Energy-Aware Scheduling for Parallel Applications on Multicore Systems

Jason Mair, Zhiyi Huang, Haibo Zhang
Department of Computer Science
University of Otago
New Zealand

ABSTRACT

This chapter discusses the energy-aware scheduling techniques for parallel applications on multicore computers. Key techniques for developing an energy-aware scheduler, such as estimation of power usage and performance features per application, are analyzed and evaluated. We first discuss the runtime profiling techniques for collecting detailed application-specific information to be used by the scheduler. Then we focus on the techniques that estimate power usage and performance features. Performance features such as speedup and CPU-intensiveness enable the scheduler to make the tradeoffs between power consumption and the performance of the application. Preliminary experimental results show that energy-aware scheduling could save a significant amount of energy by adopting novel scheduling policies based on the knowledge of performance features collected from the applications.

1. INTRODUCTION

Energy consumption has been a major concern for a long time in regards to portable consumer electronics such as smartphones and laptops. The explicit goal has been to extend the battery life of such devices, making them more useful. However, much focus has been recently shifted to improve the energy efficiency of large-scale systems such as servers, grids, and clusters due to the following two primary reasons: firstly, there is a traditional economic incentive of reducing the TCO (Total Cost of Ownership) by reducing the energy consumption. The cost of powering and cooling a large-scale system could be in the order of millions of dollars, making it an increasingly important issue due to the continuously increasing deployment of such systems. Secondly, there is an increasing awareness from consumers of the environmental impact that occurs during energy production. In developed countries, about 70% of generated electricity comes from the burning of fossil fuels (Nordman & Christensen, 2009), which in turn produces greenhouse gas emissions that impact on the environment. This environmental problem is driving the increased demand for more energy efficient solutions.

Apart from these primary reasons for the study on energy efficiency of large-scale systems, other indirect benefits are attainable through improving energy efficiency, such as increased system stability and uptime. Since a system using less power generates less heat, its stability becomes better. According to Arrenhius' Equation (which was first proposed by the Dutch chemist J. H. van 't Hoff in 1884 and then proved by Swedish chemist Svante Arrhenius in 1889), for every increase of 10°C (18°F) in temperature, the failure rate of a system doubles, which could cause more system downtime. As can be seen in Table 1, the costs of system downtime can be quite significant to businesses (Hsu & Feng, 2005).

Table 1: Estimated Cost on an Hour of System Downtime. Adapted from (Hsu & Feng, 2005)

Service	Cost of One Hour of Downtime
Brokerage Operations	$6,450,000
Credit Card Authorization	$2,600,000
eBay	$225,000
Amazon.com	$180,000
Package Shipping Services	$150,000
Home Shopping Channel	$113,000
Catalog Sales Center	$90,000

So far, energy efficient solutions are only incorporated into the deployments of large-scale systems through the use of newer, more energy efficient hardware. Relying on this approach alone has two key drawbacks. The first is the prohibitive cost of replacing the existing hardware with a more energy efficient alternative. Secondly, this approach could also bring about more waste of resources, because some of the hardware being replaced are still sufficient to complete the work, though in a less energy efficient way.

Moreover, relying solely on more energy efficient hardware to reduce the total energy consumption of the IT industry may not have its desired impact, because it may instead cause an increase in the total energy consumption. Due to the trend of improving energy efficiency and reducing operational cost, large-scale systems are more affordable, which may in turn lead to an increase in deployments (Tomlinson, Silberman, & White, 2011). However, software solutions do not suffer from this problem to the same extent because they help change the way in which the hardware is used and promote energy efficiency through their wider adoption in existing systems. Additionally, software solutions are more cost effective as they are able to leverage new technologies in commodity components, such as Dynamic Voltage and Frequency Scaling (DVFS) and multicore chips.

Because of these reasons, this chapter explores how scheduling software is able to make use of technologies in commodity processors to reduce the energy consumption of large-scale systems. The chapter is organized as follows. Section 2 discusses the motives and requirements of energy-aware scheduling. Then runtime profiling techniques are discussed in Section 3, followed by methods for estimating power usage of applications in Section 4. Section 5 discusses the performance features that can be utilized by scheduling software to save energy. Section 6 shows the effect of energy-aware scheduling through preliminary experimental results. Finally, Section 7 gives the conclusions and future work.

2. ENERGY-AWARE SCHEDULING

Traditional schedulers for large-scale systems, such as grids and clusters, commonly focus on maximizing the performance for each parallel application. They normally allocate all or a predetermined number of cores to an application to achieve the best performance, primarily attributed to the low utilization of these systems. The average utilization of many large-scale systems ranges from 10% to 50%, while in some cases are even below 5% (Barroso & Holzle, 2007). However, these traditional schedulers are not energy-efficient as energy saving is not taken into account.

Even though some energy-aware schedulers (Rong & Pedram, 2008; Mishra, Rastogi, Zhu, Mosse, & Melhem, 2003) have been proposed for real-time systems, these schedulers are constrained by stringent deadlines, thereby providing few opportunities for making significant energy savings. In contrast, many applications in large-scale systems are executed with loose deadlines. For example, a computing center may have a number of routine applications that need to be executed once a day, or are used for end-of-day

reporting. Each application has no strict performance requirements as long as execution completes within a day or by morning. These tasks are normally executed in a batch for ease of scheduling. The loose deadlines for batch processing of parallel applications in computing centers allow for a greater number of alternative scheduling schemes to be explored. There is also a greater likelihood of idle time available in large-scale systems that can be used to run applications slower in a low power state in order to achieve energy efficiency. Moreover, modern multicore computers are currently featured with Dynamic Voltage and Frequency Scaling (DVFS), which allows the computers to adjust the supply voltage and operating frequency of the cores dynamically at runtime through software.

This current situation leaves much room for the development of energy-aware scheduling policies that consider the tradeoff between application performance and power usage in large-scale systems. An energy-aware scheduler should extend the current policies to consider both performance and power usage. In order for the scheduler to make the right tradeoff, features of power and performance from each application should be profiled. The profiling techniques to be discussed in Section 3 will allow a scheduler to collect detailed information of an application at runtime. The collected information can be used to estimate the power and performance features of an application, as described in Sections 4 and 5.

3. RUNTIME PROFILING TECHNIQUES

Runtime profiling enables a scheduler to collect detailed and application-specific information for making informed scheduling decisions dynamically. Also runtime profiling allows the scheduler to adapt to changes in application input and system configuration. This section discusses different profiling techniques.

3.1 Source Instrumentation

Source instrumentation is commonly used by programmers to profile the execution behavior of applications in order to gain a good understanding of the application. This profiling information is then used to analyze performance features. The simplest technique is manual source instrumentation, where the programmer manually inserts instructions into the source code at key points of interest. These instructions may print out data values or record times spent in particular code sections, displaying performance statistics. There also exist automated tools such as gprof (Graham, Kessler, & Mckusick, 1982), which can profile applications with source instrumentation and return a set of statistics to the user. Profiling by source instrumentation has the key problem of requiring the source code for all applications. This means proprietary applications cannot be profiled with this method, which is an unacceptable limitation for a scheduler that only has binary code available. Also source instrumentation can influence the instruction access pattern (Hsu & Kremer, 2003), resulting in profiles that do not reflect the original behavior of the application. Therefore, profiling by source instrumentation is not suitable for use by the energy-aware problem we address in this chapter.

3.2 Performance Monitor Counters

Performance Monitoring Counters (PMCs) are a set of special-purpose CPU registers designed to record hardware events during the execution of an application. Low-level system events such as cache misses can be monitored at runtime on a per component basis. This enables a fine-grained profile of an application to be generated in terms of its use of each system component. Currently the number of hardware events and counter registers are model specific. AMD and Intel both have their own model-specific PMC specifications (AMD, 2009; Intel, 2011). For example, the Pentium 4 monitors 48 hardware events, allowing 18 counters to be simultaneously recorded (Sprunt, 2002), while AMD64 can monitor up to 120 hardware events but only has 4 counters (AMD 2009). The events that can be recorded into the counters include:

- Completed instructions

- Cache misses
- Memory accesses
- Branch mispredictions

With these counter values, the scheduler can correlate these events to features of the application, such as power usage, which we will discuss in Section 4.

3.3 Binary Instrumentation

Binary instrumentation is a promising runtime profiling technique for energy-aware scheduling, since it can profile an application with binary code during execution. It is key for an energy-aware scheduler to collect detailed profiling information at runtime using binary instrumentation, because in many situations it is impossible to have the source code of an application. The Pin tool from Intel (Luk et al., 2005) is a state-of-the-art binary instrumentation technique, which allows programmers to write high-level, architecture-independent profiling tools. Pin works by a JIT (Just In Time) compiler that translates selected sections of the binary. Profiling instructions are inserted into the binary code and executed. However, the average base overhead for JIT is 30%, excluding the profiling instructions (Bach et al., 2010).

To minimize the large overhead of JIT, Pin tool provides a key feature that allows it to be dynamically attached to and detached from an application without impacting the execution. This allows an energy-aware scheduler to conduct a brief profiling phase at the beginning of execution and then detach the profiler. Execution is then able to continue at the original performance level after the scheduler has collected sufficient profiles of the application.

4. POWER USAGE

The most important element for energy-aware scheduling is the knowledge of how much power is used by each application in a system. Power usage is usually measured through the use of a power meter, such as the Watts Up? Pro. However, this is not suitable for use with a scheduler for two reasons. First, the power readings, taken every second, may not be at a high enough frequency to accurately capture all of the phase changes within an application. They provide an average of power usage but at a coarse-grained scale. Second, the power readings are taken from the wall outlet, meaning the measurements reflect the power use of the entire system. An energy-aware scheduler requires power readings for individual applications in order to efficiently schedule applications concurrently.

Other methods based on shunt resistors and current probes allow finer-grained power measurements to be taken on a per component basis. However, shunt resistors can impact power consumption by increasing heat generation (Milenkovic, Milenkovic, Jovanov, Hite, & Raskovic, 2005). For multicore computers, even per component granularity is not fine enough as an energy-aware scheduler needs to know the power usage per core.

Currently the best approach to obtain power usage on a per application basis is through estimation. If a power model is constructed, application power usage can be estimated within a system. Singh, Bhadauria, & McKee (2009) presented a method for constructing a proposed power model. The power model is constructed by determining the relationship between system events occurring during application execution and the power measurements taken from a power meter. The experimental results show the model is sufficient for estimating power usage to an acceptable accuracy. An outline of the construction procedure is given below.

The model is constructed on a system using an AMD Phenom processor, which only allows four PMCs to be monitored at once. Given the requirement of generating power estimates at run-time, the model is restricted to using only four values. Therefore, the hardware events are partitioned into four categories that best represent processor behavior: FP Units, Memory, Stalls and Instructions Retired, where Stalls is the number of stalled cycles. Event counters for each category indicate major changes in the power usage of the processor. For example, the number of Retired Instructions and Stalls can indicate the overall processor performance that has a major impact on power usage.

There are 13 hardware events in the four chosen categories. A single event with the strongest relationship to power usage should be found within each category. For example, L2 cache misses could be chosen for Memory since it indicates the level of L3 cache misses and subsequent memory accesses. This selection is determined by running the entire SPEC-OMP benchmark suite (Aslot, & Eigenmann, 2001) while recording the PMCs and power, using a Watts Up? Pro power meter. Several iterations are performed to allow all 13 counters to be recorded for each benchmark without multiplexing the counters. The strength of the relationship between each event and the power usage is measured using Spearman's rank correlation (Spearman, 1904). This correlation method was chosen because it does not require any assumptions about the frequency distribution of the variables. The four chosen events for the categories are:

- FP Units - RETIRED_MMX_AND_FP_INSTRUCTIONS
- Memory - L2_CACHE_MISS
- Stalls - DISPATCH_STALLS
- Instructions Retired - RETIRED_UOPS

All the 13 performance counters are listed in Table 2, along with the correlation values for each of the four chosen events.

Table 2: PMCs Categorized by Architecture and Ordered (Increasing) by Correlation (Based on SPEC-OMP Data). Adapted from (Singh, Bhadauria, & McKee, 2009)

Category	Hardware event	Correlation
FP Units	DISPATCHED FPU:ALL	
	RETIRED MMX AND FP INSTRUCTIONS:ALL	0.23
Instruction Retired	RETIRED BRANCH INSTRUCTIONS:ALL	
	RETIRED MISPREDICTED BRANCH INSTRUCTIONS:ALL	
	RETIRED INSTRUCTIONS	
	RETIRED UOPS	0.39
Stalls	DECODER EMPTY	
	DISPATCH STALLS	0.2
Memory	DRAM ACCESSES PAGE:ALL	
	DATA CACHE MISSES	
	L3 CACHE MISSES:ALL	
	MEMORY CONTROLLER REQUESTS:ALL	
	L2 CACHE MISS:ALL	0.33

To ensure an application independent power model, each of the four chosen events is further investigated through the use of micro-benchmarks. A separate micro-benchmark is used to explore the wide range in values for each event and the respective relationship with power. Each one consists of a large for-loop containing a large number of MOV instructions and arithmetic/floating point operations to stress both the extreme values and the typical usage patterns.

The model is then derived from these data points by fitting a piecewise linear regression function. This was found to give the best results, as the behavior for lower counter values is significantly different from that of higher values. It allows the model to capture more details than would otherwise be possible. The coefficients of the function for the model are calculated using least squares. With the completed power model, power usage for each processor/core can be estimated by passing into the model function the recorded event counters as parameters, with a median error for the SPEC-OMP benchmark of only 5.8%.

5. PERFORMANCE FEATURES

Knowing the performance features of an application is important for energy-aware scheduling. It can prevent the over allocation of resources to parallel applications, where the gains in performance may not be sufficient to offset the increased cost in power. This allows the energy-aware scheduler to make a tradeoff between power and performance while still guaranteeing that all user requirements are met.

In this section, we will discuss several important performance features that can affect the power usage of parallel applications, as well as methods for generating estimates using profiling techniques. The performance features include CPU-intensiveness, memory-intensiveness, speedup, and performance bottlenecks.

5.1 CPU-intensiveness vs. Memory-intensiveness

All applications lie somewhere on the spectrum between the two extremes as either completely CPU-intensive or completely memory-intensive. A CPU-intensive application performs a large number of calculations, requiring very little data. The data it does use generally fits into the processor cache, requiring infrequent memory accesses. As a consequence, the execution time is directly proportional to the processors operating frequency. Lowering the frequency will result in a significant decline in performance of the application, as the number of calculations performed decreases.

Memory-intensive applications process large amounts of data, resulting in a large number of cache misses and frequent fetches from memory or I/O devices. Few calculations are performed during the execution. The majority of the time is spent waiting for data to be fetched from memory, resulting in the execution time being dominated by the speed of memory access, which is often hundreds of times slower than CPU speed. Therefore, a lower CPU operating frequency will not adversely impact the performance of such applications. Though the time taken to perform any calculations will increase a little with a lower frequency, the time spent on waiting for memory data or I/O devices will not be affected.

Energy-aware scheduling can explore the energy saving by lowering the CPU operating frequency without sacrificing much performance if an application is known to be memory-intensive. Fortunately, the CPU operating frequency can be changed using Dynamic Voltage and Frequency Scaling (DVFS), while memory-intensiveness can be estimated with profiling techniques.

5.1.1 Dynamic Voltage and Frequency Scaling

Dynamically adjusting the CPU operating frequency has long been a key functionality in portable consumer electronics for saving energy. Recently Dynamic Voltage and Frequency Scaling (DVFS) was built into commodity multicore processors, allowing software to dynamically adjust the processors supply

voltage and operating frequency during runtime. This is made possible through the use of the Advanced Configuration and Power Interface (ACPI) specification (ACPI, 2010). The specification defines a set of safe DVFS values known as p-states, or performance states. These states ensure stable and safe operation of the processor. For example, the AMD Opteron 8380 processor has four p-states, 800MHz, 1.3GHz, 1.8GHz and 2.5GHz.

Portable consumer electronics experience an interactive workload, which provides the operating system with many opportunities to lower the frequency. For example, when processor utilization drops below a set threshold or it becomes completely idle, the operating frequency can be lower to conserve power. Performance loss is minimal because it occurs during times of little to no processing. This approach has been adopted in modern operating systems such as Linux.

In contrast to the interactive workloads where DVFS is used in response to events, the batched applications can be scheduled with the most energy efficient configuration according to their performance features and given deadlines. Since the workloads of batched applications do not experience the same fluctuations as the interactive workloads in terms of resource utilization, it is more tractable to estimate their power usage and execution time. Once the relationship between power usage and execution time is estimated, an energy-aware scheduler is able to make the tradeoff between power and execution time. For example, applications with loose deadlines allow the scheduler to make a greater tradeoff in performance for power savings, while still meeting all of the users requirements.

However, if such a tradeoff in performance is not acceptable, DVFS can still be used to save power. The energy-aware scheduler can dynamically adjust the DVFS level during execution to reflect the CPU-/memory-intensiveness classification of the application. This allows power to be saved with minimal performance loss. The next sub-section looks at generating such estimates.

5.1.2 Estimation of CPU-intensiveness and memory-intensiveness

If an energy-aware scheduler knows the CPU- or memory-intensiveness of each application, it can select a frequency to match the amount of CPU load to be generated by the application. That is, a CPU-intensive application should be executed with a high frequency, as the performance loss at lower frequencies is significant for such an application. Memory-intensive applications will be executed at lower frequencies since performance is not severely impacted by the CPU operating frequency. In order for an energy-aware scheduler to make such subtle tradeoffs, it should be able to estimate the CPU- and memory-intensiveness.

Hsu & Kremer (2003) proposed a method to analyze the structure of program loops at compile time. The analysis determines the best frequency for each loop while limiting the slowdown within a given threshold. This approach has three drawbacks. First, the code is not very portable, as different systems have different CPU operating frequencies available, requiring the application to be recompiled on each system. Second, this method statically sets the threshold. It does not consider how the execution behavior of the application may change if the application is given different input parameters or input files. Third, the selection of the frequencies is hidden from the users and cannot be utilized by energy-aware scheduling.

A more promising method is the β-adaptation algorithm proposed by Hsu & Feng (2005), which calculates the CPU-/memory-intensiveness of an application at regular time intervals during execution. The calculation is based on the Million Instructions Per Second (MIPS) value, which indicates the rate of work completion. The algorithm is initialized by collecting the MIPS values for each of the CPU-operating frequencies, over the time interval I (e.g. 1 second). These MIPS values and frequencies are

then used as input to Equation (1) to calculate the CPU-/memory-intensiveness (β) at the end of each time interval.

$$\beta = \frac{\sum_{i=1}^{n}\left(\frac{f_{\max}}{f_i}-1\right)\left(\frac{mips(f_{\max})}{mips(f_i)}-1\right)}{\sum_{i=1}^{n}\left(\frac{f_{\max}}{f_i}-1\right)^2} \quad (1)$$

In Equation (1), β indicates the off chip access intensity level, n is the number of operating frequencies available and f_i gives each operating frequency. The equation is largely influenced by the two MIPS values. When an application is memory-intensive, the difference between $MIPS(f_i)$ and $MIPS(f_{max})$ will be minimal. The smaller the difference, the closer to zero this part of the equation becomes. β then begins to approach zero, meaning a value $\beta = 0$ indicates a completely memory-intensive application, in which case the execution time remains the same regardless of the operating frequency. $\beta = 1$ indicates the interval is CPU intensive, in which case halving the CPU frequency will double the execution time. The value of β is then used in Equation (2) to help select the best operating frequency for the next time interval.

$$f^* = \max\left(f_{\min}, \frac{f_{\max}}{1+\delta/\beta}\right) \quad (2)$$

In Equation (2), δ is the slowdown constraint as a percentage, defining a maximum allowable drop in performance. With δ, the user is allowed to set a threshold on the tolerable slowdown of the application while trying to save power. The MIPS(f*) is updated at the end of the time interval to be used in Equation (1) for calculating the next value of β.

The implementation works by forking two threads to run on each core when execution begins. The first thread executes the application given on the command line, while the second thread executes the β-adaptation algorithm. This method is extended to multiprocessor by running an independent thread for the algorithm on each core. This is done for two reasons. First, the level of off-chip accesses can be different for every core. Second, this approach mitigates the overhead in power or time incurred by the synchronization between threads running the β-adaptation algorithm. The results show that the system can save CPU energy consumption by as much as 20% for sequential benchmarks and 25% for the tested parallel benchmarks, at a cost of 3-5% performance degradation (Hsu & Feng, 2005), where the results include the overhead incurred by the β-adaptation algorithm.

This method can be adopted in energy-aware scheduling that utilizes DVFS to save energy. However, it is better to take a more coarse-grained approach that makes the scheduling decisions at the beginning of execution, instead of the runtime monitoring used in the above implementation that may cause substantial runtime overhead for real applications. In the coarse-grained approach, an initial training loop in the application could be executed, collecting the information required by the β-adaptation algorithm using the Pin tool. The binary can be instrumented to insert cycle count instructions at the beginning and end of all data accesses. When combined with a total cycle count for the given training period, the amount of time spent performing off-chip accesses can be used to estimate the memory-intensiveness. Based on this estimation, the scheduler can make the final decision on an acceptable slowdown, and then set the frequency for the application accordingly. The approach works better for batched applications that have a relatively stable performance behavior.

5.2 Speedup

The speedup of a parallel application reflects the decrease in execution time experienced by running the application in parallel, over that of sequential execution. This is why it is commonly used by developers in evaluating the performance gain of a parallel algorithm. The speedup could be typically found by executing the application in a selection of all possible hardware configurations including the number of cores and the CPU speed. The problem with this method is two fold. First, it is very time consuming to explore different configurations in this manner. Even though the speedup information collected in this way may be used for future scheduling, it raises the next problem. Second, the speedup is often dependent upon the size and nature of the input data set. For an energy-aware scheduler to be effective, the scheduling decisions need to be made fast and have to be based on up-to-date information. Since directly measuring speedup is not practical, the scheduler has to make an estimation of speedup to an acceptable level of accuracy.

According to Amdahl's Law (Amdahl, 1967), the execution of a parallel application consists of both a parallel and sequential section. The parallel section consists of the code which is able to be divided up into multiple tasks, which can in turn be concurrently executed on multiple processors. The sequential section is the remainder of the application which is not divisible and executes on a single processor. Equation (3) shows the breakdown of execution time of a parallel application:

$$time_{total} = \frac{time_{parallel}}{num_{cores}} + time_{serial} \tag{3}$$

Where $time_{parallel}$ is the normalized ratio of time spent executing parallelized code, num_{core} is the number of processors the parallelized code is executed on. The remaining time is that spent executing sequential code, $(1 - time_{parallel}) = time_{serial}$. Based on Equation (3), the speedup of a parallel application can be expressed in Equation (4):

$$Speedup = \frac{1}{\frac{time_{parallel}}{num_{cores}} + time_{serial}} \tag{4}$$

Therefore, an estimate of the speedup can be made by knowing the ratio of time spent in the respective parallel and sequential section of an application. This information can be collected using the Pin tool to instrument the execution of a parallel application. For example, in OpenMP programs (OpenMP Architecture Review Board, 2008), the beginning and end of each parallel/sequential loop/section can be selectively recompiled to insert instructions, recording the current cycle count. The time spent in each loop/section is determined by the difference between the two recorded cycle counts.

Based on the counts, the ratio of time spent in parallel/sequential sections can be calculated. Then, with Equation (3), the total time for parallel execution relative to sequential execution time (which is normalized as 1) can be found. For example, an execution of a parallel application on 2 cores where 98% of execution time is spent in the parallel region is given in Equation (5). The resulting execution time is 51% of the original sequential execution time.

$$time_{total} = \frac{0.98}{2} + 0.02 \ = 0.51 \quad (5)$$

According to Equation (4), the speedup of the application is 1/0.51=1.96.

Though this method for speedup estimation may be inaccurate, it works if the parallel/sequential execution times of an application are proportional to the size of input data. For other applications of which the speedup is not easily tractable, there are some other techniques proposed (He, Leiserson, & Leiserson, 2010; Intel, 2010). For example, Cilkview (He, Leiserson, & Leiserson, 2010) is a Pin tool designed for profiling Cilk++ applications. A profile is created by instrumenting the Cilk++ parallelization commands during a sequential execution of the application. The profile is then used to create a DAG view of multithreading, from which the scalability estimates for various core numbers are created.

5.3 Performance Bottlenecks of Multicore Applications

Commodity processors now contain multiple cores, providing an additional dimension for scheduling applications. The decision by Intel in 2004 to halt the design of the Tejas and Jayhawk processors (Flynn, 2004) is seen as the point in time when the clock frequency race ended. Up until then, every new processor generation had increasing clock frequencies, providing application developers a free ride in performance gains. It is believed the decision was made due to the difficulty of keeping the chip within a manageable power envelope and temperature range.

However, this is not the end of Moores Law (Moore et al., 1998), which states that the number of transistors able to be placed on a chip doubles every 18 months. The trend has continued with the development of the Chip Multicore Processor (CMP), which places multiple processor cores on a single silicon die. Each core is of a similar structure to traditional single core processors, with the addition of a L3 cache to facilitate the exchange of data and instructions between cores.

CMPs have led a resurgence of interest from developers in writing parallel applications. This is because parallel applications allow a task to be partitioned and executed across multiple cores, increasing performance. Most parallel applications are usually one of the following:

- **Data Parallelism** occurs through the partitioning of a data set. When the same instructions are repeatedly executed on a single dataset, the performance can be improved by partitioning the dataset across multiple processors.
- **Task Parallelism** occurs through the separation of tasks. When an application contains several independent tasks, performance can be improved by running each task on a separate processor.

One of the biggest challenges while scheduling a parallel application is to determine how many cores each parallel section of the parallel application gets allocated. Too few cores will not allow the maximum possible performance gain (speedup) to be achieved, while too many cores will not necessarily increase the performance if the application has insufficient parallelism. This is due to Amdahl's Law (Amdahl, 1967), which states that as the number of cores increases, the applications speedup becomes limited by the sequential section of the code. According to Equation (4), though the execution time for the parallel part decreases if num_{core} increases, the speedup is bounded by $time_{serial}$ no matter how large num_{core} becomes. For example, if the serial execution time of an application is 10%, the maximum speedup will not exceed 10.

There are two commonly used methods in setting the number of cores for a parallel application. First, the number of cores achieving the maximum speedup is found through manually testing different

configurations and is statically set within the code. Second, the number of cores is set by default to the number of cores available in the system. Both of these methods result in a core allocation that is static and system dependent. They are not able to adapt to common changes in applications such as the problem size. In this section, we will look at an alternative core allocation scheme where the number of cores allocated to an application is determined dynamically by the scheduler according to the synchronization and memory bandwidth generated by the application.

With the profile of an application, an energy-aware scheduler can dynamically allocate the proper number of cores to the application when execution begins. The scheduler is in the best position to make such decisions as it holds detailed information on the execution behavior of the application. It not only ensures an adequate level of performance for the application, but also provides the opportunity to adjust the number of cores based on the power usage of the application.

Run-time variables of the application, such as the size of the input data, will be taken into consideration. If the size of the input data doubles, the workload will naturally double (or could be more). A predetermined core allocation could not take such a factor into account. Therefore, a scheduler using a predetermined core allocation cannot tell when execution would clearly benefit from a greater number of cores. Other variables such as system configuration can also influence performance and power usage of the execution.

Here we present the general feedback-driven core allocation proposed by (Suleman et al., 2008), which determines the best number of cores to allocate to an application by identifying performance bottlenecks like synchronization and off-chip bandwidth. It aims to prevent the allocation of an excessive number of cores, while minimizing the performance impact of synchronization and off-chip bandwidth. Two techniques, Synchronization Aware Threading (SAT) and Bandwidth Aware Threading (BAT), are proposed in this feedback-driven core allocation. The allocation is achieved by initially running a training loop over an application. Then instructions are inserted into key points of interest using a tool similar to Pin. The relevant training loop for each technique is discussed below.

5.3.1 Synchronization Aware Threading (SAT)

Thread synchronization becomes a limiting factor for task parallel applications containing serial sections. As the number of threads increases, execution time of parallel regions decreases. However, since execution time for serial sections is not affected, it becomes the limiting factor in total execution time.

Therefore, the maximum number of threads can be determined by knowing the respective sequential and parallel ratios of total execution time. To find this, the application in question is executed in a sequential training loop. Instructions to read the cycle count are inserted at the beginning and end of the respective parallel and sequential sections. The difference in the recorded cycle counters for the entry and exit points indicates the time spent within that region. This allows for the time spent in the parallel and sequential regions to be calculated, from which the ratio of sequential to parallel code is known.

This allows the number of threads required to reach the maximum performance to be determined before the application is even scheduled to execute on the system. The training loop is terminated either when a steady state is found (less than 5% difference between 3 consecutive loops), or the number of training loop iterations executed reaches 1% of the total programs loop iterations.

5.3.2 Bandwidth Aware Threading (BAT)

Data parallel applications are likely limited by bandwidth, though they do not require much synchronization between threads. They often require large amounts of data from main memory, resulting in frequent off-chip memory accesses and high contention in the off-chip bus.

The ratio of off-chip accesses that an application makes can be profiled by inserting instructions at the points of off-chip accesses, and recording the number of cycles during accesses. The number of cycles used during off-chip accesses can then be compared to the total cycle count, giving the percentage of time spent performing off-chip accesses by each thread. Therefore, the off-chip bus will become fully utilized when the number of cores multiplied by percentage of time of the off-chip accesses becomes greater than or equal to 100%. The maximum number of cores can be found by dividing 100% by the percentage of time of the off-chip accesses. The training loop terminates after 1% of the total loop iterations or 10,000 cycles.

5.3.3 Combination of SAT and BAT

The estimation of synchronization and bandwidth can be combined using both SAT and BAT since many applications may have both synchronization overhead and bandwidth bottleneck. Firstly, the training loop does not terminate until both synchronization overhead and bandwidth bottleneck have been estimated. Secondly, use synchronization and bandwidth separately to determine different numbers of cores to be used during execution, and then the minimum of these two is taken, as it was proven by (Suleman et al., 2008) that choosing the smaller one minimizes overall execution time. The results for 12 multi-threaded applications show that the combination reduces the average execution time by 17% and power by 59% (Suleman et al., 2008).

6. EXPERIMENTAL RESULTS

Previous sections have shown how the performance features and power usage of parallel applications could be estimated and used by energy-aware scheduling to save energy. In this section, we will show with the preliminary experimental results that energy could be significantly saved by using an energy-aware scheduler if the power and performance features are available. With these features, new scheduling policies for energy-saving could be incorporated into energy-aware scheduling.

6.1 Experimental Setup

In our experiment, we use a multicore computer codenamed Dell PowerEdge R905, with four quad-core AMD Opteron 8380 processors and 16GB of RAM. Each processor has four performance states, 800MHz, 1.3GHz, 1.8GHz and 2.5GHz, which can be set by software to adjust the speed of each core.

The following two parallel applications are used as benchmarks in our experiments:

- **Raytrace Application** (Woo, Ohara, Torrie, Singh, & Gupta, 1995): an embarrassingly parallel application with an uneven workload balanced by a shared, low-access task queue (a row of pixels is a unit). The input file car.env is used, with the anti-aliasing level set to 1000 rays.
- **Gaussian Elimination (GE) application** (Huang, Purvis, & Werstein, 2005; Zhang, Huang, Chen, Huang, & Zheng, 2008): a mildly memory-intensive application, which follows the Gaussian Elimination steps over a matrix. The matrix size is 22000 × 22000, and the number of iterations is 6000 in our test.

Both applications are written in OpenMP 3.0 (OpenMP Architecture Review Board, 2008) and are compiled with GCC4.4.1 with optimization argument "-O3". They are run on a standard installation of Linux 2.6.33. In our experiments, the power is measured in watts, using a Watts Up? Pro .net power meter, with an accuracy of ±1.5% + 3 counts (Electronic Educational Devices, 2010).

To demonstrate the benefits of utilizing energy-aware schedulers to reduce energy consumption, we use the following two scheduling policies:

- **Exclusive Policy**: Each application is allocated all of the cores in the system, running at the highest frequency. This gives each application exclusive use of the system for each schedule, and this scheduling policy has been traditionally adopted in most multi-core systems.
- **Sharing Policy**: The cores can be shared by multiple applications simultaneously, i.e., multiple applications can be scheduled to run at the same time. This scheduling policy enables the dynamic configuration of the number of cores allocated to each application.

6.2 Performance Feature Collection

Previously we have shown that making application performance values available to the scheduler can guide the scheduler to change the way applications are executed and in turn save energy. In this section, we present our solution for collecting the statistics on speedup and energy usage of the two applications, Raytrace and GE, showing how much energy could be saved if the scheduler knows such performance information when creating a schedule. Speedup was chosen to illustrate the applications performance as it is often used to evaluate the performance of parallel applications. It is calculated by dividing the sequential (base) time by the time of the current hardware configuration. The base time is taken as the time of sequential execution at the highest operating frequency (i.e. 2.5GHz), as this is the standard configuration used for sequential execution. The other hardware configurations include all four operating frequencies and core counts. Therefore, the speedups for each application at different core speed with different numbers of cores are comparable.

The power is collected through the use of a power meter connected to the wall outlet for the entire system. A power reading, measured in Watts, is taken every second over the respective execution time. Since readings are taken for the entire system, the power usage of idle resources is included. Therefore, the operating frequency of all idle cores is adjusted to the lowest speed (i.e. 800MHz), which consumes the least power. The total energy, measured in *joules*, for each application is taken as the summation of all power readings recorded over the execution time of the application. This approach was used as we currently do not have an implementation of a run-time power estimation scheme.

6.3 Energy Saving with Exclusive Policy

In this experiment we look at the scenario of using the Exclusive Policy to schedule a single application on our multicore system. As was seen in Section 5.3, such naïve policies have the potential to over allocate resources, leading to inefficient schedules. As a comparison, five alternative core configurations are also explored, where all unused cores remain idle, at the lowest operating frequency. This may seem like an unrealistic scenario, where a scheduler would simply allocate all unused cores to another application, however, given that some applications have a minimum performance requirement, it is conceivable that the remaining unused cores are not sufficient to meet such a requirement. With the Sharing Policy that will be discussed shortly, those idle cores can be used by other applications, which can help save more energy when multiple applications are scheduled to run simultaneously.

[**Figure 1(a)** Gaussian Elimination Speedup **Figure 1(b)** Gaussian Elimination Energy (J)]

Figure 1(a) and Figure 1(b) show the speedup and energy usage of GE, respectively. The Exclusive Policy allocates the greatest number of available resources to GE, which in this case is 16 cores, running at 2.5GHz. Since GE is mildly memory-intensive, this results in an over allocation of resources. If the scheduler were aware of the applications performance features, as we advocate in this chapter, it would allocate fewer resources and in turn save more energy.

For the purpose of visualizing this resource over allocation, Figure 1(a) shows the speedup performance measure for different hardware configurations. It can be seen on the 1.8 and 2.5GHz curves, a tipping point is reached at 8 cores, beyond which the allocation of additional cores provides little to no performance increase. In fact, using 12 cores at 2.5GHz provides a lower performance gain than that of 8 cores, and 16 cores provides an improvement of only 0.001. Therefore, a scheduler which is aware of this performance information could be guided towards scheduling GE on 8 cores at 1.8GHz as the speedup is only 0.06 less than that of 2.5GHz. This would result in a 24.5% energy saving from the Exclusive Policy configuration (16 cores at 2.5GHz). Even running at 2.5GHz with 8 cores provides a smaller saving of 17.2%. If more performance loss is tolerable, the scheduler can choose 8 cores with a speed of 800MHz, which can provide a 28.5% reduction in energy compared with the Exclusive Policy.

The energy savings can be explained by the differences in the execution times (illustrated by speedup) and the number of idle resources, which continue to consume power while not contributing to performance. For example, when comparing the configuration of 1 and 4 cores running at 2.5GHz, we can see that the energy difference is explained by the difference in execution time. In Figure 1(a), using 4 cores is three times faster than sequential execution. It also happens to be the case in Figure 1(b), that 4 cores uses about one third of the total energy of sequential execution. This shows that the increased cost in power of using an additional 3 cores is more than offset by a decrease in execution time, thereby reducing total energy.

[**Figure 2(a)** Raytrace Speedup **Figure 2(b)** Raytrace Energy (J)]

Figure 2(a) and Figure 2(b) show the speedup and energy usage of Raytrace. Since Raytrace is almost an embarrassingly parallel application that has little communication overhead, it does not reach a speedup turning point in Figure 2(a). Therefore, the energy-aware scheduler should choose the greatest number of cores with the highest speed, which has the lowest energy usage. Interestingly, if a lower frequency (e.g. 1.8GHz) with 16 cores is chosen, it costs 3.5% more energy with a 21.3% performance loss. This significant performance loss is due to the CPU-intensive nature of the application, as was described in Section 5.1.2.

These two experiments illustrate the importance of providing an energy-aware scheduler with detailed application performance measures, as a one size fits all approach does not guarantee the most energy efficient schedule. Instead of simply allocating all available resources, an energy-aware scheduler should determine if the increased power usage of allocating an additional core is offset by the gain in performance. That is, the decrease in execution time needs to be sufficient to decrease the total energy usage.

6.4 Energy Saving with Sharing Policy

In this experiment, we will show the energy saving of the energy-aware scheduler when a Sharing Policy is used. We use a scenario where there are 16 instances of the Raytrace application to be scheduled on the AMD 16-core machine. These 16 instances can be scheduled using the Exclusive Policy. That is, they are executed one by one, with each instance running with 16 cores. Alternatively, they can be scheduled using the Sharing Policy. In this case, they can be scheduled two by two, each running with 8 cores; or they can be scheduled four by four, each running with 4 cores; and so on. We can list more possible scheduling combinations with different CPU frequencies as well. However, as a case study, we restrict our experiment to five cases running at 2.5GHz, where each case is labeled with M-C, where M represents the number of instances running in parallel and C is the number of cores allocated for each instance. The five cases are represented with 1-16, 2-8, 4-4, 8-2, and 16-1.

If the speedup and the power usage could be estimated by the scheduler, as discussed in previous sections, a metric called Power Per Speedup (PPS) can be calculated. PPS was proposed by (Mair, Leung, & Huang, 2010). It is power usage divided by speedup, a metric measuring power efficiency for speedup increase. The higher the PPS, the more power required per speedup. The scheduler can use PPS to find out which scheduling scheme is most energy efficient.

Table 3: Sharing Policy for Raytrace benchmark

Configuration (M-C)	Energy (J)	Application Speedup	PPS
1-16	176070	12.44	32.23
2-8	150516	7.32	27.39
4-4	143445	3.82	26.24
8-2	139185	1.95	25.70
16-1	131420	1.0	25.06

For example, in Table 3, we show the five scheduling schemes for the 16 instances of Raytrace under the Sharing Policy. According to Table 3, the schedule 16-1, which means 16 instances are running simultaneously, has the best PPS. If this schedule satisfies the performance requirement of the 16 instances, which are most likely true for batched applications, the scheduler can choose the scheme (16-1) with the lowest PPS, i.e. to run each instance with one core. The scheme (16-1) can save 25.36% energy compared with the Exclusive Policy (1-16). The reason for this energy reduction is the increase in throughput gained by running each instance sequentially where the total execution time for the batched job is decreased by 26%.

The throughput is higher by running each instance sequentially because of the sub-linear speedup experienced by the Raytrace benchmark. That is, the speedup gained through parallel execution is less than the number of cores used. Therefore, the cumulative speedup of the system is greater when simultaneously running sequential instances. For example, if each instance of Raytrace executed sequentially takes 60 seconds, the configuration (16-1) will take a total of 60 seconds. However, to execute (8-2) will require two schedules of (60/1.95 = 30.77), giving a total execution time of 61.54. This illustrates the importance of sharing system resources.

7. CONCLUSIONS AND FUTURE WORK

Schedulers for large-scale systems should not only target performance, but also energy efficiency. An energy-aware scheduler brings many benefits such as reduced operating costs, environmental friendliness, and improved system stability. Such a scheduler is able to make the required tradeoff between performance and power saving, taking steps to mitigate extreme performance losses.

This chapter presented and discussed the techniques required by an energy-aware scheduler for parallel applications on multicore systems: runtime profiling, estimation of power usage and performance features per application. With the knowledge of the applications, the scheduler can decide the CPU operating frequency and the number of cores for each application in order to save energy, while the performance loss of the application is kept at an acceptable level.

The proposed energy-aware scheduler was evaluated in a limited experiment, using two typical types of parallel applications. The first of which was memory-intensive with limited scalability, while the other was an embarrassingly parallel, CPU-intensive application. The experiment has been designed to show how an energy-aware scheduler can benefit from the knowledge of the performance and energy features

of different applications to make the best tradeoff between performance and power saving. Our experimental results show that an energy-aware scheduler can save up to 24% energy with little performance loss if the features of the applications are known. According to (Mair, Leung, & Huang, 2010), an energy-aware scheduler can save up to 72% energy in some scheduling scenarios of grid computing where new scheduling policies are used with the knowledge of the performance features and power usage.

Future work includes the implementation of the energy-aware scheduling framework presented in this chapter. This includes new policies such as Sharing, Tortoise and Hare (Mair, Leung, & Huang, 2010), for large-scale systems. Challenging issues include the integration of the profiling and estimation techniques into the scheduler with little overhead and acceptable accuracy, since the profiling overhead and the accuracy of estimation could be barriers for the deployment of the energy-aware scheduler, especially when it is applied to real parallel applications.

REFERENCES

ACPI (2010). Advanced Configuration and Power Interface Specification, Retrieved September 19, 2010, from: http://www.acpi.info/DOWNLOADS/ACPIspec40a.pdf

AMD (2009). BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, Retrieved April 20, 2010, from: http://support.amd.com/us/Processor_TechDocs/31116.pdf

Amdahl, G. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (p. 483-485).

Aslot, V., & Eigenmann, R. (2001), Performance characteristics of the SPEC OMP2001 benchmarks. In Proceedings of the European Workshop on OpenMP

Bach, M. M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., et al. (2010). Analyzing Parallel Programs with Pin. *Computer, 43*(3), 34-41.

Barroso, L. A., & Holzle, U. (2007). The Case for Energy-Proportional Computing. *Computer, 40*(12), 33-37.

Electronic Educational Devices (2010), Watts Up? Operators Manual, Retrieved May 18, 2010, from: https://www.wattsupmeters.com/secure/downloads/manual_rev_9UO0812.pdf

Flynn, J. (2004). Intel Halts Development Of 2 New Microprocessors, Retrieved May 10, 2011, from: http://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007

Graham, S.L., Kessler, P.B., & Mckusick, M.K., (1982). Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction* (p. 120-126).

He, Y., Leiserson, C.E., & Leiserson, W.M., (2010). The cilkview scalability analyzer. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures* (p. 145-156).

Hsu, C., & Kremer, U. (2003). The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 38-48.

Hsu, C.-H., & Feng, W.-C. (2005). A power-aware run-time system for high- performance computing. In *Sc '05: Proceedings of the 2005 ACM/IEEE conference on supercomputing* (p. 1). IEEE Computer Society.

Huang, Z., Purvis, M., & Werstein, P., (2005). Performance Evaluation of View-Oriented Parallel Programming. In *Proceedings of the 2005 International Conference on Parallel Proceessing (ICPP05)* (p. 251–258).

Intel (2010). How to Model the Parallelization of Your Serial Application Using Intel Parallel Advisor, Retrieved May 20, 2011, from: http://software.intel.com/sites/products/evaluation-guides/docs/intelparallelstudio-evaluationguide-model-parallelism.pdf

Intel (2011), Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide (Part 2), Intel, 2011; http://www.intel.com/Assets/PDF/manual/253669.pdf

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., et al. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming language design and implementation (pldi)* (p. 190-200).

Mair, J., Leung, K., & Huang, Z. (2010). Metrics and task scheduling policies for energy saving in multicore computers. In *Proceedings of the 11th IEEE/ACM international conference on grid computing*.

Milenkovic, A., Milenkovic, M., Jovanov, E., Hite, D., & Raskovic, D. (2005). An environment for runtime power monitoring of wireless sensor network platforms. *Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory (SSST 05*) (p. 406-410)

Mishra, R., Rastogi, N., Zhu, D., Mosse, D., & Melhem, R. (2003). Energy Aware Scheduling for Distributed Real-Time Systems. *Proceedings of Parallel and Distributed Processing Symposium*

Moore, G., et al. (1998). Cramming more components onto integrated circuits. In *Proceedings of the IEEE* (Vol. 86, p. 82-85).

Nordman, B., & Christensen, K. (2009). Greener PCs for the enterprise. *IT Professional, 11*(4), 28-37.

OpenMP Architecture Review Board (2008), OpenMP Application Program Interface Version 3.0

Rong, P., & Pedram, M. (2008). Energy-Aware Task Scheduling and Dynamic Voltage Scaling in a Real-Time System. *Journal of Low Power Electronics*, 4(1). 1-10

Singh, K., Bhadauria, M., & McKee, S. A. (2009). Real time power estimation and thread scheduling via performance counters. In *Sigarch computer architecture news* (Vol. 37, pp. 46–55). ACM.

Spearman, C., (1904). The proof and measurement of association between two things. *The American journal of psychology, 15*(1), 72-101

Sprunt, B. (2002). Pentium 4 performance-monitoring features. *Micro, IEEE, 22*(4), 72-82.

Suleman, M. A., Qureshi, M. K., & Patt, Y. N. (2008). Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. In *Proceedings of the 13th international conference on architectural support for programming languages and operating systems* (p. 277-286).

Tomlinson, B., Silberman, M. S., & White, J. (2011). Can more efficient it be worse for the environment? *Computer, 44*(1), 87-89.

Townsend, G. F., (1867) *Three Hundred Aesop's Fables: Literally Translated from the Greek*. London: George Routledge and Sons

Werstein, P., Pethick, M., & Huang, Z. (2003, August). A performance compar- ison of DSM, PVM and MPI. In *Proceedings of the fourth international conference on parallel and distributed computing, applications and technologies* (p. 476-482).

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., & Gupta, A. (1995). The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on computer architecture* (p. 24-36).

Zhang, J., Huang, Z., Chen, W., Huang, Q., & Zheng, W. (2008). Maotai: View-oriented parallel programming on CMT processors. In *The 37th international conference on parallel processing (icpp'08)* (p. 636-643).

ADDITIONAL READING SECTION

Baliga, J., Ayre, R., Hinton, K., & Tucker, R. (2010). Green cloud computing: Balancing energy in processing, storage and transport. In *Proceedings of the IEEE* (pp. 1–19). IEEE.

Barroso, L. A., & Holzle, U. (2007). The case for energy-proportional computing. *Computer, 40*(12), 33-37.

Becchi, M., & Crowley, P. (2006). Dynamic thread assignment on heteroge- neous multiprocessor architectures. In *Proceedings of the 3rd conference on computing frontiers* (p. 29-40).

Bertran, R., Becerra, Y., Carrera, D., Beltran, V., Gonzalez, M., Martorell, X., Torres, J., & Ayguade, E. (2010). Accurate Energy Accounting for Shared Virtualized Environments using PMC-based Power Modeling Techniques. In *International Conference on Grid Computing*

Bertran, R., Gonzalez, M., Martorell, X., Navarro, N., & Ayguade, E. (2010). Decomposable and responsive power models for multicore processors using performance counters. In *Ics '10: Proceedings of the 24th ACM International Conference on Supercomputing* (pp. 147–158). New York, NY, USA: ACM.

Bircher, W., & John, L. (2007). Complete system power estimation: A trickle- down approach based on performance events. In *Performance analysis of systems and software, IEEE international symmposium on* (Vol. 0, p. 158- 168). Los Alamitos, CA, USA: IEEE Computer Society.

Cho, S., & Melhem, R. G. (2010). On the interplay of parallelization, program performance, and energy consumption. *IEEE Transactions on Parallel and Distributed Systems, 21*(3), 342-353.

Fedorova, A., Saez, J. C., Shelepov, D., & Prieto, M. (2009). Maximizing power efficiency with asymmetric multicore systems. *Communications of the ACM, 52*(12), 48-57.

Ge, R., Feng, X., & Cameron, K. W. (2005, November). Performance- constrained distributed dvs scheduling for scientific applications on power- aware clusters. In *Proceedings of the 2005 ACM/IEEE conference on supercomputing* (p. 34).

Gonzalez, R., & Horowitz, M. (1996). Energy dissipation in general purpose microprocessors. In *IEEE Journal of Solid-State Circuits, 31*(9), p. 1277-1284

Hermenier, F., Loriant, N., & Menaud, J.-M. (2006). Power management in grid computing with Xen. *Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops*, 407-416.

Huang, S., & Feng, W. (2009). Energy-efficient cluster computing via accurate workload characterization. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid* (p. 68-75)

Isci, C., & Martonosi, M. (2003). Runtime power monitoring in high-end processors: Methodology and empirical data. In *Micro 36: Proceedings of the 36th annual IEEE/ACM international symposium on microarchitecture* (p. 93). Washington, DC, USA: IEEE Computer Society.

Lefevre, L., & Orgerie, A.-C. (2010). Designing and evaluating an energy efficient cloud. *The Journal of Supercomputing, 51*(3), 352-373.

Lee, Y.C. & Zomaya, A.Y. (2010). Energy Conscious Scheduling for Distributed Computing Systems under Different Operating Conditions. *IEEE Transactions on Parallel and Distributed Systems.*

Li, K. (2008). Performance analysis of power-aware task scheduling algorithms on multiprocessor computers with dynamic voltage and speed. *IEEE Transactions on Parallel and Distributed Systems, 19*(11), 1484-1497.

Nordman, B., & Christensen, K. (2009). Greener PCs for the enterprise. *IT Professional, 11*(4), 28-37.

Orgerie, A.-C., Lefevre, L., & Gelas, J.-P. (2008). Save watts in your grid: Green strategies for energy-aware framework in large scale distributed systems. In *14th ieee international conference on parallel and distributed systems (icpads)* (p. 171-178).

Paulson, L. D. (2010). Energy-saving pcs work while sleeping. *Computer, 43* (12).

Sharma, S., Hsu, C.-H., & Feng, W.-C. (2006, April). Making a case for a green500 list. In *Proceedings of the workshop on high-performance, power-aware computing*.

Shelepov, D., Saez Alcaide, J., Jeffery, S., Fedorova, A., Perez, N., Huang, Z., et al. (2009, April). Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev., 43*(2), 66–75.

Shin, D., Kim, J., & Lee, S. (2001, March). Intra-task voltage scheduling for low-energy hard real-time applications. *Design & Test of Computers, 18*(2), 20-30.

Singh, K., Bhadauria, M., & McKee, S. A. (2009). Real time power estimation and thread scheduling via performance counters. In *Sigarch computer architecture news* (Vol. 37, pp. 46–55). ACM.

Sinha, A., & Chandrakasan, A. P. (2001). Jouletrack - a web based tool for software energy profiling. In *Dac '01: Proceedings of the 38th annual design automation conference* (pp. 220–225). New York, NY, USA: ACM.

Subrata, R., Zomaya, A. Y., & Landfeldt, B. (2010). Cooperative power-aware scheduling in grid computing environments. *Journal of Parallel and Distributed Computing*, 70(2), 84–91.

Sun, H., Cao, Y., & Hsu, W.-J. (2011, April). Efficient adaptive scheduling of multiprocessors with stable parallelism feedback. *IEEE Transactions on Parallel and Distributed Systems*, 22(4), 594-607.

KEY TERMS & DEFINITIONS

Green Computing
Green Computing is the study and practice of improving the environmental sustainability of computing. This is achieved through improving the energy efficiency of computing equipments and reducing their environmental impact.

Energy-aware scheduling
Energy-aware scheduling is a task scheduling policy that takes into account the energy usage of computing tasks. Scheduling decisions are made based upon both energy consumption and performance.

Parallel application
A parallel application is an application software designed to execute multiple tasks in parallel on parallel computers such as multicore, multi-processors, and cluster computers. The goal of a parallel application is to improve its performance with parallel computers.

Multicore system
A multicore system is a computing system with CPU chips that have multiple processors, aka. cores. The cores in a chip may share L2/L3 caches.

Speedup
Speedup is used as a measure of the performance of parallel applications. It reflects how much faster a parallel application is than its sequential version. It is calculated with the sequential execution time divided by the parallel execution time.

CPU-intensive application
A CPU-intensive application is one whose execution time is bounded by the speed of the CPU on which it is executing. A change in the CPU speed will directly impact the execution time.

Memory-intensive application
A memory-intensive application is one whose execution time is bounded by the time spent on memory accesses. A change in the memory accessing speed will directly impact the execution time, while a change in the CPU speed will hardly impact the execution time.

EPT (Energy Per Target)
EPT is an energy metric used to evaluate the energy efficiency of different scheduling policies and hardware configurations. The energy usage is measured over a set time period, which includes both application execution time and idle time. It allows scheduling flexibility to trade off between performance and energy consumption.

Exclusive Policy
The Exclusive Policy is a scheduling policy designed to maximize the performance of parallel applications by exclusively allocating all the resources of a multicore system to a single parallel application.

Sharing Policy

The Sharing Policy is a scheduling policy designed to share the resources of a multicore system amongst multiple parallel applications, allowing for trade-off between performance and energy consumption.