Topics in Parallel and Distributed Computing: Introducing Algorithms, Programming, and Performance within Undergraduate Curricula[*][†][‡]

Chapter 5 – Energy Efficiency Issues in Computing Systems

Krishna Kant

Temple University, kkant@temple.edu

[*]How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. Topics in Parallel and Distributed Computing: Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms, Springer International Publishing, 2018, ISBN : 978-3-319-93108-1, Pages: 337.

[†]Free preprint version of this book: `https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cder_book_2`

[‡]Free preprint version of volume one: `https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book`

Abstract

The purpose of this chapter is to introduce energy efficiency issues in computer systems and its importance to PDC curriculum. This is done mostly at a basic level, i.e., definition of terms and basic concepts (K and C Bloom levels), so that the students get a broad overview of the entire field as it applies from very low hardware level up to software and service level issues. Energy management in parallel and distributed systems are also covered. The chapter attempts to convey the idea that the energy is ultimately consumed by transistors and wires, and a thorough understanding of the hardware issues is essential to effectively deal with the energy efficiency and adaptation issues. Some of the material can be considered at the A Bloom level as well.

Relevant core courses: Systems, Arch 2, ParAlgo

Relevant PDC topics: Cross-cutting topics: Power Consumption

Learning outcomes: Know basics of energy, power and thermal issues in computing, importance of and technology trends in power consumption, power-performance tradeoffs, power states and their use at HW and SW level, power adaptation, energy efficiency of parallel programs

Context for use: Traditionally, computing has focused only on performance at all levels including circuits, architecture, algorithms, and systems. With power consumption and power density playing a central role at all these levels, it is crucial to teach students about power and power-performance tradeoffs at all these levels.

5.1 Why Does Energy Efficiency Matter?

As the world gets increasingly fused with information technology, the energy consumption associated with the information technology continues to rise. For the mobile and embedded devices such as mobile phones and sensors, energy efficiency is crucial because of the increasing demands on the batteries, which have not scaled well in capacity as compared to the energy demands placed on them. In many embedded applications, particularly the emerging Internet of Things (IoT), battery replacement or charging is infeasible or very costly, and there is increasing trend towards energy harvesting from the environment in a cost effective way. This places severe limitations on energy use and thus makes energy efficiency the foremost issue in the design of both hardware and software. On the other end of the scale, data centers continue to grow in size in order to handle increasing client demands for running complex queries requiring substantial computation and processing of large amounts of data, often in real time. This has a direct impact on increasing energy consumption of data centers, and in turn requires greater emphasis on energy management energy efficient processing algorithms.

5.1.1 Energy Related Challenges

The high energy consumption in data centers poses numerous challenges which make the energy efficiency increasingly important. First, there is the cost of electricity and electric infrastructure. Large data centers consume 10MW or more power, and the corresponding energy cost may amount to 40% or more of its operating cost. (We will shortly discuss relationship between power and energy.) Large power consumption implies costly and costly-to-maintain electric infrastructure that includes large step-down transformers (along with their cooling costs), uninterruptible power supplies (UPS), relays/switch-gear, backup generators, multiple stages of AC-DC converters, etc. All of this can be scaled down by making the data center more energy efficient. Second, large data centers require large cooling infrastructures (including water chiller plants, air conditioners, fans, etc.), which too can be scaled down with more efficient operations. It is estimated that 1W saved by using energy efficient operation of the servers, can result in up to 3W of total saved power.

Other less obvious side effects of high energy consumption come into play at basic architectural levels. Ever since its inception in early 1970's, the semiconductor technology has thrived on the steady reduction of "feature size", which refers to a basic measure used for designing and patterning transistors and on-chip connections between them (the "wires"). For example, the current Intel processors are based on the 14 nanometers (nm) technology, the latest one being the so called 8th generation Coffeelake. Traditionally, the feature size has decreased by a factor of $\sqrt{2}$ every 2-3 years, however, the huge difficulties in going down to very small feature sizes has slowed this trend. One critical issue arising due to continued reduction in feature size is that in addition to the power consumption, we also need to pay attention to *power density*, or the power consumed per square cm of chip area.

Power density was already threatening to become unsustainable in early 2000's, and a variety of techniques were essential to keep it in check, and these continue to be crucial. The techniques range from the lowest level of semiconductor device physics to circuit, logic, architecture, and beyond. The current state of the technology can easily put more than a billion transistors on a cm^2. For example, Intel's core i7-6950 extreme edition built using the 14 nm technology has 10 cores and consumes 140 Watts, and packs 3.4B transistors in 246mm^2 die. The power density is (140/2.46) or 72 Watts/cm^2. Since all power consumed ultimately appears as heat, this means that we need to have enough cooling capacity to remove 72 watts power from each cm^2 area. With transistor layers already being stacked vertically in the emerging 3-D architectures, this becomes an extremely challenging task.

The net result is that it is possible to put lot more processing cores on a die than can be simultaneously powered on due limitations in removing the heat. This leads to the so called problem of "dark silicon" where it becomes necessary to keep some of the transistors unpowered in order to meet the heat dissipation requirement. In other words, energy efficient operation of cores has direct implications in terms of usable cores per package.

5.1.2 Making Computing Energy Efficient

Achieving energy efficiency involves a multi-level effort that includes an interplay of energy efficient semiconductor materials, transistor and circuit designs, hardware architecture, and software design along with suitable *power management techniques* that are engaged at various levels in order to provide a suitable trade-offs between performance and energy consumption. Unfortunately, the recent trends in the manufacturing and circuit design introduce the third element – reliability. One reason for reliability issues arises from atomic level feature sizes. In particular, the latest technology already operates at the feature size of 10nm, which amounts to only 30 Silicon atoms! Such small sizes allow for quantum mechanical tunneling and make it impossible to shape the boundaries of transistors accurately, thereby leading unreliable operation. Another reason for unreliability is the continuing decrease in operating voltage. As we shall see later, decreasing operating voltage can reduce the power consumption substantially, and there are emerging trends of operating the chips close to voltage levels where the transistors may not even switch reliably. This unreliability is handled by error detection and repetition of errored operation at a very low level; however, there may be some danger of residual undetected errors. These residual errors can be handled at higher levels by additional redundancy; however, this results in both additional power consumption and loss of performance. In other words, increasingly we need to consider a 3-way tradeoff: performance, power, and reliability. In this chapter, we do not address this tradeoff; however, this is an important emerging issue in the context of energy efficiency.

5.2 Basic Concepts

5.2.1 Power vs. Energy vs. Heat

Power and energy are often used interchangeably in informal discussions; however, they are different and it is crucial not to mix up the two. *Energy*, measured in Joules, is defined for the entire computation of interest whereas *Power*, measured in Watts, is the rate of energy consumption. That is Watts = Joules/sec. For charging purposes, electric energy is often measured in units of "Kilo-watt hours" (KWH). Obviously, 1 KWH = 3.6 Mega Joules.

For a given program running from start to finish, the important parameters are its total energy consumption and its running time. The ratio of the two gives the average power consumption. If it is important to complete the program quickly, it may be possible to run it at a higher power level and thereby finish it quickly. Conversely, if we can wait to get the results, it may be possible to run the program at a lower power. Since the electrical infrastructure at all levels (from data center to individual servers) has limits on how much power can be drawn, the maximum allowable power consumption is also often limited. Higher power also generates more heat (as discussed below) which may further require limits on power consumption. Smart power management techniques, discussed later, could reduce the power consumption, so that power circuit limits are not violated and the electronics does not overheat. In terms of electricity cost, however, it is the total energy consumed that is of primary interest. Many power management techniques can also reduce the total energy consumption, as discussed later.

Power consumption increases the average kinetic energy of the atoms of the material (e.g., silicon) which heats the material. Since energy cannot be destroyed, nearly all of the power consumed is eventually converted into heat, which manifests itself as higher temperature. When two materials (e.g., silicon and surrounding air) are at different temperatures, there is a heat flow from higher temperature material to the lower temperature material until the average energy of the two is equalized. Thus, in the long run, the temperature of a substance is determined by the balance between power consumption (which raises temperature) and cooling (forced or natural) due to the surrounding substance (e.g., air) at lower temperature. The rate at which the temperature approaches the final steady state value depends on the thermal properties of the materials which determine heat transfer via conduction, convection, and radiation. The important point to note is that even if the steady state temperature is moderate, over short periods the temperature within the material (e.g., transistor junction) may become high enough to cause damage or errors. Thus effective cooling at the points of highest heat generation is crucial but becomes harder and harder to achieve as the transistor density grows.

5.2.2 Idle vs. Active Power Consumption

In the traditional semiconductor technology, there are two types of power consumptions that are of interest: (a) static (also known as *idle* or passive) power, and (b) dynamic (also known as *active*) power. A conventional transistor consists of a silicon "channel" from *Source* to the *Drain* terminal. This channel is controlled by a "gate" that can apply a positive or negative voltage to the channel. The positive voltage causes a flow of electrons through the channel, which turns the transistor on, whereas a negative voltage cuts off the electron flow and turns the transistor off. Unfortunately, even with negative voltage, there is a small flow of current in the channel, known as "leakage current", which tends to increase steadily as the transistor feature size shrinks. A higher current flow means higher power consumption at the same voltage, since Power = Current x Voltage. The leakage current is the primary source of static or idle power, since this consumption happens even if there is no activity.

In particular, let V denote the Source-Drain voltage (often denoted in the literature as V_{cc} or V_{dd}). Let I_L denote the leakage current. Then the idle power consumption $P_{idle} = V \times I_L$. The idle power may form 20-40% of the total power consumption of a CPU or a memory module, and is expected to increase as feature sizes shrink. Thus, it is important to devise mechanisms to reduce it via a set of *Idle power management* techniques.

Explaining the dynamic power consumption requires a bit more understanding of the CMOS (complementary metal oxide semiconductor) technology which used almost universally in current digital designs. Suffice to say that a CMOS device actually uses two basic transistors such that while one of them is on, the other is off. Thus the functioning of the CMOS device amounts to switching back and forth between these two complementary configurations, which can be identified as representing the logical 0 and 1. Each switch from 1 to 0 or 0 to 1 consumes power that is over and above the static (idle) power. This is the dynamic power and it clearly is proportional to the rate of switching – or the frequency at which the electronic component operates, henceforth denoted as f and measured in Hertz. Every transistor has an inherent capacitance, denoted C, and each switching amounts to charging or discharging this capacitor. The charge held by a capacitor is given by the product of capacitance and operating voltage, and the current, by definition, rate at which charging (or discharging) happens. That is, dynamic current = Capacitance x Voltage /switching_time $= C \times V \times f$. Since, dynamic power = Voltage x dynamic current, the dynamic power consumption of a CMOS transistor, $P_{dynamic} = 1/2 \times C \times V^2 \times f$. Here, the factor 1/2 results from the fact that half the energy is cycled in 0-1 transition and the other half in 1-0 transition.

It is important to note that the $P_{dynamic}$ computed here will be consumed only when the transistor switches in every cycle. In general, the transistor will switch only some fraction of time, denoted U. Then the total power consumption of a transistor is given by:

$$P_{total} = P_{idle} + U \times P_{dynamic} = V.I_L + 1/2\ UC.V^2.f \tag{5.1}$$

Notice that reducing V reduces both idle and dynamic power, and hence is an attractive way to reduce the total power consumption.

The above equation can be used to compute power consumption of the entire core or CPU by considering all of the transistors that comprise the core or CPU. For example, I_L can be thought of as the total leakage current from all the transistors comprising the core/CPU. Furthermore, we can think of U as the *utilization*, defined as the fraction of cycles for which the core/CPU is busy. That is, the capacitance C then corresponds to an effective value that corresponds to capacitance of all those transistors that switch, on the average, in one cycle.

Let us illustrate this with an example of a core with operating voltage (V) = 1.2 Volts, Leakage current (I_L) = 7.5 Amp, Effective Capacitance (C) = 10 nanoFarad, and operating frequency (f) = 2 GHz. In this case, the idle power is $1.2 \times 7.5 = 9.0$ Watts, and maximum dynamic power $= 0.5 \times 10 \times 1.44 \times 2 = 14.4$ Watts. Thus, the core will consume 23.4 Watts when 100% busy, or 16.2 Watts when 50% busy.

The above equation applies to not only the CPUs but also to other components of the system (e.g., cache, DRAM, links connecting various components, etc.), although the details vary. For example, many links within the chip are "synchronous" in that they continuously transmit either real frames or small "fill-in" frames. For such links, the power consumption remains the same irrespective of their utilization. The actual power consumption also depends on circuit level power management via "power gating", or not supplying power to any entities that do not need to be powered up. Aggressive power gating can significantly reduce the transistors that need to be powered up and thereby reduce the overall power consumption. Of course, power gating is not free, since the power gating itself requires transistors that consume extra power. In this chapter, we will largely talk about power management on the coarser scale than targeted by power gating, and this may be suitably accomplished in hardware or software.

5.3 Power States and their Management

Almost all major components in a modern server offer control knobs in the form of power states – a collection of operational modes that trade off power consumption for performance in different ways. Power control techniques can be defined at multiple levels within the hardware/software hierarchy with intricate relationships between knobs across layers. We call a power state for a component *active* if the component remains operational while in that state; otherwise we call the state *inactive* or *idle*. These active and inactive states offer *temporal* power control for the associated components. Another form of power control is *spatial* in nature, wherein only a subset of a set of identical components are operational.

At the highest level, the OS-directed Power Management (OSPM) defines a set of power states for the entire machine that most users are already familiar with. There are five such "system" states denoted S0..S5, with the following being the most relevant: S0 (working), S3 (standby – or inactive with state saved into the DRAM), S4 (hibernating – or inactive with state saved into the secondary storage), and S5 (essentially turned off and requiring reboot, but rebooting possible remotely). For example, when you press the sleep button on a laptop, it enters the S3 state (unless hibernate or hybrid sleep-hibernate states are chosen in the options). In the system state S0, individual devices (e.g., CPU, memory, links, etc.) have further power states defined. In particular, the CPU offers three types of states, C states (inactive) and P and T states (active) as shown in Fig. 5.1.

5.3.1 Processor Power States

The most commonly known processor states are the P (performance), where P0 refers to the highest frequency state and P1, P2, etc. refer to progressively lower frequency states. Lower frequency allows operation at a lower voltage as well and thus each P state corresponds to a supported (voltage, frequency) pair as illustrated in Fig 5.1. The active power consumption is proportional to frequency but goes as the square of the voltage, as discussed in equation 5.1. The combined effect of reduced frequency and voltage makes the dynamic power consumption in higher numbered P states quite low. For instance, consider our earlier example where a core runs at 2.0GHz with V=1.2 Volts. Assume that this is the P_0 state. Now suppose, that in the P_2 state the core runs at 1.0GHz which allows the voltage to go down to 0.9 volts. Thus, the effective reduction in the dynamic power is given by:

$$\text{Power Ratio} = V_0^2 f_0 / V_1^2 f_1 = 1.2^2 \times 2.0/(0.9^2 \times 1.0) = 3.56 \tag{5.2}$$

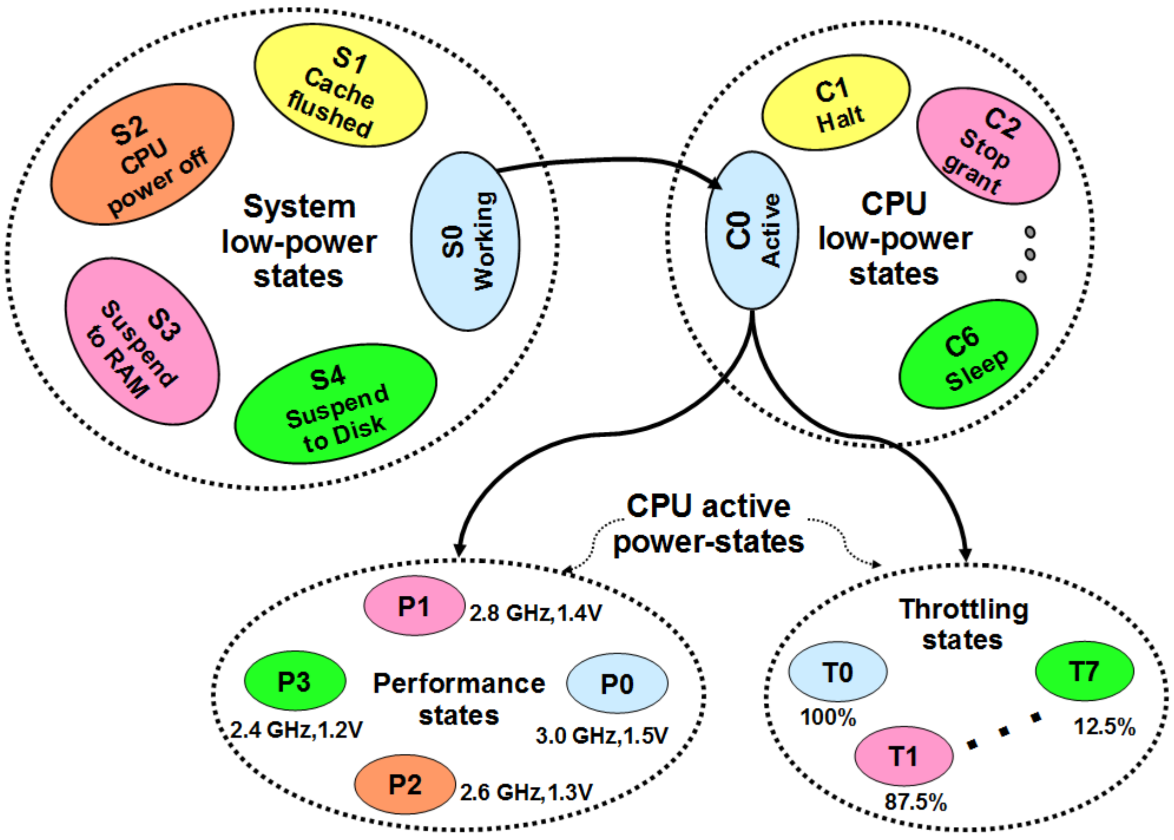

Fig 5.1: System and Device Power States

In other words, the dynamic power of a 100% busy core will go down from 14.4 Watts to 4.05 Watts!

In addition to the reduction in dynamic power, a lower voltage decreases the static power consumption also. For this reason, *dynamic voltage-frequency switching* (DVFS), which suitably controls both the frequency and voltage, are among the most explored *active power management* technologies [2]. A changeover to a higher numbered P state involves two steps: (1) Reduce the voltage to the new level and let it settle down, and (2) Lower the frequency and let the phase-locked loop (PLL) circuit settle down.

Modern CPUs also provide inactive or "sleep" states are denoted as C0, C1, C2, ..., where C0 is the active (operational) state and others are inactive states with increasing power savings. The power saving in inactive state is achieved by several techniques including turning off clocks, lowering the voltage (to lower the leakage power), and taking some additional actions such as flushing the cache and powering it off. The precise action taken is architecture dependent. For example, C1 and C2 states only turn off clock, C3 flushes the lowest level cache as well, and C6 flushes the next level cache and powers down some links. It is expected that future processors will have even deeper inactive states. The deeper C state provides higher power savings but at the cost of longer transition times into and out of the state. With actions such as cache flushing, the impact of the sleep persists even beyond the point when the CPU is active again, since much of the flushed data may need to be fetched again from higher level cache or memory.

The *idle power management* techniques must intelligently decide when to put the CPU in a C state, which one, and when to exit it. With multiple C states, there is also the question of transitioning between different C states. Unfortunately, the CPU cannot directly go from one C state to another – instead it must first become active (i.e., transition to C0 state) and then choose another inactive state to go into.

In case of a multi-core CPU, most of the above states apply to individual cores as well. However, depending on the architecture, certain features may not be available independently to each core. For example, if all (or a group of) cores lie on the same voltage bus, it is not possible to change their voltages independently. For sensible power management, it is necessary to relate the core states to the CPU (i.e., entire package) states. The general rule is that the state of the entire package corresponds to the state of the core that is in the shallowest C state. For example, if even one core in a package is active (in C0 state), the entire package must be considered to be active, so that it can function normally in terms of data transfer to and from the package level cache or the memory.

Processors often also implement T (throttling) states, but these are entirely intended to handle thermal emergencies by introducing gaps in the clock (to allow the processor to cool down). We will not discuss them further here.

5.3.2 Memory Power States

The ever increasing appetite for more memory is already making memory power consumption rival CPU consumption. Thus aggressive management of memory power is essential. A memory stick or DIMM (Dual inline memory module) of DRAM (dynamic random access memory) consists of several memory devices (or chips), each of which typically provides 8 bits of data in parallel. There is considerable internal structure to the modern DDR (dual data rate) DIMMs. In particular, each DIMM is divided into "ranks", with 1, 2 or 4 ranks per DIMM. A "rank" is usually a set of 9 memory devices (8 for data and 1 for parity). The 8 devices of a rank collectively and in parallel provide the entire 64 bit (8 bits from each of 8 devices) "chunk" over the memory channel. Memory controllers often support multiple channels, each allowing one or more DIMMs and capable of independent data transfer. Since the data from all ranks of all DIMMs on a channel must flow over that channel, the ranks can be lightly utilized even if the channel is quite busy.

As its name implies, a DDR DRAM transfers 8 bytes (64 bits) of data on both edges of the clock. It thus takes 4 cycles to transfer a typical 64byte cacheline. The total power consumption of a DIMM is 3-4 W range with idle power of about 1 W. Several sleep states are available for the *idle power management* of the DRAM. The two shallowest sleep states are called "fast" and "slow" CKE (clock enable) states; these allow an inactive rank to deactivate parts of the circuitry such that it is possible to reactivate them in a few ten's of nanoseconds. As the name implies, the slow CKE is somewhat slower, but achieves higher power savings. For a much deeper sleep, it is possible to put a DIMM in "self-refresh" mode. The essential characteristic of the DRAM technology is that the stored data must be constantly "refreshed", i.e., read and then written back every 64 ms or less; otherwise, it will be lost. Normally, the refreshing is done by the *memory controller*, which is the intelligent entity that interfaces CPU and the DRAM. In self-refresh mode, the DRAM refreshes itself, so that the memory controller and the link between the memory controller and DRAM can become inactive. Self-refresh is typically used when the CPU is placed in C6 state. The memory can also use *active power management* provided that it can run at several different frequencies. As with CPU, these active states are most useful if the lowering of frequency is also accompanied by a suitable lowering of the voltage.

5.3.3 Link Power States

Modern computer systems use a variety of networking media both “inside-the-box” and outside. The best known outside-the-box networking technology is Ethernet, but there are others as well, such as Fiber-Channel (used for storage networking), and InfiniBand (for low latency interconnection in high performance computing). Ethernet and other technologies can consume a substantial percentage of the IT power in a large data center, and their power management is becoming essential. These technologies continue to increase in speed – for example, 10 Gb/sec Ethernet is becoming quite popular in data centers, and the higher end data centers are moving to 40 Gb/sec Ethernet. Although the power consumption of a 10 Gb/sec Ethernet is only about 3 times that of 1 Gb/sec Ethernet, the increased speed often results in power inefficiency. The main reason for this is that most network links carry very little traffic most of the time, and high bandwidth is required only sporadically. Thus an upgrade from 1 Gb/sec to 10 Gb/sec Ethernet increases the power consumption by a factor of 3, with very little additional traffic carried on the average.

Moden computer systems have several internal interconnects, that run at much higher speeds than outside-the-box interconnects, and can collectively consume a significant percentage of platform power. With many cores per CPU, a significant number of links and/or link interfaces are required for interconnecting the cores, and this interconnect must support extremely low latency and very high bandwidth. The IO interconnects, including PCI-Express, SATA, SAS form other prominent inside-the-box interconnects. There are still others such as interface between memory controller and DIMMS or interconnect between CPU and IO complexes. An intelligent power management of such interconnects also becomes crucial for platform power reduction. Such links are invariably “synchronous” which means that the power consumption does not depend on the data rate.

Most current links support at least two PHY (physical) layer low power states, called L0s and L1, respectively which can be used for *idle power management*. The L0s power state is unidirectional, in that the transmitter for each direction of the link can independently decide to go into low power mode when it has nothing to transmit, whereas the receiver side remains active. The L1 power state involves a handshake between transmitter and receiver, and thus allows both of them to go into low power when there is nothing to transmit. The L1 state can reduce the idle power quite substantially but this comes at the cost of substantial latency; therefore, L1 is typically used with the C6 CPU state.

As with other devices, links can also be operated at lower speeds in order to reduce their active power, and thus allow for their *active power management*. Depending on the type of link, the speed change may be either a matter of simply changing the clock rate or a switch-over to a different PHY. An example of the latter is the 40 Gb/sec Ethernet operating at 10 Gb/s or 1 Gb/s. Such a PHY switch can be extremely slow, and the power reduction may not be significant. Furthermore, the lower speed means longer data transmission time and may not provide any gains in energy consumption.

The energy efficient Ethernet (EEE), also known as Green Ethernet, provides a software controlled low-power idle (LPI) mode initiated by the transmitter, and thus can be used independently for each direction of the link. An LPI enabled transmitter sends a LP_Sleep signal to the receiver, so that the receiver can place its side also in low power mode. The transmitter can tell the receiver how long it will be in LPI mode. On wake up, the transmitter sends a LP_Wakeup signal to the receiver. When the transmitter is in the LPI mode, it continues to send periodic refresh or heart beat signal to maintain the synchronization while consuming only about 10% of the normal power. Wakeup from LPI involves a significant exit latency since in addition to both transmitter needs to wake up the receiver before transmitting anything. The two relevant parameters in this regard are Sleep time (T_s) and Wake-Up time (T_w). For a 10 Gb/s link, with 1500 bytes packet size T_s and T_w will be 2.88 μs and 4.48 μs respectively. This amounts to transmission time of several packets and thus the mechanism is useful when the traffic shows significant gaps between packet bursts. If the workload does not have such characteristics and there is no traffic shaping to make it behave so, LPI can be useful only at very low utilization levels. Consequently, we also study the usefulness of basic L0s state based control assuming that is provided by the Ethernet interface. Such a control will keep the receiver side always awake and only the transmitter can sleep; however, the low transition latencies and lack of handshake between transmit and receive sides makes the mechanism suitable at higher utilization levels as well.

5.3.4 Collective Power Management

In the above, we consider the power states of a single entity (e.g., a CPU core, memory rank, or a link) or a composite entity considered as a single unit (e.g., entire CPU with all its cores, entire DIMM, or a set of parallel links between two devices). However, whenever we have a composite entity or a set of entities (e.g., a set of servers), it is possible to power manage them together without hurting the performance. The basic idea is to consolidate the load on a certain subset of devices, so that the others can go into low power mode. What is significant here is that the vacated devices can generally go into a deep sleep state and stay there for long periods of time. In constrast, if each individual device is power managed separately, it may be able to sleep only briefly, and thus it cannot go into a deep sleep mode. The reason for the latter is that a deeper sleep mode invariably comes with long latencies to transition in and out of the sleep mode.

A special case of collective power management occurs in modern interconnection links, which are made up of several multiple "lanes" of "serial" links. A lane of a serial link carries only 1-bit of data at a time and uses mechanisms (e.g., differential signaling) to make it very robust and noise free. Current systems have generations 1, 2, or 3 of the technology, which supports respectively, 2.0 Gb/sec, 4.0 Gb/sec, and 7.88 Gb/sec bandwidth per lane. Thus, an 8-lane link, referred to as x8 link, can support 1, 2, and 0.98 GB/sec bandwidth depending on the generation. For example, graphics cards typically use x8 or x16 PCI-E links.

Serial links allow *dynamic width management* wherein certain lanes can be put in low power mode to reduce power consumption when the traffic (i.e., the bandwidth requirement) is low. A highly desirable feature of width control is that so long as some lanes are active and traffic is low enough for the available bandwidth, there is very little delay impact of the power management on the traffic. A dynamic width control algorithm has to operate within constraints of supported widths associated with the underlying link hardware.

Dynamic width management can also be used for DRAM by keeping only some of the ranks active at a time, as discussed in [5]. For example, if a server has two DIMMs (presumably on two different memory channels) and each is a 2-rank DIMM, we have a total of 4 ranks and we could rotate among them so that, say, only 2 ranks are active on the average. Such a mechanism trades off memory access latency (and hence performance) against the power consumption since the inactive ranks can be put in one of the sleep modes.

For more general collective power management, let us consider the set of cores in a CPU. In general, only some of the cores may be needed to handle the current workload. In this case, the other cores can go into a deep sleep state such as C6 or could even be powered off depending on how quickly we want to be able to turn on those cores. Even if all cores in the CPU are identical, it does matter which cores are turned on or off by the power management algorithm. One reason for this that the heat produced by one core affects the adjacent cores, which means that it may be undesirable to simultaneously operate two adjacent cores. A well-known scheme in this regard is called “core hopping”, where the cores are used in a cyclic fashion to ensure that all cores generate approximately the same amount of heat. Another reason why the choice of core to turn on/off matter is that each active core typically accumulates its “working set” (i.e., the data most essential to the operation of the program) in its cache. Turning off this core and restarting the computation on another core would force that core to fetch its working set from the shared cache or memory, and thus slow it down. This is a particularly important issue for core hopping – while systematic cycling through different cores may balance out the heat generation, it may also hurt performance due to disturbances to the working sets. Yet another form of power management for a multicore CPU involves the so called “turbo mode” operation. If only some of the cores are active, they can run at a higher frequency and still maintain the desired thermal envelope.

Collective power management also applies at higher levels. For example, if a server rack has 10 servers, each running at a utilization of at most 20% (a fairly typical situation), a significant amount of power can be saved by consolidating all of the workloads and distributing it to only 3 servers. In this case, each server will run at 67% utilization, which may be reasonable. (Except in case of long-running, CPU bound tasks, it is generally not possible or desirable to run a server close to 100% utilization on a sustained basis.) The remaining 7 servers could then be put in one of the system sleep states such as S3, S4, or S5 depending on how much restart delay we are willing to tolerate. For example, suppose that each server consumes 100W of idle power, 5W in S4 state (note that in S4 only a small wakeup circuitry is powered on), and the active power at 100% utilization is 150W. Then, in the original configuration the total power consumption is $10x(100 + 0.2 * 150) = 1300W$. With consolidation, we instead have $3 * (100 + 0.667 * 150) + 7 * 5 = 635W$, a more than 50% saving.

5.4 Energy Management Algorithms

A smart energy management may involve use of several mechanisms used at different time scales. In the above, we discussed the idle and active power states, which can be controlled either individually for each device or as a set. The time scale aspect is crucial since the idle durations of any resource can vary over an extremely large range. For example, the CPU may experience stalls at the level of individual instructions while it is waiting for data from memory or last level cache – these stalls would be in the range of 10's to 100's of ns. Larger idle periods – in the range of microseconds to milliseconds could occur due to wait for network or storage devices. Even longer idle periods may be governed by user demands (e.g., queries that require processing) which itself involves variations over 1-100's seconds (e.g., gaps between successive queries) and over hours or longer (e.g., hourly and daily variations).

The duration of idle period is crucial when using *idle power management*, since a longer idle period allows deeper sleep, less latency impact of state transition, and more sophisticated

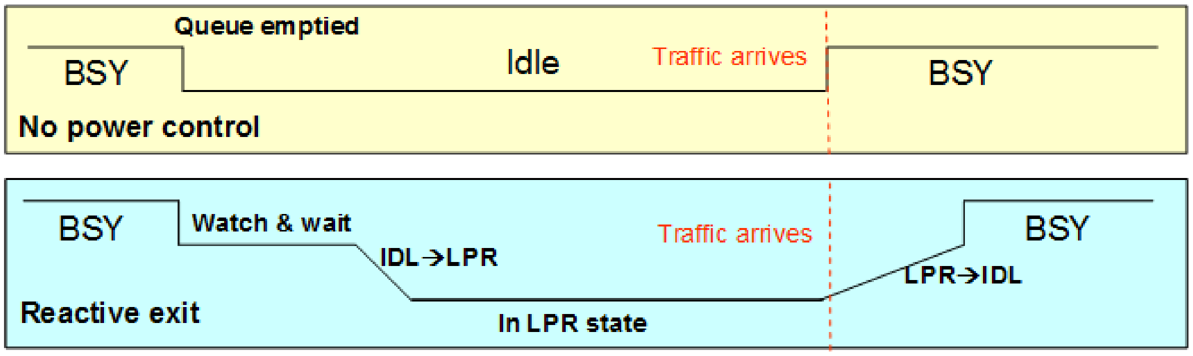

Fig 5.2: State Illustration without and with power control

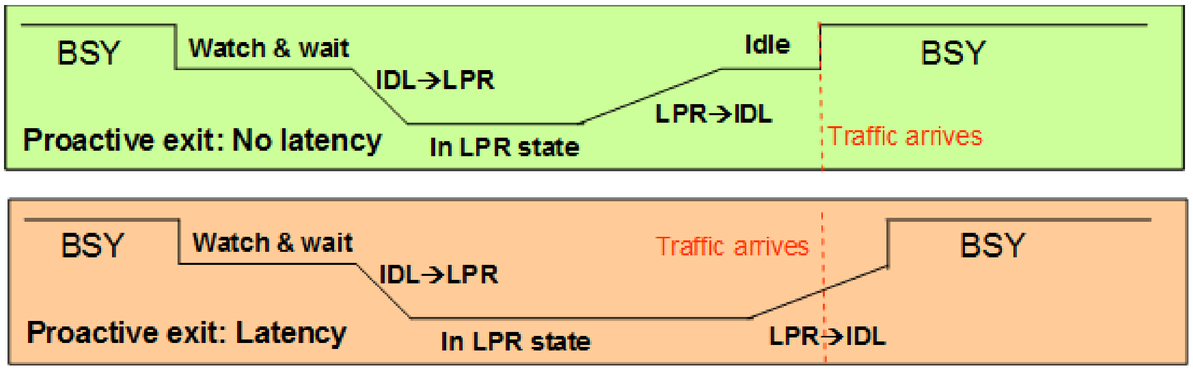

Fig 5.3: State Illustration with proactive power control

control. However, with *active power management*, the duration of the idle period does not matter; instead, what matters is how quickly the activity level (e.g., device utilization level) changes. We discuss these in the following for both short time scales (fine grain power management) and longer time scales (medium and coarse grain power management).

5.4.1 Fine Grain Power Management

Fine-grain power management refers to power management actions that can capture traffic intensity changes and idle periods ranging from 10's of ns to 10's of us. In this space, active power management may not be useful, and will not be discussed here. As for the idle power management, the very short sleep durations demand very simple, hardware based solutions. Most of the hardware sleep states discussed above (e.g., C0, C3, C6 for CPU, L0s/L1 for links, fast/slow CKE for DRAM, etc.) can be exploited here by a hardware algorithm. Note that collective control using sleep states, such as link width control, are also useful here.

The top part of Fig 5.2 shows a device (CPU core, link, DRAM rank, etc.) without any power management. In this case, the device becomes idle (IDL) once its transaction queue empties out, and then busy (BSY) again when a request (or "traffic") arrives. The bottom part of Fig 5.2 shows the simplest possible approach to using the low-power (LPR) mode. Since we do not know when the next request will arrive, we wait for a while (called "runway"), and then transition to LPR mode. The runway is a parameter of the algorithm and can be set to a fixed value or adjusted dynamically based on the device utilization. The transition takes some time as shown. Eventually, when the traffic arrives, we exit the LPR mode and become busy again. Notice that this is a *reactive* control with respect to exit from LPR mode. It has the property to let the device stay in LPR mode as long as possible, but the incoming traffic is always delayed by the exit delay. Clearly, this algorithm is trivial to implement in HW and will have negligible overhead.

Fig 5.3 shows a *proactive* variation of the algorithm where the device stays in LPR state for a certain amount of time, say T_L, and then exits proactively. Of course, if the duration T_L has not expired and traffic arrives, the device will still exit reactively from LPR mode. The figure shows two cases. In case (a) the T_L estimate falls too short, in which case the device exits LPR state prematurely, which is not good for power savings. In case (b), T_L estimate is too long, which means that the traffic experiences some delay (although the delay will always be bounded by exit latency). In order to make the proactive algorithm work properly, we need a good estimate of T_L, and this estimate needs to be continuously updated. A simple way to do so is to update T_L based on the idle periods observed in the recent past, but more detailed information about the traffic can provide better predictions.

As discussed above, a device can have multiple sleep states with increasing power savings and exit delays. The normal way to use multiple states is to first enter the shallowest state (with smallest exit latency), and then progressively move to deeper sleep states if no traffic arrives. Unfortunately, it is not possible to directly switch from one sleep state to another; instead, it is necessary to wake up the device to full idle mode and then transition it to the desired deeper sleep mode. This limits the usefulness of successive "promotion" to deeper sleep states; instead, it may be better to simply choose the most suitable state upfront based on current utilization level, and promote it further only if the idle period turns out to be extremely long.

In addition to sleep state control, the width control can also be exercised easily by the HW. An algorithm for this is described in [6]. The basic idea is to monitor the device utilization, and if the utilization crosses a predefined threshold, change the width. The algorithm needs to avoid ping-ponging (i.e., a rapid switching between high and low widths), but this is easily accomplished by introducing some hysteresis in the algorithm.

5.4.2 Medium and Coarse Grain Power Management

At medium/coarse grain, the algorithms can be implemented in firmware or software and can be more and more sophisticated as the time granularity increases. The algorithms can make use of both active and idle state controls, as discussed below.

The active state control such as DVFS reduces power by matching the throughput capability of the device to the current needs. The best known active controls are DVFS controls for CPUs – they are an integral part of power management in current systems. For example, Intel's *SpeedStep technology* and AMD's *PowerNow technology* make use of P states to dynamically switch the processor to higher numbered P state when the CPU utilization is low, and to a lower numbered P state when the utilization increases. The net effect is to keep the effective CPU utilization after change over to suitable P state at a fairly high level (say, around 70-80%) irrespective of the actual load. This allows a significant reduction in the average CPU power consumption. Both the *SpeedStep* and *PowerNow* can be characterized as reactive in nature in that they change state based on the utilization in recent past. It is possible to make use of predictive algorithms here, but may not provide any significant advantages.

It is important to note that a change in P state is not instantaneous and could require a few microseconds or more depending on how it is done. This is because a voltage change needs time to settle down, and a frequency change requires to phase locked loop (PLL) circuitry to lock the new frequency. Intel's improvement implementation, called *Enhanced SpeedStep*, tries to reduce this latency with a small increase in power. If a software based algorithm is used to decide and switch among P states, the delays could be in milliseconds or more. An important point about P states is, however, that multiple cores may be supplied voltage from the same "rail" and thus all of those cores must use the same voltage. This considerably limits the flexibility of DVFS. Furthermore, with voltage levels shrinking and getting closer to thresholds for reliable switching, there may not be much scope for reducing voltages at lower frequencies. Note that simply lowering the frequency without lowering the voltage may be detrimental from the energy perspective. To see this, consider an entirely CPU bound task which runs for 10 seconds on a 2.0GHz processor. Then the task would take 20 seconds to complete if the processor frequency is reduced to 1.0GHz. Obviously, the active power in this case will be 1/2 of the original, but the active energy will be the same (twice as long at half the rate). Moreover, the processor will now be consuming idle power for 20 seconds, instead of only 10 seconds (and perhaps placed in a low power mode rest of the time). Thus, the total energy consumption is actually larger!

The idle state control can be much like the fine grain HW algorithms described above, except that there is a scope for more sophisticated decision making here. For example, if the workload is rather stable, one could learn its characteristics and use them to determine the "runway" and the low power state to be used. In some cases, it may be possible to even eliminate the runway since a more detailed understanding of the application behavior may tell us when the application is unlikely to use a particular resource.

Although medium and coarse grain power management are often integrated together, it is worth making a clear distinction between the use of lower speeds and sleep modes (usually done at the time granularity of minutes or lower) and simply powering off the resources (which becomes attractive at the granularity of 10's of minutes or longer). Two prominent examples of the latter are: (a) consolidating servers by "packing" the workload on as few a servers as possible and shutting down the rest, and (b) copying actively used data to certain disks and spinning down the rest [7171]. Such consolidation can save a substantial amount of energy when the usage pattern is easily predictable or known. For example, once we know the low usage periods during a day (typically late night and early morning), we can decide how many servers and disks to keep active. This number may be somewhat overestimated to deal with uncertainties, and then medium grain controls can extract further savings by making use of active and passive controls.

5.5 Software Energy Efficiency

In the above, our focus has been almost exclusively on hardware energy efficiency and management techniques. Energy is ultimately consumed by transistors and wires; in fact, energy is consumed even when the transistors are not switching (i.e., not computing). This makes software level energy consumption discussion rather difficult. For example, it is not possible to associate a fixed amount of energy with basic operations such as add, multiply, data copy, etc. The problem is that the energy consumption associated with any operation depends on many hardware level details, including the number of operation units, how they are used, the location of the instruction and data (core level cache, shared cache, or memory), and how any given operation relates to other around it because of pipelining, prefetching, speculative execution, etc. Furthermore, even if one could estimate per operation energy consumption, it is not necessarily meaningful because a hardware unit that does not perform any operation may still consume some idle power, and this power consumption depends on how the unit is power managed. Because of these difficulties, it is usually not possible to consider energy consumption in the same simple way that we use for estimating the complexity of the algorithm. Nevertheless, it is of great interest to consider how to reduce the energy consumption of the system during the time the program of interest is running. In this section we discuss this issue in the context sequential programs. Parallel program related issues are discussed in the following section.

5.5.1 Algorithmic vs. Energy Efficiency

In general, if we can restructure the program or change the underlying algorithm so that it finishes more quickly, it will likely also consume less energy. In other words, a more efficient algorithm (along with its efficient implementation) should generally lead to less energy consumption. This is because the more efficient algorithm will likely do one or more of the following: (1) execute fewer instructions and/or touch less data, (2) improve the hit rates in processor cache(s) and thus reduce data movement including traffic on various interconnects, (3) reduce memory footprint and/or lay out data in memory in a way that results in more efficient accesses to memory, (4) fewer or more efficient data transfers over the network, (5) better layout and access to the disk to reduce I/O overhead, and (6) less contention for shared resources such as locks. All of these can directly reduce the power consumption by reducing the activity level in the computing infrastructure.

However, we need to be really careful and not equate shorter running time with less energy consumption. DVFS actually provides a direct contradiction to that idea – by running slower, and thereby taking longer, we save energy. One could argue (correctly), that DVFS changes the behavior of the hardware, rather than the software, and thus does not violate the idea of reducing energy consumption by making the program run faster. Nevertheless, even software restructuring to reduce the run-time may not always reduce energy consumption. This could happen because a more efficient algorithm puts more stress on the CPU and increases the energy consumption more than the amount by which it reduces the run-time. In general, the energy consumption and run-time of a program are affected differently by the choice of algorithm, data structure, data locality, caching behavior, etc. and the overall impact could be difficult to model and predict.

5.5.2 Enhancing Energy Efficieny Opportunities

It is often possible to increase energy efficiency of a program/service by enhancing opportunities for it to make use of energy saving techniques discussed above. This applies to both terminating programs (that take some inputs and then run until completion) and services (that receive an input or query, execute, and then wait for the next one). With idle power management, if we can increase the idle periods, the underlying hardware can use low power modes more effectively and thereby save energy. In a terminating program, this could be done by bunching together periods of IO, memory accesses to fetch data into the cache, and execution from the cache. In a service, this can be done by batching the queries together suitably so that multiple small idle periods turn into one larger idle period. With active power management, restructuring the workload so that the utilization of various devices can be more balanced allows energy saving by not having to change the active state too frequently.

As a concrete example, consider the case of network traffic flow coming into a switch or router. Suppose that we collect a batch of $n > 1$ packets and then forward them. In this case, we can put the switch/router into low power mode while we are collecting the next batch. If this batching period is long enough that the overhead of transition into and out of low power state is a relatively small fraction of the total idle period, better energy efficiency is achieved than by forwarding the packets one by one. Of course, the cost is extra delay caused to the packets. Note that for this solution to work, we should be able to receive and queue up the packets any time; it is only the forwarding part that can go into the low power mode. This requirement may be difficult to satisfy or require additional hardware to capture packets arriving while the port is in low power mode. Another important issue is that the batching needs to happen for all of the traffic coming into a port, rather than for only some of the flows. For example, if the port is receiving several flows, and at least one of them is too latency sensitive to allow for batching, there may not be much advantage in batching the others.

5.5.3 Data Movement vs. Computation

Computation and communications are two key functions in all of the computing technology and both need to be fast and energy efficient. Unfortunately, communications (or data transfer) has not kept pace with computing on either front. This is true at all levels starting with on-chip wires. As the width of the wires shrinks, their resistance goes up, and is already measured in Mega-Ohms per inch. This makes data movement increasingly more costly in terms of power as compared with computation. While the transistor power decreases with smaller "feature size", the wire power does not necessarily scale down due to the wires becoming thinner. Also, unless the wire length also decreases in the same proportion as the feature size (generally not true), the increased capacitance makes the wires slower. The net effect is that the energy consumed to move data on-chip by 1cm is increasing while the energy required for computation (say, addition of two numbers in registers) has been going down. This is true for all interconnects/links including interconnect among cores, DRAM to memory controller bus, etc. The net result is that more attention must be paid to data movement to both on-chip and off-chip than has been done in the past. Even the system level interconnects such as PCI-E, Ethernet, Infiniband, etc. suffer from similar issues – their speed increase and energy efficiency has not kept pace with the computing.

There are several aspects to be considered in minimizing data movement. One key issue concerns data representation. For elementary data types, it helps to minimize their size, so that more "information" can be moved using the same number of bytes. For example, signed integers that are not expected to go beyond $2^{15}-1$, are better represented as "short" rather than "int". Similarly, the floating point variables whose computation would be acceptable as single-precision (in terms of range and precision), should not be declared as double. A compact representation at larger granularity (e.g., arrays, structures, etc.) is also highly desirable provided it does not make the algorithm inefficient and thereby erase the advantages of compact representation.

Another key concept in minimizing data movement is *locality of access.* The basic idea is that if some data items are placed close together, the algorithm should be designed to access them together so that the data can be brought in larger chunks. Conversely, the data that is normally accessed together should be placed together. This applies to the algorithm design, memory allocation of variables by the compiler, placement and access of data on the disk, etc. Locality can be very difficult to achieve when the data to be accessed depends on external queries, and the query workload is highly variable.

The third key concept is a tradeoff between computation and data movement. In the past, computation was expensive and the algorithms emphasized the need to reuse what has already been computed. This is increasingly not true – it may be better to recompute the result locally yet again, instead of fetching it from elsewhere. For example, consider two nodes N_1 and N_2 in a multiprocessor system, each of which holds data items A and B in their caches. (Here A and B could be scalars or vectors.) Suppose that N_1 has already computed $A + B$, and N_2 needs it. It may be faster for N_2 to recompute it (in a separate variable) instead of requiring it to be transferred from N_1 (by accessing the variable holding $A+B$). This situation applies at higher levels as well. For an example, consider two joinable relational tables A and B stored on a disk accessible to two nodes N_1 and N_2. Suppose that both nodes have cached A and B in the memory and N_1 has computed $A \bowtie B$. Now if N_2 needs $A \bowtie B$, it may be more efficient for it to compute on its own, rather than asking N_1 to either send the result over the network or write it to the disk from where N_2 can read it.

5.5.4 Tradeoff Between Energy and Performance

For best performance, it is usually desirable to spread the load across all resource instances (e.g., servers in a cluster, cores in a CPU, all channels of DRAM, etc.) Such *load balancing* minimizes bottlenecks and hence leads to better performance. However, from the energy perspective, it is better to concentrate load on as few resource instances as possible so that the rest can be put in low power mode. The load balancing among the used resource is still important, although ideally this would be balancing of power consumption rather than load.

Restructuring of a program to enable better energy efficiency and actually exploiting this potential by power management technique itself has impact on the performance. For example, batching of requests means additional delays, which may be undesirable. Thus, the extent of permissible batching will be governed by the maximum tolerable delay. Also, any kind of power state transition causes delays, which again must be controlled. For example, placing the unused servers in S5 (hibernate) state may require 10's of seconds to resume operation, possibly followed by some task migration before normal operation can be resumed.

Given the different impact of various techniques on power consumption and performance, it is often desirable to speak of a metric that involves both performance and power. The simplest one is performance per watt, which may be reported as transactions/watt for transactional workload. Unfortunately, such a metric very much depends on the utilization level with or without power management. Without any power management, the metric will be the highest if the resource is 100% utilized and become very small as the utilization goes down to zero if the idle power consumption is significant. With aggressive power management, the maximum may be achieved at a very low performance level because at that level one could set the voltage to the lowest level and thereby gain a substantial reduction in active power. Thus performance per watt (or a similar work done per Joule of energy) needs to be interpreted carefully to be meaningful.

5.6 Parallelism vs. Energy Efficiency

Parallelism is a property of both hardware and software, and to the extent the available parallelism in the software can be mapped to that in the hardware, we can reduce the execution time and possibly the energy consumption. This section provides an overview of some of these issues.

5.6.1 Hardware vs. Software Parallelism

The hardware provides the following types of parallelisms

1. Instruction Level Parallelism (ILP), where the goal is to complete (or "retire") as many instructions as possible in one clock cycle. This is achieved by two techniques: (a) pipelining, divide the execution into a pipeline of stages, so that while stage i is working on nth instruction, stage $i-1$ could be working on $(n+1)$st instruction, etc., and (b) superscalar processing, where several independent instructions are processed in parallel by independent hardware units.

2. Data Level Parallelism (DLP), where multiple data elements (e.g., elements of a vector) are processed in parallel either by the same operation (SIMD) or different operations (MIMD). For example, two vectors can be added in the time it takes to add two elements.

3. Thread Level Parallelism (TLP), where multiple software threads execute in parallel on different cores or HW threads (usually followed by a synchronization point where the computed results are exchanged and prepared for next phase). The well-known map reduce framework is a good example of this type of parallelism.

The ILP occurs naturally in software in that the sequence of instructions in a program need not be executed in that order – the only thing that matters is ordering with respect to the dependencies that result from one instruction using the result produced by another instruction. Branches are also problematic because they force the execution to move to another spot in the program. Program ILP is automatically exploited by the current architectures. For exploiting ILP, the instruction execution is divided into multiple "stages", such as instruction fetch, decode, operand fetch, execution, and result generation. These stages work in parallel in that while one instruction is in the nth stage, the next instruction could simultaneously be in (n-1)st stage. The hardware required to handle pipelining includes interface registers between stages and extensive logic to manage dependencies, forward results, handle branches and exceptions. A deeper pipeline requires more hardware and may need more complex logic. Although a non-pipelined CPU would not optimally use the hardware, it can be largely power gated and thus does not result in much of an energy burden. The power gating of many small stages with complex interface logic is more difficult and carries more overhead. The biggest problem with deep pipelines however is the handling of branches, which cause substantial inefficiency. Superscalar designs also suffer from inefficiency if there are not enough independent instructions to pack together. In general, simpler designs (e.g., Intel Atom vs. Core processor, or Arm vs. Intel) tend to be much more energy efficient.

Vector computations occur naturally in software and can exploit the hardware vector processing capabilities easily for faster and more energy efficient processing. The energy efficiency results from less per operation overhead of vector processor vs. scalar processor and the overall shorter processing time. The overhead of properly using the vector processing units largely occurs in terms of proper code generation by the compiler, rather than at runtime. Furthermore, unused units can be easily put in low power mode.

A similar situation occurs with TLP, since the "uncore" hardware is shared among all cores (discussed in the next section) and the cores can work in parallel to finish the task much faster. Exploiting the multi-core architecture generally requires parallelism at a much coarser granularity than provided by ILP or vector processing.

5.6.2 Application Level Parallelism vs. Energy Efficiency

Application level parallelism can be achieved by spreading the operation over multiple resources such as servers, storage devices, network interfaces, etc. For example, running the same application on multiple servers can scale up the overall query processing rate up to the point where some bottleneck develops due to software contention (e.g., locks), access to shared disks, limited network bandwidth, etc. Storage system bottlenecks can be relieved by duplicating read-intensive data across multiple storage devices, or partitioning data intelligently across them. Similarly, network bottlnecks may be relieved by using multiple network interfaces.

There are two related perspectives on energy efficiency for this type of parallelism. One is that if the system is limited by some significant bottleneck, the less used resources will not be used efficiently. For example, if the throughput is limited by the storage system, the servers will experience stalls, and it may be possible to retain the same throughput by simply reducing the number of servers to the point where the storage bottleneck is relieved. The extra servers could then be shut down or put in deep sleep mode to conserve energy. The other perspective is of dynamic sizing of resources to match the needs. That is, if the resources (servers, disks, network interfaces) are not well utilized, better energy efficiency can be achieved by consolidating the workload on fewer resources (and shutting down the rest). Thus, the general principle is to size the resources such that all resources operate close to their bottleneck point, but not above it.

It is important to note that dynamic sizing may be very difficult in many cases and may itself have significant energy overhead. We will illustrate this with two examples. First, consider a cluster of web servers that are dynamically sized to match the web query load. Each time a web server is to be removed from service, we first need to direct all new queries to other servers, let it finish all existing queries and then put it to sleep. This involves energy overhead. Similarly, when a web-server is to be fired up, it needs to fetch all required content to its DRAM and processor caches, which again involves a significant energy overhead. Thus, the frequency of changes needs to be properly controlled. The second example concerns disks where dynamic movement of a lot of data is impractical. This is usually handled by staging the data on the largest set of disks needed and then varying the number of active disks. The problem here is that irrespective of whether a disk is in use or not, its data needs to be kept up to date. This requires that the disk be periodically spun up, and the data updated. The energy and reliability implications of repeated spin up/down of disks need to be considered in designing an algorithm for matching the demanded throughput with the supply.

Application level parallelism can also be used across data centers – for example, by running the application in multiple data centers in order to do the processing closer to the demand. A proper application of this idea can reduce delays, lower network bandwidth requirements, and also save energy by virtue of the increased locality and less data movement. It can also handle energy supply limitations – for example, by processing queries or tasks where the energy supply is plentiful.

5.6.3 Thread Level Parallelism

Because of the proliferation of multicore architectures, the topic of designing parallel algorithms has gained considerable importance. These algorithms are focused on completing the processing as quickly as possible on the available cores, but are limited by the available parallelism. A well formulated parallel algorithm should allow assignment of a substantial chunk of work to each core such that they can all work in parallel without any need for any synchronization or data sharing with other cores. A typical algorithm would then follow a map-reduce type of paradigm: execute in several rounds, where each round involves the following 3 steps: (a) data distribution to some subset of cores, (b) parallel computation on all these cores, and (c) shuffling or summarization of the data to prepare for the next round.

The key challenges in devising such an algorithm include: (a) coarse granularity of work assignment to the core so that a significant amount of parallel computation is done in each round, (b) ability to use all or most of the available cores, and (c) workload balancing across the cores so that the computation on all the cores finishes at about the same time. These objectives could be difficult to achieve in general, and the parallel algorithm forumulation to achieve them could make it substantially different from the sequential algorithm. Often this amounts to an algorithm that would be less efficient in time and space than the original sequential algorithm if executed on a single core machine. This difference, along with the overheads of parallel execution could make the parallel algorithm inherently less energy efficient than the sequential algorithm, even though the parallel algorithm would most likely take much less time to execute than the sequential version.

Designing the parallel algorithm to engage all cores and equalize work for them is often not possible because the number of data partitions and the size of each partition is often data dependent. For example, with parallel merge sort, the number of lists to be merged varies at different levels of the merge tree. As another example, the pivioting in quicksort invariably results in unequal size lists. This results in achivable speedup of parallel processing significantly less than the number of processors, say N, and often decreases with N.

Comparing the energy consumption of parallel vs. sequential versions of a program could itself be a bit tricky. Consider, for simplicity, a situation where all of the k cores in a system are engaged. Suppose that the sequential algorithm runs for time τ on one core, and the parallel version is able to divide much of the work into k equal parts, one allocated to each core. Still, there will be some overhead of parallelization, and we can consider this as a fixed fraction of the code, as suggested by Amdahl's Law. Accordingly, let this sequential fraction be f_a.

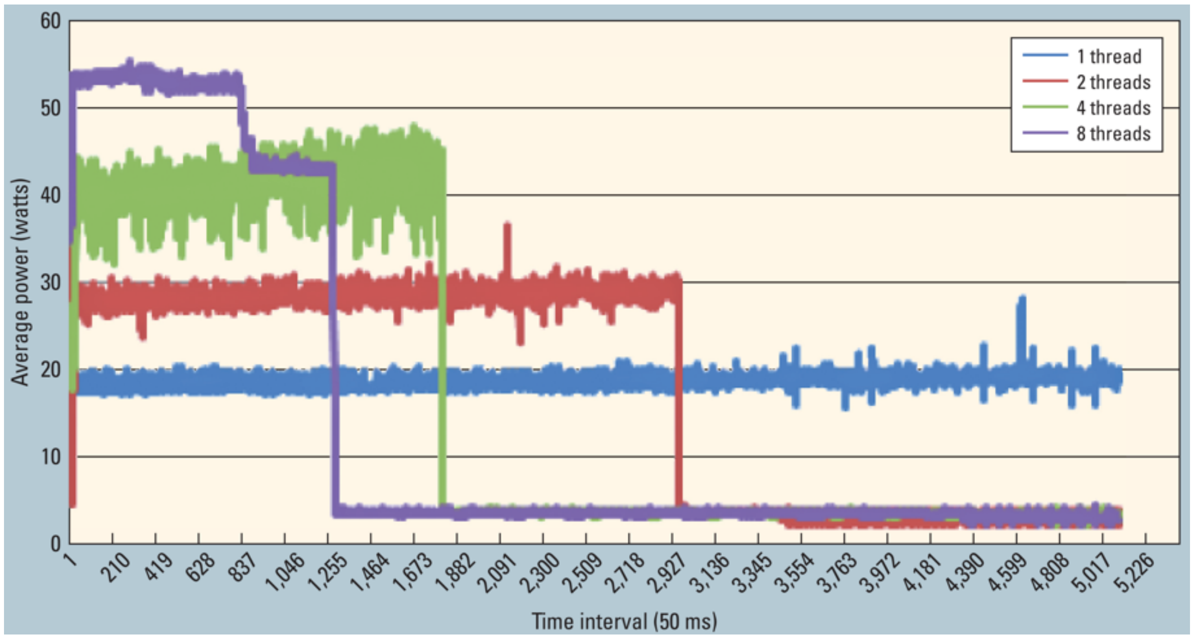

Fig 5.4: Power consumption vs. number of cores, Taken from [8]

Then the parallel algorithm will take $\tau(f_s + (1-f)/k)$ time. Suppose that the power consumed for each core when executing the algorithm is $P_A^{(c)}$ and when idle $P_I^{(c)}$. (Note that the active power does include the idle part here.) Let $P_A^{(u)}$ and $P_I^{(u)}$ denote the active and idle power consumption of "uncore", or parts of the CPU other than cores (e.g., core interconnect, memory controller, etc.). Since the $(k-1)$ cores must remain idle in the sequential program for the entire period τ, the energy consumption of the sequential program is given by:

$$E_S = [P_A^{(u)} + P_A^{(c)} + (k-1) * P_I^{(c)}] \times \tau \tag{5.3}$$

Since all the cores are active simultaneously for a parallel algorithm, its energy consumption is given by:

$$E_P = [P_A^{(u)} + k \times P_A^{(c)}] \times \tau[f_s + (1-f_s)/k] = [P_A^{(u)}/k + P_A^{(c)}] \times \tau[kf_s + (1-f_s)] \tag{5.4}$$

Notice that in this comparison, we assume that the idle cores are not doing anything, and thus extra power is consumed even if they are put in a low power mode. If we instead assume that idle cores can be used for something else, their energy consumption should not be charged to the sequential program. With our assumptions, the comparison comes down to the following tradeoff: (a) additional "uncore" power spent by the sequential algorithm, and (b) additional overhead of the parallel implementation.

$$[P_A^{(u)}/k + P_A^{(c)}] \times [kf_s + (1 - f_s)] < [P_A^{(u)} + P_A^{(c)}] \tag{5.5}$$

which can be simplified to yield:

$$P_A^{(c)} < P_A^{(u)}(1/f_s - 1)/k \tag{5.6}$$

For example, if the core and uncore power are identical, this equation reduced to $k < (1/f_s - 1)$. This means that so long as the number of cores is not too large, the reduction in execution time will be more than the overhead of parallelism, and hence the power consumption of the parallel program will be lower. However, with large number of cores, this is not true and the parallel program will not provide any power advantage. As an illustration, Fig. **??** shows the power consumption for the Cinebench 11.5 benchmark run on different number of cores. This figure is taken from [8], which talks about techniques for making software energy efficient. Notice the Amdahl's law in play – the execution time with 2 cores is more than 1/2 of execution time for 1 core, and similarly for 4 vs. 2 cores, and 8 vs. 4 cores. Similarly, 2 core power is less than twice that of single core power because not all the power consumed in the core.

In the more general situation where the different cores do different amount of work, the cores that finish earlier will remain idle, and will consume extra energy even if they are placed in a low power mode. This would make the parallel program less efficient.

5.6.4 Power Management of Parallel Computations

With both sequential and parallel programs, smart energy management can be exploited to reduce the energy consumption of the cores that are underutilized. In particular, cores that are idle can autonomously go into a suitable sleep mode using a suitable "runway".

This also includes progressive algorithms where the core first enters a shallow sleep mode and then is promoted to a deeper sleep. However, such autonomous actions are not the most efficient, and an explicit control by the program (or a middleware that is aware of the program behavior) can do a much better job by properly scheduling tasks and taking energy management actions with a better knowledge of task schedules.

In particular, consider the scenario above where multiple cores do some computation in parallel and then "join" at the end. The cores may finish their work at different times either because the cores have nonhomogenous characteristics or the work given to them is unequal. In this case, the cores that can finish early can be managed in the following 3 ways:

1. Schedule another unrelated task on the core and preempt this task when other cores reach the synchronization point. This approach is workable if the core has to wait for a long time before others reach the synchronization point, otherwise it could lead to significant switching back and forth and corresponding inefficiencies due to disturbance to the working sets. Also, the unrelated tasks that we run must be of the type that does not have any strict QoS requirements, since they can be preempted whenever the other tasks are ready.

2. Put the core in a suitable low power mode until other cores complete.

3. Stretch the completion time of cores that will finish fast by using DVFS controls.

In order to compare the last two options with respect to energy efficiency, consider two entirely CPU-bound tasks T1 and T2 running on identical cores that need to "join" when finished. Suppose that task T2 takes only half the time as task T1. Then the energy consumption of the two cores, denoted E_1 and E_2 respectively, is given by

$$\begin{aligned} E_1 &= (I_L.V_0 + 1/2.C.V_0^2.f_0)\tau \\ E_2 &= (I_L.V_0 + 1/2.C.V_0^2.f_0)\tau/2 + I_L.V_0.\alpha_{lp}.\tau/2 \end{aligned} \tag{5.7}$$

where V_0 is the normal voltage, f_0 the normal frequency, and I_L is the leakage current.

We assume that the leakage current reduces by a factor α_{lp} in the low power state. Now, if the core frequency for T2 is halved, i.e., core runs at $f_1 = f_0/2$, both tasks will finish at the same time. Let $V_1 < V_0$ denote the voltage compatible with the halved frequency. Then the energy consumption of T2 is given by

$$E_2' = (I_L.V_1 + 1/2.C.V_1^2.f_0/2)\tau \tag{5.8}$$

Generally, we expect $E_2' < E_2$ if the frequency halving allows for significant voltage reduction. For instance, consider the earlier example where $f_0 = 2.0$GHz with $V_0 = 1.2$ Volts, $V_1 = 0.9$ Volts, $I_L = 7.5$ Amp, and effective Capacitance $(C) = 10$ nanoFarad. Then, $E_1 = 23.4W$ as before, but

$$\begin{aligned} E_2/\tau &= (7.5 \times 1.2 + 5 \times 1.2^2 \times 2) \times 0.5 + 7.5 \times 1.2 \times 0.5 \times \alpha_{lp} = 11.7 + 4.5\alpha_{lp}W \\ E_2'/\tau &= (7.5 \times 0.9 + 5 \times 0.9^2 \times 1) = 10.8W \end{aligned} \tag{5.9}$$

It is seen that DVFS provides lower energy consumption (E_2') here irrespective of the value of α_{lp}. However, if the voltage reduction is limited to 1.1 Volts at the 1/2 frequency, we have $E_2'/\tau = 14.3W$. In this case, we need $\alpha_{lp} > 0.578$, for the DVFS control to beat the idle power control. In reality, we expect the α_{lp} to be much smaller than this threshold, and hence low power control will be preferred.

5.7 Energy Adaptation

Until now we have largely considered opportunistic reduction of energy consumption with as little impact on performance as possible. While this is very valuable, there are many situations where such an approach is inadequate. At the architectural level, the power and thermal densities continue to grow due to shrinking feature size and it is becoming necessary to limit the power consumption to ensure that the thermal limits are not exceeded. At the rack level, *power capping* may be necessary to ensure that the power circuits remain within their capacity. A typical reason for stress on power circuit capacity is that the physical infrastructure in a data center (e.g., racks, power distribution, cooling, etc.) is originally installed based on the requirements of the servers at that time. However, with increasing density of the servers, the power limits may be exceeded in the future, thereby requiring power capping. At the data center level, power capping may be necessary either due to use of renewable power supply (that naturally fluctuates) or due to the inadequacy of power and cooling infrastructure.

The worst case situation of power and cooling requirements in a data center will occur if all of the components of all the servers are simultaneously working at peak capacity. However, this is unrealistic and undesirable from the infrastructure cost perspective. Most data center servers run at a very low utilization typically – 10-20% range, and may hit 70-80% only a few times in a year. In fact, most data center operators will upgrade/expand their computing capacity much before it threatens to be a frequent bottleneck. Furthermore, even if the utilization of one component (say, CPU) goes to 100%, it is highly unlikely that the others (e.g., memory, network or the storage) could simultaneously be working at their maximum capacity. In particular, if the CPU is doing heavy IO or pulling in a lot of data from the memory, it will mostly be stalling, and not executing to the best of its capability.

In the so called “co-lo” (colocation) environments that are becoming popular, multiple companies lease and operate servers from a single server-farm operator. In such an environment, a simultaneous peak usage of all the servers belonging to various client companies is even less likely. Thus, it is highly desirable to design the power infrastructure in the data centers to a value that is substantially below the theoretical peak (often by a factor of 3 or more). The same goes for the cooling infrastructure – provisioning enough cooling capacity to handle the worst case heat generation situation is usually not sensible.

The underprovisioning of the power/cooling infrastructure does require the ability to handle those rare situations where the demand may exceed the supply. It is important to note here the relationship between power and cooling. If the cooling capacity is inadequate, the power consumption must be capped (even if there is no power constraint) so that no thermal emergencies are created. An intelligent power capping can adapt the system to the available power/cooling capacities without any significant impact on the performance. In fact, with an intelligent adaptation mechanism, it is possible and highly desirable to deliberately underdesign the power and cooling infrastructure so that a suitable balance between cost (both infrastructure and operational) and risk of violating quality of service (QoS) requirements is achieved. Such a tradeoff can result in huge cost savings with only a minor increase in the risk of degraded QoS.

The key difference between adaptation and the opportunistic energy management is that the former is a *mandatory reduction in energy consumption* and can only be done by compromising on some performance aspect such as delay or throughput. For this reason, it needs to be performed carefully, else the performance impact could be substantial. In particular, the energy deficit must be properly distributed among various resources with an eye towards the requirements and priorities of the applications using those resources. For example, consider a map-reduce application running on N servers. Such applications divide the work among all servers (map phase) and eventually collect/merge results (reduce phase). For the best performance, all servers should finish at about the same time, and under energy capping, it is necessary to allocate budgets so that this will be the case. A haphazard energy capping of the servers may result in significant imbalance and thus affect both the performance and the energy efficiency of the application.

A similar situation arises if multiple cores used to exploit thread level parallelism are energy capped – the energy budget of each core (and hence the DVFS controls used by it) should assure the balance. Finally, a balance is necessary across resources of various types as well. For example, running the memory or links at lower frequency while the cores are running at full frequency would result in CPU stalls and thus slower progress than if the energy budgets were balanced. Similarly, if the storage system in a data center is not power managed but the servers are, this would result in suboptimal performance, and hence less work done under energy limited situations than is possible.

It follows from the above discussion that there are two major issues in adaptation to energy limitations: (a) how to estimate suitable power budget for each system, subsystem and down to individual resources (CPU cores, links, DRAM channels, DRAMs, IO controllers, disks, etc.), and (b) how to apply power management in order to keep power consumption very close to the budget. For the first problem, we ideally want to assign power budgets so that it is possible to achieve the balance discussed above. Simple models that express power consumption as a linear equation $ax + b$ as a function of performance (x) are often used for this. (Here b represents the idle power and a is the power consumption per unit of work.) Although such an approach is often adequate, it is important to note that nonlinearities often arise when there is a resource bottleneck or contention for a shared resource. For example, while we may be able to use $ax + b$ type of expression for programs running individually on a machine, nonlinearities may arise when multiple such programs run concurrently on the same machine because of contention for cache or other resources.

An additional complication arises in energy adaptation due to variability in the workload itself. If the workload (e.g., rate or type of queries arriving at a server) varies, so will the energy consumption. Thus a budget computed initially cannot be kept constant even if the energy availability does not change; instead, it needs to vary with the workload as well. A too frequent a change is itself undesirable because a change in energy budget may require a change of the DVFS state, redirection of queries to other servers, or even migration of applications to other servers. A poorly designed control mechanism could result in ping-ponging, i.e., reduction of energy budget (and corresponding corrective actions) followed by an increase, followed by a reduction, etc. Such a behavior must be avoided.

Several of these issues have been explored in [3] which discusses the notion of "energy adaptive computing" in detail [4]. There are many other investigations related to energy capping; for example, Sharma et al. [9] propose a scheme to handle intermittent energy constraints by power cycling servers. The scheme is a purely power driven management scheme and independent of workload demands. An interesting aspect of this work is to pay attention to the energy surplus periods as well, when it may be possible to do additional computation, or speculatively prefetch data so that less data transfer activity takes place during the low energy periods.

5.8 Emerging Issues and Outlook

For a long time, the semiconductor industry thrived on the voltage reduction each time the technology moved to the next lower "feature size". This is known as "Dennard Scaling", introduced by Robert Dennard in 1974. The basic idea is as follows: Let's say we reduce the feature size by a factor of $\sqrt{2}$, which is typically how generations of semiconductor processes advance. An example of this is feature size going from 90 nm (nanometer) to 64 nm in late 1990's. Ideally, the reduction in the feature size reduces the capacitance C by the same factor and allows voltage reduction by $\sqrt{2}$ as well. Thus the overall dynamic power reduces by a factor of 2.0. Now, the feature size reduction allows twice as many transistors to be packed in the same chip, and we can raise the frequency by a factor of $\sqrt{2}$, which by itself would increase the power consumption by a factor of 2.0. But the two aspects combined result in *a new chip with same power consumption as before, but with twice as many transistors, each operating 1.414 times faster!*

Denard scaling started to break down in early 2000's and stopped around 2004. The reasons for this include the inability to lower voltage much, challenges in reducing transistor capacitance, difficulties in increasing frequency, and the difficult problem of increasing wire resistance and capacitance. This gave rise to growth in the "other dimension", i.e., the number of cores rather than performance of a single core. In fact, Moore's law was redefined to refer to the computing power of all the cores and thus continued unabated. Recently, even this trend has proved unsustainable. With the industry going for 7 nm technology next, the transistor widths are only a few tens of atoms wide (one nm = 3 Si atoms), which brings in numerous technological challenges including very high power densities (power dissipated per unit area). In fact, the technology is already at a point where we can easily put hundreds of CPU cores on a chip, but powering them all simultaneously is not possible.

Lowering the power density requires further lowering of the voltage. This is challenging since voltages are already close to the threshold required for reliable switching, and lowering them further introduces timing errors. Such errors must be detected and handled either by repeating the operation or by introducing additional delays to avoid the errors. This results in concerns of unreliability and residual “silent errors” in the computations – errors that somehow escape the mechanisms designed to catch them.

An added problem is that at such small feature sizes, it is extremely difficult to pattern each transister or wire accurately, which means that supposedly identical CPU cores, caches, logic units, etc. show considerable variability in their characteristics. This essentially makes the entire chip very heterogeneous, and poses challenges in using it optimally both in terms of its performance capabilities and power/thermal limits. In particular, some of the cores may run reliably only at much lower frequencies than the target frequency, while others may exceed their normal TDP (total dissipated power). In the past, such cores would be simply disabled, and the CPU sold with fewer cores, but this could reduce the yields considerably going forward.

We already discussed the need to minimize data movement for both performance and energy reasons, but effective techniques to simultaneously handle both data movement and computation remain unresolved. The traditional notions of algorithmic efficiency only concern computational complexity, rather than the data movement. A proper consideration of both requires new ways of designing algorithms and evaluating their complexity both in terms of performance and energy.

In short, the importance of power/energy considerations in both hardware and software design will continue to increase, and so will the additional challenges brought about by smaller feature sizes and low voltages. The main issues are high variability and increasing the chance of unreliability or “silent errors” which must be tackled along with the energy management.

REFERENCES

[1] Dennis Colarelli and Dirk Grunwald. Massive arrays of idle disks for storage archives. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[2] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *ISLPED '07: Proceedings of the 2007 international symposium on Low power electronics and design*, 2007.

[3] K. Kant and M. Murugan and D. H. C. Du. Willow: A Control System for Energy and Thermal Adaptive Computing. In *IPDPS '11*, 2011.

[4] K. Kant, M. Murugan and D. H. C. Du. Enhancing data center sustainability through energy adaptive computing. *ACM JETC (Special Issue)*, April 2012.

[5] Krishna Kant. A control scheme for batching dram requests to improve power efficiency. In *Proc. of ACM SIGMETRICS*, pages 139–140, 2011.

[6] Krishna Kant. Multi-state power management of communication links. In *Communication Systems and Networks (COMSNETS), 2011 Third International Conference on*, pages 1–10. IEEE, 2011.

[7] Eduardo Pinheiro and Ricardo Bianchini. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, New York, NY, USA, 2004. ACM.

[8] M. Sabharwal, A. Agrawal, and G. Metri. Enabling green it through energy-aware software. *IT Professional*, 15(1):19–27, Jan 2013.

[9] Navin Sharma, Sean Barker, David Irwin, and Prashant Shenoy. Blink: managing server clusters on intermittent power. In *Proceedings of the sixteenth international conference*

on Architectural support for programming languages and operating systems (ASPLOS), 2011.