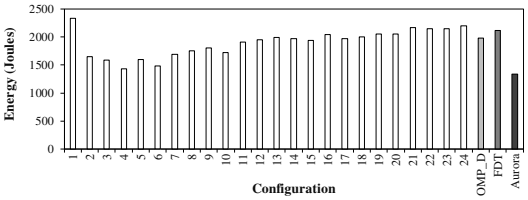
(e) NB - Small



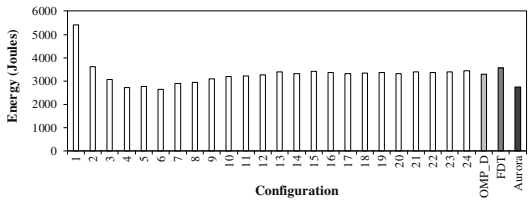
(f) NB - Medium



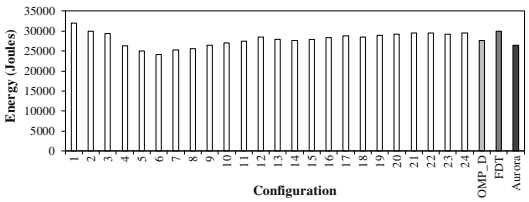
(g) JA - Small



(h) JA - Medium



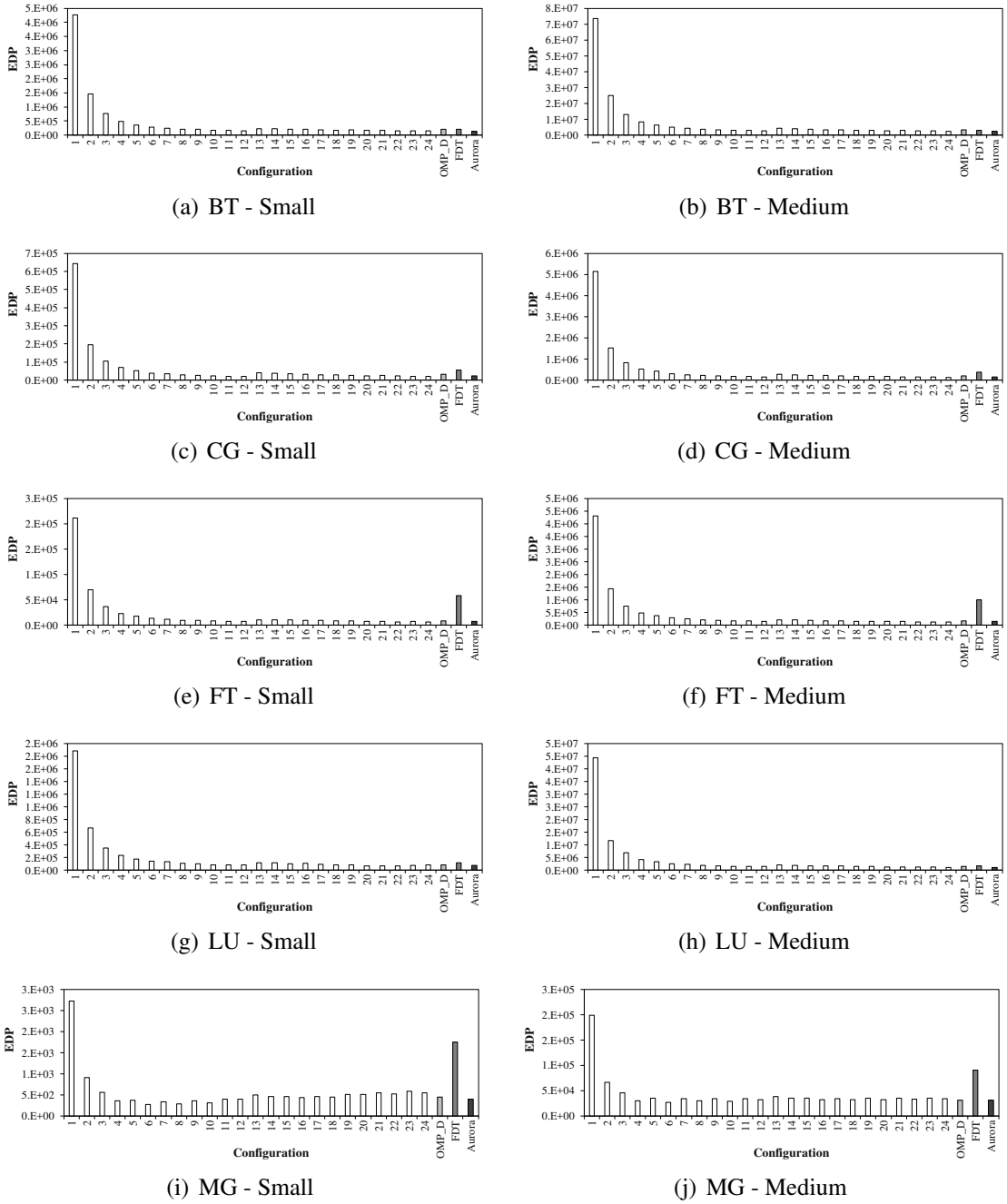
(i) HPCG - Small



(j) HPCG - Medium

Source: The Author

Figure A.22: EDP - 24-Core System



(a) BT - Small

(b) BT - Medium

(c) CG - Small

(d) CG - Medium

(e) FT - Small

(f) FT - Medium

(g) LU - Small

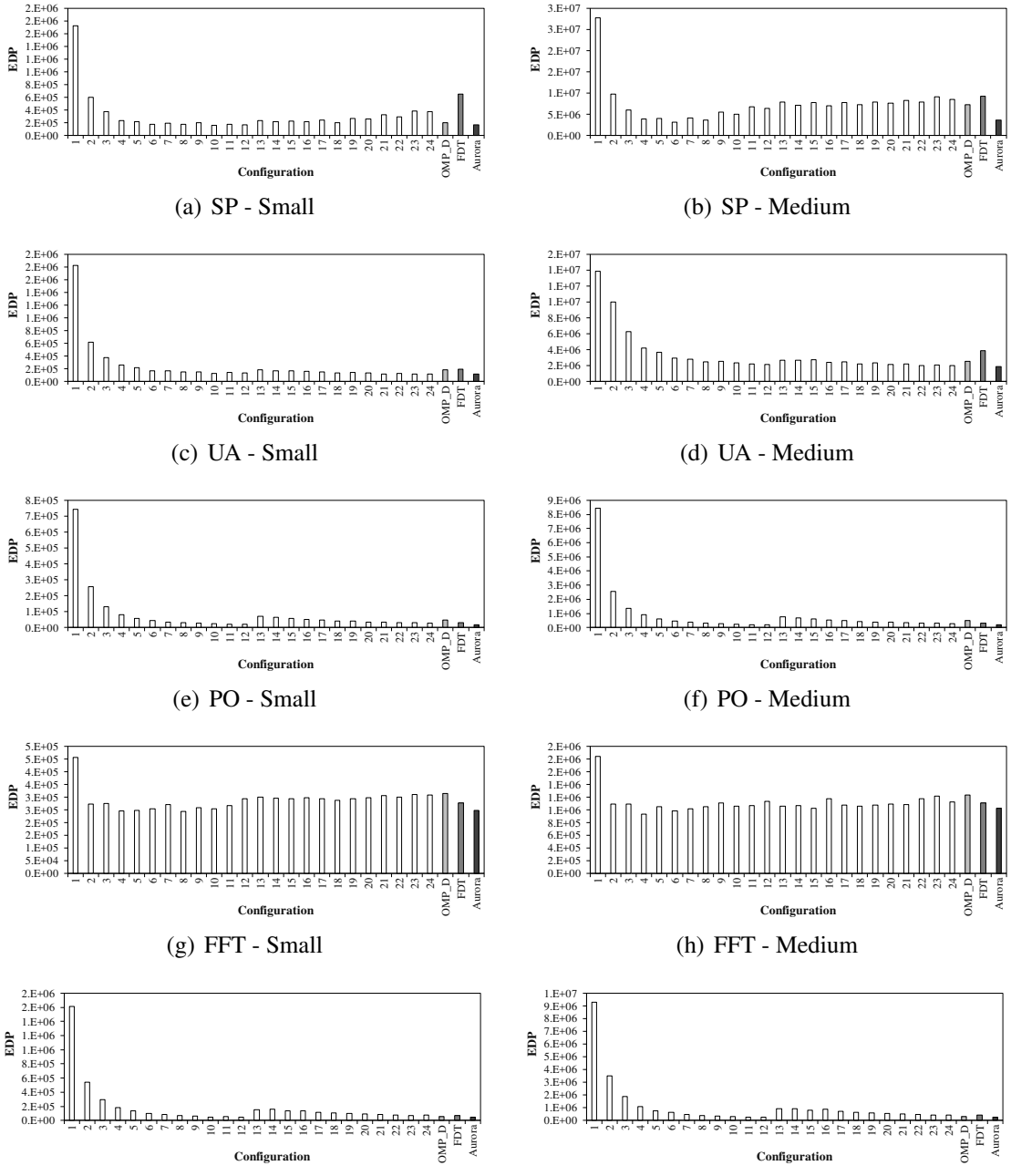
(h) LU - Medium

(i) MG - Small

(j) MG - Medium

Source: The Author

Figure A.23: EDP - 24-Core System (Continuation)



(a) SP - Small

(b) SP - Medium

(c) UA - Small

(d) UA - Medium

(e) PO - Small

(f) PO - Medium

(g) FFT - Small

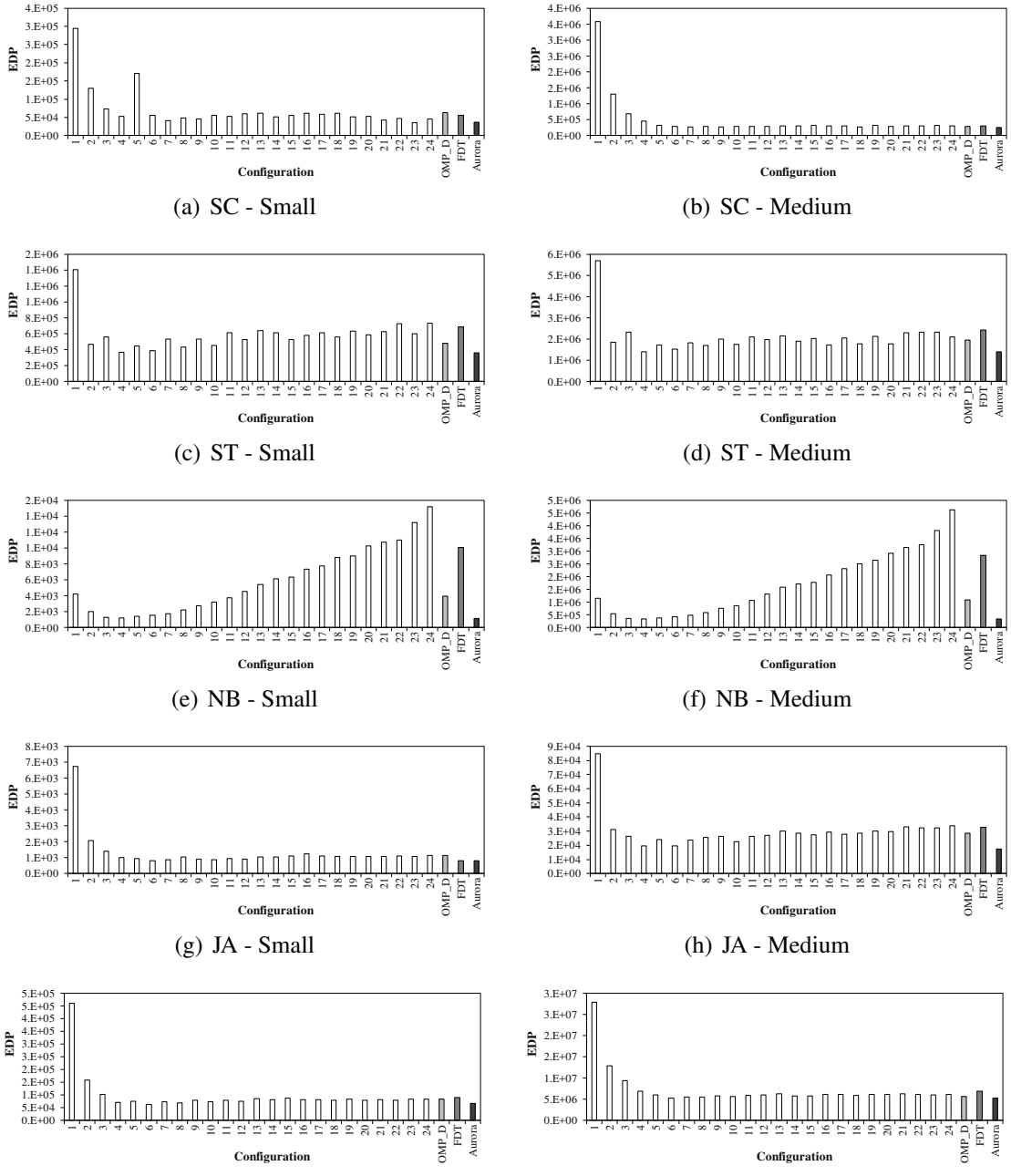
(h) FFT - Medium

(i) HS - Small

(j) HS - Medium

Source: The Author

Figure A.24: EDP - 24-Core System (Continuation)



(a) SC - Small (b) SC - Medium

(c) ST - Small (d) ST - Medium

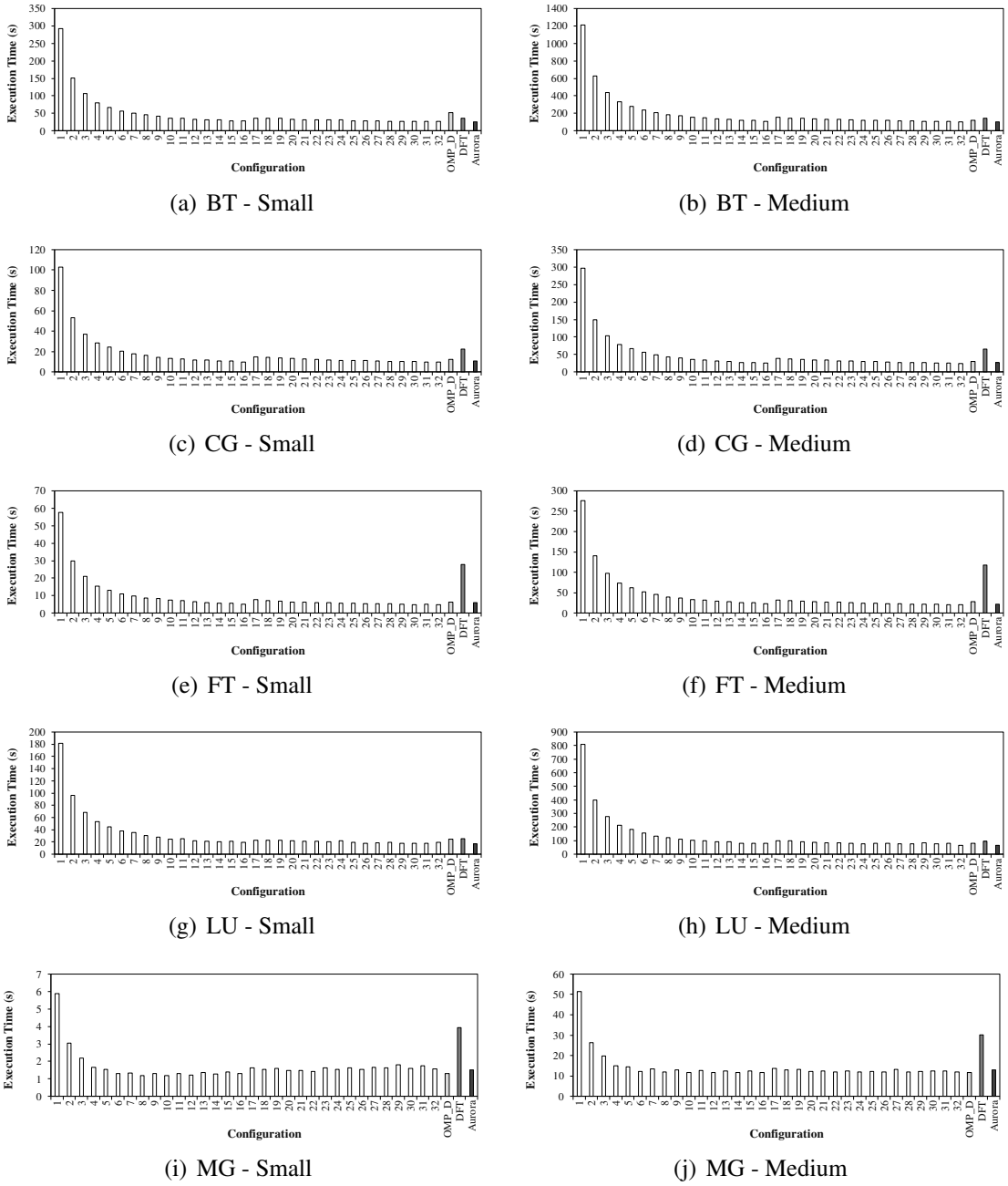
(e) NB - Small (f) NB - Medium

(g) JA - Small (h) JA - Medium

(i) HPCG - Small (j) HPCG - Medium

Source: The Author

Figure A.25: Performance - 32-Core System



(a) BT - Small

(b) BT - Medium

(c) CG - Small

(d) CG - Medium

(e) FT - Small

(f) FT - Medium

(g) LU - Small

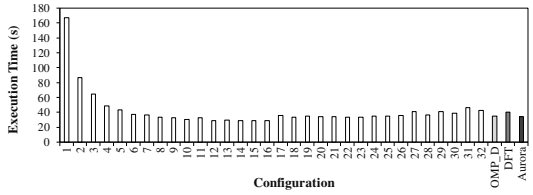
(h) LU - Medium

(i) MG - Small

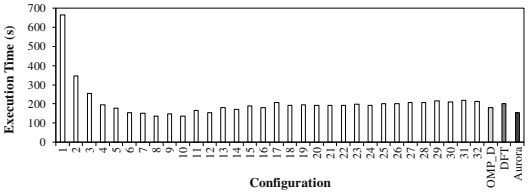
(j) MG - Medium

Source: The Author

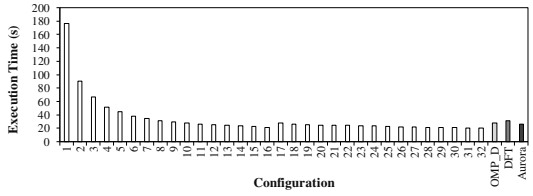
Figure A.26: Performance - 32-Core System (Continuation)



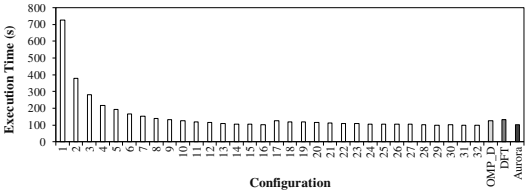
(a) SP - Small



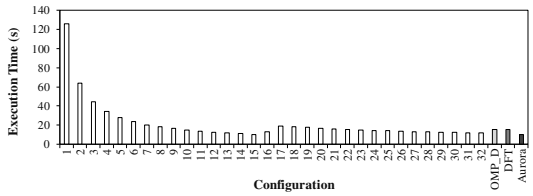
(b) SP - Medium



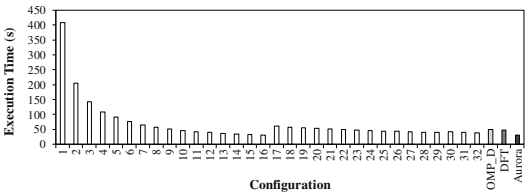
(c) UA - Small



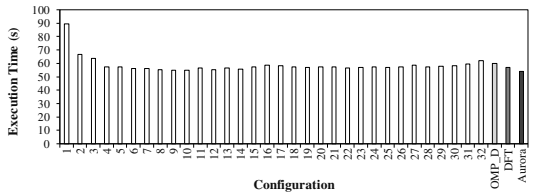
(d) UA - Medium



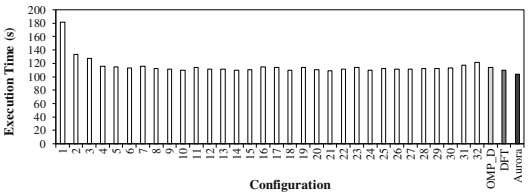
(e) PO - Small



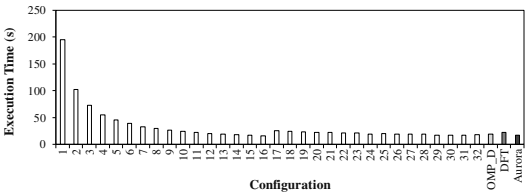
(f) PO - Medium



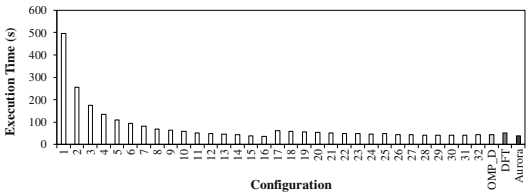
(g) FFT - Small



(h) FFT - Medium



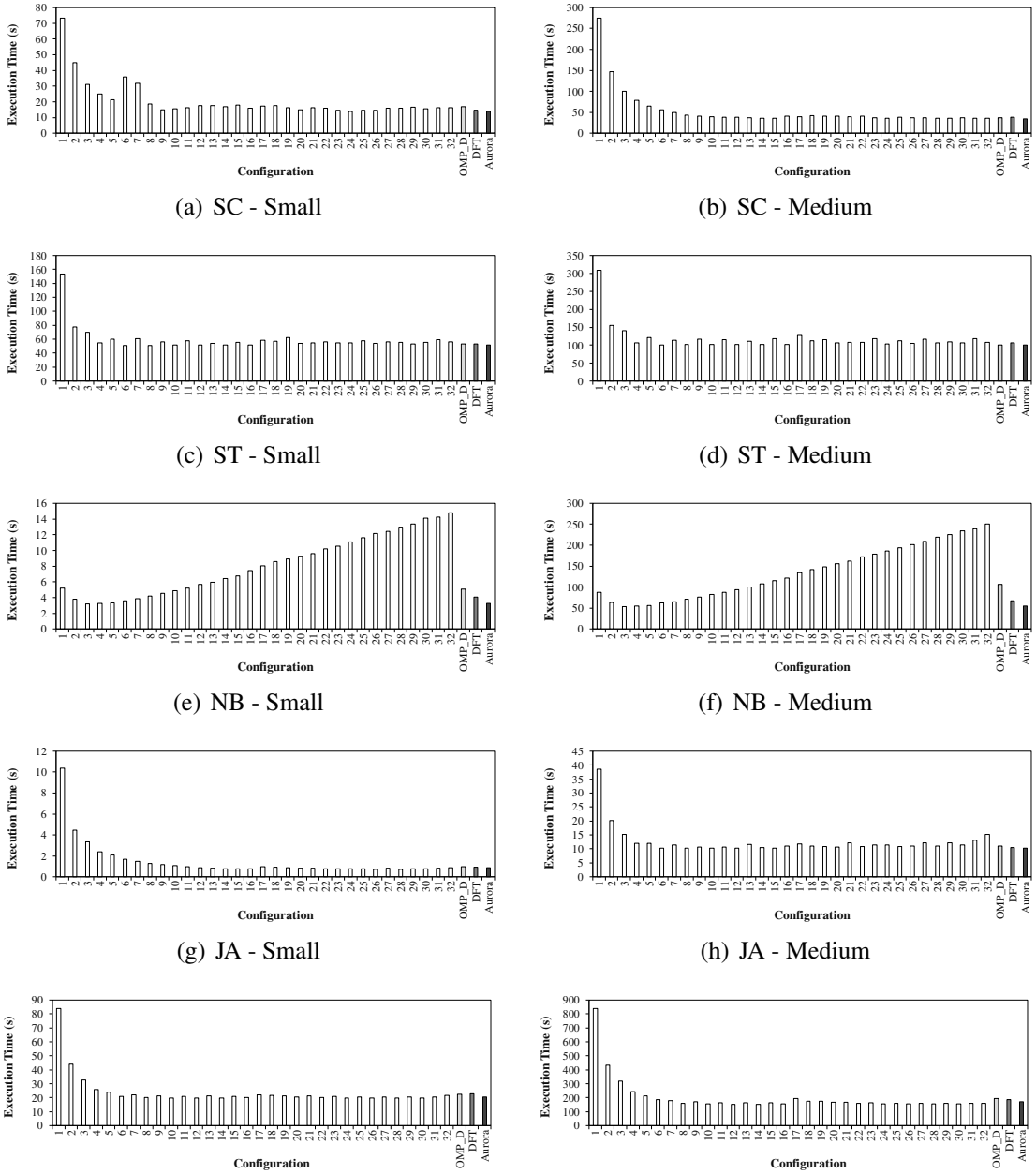
(i) HS - Small



(j) HS - Medium

Source: The Author

Figure A.27: Performance - 32-Core System (Continuation)



(a) SC - Small

(b) SC - Medium

(c) ST - Small

(d) ST - Medium

(e) NB - Small

(f) NB - Medium

(g) JA - Small

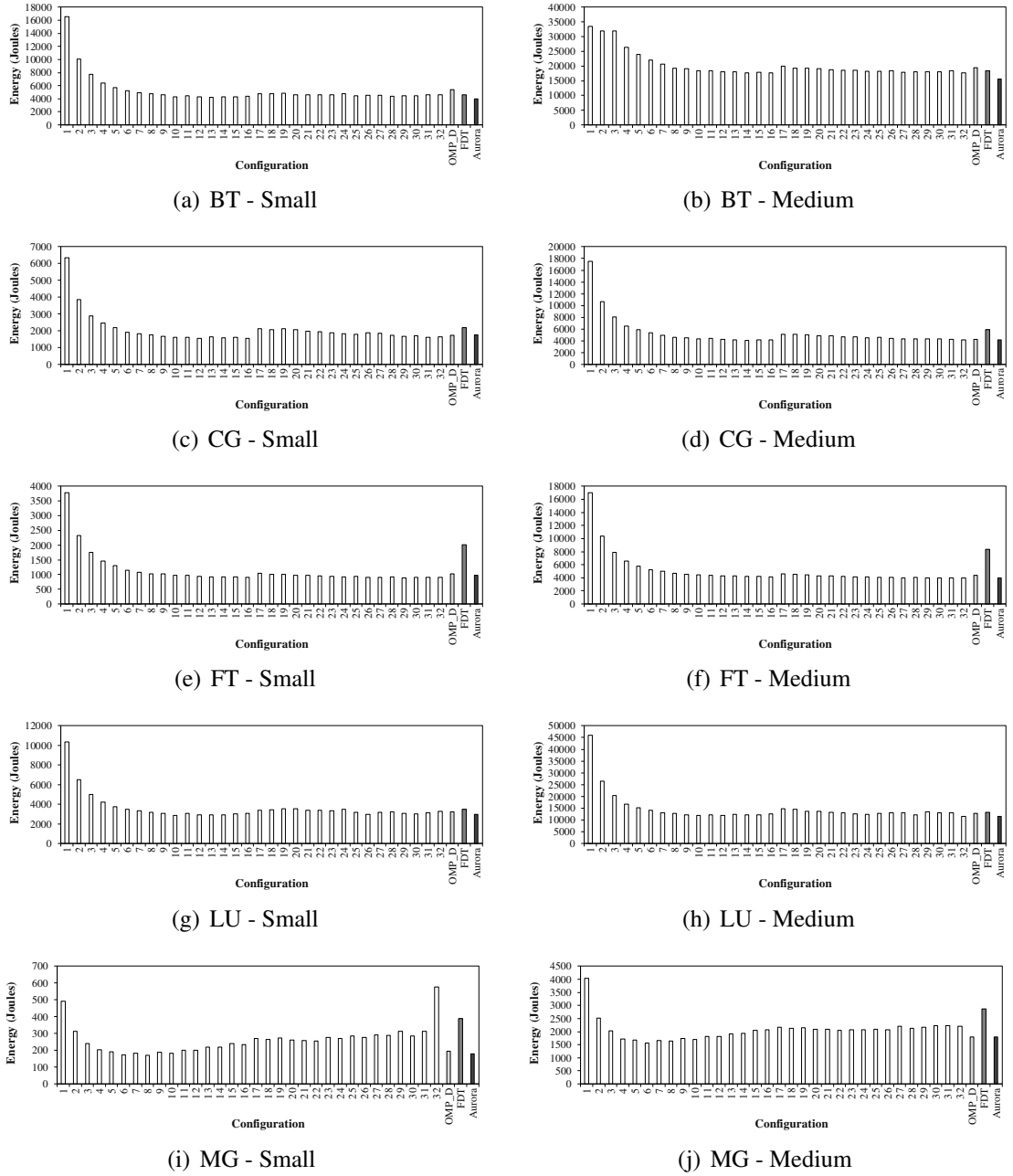
(h) JA - Medium

(i) HPCG - Small

(j) HPCG - Medium

Source: The Author

Figure A.28: Energy Consumption - 32-Core System



(a) BT - Small (b) BT - Medium

(c) CG - Small (d) CG - Medium

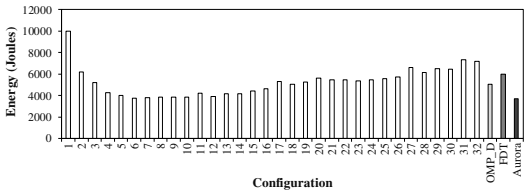
(e) FT - Small (f) FT - Medium

(g) LU - Small (h) LU - Medium

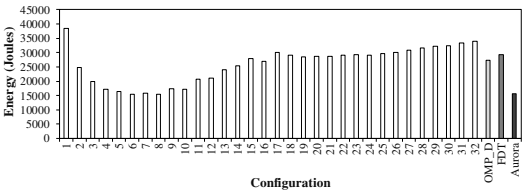
(i) MG - Small (j) MG - Medium

Source: The Author

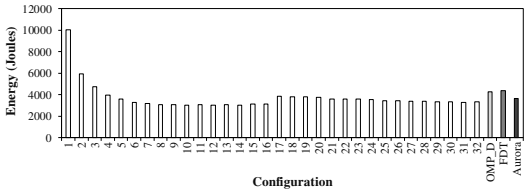
Figure A.29: Energy Consumption - 32-Core System (Continuation)



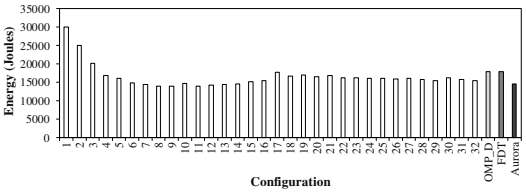
(a) SP - Small



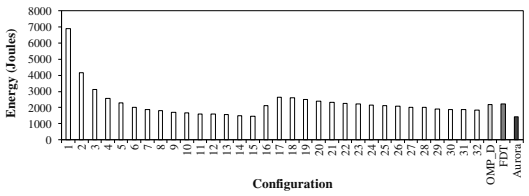
(b) SP - Medium



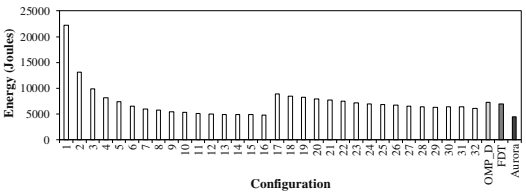
(c) UA - Small



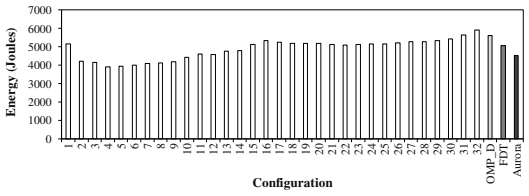
(d) UA - Medium



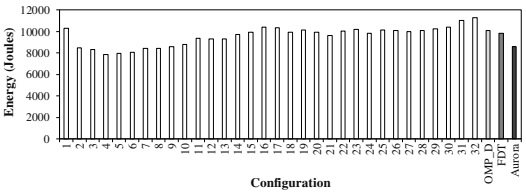
(e) PO - Small



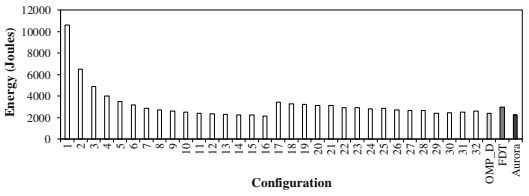
(f) PO - Medium



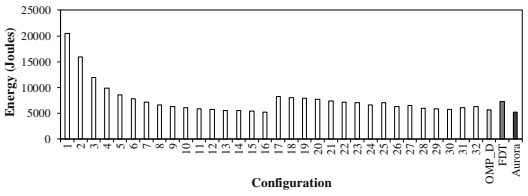
(g) FFT - Small



(h) FFT - Medium



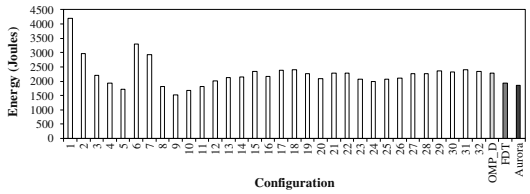
(i) HS - Small



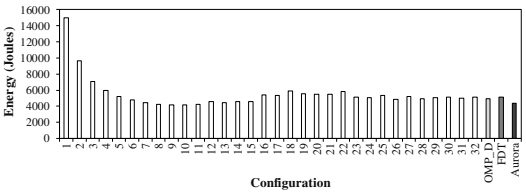
(j) HS - Medium

Source: The Author

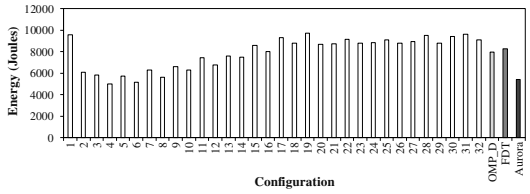
Figure A.30: Energy Consumption - 32-Core System (Continuation)



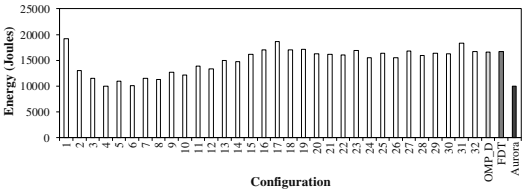
(a) SC - Small



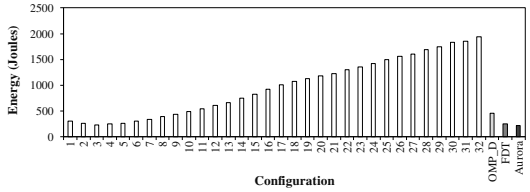
(b) SC - Medium



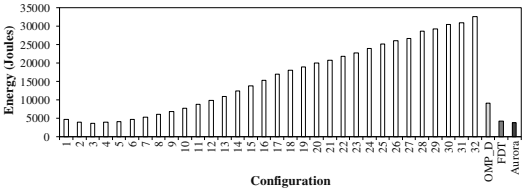
(c) ST - Small



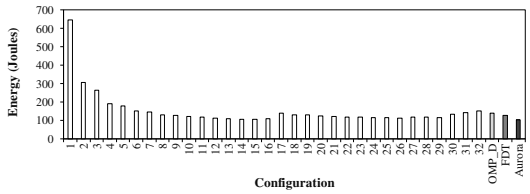
(d) ST - Medium



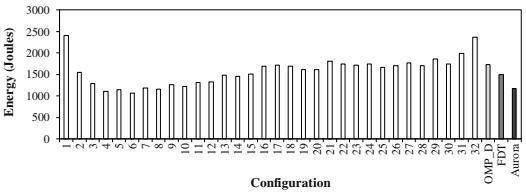
(e) NB - Small



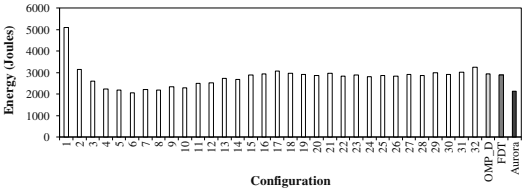
(f) NB - Medium



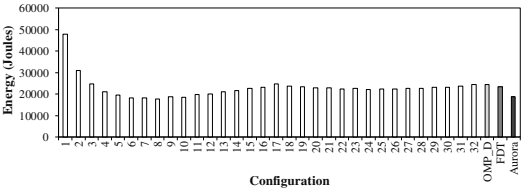
(g) JA - Small



(h) JA - Medium



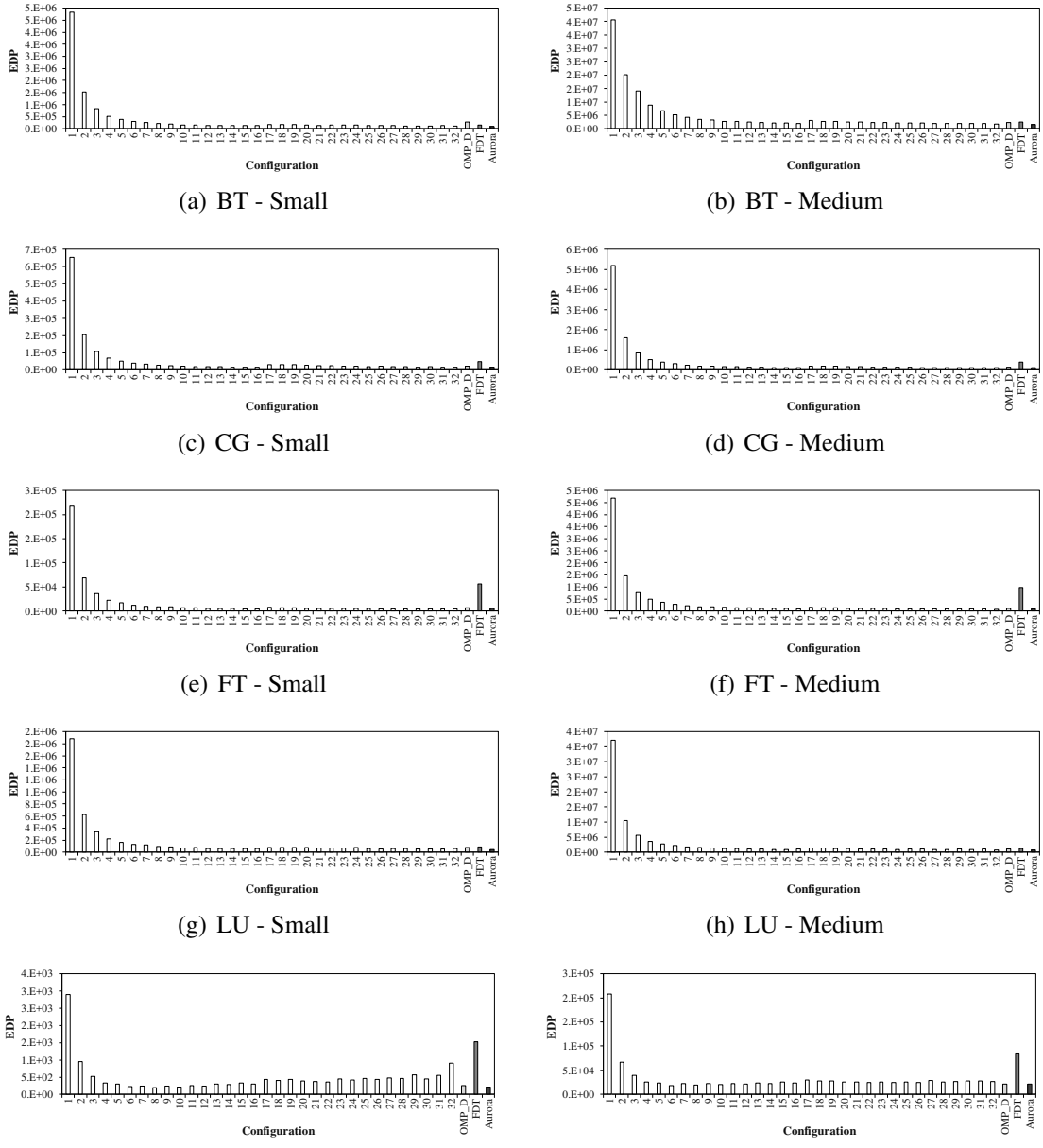
(i) HPCG - Small



(j) HPCG - Medium

Source: The Author

Figure A.31: EDP - 32-Core System



(a) BT - Small

(b) BT - Medium

(c) CG - Small

(d) CG - Medium

(e) FT - Small

(f) FT - Medium

(g) LU - Small

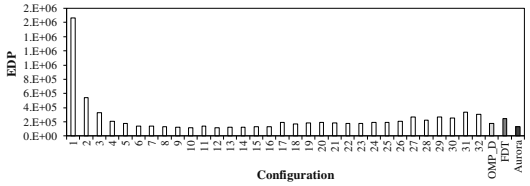
(h) LU - Medium

(i) MG - Small

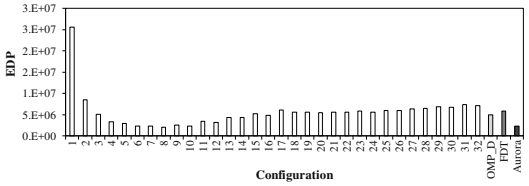
(j) MG - Medium

Source: The Author

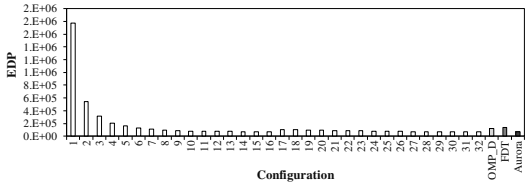
Figure A.32: EDP - 32-Core System (Continuation)



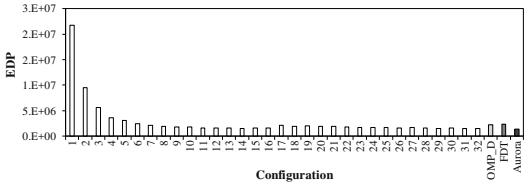
(a) SP - Small



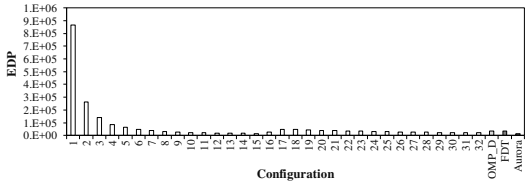
(b) SP - Medium



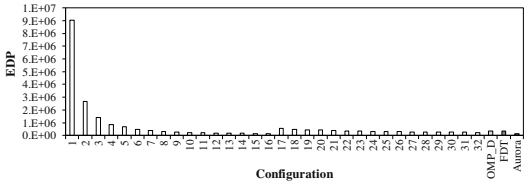
(c) UA - Small



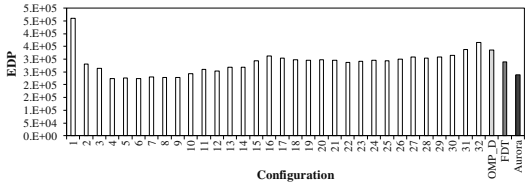
(d) UA - Medium



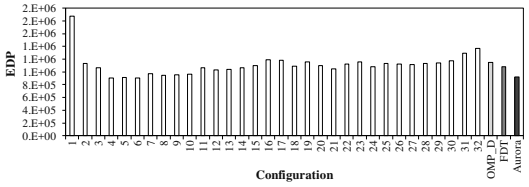
(e) PO - Small



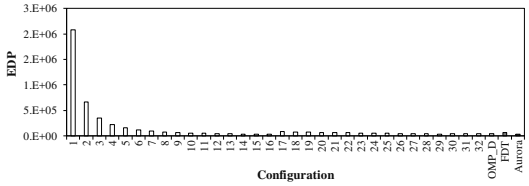
(f) PO - Medium



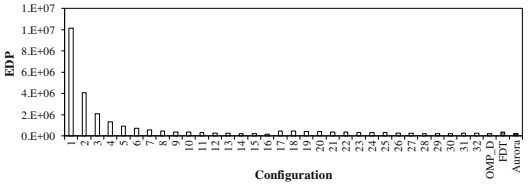
(g) FFT - Small



(h) FFT - Medium



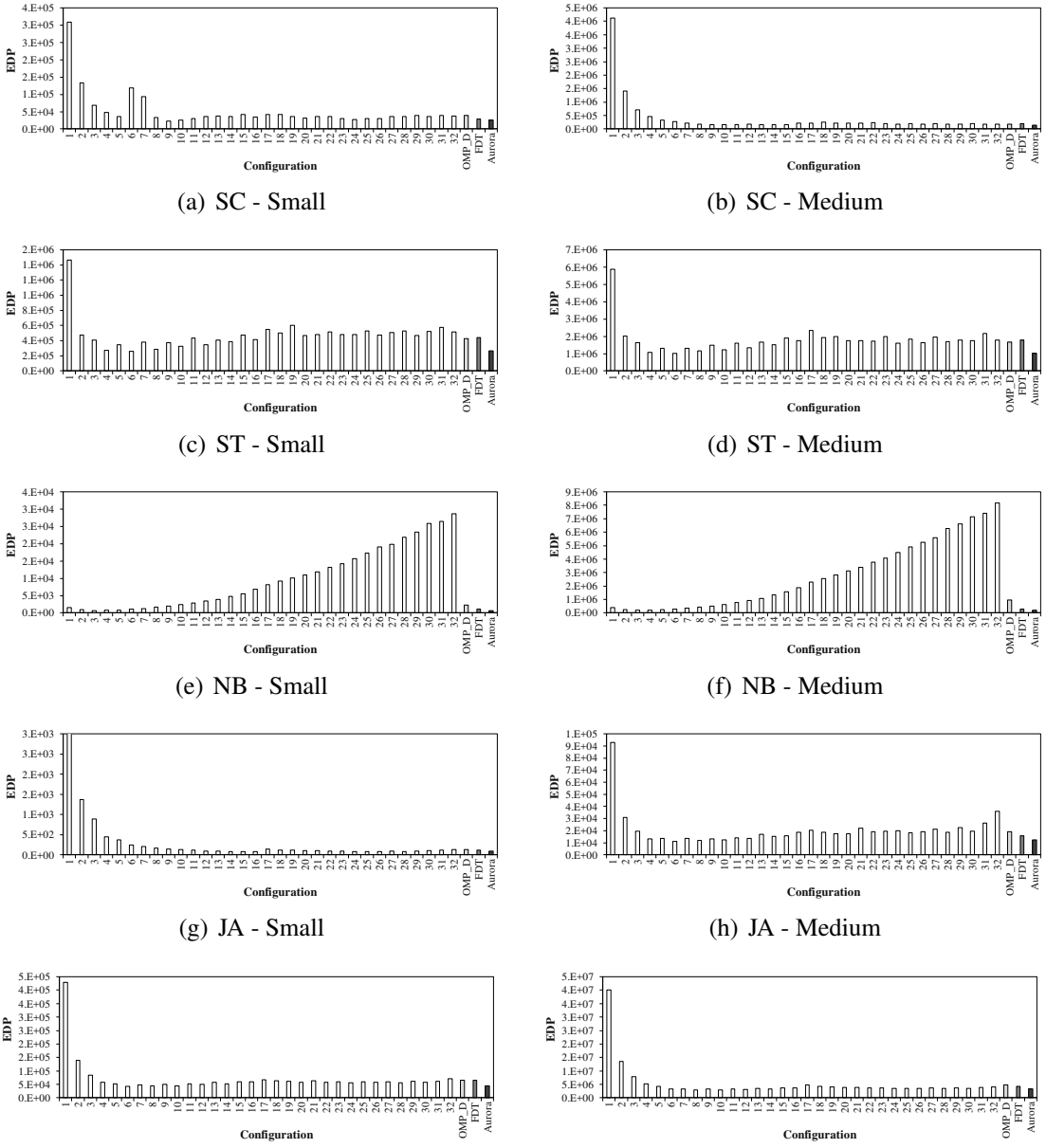
(i) HS - Small



(j) HS - Medium

Source: The Author

Figure A.33: EDP - 32-Core System (Continuation)



(a) SC - Small

(b) SC - Medium

(c) ST - Small

(d) ST - Medium

(e) NB - Small

(f) NB - Medium

(g) JA - Small

(h) JA - Medium

(i) HPCG - Small

(j) HPCG - Medium

Source: The Author

APPENDIX B — RESUMO EM PORTUGUÊS

B.1 Introdução

Com o aumento da complexidade de aplicações embarcadas que demandam maior eficiência computacional e a chegada da computação exascale, a grande preocupação está relacionada com a necessidade de aumentar o desempenho com o mínimo impacto possível no consumo de energia. Máquinas exascale possuirão 100 vezes mais poder de processamento que as melhores máquinas disponíveis atualmente, com requerimentos de energia que corresponde a energia gerada por uma usina nuclear de tamanho médio. Da mesma forma, a maioria dos sistemas embarcados são móveis e, portanto, dependentes de bateria. Assim, o desafio não é somente aumentar o desempenho, mas também melhorar a eficiência energética, isto é, maximizar a quantidade de computação por joule gasto.

A melhoria no desempenho pode ser obtida através da exploração de paralelismo no nível de instruções ou threads (ILP – Instruction Level Parallelism ou TLP – Thread Level Parallelism). No ILP, independentes instruções de um único programa são simultaneamente executadas em um processador superescalar enquanto houver unidades funcionais disponíveis. No entanto, os fluxos de instruções têm apenas uma quantidade limitada de paralelismo, resultando em enormes esforços para projetar uma microarquitetura que irá trazer marginal ganhos de desempenho com significativa sobrecarga no consumo de energia e área do processador. Mesmo se considerar um processador perfeito, a exploração de ILP chegará a um limite superior.

Desta forma, para continuar melhorando o desempenho e proporcionar uma melhor utilização dos transistores extras disponíveis, novos projetos começaram a explorar o paralelismo no nível de threads de forma mais agressiva. Neste caso, vários processadores executam simultaneamente partes do mesmo programa. No entanto, esta melhoria de desempenho não é linear e, por vezes, não escala com o número de threads devido a vários fatores, tais como a sincronização de dados, comunicação entre as threads, lock contention, a largura de banda do barramento de comunicação. Além disso, a sincronização e comunicação podem aumentar o consumo de energia das aplicações paralelas, uma vez que elas ocorrem através de regiões compartilhadas da memória. Estas regiões são mais distantes do processador (por exemplo, memória cache L3 e compartilhada) e têm maior consumo de energia e tempo de acesso quando comparadas a memórias que estão mais próximas do processador (por exemplo, registradores e caches L1 e L2).

Considerando o cenário acima, escolher o número correto de threads a ser usado no desenvolvimento de aplicações paralelas pode fornecer melhorias no desempenho e economia de energia. Por exemplo, uma aplicação pode ser executada com poucas threads, deixando recursos de hardware inativos (por exemplo, memórias, cores e barramento de comunicação). Por outro lado, esses mesmos recursos podem saturar devido à alta taxa de comunicação e sincronização se muitas threads são criadas. Em ambas as situações, a execução paralela será ineficiente, sendo mais lenta e consumindo mais energia que se estivesse sendo executada com o número apropriado de threads. Mesmo assumindo que o desenvolvedor pode decidir o número ideal de threads para executar uma aplicação em antecipação, este número pode mudar devido diversas razões: conjunto de entrada, mética avaliada, processador utilizado e númeo de regiões paralelas de uma aplicação.

Para abordar esse complexo problema de escolha do número apropriado de threads para executar uma aplicação paralela, métodos para encontrar automaticamente este número são usados com três abordagens principais: off-line, on-line e híbrido. Em um método off-line, o framework executa uma aplicação com um número diferente de threads de cada vez e escolhe o número que oferece o melhor resultado. No entanto, o desenvolvedor de software deve reexecutar o framework sempre que o conjunto de entrada e/ou processador for alterado. Por outro lado, em métodos on-line, o framework ajusta o número de threads em tempo execução através de informações do ambiente de execução, impondo um overhead extra para a execução da aplicação. Por fim, métodos híbridos compreendem uma fase off-line inicial para aprender as características da aplicação e do ambiente de execução; e uma fase on-line na qual o número de threads é ajustado dinamicamente durante a execução da aplicação. Neste caso, as limitações são as mesmas que as encontradas no método off-line e on-line.

B.2 Contribuições

Considerando o cenário apresentado acima, esta tese apresenta as seguintes contribuições:

- Conduzir um extenso estudo ds oportunidades de computação paralela com relação a diferentes interfaces de programação paralela e processadores de propósito geral e embarcados.

- Desenvolver uma biblioteca para adaptar o número de threads de aplicações OpenMP, de maneira automática ao usuário.
- Através de correlação matemática, apresentar os gargalos que afetam a escalabilidade das aplicações paralelas.
- Incorporar a abordagem desenvolvida na biblioteca do OpenMP (*libgomp*), para que a adaptação do número de threads ocorra de maneira transparente e automática para o usuário.

B.3 Possibilidade de Exploração de Computação Paralela

Nesta etapa da tese, a exploração estática para combinações ideais de processadores, modelos de comunicação e exploração de TLP para alcançar os melhores resultados em desempenho, energia e EDP foi realizada. Um grande número de variáveis foi considerado: 5 processadores multicore com diferentes microarquiteturas e ISAs; 14 benchmarks paralelos classicados de acordo com a taxa de comunicação; quatro interfaces de programação paralela classificadas em duas classes de modelos de comunicação; diferentes níveis de exploração de TLP; e quatro níveis diferentes de potência estática do processador. Demonstramos que apesar de haver combinações com o melhor desempenho e o menor consumo de energia, não existe uma única combinação que ofereça o melhor resultado para ambos ao mesmo tempo. No entanto, encontramos alguns resultados significativos, resumidos a seguir.

Para as aplicações com alta demanda de comunicação, a escolha da PPI é importante para o resultado final: PThreads mostrou ser a melhor escolha para todos os processadores Intel (GPP ou embarcados), pois proporciona consideráveis melhorias de desempenho em relação aos outros ao mesmo preço do consumo de energia da versão sequencial. Por outro lado, ao explorar laços paralelos, o OpenMP é melhor para processadores ARM, já que o impacto do mecanismo de busy-waiting é menor nesses processadores do que nos Intel. Em geral, o MPI é a pior escolha para todos os processadores, apresentando pouca escalabilidade: à medida que a exploração de TLP aumenta, os ganhos de desempenho são limitados pela comunicação baseada em mensagens; e o consumo de energia aumenta quando comparado à sua versão sequencial.

O cenário é diferente para benchmarks com baixa demanda de comunicação. Para estes, o que importa é o modelo de comunicação, em vez de uma PPI específica. Uma vez

que as aplicações são CPU-bound, a maneira como o processador explora o ILP e a frequência de operação tem maior importância. Independentemente da PPI, o desempenho aumenta e a energia diminui à medida que o TLP aumenta, resultando em um melhor EDP. Portanto, mesmo que essas aplicações escalam melhor que as com alta demanda de comunicação, o espaço de design é mais restrito, oferecendo menos oportunidades de otimização.

Com toda a análise realizada, mostrou-se que existe um grande espaço de exploração no que tende a escolha do número ótimo de threads para executar uma dada aplicação. No entanto, o desenvolvimento de uma única abordagem para otimizar o número de threads envolvendo todas as PPIs estudadas neste capítulo não é factível devido a maneira como cada uma delas gerencia a exploração do paralelismo. Desta maneira, o OpenMP foi escolhido para receber a implementação de tal abordagem.

B.4 Otimização de Aplicações OpenMP

Duas abordagens foram propostas para realizar a otimização de aplicações implementadas com OpenMP através da busca pelo número ideal de threads para executar cada região paralela. Elas são apresentadas nas subseções a seguir.

B.4.1 LAANT

LAANT é uma biblioteca desenvolvida para adaptar automaticamente o número de threads das aplicações OpenMP. Ela é capaz de encontrar em tempo de execução o número ideal de threads para cada região paralela da aplicação, aprendendo o melhor número de threads conforme o programa é executado e resultando em melhorias significativas no EDP com uma sobrecarga quase desprezível.

O processo proposto é completamente automático para o desenvolvedor. Ele pode ser aplicado a qualquer aplicativo paralelo desenvolvido com a interface OpenMP, anotando código nas regiões paralelas, que já foram identificadas pelas diretivas OpenMP. LAANT pode otimizar as regiões paralelas para diferentes métricas, como EDP, desempenho, consumo de energia, entre outros.

B.4.1.1 Metodologia e Avaliação

Nove aplicações já paralelizadas com OpenMP e escritas em C e C++ de um vasto conjunto de benchmarks e domínios foram escolhidas. Os experimentos foram realizados em três diferentes processadores multicore: Intel Core i7, Xeon E5-2630 e Xeon E5-2650. Cada benchmark foi executado com dois conjunto de entrada: pequeno e médio. LAANT foi comparado com duas abordagens: a maneira usual de executar aplicações paralelas, onde o número de threads é igual ao número de núcleos do processador (chamado de baseline); e OpenMP *Dynamic*, uma abordagem do próprio OpenMP que ajusta o número de threads de uma região paralela buscando o melhor aproveitamento dos recursos computacionais.

Comparando com o baseline e considerando a média geométrica de todo o conjunto de benchmarks, LAANT reduziu o EDP em até 29% no processador com 24-cores. O menor ganho de LAANT foi obtido no processador com 32-cores, e, mesmo neste caso, LAANT apresentou 15% de redução no EDP.

Já na comparação com o OpenMP *Dynamic*, LAANT é melhor na maioria dos casos. Na média geométrica de todo o conjunto de benchmarks, LAANT tem 21% de ganhos de EDP no processador com 8-cores, 44% no processador com 24-cores, e 32% no processador com 32-cores.

B.4.2 Aurora

Aurora é uma extensão de LAANT. Da mesma forma, é uma abordagem que automaticamente encontra, em tempo de execução e de acordo com uma determinada métrica definida a priori pelo usuário, o número ideal de threads para cada região paralela de qualquer aplicação OpenMP. Além disso, Aurora pode se readaptar no caso de uma mudança no comportamento de uma determinada região paralela durante a execução do programa.

Aurora foi construída sobre a biblioteca original do OpenMP, sendo completamente transparente para o designer e usuário final: dado um binário de aplicação OpenMP, Aurora é executada sem nenhuma alteração de código. Portanto, as aplicações OpenMP existentes não precisam ser anotadas, recompiladas ou passar por qualquer transformação de código. Essa transpartência é obtida redirecionando as chamadas originalmente direcionadas para a biblioteca OpenMP dinamicamente conectada ao Aurora. Esse redirecionamento é configurado através da definição de uma variável de ambiente no sistema operacional.

B.4.2.1 Metodologia e Avaliação

Quinze aplicações já paralelizadas em OpenMP, de um extenso conjunto de benchmarks e classificadas de acordo com a característica que limita sua escalabilidade foram consideradas. Cada aplicação foi executada com dois conjuntos de entrada: pequeno e médio. Os experimentos foram realizados em quatro processadores multicore: Intel Core i5-4460, Intel Core i7-6700, Intel Xeon E5-2630 e Intel Xeon E5-2640.

Aurora foi comparada com outras quatro abordagens: **Baseline**: a aplicação é executada com o número máximo de threads disponíveis no sistema; **OMP_Dynamic**: abordagem interna do OpenMP; **Feedback-driven threading (FDT)**: abordagem proposta por Suleman, et al. em (SULEMAN; QURESHI; PATT, 2008); e **Oráculo**: que consiste da melhor solução possível para cada região paralela.

Aurora vs Baseline: na maioria dos casos, Aurora mostra melhorias em relação a qualquer métrica. Por outro lado, se considerarmos a média geométrica em qualquer cenário, Aurora é sempre melhor (em cenários muito específicos onde a exploração do espaço de design é limitada, ela apresenta resultados similares ao baseline). Considerando seu melhor caso comparado com o baseline, o tempo de execução foi reduzido em 16% com o conjunto de entrada média executando no sistema de 32 núcleos. O melhor cenário para o consumo de energia e EDP foi com o conjunto de entradas pequeno no sistema de 32 núcleos: a energia é reduzida em 34% e o EDP é melhorado em 47%. Ao considerar a média geométrica geral (todo o conjunto de referência e todos os processadores), Aurora forneceu 10% de melhorias de desempenho, 20% de reduções de energia e 28% de melhorias no EDP.

Aurora vs OMP_Dynamic: Considerando o melhor caso para cada métrica considerando a média geométrica, Aurora reduziu o tempo de execução em 26% (entrada média na máquina de 4 núcleos), consumo de energia em 24% (entrada média no sistema de 32 núcleos) e EDP em 38% (entrada pequena no sistema de 4 núcleos). Na média geométrica geral, a Aurora foi 11% mais rápida, economizou 17% de energia e melhorou o EDP em 32%.

Aurora vs FDT: Aurora supera o FDT em todos os casos em relação a qualquer métrica e processador. No melhor caso da Aurora para cada métrica em relação à média geométrica, Aurora reduziu o tempo de execução e o consumo de energia em 34%, e o EDP em 56% (entrada pequena na máquina de 24 núcleos). Ao considerar a média geométrica geral, Aurora forneceu 26% de melhorias de desempenho, 25% de reduções de energia e 45% de melhorias no EDP.

Aurora vs Oráculo: Ao considerar essa solução ótima, podemos medir o custo da curva de aprendizado, ou seja, a sobrecarga de Aurora, que é originada de duas situações diferentes: a execução do próprio algoritmo de busca; e a execução de uma determinada região paralela com um número de threads que não é o ideal, enquanto o algoritmo de busca está tentando diferentes possibilidades de convergir para o número ideal. Aurora mostrou alto sobrecusto nas seguintes situações:

- O melhor resultado é obtido com o número máximo de threads ou um número próximo a ele.
- A região paralela tem um número relativamente pequeno de interações, mas é executada por um tempo significativo.
- Aplicatções com tempo de execução curto (isto é, menos de 10 segundos).
- Aplicações com muitas regiões paralelas, nas quais a maioria delas tem uma carga de trabalho baixa.