Chapter 3
The Impact of Parallel Programming Interfaces on Energy

3.1 Methodology

3.1.1 Benchmarks

In order to study the characteristics of each PPI regarding the thread/process management and synchronization/communication, fourteen parallel benchmarks were implemented and parallelized in C language and classified into two classes: high and low communication (HC and LC). For that, we 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 3.1 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 [74]. 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 Fig. 3.1a. 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 Fig. 3.1b).

Since the way a parallel application is written may influence its behavior during execution, we have followed the guidelines indicated by [17, 36, 38] and [22]. The OpenMP implementations were parallelized using parallel loops, splitting the number of loops iterations (*for*) among threads. As discussed in [22], this approach is ideal for applications that compute on uni- and bi-dimensional 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 [69], the static scheduler with coarse granularity presents

A. Francisco Lorenzon, A. C. S. Beck Filho, *Parallel Computing Hits the Power Wall*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-030-28719-1_3

Table 3.1 Main characteristics of the benchmarks

Benchmarks		Operations to exchange data (Total per no. of threads/processes) 2	3	4	8	Input size
HC	Game of life	414	621	1079	1625	4096 × 4096
	Gauss–Seidel	20,004	20,006	20,008	20,016	2048 × 2048
	Gram–Schmidt	3,009,277	4,604,284	6,385,952	12,472,634	2048 × 2048
	Jacobi	4004	6006	8008	16,016	2048 × 2048
	Odd–even sort	300,004	450,006	600,008	1,200,016	150,000
	Turing ring	16,000	24,000	32,000	64,000	2048 × 2048
LC	Calc. of the PI number	4	6	8	16	4 billions
	DFT	4	6	8	16	32,368
	Dijkstra	4	6	8	16	2048 × 2048
	Dot-product	4	6	8	16	15 billions
	Harmonic series	8	12	16	32	100,000
	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

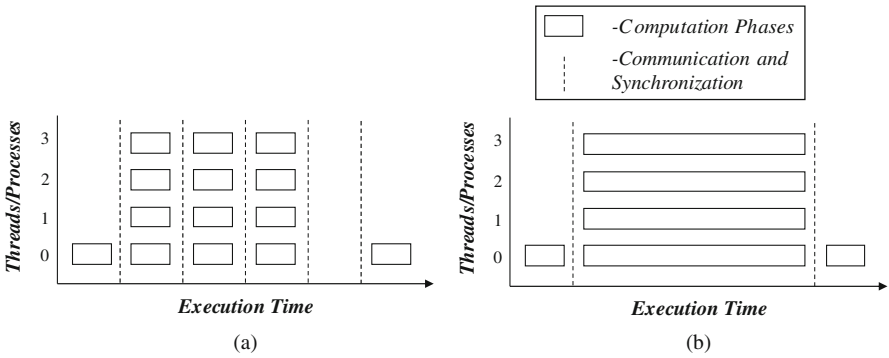

Fig. 3.1 Behavior of benchmarks. (**a**) High communication. (**b**) Low communication

the best results for the same benchmark set used in this study and, therefore, this scheduling mechanism is used here.

As indicated by [17, 36] and [38], we have used parallel tasks for the PThreads and MPI implementations. In such cases, the iterations of the loop were distributed based on the best workload balancing between threads/processes. Moreover, the communication between MPI processes was implemented by using nonblocking operations, to provide better performance, as showed in [44].

3.1.2 Multicore Architectures

3.1.2.1 General-Purpose Processors

Core2Quad The Intel Core2Quad is an implementation of the ×86-64 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 Fig. 3.2a 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 ×86-64 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 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 (Fig. 3.2a): 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.

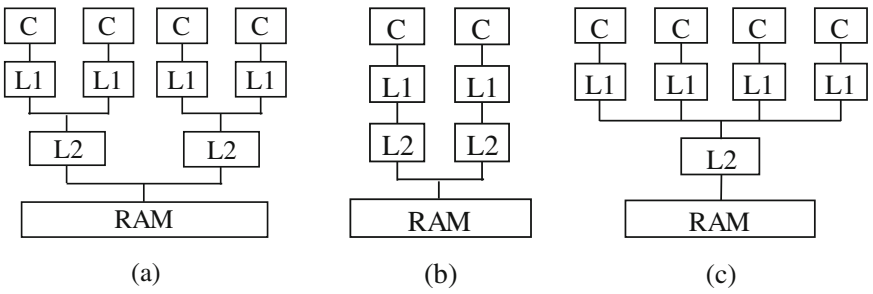

Fig. 3.2 Memory organization of each processor used in this study. (**a**) Intel Core2Quad and Xeon. (**b**) Intel Atom. (**c**) ARM Cortex-A9/A8

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

3.1.2.2 Embedded Processors

Atom The Intel Atom is also an ×86-64 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 ×86 processors is the backward compatibility with the ×86 instructions set, programs already compiled for these processors will run without changes on Atom.[2] The memory system is organized as illustrated in Fig. 3.2b: 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 as illustrated in Fig. 3.2c: 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.

3.1.3 Execution Environment

The Performance Application Programming Interface (PAPI) [14] 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 [13] (for the processors) and Cacti Tool (for the memory systems), as shown in Table 3.2. To estimate the total energy consumption (Et), we have taken into account

[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.

Table 3.2 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

the energy consumed for the executed instructions (E_{inst}), cache and main memory accesses (E_{mem}), and static energy (E_{static}), as given by Eq. (3.1).

$$Et = E_{\text{inst}} + E_{\text{mem}} + E_{\text{static}} \tag{3.1}$$

To find the energy consumed by the instructions, Eq. (3.2) was used, where I_{exe} is the number of executed instructions multiplied by the average energy spent by each one of them (E_{perinst}).

$$E_{\text{inst}} = I_{\text{exe}} \times E_{\text{perinst}} \tag{3.2}$$

The energy consumption for the memory system was obtained with Eq. (3.3), where $(L1DC_{\text{acc}} \times E_{L1DC})$ is the energy spent by accessing the L1 data cache memory; $(L1IC_{\text{acc}} \times E_{L1IC})$ is the same, but for the L1 instruction cache; $(L2_{\text{acc}} \times E_{L2})$ is for the L2 cache; and $(L2_{\text{miss}} \times E_{\text{main}})$ is the energy spent by the main memory accesses.

$$\begin{aligned} E_{\text{mem}} = {} & (L1DC_{\text{acc}} \times E_{L1DC}) + (L1IC_{\text{acc}} \times E_{L1IC}) + (L2_{\text{acc}} \times E_{L2}) \\ & + (L2_{\text{miss}} \times E_{\text{main}}) \end{aligned} \tag{3.3}$$

The static consumption of all components is given by Eq. (3.4). 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}).

$$E_{\text{static}} = \left(\frac{\#Cycles}{Freq} \right) \times (S_{CPU} + S_{L1DC} + S_{L1IC} + S_{L2} + S_{\text{MAIN}}) \tag{3.4}$$

3.1.4 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 3.1. 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 some observations regarding the results presented next:

- 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.

3.2 Results

3.2.1 *Performance and Energy Consumption*

Figures 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9 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 3.3 and 3.7 show raw numbers, where the *x*-axis of each chart is the energy consumption, and the *y*-axis is the execution time. Figures 3.4 and 3.8 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, Figs. 3.5 and 3.9 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 subsections, considering both classes of programs separately.

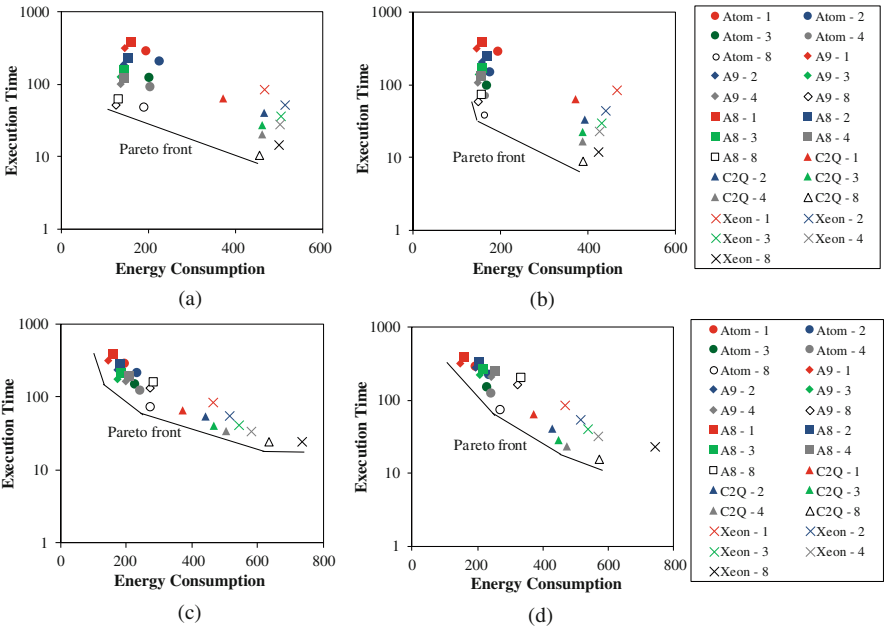

Fig. 3.3 Performance (seconds) and energy consumption (joules) results for high-communication programs. (**a**) OpenMP. (**b**) PThreads. (**c**) MPI-1. (**d**) MPI-2

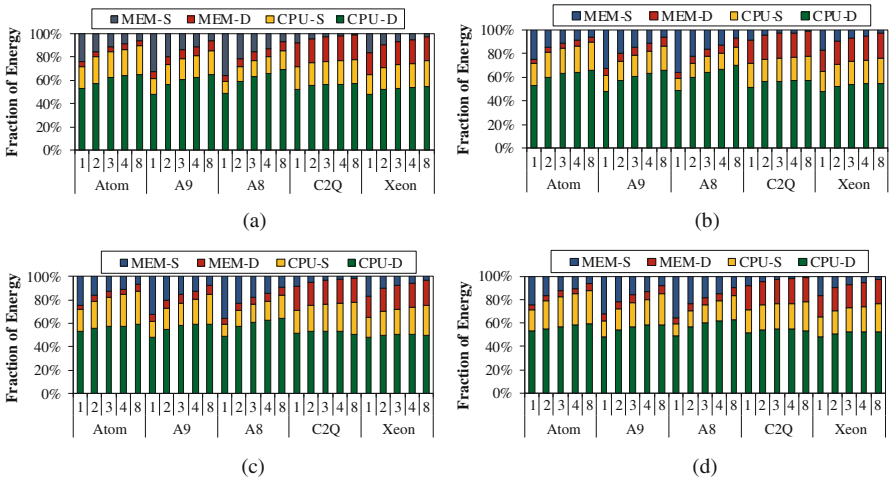

Fig. 3.4 Fraction of energy consumed by each hardware component (MEM: memory; CPU: processor; D: dynamic; S: static) for HC applications. (**a**) OpenMP. (**b**) PThreads. (**c**) MPI-1. (**d**) MPI-2

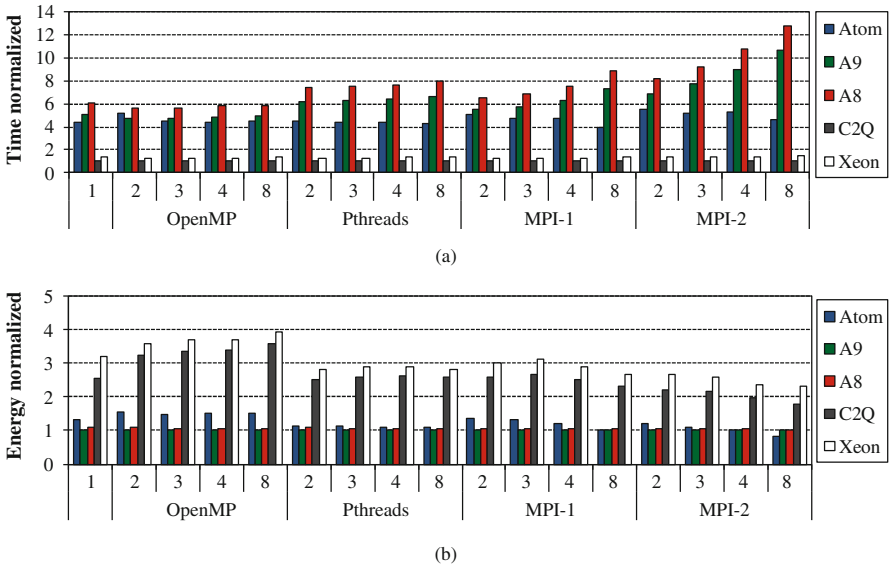

Fig. 3.5 Results normalized to Core2Quad (performance) and A9 (energy)—HC Programs. (**a**) Performance normalized to Core2Quad. (**b**) Energy consumption normalized to A9

3.2.1.1 High-Communication Programs

Figure 3.3 shows the performance and energy consumption for each processor running a different number of threads/processes. Each chart analyzes a different

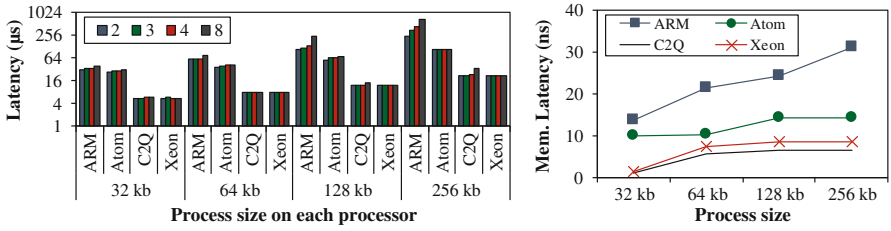

Fig. 3.6 Overhead to execute context switching on each processor. (**a**) Time to execute context switching. (**b**) Memory system latency for each process size during context switching

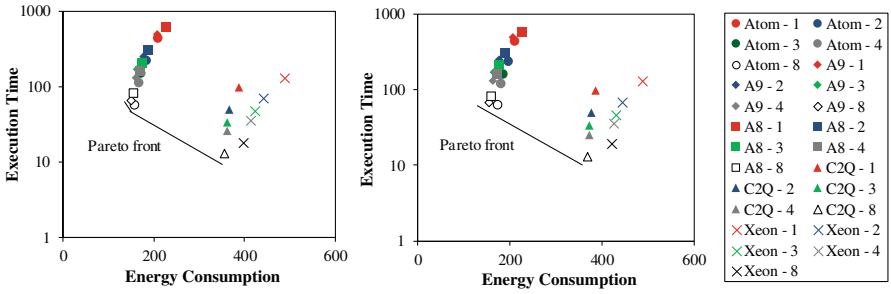

Fig. 3.7 Performance (seconds) and energy consumption (joules) results for low-communication programs. (**a**) Shared variables. (**b**) Message passing

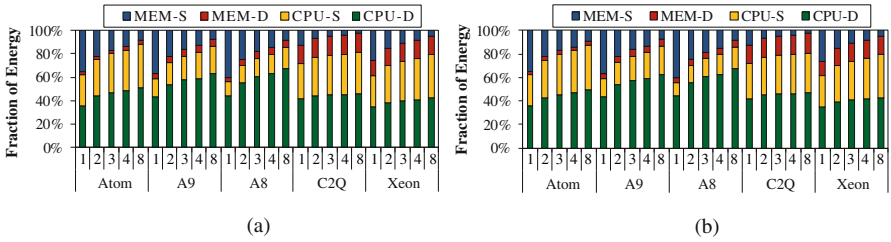

Fig. 3.8 Fraction of energy consumed by each component—LC applications (MEM: memory; CPU: processor; D: dynamic; S: static). (**a**) Shared variables. (**b**) Message passing

parallel programming interface. Considering the performance, regardless of 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.

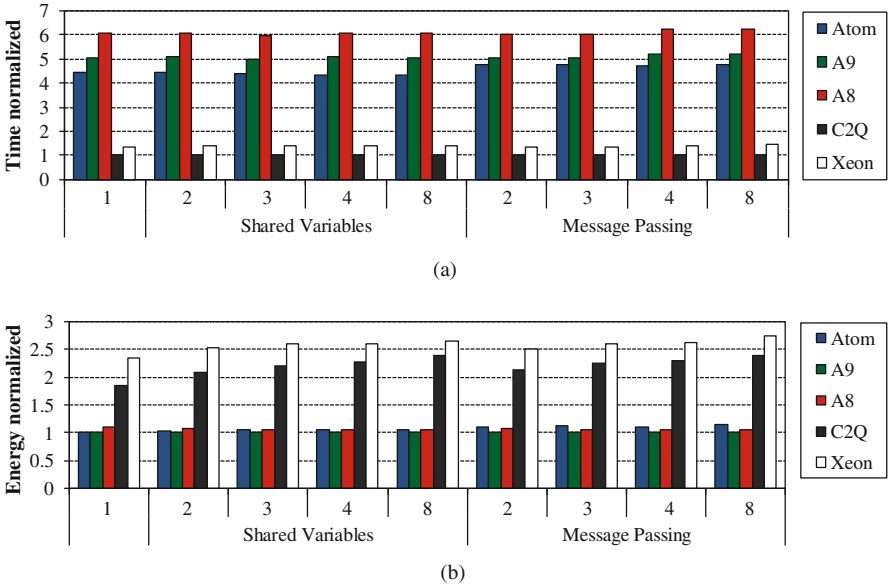

Fig. 3.9 Results normalized to Core2Quad (performance) and A9 (energy)—LC programs. (**a**) Performance normalized to Core2Quad. (**b**) Energy normalized to A9

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, 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 even when 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 (Fig. 3.3). 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 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 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 Fig. 3.3 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 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 (Fig. 3.4a), threads come into a busy-waiting state, accessing the shared memory repeatedly until the end of synchronization [22]. This synchronization mechanism does not incur significant performance overhead, so all processors have similar behavior as TLP exploitation increases (as can be seen in Fig. 3.5a, 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 than in the Intel ones (Fig. 3.5a). 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 (Fig. 3.3b), 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 (Fig. 3.5a). In order to understand this behavior, LMbench (a suite to measure system performance) [80] was used to measure the impact of context switching on each processor. Figure 3.6a 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 Fig. 3.6b. 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 (Fig. 3.5b). 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 (Fig. 3.5a), and influencing the energy consumption. In such cases, as more TLP is exploited, the energy difference between ARM and Intel decreases (Fig. 3.5b); and in the execution of 8 processes Atom got to a point where it consumed less energy than ARM processors. This scenario worsens when MPI-2 applications are executed (Fig. 3.3d), in which, as the number of processes increases, the performance gains are 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 [19].

3.2.1.2 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 (Fig. 3.7). 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 [117].

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, energy consumption decreases as TLP exploitation increases, regardless of the processor and 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 (Fig. 3.9b). 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 Fig. 3.7a 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 reduce the energy consumption maintaining similar execution time to the best possible, there is no satisfactory option available. For message passing (Fig. 3.7b), 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.

3.2.2 *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 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 [13] 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 3.10, 3.11, 3.12, and 3.13 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 3.10 shows the results of the sequential executions, while Figs. 3.11, 3.12, and 3.13 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

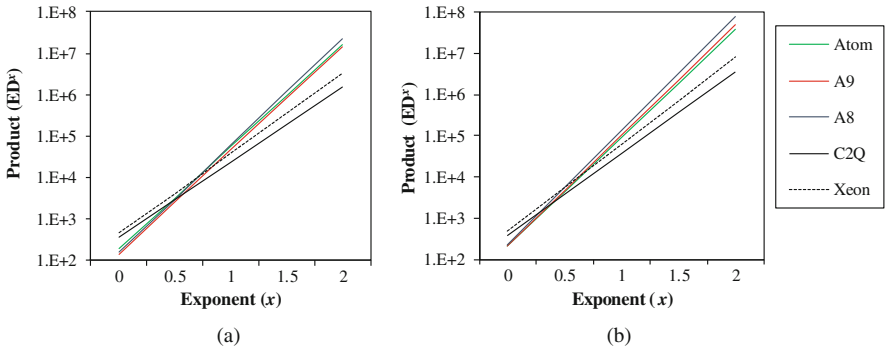

Fig. 3.10 Impact of exponent, x, on product ED^x—sequential execution. (**a**) High communication. (**b**) Low communication

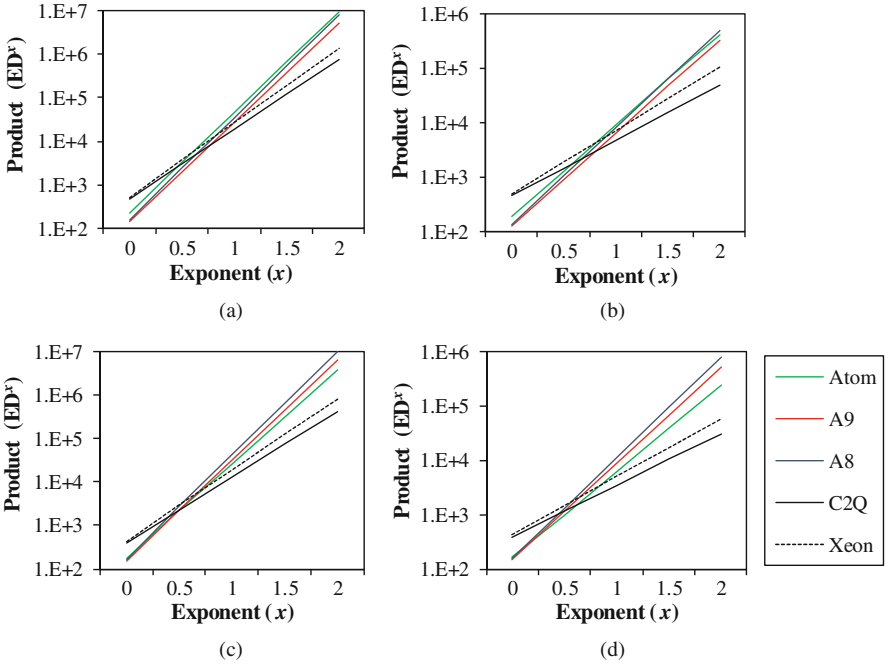

Fig. 3.11 Impact of exponent, x, on product ED^x of HC programs implemented with shared variables. (**a**) OpenMP—2 threads. (**b**) OpenMP—8 threads. (**c**) PThreads—2 threads. (**d**) PThreads—8 threads

are more focused on performance rather than energy, corroborating the authors' research in [13].

Let us discuss the results for the sequential versions (Fig. 3.10). For HC programs (Fig. 3.10a), 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 (Fig. 3.10b), 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 (Figs. 3.11, 3.12, and 3.13), in all cases they achieved better ED^xP than their sequential counterparts, regardless of 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 (Fig. 3.11a and b), Cortex-A9 has better ED^xP than the other

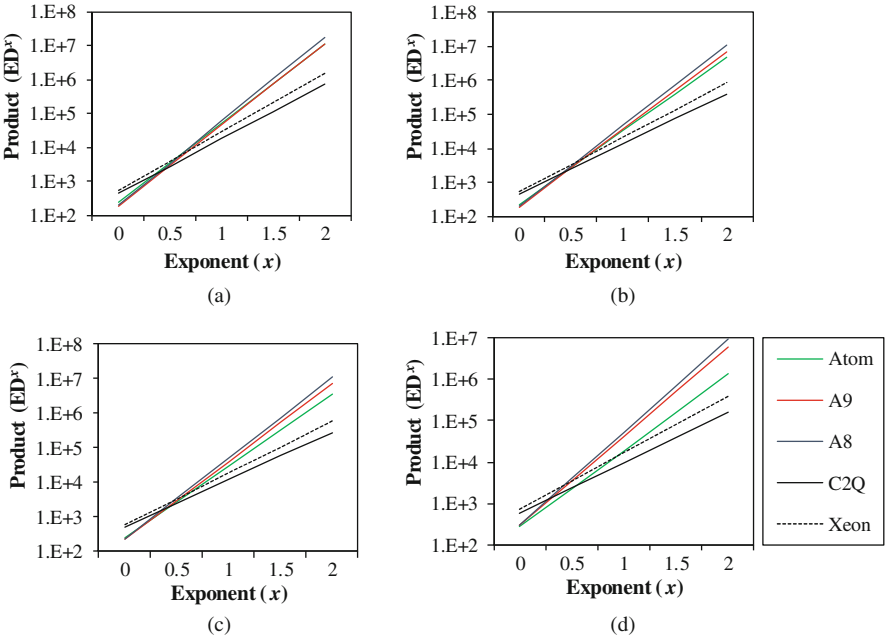

Fig. 3.12 Impact of exponent, x, on product ED^x of HC programs implemented with message passing. (**a**) MPI-1—2 processes. (**b**) MPI-1—8 processes. (**c**) MPI-2—2 processes. (**d**) MPI-2—8 processes

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 3.3). For PThreads implementations, the behavior is different (Fig. 3.11c and 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 3.12 shows the results when HC programs are implemented with message passing. Let us first discuss the MPI-1 results, where the GPPs outperform 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 been discussed in Sect. 3.2.1: 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 (Fig. 3.13), Core2Quad continues offering the best ED^xP in most cases (mainly when performance and energy have the

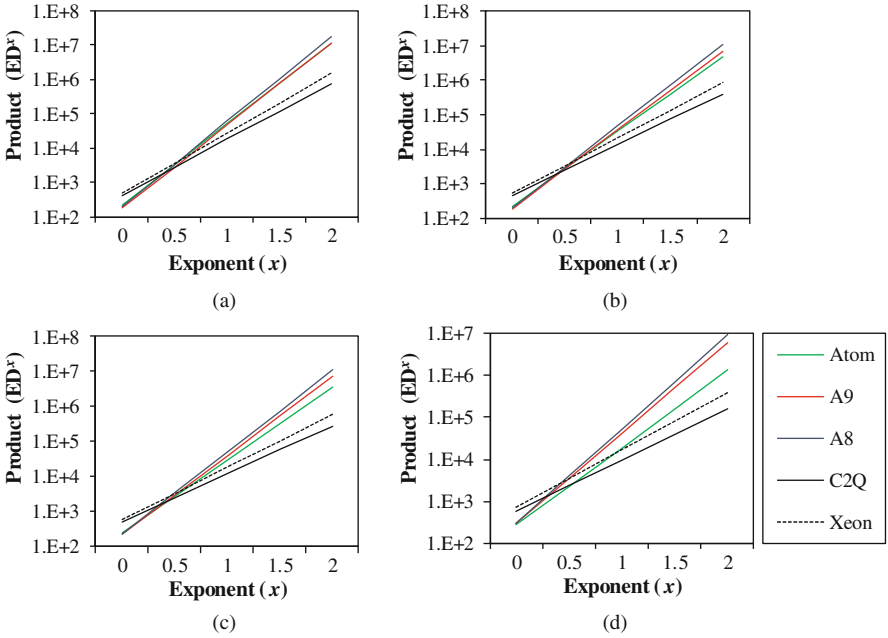

Fig. 3.13 Impact of exponent, x, on product ED^x of LC programs. (**a**) Shared variables—2 threads. (**b**) Shared variables—8 threads. (**c**) Message passing—2 processes. (**d**) Message passing—8 processes

same weight). Comparing only the embedded processors: when they communicate through 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 again 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 3.3 shows the intersection points to figure out which is the best processor in between the intervals of x considering the charts of Figs. 3.10, 3.11, 3.12, and 3.13. 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.

Table 3.3 Intervals of x where each processor is better on the ED^xP, when energy is the most important

			Embedded processors			GPPs	
		TLP	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	OMP	2	–	0.0–0.77	–	> 0.7	–
Shared variables		8	–	0.0–0.81	–	> 0.81	–
Figure	PT	2	0.36–0.55	0.0–0.36	–	> 0.55	–
		8	0.19–0.61	0.0–0.19	–	> 0.61	–
HC	MPI-1	2	–	0.0–0.56	–	> 0.56	–
Message passing		8	0.0–0.61	–	–	> 0.61	–
Figure	MPI-2	2	–	0.0–0.42	–	> 0.42	–
		8	0.0–0.49	–	–	> 0.49	–
LC	SV	2	0.37–0.48	–	–	> 0.49	–
Figure		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	–

3.2.3 The Influence of the Static Power Consumption

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 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 Sect. 2.2, 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. As an example, although increasing doping level allows higher on current for faster transitions, it also causes more considerable leakage. Therefore, companies can tune the circuits during the manufacturing process to be faster and consume more static power or vice versa [83]. In some cases, the static power in the processor may represent up to 40% of the total energy consumption [35, 60, 83].

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 over the processors will take longer to execute than others.

Table 3.4 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

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 [13] and CACTI 5.1[4]: 10, 20, 30, and 40%. Table 3.4 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 3.14 and 3.15 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 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

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

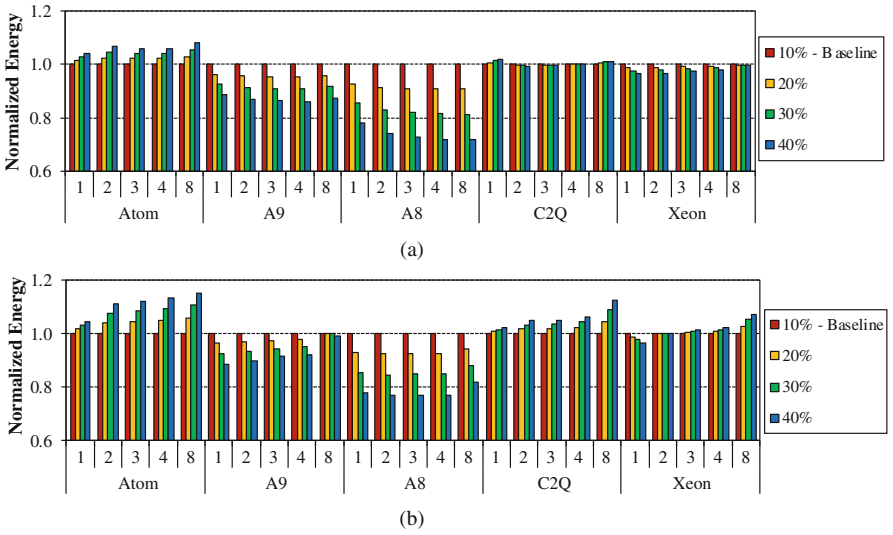

Fig. 3.14 Impact on the total energy consumption when the static power of processor varies from 10%—HC Programs. (**a**) Shared variables. (**b**) Message passing

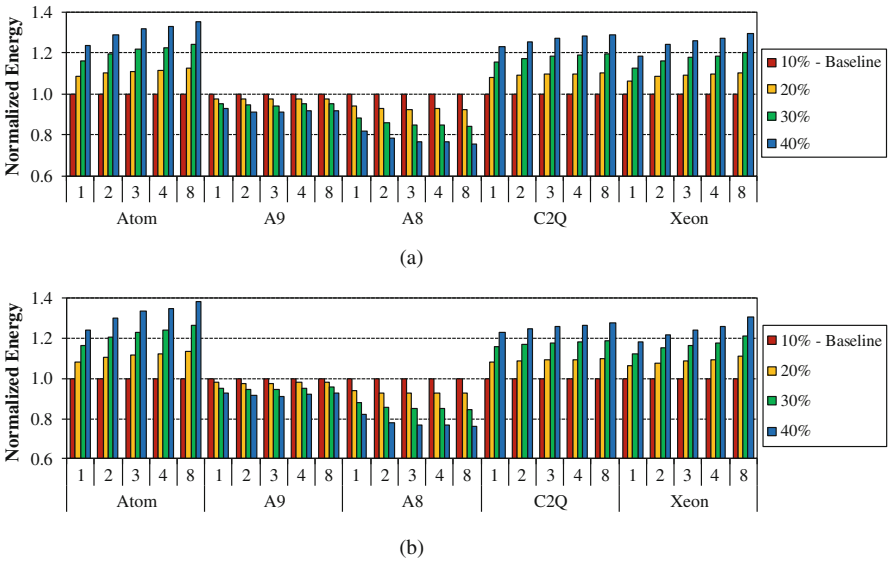

Fig. 3.15 Impact on the total energy consumption when the static power of processor varies from 10%—LC programs. (**a**) Shared variables. (**b**) Message passing

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.

Table 3.5 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

Table 3.6 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

Let us first discuss the results of the Intel processors executing HC programs (Fig. 3.14). In such cases, the effect of changing the proportion of static power is negligible in most cases. To better understand that, let us consider Tables 3.5 and 3.6. The former presents the number of executed instructions by core per second. To compare only the behavior of each PPI on each processor, Table 3.6 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 3.5) because of the kind of application, their parallel versions proportionally 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 Tables 3.5 and 3.6: 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 (Fig. 3.14), 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 (Fig. 3.15), 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 of 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 changes from 10 to 40%, the total energy consumption increases by almost 24% on both Atom and Core2Quad, and 18% in the Xeon processor. As for the execution with eight threads/processes, this energy difference is even higher: 35 and 38% for shared variables and message passing, respectively, in the Atom processor; and 28 and 30% in the Core2Quad and Xeon, respectively, regardless of the communication model. As for ARM processors, which have a high number of executed instructions per second (see Table 3.5), 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, 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.

3.3 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 (Sect. 3.2.1), 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 as 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 it 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 Sect. 3.2.2), 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 Sect. 3.2.3, 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.