Understanding the Interactions Hardware/Software Parameters on the Energy Consumption of Multi-Threaded Applications

Jeyarajan Thiyagalingam, Anne E. Trefethen

April 29, 2014

Abstract

In recent years, the energy consumption of computing systems has become a major concern. Driven by cost, environmental concerns and policy, minimising the energy footprint of computing systems is one of the primary goals of many initiatives. The approaches taken to reduce energy consumption have largely been by design, building systems from low-energy components, or through operational and environmental adaptations such as powering down idle systems or using natural cooling mechanisms.

An alternative route is to make energy savings from software, ideally with little or no change in runtime performance. Although the awareness on the importance of effective software implementations on runtime and energy performance is considerably widespread, the knowledge on the parameter space affecting the energy efficiency is limited.

In this paper, we investigate the impact of a number of parameters on runtime and energy performance, especially in the context of multi-threaded applications. In particular, we consider the influence of system and software parameters, such as core frequency, number of threads and types of applications. We find that with the right combination of tuned parameters, it is possible to secure significant savings on energy consumption while retaining / achieving acceptable runtime performance.

1 Introduction

Optimising software for their energy consumption or power usage is not a new challenge. Detailed literature can be found in several domains including embedded computing and VLSI systems design, for example [1, 2]. However, much of these research have been focused on the architectural design using low-power and low-frequency components. Hardware manufacturers, in particular processor manufacturers, have been trying to seek better power/energy regimes through several techniques, such as lowering the voltage or frequency. Such approaches have successfully driven processor design towards the low-power regimes. However, the laws of physics prevent this from being applied endlessly. For example, lowering

the voltage below a certain threshold starts to cause undesirable behaviours at the semi-conductor level [3, 4].

As a result, alternative or complementary approaches for minimising the energy consumption are actively sought after. One such approach is to explore energy-efficient software. The fact that the overall energy consumption of a computer system is made up of the energy consumption of several components, such as CPUs, accelerator cards (GPUs and the like), memory subsystems, disk drives and other components, allows us to optimise the balance of the contributions from each, to minimise the overall energy consumption. For example, different data structures may have different data transfer behaviours and thus different energy consumption characteristics [5, 6, 7]. The variation of energy consumption is highly dependent on the architecture, number of cores, amount of data moved around and thus on the re-use of the data. As we consider options for reducing energy consumption, we need to ensure that we do not impact the performance of the software significantly.

Parallelisation through multithreading is a viable technique for creating high performance software. It is seldom the case that compilers can expose all available concurrency to the underlying system. As a result, architecture agnostic programming models such as OpenMP or PThreads are used to aid the compiler in identifying dependencies and parallelising regions of interest. However, the determining the exact degree of threading to be applied for better performance is often difficult. This is often exercised by manual tuning or by simply relying on the underlying operating system or by resorting to maximum degree of threading. Having given the direction through these mechanisms, the programmer has little control over what happens during the execution. With all these techniques in place, the programmer will then hope that threading is taken care of and thus the application is faster. Energy efficiency or consumption, however, is not a concern of contemporary compilers and thus no insights are gained. To add to the problem, although there are sufficient number of performance tuning tools, there are no software tools for analysing the energy efficiency of multithreaded applications nor to fully understand the interaction of different parameters, such as with core frequencies, number of threads and compilers, on the applications' runtime and energy performance.

In this paper, we investigate the impact of multiple parameters on the energy and runtime performance of multithreaded software: CPU's clock frequency, thread-count and compilers. By relying on a simple yet effective energy model, we establish a metric for quantifying the energy efficiency of multithreaded applications. Using this metric, we evaluate the energy efficiency of a standard multithreaded computational benchmark suite, namely the OpenMP segment of NAS Parallel Benchmarks [8], on a range of architectures. Through an exhaustive evaluation, we are able to answer many questions relating to energy efficiency and multithreading. This paper makes the following contributions:

- We propose a metric for establishing the energy efficiency of multithreaded applications taking in to account both energy and performance measures
- Using this metric, we exhaustively evaluate a multithreaded benchmark suite

for multiple parameter interactions on different architectures

- We show how individual parameters, such as compilers and simultaneous multithreading, influence the overall energy efficiency of software.
- Through the evaluation we are able to identify a dependency on the degree of threading that allows optimisation of both energy consumption and performance in non obvious ways. For example, in one of the cases, we were able to make 91% energy saving along with a performance gain of 180% from the standard case.

The rest of the paper is organised as follows: In Section 2, we formulate a metric for quantifying the energy efficiency of a multithreaded software. Section 3 outlines our evaluation procedure, experimental setup and the benchmarks used in the study. We present and discuss the results in Section 4. In Section 5 we discuss the related work in connection to our work followed by our conclusions and directions for further work in Section 6.

2 A Metric for Assessing the Energy Efficiency of Multithreaded Software

The Energy-Delay product (ED product) introduced by Gonzalez and Horowitz [9] was advocated as a metric for correlating energy and performance of alternate architectural design choices. Although our focus is entirely on the software-end, we derive our metric based on the same ED-product but with the special emphasis on tunable baselines. Let T_1, P_1, and E_1 denote the execution time, instantaneous power values and energy consumption of an application with a single thread (sequential case). Similarly, let T_n, P_n, and E_n denote the same for multiple, n, threads. Figure 1 shows the instantaneous power variations of three different threading cases, when $n = 1, 2, 32$. With reference to this figure, multithreaded code performs faster than the sequential case and hence

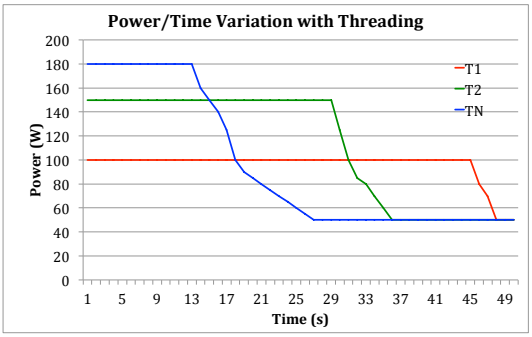


Figure 1: Power-Time Variation of Multiple Thread Configurations.

$$T_1 = \alpha T_n \qquad \text{for} \qquad \alpha > 1$$

where α is the speed up of the application. As more and more cores become active with additional threads, the runtimes will become shorter but the instantaneous power values will increase. With this, let

$$P_n = \gamma P_1 \qquad \text{for} \qquad \gamma > 1$$

However, since we are interested in overall energy consumption,

$$E_n = \beta E_1 \qquad \text{for} \qquad \beta \lessgtr 1$$

Ideally, when multithreading, energy scaling should not outweigh the speedup and hence it is ideal if,

$$\alpha \gg \beta$$

and we define our metric for energy efficiency based on the ratio of α and β as follows:

$$\Omega = \frac{\alpha}{\beta}$$

For example, $\Omega = 1$ indicates both energy and runtime speedups are scaling at the same pace; $\Omega = 2$, indicates that the speedup is twice the energy scaling; and $\Omega = 0.5$ indicates that energy scaling is twice that of runtime speedups. With this, higher the value of Ω better the energy efficiency it is. However, a baseline b is often required against which the energy efficiency can be established. In some instances it makes sense to consider the sequential case (single thread case) as the baseline ($b = 1$). However, as discussed in Section 1, with the prevalence of multicore, multiprocessor systems, it is the norm that applications are always fully threaded, matching the number of cores in the system. Hence it may be more suitable to consider this as a baseline case ($b = n_c$, where n_c is the number of cores). With this, for a given core frequency f, we define the metric for energy efficiency for an application with p threads as follows:

$$\Omega_{p,f} = \frac{\alpha_{p,f}}{\beta_{p,f}} \tag{1}$$

where

$$\alpha_{p,f} = \frac{T_{b,f}}{T_{p,f}}, and$$

and

$$\beta_{p,f} = \frac{E_{p,f}}{E_{b,f}}$$

where $E_{b,f}$ and $T_{b,f}$ are the baseline energy and runtime values at frequency f, with which we would like to compare the performance or energy scaling. The Equation (1) attempts to compare the performance improvement to the variation

in energy from the baseline. For a given frequency, the multithreaded application with the highest Ω value is the most energy efficient variant. Furthermore, given a multithreaded code, for a given frequency f, the optimal number of threads $n_o = p_f$ is such that:

$$p_f = \arg\max_n \Omega_{n,f} \tag{2}$$

However, if the frequency can be varied, the optimal energy efficiency is

$$\Omega_p = \frac{\alpha_{p,f}}{\beta_{p,f}} \tag{3}$$

The corresponding expression for an optimal number of threads is :

$$p = \arg\max_n \Omega_n \tag{4}$$

3 Experimental Evaluation

To understand the impact of multi-parameter interactions on the energy consumption of applications, we carried out a set of detailed experiments. The experiments used a suite of benchmarks, on a number of platforms. Below, we describe the procedure, experimental platform and the suite of benchmarks we used for the evaluation.

3.1 Estimating Energy Consumption

Estimating the energy consumption of an application requires the knowledge of the power profile. The power profile of a piece of software is given by the variation of power of the system over time, $P(t)$. Given that the idle power of the system P_I is invariant and perturbations on the system are negligible, the energy consumption of an application run between two time points T_1 and T_2 can be estimated as follows:

$$E \approx \int_{T_1}^{T_2} P(t)dt - P_I(T_2 - T_1) \tag{5}$$

In order to estimate the energy consumption, in our experimental setup, we placed a number of sensors between the power source and the system. Figure 2 shows a sample configuration.

With such a configuration, we monitored the current, voltage and power by recording at one-second interval. To estimate the energy consumption, we took measurements of power usage and integrated it across the period of the software run. Prior to launching an application, we triggered the sensors to record the line conditions to a central data collection server and collect the meta-data that describes the experiment. In the case where a system consists of more than one power-supply, we used multiple sensors and performed data fusion on the two sets of readings. This type of configuration with external monitors is advantageous as

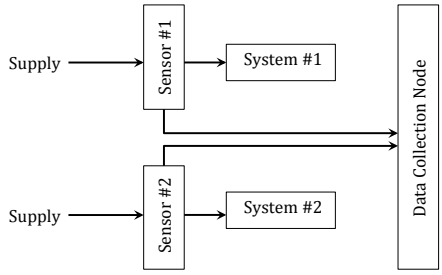


Figure 2: A simple configuration for monitoring the energy consumption of applications.

it is non-intrusive and therefore, the system itself need not be adapted. Although, a rather fine-grained and detailed profiling can be performed at the component level such as one outlined in [10], we will not be using this technique in this paper.

3.2 Experimental Platforms

We used three different hardware platforms for our study. Each of the systems was equipped with dual processors, each with multiple cores. The details of the systems on which we performed our experiments are shown in Table 1.

Table 1: Details of the systems on which the experiments were conducted.

	dori	**zen**	**skyray**
Monitoring Level	Internal and External	External	External
Processor / Model	AMD Opteron 265	Intel Xeon HarperTown 5650	AMD Opteron Magny Cours (6128)
Operating System	Linux 2.6.35	Linux 2.6.35	Linux 2.6.38
Clock Speeds	1.0-1.8 GHZ	1.6-2.8GHz	0.8-2.0GHz
Common Memory	12GB	24GB	5 GB
Number of Processors	2	2	2
Cores per Processor Total Cores	2 4	6 12 (24 on Hyper-Threading)	8 16
L1 Cache	64KB	64KB	64KB
L2 Cache	2×1MB (1MB per core)	12 MB (per processor)	4MB (per processor)
L3 Cache	-	-	12MB
Compilers	GNU C/Fortran 4.2	GNU C/Fortran 4.2	GNU C/Fortran 4.3
	Intel C++/Fortran XE 12.1	Intel C++/Fortran XE 11.1	Intel C++/Fortran XE 12.0
Common GNU Flags	-O3 -fopenmp -lgomp -mfused-madd -fPIC		
Special GNU Flags	-march=athlon64	-march=native	-march=native
Common Intel Flags	-O3 -openmp -liomp5 -mcmodel=medium -shared-intel		
Special Intel Flags	-xSSE2	-xSSE2	-xSSE4.2

3.3 Benchmarks

For benchmarking, we used the OpenMP segment of the NAS Parallel Benchmark suite (version 3.3.1) [8]. We used only a sub-set of these benchmarks for our evaluation, chosen with different memory access characteristics, different compute-to-communication ratios and sufficiently long but feasible runtimes. We provide a summary of these benchmarks and the problem sizes specific to the classes that we have chosen to evaluate in Table 2 . Although it is possible to evaluate all classes, we have chosen the classes so as to exceed the resolution for measurement and to meet the capacity of the systems under test.

Table 2: Details of the benchmarks used in the paper.

Benchmark	**Description**	Classes (Problem Size)
DC	Data Cube Operator	A (10^6,B (10^7)
LU	LU Gauss-Seidel PDE solver	B (102^2),C (162^3)
SP	Scalar Pentadiagonal PDE solver	B(102^3),C(162^3)
UA	Unstructured Adaptive	B(7-Levels),C(8-Levels)

3.4 Experimental Procedure

We have chosen an exhaustive experimental process taking the following into consideration:

1. For a given platform, we performed multiple runs, covering different numbers of threads, all classes, all benchmarks, all compiler variants and all frequencies. In total this involves a minimum of 1560 runs for the Zen platform, 720 runs for the Skyray platform and 192 runs for the Dori platform. To statistically improve the confidence level of the results, we repeated each of these runs at least 20 times, performed the Q-Test on the samples for eliminating any outliers and reported the median of the measurements. For each run, we ensured that the NAS parallel benchmark verification was successful.

2. For each benchmark, we logged the current, voltage and power at regular intervals and stored to a database along with the runtime. The logging is completely decoupled from the hosts where execution takes place and therefore the logging or monitoring has no influence on the performance or energy consumption of the experiments.

3. The whole process was automated with the help of code generators and launching scripts, to ensure appropriate meta-data about the experiment was captured including platform, experiment, class of the benchmark, clock frequency of the CPU, and the compiler settings.

4. To ensure that energy consumption was estimated properly, we imposed a mandatory pre- and post-delay before and after the execution of the actual binary while measuring the line conditions. We then integrated the logged power readings to estimate the energy consumption using the Simpson/Trapezoidal rule.

5. During each run, we varied the number of threads from 1 through to $2\times$ the available number of cores, wherever possible. This gave us an opportunity to understand not only the impact of threading, but also the impact of over-threading. We also varied the frequency, covering most values supported by the given platform. Under this case, we also studied the impact of hyper-threading on energy, which is primarily supported by the Intel-based platforms.

6. In all cases, we ensured that the turbo-mode (or the on-demand mode), which automatically varies the core frequencies upon load on the system, is turned off. Throughout our experiments, the core voltages remained fixed.

4 Results and Analysis

4.1 Analysis of Results

The full set of results from the evaluation of the benchmarks are provided in the supplementary material. In order to provide insight into the results, we focus here on one benchmark, namely SP, for which we provide a full analysis.

We have provided a summary of the parameter space for each of the benchmarks using two different baselines: non-threaded case (i.e. thread-count is one T_1) and fully-threaded case (T_{max}), both at the maximum core frequency. This is acceptable as the core frequency is rarely varied in systems.

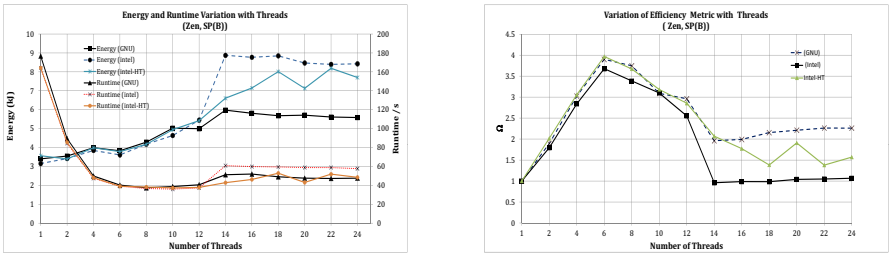
Initially, we varied the parameter space (thread-count, compilers, hyper-threading) with the core frequency being fixed. The results for this initial set of experiments are presented in Section 4.2. We used these results to prune the parameter space and to focus on a given compiler and platform.

For each platform and for each benchmark, we investigated the effects of three different configurations against the two baselines mentioned above. These are outlined and discussed in Section 4.3.

4.2 Example Analysis

Figure 3 shows the effect of multi-threading on the SP benchmark on the Zen platform. The results cover the impact of threading both on energy and on runtime performance. The illustrations are dual-axis graphs, where the left Y axis corresponds to energy consumption (in KiloJoules) and the right Y axis corresponds to runtimes. The variations are given against the number of threads. The results in

Figure 3 include both the hyper-threaded and non-hyper-threaded variants and all compilers.



(a) (b)

Figure 3: The Variation of Energy, Runtime and the Efficiency Metric Ω with the Number of Threads on the Zen platform. Figure 3(a) shows the variation of energy and runtimes against the number of threads. In Figure 3(b) we show the corresponding variations of the Ω values. Benchmark: SP, Class: B, Frequency Setting: 2668MHz, SMT: Enabled

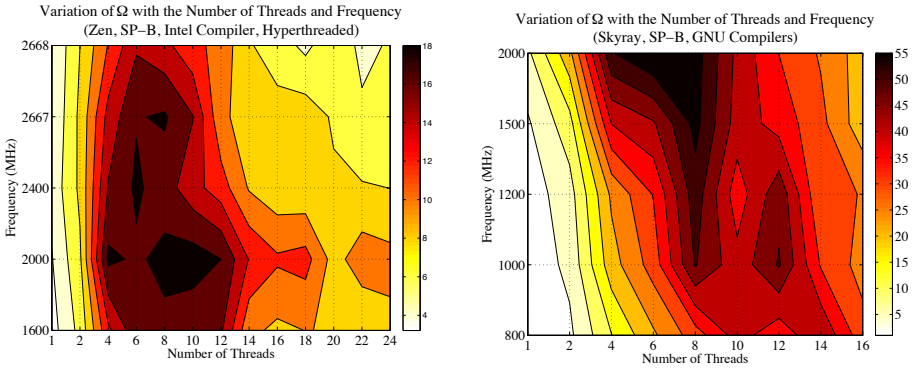
As can be observed from the figure, the overall energy consumption increases with the number of threads while the corresponding runtimes decreases. Energy consumption for the fully threaded case is three times that of the single-threaded case with the speed up of $3.3\times$ (or 330%). The best speedup for class B of SP is achieved when using eight threads for the binary produced by the Intel compiler. Although increasing the thread count beyond this point has only increased the computational time by a small amount, the energy consumption increases very significantly. Increased thread-count has multiple side effects: increased data movements, additional thread management overheads and increased wait-states. These increase the runtimes. Increased runtimes and data movements increases the overall energy consumption.

Figure 3(b) shows the variation of energy efficiency metric Ω ($\Omega_{p,f}$) against the number of threads, where the values of Ω, directly represents relative energy efficiency. We find that the Intel compiler-based binary are more energy efficient compared to the non-hyper-threaded version produced by the same compiler. The relative energy efficiency of binaries produced by the GNU and Intel compilers are almost the same and only vary within 10% of their values upto 14-threads. When the thread count is increased beyond 14, their energy efficiencies vary by a large value as can be observed in Figure 3(b).

One of the features of the Intel-based Zen platform is the ability to perform simultaneous multi-threading, or hyper-threading. On the Zen platform, the ratio of virtual cores to physical cores is 2:1. With this, we would expect instruction streams from two threads to be executed concurrently in a single core leading to

better runtimes and thus energy consumption. This is clearly visible in the plots: in the absence of hyper-threading, the energy efficiency drops rather dramatically from 2.5 to 1.0 when the thread count exceeds the size of physical cores. Since the SP benchmark is considerably computationally bound, it benefited from the capability of hyperthreading, where two streams of instructions can be processed concurrently, and its average energy consumption is reduced by 50%.

We show how Ω varies when both the thread count and frequency are changed for the class B of the SP benchmark on two different platforms (Zen and Skyray) in Figure 4. The plots in the figure illustrates a number of maxima for Ω, whose positions indicate an appropriate thread-count/frequency configuration. For the Zen platform, the results are presented for the hyper-threaded case using the binary produced by the Intel compiler. For the Skyray platform we use the binary produced by the GNU compiler.



(a) (b)

Figure 4: The Variation of Energy Efficiency, Ω, against the Number of Threads and Frequency on the Zen (left) and Skyray (right) platforms with Hyper-threading Enabled. Benchmark: SP, Class: B, Frequency Settings: Zen - 1600MHz-2668MHz, SMT: Enabled, Skyray - 800-2000MHz

The intensity values of the contour plots signify the value of Ω as the number of threads and the operating frequency of the cores are varied. Darker regions signify higher values of Ω and lighter regions represents low values. Since the frequency stepping values are discrete, only the values at the grid-points are considered to be valid.

For the SP (class B) benchmark on the Zen platform (Figure 4(a)), we found several combinations that give improved energy efficiencies ($17 < \Omega < 18$). Some of the (thread-count, frequency) pairs are : (4,2000), (6,2400),(8,2000), (8,2667) and (10,2000). Among these, the (10,2000) pair yielded the maximum Ω value. There are several choices which can be explored. One possibility is that fixing the frequency at the maximum (2668MHz) and varying the thread count to 6 can lead to better speedups ($4.2\times$ w.r.t the sequential case) and significant energy savings (about 110% w.r.t the sequential case). Another option is vary both thread count

and frequency (8, 2000) which can lead to even more energy savings (125%) and additional speedups ($4.3\times$) both in reference to the the sequential case.

On the Skyray platform (Figure 4(b)), although scaling the threads provided significant improvements from the sequential case, frequency scaling did not provide any additional improvements. However, using six threads is sufficient to secure maximum speedups and maximum energy efficiency ($\Omega = 6.7$). This configuration (six threads at maximum frequency) saved 34% of energy consumption while providing $4.4\times$ speedup compared to the sequential case.

The supplementary materials contain the full set of results for all the benchmarks, the platforms, the different frequency settings and compiler binaries.

4.3 Summary of Savings/Gains

To summarise the interaction of the parameter space, we provide a summary of each of the benchmarks. As mentioned above, we analyse the results using three different configurations outlined below in Table 3:

Table 3: Configurations Used to Understand the Interaction.

$T_{max} f_{max}$	Frequency is at maximum (f_{max}) Thread-count is at maximum (T_{max}).
$T_{opt} f_{max}$	Frequency is at maximum (f_{max}), thread-count is varied to find the optimal number of threads
$T_{opt} f_{opt}$	Both frequency and the number of threads are varied.

The results of each of these configurations are compared to two baseline cases: $T_1 f_{max}$ and $T_{max} f_{max}$. For each platform and benchmark combination, we compare the energy consumption, runtime performance and energy efficiency against these two baseline cases.
For each benchmark , we list the optimal frequency and the number of threads for which the maximum Ω value was obtained. For clarity, the parameter space variation is provided in a table and consumption and performance speedups are illustrated in charts.

For our example (class B of the SP benchmark) we show how the energy efficiency varies across the parameter space in Table 4 and energy/performance savings/speedups in Figure 5.

Table 4: Summary of the effect of varying the parameter set on the Zen platform for the SP Benchmark for class B. Please refer to Figure 5 and Figure 6 for the Savings/Improvements in Energy/Performance.

	Optimisation		
Values	None	T	T,f
Frequency / MHz	2668	2668	2000
Threads	24	6	10
Ω	1.6	4.0	19.0

The first column of the Table 4 contains the values for default configuration (maximum frequency and maximum thread count) and the resulting Ω value ($T_{max}f_{max}$ configuration). The second column includes the same when only thread count is varied while keeping the frequency fixed ($T_{opt}f_{max}$ configuration). It can be seen that reducing the thread count to six, results in improved energy efficiency. This is further improved when optimising over both frequency and thread-count, as shown in the third column ($T_{opt}f_{opt}$ configuration). This clearly shows that optimising against the frequency-thread space is more beneficial than optimising against one of them.

We first compare the energy savings and speedups of the three different configurations outlined in Table 3 against against the non-threaded sequential code (denoted by T_1). Figure 5, has two different sub-plots illustrating the cases given in Table 3. In the left, the energy savings of the configurations are measured against the case of T_1. As can be observed from the graph, the default configuration (where maximum threading is performed with maximum core frequency) has the greatest loss in energy, as much as 120%. Varying thread-count minimises this loss but still without any savings. Finally, varying both the frequency and thread-count improves this, resulting in noticeable savings. In terms of performance (illustrated by the subplot on the right), we see that using all threads gives only 32% of speedup and optimising the thread count leads to better performance gains (41%). Adjusting the frequency and thread-count leads to additional benefits (42%).

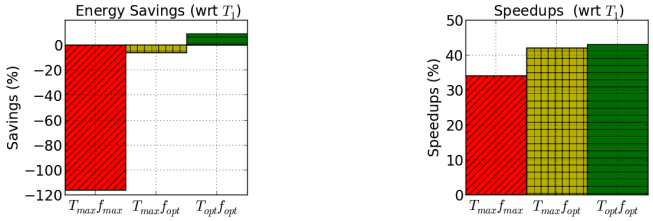


Figure 5: Summary of The Effect of Multi-Threading on Energy Savings and Performance Improvements on the Zen platform for the SP Benchmark for class B. See Table 5 for the Effect on Energy Efficiency and Parameter Space Variation.

As mentioned before, it is arguable whether the results should be compared against non-threaded case or against fully-threaded case. For the reasons discussed in Section 2, we present the relative energy/runtime performance gains against the second baseline, fully threaded, case ($T_{max}f_{max}$) in Figure 6. With this case, we see that varying the thread-count offers almost 50% energy savings and improves even further when both frequency and thread counts are varied. The similar observation is made for the runtime performance as well, where optimising both thread-count and frequency resulting in maximum runtime performance.

This observation is maintained for the energy consumption of the Class C of the same benchmark, the speedups do not improve consistently. We show the results

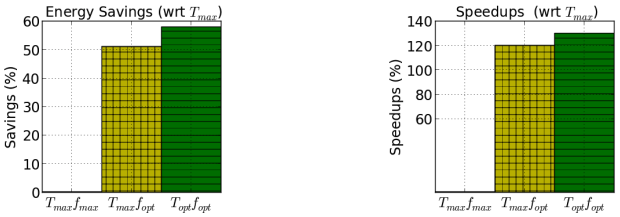


Figure 6: Summary of The Effect of Multi-Threading on Energy Savings and Performance Improvements on the Zen platform for the SP Benchmark for class B. See Table 5 for the Effect on Energy Efficiency and Parameter Space Variation.

for the Class C below in Table 5 and in Figure 7.

Table 5: Summary of The Effect of Varying the Parameter Space on the Zen platform for the SP Benchmark for class C. Please refer to Figure 7 for the Savings/Improvements in Energy/Performance.

	Optimisation		
Values	None	T	T,f
Frequency / MHz	2668	2668	2400
Threads	24	4	4
Ω	1.3	3.6	16.7

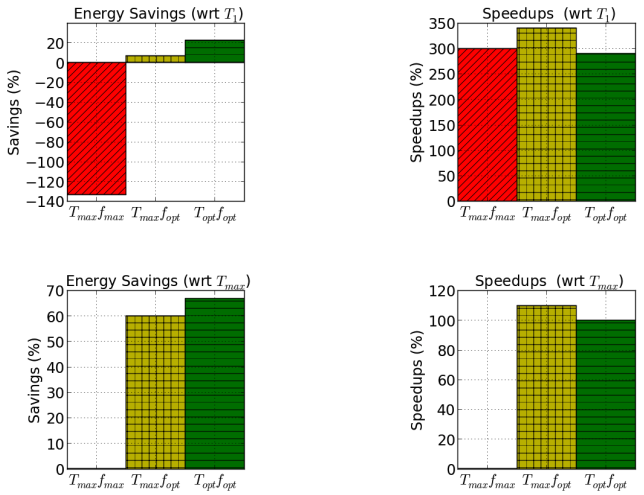


Figure 7: Summary of The Effect of Multi-Threading on Energy Savings and Performance Improvements on the Zen platform for the SP Benchmark for class C. Please refer to Table 5 for the Effect on Energy Efficiency and Parameter Space Variation.

4.4 Summary of Observations

Repeating a similar analysis on each of the platforms for each of the benchmarks yields the following observations:

4.4.1 Zen Platform

Some of the key findings on the Zen platform are:

- Although fully threading an application (without altering the core frequency from a selected value) improves the overall energy efficiency (relative to single-threaded case), they did not achieve the best possible energy efficiency. It was often the case that best energy efficiency was achieved when the application was not fully threaded. In our benchmark suite, we found three cases (UA-B, UA-C and LU-B) achieving the best energy efficiency when the thread count was not maximum.
- By fixing the thread count to the maximum and varying the core frequency, we were able to see energy efficiency improvements only for the UA benchmark. The energy efficiencies of the other benchmarks did not vary very much. Moreover, in no configuration the maximum frequency yielded best energy efficiency. For the Zen platform, we found that 2000MHz being the modal frequency for best energy efficiency.
- In general, across our benchmark suite, decreasing the thread-count with the increasing problem size delivered better energy efficiency.
- The optimal thread count varied from problem to problem and this is discussed in detail in Section 4.5.
- When varying both the thread-count and the core frequency (3rd configuration $T_{opt}f_{opt}$), we managed to secure both energy savings and runtime performance improvements in all but three cases (LU-B, LU-C and UA-B)

4.4.2 Skyray Platform

Some of the key findings on the Skyray platform are:

- We found that in four out of eight cases, fully threading an application yielded the best energy efficiency (UA-B, UA-C, LU-B and LU-C).
- Clocking the cores at maximum frequency delivered better energy efficiency in seven out of eight cases (DA-A,LU,SP,UA).

- We found the maximum frequency being the modal frequency for the best energy efficiency. The second modal frequency for the best energy efficiency is 1500MHz, which is the next available frequency for scaling.
- In general, the optimal thread-count did not vary very much with the varying problem sizes except for DC. This observation is noticeably different from the one with the Intel-based Zen platform.
- When varying only the thread-count, energy savings were made along with the performance improvements for all but one case (SP-B). The improvements were recorded relative to the non-threaded and fully-threaded cases.
- When varying both the thread-count and the core frequency, it was possible to secure maximum energy savings without impacting performance in all cases. However, for two cases (DC-B and SP-C), maximum energy savings were possible with slightly degraded performance benefits relative to the best possible performance. This was a clear demonstration of the trade-off of performance and energy.
- Varying the core frequency and the thread-count did not bring any significant benefits on energy savings. This was clearly demonstrated for two benchmarks: LU and UA, where the savings did not vary at all.

4.4.3 Dori Platform

Some of the key findings on the Dori platform are (the number of cores are only 4):

- When the frequency was fixed to a particular value, maximum threading yielded best energy efficiency in four out of the eight cases.
- When the frequency was varied, maximum threading yielded best energy efficiency in five out of the eight cases (all but DC-A, DC-B and SP-C).
- When frequency was varied, in two out of eight cases we found maximum frequency giving best energy energy efficiency (DC-A and SP-C).
- We found 1000MHz being the modal core frequency for the best energy efficiency.
- Just varying only the thread-count brought energy savings along with considerable performance improvements (relative to two baseline cases) for most of the cases. The exceptions are that for the DC-A benchmark, we found that non-threaded case being the most energy efficient variant and it was not possible to secure any savings for the UA and LU benchmarks without altering the core frequency.

- When both thread-count and frequency were varied, additional energy savings was possible with a slight decrease in performance except for the DC-B (with 10% performance improvement w.r.t to T_{max}) and SP-C (no performance improvement w.r.t to T_{max}). If only the energy savings to be accounted, it can be said that energy savings was possible in seven out of eight cases (except SP-C).

4.5 Overall Summary

We summarise all these findings in Tables 6, 7 and 8. The tables are presented here capturing the numerical significance of the results presented in previous sections and in the supplementary material. In summary, the following can be concluded from these results:

- The hyper-threading, when present, clearly sustains the energy efficiency in highly computational applications. This is true, almost across all benchmarks, except the Datacube Benchmark (Class B). We conclude that when hyper-threading is enabled, computationally bound applications benefits from reduced runtimes while consuming very little additional energy and thus leads to better energy efficiency. Furthermore, when hyper-threading is enabled, the context switches are easily facilitated. In the absence of hyper-threading, although cache-size per thread increases, the context switching causes extra energy to be consumed.

- As the number of threads increases, it is common that runtimes decrease to a certain extent. However, the energy consumption does not necessarily have to *monotonically* increase or decrease. In general, the energy consumption decreases and then increases beyond a certain number of threads on the AMD platform. On the Intel platform, however this increases almost in all cases.

- The energy profile of binaries produced by different compilers vary significantly even when their runtime profiles are matched. In some cases, a binary produced by a specific compiler offers consistently better efficiency throughout all the threading regions. We believe that this could be due to better instruction selection or ability to vectorize, which provides better energy savings although the runtime behaviours of the instruction mix may be the same.

- In most cases, fully threading an application rarely offers best energy efficiency. Although this is a known behaviour with respect to performance, the exact thread-count for the best energy savings is not the same as the one which gives the best runtime. When the core frequency is fixed, under-threading enables securing a better energy efficiency metric than a fully-threaded case. For example, on Skyray, for the Datacube benchmark, comparing the energy savings and runtime benefits against the default baseline

Table 6: Effect of Multithreading on Energy Consumption. Summary of findings for the zen platform.

Benchmark	**Class**	**Parameter**	**Default Values at f_{max}**	**Optimal Values at f_{max}**	**Optimal Values at f_{opt}**
DC	A	Frequency / MHz	2668	2668	2000
		Threads	24	2	6
		Ω	1.1	1.8	5.5
		Savings (wrt T_1)	-34%	16%	21%
		Gain (wrt T_1)	1.5×	1.5×	1.8×
		Savings (wrt T_{max})	-	37%	41%
		Gain (wrt T_{max})	-	1.0×	1.2×
DC	B	Frequency / MHz	2668	2668	2000
		Threads	24	10	2
		Ω	0.2	1.8	12.0
		Savings (wrt T_1)	-109%	-1%	48%
		Gain (wrt T_1)	0.4×	1.8×	1.3×
		Savings (wrt T_{max})	-	51%	75%
		Gain (wrt T_{max})	-	4.2×	3.0×
LU	B	Frequency / MHz	2668	2668	2000
		Threads	24	24	20
		Ω	9.4	9.4	34.2
		Savings (wrt T_1)	10%	10%	45%
		Gain (wrt T_1)	8.4×	8.4×	6.8×
		Savings (wrt T_{max})	-	0%	38%
		Gain (wrt T_{max})	-	1.0×	0.8×
LU	C	Frequency / MHz	2668	2668	2400
		Threads	24	10	12
		Ω	8.2	10.1	21.5
		Savings (wrt T_1)	0%	24%	34%
		Gain (wrt T_1)	8.2×	7.7×	7.7×
		Savings (wrt T_{max})	-	24%	34%
		Gain (wrt T_{max})	-	0.9×	0.9×
SP	B	Frequency / MHz	2668	2668	2000
		Threads	24	6	10
		Ω	1.6	4.0	19.0
		Savings (wrt T_1)	-116%	-6%	9%
		Gain (wrt T_1)	3.4×	4.2×	4.3×
		Savings (wrt T_{max})	-	51%	58%
		Gain (wrt T_{max})	-	1.2×	1.3×
SP	C	Frequency / MHz	2668	2668	2400
		Threads	24	4	4
		Ω	1.3	3.6	16.7
		Savings (wrt T_1)	-133%	7%	23%
		Gain (wrt T_1)	3.0×	3.4×	2.9×
		Savings (wrt T_{max})	-	60%	67%
		Gain (wrt T_{max})	-	1.1×	1.0×
UA	B	Frequency / MHz	2668	2668	2400
		Threads	24	24	24
		Ω	5.8	5.8	21.4
		Savings (wrt T_1)	-34%	-34%	0%
		Gain (wrt T_1)	7.7×	7.7×	7.0×
		Savings (wrt T_{max})	-	0%	26%
		Gain (wrt T_{max})	-	1.0×	0.9×
UA	C	Frequency / MHz	2668	2668	2667
		Threads	24	24	24
		Ω	5.6	5.6	18.2
		Savings (wrt T_1)	-30%	-30%	-1%
		Gain (wrt T_1)	7.3×	7.3×	7.1×
		Savings (wrt T_{max})	-	0%	22%
		Gain (wrt T_{max})	-	1.0×	1.0×

Table 7: Effect of Multithreading on Energy Consumption. Summary of findings for the skyray platform.

Benchmark	**Class**	**Parameter**	**Default Values at f_{max}**	**Optimal Values at f_{max}**	**Optimal Values at f_{opt}**
DC	A	Frequency / MHz	2000	2000	2000
		Threads	16	4	4
		Ω	1.9	2.6	28.9
		Savings (wrt T_1)	-16%	10%	10%
		Gain (wrt T_1)	2.2×	2.3×	2.3×
		Savings (wrt T_{max})	-	22%	22%
		Gain (wrt T_{max})	-	1.1×	1.1×
DC	B	Frequency / MHz	2000	2000	1500
		Threads	16	4	8
		Ω	0.0	3.2	5112.2
		Savings (wrt T_1)	-973%	22%	7%
		Gain (wrt T_1)	0.0×	2.5×	3.1×
		Savings (wrt T_{max})	-	93%	91%
		Gain (wrt T_{max})	-	144.0×	179.4×
LU	B	Frequency / MHz	2000	2000	2000
		Threads	16	16	16
		Ω	44.9	44.9	295.0
		Savings (wrt T_1)	64%	64%	64%
		Gain (wrt T_1)	16.0×	16.0×	16.0×
		Savings (wrt T_{max})	-	0%	0%
		Gain (wrt T_{max})	-	1.0×	1.0×
LU	C	Frequency / MHz	2000	2000	2000
		Threads	16	16	16
		Ω	42.4	42.4	302.0
		Savings (wrt T_1)	62%	62%	62%
		Gain (wrt T_1)	16.0×	16.0×	16.0×
		Savings (wrt T_{max})	-	0%	0%
		Gain (wrt T_{max})	-	1.0×	1.0×
SP	B	Frequency / MHz	2000	2000	2000
		Threads	16	6	6
		Ω	2.5	6.7	59.9
		Savings (wrt T_1)	-42%	34%	34%
		Gain (wrt T_1)	3.6×	4.4×	4.4×
		Savings (wrt T_{max})	-	54%	54%
		Gain (wrt T_{max})	-	1.2×	1.2×
SP	C	Frequency / MHz	2000	2000	2000
		Threads	16	4	4
		Ω	1.4	4.2	34.4
		Savings (wrt T_1)	-73%	29%	29%
		Gain (wrt T_1)	2.4×	3.0×	3.0×
		Savings (wrt T_{max})	-	59%	59%
		Gain (wrt T_{max})	-	1.3×	1.3×
UA	B	Frequency / MHz	2000	2000	2000
		Threads	16	16	16
		Ω	14.3	14.3	134.6
		Savings (wrt T_1)	36%	36%	36%
		Gain (wrt T_1)	9.2×	9.2×	9.2×
		Savings (wrt T_{max})	-	0%	0%
		Gain (wrt T_{max})	-	1.0×	1.0×
UA	C	Frequency / MHz	2000	2000	2000
		Threads	16	16	16
		Ω	13.4	13.4	119.8
		Savings (wrt T_1)	32%	32%	32%
		Gain (wrt T_1)	9.1×	9.1×	9.1×
		Savings (wrt T_{max})	-	0%	0%
		Gain (wrt T_{max})	-	1.0×	1.0×

Table 8: Effect of Multithreading on Energy Consumption. Summary of findings for the dori platform.

Benchmark	**Class**	**Parameter**	**Default Values at f_{max}**	**Optimal Values at f_{max}**	**Optimal Values at f_{opt}**
DC	A	Frequency / MHz	1800	1800	1000
		Threads	4	2	1
		Ω	1.0	1.1	2.4
		Savings (wrt T_1)	-24%	-14%	48%
		Gain (wrt T_1)	1.2×	1.2×	0.7×
		Savings (wrt T_{max})	-	8%	58%
		Gain (wrt T_{max})	-	1.0×	0.5×
DC	B	Frequency / MHz	1800	1800	1200
		Threads	4	2	2
		Ω	0.9	1.2	2.7
		Savings (wrt T_1)	-24%	-9%	17%
		Gain (wrt T_1)	1.1×	1.3×	1.2×
		Savings (wrt T_{max})	-	12%	33%
		Gain (wrt T_{max})	-	1.2×	1.1×
LU	B	Frequency / MHz	1800	1800	1000
		Threads	4	4	4
		Ω	2.4	2.4	5.7
		Savings (wrt T_1)	-14%	-14%	24%
		Gain (wrt T_1)	2.8×	2.8×	2.2×
		Savings (wrt T_{max})	-	0%	33%
		Gain (wrt T_{max})	-	1.0×	0.8×
LU	C	Frequency / MHz	1800	1800	1000
		Threads	4	4	4
		Ω	3.1	3.1	5.3
		Savings (wrt T_1)	-8%	-8%	28%
		Gain (wrt T_1)	3.4×	3.4×	2.7×
		Savings (wrt T_{max})	-	0%	33%
		Gain (wrt T_{max})	-	1.0×	0.8×
SP	B	Frequency / MHz	1800	1800	1000
		Threads	4	2	4
		Ω	1.6	1.7	4.2
		Savings (wrt T_1)	-29%	-6%	13%
		Gain (wrt T_1)	2.1×	1.8×	1.7×
		Savings (wrt T_{max})	-	18%	32%
		Gain (wrt T_{max})	-	0.9×	0.8×
SP	C	Frequency / MHz	1800	1800	1800
		Threads	4	2	2
		Ω	1.4	1.7	3.7
		Savings (wrt T_1)	-38%	-8%	-8%
		Gain (wrt T_1)	1.9×	1.8×	1.8×
		Savings (wrt T_{max})	-	22%	22%
		Gain (wrt T_{max})	-	1.0×	1.0×
UA	B	Frequency / MHz	1800	1800	1200
		Threads	4	4	4
		Ω	2.1	2.1	5.5
		Savings (wrt T_1)	-32%	-32%	6%
		Gain (wrt T_1)	2.8×	2.8×	2.1×
		Savings (wrt T_{max})	-	0%	29%
		Gain (wrt T_{max})	-	1.0×	0.8×
UA	C	Frequency / MHz	1800	1800	1200
		Threads	4	4	4
		Ω	2.7	2.7	6.0
		Savings (wrt T_1)	-18%	-18%	21%
		Gain (wrt T_1)	3.2×	3.2×	2.6×
		Savings (wrt T_{max})	-	0%	33%
		Gain (wrt T_{max})	-	1.0×	0.8×

($T_{max}f_{max}$) shows that using eight threads saves up to 93% of energy and offers around $144\times$ performance improvement. This is primarily because, considering that the Datacube benchmark is rather data intensive, as the threading increases, cache-per-thread decreases which eventually leads to excess energy consumption due to data movements and prolonged wait-states.

- In many cases, varying the frequency along with the number of threads, offers a wide range of selections for an optimum number of threads to give maximum energy efficiency. This can lead to substantial energy savings and performance benefits. For example, on the Zen platform, for the SP benchmark (class B), lowering the core frequency to 2000MHz and the thread count to 10 can save up to 58% of energy while yielding 30% performance improvement over the fully-threaded baseline ($T_{max}f_{max}$). However, not all architectures, may lead to benefits when changing the core frequency.

- When threading an application, careful consideration must be given to the overall size of the cache memories and to the size of the data involved in computation as they directly affect the energy consumption. Systems with larger caches involve considerably less data movements and thus reduced energy consumption due to data movements.

- The optimal values for the thread-count and frequency are heavily dependent on the nature of the application and the platform. In general platforms we found fully threading the highly computational benchmarks (LU and UA) led to better energy efficiency and reducing the thread-count for highly communication benchmarks (SP) or I/O intensive ones (DC) is always beneficial. Furthermore, reducing the thread-count with the increasing problem sizes is also beneficial.

- In particular, we observed that on the Zen platform, indirect memory accesses leads to reduced energy efficiency - whereas on the other two platforms, this is sustained throughout.

5 Related Work

The investigation of energy savings in the context of software has mainly been explored from the point of dynamic voltage and frequency scaling (DVFS) [11, 12, 13, 14, 15] and dynamic concurrent throttling (DCT) setup. In the former, either the voltage or the frequency or both are varied at runtime to save energy from processors. The idea of DVFS is mainly driven by capturing the state of the processor during the execution of a software and clocking the processor accordingly. For example, if a heavy memory or DMA transfer is required, the processor may be clocked down and the DVFS is utilised. The motivation is to optimise both the energy consumption and performance of software.

This idea of exploiting the available slack time has been extensively studied, in particular in the embedded and in the single processor settings [13, 14]. Studies by Zhu and Mishra *et. al.* address the case of covering DVFS scheduling when slacks can be shared across a pool of processors [15] or when known statically [12]. The concept has also been stretched to application domains: for example, in [11], Horvath *et. al.* utilises DVFS to save energy on large-scale multi-tier web servers; Cao *et. al.* conduct a suite of experiments on a large-scale optimisation framework [16].

In [17], Li and Martinez present an analytical model to derive power consumption of a parallel region of an application when the parallel efficiency and the number of processors are known in advance. In [18], Cho and Melhem derived an analytical model for optimal energy consumption in parallel applications by understanding the interaction between serial and parallel regions, performance and energy consumption.

In the context of high-performance and parallel applications, Ge *et. al.* [19] studied opportunities for applying DVFS in parallel applications running on clusters, where they exploit fine-grained timing slacks. In [10], Song *et. al.* propose an iso-efficiency model where they apply DVFS for retaining the efficiency of applications. In relation to this, Cameron *et. al.* studied means [20, 21] for deriving more accurate parallel speedup model for modern processors capable of DVFS. In other words, through their models, they capture the impact on the performance by different power modes of the system. A similar, but probability-based approach has been outlined by Lorch and Smith [22].

In contrast, our aim in this paper is to understand the interactions of different parameters, such as compilers, thread-count, frequency, hyper-threading with energy and performance in a rather exhaustive fashion. Although these studies have provided an insight into the actual context where DVFS can be applied, the overall impact and understanding, as outlined in this paper, is not obvious. The model outlined in our paper is relatively simple variant of energy-delay product in assessing the energy efficiency as we are not exploring any slacks in the applications for regional adjustments of parameters.

6 Conclusions and Further Work

In this paper, we analysed the effect of multi-parameter interactions on the energy consumption and energy efficiency of multithreaded software. For this purpose, we used a variant of a time-delay product, yet reusable, metric. The metric is an alternative to existing metrics and it helps in quantifying the energy consumption against performance. Our exhaustive evaluation based on this metric using a suite of benchmarks has provided us with a detailed insight. This study is the first of its kind to exhaustively evaluate and understand the impact of multiple parameters on energy and performance.

Our results show that, in many cases, energy savings can be made with little or

no impact on performance by carefully adjusting these parameters. We show that in some cases, simply using maximum permissible number of threads can lead to a non-optimal condition where the energy efficiency is not maximum. By choosing an appropriate compiler, enabling hyper-threading, adjusting the thread-count and frequency can lead to substantial energy savings and runtime benefits. Knowing the nature of the application, understanding the data size and its impact on the caching, can also lead to better energy savings. In general, maximal threading is useful for applications which are highly computational and non-maximal threading for applications with higher communication to computation or I/O to computation ratio.

In future work, we plan to extend this work to account for other parameters such as tiling and voltage along with an enhanced metric to account their contributions.

Acknowledgments

This research is supported by the Oxford Martin School, University of Oxford. We also thank the Scalable Performance Laboratory of VirginaTech for providing access to their Dori system.

References

[1] Tiwari V, Malik S, Wolfe A. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on VLSI Systems* 1994; **2**(4):437–445.

[2] Poon ASY. An energy-efficient reconfigurable baseband processor for wireless communications. *IEEE Transactions on VLSI Systems* 2007; **15**(3):319–327.

[3] Borkar SY. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro* 2005; **25**(6):10–16.

[4] The International Technology Roadmap for Emerging Research Devices 2011. URL `http://www.itrs.net/`.

[5] Lipshitz B, Ballard G, Demmel J, Schwartz O. Communication-avoiding parallel Strassen: Implementation and performance. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, IEEE Computer Society Press: Los Alamitos, CA, USA, 2012; 101:1–101:11.

[6] Ballard G, Demmel J, Holtz O, Schwartz O. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications* 2011; **32**(3):866–901.

[7] Anderson M, Ballard G, Demmel J, Keutzer K. Communication-avoiding QR decomposition for GPUs. *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, IEEE Computer Society: Washington, DC, USA, 2011; 48–58.

[8] Bailey D, Barszcz E, Barton J, Browning D, etal. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications* Fall 1991; **5**(3):63–73.

[9] Gonzalez R, Horowitz M. Energy dissipation in general purpose microprocessors September 1996.

[10] Ge R, Feng X, Song S, Chang HC, Li D, Cameron KW. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems* 2010; **21**(5):658–671.

[11] Horvath T, Abdelzaher T, Skadron K, Liu X. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Trans. Comput.* Apr 2007; **56**(4):444–458.

[12] Mishra R, Rastogi N, Zhu D, Mossé D, Melhem R. Energy aware scheduling for distributed real-time systems. *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, IEEE Computer Society, 2003; 21.2–.

[13] Shin D, Kim J, Lee S. Intra-task voltage scheduling for low-energy, hard real-time applications. *IEEE Des. Test* Mar 2001; **18**(2):20–30.

[14] Yao F, Demers A, Shenker S. A scheduling model for reduced cpu energy. *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, IEEE Computer Society, 1995; 374–.

[15] Zhu D, Melhem R, Childers BR. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems. *IEEE Trans. Parallel Distrib. Syst.* Jul 2003; **14**(7):686–700.

[16] Cao Z, Easterling DR, Watson LT, Li D, Cameron KW, Feng WC. Power saving experiments for large-scale global optimisation. *Int. J. Parallel Emerg. Distrib. Syst.* Oct 2010; **25**(5):381–400.

[17] Li J, Martínez JF. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Trans. Archit. Code Optim.* Dec 2005; **2**(4):397–422.

[18] Cho S, Melhem R. Corollaries to amdahl's law for energy. *IEEE Comput. Archit. Lett.* Jan 2008; **7**(1):25–28.

[19] Ge R, Feng X, Cameron KW. Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters. *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, IEEE Computer Society: Washington, DC, USA, 2005; 34–.

[20] Li D, de Supinski BR, Schulz M, Cameron KW, Nikolopoulos DS. Hybrid MPI/openMP power-aware computing. *IPDPS*, IEEE, 2010; 1–12.

[21] Ge R, Cameron KW. Power-aware speedup. *IPDPS*, IEEE, 2007; 1–10.

[22] Lorch JR, Smith AJ. PACE: A new approach to dynamic voltage scaling. *IEEE Trans. Computers* 2004; **53**(7):856–869.