Survey on Power Management Techniques for Energy Efficient Computer Systems

Wissam Chedid and Chansu Yu

Department of Electrical and Computer Engineering
Cleveland State University
2121 Euclid Avenue, SH 332, Cleveland, OH 44115

Abstract

In this paper we describe different power management techniques aiming to reduce power consumption in computer systems. On one hand, static techniques, applied at design time, have been presented with a variety of simulation and measurement tools, targeting different levels of the system's hardware and software components. On the other hand, dynamic power management techniques, applied at runtime, were also discussed. These dynamic techniques aim at reducing energy consumption at CPU level, using DVS, but also try to save energy of the overall system and even on a cluster system level.

1 Introduction

Performance optimization has long been the goal of different architectural and systems software studies, driving technological innovations to the limits for getting the most out of every cycle. This quest for performance has made it possible to incorporate millions of transistors on a very small die, and to clock these transistors at very high speeds. While these innovations and trends have helped provide tremendous performance improvements over the years, they have at the same time created new problems that demand immediate consideration. An important and daunting problem is the power consumption of hardware components, and the resulting thermal and reliability concerns that it raises making power as important a criterion for optimization as performance in both high-end and low-end systems design.

Reducing power consumption is a challenge to system designers. Portable systems, such as laptop computers and personal digital assistants (PDAs) draw power from batteries; so reducing power consumption extends their operating times. For desktop computers or servers, high power consumption raises temperature and deteriorates performance and reliability.

In this paper we will discuss some of the power management techniques proposed so far in the literature. Power reduction techniques can be classified as static and dynamic. *Static Power Management (SPM)* techniques, such as synthesis and compilation for low power, are applied at design time (off-line) and target both hardware and software. In contrast, dynamic techniques use runtime (on-line) behavior to reduce power when systems are serving light workloads or are idle. These techniques are known as *Dynamic Power Management (DPM)*. An overview of these techniques is presented in Table 1.

Table 1: Power Management Techniques Classification Overview

SPM (off-line optimization)				
System/ Component Under Test (SUT/CUT)	Level of Detail	Evaluation Methodology	Description	Section
CPU	RTL level	Cycle-level simulation	Energy models on *Simplescalar* simulator [18] such as *Wattch* [2] and *SimplePower* [3]	2.1.1
	Instruction level	Instruction-level simulation	Energy models on *ARMulator* simulator [4], *JouleTrack* [5]	2.1.2
System	Hardware component level (e.g. hardware state such as CPU sleep/ doze/busy, LCD on/off, etc.)	Functional simulation (Parameters via measurements)	*POSE* (Palm OS Emulator) [6]	2.2.1
	Software component level (Procedure / process / task)	Measurements (Monitoring tools)	Time driven sampling - *PowerScope* [7] and Energy driven sampling [8]	2.2.2
	Hardware & software component level	Complete system simulation (CPU, disc, memory, OS, and application)	*SoftWatt* built upon *SimOS* system simulator	2.2.3
DPM (on-line optimization)				
System/ Component Under Test (SUT/CUT)	Implementation level	Methodology	Description	Section
CPU	CPU and system software	DVS (Dynamic Voltage Scaling)	Interval-based scheduler [10,11] and Real-time system schedulers [12,13]	3.1
System	Components hardware (Disk, network interfaces, displays, I/O devices, etc.) and system software	Low power mode of operation	Shutdown/low- power unused devices [15,16]	3.2
Cluster system	CPU and system software	CVS (Coordinated Voltage Scaling)	Coordinated DVS among multiple nodes [17]	3.3

In the next section we present an overview of the classification of these different techniques. Section 2 describes some power measurement and simulation tools targeting different levels of hardware and software implementations and trying to achieve improvements and optimizations during design time. Some of these static power management techniques, targeting off-line optimization, are presented. In section 3 we discuss dynamic power management techniques targeting system runtime behavior in order to reduce energy at different system levels, achieving on-line optimization.

2 Static Power Management (SPM) Techniques

Power dissipation limits have emerged as a major constraint in the design of microprocessors, and just as with performance, power optimization requires careful design at several levels of the system architecture. Different energy models were presented in previous studies and integrated with already known simulators and measurement tools, targeting different system levels and providing power estimation, measurement and optimization at design time.

These studies can be divided mainly into two areas. The first one is a low-level approach targeting the CPU and investigating its power consumption at both cycle and instruction levels. This *CPU-level* approach will be described in subsection 2.1. The second approach is a high-level approach targeting different or all system components. This *system-level* approach will be described in subsection 2.2.

2.1 CPU-level SPM

Power consumed by the CPU is significant. In papers [1-5], the CPU was the main target of power consumption analysis. Many power-aware models were presented and integrated into already-in-use performance simulators in order to investigate power consumption of the CPU, on a unit basis or for the processor as whole. These investigations were mainly held on two abstraction levels: the *register-transfer level* (or *cycle-level*), described in subsection 2.1.1, and the *instruction-level*, in subsection 2.1.2.

2.1.1 Register-transfer level

Processor energy consumption is generally estimated by *register-transfer level* (RTL) or *cycle-level* simulators [1-3]. During every cycle of the simulated processor operation, the activated (or busy) microarchitecture-level units or blocks are known from the simulation state. Depending on the workload, a fraction of the processor units are active at any given cycle. We can use these

cycle-by-cycle resource usage statistics, available from a trace or execution-driven performance simulator, to estimate the unit-level activity factors. If accurate *energy models* exists for each modeled resource, on a given cycle and if unit *i* is accessed or used, we can estimate the corresponding energy consumed and add it to the net energy spent overall for that unit. So, at the end of a simulation, we can estimate the total energy spent on a unit basis as well as for the whole processor.

In these papers [1-3], the proposed energy models were all built upon *SimpleScalar* [18], a suite of publicly available tools simulating modern microprocessors that implements the *SimpleScalar* architecture (a close derivative of MIPS architecture). The tool set takes binaries compiled for the *SimpleScalar* architecture and simulates their execution on one of several provided processor simulators. The advantages of the SimpleScalar tools are high flexibility, portability, extensibility, and performance.

In [1], two types of energy models, were presented: (1) power-density-based models for design lines with available power and area measurements from implemented chips; and (2) analytical models in terms of microarchitecture-level design parameters such as issue width, number of physical registers, pipeline stage lengths, misprediction penalties, cache geometry parameters, queue/buffer lengths, and so on. The analytical energy behavior was based on simple area determinants and the constants validated with circuit-level simulation experiments using representative tests.

The base microarchitecture model adopted in [1] is a generic, parameterized, out-of-order superscalar processor. The power-performance simulation shows a cycle-by-cycle resource usage statistics to estimate the total energy spent on a unit based as well as for the whole processor. The relationship between performance, power and L1 data cache size was evaluated: the small CPI benefits of increasing the data cache are outweighed by the increases in power dissipation due to large cache. Then, the effects of varying all of the resource sizes within the processor core are shown, including issue queues, rename registers, branch predictor tables, memory disambiguation hardware, and the completion table. Finally, some power-efficient microarchitecture design ideas and trends were discussed: (1) S*ingle-core, wide-issue, superscalar processor chip paradigm*, where increasing the superscalar width beyond a certain limit seems to yield diminishing gains in performance, while continuously reducing the performance power efficiency metric. (2) *Multicluster superscalar processors*, where the classical superscalar CPU is replaced with a set of clusters, so that all key energy consumers are split among clusters. (3) *VLIW or EPIC microarchitectures*, where much of the hardware complexity could be moved to the software. (4) *Chip multiprocessing*, where multiple processor cores on a single die can operate in various ways

to yield scalable performance in a complexity and power-efficient manner. (5) *Multithreading*, where the processor core shares its execution-time resources among several simultaneously executing threads. (6) *Compilers*, supporting the microarchitecture in reducing the power consumption. (7) *Energy-efficient cache architectures*, like filter cache and the dynamic cache size changing during program execution.

For *Wattch* [2], the foundations for the power-modeling infrastructure are parameterized power models of common structures present in modern superscalar microprocessors. The power consumption of the units modeled was considered to depend particularly on the internal capacitances for the circuits that make up the processor. The main modeled processor units fall into four categories: (1) Array structures, (2) fully associative content-addressable memories, (3) combinational logic and wires, and (4) clocking.

Wattch was found to be 1000x faster than existing layout-level power tools, and yet maintains accuracy within 10% of their estimates. Three possible usage flows of *Wattch* were presented in [2] making it as handy for architects as for compiler writers who might use *Wattch* to evaluate power consumption in their design process. The first usage scenario applies to cases where the user is interested in comparing several design configurations that are achievable simply by varying parameters for hardware structures already modeled. The second usage scenario is for software or compiler development, where a single hardware configuration is used and several programs are simulated and compared. The last scenario highlights *Wattch*'s modularity. Additional hardware modules can be added to the simulator. When performing power studies, *Wattch* provides result for a variety of metrics like power, performance, energy, and energy-delay product.

Another execution driven, cycle-accurate RT level energy estimation tool, *SimplePower*, was presented in [3]. *SimplePower* was developed based on transition-sensitive energy models, where each functional unit has its own energy model from a table containing the power consumed for each input transition. It simulates the integer subset of the instruction set of *SimpleScalar*. The simulation flow uses *SimpleScalar* compiler toolset to convert the C source benchmarks to *SimplePower* executables. *SimplePower* simulates these executables providing cycle-by-cycle energy estimates and the switch capacitance statistics for the processor datapath, memory and on-chip buses. The major components of *SimplePower* are: *SimplePower* core, RTL power estimation interface, technology dependent switch capacitance tables, cache/bus estimator, and loader.

SimplePower was used to study different architectural optimizations using a set of benchmarks: (1) *Gating pipeline registers*: The selective gated pipeline register optimization

focuses on reducing the power consumption by using the control signals of the datapath for selectively gating subsets of the pipeline registers. It was observed that 23-36% energy reduction is possible in the datapath, 50% in the pipeline registers and 46% in the register file. (2) *Memory system power optimization*: data transformations are proposed to improve spatial locality in situations where loop transformations are not effective. These transformations, instead of changing the loop execution order, modify the underlying memory layouts of multidimensional arrays, corresponding to variable-renaming operations. Significant energy savings was observed only for smaller cache sizes and associativities, but it had no effect on the processor core consumption. (3) *Bus power optimization*: it consists of reducing the switching activity on the Icache data bus by relabeling the register fields of the compiler-generated instructions. 12% reduction in the total energy reduction in the Icache data bus using the register relabeling optimization was observed.

2.1.2 Instruction-level

An additional approach for energy estimation, using *instruction-level* power analysis, was presented in [4,5]. This technique estimates the energy consumed by a program by summing the energy consumed by the execution of each instruction. Instruction-by-instruction energy costs are precharacterized once for all for each target processor.

In [4], processor instruction-level simulator and memory model were tightly integrated together with an accurate battery model. A methodology was developed were each component is characterized with equivalent capacitance for each of its power states. Energy spent per cycle is a function of equivalent capacitance, current voltage, and frequency. The equivalent capacitance allows to easily scaling energy consumed for each component as frequency or voltage of operation change. Equivalent capacitances are calculated given the information provided in data sheets. The total energy consumed by the system per cycle is the sum of energies consumed by the processor and L1 cache, interconnects and pins, memory, L2 cache, the DC-DC converter and the efficiency losses in the battery. Models for energy consumption and performance estimation of each of these system components were described in [4]. The system used to illustrate this methodology is the SmartBadge with an ARM processor. As a result, the energy models were implemented as extensions to the instruction-level simulator for the ARM processor family called the ARMmulator normally used for functional and performance validation.

A software energy estimation methodology, presented in [5], avoids explicit characterization of instruction energy consumption and predicts energy consumption to within 3% accuracy for a set of benchmark programs evaluated on the StrongARM SA-1100 and Hitachi

SH-4 microprocessors. Based on experiments done on these processors, it was concluded that the common overheads present across instructions result in the variation in current consumption of different instructions being small. The variation in current consumption of programs is even smaller. Therefore, to a first order, we can assume that the current consumption depends only on operating frequency and supply voltage and is independent of the executing program. A second order model that uses energy differentiated instruction classes/cycles is also proposed and it was shown that the resulting current prediction error was reduced to less than 2%. A methodology for separating the leakage and switching energy components was also discussed.

2.2 System-level SPM

There is little benefit in optimizing only the CPU core if other elements participate or sometimes even dominate the energy consumption. To effectively optimize system energy, it is necessary to consider all of the critical components. Different papers [6-9] investigate the power consumption on different system levels, targeting both hardware and software on different levels of abstraction.

In subsection 2.2.1, *state-level models* and measurements are used to account for the energy consumption of the whole system based on the state each device is in or transiting from or to. In subsection 2.2.2, measurements are used to find the system power consumption and help targeting the hotspots in applications and operating system procedures. This approach tries to reduce energy consumption by acting on the *application- and OS-level*. Finally, subsection 2.2.3 describes a *complete system level* simulation tool which models the CPU, memory hierarchy and a low power disk subsystem and quantifies the power behavior of both the application and operating system.

2.2.1 State-level models

A high-level energy optimization technique was presented in [6]. The proposed energy model uses a level of abstraction that reduces the complexity of the hardware state sufficiently by encapsulating low-level details, but encompasses enough information to allow high-level energy optimization. This energy model identifies a set of device power states and transitions between the states that are based on the hardware subsystems of the device. It assumes that the relevant transitions between states occur as the result of system calls, and that, by keeping track of system calls, the system may monitor its own energy consumption. For each separate hardware subsystem, a set of device states is defined. The device states are differentiated by the power consumption of the hardware during steady state (e.g. backlight ON versus backlight OFF). Each state is assigned a power consumption cost by measuring the steady state power consumption in

that state. Each transition between states is assigned an energy consumption cost by measuring the transitional energy consumption. The total energy consumed by the system is determined by summing the power of each device state multiplied by the time spent in each state plus the total energy consumption for all transitions.

The simulation environment was implemented as an extension of the Palm OS Emulator (POSE). POSE is a Windows based application that simulates functionality of the Palm device, emulating its operating system and instruction execution of the Motorola Dragonball processors used in the Palm. The power state model for the Palm was incorporated into this existing environment.

To quantify the power consumption of a device and parameterize the simulator, experiments were held and measurements were taken using the energy model described above in order to capture transient energy consumption as well as steady state power consumption. A Workpad device was connected to a power supply with an oscilloscope measuring the voltage across a small resistor. The power consumption of the basic hardware subsystems of the IBM Workpad device was measured: CPU, LCD, Backlight, Buttons, Pen and Serial link. Measurement programs, like *Power* and *Millywatt*, were used to provide a user interface to call some of the basic functions of the device for measurement intervals. Another category of Palm applications, like *Energy Monitor* and *Hackmaster*, were used to validate the results of the simulator.

2.2.2 Application- and OS-level

An alternative approach for energy estimation using measurements as a basic for estimation was presented in [7,8]. Targeting the power consumption of the whole system and trying to pinpoint the hotspots in applications and operating system procedures, these tools mainly help programmers to produce power aware programs.

In [7], *PowerScope* maps energy consumption to program structure by augmenting the information gathered by a *time-driven statistical sampler*. Using *PowerScope*, one can determine what fraction of the total energy consumed during a certain time period is due to specific processes in the system. Further, one can drill down and determine the energy consumption of different procedures within a process. By providing such fine-grained feedback, *PowerScope* allows attention to be focused on those system components responsible for the bulk of energy consumption. As improvements are made to these components, *PowerScope* quantifies their benefits and helps expose the next target for optimization. Through successive refinement, a system can be improved to the point where its energy consumption meets design goals.

The functionality of *PowerScope* is divided among three software components. Two components, the *System Monitor* and *Energy Monitor*, share responsibility for data collection. The *System Monitor* samples system activity on the profiling computer by periodically recording information that includes the program counter (PC) and process identifier (PID) of the currently executing process. The *Energy Monitor* runs on the data-collection computer, and is responsible for collecting and storing current samples from the digital multimeter. Because data collection is distributed across two monitor processes, it is essential that some synchronization method ensure that they collect samples closely correlated in time. Synchronizing the components was achieved by having the digital multimeter signal the profiling computer after taking each sample. The final software component, the *Energy Analyzer*, uses the raw sample data collected by the monitors to generate the energy profile, off-line. The analyzer runs on the profiling computer since it uses the symbol tables of the executables on disk to map samples to specific procedures.

The tool presented in [8] is based on *energy-driven statistical sampling* and use energy consumption to drive sample collection. A simple 'energy counter' is interposed between the energy supply and the system under study. This counter measures the energy consumed by the system and causes an interrupt to be generated on the system whenever a predefined amount of energy, i.e. energy quanta, has been consumed. The system handles these interrupts by executing a particular interrupt service routine that will record samples identifying the program instructions that were interrupted. The recorded samples are processed, off-line, to generate energy consumption estimates for each application, procedure, and instruction. Results using the energy profiler to study the behavior of 13 benchmarks programs show that a non-trivial amount of energy is spent by the operating system. Additionally, there are often significant differences between the profiles generated by time and energy profiling, especially in workloads that transition between multiple energy states.

In 3.1.2, we discussed some power models built upon *ARMulator*, the instruction-level-simulator for the ARM processors. However in this same paper, [4], a system profiler that correlates both energy consumption and performance to the source code was also integrated into the simulator. During each time interval, the simulator calculates energy consumption of all system components. The profiler works concurrently with the simulator. It periodically samples the simulation results (using sample interval specified by the user) and maps the energy and performance to the function executed using information gathered at the compile time. Once the simulation is complete, the results of profiling can be printed out by the total energy or time spent in each function. MP3 audio decoder was presented as an example of how to use the profiler to quickly and easily target the redesign of the software to run in real time with low energy. Code

transformations were applied in layers, starting from a high level of abstraction with *algorithmic optimization*, and moving down to very detailed and architecture- specific optimization, with *data* then *instruction flow optimizations*. Using this design tool on the MP3 decoder, performance was increased by 92% while decreasing energy consumption by 77%.

2.2.3 Complete system level

Finally, a complete system power simulator was presented in [9], *SoftWatt*, which models the CPU, memory hierarchy and a low power disk subsystem and quantifies the power behavior of both the application and operating system. This tool, built on top of the *SimOS* infrastructure, uses validated analytical energy models to identify the power hotspots in the system components, capture relative contributions of the user and kernel code to the system power profile, identify the power-hungry operating system services and characterize the variance in kernel power profile with respect to workload.

Experiments were held using different Spec JVM98 benchmarks together with the Java Virtual Machine (JVM) runtime system executing on IRIX 5.3. The performance and power profiles are given for different hardware components such as the processor datapath, L1 and L2 I/D caches, memory and disk.

The result of this characterization study can be summarized as follow: (1) From a system perspective, the disk is the single largest consumer of power accounting for 34% of the system power. The adoption of the disk with low-power features shifts this power hotspot to the clock distribution and generation network and the on-chip first level instruction cache. The setting of the disk spin down threshold is critical in the shifting of this hotspot. Further, for single-issue processor configurations, we find that the memory subsystem has a higher average power than the processor core. (2) Among the four different software modes, the user mode consumes the maximum power. Among the other modes, the kernel synchronization operations are expensive in terms of their power consumption. However, their contribution to overall system energy is small due to the infrequent synchronization operations when executing the Spec JVM98 benchmarks. Though the kernel mode has the least power consumption overall, due to the frequent use of kernel services, it accounts for 15% of the energy consumed in the processor and memory hierarchy. Thus, accounting for the energy consumption of the kernel code is critical for estimating the overall energy budget. (3) The per-invocation of the kernel services is fairly constant across different applications. Thus, it is possible to estimate the energy consumed by kernel code with an error margin of about 10% without detailed energy simulation. (4) Whenever the operating system does not have any process to run, it schedules the idle-process. Though this

has no performance implications, over 5% of the system energy is consumed during this period. This energy consumption can be reduced by transitioning the CPU and the memory-subsystem to a low-power mode or by even halting the processor, instead of executing the idle-process.

3 Dynamic Power Management (DPM)

Dynamic power reduction techniques use runtime behavior to reduce power when systems are serving light workloads or are idle. DPM can be achieved in different ways; for example, dynamic voltage scaling (DVS) changes processor supply voltage at runtime as a method of power management [10-13]. DPM can also be used for shutting down unused I/O devices [15,16], or unused nodes of server clusters [17].

Three Dynamic Power Management implementation levels will be discussed in this section. Subsection 3.1 discusses DPM techniques applied at the *CPU- level*, using DVS. In subsection 3.2, a more general approach uses DPM at the *system-level* to save energy of all system components (memory, hard drive, I/O devices, display…). Finally, subsection 3.3 generalizes DPM techniques to be used on *multiple systems*, like a server cluster, where more than one system collaborates to save overall power.

3.1 CPU-level DPM: Dynamic Voltage Scaling (DVS)

A new metric for CPU energy performance, MIPJ or millions-of-instructions-per-joule, was introduced in [10]. Reducing the clock speed causes a linear reduction in energy consumption, but a similar reduction in MIPS. The two effects cancel. MIPJ is unchanged by changes in clock speed. However, a reduced clock speed creates the opportunity for quadratic energy savings; as the clock speed is reduced by n, energy per cycle can be reduced by n^2 by reducing voltage. In order to achieve an increase in MIPJ, the energy savings must be greater than the amount by which the clock rate is reduced.

DVS allows a processor to dynamically changes speed and voltage at run time, thereby saving energy by spreading run cycles into idle time. Because of the non-linear relationship between CPU speed and power consumption, it is better to spread work out by reducing cycle time and voltage than to run the CPU at full speed for short burst and then idle. Knowing when to use full power and when not requires the cooperation of the operating system scheduler.

3.1.1 Interval-based scheduler

In [10,11] *interval based* voltage scheduler has been proposed, which divides time into uniform-length intervals and analyses system utilization of the previous intervals to determine the voltage of the next interval accordingly.

In [11], some complicated algorithms estimate the future workload based on two parameters: *run_percent* and *excess_cycles*. *run_percent* is the fraction of cycles where the CPU is active in an interval. *excess_cycles* is the cycles left over from the previous interval spilled over into later intervals when speed is not fast enough to complete and interval's work. Seven dynamic speed-setting policies were explained, discussed and compared: (1) PAST: a bounded-delay limited-past algorithm that uses the recent past as a predictor of the future. (2) FLAT: Weak on prediction, this policy simply try to smooth speed to a global average. (3) LONG_SHORT: it's a more predictive policy that attempts to find a golden mean between local behavior and a more long-term average. (4) AGED_AVERAGES: this policy employs an exponential-smoothing method, attempting to predict via a weighted average: one which geometrically reduces the weight given to each previous interval as we go back in time. (5) CYCLE: a more sophisticated prediction algorithm. It tries to take advantage of some previous *run_percent* values that looks quite cyclical, to predict. (6) PATTERN: a generalized policy from CYCLE. It attempts to identify the most recent *run_percent* values as repeating a pattern seen earlier in the trace. (7) PEAK: a more specialized version of PATTERN. It uses heuristics based on the expectation of narrow peaks. It expects rising *run_percents* to fall symmetrically back down and falling *run_percents* to continue falling.

Surprisingly, the simplest policy, FLAT, is optimal for low delay values, while LONG_SHORT, which is scarcely more complex, is optimal for the higher delay values. Of the most sophisticated predicting algorithms, PEAK does best, coming close to FLAT and LONG_SHORT in the medium-delay range. AGED_AVERAGE, CYCLE, and PATTERN were all disappointing. From this paper [11], we found that several of the predictive algorithms performed poorly. We might then conclude that simple algorithms based on rational smoothing rather than "smart" predicting may be most effective. Nevertheless, further possibilities for prediction remain to be tried, like policies that might sort past information by process-type, or policies where applications could provide the system with useful information.

In [10], trace driven simulation was used to compare three classes of schedules: (1) an unbounded-delay perfect-future algorithm (OPT) that spreads computation over the whole trace period to eliminate all idle time (regardless of deadlines), (2) a bounded-delay limited-future algorithm (FUTURE) that uses a limited future look ahead to determine the minimum clock rate,

and (3) a bounded-delay limited-past algorithm (PAST) that uses the recent past as a predictor of the future. A PAST scheduler with a 50 ms window shows power savings of up to 50% for conservative circuit design assumptions (e.g., 3.3 V), and up to 70% for more aggressive assumptions (2.2 V). These savings are in addition to the obvious savings that come from stopping the processor in the idle loop, and powering off the machine all together after extended idle periods. The energy savings depends on the interval between speed adjustments. If it is adjusted at too fine a grain, then less power is saved because CPU usage is bursty. If it is adjusted at too coarse a grain, then the excess cycles built up during a slow interval will adversely affect interactive response. Interestingly, having too low a minimum speed results in less efficient schedules because there is more of a tendency to have excess cycles and therefore the need to speed up to catch up.

3.1.2 Schedulers for real-time systems

Interval based scheduling is simple and easy to implement, but it often incorrectly predicts future workloads and degrades the quality of service. In non-real-time systems, excess cycles left over from the previous interval might be spilled over into later intervals when speed is not fast enough to complete an interval's work. In a real-time system, tasks are specified by the task start time, the computational resources required and the task deadline. The voltage-clock scaling must be carried out under the constraint that no deadline is missed. Optimal voltage is schedule is defined to be one for which all tasks complete on or before deadlines and the total energy consumed is minimized.

Scheduling algorithms for real time systems that minimizes energy consumption while all tasks are guaranteed to complete on or before deadlines were proposed in [12]. This technique is based on the assumption that the timing parameters of each job are known off-line. Two algorithms are given in the paper. The first one takes $O(N^2)$ time (where N is the number of jobs) to find the minimum constant speed needed to complete each job, since constant voltage tends to result in a low power consumption. This minimum constant speed is computed on some intervals, after determining some parameters like *scheduling points, earliest and latest time, job intensity, busy interval* and *essential interval*. The second algorithm, with $O(N^3)$ time complexity, build on the first one and give two results. First, the minimum constant voltage (or speed) needed to complete a set of jobs is obtained. Secondly, a voltage schedule is produced. This algorithm determines the critical intervals. The set of critical intervals and their associated speeds form the voltage schedule. This voltage schedule always saves more energy than the one that applies the

minimum constant speed when the processor is busy while shuts down the processor when it is idle.

This approach to construct a low-energy voltage schedule is a greedy approach since it strives to find the minimum constant speed during any critical interval. It guarantees to result the minimum peak power consumption. However this algorithm may not always produce the minimum-energy schedule. The experimental results obtained from both randomly generated and real-world real-time systems have shown that this voltage schedule algorithm consistently leads to more energy than existing approaches. Furthermore, these algorithms do not limit to only periodic tasks, and can be optimized to find an optimal voltage schedule

In [13], two DVS algorithms on MPEG decoding were proposed. The first algorithm is DVS-DM (*DVS with delay and drop rate minimizing algorithm*), which is a kind of interval based DVS in a sense that it schedules voltage based on previous workload. This algorithm tries to scale the supply voltage according to the delay value and the drop rate. The second algorithm is DVS-PD (*DVS with decoding time prediction*), which determines the voltage not only by previous workload but also by predicted MPEG decoding time. The prediction, in this case, is based on frame size and frame type.

Four MPEG decoders with six sample streams were compared and it was found that DVS-PD shows the best performance with respect to energy consumption and DVS-DM is slightly better that the conventional shutdown algorithm. Outstanding energy saving with DVS-PD is due to higher prediction accuracy of future workload than other approaches. It's also found that energy saving is closely related with average decoding time and fluctuation. With DVS-DM, high fluctuation makes it difficult to predict future workload based on the previous workload only and it results in low efficiency. On the contrary, it's found that that DVS-PD is not much affected by the fluctuation. Instead, performance of DVS-PD in terms of energy consumption depends on the error rate of the predictor, which implies that if decoding time is predicted more accurately, DVS algorithm can be more efficient.

3.2 System-level DPM

To effectively optimize system energy, it is necessary to consider all of the critical components: there is little benefit in optimizing the microprocessor core if other required elements dominate the energy consumption.

The design of the low-power microprocessor system introduced in [14] includes the microprocessor core, data cache, processor bus, and external SRAM. To reduce the energy consumption of the memory system, a highly optimized SRAM design was used, which is 32

data-bits wide, requiring only one device be activated for each access. To alleviate the high pin count problem, data address is multiplexed onto the same bit-lines as the data words.

In [15], system-level power management was studied to save power of subsystems or devices. Examples of devices include I/O controllers, hard disk drives, network interface cards, and displays. Shutting down hard disks and displays is the most widely adopted system-level power management on PCs. [15] discusses the use of DPM techniques specifically for shutting down unused I/O devices. But, changing power states has overhead. Consequently, a device should sleep only if the saved energy justifies the overhead.

Power management policies can be classified into three categories based on the methods to predict whether a device can sleep long enough: (1) *Time-out policies*: assume that after a device is idle for a certain time-out value, it will remain idle for at least T_{be} (*break-even time*, the minimum length of an idle period to save power). This category includes (i) fixed-timeout, (ii) adaptive time-out (ATO) and (iii) device-dependent time-out policies (DDT). An obvious drawback is the energy wasted during this time-out period. (2) *Predictive policies*: predict the length of an ideal period before it starts, eliminating the time-out period. If an idle period is predicted to be longer than the break-even time, the device sleeps right after it's idle. This category includes (i) the L-shape, (ii) the adaptive learning tree (LT) and (iii) the exponential average policies (EA). (3) *Stochastic policies*: model the arrival requests and device power-state changes as stochastic processes, such as Markov processes. This category includes (i) Discrete-time Markov processes (DM), (ii) Time-index semi-Markov models (SM) and (iii) non-stationary requests (NS).

The policies, mentioned above, were implemented using *filter driver*, a device driver inserted between the operating system kernel and another device driver. The filter driver intercepts communications between the operating system and the other driver and can pass, add, delete or change the exchanged messages.

Each policy was graded by six criteria: power, number of shutdowns, shutdown accuracy, interactive performance, memory requirements, and computation requirements. No policy was found to have best grades for all criteria. If a policy aggressively saves power (such as SM), it's likely to issue more shutdown commands and degrade performance. On the other hand, a policy can be conservative in power saving and issue fewer power shutdown. While performance and accuracy improve, these policies consume more power. Finally, the resource requirements are also important. The NS policy has excellent power savings but requires substantial amount of memory.

In [16], an OS-directed power management technique was proposed in order to improve the energy efficiency of sensor nodes using DPM. The basic idea is to shut down devices (CPU, memory, sensor, radio…) when not needed and wakes them up when necessary. A power-aware sensor node model essentially describes the power consumption in different levels of node sleep states. Every component in the node can have different power modes, but also has latency overhead associated with transitioning to that mode. Therefore each node sleep mode is characterized by power consumption and latency overhead. In general a deeper sleep state consumes less power and has a longer wake-up time.

3.3 Cluster System-level DPM

Dynamic Power Management was also used in server clusters, [17], in order to reduce the energy consumption of the whole cluster by coordinating and distributing the work between all available nodes.

Five policies for reducing the energy consumption of server clusters with varying degrees of implementation complexity were presented. The first policy, *Independent Voltage Scaling* (IVS), simply uses voltage scaled processors. Each node independently manages its own power consumption. The second policy also uses DVS but in a coordinated manner between nodes to reduce cluster power consumption. It's called *Coordinated Voltage Scaling* (CVS). The third policy, called vary-on/vary-off (VOVO), turns off server nodes so that only the minimum number of servers required to support the workload are kept alive. Nodes are brought online as and when required. The fourth policy, called *Combined Policy*, combines IVS and VOVO while the fifth uses a combination of CVS and VOVO and is called *Coordinated Policy*.

These policies were evaluated in terms of both their response time and energy savings. The result of the simulation was that IVS, the simplest of all policies in terms of implementation complexity, offers energy savings ranging from 20% to 29%. CVS offers slightly better savings than IVS, but this benefit is probably not sufficient to justify the increased implementation complexity. The energy savings afforded by VOVO are workload dependent. For the fluctuating workloads, VOVO saves more energy than IVS. However for more stable workload, IVS saves more energy than VOVO. Combining DVS with VOVO offers the most energy savings with VOVO-IVS saving more energy then either DVS or VOVO in isolation. VOVO-CVS saves the most energy at the expense of a more complicated implementation. Compared to a cluster that is not power managed, these combined policies save between 33% and 50% of the cluster energy. All five policies can be engineered to keep server response times within acceptable norms.

4 Conclusion

The need for robust power-performance modeling and optimization at all system levels will continue to grow with tomorrow's workload and performance requirements for both high-end and low-end systems. Such models, providing design-time or run-time optimizations or maybe both, will enable designers to make the right choices in defining the future generation of energy-efficient systems.

The successful design and evaluation of power optimization techniques to address this vital issue is invariably tied to the availability of a broad and accurate set of simulation tools. Existing power simulators described in this paper (except for [9]), are mainly targeted for particular hardware components such as CPU or memory systems and do not capture the interaction between different system components. For future works it might be useful to develop a complete system power simulator that models all the system components including I/O devices and the interaction between them, and quantifies the power behavior of both the application and operating system. This simulator should be modular and flexible in a sense that additional modules are accepted and existing one can be replaced or modified. In addition to that, the ability to support dynamic power management techniques like DVS, will give the simulator a more global usability developing or studying any power management technique.

5 References

[1] D.Brooks, P.Bose, S.Schuster, H.Jacobson, P.Kudva, A.Buyuktosunoglu, J.Wellman, V.Zyuban, M.Gupta and P.Cook, "**Power Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors**", IEEE Micron, December 2000, p26-44.

[2] D.Brooks, V.Tiwari, and M.Martonosi, "**Wattch: A Framework for Architectural-Level Power Analysis and Optimizations**", *In Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, June 2000.

[3] W.Ye, N.Vijaykrishan, M.Kandemir, and M.J.Irwin, "**The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool**", *In Proceedings of the Design Automation Conference*, June 2000.

[4] T.Simunic, L.Benini, and G. De Michelli, "**Energy-Efficient Design of Battery-Powered Embedded Systems**", *Int'l Symposium on Low Power Electronics and Design*, 1999.

[5] A. Sinha and A.Chandrakasan, "**JouleTrack – A Web Based Tool for Software Energy Profiling**", *Proc. 38th Design Automation Conference*, June 2001.

[6] T.Cignetti, K.Komarov and C.Ellis, "**Energy Estimation Tools for the Palm™**", *In Proceedings of the ACM MSWiM'2000: Modeling, Analysis and Simulation of Wireless and Mobile Systems*, August 2000.

[7] J.Flinn and M.Satyanarayanan. "**PowerScope: A tool for Profiling the Energy Usage of Mobile Applications**". *In Proceedings of the Workshop on Mobile Computing Systems and Applications (WMCSA)*, February 1999.

[8] F.Chang, K.Farkas and P.Ranganathan, **"Energy-driven Statistical Profiling: Detecting Software Hotspots"**, *In Proceedings of the Workshop on Power Aware Computing Systems*, Feb 2002.

[9] S.Gurumurthi, A.Sivasubramaniam, M.J.Irwin, N.Vijaykrishnan, and M.Kandemir, "**Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach**", *In the Proceedings of the International Symposium on High Performance Computer Architecture (HPCA-8)*, Feb 2002.

[10] M.Weiser, B.Welch, A.Demers and S.Shenker, "**Scheduling for Reduced CPU Energy**", *Usenix Association*, Nov. 1994.

[11] K.Govil, E.Chan and H.Wasserman, "**Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU**", *MobiCom* 1995.

[12] G.Quan and X.Hu, "**Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors**", *Design Automation Conference*, 2001.

[13] D.Son, C.Yu and H.Kim, "**Dynamic Voltage Scaling on MPEG Decoding**", *International Conference on Parallel and Distributed Systems (ICPADS)*, 2001.

[14] T.Pering, T.Burd and R.Brodersen, "**Dynamic Voltage Scaling and the Design of a Low-Power Microprossor System**", *Workshop on Low-Power Microprocessors*, 1998.

[15] Y.Lu and G.De Micheli, "**Comparing System-Level Power Management Policies**", *IEEE Design & Test of Computers*, March 2001, pp10-19.

[16] A.Sinha and A.Chandrakasan, "**Dynamic Power Management in Wireless Sensor Networks**", *IEEE Design & Test of Computers*, March 2001, pp62-74.

[17] E.N.Elnozahy, M.Kistler and R.Rajamony, **"Energy-Efficient Server Clusters"**, *In Proceedings of the Second Workshop on Power Aware Computing Systems*,Feb 2002

[18] D.Burger and T.Austin, "**The SimpleScalar Tool Set, Version 2**", *Tech. Report No. 1342, Computer Sciences Dept. Univ. of Wisconsin*, June1997.