Power Analysis and Optimization Techniques for Energy Efficient Computer Systems

Wissam Chedid[†], Chansu Yu[†] and Ben Lee[‡]

[†]Department of Electrical and Computer Engineering
Cleveland State University
2121 Euclid Avenue, Stilwell Hall 340, Cleveland, OH 44115
wissam@chedid.com, c.yu91@csuohio.edu

[‡]School of Electrical Engineering and Computer Science
Oregon State University
Owen Hall 302, Corvallis, OR 97331
benl@eecs.orst.edu

Abstract

Reducing power consumption has become a major challenge in the design and operation of today's computer systems. This chapter describes different techniques addressing this challenge at different levels of system hardware, such as CPU, memory, and internal interconnection network, as well as at different levels of software components, such as compiler, operating system and user applications. These techniques can be broadly categorized into two types: Design time power analysis versus run-time dynamic power management. Mechanisms in the first category use analytical energy models that are integrated into existing simulators to measure the system's power consumption and thus help engineers to test power-conscious hardware and software during design time. On the other hand, dynamic power management techniques are applied during run-time, and are used to monitor system workload and adapt the system's behavior dynamically to save energy.

Table of contents

Abbreviations

ACPI:	Advanced Configuration and Power Interface
APM:	Advanced Power Management
AVS:	Automatic Voltage Scaler
BIOS:	Basic Input/Output Services
CMOS:	Complementary Metal Oxide Semiconductor
COPPER:	Compiler-controlled continuous Power-Performance
CPU:	Central Processing Unit
CVS:	Coordinated Voltage Scaling
DLS:	Dynamic Link Shutdown
DPM:	Dynamic Power Management
DVSL	Dynamic Voltage Scaling
DVS-DM:	DVS with Delay and Drop rate Minimizing
DVS-PD:	DVS with Predicting Decoding time
Flops:	Floating-point Operation Per Second
GOP:	Group Of Pictures
I/O:	Input/Output
IVS:	Independent Voltage Scaling
JVM:	Java Virtual Machine
LCD:	Liquid-Crystal Display
MPEG:	Moving Pictures Expert Group
OS:	Operating System
PLL:	Phase-Locked Loop
POSE:	Palm Operating System Emulator
RTL:	Register Transfer Language
RWEC:	Remaining Worst-case Execution Cycles
SPEC:	Standard Performance Evaluation Corporation
VOVO:	Varry-On/Varry-Off

1. Introduction

Information processing engines are really seen as just "heat engines."
Mead and Conway, "*Introduction to VLSI Systems*," 1980

Innovations and improvements have long been made in computer and system architectures to essentially increase the computing power truly observing the Moore's Law for more than three decades. Improvements in semiconductor technology make it possible to incorporate millions of transistors on a very small die and to clock them at very high speeds. Architecture and system software technology also offer tremendous performance improvements by exploiting parallelism in a variety of forms. While the demand for even more powerful computers would be hindered by the physics of computational systems such as the limits on voltage and switching speed [39], a more critical and imminent obstacle is the power consumption and the corresponding thermal and reliability concerns [27]. This applies not only to low-end portable systems but also to high-end system designs.

Since portable systems such as laptop computers and cell phones draw power from batteries, reducing power consumption to extend their operating times is one of the most critical product specifications. This is also a challenge for high-end system designers because high power consumption raises temperature, which deteriorates performance and reliability. In some extreme cases, this requires an expensive, separate power facility, as in the *Earth Simulator* [18], which achieves a peak performance of 40 Tflops but dissipates 5 MWatts of power.

This chapter provides a comprehensive survey of power analysis and optimization techniques proposed in the literature. Techniques for power efficient computer systems can be broadly categorized into two types: *Offline power analysis* and *dynamic power management* techniques. *Offline power analysis* techniques are based on analytical *energy models* that are incorporated into existing performance-oriented simulators to obtain power and performance information and help system architects select the best system parameters during design time. *Dynamic power*

management (*DPM*) schemes monitor system workload and adapt the system's behavior to save energy. These techniques are dynamic, run-time schemes operating at different levels of a computer system. They include *Dynamic Voltage Scaling* (*DVS*) schemes that adjust the supply voltage and operating frequency of a processor to save power when it is idle [27, 68]. A similar idea can be applied to I/O devices by monitoring their activities and turning them off or slowing them down when the demand on these devices is low [7, 14, 28]. Another possibility to conserve energy at run-time is when a system has more than one resource of the same kind, which is typically found in parallel and networked cluster systems. In this case, applying a DPM scheme in a coordinated way rather than applying it to individual resources independently can better manage the entire system. For instance, the DVS technique can be extended to a cluster system of multiple nodes by coordinating multiple DVS decisions [19].

This chapter is organized as follows: Section 2 discusses several energy models and the corresponding power analysis and optimization techniques integrated into existing simulation environments. These energy models cover various levels of a system with a varying degree of granularity and accuracy, which includes CPU-level, system-level, and parallel system-level power analysis techniques. Section 3 presents various hardware and software DPM techniques that also differ in granularity as well as accuracy. Fine-grained monitoring and power management is possible at a smaller scale but it may not be feasible at a larger scale because of the corresponding overhead of gathering information and making power-related decisions. Therefore, this section presents the various CPU-level, system-level, and parallel system-level DPM techniques. Section 4 provides a conclusion and discusses possible future research.

2. Power Analysis and Optimization Using Energy Models

Power dissipation has emerged as a major constraint in the design of processors and computer systems. Power optimization, just as with performance, requires careful design at several levels

of the system architecture. The first step toward optimizing power consumption is to understand the sources of energy consumption at different levels. Various energy models have been developed and integrated with existing simulators or measurement tools to provide accurate power estimation, which can be used to optimize the system design.

Section 2.1 describes processor-based energy models that estimate power consumption at cycle- or instruction-level. Section 2.2 discusses system-based energy models that study power consumption of both hardware and software components. Finally, Section 2.3 targets multiprocessor–based or cluster-based energy models. These studies in particular focus on the system interconnect since energy performance of individual processors or nodes can be estimated based on techniques described in Sections 2.1 and 2.2. Table 1 summarizes these energy model-based off-line approaches.

Table 1: Taxonomy of power analysis techniques using energy models.

Type	Level of detail	Energy models	Simulation tools	Section
CPU	Cycle level or RTL	Power density-based or capacitance-based model for cycle-level simulation	*PowerTimer* [9], *Wattch* [10] and *SimplePower* [70]	2.1.1
	Instruction level	Instruction-based energy model with the measurement of instruction counts	Power profiles for *Intel 486DX2*, *Fujitsu SPARClite'934* [65] and *PowerPC* [49]	2.1.2
System	Hardware component level	State-based model (*e.g.*, sleep/doze/ busy) for functional simulation	*POSE* (Palm OS Emulator) [16]	2.2.1
	Software component level	Process-based model with time-driven and energy-driven sampling	Time driven sampling, *PowerScope* [20], and energy driven sampling [12]	2.2.2
	Hardware and software component level	Component-specific energy models for complete system simulation	*SoftWatt* built upon *SimOS* system simulator [27]	2.2.3
Parallel system	Interconnection network architecture level	Bit energy model for bit-level simulation	*Simulink*-based tool [71]	2.3
		Message-based energy model for simulating interconnection network	*Orion*, the simulator for power-performance interconnection networks [67]	2.3

2.1 CPU-level Energy Models

Power consumed by the CPU is a major part of the total power consumption of a computer system and thus has been the main target of power consumption analysis [9, 10, 49, 65, 70]. Several power models have been developed and integrated into existing performance simulators in order to investigate power consumption of CPU either on a functional unit basis or processor as a whole. These analyses are based on two abstraction levels; *cycle-level* (or *register-transfer level*) and *instruction-level* as described in the following two subsections, respectively.

2.1.1 Cycle-level CPU energy model

Energy consumption of a processor can be estimated by using cycle-level architecture simulators. This is done by identifying the active (or busy) microarchitecture-level units or blocks during every execution cycle of the simulated processor [9, 10, 70]. These cycle-by-cycle resource usage statistics can then be used to estimate the power consumption. An energy model describing how each unit or block consumes energy is a key component in any power-aware cycle-level simulators. Figure 1 illustrates a high-level block diagram of power-aware cycle-level simulators.

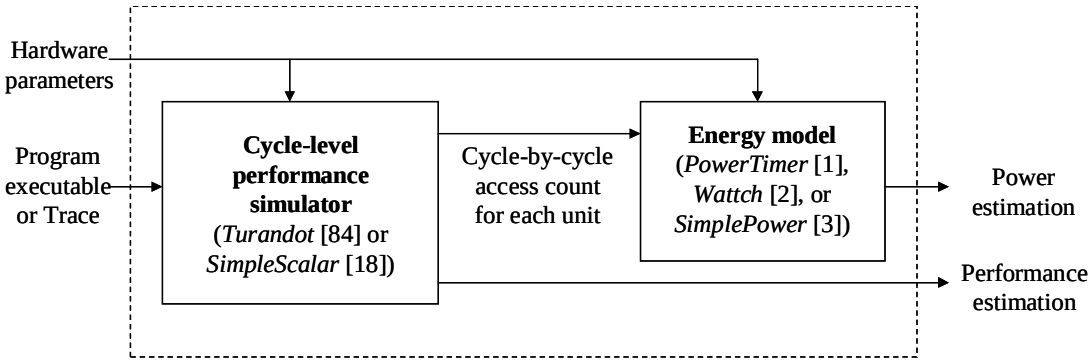


Figure 1: Block diagram of a power-aware, cycle-level simulator.

Brooks *et al.* presented two types of energy models for their *PowerTimer* simulator [9]: (i) Power density-based energy model is used for components when detailed power and area measurements are available; and (ii) analytical energy models are used for the rest of the components in a CPU. Analytical equations formulate the energy characteristics in terms of microarchitecture-level design parameters such as cache size, pipeline length, number of registers, etc. These two types of energy models were used in conjunction with a generic, parameterized, out-of-order superscalar processor simulator called *Turandot* [44]. Using *PowerTimer*, it is possible to study the power-performance trade-offs for different system configurations with varying resource sizes of caches, issue queues, rename registers, and branch predictor tables, which will help in building power-aware microarchitectures.

Wattch [10] and *SimplePower* [70] are two other CPU-level power-monitoring tools based on S*impleScalar* [11], which is the most popular microarchitecture simulator. In *Wattch*, the energy models depend on the internal capacitances of the circuits that make up each unit of the processor. Each modeled unit falls into one of the following four categories: Array structures, memories, combinational logic and wires, and the clocking network. A different power model is used for each category and integrated in the *SimpleScalar* simulator to provide a variety of metrics such as power, performance, energy, and energy-delay product. Table 2 shows the energy expenditure of various components from measurements as well as from the *Wattch* simulator.

Table 2: Comparison of power breakdowns between measurement (Alpha 21264) and analytical energy model in the Wattch simulator [10].

Hardware structure	**Measurement (Alpha 21264)**	**Analytical model (Wattch)**
Caches	16.1%	15.3%
Out-of-order issue logic	19.3%	20.6%
Memory	8.6%	11.7%
Memory management unit	10.8%	11.0%
Floating point execution unit	10.8%	11.0%
Clocking network	34.4%	30.4%

SimplePower, on the other hand, is based on *transition-sensitive energy model*, where each modeled functional unit has its own switch capacitance for every possible input transition [70]. This is then used to calculate the power consumed in a particular functional unit based on the input transition while executing a given instruction. *SimplePower* is used to evaluate the impact of an architectural modification as well as the effect of a high-level compiler optimization technique on system power. Example uses of *SimplePower* include selective gated pipeline technique to reduce the datapath switch capacitance, loop and data transformation to reduce the memory system power, and register relabeling to conserve power on the data buses [70].

2.1.2 Instruction-level CPU energy model

In contrast to the fine-grain *cycle-level* techniques, coarse-grain *instruction-level* power analysis techniques estimate the total energy cost of a program by adding the energy consumed while executing instructions of a program [65, 9]. Instruction-by-instruction energy costs, called *base costs*, can be measured for individual instructions for a target processor. However, there is extra power consumption due to "interaction" between successive instructions caused mainly by pipeline and cache effects. The base costs of individual instructions and the power cost of *inter-instruction effects* are determined based on the experimental procedure using a program containing several instances of the targeted instruction (for base cost measurement) and an alternating sequence of instructions (for inter-instruction effects costs). Table 3 illustrates a subset of the base costs for Intel 486DX2 and Fujitsu SPARClite'934 [65]. A similar study has also been conducted for PowerPC microprocessor [49].

Table 3: Base costs for Intel 486DX2 and Fujitsu SPARClite '394 processors [65]. (Cycles and energy numbers in the table are per-instruction values.)

Intel 486DX2				Fujitsu SPARClite '934			
Instruction	**Current (mA)**	**Cycles**	**Energy (10^{-8} J)**	**Instruction**	**Current (mA)**	**Cycles**	**Energy (10^{-8} J)**
nop	276	1	2.27	nop	198	1	3.26
mov dx,[bx]	428	1	3.53	ld [10],i0	213	1	3.51
mov dx,bx	302	1	2.49	or g0,i0,10	198	1	3.26
mov [bx],dx	522	1	4.30	st i0,[10]	346	2	11.4
add dx,bx	314	1	2.59	add i0,o0,10	199	1	3.28
add dx,[bx]	400	2	6.60	mul g0,r29,r27	198	1	3.26
jmp	373	3	9.23	srl i0,1,10	197	1	3.25

Once the instruction-by-instruction energy model is constructed for a particular processor, the total energy cost, E_P, of any given program, P, is given by:

$$E_P = \sum_i (Base_i * N_i) + \sum_{i,j} (Inter_{i,j} * N_{i,j}) + \sum_k E_k \quad (1)$$

where $Base_i$ is the base cost of instruction i and N_i is the number of executions of instruction i. $Inter_{i,j}$ is the inter-instruction power overhead when instruction i is followed by instruction j, and $N_{i,j}$ is the number of times the (i,j) pair is executed. Finally, E_k is the energy contribution of other inter-instruction effects due to pipeline stalls and cache misses.

2.2 Complete System-level Energy Models

There is little benefit in studying and optimizing only the CPU core if other components have significant effect on or even dominate the energy consumption. Therefore, it is necessary to consider other critical components to reduce the overall system energy. Subsection 2.2.1 discusses the *hardware state-level models*, where the total energy consumption of the entire system is estimated based on the state each device is in or transitioning to/from. Here, it is assumed that each device is capable of switching into one of several power-saving states, such as sleep state, depending on the demand on that particular device [16]. This capability is usually provided in portable systems to extend their lifetimes as longer as possible. *Software-based* approaches presented

in Subsection 2.2.2 identify energy hotspots in applications and operating system procedures and thus allow software programmers to remove bottlenecks or modify the software to be energy-aware. Finally, a *complete system level* simulation tool, which models the hardware components, such as CPU, memory hierarchy, and a low power disk subsystem as well as software components, such as OS and application, is presented in Subsection 2.2.3.

2.2.1 Hardware state-based energy model

Cignetti *et al.* presented a system-wide energy optimization technique with a hardware state-based energy model [16]. This power model encapsulates low-level details of each hardware subsystem by defining a set of power states (*e.g.*, sleep, doze or busy for CPU) for each device. Each power state is characterized by the power consumption of the hardware during the state, which is called *steady state power*. In addition, each transition between states is assigned an energy consumption cost, called *transient energy*. Since transitions between states occur as a result of system calls, the corresponding energy can be measured by keeping track of system calls. The total energy consumed by the system is then determined by adding the power of each device state multiplied by the time spent in that state plus the total energy consumption for all the transitions.

The abovementioned state-based energy model was implemented as an extension to the *Palm OS Emulator* (*POSE*) [48], which is a Windows based application that simulates the functionalities of a Palm device. POSE emulates *Palm OS* and instruction execution of the *Motorola Dragonball* microprocessor [43]. To quantify the power consumption of a device and to provide parameters to the simulator, measurements were taken in order to capture transient energy consumption as well as steady state power consumption as presented in Table 4 [16]. A Palm device from IBM was connected to a power supply with an oscilloscope measuring the voltage across a small resistor. The power consumption of the basic hardware subsystems, such as CPU, LCD, backlight, buttons, pen, and serial link, was measured using measurement programs called *Power* and *Millywatt* [40].

Table 4: Steady state and transient power of a Palm device from IBM. (Steady state power shown is the relative value to the default state; CPU doze, LCD on, backlight off, pen and button up. State transition is caused by system calls, which are shown on the right hand side.)

Steady state power		
Device	**State**	**Power (mW)**
CPU	Busy	104.502
	Idle	0.0
	Sleep	-44.377
LCD	On	0.0
	Off	-20.961
Backlight	On	94.262
	Off	0.0
Button	Pushed	45.796
Pen	On Screen	82.952
	Graffitti	86.029

Transient energy	
System Call	**Transient energy (mJ)**
CPU Sleep	2.025
CPU Wake	11.170
LCD Wake	11.727
Key Sleep	2.974
Pen Open	1.935

2.2.2 Process-based energy model

Since software is the main determinant for the activities of hardware components, such as the processor core, memory system and buses, there is a need for investigating energy-oriented software techniques and their interaction and integration with performance-oriented software design. This subsection presents process-based power measurement techniques for system optimization [12, 20]. Using specially designed monitoring tools, these measurement-based techniques target the power consumption of the entire system and try to point out the hotspots in applications and operating system procedures. It is noted that these techniques are process-based in the sense that they assume different processes consume different amount of energy not only because they execute for different amount of time or different number of instructions but also because they use different sets of resources in different sequences.

In *PowerScope* [20], a *time-driven statistical sampler* is used to determine what fraction of the total energy is consumed, during a certain time period, due to specific processes in the system. This technique can be further extended to determine the energy consumption of different procedures within a process. By providing such a fine-grained feedback, *PowerScope* helps focus

on those system components responsible for the bulk of energy consumption. Chang *et al.* presented a similar tool but it is based on *energy-driven statistical sampling*, which uses energy consumption to drive sample collection [12]. The *multimeter* [20] (or the *energy counter* [12]) monitors the power consumption of the system and the software under test by generating an interrupt for each time interval [20] (or each energy quanta [12]). This interrupt will prompt the system to record the process ID of the currently running process as well as to collect a current [20] (or energy [12]) sample. After the experiment, the collected data, *i.e.* process IDs and current/energy sample, is analyzed offline to match the processes with the energy samples to create the energy profile.

The result from this study showed that a non-trivial amount of energy was spent by the operating system compared to other user processes. In addition, there are often significant differences between time-driven and energy-driven profiles and therefore, it is necessary to carefully combine both sampling methods to obtain more accurate energy profile information.

2.2.3 Component-specific energy model

Power profiling techniques mentioned above provide energy cost for executing a certain program but without understanding the overall system behaviors in sufficient detail to capture the interactions among all the system components. A complete system power simulator, *SoftWatt* [27] overcomes this problem by modeling hardware components such as CPU, memory hierarchy, and disk subsystem, and quantifying the power behavior of both application software and operating system. *SoftWatt* was built on top of *SimOS* infrastructure [55], which provides detailed simulation of both the hardware and software including the *IRIX* operating system [30]. In order to capture the complete system power behavior, *SoftWatt* integrates different analytical power models available from other studies into the different hardware components of *SimOS*. The modeled units in *Softwatt* include cache-structure, datapath, clock generation and distribution network, memory, and hard drive.

Experience with *Softwatt* running *JVM98* benchmark suite [59] from *SPEC* (*Standard Performance Evaluation Corporation*) [62] emphasized the importance of a complete system simulation to analyze the power impact of both architecture and OS on the execution of applications. From a system hardware perspective, the disk is the single largest power consumer of the entire system. However, with the adoption of a low-power disk, the power hotspot was shifted to the CPU clock distribution and generation network (similar results are shown in Table 2). Also, the cache subsystem was found to consume more power than the processor core. From the software point of view, the user mode consumes more power than the kernel mode. However, certain kernel services are called so frequently that they accounted for significant energy consumption in the processor and memory hierarchy. Thus, taking into account the energy consumption of the kernel code is critical for reducing the overall energy cost. Finally, transitioning the CPU and memory subsystem to a low-power mode or even halting the processor when executing an idle process can considerably reduce power consumption.

2.3 Interconnect-level Energy Models in Parallel Systems

After presenting energy models at the CPU-level (Section 2.1) and the system-level (Section 2.2), this section describes energy models at the parallel system-level with the focus on interconnection networks. With the ever-increasing demand for computing power, processors are becoming more and more interconnected to create large clusters of computers communicating through interconnection networks. Wang *et al.* showed that the power consumption of these communication components is becoming more critical, especially with increase in network bandwidth and capacity to the gigabit and terabit domains [67]. Thus, power analysis in this area usually targets the building blocks inside a network router and a switch fabric.

Bit energy model [71] considers the energy consumed for each bit, moving inside the switch fabric from the input to the output ports, as the summation of the bit energy consumed on each of the following three components; (i) the internal node switches that direct a packet from

one intermediate stage to the next until it reaches the destination port; (ii) the internal buffer queues that store packets with lower priorities when contention occurs; and (iii) the interconnect wires that dissipate power when the bit transmitted on the wire flips polarity from the previous bit. Different models were employed for each one of these components based on their characteristics. For example, the bit energy of a node switch is state-dependent; it depends on the presence or absence of packets on other input ports. On the other hand, power consumption of the internal buffer can be expressed as the sum of data access energy (read and write) and the memory refreshing operation. Finally, the bit energy of interconnect wires depends on the wire capacitance, length, and coupling between adjacent wires. The bit energy model was incorporated into a *Simulink* [56] based bit-level simulation platform to trace the dataflow of every packet in the network to summarize the total energy consumption in the interconnect.

As opposed to the bit-level approach mentioned above, an architecture-level network power-performance simulator, *Orion*, was presented in [67]. Orion models an interconnection network as comprising of message generating (such as sources), transporting (router buffers, crossbars, arbiters, and link components), and consuming (sinks) agents. Each of these agents is a building block of the interconnection network, and is represented by an architecture-level energy model. This energy model is based on the switch capacitance of each component including both gate and wire capacitances. These capacitance equations are combined with the switching activity estimation to compute the energy consumption per component operation. Orion can be used to plug-and-play router and link components to form different network fabric architectures, run varying communication workloads, and study their impact on overall network power and performance.

3. Dynamic Power Management (DPM) Techniques

While the simulation and measurement techniques described in Section 2 aim to optimize power performance at design time, DPM techniques target energy consumption reduction at run-time by selectively turning off or slowing down components when the systems is idle or serving light workloads. As in Section 2, DPM techniques are applied in different ways and at different levels. For example, *Dynamic Voltage Scaling* (*DVS*) technique operates at the CPU-level and changes processor's supply voltage and operating frequency at run-time as a method of power management [68]. A similar technique, called *Dynamic Link Shutdown* (*DLS*), operates at the interconnect-level and puts communication switches in a cluster system into a low-power mode to save energy [32]. DPM techniques can also be used for shutting down idle I/O devices [49], or even nodes of server clusters [19, 50].

As summarized in Table 5, this section discusses DPM techniques that are classified based on the implementation level. Section 3.1 discusses DPM techniques applied at the *CPU-level*. In Section 3.2, *system-level* DPM approaches that consider other system components (memory, hard drive, I/O devices, display, etc.) than CPU are discussed. Finally, Section 3.3 presents DPM techniques proposed for *parallel systems*, where multiple nodes collaborate to save the overall power while collectively performing a given parallel task.

Table 5: Taxonomy of dynamic power management techniques.

Type	Implementation level	Monitoring mechanism	Control mechanism	Section
CPU	CPU-level	Monitor internal bus activity to reduce switching activity	Different encoding schemes [29, 61, 69], compiler-based scheduling [31, 63, 66]	3.1.1
		Monitor CPU instruction in execution to control clock supply to each component	Clock gating for CPU components [21, 26]	3.1.2
		Monitor CPU workload to adjust supply voltage to CPU	DVS with interval-based or history-based scheduler [25, 52, 58, 68], compiler-based scheduler [4, 5, 22, 54]	3.1.3
System	Hardware device-based	Monitor device activities to shut it or slow it down	Timeout, predictive or stochastic policies [7, 8, 14, 15, 17, 24, 28, 51, 57, 60]	3.2.1
	Software-based	Monitor device activity via application or system software to shut it or slow it down	Prediction of future utilization of device [24, 35, 36, 38], ACPI [1, 2, 47]	3.2.2
Parallel system	Hardware-based	Monitor multiple CPU's workloads to cooperatively adjust supply voltages	CVS (Coordinated DVS) [19]	3.3.1
		Monitor switch/router activity to rearrange connectivity or put into reduced power mode	History-based DVS on switch/router [53], Dynamic Link Shutdown [32]	3.3.1
	Software-based	Monitor synchronization activities to power down spinning nodes	Thrifty barrier [34]	3.3.2
		Monitor workload distribution to shut off some nodes	Load unbalancing [50]	3.3.3

3.2 CPU-level DPM

The intuition behind power saving at the CPU-level comes from the basic energy consumption characteristics of digital static CMOS circuits, which is given by

$$E \propto C_{eff} V^2 f_{CLK} \quad (2)$$

where C_{eff} is the effective switching capacitance of the operation, V is the supply voltage, and f_{CLK} is the clock frequency [25]. The DPM techniques presented in this section reduce the power consumption by targeting one or more of these parameters. Subsection 3.1.1 discusses techniques to reduce the switching activity of the processor, mainly at the datapath and buses. In Subsection 3.1.2, *clock gating* techniques are discussed, which reduce power consumption by turning off the

idle component's clock, *i.e.* $f_{CLK} = 0$. Finally, Subsection 3.1.3 presents one of the most promising, and also the most complicated, CPU-level DPM technique based on DVS. DVS scales both V and f_{CLK} to serve the processor workload with the minimum required power. If applied properly, DVS allows substantial energy saving without affecting performance.

3.1.1 Reducing switching activity

As discussed earlier, reducing switching activity plays a major role in reducing power consumption. A number of such optimization techniques have been proposed to reduce switching activity of internal buses [29, 61, 69] and functional units [31, 63, 66] of a processor. In case of buses, energy is consumed when wires change states (between 0 and 1). Different techniques are used to reduce the switching activity on buses by reducing the number of wire transitions. Stan and Burleson proposed *bus-invert coding* where the bus value is inverted when more than half the wires are changing state [61]. In other words, when the new value to be transmitted on the bus differs by more than half of its bits from the previous value, then all the bits are inverted before transmission. This reduces the number of state changes on the wire, and thus, save energy.

Henkel and Lekatsas proposed a more complicated approach where cache tables are used on the sending and receiving sides of the channel to further reduce transitions [29]. That is, when a value "hit" is observed at the input of the channel, the system will only send the index of the cache entry instead of the whole data value, which will reduce the number of transitions. Finally, Wen *et al.* used *bus transcoding* to reduce bus traffic and thus power based on data compression on bus wires [69]. As an enhancement to this technique, *transition coding* was also proposed where the encoding of data represents the wire changes rather than the absolute value, which simplifies the energy optimization problem.

On the other hand, the processor's switching activity can also be reduced by using power-aware compiler techniques. Although applied at compile time, these are considered as DPM techniques because their effect is closely tied to the system's run-time behavior. For example, in

instruction scheduling technique [63, 66], instructions are reordered to reduce the switching activity between successive instructions. More specifically, it minimizes the switching activity of a data bus between the on-chip cache and main memory when instruction cache misses occur [66]. *Cold scheduling* [63] prioritizes the selection of the next instruction to execute based on the energy cost of placing that instruction into the schedule. Another compiler based technique called *register assignment* [31] focuses on reducing the switching activity on the bus by re-labeling the register fields of the compiler-generated instructions. A simulator, such as *SimplePower* [70], is used to parameterize the compiler with sample traces. In other words, it records the transition frequencies between register labels in the instructions executed in consecutive cycles and this information is then used to obtain a better encodings for the registers such that the switching activity and consequently the energy consumption on the bus is reduced.

3.1.2 Clock gating

Clock gating involves freezing the clock of an idle component. Energy is saved because no signal or data will propagate in these frozen units. Clock gating is widely used because it is conceptually simple; the clock can be restarted by simply de-asserting the clock-freezing signal. Therefore, only a small overhead in terms of additional circuitry is needed, and the component can transit from an idle to an active state in only a few cycles. This technique has been implemented in several commercial processors such as Alpha 21264 [26] and PowerPC 603 [21]. The Alpha 21264 uses a hierarchical clocking architecture with gated clocks. Depending on the instruction to be executed, each CPU unit (*e.g.*, floating point unit) is able to freeze the clock to its subcomponents (*e.g.*, adder, divider and multiplier in floating point unit).

The PowerPC 603 processor supports several sleep modes based on clock gating. For this purpose, it has two types of clock controllers: *global* and *local*. Clocks to some components are globally controlled while others are locally controlled. For example, consider *PLL* (*Phase Locked Loop*) that acts mainly as a frequency stabilizer and does not depend on global clock.

Even though clocks to all units are globally disabled and the processor is in sleep state, the PLL can continue to function which makes a quick wake-up (within ten clock cycles) possible. On the other hand, if the PLL is also turned off, maximum power saving would be achieved but the wake-up time could be as long as 100μs, to allow the PLL to relock to the external clock.

3.1.3 Dynamic Voltage Scaling (DVS)

In contrast to clock gating, which can only be applied to idle components, DVS targets components that are in active state, but serving a light workload. It has been proposed as a mean for a processor to deliver high performance when required, while significantly reducing power consumption during low workload periods. The advantage of DVS can be observed from the power consumption characteristics of digital static CMOS circuits (2) and the clock frequency equation:

$$delay \propto \frac{V}{(V - V_k)^{\alpha}} \text{ and } f_{CLK} \propto \frac{(V - V_k)^{\alpha}}{V} \quad (3)$$

where V is the supply voltage, and f_{CLK} is the clock frequency. α ranges from 1 to 2, and V_k depends on threshold voltage at which *velocity saturation*[1] occurs [25].

Decreasing the power supply voltage would reduce power consumption quadratically as shown in Equation (2). However, this would create a higher propagation delay and at the same time force a reduction in clock frequency as shown in Equation (3). While it is generally desirable to have the frequency set as high as possible for faster instruction execution, the clock frequency and supply voltage can be reduced for some tasks where maximum execution speed is not required. Since processor activity is variable, there are idle periods when no useful work is being performed and DVS can be used to eliminate these power-wasting idle times by lowering the processor's voltage and frequency.

[1] Velocity saturation is related to the semiconductor voltage threshold after which saturation occurs and the transistor's behavior becomes non-linear.

In order to clearly show the advantage of DVS techniques, Figure 2 compares DVS with the simple On/Off scheme, where the processor simply shuts down when it is idle (during time 2~4, 5~7 and 8.5~11 in the figure). DVS reduces the voltage and frequency, spreading the workload to a longer period, but more than quadratically reducing energy consumption. A quick calculation from Figure 2 shows about 82% reduction in power based on Equation (2) because E_{DVS} / $E_{On/Off}$ = $(4\times(0.5)^3 + 3\times(0.33)^3 + 4\times(0.375)^3) / (2\times1^3 + 1\times1^3 + (1.5)\times1^3) = 0.82/4.5 = 0.18$. Note that each task workload, which is represented by the area inside the rectangle in Figure 2, remains the same for both the simple On/Off and DVS mechanisms.

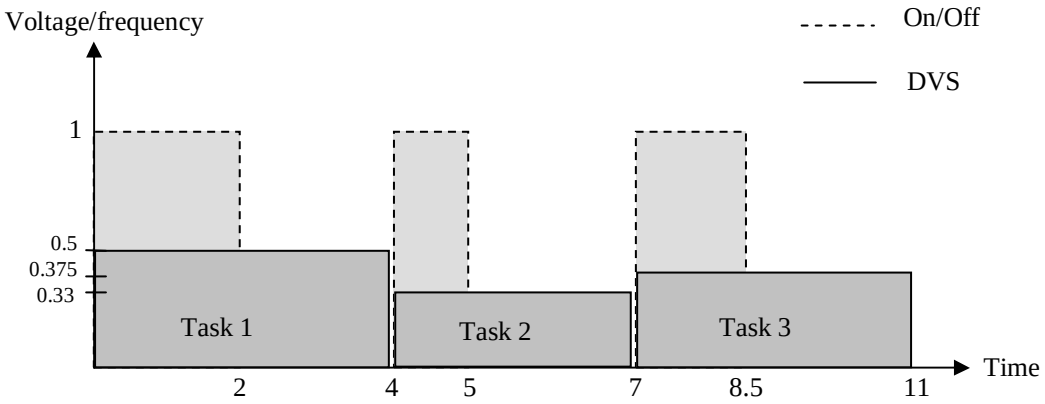


Figure 2: Voltage scheduling graph with On/Off and DVS mechanisms.

Current custom and commercial CMOS chips are capable of operating reliably over a range of supply voltages [46, 64] and there are a number of commercially available processors that support DVS mechanisms. Table 6 shows the Mobile Intel Pentium III processor with 11 frequency levels and 6 voltage levels with two performance modes: *Maximum performance* mode and *battery optimized performance* mode [41]. The maximum performance mode is designed to provide the best performance while the battery optimized performance mode provides the balance between performance and battery lifetime. *Crusoe* processor from Transmeta, Inc. also has variable voltage and frequency as presented in Table 7 [33].

Table 6: Clock frequency versus supply voltage for the Mobile Intel Pentium III processor [41].

Maximum performance mode			Battery optimized mode		
Frequency (MHz)	**Voltage (V)**	**Max. power consumption (Watt)**	**Frequency (MHz)**	**Voltage (V)**	**Max. power consumption (Watt)**
500	1.10	8.1	300	.975	4.5
600	1.10	9.7	300	.975	4.5
600	1.35	14.4	500	1.10	8.1
600	1.60	20.0	500	1.35	12.2
650	1.60	21.5	500	1.35	12.2
700	1.60	23.0	550	1.35	13.2
750	1.35	17.2	500	1.10	8.1
750	1.60	24.6	550	1.35	13.2
800	1.60	25.9	650	1.35	15.1
850	1.60	27.5	700	1.35	16.1
900	1.70	30.7	700	1.35	16.1
1000	1.70	34.0	700	1.35	16.1

Table 7: Clock frequency versus supply voltage for the Transmeta Crusoe processor [33].

Frequency (MHz)	**Voltage (V)**	**Power consumption (Watt)**
667	1.6	5.3
600	1.5	4.2
533	1.35	3.0
400	1.225	1.9
300	1.2	1.3

The main challenge in applying DVS is to know when and how to scale the voltage and frequency. In the following discussion, three different voltage schedulers are presented: *Interval-based*, *inter-task*, and *intra-task* scheduler. Interval-based scheduler is a time-based voltage scheduler that predicts the future workload using the workload history. Inter-task and intra-task schedulers target real-time applications with deadlines to meet for tasks. Inter-task scheduler changes speed at each task boundary, while intra-task scheduler changes speed within a single task with the help from compilers. Inter-task approaches make use of a prior knowledge of the application to produce predictions for the given task, while intra-task approaches try to take ad-

vantage of slack time that results from the difference in program execution path caused by conditional statements.

Interval-based scheduler

Interval-based voltage schedulers [25, 68] divide time into uniform length intervals and analyze CPU utilization of the previous intervals to determine the voltage/frequency of the next interval. Govil *et al.* discussed and compared seven such algorithms [25]: (i) PAST uses the recent past as a predictor of the future. (ii) FLAT simply tries to smooth the processor speed to a global average. (iii) LONG_SHORT attempts to find a golden mean between the most recent behavior and a more long-term average. (iv) AGED_AVERAGES employs an exponential-smoothing method, attempting to predict via a weighted average. (v) CYCLE is a more sophisticated prediction algorithm that tries to take advantage of previous *run_percent* values that have cyclical behavior, where *run_percent* is the fraction of cycles in an interval during which the CPU is active. (vi) PATTERN is a generalized form of CYCLE that attempts to identify the most recent *run_percent* values as a repeating pattern. (vii) PEAK is a more specialized version of PATTERN and uses the following heuristics based on observation on narrow peaks: Increasing *run_percents* would fall but decreasing *run_percents* would continue falling [25].

According to their simulation studies, simple algorithms based on rational smoothing rather than complicated prediction schemes showed better performance. Their study also shows that further possibilities exist by improving predictions, such as sorting past information by process-type or providing useful information by applications [25].

Inter-task techniques for real-time applications

Interval-based scheduler is simple and easy to implement but it often incorrectly predicts future workloads and degrades the quality of service. In non-real-time applications, unfinished task from the previous interval would be completed in later intervals and does not cause any serious

problems. However, in real-time applications, tasks are specified by the task start time, the computational resources required, and the task deadline. Therefore, the voltage/frequency scaling must be carried out under the constraint that no deadlines are missed. An optimal schedule is defined to be the one for which all tasks complete on or before deadlines and the total energy consumed is minimized.

For a set of tasks with the given timing parameters, such as deadlines, constructing the optimal voltage schedule requires super-linear algorithmic complexity. One simple heuristic algorithm is to identify the task with the earliest deadline and find the minimum constant speed needed to complete the task within the time interval before deadline. Repeating the same procedure for all tasks provides a voltage schedule. Quan and Fu suggested a more efficient inter-task scheduling algorithm for real-time applications [52]. This approach tries to find the critical intervals using the given timing parameters, such as start times and deadlines, which can be bottlenecks in executing a set of tasks. Then, a voltage schedule is produced for the set of critical intervals, and a complete low-energy voltage schedule is constructed based on the minimum constant speed found during any critical interval. Although this greedy approach guarantees minimum peak power consumption, it may not always produce the minimum-energy schedule.

Another inter-task DVS technique has been proposed for a specific real-time application, MPEG player [13, 58]. The task here is to decode an MPEG frame or a *group of pictures* (*GOP*) [45]. Since different frames require an order of different computational overhead for decoding, it is more beneficial to change the supply voltage and operating frequency depending on frames rather than GOP. The main difficulty is to predict the next workload (*e.g.*, decoding the next frame) in order to assign a proper voltage and frequency setting. If the next workload (frame decoding time) is underestimated, a voltage/frequency will be assigned that is lower than required, and the job will not meet its deadline causing either jitters or frames to be dropped and the video quality will degrade. On the other hand, if the next workload is overestimated, a volt-

age/frequency that is higher than required will be assigned, leading to more power consumption than necessary.

Son *et al.* proposed two heuristic DVS algorithms for MPEG decoding [58]: *DVS-DM* (*DVS with delay and drop rate minimizing algorithm*) and *DVS-PD* (*DVS with decoding time prediction*). DVS-DM is an interval-based DVS in the sense that it schedules voltage at every GOP boundary based on parameters (mainly delay and drop rate) obtained from previous decoding history. DVS-PD determines the voltage based on information from the next GOP (like frame sizes and frame types) as well as previous history. Since frames exhibit different characteristics depending on the frame type, DVS-PD offers higher prediction accuracy for future workload compared to DVS-DM [58].

Chedid proposed another set of techniques for power aware MPEG decoding [13]: *regression*, *range-avg* and *range-max*. The *regression* technique is based on the observation that the frame-size/decoding-time distribution follows a linear regression model [6] with high accuracy as shown in Figure 3. The regression line is built dynamically at run-time by calculating the slope of the frame-size/decoding-time relationship based on past history. The other two techniques, *range-avg* and *range-max*, alleviate the computational overhead found in the regression algorithm. These approaches divide the decoding-time/frame-size distribution into several ranges as in Figure 3 and make estimation decision based on the average decoding time (*range-avg*) or the maximum decoding time (*range-max*) in each range. The accuracy of these two techniques is only slightly worse than regression, but has the advantages of lower complexity and being able to dynamically increase or decrease the range size in order to better respond to any system requirement such as more power reduction or better video quality [13].

Table 8 summarizes the different inter-task DVS techniques for MPEG decoding discussed in the previous paragraphs.

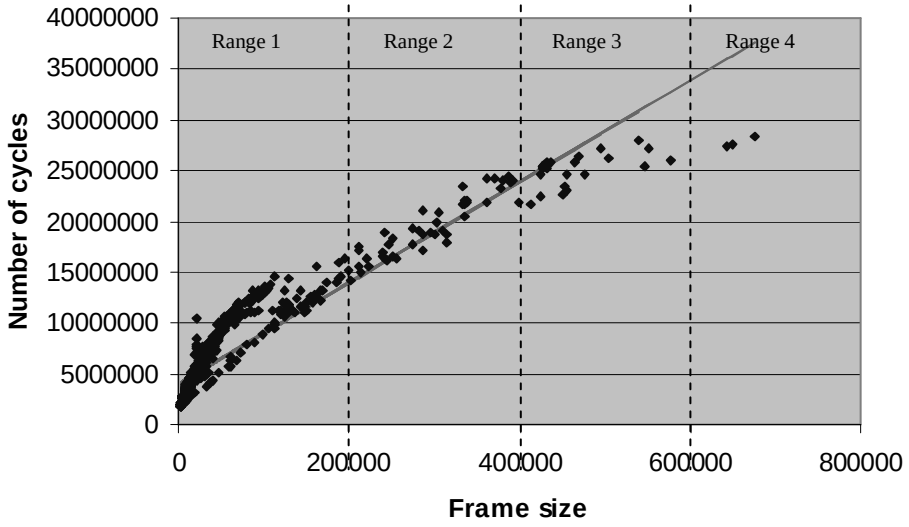


Figure 3: Decode time as a function of frame size (based on the movie, "Undersiege").

Table 8: Inter-task DVS techniques for a real-time application (MPEG player).

Technique	Implementation level	Method used to predict future workload	Advantages	Disadvantages
DVS-DM	GOP (Group of Pictures)	Previous history of delay and drop rate	Easy to implement	Inaccurate if decoding workload fluctuates
DVS-PD	GOP (Group of Pictures)	Weighted average of previous history and next GOP information	More accurate and less vulnerable to fluctuations than *DVS-DM*	Vulnerable to fluctuations between frames within each GOP
Regression	Picture frame	Dynamic regression of previous history and next frame information	Highly accurate prediction	Computationally expensive
Range-avg	Picture frame	Average workload of past picture frames with similar frame type and size	Easy to implement and flexible in balancing between power saving and video quality	Less accurate than *Regression*
Range-max	Picture frame	Maximum workload of past picture frames with similar frame type and size	Easy to implement and more flexible than *Range-avg*	Less accurate than *Regression* and *Range-avg*

Intra-task techniques for real-time applications

As opposed to the inter-task DVS techniques mentioned above, where voltage/frequency changes occur between consecutive tasks, intra-task DVS techniques are applied during the execution of a task with the help of a power-aware compiler. The compiler identifies different possible execu-

tion paths within a task, each requiring a different amount of work and thus different voltage/frequency setting. Consider an example of a real-time task and its flow graph in Figure 4. In Figure 4(b), each node represents a basic block, B_i, of this task and the number in each node denotes the number of execution cycles required to complete the block. The total number of cycles varies for the same task depending on the chosen path and the resultant slack time is the target of optimization in the following intra-task techniques.

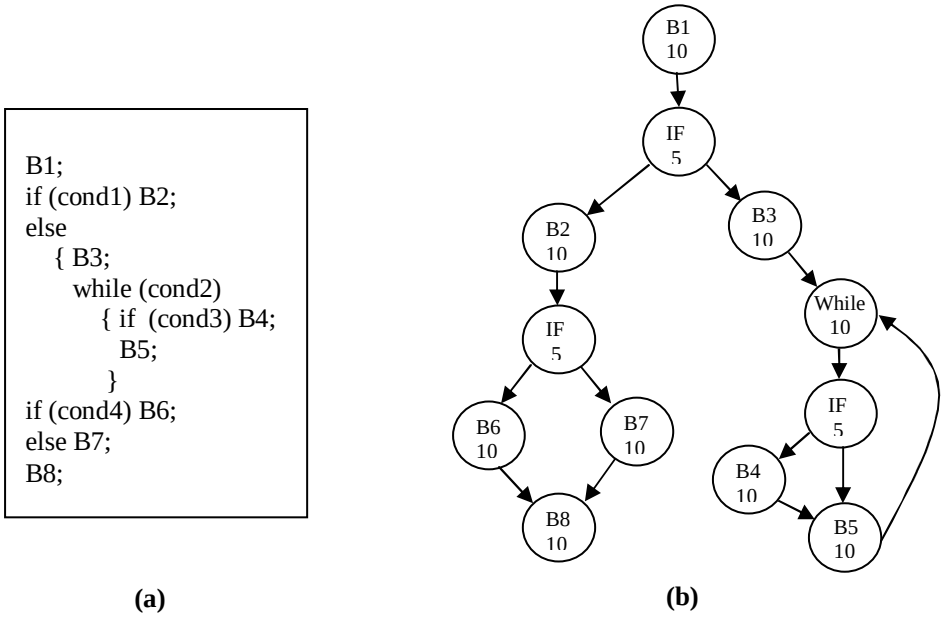


Figure 4: Intra-task paths. (a) Example program and (b) its flow graph (each circle representing a basic block of a task and the number representing the cycles to execute the block).

Azevedo *et al.* introduced an intra-task DVS technique using *program checkpoints* under compiler control [5]. Checkpoints indicate places in a program where the processor voltage/frequency should be re-calculated and scaled. The program is profiled, using a representative input data set, and information about minimum/maximum energy dissipated and cycle count between checkpoints is collected. This information is used in a run-time voltage scheduler to adjust the voltage in an energy efficient way, while meeting the deadline.

Similarly, Shin and Kim proposed a compiler-based conversion tool, called *Automatic Voltage Scaler* (*AVS*), that converts DVS-unaware programs into DVS-aware ones [54]. The compiler profiles a program during compile-time and annotates the *Remaining Worst-case Execution Cycles* (*RWEC*) information, which represents the remaining worst-case execution cycles among all the execution paths that start from each corresponding checkpoint. It automates the development of real-time power-aware programs on a variable-voltage processor in a way completely transparent to software developers.

In the previously discussed approaches, voltage/frequency scaling must be computed and executed at every checkpoint, which may introduce an uncontrollable overhead at run-time. Ghazaleh *et al*. reported a similar compiler-based approach but requires collaboration between the compiler and the operating system [22]. As before, the compiler annotates the checkpoints with the RWEC temporal information. During program execution, the operating system periodically adapts the processor's voltage and frequency based on this temporal information. Therefore, this approach separates the checkpoints into two categories: The first one is only used to compute the temporal information and adjust the dynamic run-time information. The second one is used by the OS (which has more information on the overall application behavior) to execute the voltage/frequency change. This approach relies on the strengths of both the compiler and OS to obtain fine-grain information about an application's execution to optimally apply DVS.

COPPER (*Compiler-controlled continuous Power-Performance*) [4] is another compiler-based approach that also relies on the characteristics of the microarchitecture to optimize the power performance of the application. Among many possibilities, it focuses on combining dynamic register file reconfiguration with voltage/frequency scaling. During compile time, different versions of the given program code are produced under varying architectural parameters, mainly the number of available registers, and the corresponding power profiles are evaluated using energy simulator such as Wattch presented in Subsection 2.1.1. Since running a code version compiled for less number of registers may lead to lower energy consumption but higher execution

delay, it is possible to tradeoff between the average power consumption and the execution time with code versioning. The run-time system selects a code version to help achieve performance goals within a given energy constraints.

3.2 Complete system-level DPM

As discussed before, the CPU does not dominate the power consumption of the entire system. Other system components, such as disk drives and displays, have a much larger contribution. Therefore, it is necessary to consider all of the critical components of the system to effectively optimize power. A well-known system-level power management technique is shutting down hard drives and displays when they are idle. A similar idea can also be applied to other I/O devices to save energy. However, changing power states of hardware components incurs not only time delay but also energy overhead. Consequently, a device should be put to sleep only if the energy saved justifies the overhead. Thus, the main challenge in successfully applying this technique is to know when to shut down the devices and to wake them up.

A straightforward method is to have individual devices make such decisions by monitoring their own utilization. One clear advantage of this device-based scheme (Subsection 3.2.1) is transparency, *i.e.*, energy saving is achieved without involving or changing application or system software. On the contrary, this scheme may perform poorly because it is unaware of the tasks requesting the service of the device. Software-based DPM techniques (Subsection 3.2.2) have been proposed to alleviate this problem. Application or system software takes full responsibility on power-related decisions assuming that devices can operate in several low power modes using control interfaces such as *Advanced Configuration and Power Interface* (*ACPI*) [2].

3.2.1 Hardware device-based DPM policies

Hardware device-based policies observe hardware activities and workloads of the target device and change power states accordingly. They are usually implemented in hardware or device drivers without direct interaction with application or system software as illustrated in Figure 5. Based on prediction mechanisms for future device usage, these methods can be classified into three categories: *Time-out*, *Predictive*, and *Stochastic* policies [7, 37].

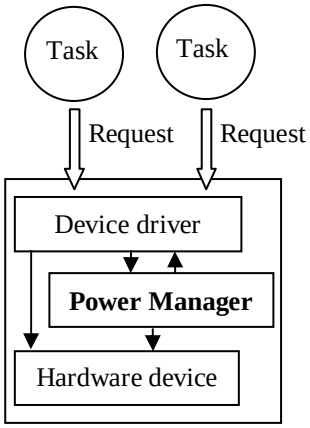


Figure 5: Hardware device-based DPM.

For time-out policies, *break-even time*, T_{BE}, is defined as the minimum time length of an idle period during which shutting down a particular device will save power. When transition power, P_{TR}, and transition time, T_{TR}, (required when changing power states of the device) are negligible, T_{BE} is zero because there is no delay and energy overhead for shutting down and waking up the device. In practice, T_{BE} is calculated based on P_{TR} and T_{TR} as in Figure 6. Time-out policies work as follows: When an idle period begins, a timer is started with a predefined duration $T_{timeout}$, which is usually set as a certain fraction of T_{BE}. If the device remains idle after $T_{timeout}$, then the power manager makes the transition to the low-power or off state because it assumes that the device will remain idle for at least another T_{BE} seconds. These policies are relatively easy to implement but they have two major drawbacks. First, a large amount of energy is wasted waiting for the timeout to expire, during which the device is idle but still fully powered. Second, there is

always a performance penalty to wakeup devices upon receiving a request signal. Devices typically have a long wakeup time, and thus the request to wake up may incur significant delays.

T_{TR}: transition time required to enter ($T_{On,Off}$) and exit ($T_{Off,On}$) the inactive state
P_{TR}: transition power
P_{On}, P_{Off}: power when device is On and Off
T_{BE}: break-even time, the minimum length of an idle period during which shutting down the device will save power.

$$T_{TR} = T_{On,Off} + T_{Off,On}$$
$$P_{TR} = (T_{On,Off}\, P_{On,Off} + T_{Off,On}\, P_{Off,On}) / T_{TR}$$

$$T_{BE} = \begin{cases} T_{TR} & \text{if } P_{TR} \le P_{On} \\ T_{TR} + T_{TR}\,(P_{TR} - P_{On}) / (P_{On} - P_{Off}) & \text{if } P_{TR} > P_{On} \end{cases}$$

where "$T_{TR}\,(P_{TR} - P_{On}) / (P_{On} - P_{Off})$" represents the additional time needed to spend in the Off state to compensate the excess power consumed during state transition.

Figure 6: Calculation of break-even time, T_{BE} [7].

Predictive policies counter the drawbacks of the time-out policies using techniques such as *predictive shutdown* [15, 17, 24, 60] and *predictive wakeup* [28]. The predictive shutdown policy eliminates the time-out period by predicting the length of an idle period beforehand. Srivastava *et al.* suggested that the length of an idle period can be predicted by the length of the previous busy period [60]. Chung *et al.* observed the pattern of idle periods, and then the length of the current idle period is predicted by matching the current sequence that led to this idle period with the observed history of idle periods [15]. In [17, 24], researchers suggested dynamically adjusting $T_{timeout}$ based on whether the previous predictions were correct or not. The predictive wakeup policy reduces the performance penalty by waking up the device on time so that it becomes ready when the next request arrives. Hwang *et al.* employed the *exponential-average* (weighted-average with exponential weight values) approach to predict the wake-up time based on past history [28]. This policy may increase power consumption but will decrease the delay for serving the first request after an idle period.

One of the shortcomings of predictive policies is that they assume a deterministic arrival of device requests. *Stochastic policies* model the arrival of requests and device power-state changes as stochastic processes, *e.g.*, *Markov processes*. Benini *et al.* modeled the arrival of I/O requests using stationary discrete-time Markov processes [8]. This model was used to achieve optimal energy saving by shutting down and waking up a device in the most efficient way in terms of energy as well as performance. In this model, time is divided into small intervals with the assumption that the system can only change its state at the beginning of a time interval. Chung *et al.* extended the model by considering non-stationary processes [14]. They pre-computed the optimal schedule for different I/O request patterns, and at run-time these schedules are used to more accurately estimate the next I/O request time.

However, for discrete-time Markov models, the power manager needs to send control signals to the components every time interval, which may result in heavy signal traffic and therefore more power dissipation. Qiu *et al.* used continuous-time Markov models to help prevent this "periodic evaluation" and instead uses event-triggered evaluation [51]. They consider both request arrival and request service events, upon which the system determines whether or not to shut down a device. Finally, Simunic *et al.* suggested adding timeout to continuous-time Markov models so that a device would be shut down if it has been continuously idle for a predefined timeout duration [57]. In general, stochastic policies provide better performance than predictive and time-out policies. In addition, they are capable of managing multiple power states, making decisions not only *when* to perform state transition but also *which* transition should be made. The main disadvantage of these policies is that they require offline preprocessing and are more complicated to implement. For a detailed comparison of abovementioned device-based DPM schemes, please refer to [7, 37].

3.2.2 Software-based DPM policies

While hardware device-based power management policies can optimize energy-performance of individual devices, they do not consider system-wide energy consumption due to the absence of global information. Therefore, software-based DPM policies have been proposed to handle system-level power management. *Advanced Configuration and Power Interface* (*ACPI*) [2], proposed as an industrial standard by Intel, Microsoft and Toshiba, provides a software-hardware interface allowing power managers to control the power of various system components. Application and operating system-based DPM techniques, which will be discussed later in this section, utilize such interface to conserve power. Although application-based schemes can be the most effective because future workloads are best known to applications, OS-based schemes have a benefit that existing applications do not need to be re-written for energy savings.

Advanced Configuration and Power Interface (ACPI)

ACPI [2] evolved from the older *Advanced Power Management* (*APM*) standard targeting desktop PCs [3]. Implemented at the BIOS (Basic I/O Services)-level, APM policies are rather simple and deterministic. Application and OS make normal BIOS calls to access a device and the APM-aware BIOS serve the I/O requests while conserving energy. One advantage of APM is that the whole process is transparent to application and OS software. ACPI is a substitute for APM at the system software level. Unlike APM, ACPI does not directly deal with power management. Instead, it exposes the power management interfaces for various hardware devices to the OS and the responsibility of power management is left to application or operating system software. Figure 7 overviews the interactions among system components in ACPI [2]. The front-end of the ACPI is the ACPI-compliant device driver. It maps kernel requests to ACPI commands and maps ACPI responses to I/O interrupts or kernel signals. Note that the kernel may also interact with non-ACPI-compliant hardware through other device drivers.

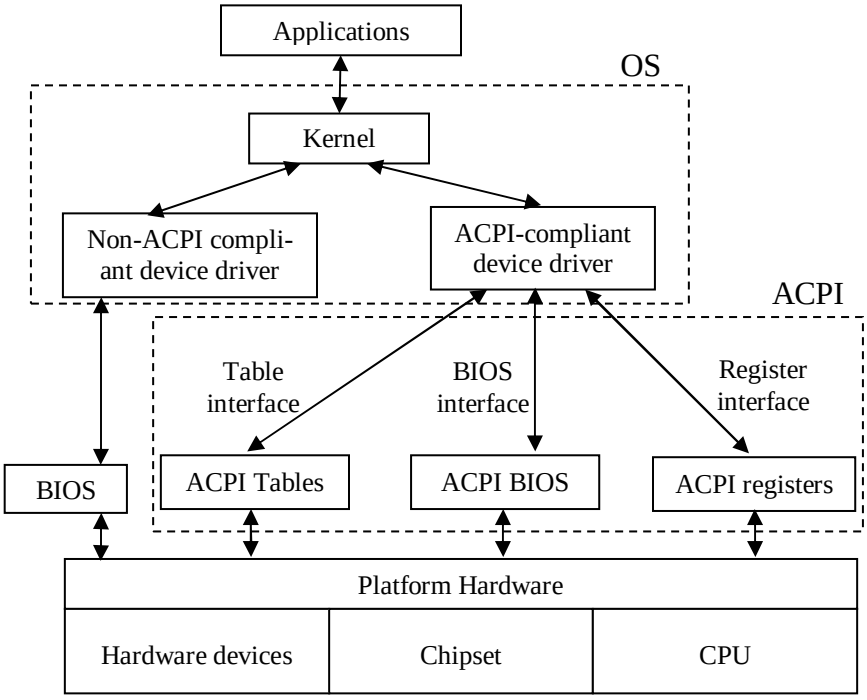


Figure 7: Interaction among system components with ACPI [2].

The ACPI specification defines five *global* system power states: G0 represents the working state, G1 to G3 represent the sleeping states, and lastly one legacy state for non-ACPI-compliant devices. The specification also refines the sleeping state by defining additional four *sleeping* states (S1-S4) within the state G1 as shown in Table 9. In addition to the global states, ACPI also defines four *device* (D0-D3) and four *processor* states (C0-C3). Different states differ in the power they consume and the time needed for wake up. For example, a deep sleep state, such as S4 state in Table 9, saves more power but takes longer to wake up.

Table 9: ACPI global states [2].

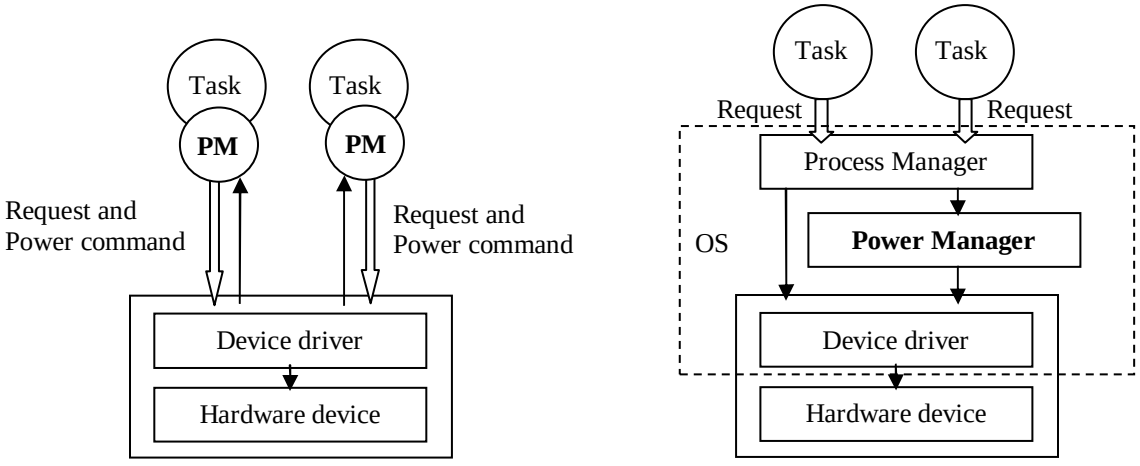
Global states		Description
G3		Mechanical off: no power consumption, system off.
G2		Soft off: full OS reboot needed to restore working state.
G1	***Sleeping states***	Sleeping: system appears to be Off, and will return to working state in an amount of time that increases with the inverse of power consumption.
	S4	Longest wake-up latency and lowest power. All devices are powered off.
	S3	Low wake-up latency. All system contexts are lost except system memory.
	S2	Low wake-up latency. Only CPU and system cache context is lost.
	S1	Lowest wake-up latency. No system context is lost.
G0		Working: system On and fully operational.
Legacy		Legacy: entered when system is non-ACPI compliant.

Less Power
More latency

More Power
Less latency

Application-based DPM

Application-based DPM policies were made possible by the emergence of the ACPI standard state above. Theses policies move the power manager from the device or hardware level to the application level. The application, which is the source of most requests to the target device, is now in charge of commanding the power states of that device. These policies allow application programs to put a device in the fully working state, send it to sleep mode, wake it up, or receive notice about the device power-state changes. For example, Microsoft's *OnNow* [47] and *ACPI4Linux* [1] support power management for ACPI-compliant devices. Figure 8(a) illustrates the application-based DPM scheme.



(a) Application-level power management (b) OS-level power management

Figure 8: Software-based DPM policies.

Alternatively, Lu *et al.* proposed a software architecture that exports power management mechanisms to the application level through a template [38]. This scheme makes power state decisions based on information about the system parameters such as power at each state, transition energy, delay, and future workload. This software architecture differs from the ACPI-based techniques in that the power management is centralized to one application, which makes it safer and more efficient in a single application system.

Operating system-based DPM

Application-based power management has several drawbacks. First, they require modifications to existing applications, and thus implementing these policies will place additional burden on the programmers. Second, advances in technology constantly change hardware parameters, which make a policy optimized for a certain environment inefficient after a device is replaced. Finally, different programs may set the same device to different power states causing the system become unstable. OS-based DPM techniques use the operating system's knowledge of all the running processes and their interaction with the hardware to optimize energy performance. A number of OS-based DPM schemes have been proposed in the literature [23, 35, 36]. Figure 8(b) illustrates the implementation of OS-based power management.

Lu *et al*. proposed *task-based* power management, which uses process IDs at the OS-level to differentiate tasks that make I/O requests [35, 36]. This has a major advantage over device-based policies in that it offers a better understanding of device utilization as well its future usage pattern. For example, an OS-based DPM scheme, called *power-oriented process scheduling*, schedules tasks by clustering idle periods to reduce the number of power-state transitions and state-transition overhead [35]. Finally, Gniady *et al.* proposed to use program counters to predict I/O activities in the operating system [23]. Their *program-counter access predictor* dynamically learns the access patterns of an application using path-based correlation to match a particular se-

quence of program counters leading to each idle period. This information is then used to predict future occurrences of this idle period and thus optimize power.

3.3 Parallel System-level DPM

Most of the power related research topics are devoted to uni-processor systems. However, due the co-operative nature of computation in a parallel computing environment, the most energy-efficient execution for each individual processor may not necessarily lead to the best overall energy-efficient execution. Reducing power consumption not only reduces the operation cost for cooling but also increases reliability, which is often critically important for these high-end systems. This section introduces DPM techniques for parallel systems proposed in the literature. First, hardware-based power optimization techniques such as coordinated DVS [19] and low-power interconnection networks [32, 53] in cluster systems are presented in Subsection 3.3.1. Second, software-based DPM techniques such as energy-aware synchronization for multiprocessors systems [34] are introduced in Subsection 3.3.2. This subsection also presents DPM techniques used in server clusters to reduce the energy consumption of the whole cluster by coordinating and distributing the workload among all available nodes [50].

3.3.1 Hardware-based DPM techniques

Coordinated dynamic voltage scaling (CVS)

Elnozahy *et al.* also proposed to use the DVS scheme discussed in Subsection 3.1.3 in a cluster system [19]. They presented five such policies for comparison. The first policy, *Independent Voltage Scaling* (*IVS*) simply uses voltage scaled processors, where each node independently manages its own power consumption. The second policy, called *Coordinated Voltage Scaling* (*CVS*), uses DVS in a coordinated manner so that all cluster nodes operate very close to the average frequency setting across the cluster in order to reduce the overall energy cost. This can be achieved by periodically computing the average frequency setting of all active nodes by a central-

ized monitor and broadcasting it to all the nodes in the cluster. The third policy, called *vary-on/vary-off* (*VOVO*), turns off some nodes so that only the minimum number of nodes required to support the workload are kept alive. The fourth policy, called *Combined Policy*, combines IVS and VOVO, while the fifth policy, called *Coordinated Policy*, uses a combination of CVS and VOVO. According to their evaluation, the last two policies offer the most energy savings. Among the two, the *Coordinated Policy* (CVS-VOVO) saves more energy at the expense of a more complicated implementation.

Network interconnect-based DPM

One of the most critical power drains in parallel systems is the communication links between nodes, which is an important differentiating factor compared to uni-processor systems. The communication facilities (switches, buses, network cards, etc.) consume a large amount of the power budget of a cluster system [32, 53], which is particularly true with the increasing demand for network bandwidth in such systems. Shang *et al.* proposed to apply DVS to the internetworking links [53]. The intuition is that if network bandwidth could be tuned accurately to follow the link usage, huge power saving can be achieved. A *history-based DVS* policy uses past network traffic in terms of link utilization and receiver input buffer utilization to predict future traffic. It then dynamically adjusts the voltage and frequency of the communication links to minimize the network power consumption while maintaining high performance.

An alternative DPM scheme applied to interconnection links was suggested by Kim *et al.* [32]. They addressed the potential problem of performance degradation, particularly in low to medium workload situations. This may result in more buffer utilization which also increases the overall leakage energy consumption. Their method called *Dynamic Link Shutdown* (*DLS*) scheme attempts to alleviate the problem based on the fact that a subset of under-utilized links (with utilization under a certain threshold) could be completely shut down assuming another subset of highly used links can be found to provide connectivity in the network.

3.3.2 Software-based DPM techniques

Barrier operation-based DPM

As discussed in the previous subsection, interconnection links may be the most critical bottleneck in parallel systems with respect to energy consumption as well as computing performance. From the perspective of application software, collective communications such as *barriers* are often considered the most critical bottleneck during the execution of a parallel application [42]. In a conventional multiprocessor system, an early arriving thread stops (typically by spin-waiting) at the barrier and waits for all slower threads to arrive before proceeding with the execution past the barrier. This barrier spin-waiting is highly inefficient since power is wasted performing unproductive computations.

Li *et al.* proposed *thrifty barrier* [34], where an early arriving thread tries to put its processor into a low power state instead of just spinning. When the last thread arrives, dormant processors are woken up and all the threads proceed past the barrier. However, as discussed earlier in Subsection 3.2.1, power state transitions should justify the power saving versus the delay time incurred in the process. Therefore, each thread that arrives early should predict the length of the pending stall time and decide whether to transit to low power state or not, and if so, choose the best low power state. At the same time, the wake up time must also be predicted to tradeoff power saving versus performance degradation. To tackle these problems, the thrifty barrier uses the past history of interval time between two consecutive barriers to predict the stall time at the barrier [34]. The main objective is to wake up dormant threads just in time for the proceeding execution, thereby achieving significant energy savings without causing performance degradation.

DPM with load balancing

In a cluster system, load balancing is a technique used to evenly distribute the workload over all available nodes in a way that all idle nodes are efficiently utilized. Pinheiro *et al.* used the concept of *load unbalancing* to reduce power consumption of a cluster system [50]. Unlike load balancing, it concentrates work in fewer nodes while idling others that can be turned off, which will lead to power saving but at the same time may degrade performance. Their algorithm periodically evaluates whether some nodes should be removed from or added to the cluster based on the predicted power consumption and the given total workload imposed on the cluster with different cluster configurations. If nodes are underutilized, some of them will be removed, and if nodes are overused, new nodes should be added. In both cases, the algorithm redistributes the existing workload to the active nodes in the cluster. Significant power reduction was reported with only negligible performance degradation [50].

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 low-end and high-end systems. Such models, providing design-time or run-time optimizations, will enable designers to make the right choices in defining the future generation of energy efficient systems.

This chapter discussed different ideas and techniques proposed in the literature with the goal of developing power-aware computer systems. The various methods surveyed were differentiated by design time *power analysis* and run-time *dynamic power management* techniques. Power analysis techniques are mainly based on simulation, sometimes assisted by measurements. These techniques integrate various energy models into existing simulation tools and analyze and profile the power consumption on different levels of a computer system at design time in order to help build power efficient hardware and software systems. On the other hand, dynamic power management techniques are applied during run-time. They monitor the system workload to pre-

dict the future computational requirements and try to dynamically adapt the system behavior accordingly.

Successful design and evaluation of power management and optimization techniques are highly dependent on the availability of a broad and accurate set of power analysis tools, which will be crucial in any future study. While the focus should be more on the overall system behavior capturing the interaction between different system components, it is also important to have a better and more accurate understanding of particular hardware components or software programs with respect to power consumption. While power efficient system design is important in low-end portable systems, this is also true in high-end parallel and clustered systems. This is because high power consumption raises temperature and thus deteriorates performance and reliability. Therefore, a holistic approach that considers all the components in such systems will need to be further investigated.

References

[1] ACPI4Linux. http://phobos.fs.tum.de/acpi/.

[2] Advanced Configuration and Power Interface (ACPI), http://www.acpi.info.

[3] "Advanced Power Management (APM),"
http://www.microsoft.com/whdc/archive/amp_12.mspx, Microsoft Corporation.

[4] A. Azevedo, R. Cornea, I. Issenin, R. Gupta, N. Dutt, A. Nicolau, and A. Veidenbaum, "Architectural and Compiler Strategies for Dynamic Power Management in the COPPER Project," *IWIA 2001 International Workshop on Innovative Architecture*, 2001.

[5] A. Azevedo, R. Cornea, I. Issenin, R. Gupta, N. Dutt, A. Nicolau, and A. Veidenbaum, "Profile-based Dynamic Voltage Scheduling using Program Checkpoints," *Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, 2002.

[6] A. Bavier, A. Montz and L. Peterson, "Predicting MPEG Execution Times," *Proceedings of SIGMETRICS '98/PERFORMANCE '98*, 1998.

[7] L. Benini, A. Boglio, and G. De Micheli, "A Survey of Design Techniques for System-level Dynamic Power Management," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume 8, Issue 3, June 2000.

[8] L. Benini, A. Bogliolo, G. A. Paleologo, and G. De Micheli, "Policy Optimization for Dynamic Power Management," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 18, No. 6, pp. 813–833, June 1999.

[9] 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 Micro*, pp. 26-44, December 2000.

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

[11] D. Burger and T. Austin, “The SimpleScalar Tool Set, Version 2,” *Tech. Report No. 1342*, Computer Sciences Dept. Univ. of Wisconsin, June 1997.

[12] F. Chang, K. Farkas and P. Ranganathan, “Energy-driven Statistical Profiling: Detecting Software Hotspots,” *Proceedings of the Workshop on Power Aware Computing Systems*, February 2002.

[13] W. Chedid and C. Yu, “Dynamic Voltage Scaling Techniques for Power-Aware MPEG Decoding”, *Master’s Thesis*, ECE Dept., Cleveland State University, December 2003.

[14] E.-Y. Chung, L. Benini, A. Bogliolo, Y.-H. Lu, and G. De Micheli, “Dynamic Power Management for Nonstationary Service Requests,” *IEEE Trans. on Computers*, Vol. 51, No. 11, pp. 345–1361, November 2002.

[15] E.-Y. Chung, L. Benini, and G. De Micheli, “Dynamic Power Management Using Adaptive Learning Tree,” *Proceedings of the International Conference on Computer-Aided Design*, pp. 274–279, November 1999.

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

[17] F. Douglis, P. Krishnan, and B. Bershad, “Adaptive Disk Spin-down Policies for Mobile Computers,” *Proceedings 2nd USENIX Symp. on Mobile and Location-Independent Computing*, pp. 381–413, 1995.

[18] Earth Simulator, http://www.es.jamstec.go.jp/esc/eng/index.html.

[19] E. N. Elnozahy, M. Kistler and R. Rajamony, “Energy-Efficient Server Clusters,” *Proceedings of the Second Workshop on Power Aware Computing Systems*, February 2002.

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

[21] S. Gary, P. Ippolito, G. Gerosa, C. Dietz, J. Eno, and H. Sanchez, “PowerPC 603, A Microprocessor for Portable Computers,” *IEEE Design & Test of Computers*, Volume 11, Issue 4, pp. 14-23, October 1994.

[22] N. A. Ghazaleh, D. Mosse, B. Childers, R. Melhem, and M. Craven, “Collaborative Operating System and Compiler Power Management for Real-Time Applications,” *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.

[23] C. Gniady, Y. C. Hu, Y.-H. Lu, “Program Counter Based Techniques for Dynamic Power Management,” *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, February 2004.

[24] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes, “Idleness is Not Sloth,” *Proceedings of the USENIX Winter Conference*, pp. 201–212, 1995.

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

[26] M. Gowan, L. Biro, and D. Jackson, "Power Considerations in the Design of the Alpha 21 264 microprocessor," *ACM Design Automation Conference*, pp. 726-731, June 1998.

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

[28] C.-H. Hwang and A. Wu, "A Predictive System Shutdown Method for Energy Saving of Event-driven Computation," *International Conference on Computer-Aided Design*, pp. 28–32, November 1997.

[29] J. Henkel and H. Lekatsas, "A^2BC: Adaptive Address Bus Coding for Low Power Deep Sub-Micron Designs," *ACM Design Automation Conference*, 2001.

[30] IRIX OS, http://www.sgi.com/developers/technology/irix/.

[31] M. Kandemir, N. Vijaykhrishnan, M. Irwin, "Compiler Optimizations for Low Power Systems", *Workshop on Power Aware Computing Systems*, pp. 191-210, 2002.

[32] E. J. Kim, K. H. Yum, G. M. Link, C. R. Das, N. Vijaykrishnan, M. Kandemir and M. J. Irwin, "Energy Optimization Techniques in Cluster Interconnects", *International Symposium on Low Power Electronics and Design (ISLPED'03)*, August, 2003.

[33] A. Klaiber, "The Technology Behind Crusoe Processors," Transmeta Corporation, http://www.transmeta.com/pdfs/paper_aklaiber_19jan00.pdf, January 2000.

[34] J. Li, J. F. Martíne, and M. C. Huang, "The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors," *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, February 2004.

[35] Y.-H. Lu, L. Benini, and G. De Micheli, "Low-Power Task Scheduling for Multiple Devices," *International Workshop on Hardware/Software Codesign*, May 2000.

[36] Y.-H. Lu, L. Benini, and G. De Micheli, "Operating-System Directed Power Reduction", *International Symposium on Low Power Electronics and Design*, July 2000.

[37] Y.-H. Lu and G. De Micheli, "Comparing System-Level Power Management Policies," *IEEE Design & Test of Computers*, Volume 18, Issue 2, pp. 10-19, March 2001.

[38] Y.-H. Lu, T. Simunic, and G. De Micheli, "Software Controlled Power Management," *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES99)*, pages 157–161, May 1999.

[39] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, Inc., 1980.

[40] Millywatt, http://www.cs.duke.edu/ari/millywatt/.

[41] Mobile Intel Pentium III Processor in BGA2 and Micro-PGA2 Packages Datasheet, Intel Corporation.

[42] S. Moh, C. Yu, B. Lee, H. Y. Youn, D. Han, and D. Lee, "4-ary Tree-Based Barrier Synchronization for 2-D Meshes without Nonmember Involvement," *IEEE Transactions on Computers*, Vol. 50, No. 8, pp. 811-823, August 2001.

[43] Motorola Dragonball, http://www.motorola.com/dragonball/.

[44] M. Moudgill, P. Bose and J. Moreno, "Validation of Turandot, a Fast Processor Model for Microarchitecture Exploration," *Proceedings of IEEE Int'l Performance, Computing and*

Communication Conf., pp. 451-457, 1999.

[45] "MPEG-2: The Basics of how it Works," Hewlett Packard Lab.

[46] W. Namgoong, M. Yu, and T. Meng, "A High-Efficiency Variable-Voltage CMOS Dynamic DC-DC Switching Regulator," *IEEE Int'l Solid-State Circuits Conf.*, pp. 380-381, 1997.

[47] OnNow. http://www.microsoft.com/hwdev/onnow/ .

[48] Palm OS Emulator, http://www.palmos.com/dev/tools/emulator/.

[49] O. A. Patino and M. Jimenez, "Instruction Level Power Profile for the PowerPC Microprocessor," *Computing Research Conference 2003, University of Puerto Rico-Mayagüez*, pp. 120-123, April 2003.

[50] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath, "Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems," *Technical Report DCS-TR-440*, Dept. Computer Science, Rutgers University, May 2001.

[51] Q. Qiu and M. Pedram, "Dynamic Power Management Based on Continuous-Time Markov Decision Processes," *Proceedings of the Design Automation Conference*, pp. 555–561, June 1999.

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

[53] L. Shang, L.-S. Peh,and N. K. Jha, "Dynamic Voltage Scaling with Links for Power Optimization of Interconnection Networks", *Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, February 2003.

[54] D. Shin and J. Kim, "Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications," *IEEE Design & Test of Computers*, Volume 18, Issue 2, pp. 20-30, March 2001.

[55] SimOS, http://simos.stanford.edu/

[56] Simulink, http://www.mathworks.com/products/simulink/, The MathWorks.

[57] T. Simunic, L. Benini, P. Glynn, and G. De Micheli, "Dynamic Power Management for Portable Systems," *Proceedings of the International Conference on Mobile Computing and Networking*, pp. 11–19, 2000.

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

[59] SPEC JVM98, http://www.specbench.org/osg/jvm98/.

[60] M. B. Srivastava, A. P. Chandrakasan, and R.W. Brodersen. "Predictive System Shutdown and Other Architecture Techniques for Energy Efficient Programmable Computation," *IEEE Transaction on VLSI Systems*, Vol. 4, No. 1, pp. 42–55, March 1996.

[61] M. Stan and W. Burleson, "Bus-Invert Coding for Low-Power I/O," *IEEE Transaction on VLSI*, Vol. 3, No. 1, pp. 49-58, 1995.

[62] The Standard Performance Evaluation Corporation (SPEC), http://www.spec.org/.

[63] C. L. Su, C. Y. Tsui, and A. M. Despain, "Low Power Architecture Design and Compilation Techniques for High-Performance Processors", *COMPCON'94*, 1994.

[64] K. Suzuki, et al., "A 300 MIPS/W RISC Core Processor with Variable Supply-Voltage Scheme in Variable Threshold-Voltage CMOS," *IEEE Custom Integrated Circuits Conf.*, pp. 587-590, 1997.

[65] V. Tiwari, S. Malik, A. Wolfe, and M. T. Lee “Instruction Level Power Analysis and Optimization of Software,” *Journal of VLSI Signal Processing*, Vol. 13, Issue 2-3, August 1996.

[66] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura, “Instruction Scheduling for Power Reduction in Processor-Based System Design”, *Proceedings of Design Automation and Test in Europe (DATE98)*, pp. 855-860, 1998.

[67] H.-S. Wang, X.-P. Zhu, L.-S. Peh and S. Malik, "Orion: A Power-Performance Simulator for Interconnection Networks”, *Proceedings of MICRO 35*, November 2002.

[68] M. Weiser, B. Welch, A. Demers and S. Shenker, “Scheduling for Reduced CPU Energy,” *USENIX Symposium on Operating Systems Design and Implementation* (*OSDI*), pp. 13-23, November 1994.

[69] V. Wen, M. Whitney, Y. Patel, J. D. Kubiatowicz, “Exploiting Prediction to Reduce Power on Buses,” *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, 2004.

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

[71] T. T. Ye , G. De Micheli , L. Benini, “Analysis of Power Consumption on Switch Fabrics in Network Routers”, *Proceedings of the 39th conference on Design automation*, June 2002.