Power Reduction Techniques For Microprocessor Systems

VASANTH VENKATACHALAM AND MICHAEL FRANZ

University of California, Irvine

Power consumption is a major factor that limits the performance of computers. We survey the "state of the art" in techniques that reduce the total power consumed by a microprocessor system over time. These techniques are applied at various levels ranging from circuits to architectures, architectures to system software, and system software to applications. They also include holistic approaches that will become more important over the next decade. We conclude that power management is a multifaceted discipline that is continually expanding with new techniques being developed at every level. These techniques may eventually allow computers to break through the "power wall" and achieve unprecedented levels of performance, versatility, and reliability. Yet it remains too early to tell which techniques will ultimately solve the power problem.

Categories and Subject Descriptors: C.5.3 [**Computer System Implementation**]: Microcomputers—*Microprocessors*; D.2.10 [**Software Engineering**]: Design—*Methodologies*; I.m [**Computing Methodologies**]: Miscellaneous

General Terms: Algorithms, Design, Experimentation, Management, Measurement, Performance

Additional Key Words and Phrases: Energy dissipation, power reduction

1. INTRODUCTION

Computer scientists have always tried to improve the performance of computers. But although today's computers are much faster and far more versatile than their predecessors, they also consume a lot of power; so much power, in fact, that their power densities and concomitant heat generation are rapidly approaching levels comparable to nuclear reactors (Figure 1). These high power densities impair chip reliability and life expectancy, increase cooling costs, and, for large

Parts of this effort have been sponsored by the National Science Foundation under ITR grant CCR-0205712 and by the Office of Naval Research under grant N00014-01-1-0854.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the National Science foundation (NSF), the Office of Naval Research (ONR), or any other agency of the U.S. Government.

The authors also gratefully acknowledge gifts from Intel, Microsoft Research, and Sun Microsystems that partially supported this work.

Authors' addresses: Vasanth Venkatachalam, School of Information and Computer Science, University of California at Irvine, Irvine, CA 92697-3425; email: vvenkata@uci.edu; Michael Franz, School of Information and Computer Science, University of California at Irvine, Irvine, CA 92697-3425; email: franz@uci.edu.

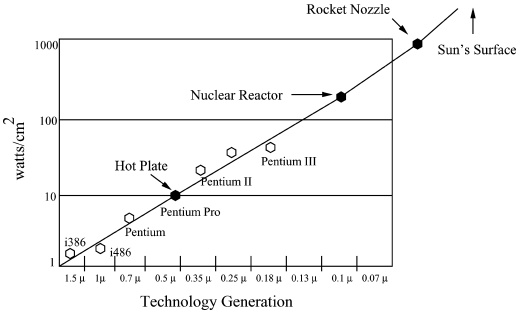

Fig. 1. Power densities rising. Figure adapted from Pollack 1999.

data centers, even raise environmental concerns.

At the other end of the performance spectrum, power issues also pose problems for smaller mobile devices with limited battery capacities. Although one could give these devices faster processors and larger memories, this would diminish their battery life even further.

Without cost effective solutions to the power problem, improvements in micro-processor technology will eventually reach a standstill. Power management is a multidisciplinary field that involves many aspects (i.e., energy, temperature, reliability), each of which is complex enough to merit a survey of its own. The focus of our survey will be on techniques that reduce the total power consumed by typical microprocessor systems.

We will follow the high-level taxonomy illustrated in Figure 2. First, we will define power and energy and explain the complex parameters that dynamic and static power depend on (Section 2). Next, we will introduce techniques that reduce power and energy (Section 3), starting with circuit (Section 3.1) and architectural techniques (Section 3.2, Section 3.3, and Section 3.4), and then moving on to two techniques that are widely applied in hardware and software, *dynamic voltage scaling* (Section 3.5) and *resource hibernation* (Section 3.6). Third, we will examine what compilers can do to manage power (Section 3.7). We will then discuss recent work in application level power management (Section 3.8), and recent efforts (Section 3.9) to develop a holistic solution to the power problem. Finally, we will discuss some commercial power management systems (Section 3.10) and provide a glimpse into some more radical technologies that are emerging (Section 3.11).

2. DEFINING POWER

Power and energy are commonly defined in terms of the work that a system performs. *Energy* is the total amount of work a system performs over a period of time, while *power* is the rate at which the system performs that work. In formal terms,

$$P = W/T \tag{1}$$

$$E = P * T, \tag{2}$$

where P is power, E is energy, T is a specific time interval, and W is the total work performed in that interval. Energy is measured in *joules*, while power is measured in *watts*.

These concepts of work, power, and energy are used differently in different contexts. In the context of computers, work involves activities associated with running programs (e.g., addition, subtraction, memory operations), power is the rate at which the computer consumes electrical energy (or dissipates it in the form of heat) while performing these activities, and energy is the total electrical energy the computer consumes (or dissipates as heat) over time.

This distinction between power and energy is important because techniques that reduce power do not necessarily reduce energy. For example, the power consumed by a computer can be reduced by halving the clock frequency, but if the computer then takes twice as long to run the same programs, the total energy consumed will be similar. Whether one should reduce power or energy depends on the context. In mobile applications, reducing energy is often more important because it increases the battery lifetime. However, for other systems (e.g., servers), temperature is a larger issue. To keep the temperature within acceptable limits, one would need to reduce instantaneous power regardless of the impact on total energy.

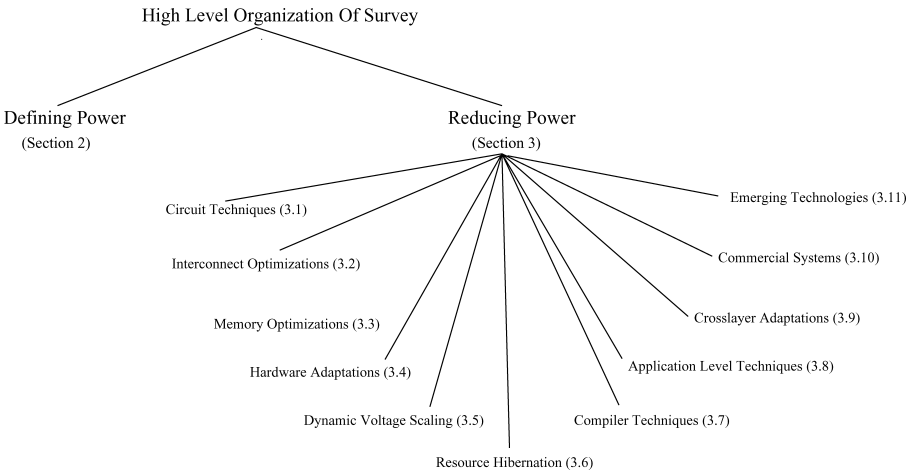

Fig. 2. Organization of this survey.

2.1. Dynamic Power Consumption

There are two forms of power consumption, *dynamic power consumption* and *static power consumption*. Dynamic power consumption arises from circuit activity such as the changes of inputs in an adder or values in a register. It has two sources, switched capacitance and short-circuit current.

Switched capacitance is the primary source of dynamic power consumption and arises from the charging and discharging of capacitors at the outputs of circuits.

Short-circuit current is a secondary source of dynamic power consumption and accounts for only 10-15% of the total power consumption. It arises because circuits are composed of transistors having opposite polarity, negative or *NMOS* and positive or *PMOS*. When these two types of transistors switch current, there is an instant when they are simultaneously on, creating a short circuit. We will not deal further with the power dissipation caused by this short circuit because it is a smaller percentage of total power, and researchers have not found a way to reduce it without sacrificing performance.

As the following equation shows, the more dominant component of dynamic power, switched capacitance ($P_{dynamic}$), depends on four parameters namely, supply voltage (V), clock frequency (f), physical capacitance (C) and an activity factor (a) that relates to how many $0 \rightarrow 1$ or $1 \rightarrow 0$ transitions occur in a chip:

$$P_{dynamic} \sim aCV^2 f. \tag{3}$$

Accordingly, there are four ways to reduce dynamic power consumption, though they each have different tradeoffs and not all of them reduce the total energy consumed. The first way is to reduce the physical capacitance or stored electrical charge of a circuit. The physical capacitance depends on low level design parameters such as transistor sizes and wire lengths. One can reduce the capacitance by reducing transistor sizes, but this worsens performance.

The second way to lower dynamic power is to reduce the switching activity. As computer chips get packed with increasingly complex functionalities, their switching activity increases [De and Borkar 1999], making it more important to develop techniques that fall into this category. One popular technique, *clock gating*, gates the clock signal from reaching idle functional units. Because the clock network accounts for a large fraction of a chip's total energy consumption, this is a very effective way of reducing power and energy throughout a processor and is implemented in numerous commercial systems

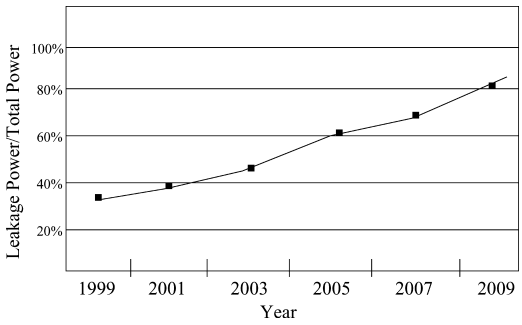

Fig. 3. ITRS trends for leakage power dissipation. Figure adapted from Meng et al., 2005.

including the Pentium 4, Pentium M, Intel XScale and Tensilica Xtensa, to mention but a few.

The third way to reduce dynamic power consumption is to reduce the clock frequency. But as we have just mentioned, this worsens performance and does not always reduce the total energy consumed. One would use this technique only if the target system does not support voltage scaling and if the goal is to reduce the peak or average power dissipation and indirectly reduce the chip's temperature.

The fourth way to reduce dynamic power consumption is to reduce the supply voltage. Because reducing the supply voltage increases gate delays, it also requires reducing the clock frequency to allow the circuit to work properly.

The combination of scaling the supply voltage and clock frequency in tandem is called *dynamic voltage scaling* (DVS). This technique should ideally reduce dynamic power dissipation cubically because dynamic power is quadratic in voltage and linear in clock frequency. This is the most widely adopted technique. A growing number of processors, including the Pentium M, mobile Pentium 4, AMD's Athlon, and Transmeta's Crusoe and Efficieon processors allow software to adjust clock frequencies and voltage settings in tandem. However, DVS has limitations and cannot always be applied, and even when it can be applied, it is nontrivial to apply as we will see in Section 3.5.

2.2. Understanding Leakage Power Consumption

In addition to consuming dynamic power, computer components consume *static power*, also known as *idle power* or *leakage*. According to the most recently published industrial roadmaps [ITRSRoadMap], leakage power is rapidly becoming the dominant source of power consumption in circuits (Figure 3) and persists whether a computer is active or idle. Because its causes are different from those of dynamic power, dynamic power reduction techniques do not necessarily reduce the leakage power.

As the equation that follows illustrates, leakage power consumption is the product of the supply voltage (V) and *leakage current* (I_{leak}), or parasitic current, that flows through transistors even when the transistors are turned off.

$$P_{leak} = V I_{leak}. \tag{4}$$

To understand how leakage current arises, one must understand how transistors work. A transistor regulates the flow of current between two terminals called the *source* and the *drain*. Between these two terminals is an insulator, called the channel, that resists current. As the voltage at a third terminal, the *gate*, is increased, electrical charge accumulates in the channel, reducing the channel's resistance and creating a path along which electricity can flow. Once the gate voltage is high enough, the channel's polarity changes, allowing the normal flow of current between the source and the drain. The threshold at which the gate's voltage is high enough for the path to open is called the *threshold voltage*.

According to this model, a transistor is similar to a water dam. It is supposed to allow current to flow when the gate voltage exceeds the threshold voltage but should otherwise prevent current from flowing. However, transistors are imperfect. They leak current even when the gate voltage is below the threshold voltage. In fact, there are six different types of current that leak through a transistor. These

include reverse-biased-junction leakage, gate-induced-drain leakage, subthreshold leakage, gate-oxide leakage, gate-current leakage, and punch-through leakage. Of these six, subthreshold leakage and gate-oxide leakage dominate the total leakage current.

Gate-oxide leakage flows from the gate of a transistor into the substrate. This type of leakage current depends on the thickness of the oxide material that insulates the gate:

$$I_{ox} = K_2 W \left(\frac{V}{T_{ox}}\right)^2 e^{-\alpha \frac{T_{ox}}{V}}. \qquad (5)$$

According to this equation, the gate-oxide leakage I_{ox} increases exponentially as the thickness T_{ox} of the gate's oxide material decreases. This is a problem because future chip designs will require the thickness to be reduced along with other scaled parameters such as transistor length and supply voltage. One way of solving this problem would be to insulate the gate using a high-k dialectric material instead of the oxide materials that are currently used. This solution is likely to emerge over the next decade.

Subthreshold leakage current flows between the drain and source of a transistor. It is the dominant source of leakage and depends on a number of parameters that are related through the following equation:

$$I_{sub} = K_1 W e^{\frac{-V_{th}}{nT}} \left(1 - e^{\frac{-V}{T}}\right). \qquad (6)$$

In this equation, W is the gate width and K and n are constants. The important parameters are the supply voltage V, the threshold voltage V_{th}, and the temperature T. The subthreshold leakage current I_{sub} increases exponentially as the threshold voltage V_{th} decreases. This again raises a problem for future chip designs, because as technology scales, threshold voltages will have to scale along with supply voltages.

The increase in subthreshold leakage current causes another problem. When the leakage current increases, the temperature increases. But as the Equation (6) shows, this increases leakage further, causing yet higher temperatures. This vicious cycle is known as *thermal runaway*. It is the chip designer's worst nightmare.

How can these problems be solved? Equations (4) and (6) indicate four ways to reduce leakage power. The first way is to reduce the supply voltage. As we will see, *supply voltage reduction* is a very common technique that has been applied to components throughout a system (e.g., processor, buses, cache memories).

The second way to reduce leakage power is to reduce the size of a circuit because the total leakage is proportional to the leakage dissipated in all of a circuit's transistors. One way of doing this is to design a circuit with fewer transistors by omitting redundant hardware and using smaller caches, but this may limit performance and versatility. Another idea is to reduce the effective transistor count *dynamically* by cutting the power supplies to idle components. Here, too, there are challenges such as how to predict when different components will be idle and how to minimize the overhead of shutting them on or off. This, is also a common approach of which we will see examples in the sections to follow.

The third way to reduce leakage power is to cool the computer. Several cooling techniques have been developed since the 1960s. Some blow cold air into the circuit, while others refrigerate the processor [Schmidt and Notohardjono 2002], sometimes even by costly means such as circulating cryogenic fluids like liquid nitrogen [Krane et al. 1988]. These techniques have three advantages. First they significantly reduce subthreshold leakage. In fact, a recent study [Schmidt and Notohardjono 2002] showed that cooling a memory cell by 50 degrees Celsius reduces the leakage energy by five times. Second, these techniques allow a circuit to work faster because electricity encounters less resistance at lower temperatures. Third, cooling eliminates some negative effects of high temperatures, namely the degradation of a chip's reliability and life expectancy. Despite these advantages,

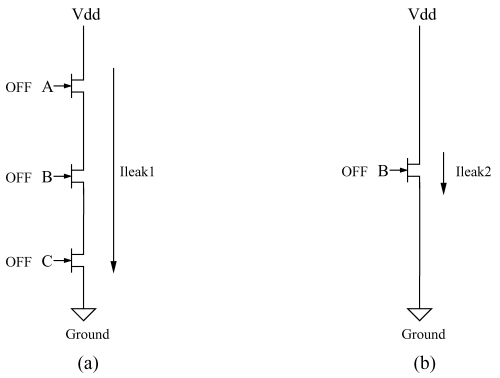

Fig. 4. The transistor stacking effect. The circuits in both (a) and (b) are leaking current between Vdd and Ground. However, Ileak1, the leakage in (a), is less than Ileak2, the leakage in (b). Figure adapted from Butts and Sohi 2000.

there are issues to consider such as the costs of the hardware used to cool the circuit. Moreover, cooling techniques are insufficient if they result in wide temperature variations in different parts of a circuit. Rather, one needs to prevent hotspots by distributing heat evenly throughout a chip.

2.2.1. Reducing Threshold Voltage. The fourth way of reducing leakage power is to increase the threshold voltage. As Equation (6) shows, this reduces the subthreshold leakage exponentially. However, it also reduces the circuit's performance as is apparent in the following equation that relates frequency (f), supply voltage (V), and threshold voltage (V_{th}) and where α is a constant:

$$f \infty \frac{(V - V_{th})^{\alpha}}{V}. \qquad (7)$$

One of the less intuitive ways of increasing the threshold voltage is to exploit what is called the *stacking effect* (refer to Figure 4). When two or more transistors that are switched off are stacked on top of each other (a), then they dissipate less leakage than a single transistor that is turned off (b). This is because each transistor in the stack induces a slight *reverse bias* between the gate and source of the transistor right below it,

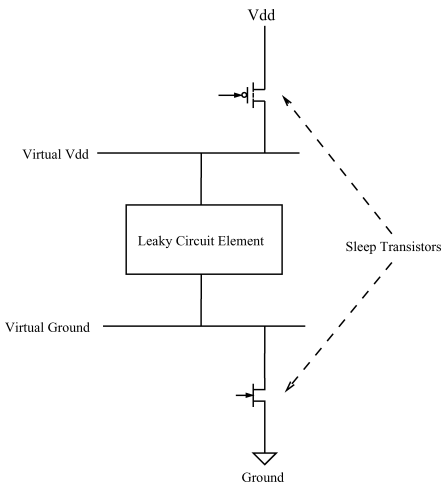

Fig. 5. Multiple threshold circuits with sleep transistors.

and this increases the threshold voltage of the bottom transistor, making it more resistant to leakage. As a result, in Figure 4(a), in which all transistors are in the Off position, transistor B leaks less current than transistor A, and transistor C leaks less current than transistor B. Hence, the total leakage current is attenuated as it flows from Vdd to the ground through transistors A, B, and C. This is not the case in the circuit shown in Figure 4 (b), which contains only a single off transistor.

Another way to increase the threshold voltage is to use *Multiple Threshold Circuits With Sleep Transistors* (MTCMOS) [Calhoun et al. 2003; Won et al. 2003] . This involves isolating a leaky circuit element by connecting it to a pair of virtual power supplies that are linked to its actual power supplies through *sleep transistors* (Figure 5). When the circuit is active, the sleep transistors are activated, connecting the circuit to its power supplies. But when the circuit is inactive, the sleep transistors are deactivated, thus disconnecting the circuit from its power supplies. In this inactive state, almost no leakage passes through the circuit because the sleep transistors have high threshold voltages. (Recall that subthreshold leakage drops exponentially with a rise in threshold voltage, according to Equation (6).)

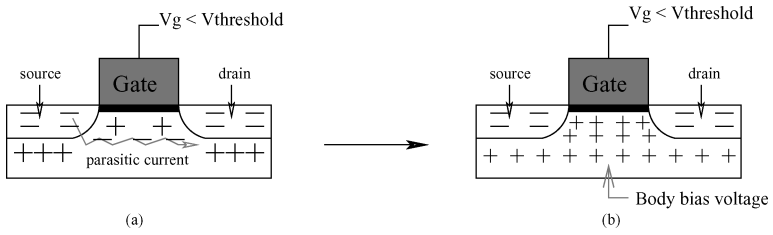

Fig. 6. Adaptive body biasing.

This technique effectively confines the leakage to one part of the circuit, but is tricky to implement for several reasons. The sleep transistors must be sized properly to minimize the overhead of activating them. They cannot be turned on and off too frequently. Moreover, this technique does not readily apply to memories because memories lose data when their power supplies are cut.

A third way to increase the threshold is to employ dual threshold circuits. *Dual threshold circuits* [Liu et al. 2004; Wei et al. 1998; Ho and Hwang 2004] reduce leakage by using high threshold (low leakage) transistors on noncritical paths and low threshold transistors on critical paths, the idea being that noncritical paths can execute instructions more slowly without impairing performance. This is a difficult technique to implement because it requires choosing the right combination of transistors for high-threshold voltages. If too many transistors are assigned high threshold voltages, the noncritical paths in the circuit can slow down too much.

A fourth way to increase the threshold voltage is to apply a technique known as *adaptive body biasing* [Seta et al. 1995; Kobayashi and Sakurai 1994; Kim and Roy 2002]. Adaptive body biasing is a runtime technique that reduces leakage power by dynamically adjusting the threshold voltages of circuits depending on whether the circuits are active. When a circuit is not active, the technique increases its threshold voltage, thus saving leakage power exponentially, although at the expense of a delay in circuit operation. When the circuit is active, the technique decreases the threshold voltage to avoid slowing it down.

To adjust the threshold voltage, adaptive body biasing applies a voltage to the transistor's body known as a *body bias voltage* (Figure 6). This voltage changes the polarity of a transistor's channel, decreasing or increasing its resistance to current flow. When the body bias voltage is chosen to fill the transistor's channel with positive ions (b), the threshold voltage increases and reduces leakage currents. However, when the voltage is chosen to fill the channel with negative ions, the threshold voltage decreases, allowing higher performance, though at the cost of more leakage.

3. REDUCING POWER

3.1. Circuit And Logic Level Techniques

3.1.1. Transistor Sizing. Transistor sizing [Penzes et al. 2002; Ebergen et al. 2004] reduces the width of transistors to reduce their dynamic power consumption, using low-level models that relate the power consumption to the width. According to these models, reducing the width also increases the transistor's delay and thus the transistors that lie away from the critical paths of a circuit are usually the best candidates for this technique. Algorithms for applying this technique usually associate with each transistor a tolerable delay which varies depending on how close that transistor is to the critical path. These algorithms then try to scale each transistor to be as small as possible without violating its tolerable delay.

3.1.2. Transistor Reordering. The arrangement of transistors in a circuit affects energy consumption. Figure 7

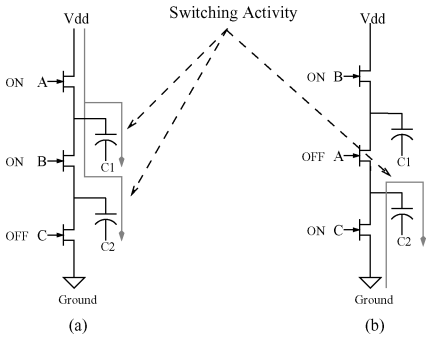

Fig. 7. Transistor Reordering. Figure adapted from Hossain et al. 1996.

shows two possible implementations of the same circuit that differ only in their placement of the transistors marked A and B. Suppose that the input to transistor A is 1, the input to transistor B is 1, and the input to transistor C is 0. Then transistors A and B will be on, allowing current from Vdd to flow through them and charge the capacitors C1 and C2.

Now suppose that the inputs change and that A's input becomes 0, and C's input becomes 1. Then A will be off while B and C will be on. Now the implementations in (a) and (b) will differ in the amounts of switching activity. In (a), current from ground will flow past transistors B and C, discharging both the capacitors C1 and C2. However, in (b), the current from ground will only flow past transistor C; it will not get past transistor A since A is turned off. Thus it will only discharge the capacitor C2, rather than both C1 and C2 as in part (a). Thus the implementation in (b) will consume less power than that in (a).

Transistor reordering [Kursun et al. 2004; Sultania et al. 2004] rearranges transistors to minimize their switching activity. One of its guiding principles is to place transistors closer to the circuit's outputs if they switch frequently in order to prevent a domino effect where the switching activity from one transistor trickles into many other transistors causing widespread power dissipation. This requires profiling techniques to determine how frequently different transistors are likely to switch.

3.1.3. Half Frequency and Half Swing Clocks. Half-frequency and half-swing clocks reduce frequency and voltage, respectively. Traditionally, hardware events such as register file writes occur on a rising clock edge. Half-frequency clocks synchronize events using both edges, and they tick at half the speed of regular clocks, thus cutting clock switching power in half. Reduced-swing clocks also often use a lower voltage signal and thus reduce power quadratically.

3.1.4. Logic Gate Restructuring. There are many ways to build a circuit out of logic gates. One decision that affects power consumption is how to arrange the gates and their input signals.

For example, consider two implementations of a four-input AND gate (Figure 8), a chain implementation (a), and a tree implementation (b). Knowing the signal probabilities (1 or 0) at each of the primary inputs (A, B, C, D), one can easily calculate the transition probabilities ($0 \rightarrow 1$) for each output (W, X, F, Y, Z). If each input has an equal probability of being a 1 or a 0, then the calculation shows that the chain implementation (a) is likely to switch less than the tree implementation (b). This is because each gate in a chain has a lower probability of having a $0 \rightarrow 1$ transition than its predecessor; its transition probability depends on those of all its predecessors. In the tree implementation, on the other hand, some gates may share a parent (in the tree topology) instead of being directly connected together. These gates could have the same transition probabilities.

Nevertheless, chain implementations do not necessarily save more energy than tree implementations. There are other issues to consider when choosing a topology. One is the issue of glitches or spurious transitions that occur when a gate does not receive all of its inputs at the same time. These glitches are more common in chain implementations where signals can travel along different paths having widely varying delays. One solution to reduce glitches is to change the topology so that the different paths in the circuit have similar

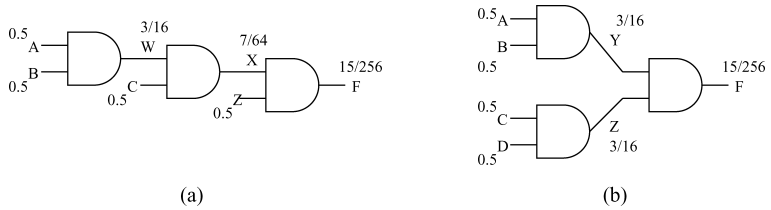

Fig. 8. Gate restructuring. Figure adapted from the Pennsylvania State University Microsystems Design Laboratory's tutorial on low power design.

delays. This solution, known as *path balancing* often transforms chain implementations into tree implementations. Another solution, called *retiming*, involves inserting flip-flops or registers to slow down and thereby synchronize the signals that pass along different paths but reconverge to the same gate. Because flip-flops and registers are in sync with the processor clock, they sample their inputs less frequently than logic gates and are thus more immune to glitches.

3.1.5. Technology Mapping. Because of the huge number of possibilities and tradeoffs at the gate level, designers rely on tools to determine the most energy-optimal way of arranging gates and signals. Technology mapping [Chen et al. 2004; Li et al. 2004; Rutenbar et al. 2001] is the automated process of constructing a gate-level representation of a circuit subject to constraints such as area, delay, and power. Technology mapping for power relies on gate-level power models and a library that describes the available gates, and their design constraints. Before a circuit can be described in terms of gates, it is initially represented at the logic level. The problem is to design the circuit out of logic gates in a way that will mimimize the total power consumption under delay and cost constraints. This is an NP-hard Directed Acyclic Graph (DAG) covering problem, and a common heuristic to solve it is to break the DAG representation of a circuit into a set of trees and find the optimal mapping for each subtree using standard tree-covering algorithms.

3.1.6. Low Power Flip-Flops. Flip-flops are the building blocks of small memories such as register files. A typical master-slave flip-flop consists of two latches, a master latch, and a slave latch. The inputs of the master latch are the clock signal and data. The inputs of the slave latch are the inverse of the clock signal and the data output by the master latch. Thus when the clock signal is high, the master latch is turned on, and the slave latch is turned off. In this phase, the master samples whatever inputs it receives and outputs them. The slave, however, does not sample its inputs but merely outputs whatever it has most recently stored. On the falling edge of the clock, the master turns off and the slave turns on. Thus the master saves its most recent input and stops sampling any further inputs. The slave samples the new inputs it receives from the master and outputs it.

Besides this master-slave design are other common designs such as the *pulse-triggered flip-flop* and *sense-amplifier flip-flop*. All these designs share some common sources of power consumption, namely power dissipated from the clock signal, power dissipated in internal switching activity (caused by the clock signal and by changes in data), and power dissipated when the outputs change.

Researchers have proposed several alternative low power designs for flip-flops. Most of these approaches reduce the switching activity or the power dissipated by the clock signal. One alternative is the *self-gating flip-flop*. This design inhibits the clock signal to the flip-flop when the inputs will produce no change in the outputs. Strollo et al. [2000] have proposed two versions of this design. In the *double-gated flip-flop*, the master and slave latches each have their own clock-gating circuit.

The circuit consists of a comparator that checks whether the current and previous inputs are different and some logic that inhibits the clock signal based on the output of the comparator. In the *sequential-gated flip-flop*, the master and the slave share the same clock-gating circuit instead of having their own individual circuit as in the double-gated flip-flop.

Another low-power flip-flop is the *conditional capture* FLIP-FLOP [Kong et al. 2001; Nedovic et al. 2001]. This flip-flop detects when its inputs produce no change in the output and stops these redundant inputs from causing any spurious internal current switches. There are many variations on this idea, such as the *conditional discharge flip-flops* and *conditional precharge flip-flops* [Zhao et al. 2004] which eliminate unnecessary precharging and discharging of internal elements.

A third type of low-power flip-flop is the *dual edge triggered* FLIP-FLOP [Llopis and Sachdev 1996]. These flip-flops are sensitive to both the rising and falling edges of the clock, and can do the same work as a regular flip-flop at half the clock frequency. By cutting the clock frequency in half, these flip-flops save total power consumption but may not save total energy, although they could save energy by reducing the voltage along with the frequency. Another drawback is that these flip-flops require more area than regular flip-flops. This increase in area may cause them to consume more power on the whole.

3.1.7. Low-Power Control Logic Design. One can view the control logic of a processor as a Finite State Machine (FSM). It specifies the possible processor states and conditions for switching between the states and generates the signals that activate the appropriate circuitry for each state. For example, when an instruction is in the execution stage of the pipeline, the control logic may generate signals to activate the correct execution unit.

Although control logic optimizations have traditionally targeted performance, they are now also targeting power. One way of reducing power is to encode the FSM states in a way that minimizes the switching activity throughout the processor. Another common approach involves decomposing the FSM into sub-FSMs and activating only the circuitry needed for the currently executing sub-FSM. Some researchers [Gao and Hayes 2003] have developed tools that automate this process of decomposition, subject to some quality and correctness constraints.

3.1.8. Delay-Based Dynamic-Supply Voltage Adjustment. Commercial processors that can run at multiple clock speed normally use a lookup table to decide what supply voltage to select for a given clock speed. This table is use built ahead of time through a worst-case analysis. Because worst-case analyses may not reflect reality, ARM Inc. has been developing a more efficient runtime solution known as the Razor pipeline [Ernst et al. 2003].

Instead of using a lookup table, Razor adjusts the supply voltage based on the delay in the circuit. The idea is that whenever the voltage is reduced, the circuit slows down, causing timing errors if the clock frequency chosen for that voltage is too high. Because of these errors, some instructions produce inconsistent results or fail altogether. The Razor pipeline periodically monitors how many such errors occur. If the number of errors exceeds a threshold, it increases the supply voltage. If the number of errors is below a threshold, it scales the voltage further to save more energy.

This solution requires extra hardware in order to detect and correct circuit errors. To detect errors, it augments the flip-flops in delay-critical regions of the circuit with *shadow-latches*. These latches receive the same inputs as the flip-flops but are clocked more slowly to adapt to the reduced supply voltage. If the output of the flip-flop differs from that of its shadow-latch, then this signifies an error. In the event of such an error, the circuit propagates the output of the shadow-latch instead of the flip-flop, delaying the pipeline for a cycle if necessary.

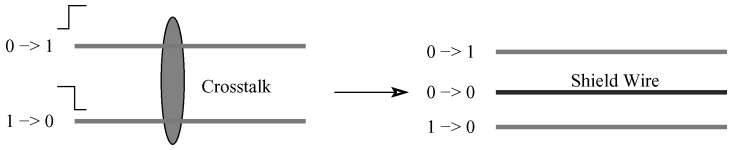

Fig. 9. Crosstalk prevention through shielding.

3.2. Low-Power Interconnect

Interconnect heavily affects power consumption since it is the medium of most electrical activity. Efforts to improve chip performance are resulting in smaller chips with more transistors and more densely packed wires carrying larger currents. The wires in a chip often use materials with poor thermal conductivity [Banerjee and Mehrotra 2001].

3.2.1. Bus Encoding And CrossTalk. A popular way of reducing the power consumed in buses is to reduce the switching activity through intelligent bus encoding schemes, such as *bus-inversion* [Stan and Burleson 1995]. Buses consist of wires that transmit bits (logical 1 or 0). For every data transmission on a bus, the number of wires that switch depends on the current and previous values transmitted. If the Hamming distance between these values is more than half the number of wires, then most of the wires on the bus will switch current. To prevent this from happening, bus-inversion transmits the inverse of the intended value and asserts a control signal alerting recipients of the inversion. For example, if the current binary value to transmit is 110 and the previous was 000, the bus instead transmits 001, the inverse of 110.

Bus-inversion ensures that at most half of the bus wires switch during a bus transaction. However, because of the cost of the logic required to invert the bus lines, this technique is mainly used in external buses rather than the internal chip interconnect.

Moreover, some researchers have argued that this technique makes the simplistic assumption that the amount of power that an interconnect wire consumes depends only on the number of bit transitions on that wire. As chips become smaller, there arise additional sources of power consumption. One of these sources is crosstalk [Sylvester and Keutzer 1998]. *Crosstalk* is spurious activity on a wire that is caused by activity in neighboring wires. As well as increasing delays and impairing circuit integrity, crosstalk can increase power consumption.

One way of reducing crosstalk [Taylor et al. 2001] is to insert a shield wire (Figure 9) between adjacent bus wires. Since the shield remains deasserted, no adjacent wires switch in opposite directions. This solution doubles the number of wires. However, the principle behind it has sparked efforts to develop *self-shielding codes* [Victor and Keutzer 2001; Patel and Markov 2003] resistant to crosstalk. As in traditional bus encoding, a value is encoded and then transmitted. However, the code chosen avoids opposing transitions on adjacent bus wires.

3.2.2. Low Swing Buses. Bus-encoding schemes attempt to reduce transitions on the bus. Alternatively, a bus can transmit the same information but at a lower voltage. This is the principle behind *low swing buses* [Zhang and Rabaey 1998]. Traditionally, logical one is represented by 5 volts and logical zero is represented by -5 volts. In a low-swing system (Figure 10), logical one and zero are encoded using lower voltages, such as $+300$mV and -300mV. Typically, these systems are implemented with differential signaling. An input signal is split into two signals of opposite polarity bounded by a smaller voltage range. The receiver sees the difference between the two transmitted signals as the actual signal and amplifies it back to normal voltage. Low Swing Differential Signaling has several advantages in addition to reduced power consumption. It is immune to

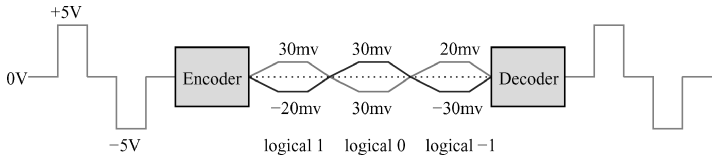

Fig. 10. Low voltage differential signaling

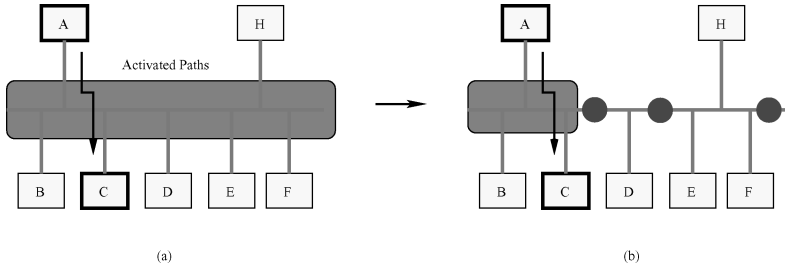

Fig. 11. Bus segmentation.

crosstalk and electromagnetic radiation effects. Since the two transmitted signals are close together, any spurious activity will affect both equally without affecting the difference between them. However, when implementing this technique, one needs to consider the costs of increased hardware at the encoder and decoder.

3.2.3. Bus Segmentation. Another effective technique is *bus segmentation*. In a traditional shared bus architecture, the entire bus is charged and discharged upon every access. Segmentation (Figure 11) splits a bus into multiple segments connected by links that regulate the traffic between adjacent segments. Links connecting paths essential to a communication are activated independently, allowing most of the bus to remain powered down. Ideally, devices communicating frequently should be in the same or nearby segments to avoid powering many links. Jone et al. [2003] have examined how to partition a bus to ensure this property. Their algorithm begins with an undirected graph whose nodes are devices, edges connect communicating devices, and edge weights display communication frequency. Applying the Gomory and Hu algorithm [1961], they find a spanning tree in which the product of communication frequency and number of edges between any two devices is minimal.

3.2.4. Adiabatic Buses. The preceding techniques work by reducing activity or voltage. In contrast, *adiabatic circuits* [Lyuboslavsky et al. 2000] reduce total capacitance. These circuits reuse existing electrical charge to avoid creating new charge. In a traditional bus, when a wire becomes deasserted, its previous charge is wasted. A *charge-recovery bus* recycles the charge for wires about to be asserted.

Figure 12 illustrates a simple design for a two-bit adiabatic bus [Bishop and Irwin 1999]. A comparator in each bitline tests whether the current bit is equal to the previously sent bit. Inequality signifies a transition and connects the bitline to a shorting wire used for sharing charge across bitlines. In the sample scenario, Bitline1 has a falling transition while Bitline0 has a rising transition. As Bitline1 falls, its positive charge transfers to Bitline0, allowing Bitline0 to rise. The power saved depends on transition patterns. No energy is saved when all lines rise. The most energy is saved when an equal number of lines rise and fall simultaneously.

The biggest drawback of adiabatic circuits is a delay for transfering shared charge.

3.2.5. Network-On-Chip. All of the above techniques assume an architecture in

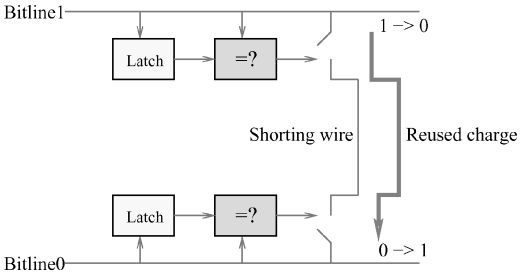

Fig. 12. Two bit charge recovery bus.

which multiple functional units share one or more buses. This shared-bus paradigm has several drawbacks with respect to power and performance. The inherent bus bandwidth limits the speed and volume of data transfers and does not suit the varying requirements of different execution units. Only one unit can access the bus at a time, though several may be requesting bus access simultaneously. Bus arbitration mechanisms are energy intensive. Unlike simple data transfers, every bus transaction involves multiple clock cycles of handshaking protocols that increase power and delay.

As a consequence, researchers have been investigating the use of *generic interconnection networks* to replace buses. Compared to buses, networks offer higher bandwidth and support concurrent connections. This design makes it easier to troubleshoot energy-intensive areas of the network. Many networks have sophisticated mechanisms for adapting to varying traffic patterns and quality-of-service requirements. Several authors [Zhang et al. 2001; Dally and Towles 2001; Sgroi et al. 2001] propose the application of these techniques at the chip level. These claims have yet to be verified, and interconnect power reduction remains a promising area for future research.

Wang et al. [2003] have developed hardware-level optimizations for different sources of energy consumption in an on-chip network. One of the alternative network topologies they propose is the segmented crossbar topology which aims to reduce the energy consumption of data transfers. When data is transfered over a regular network grid, all rows and columns corresponding to the intersection points end up switching current even though only parts of these rows and columns are actually traversed. To eliminate this widespread current switching, the segmented crossbar topology divides the rows and columns into segments demarcated by tristate buffers. The different segments are selectively activated as the data traverses through them, hence confining current switches to only those segments along which the data actually passes.

3.3. Low-Power Memories and Memory Hierarchies

3.3.1. Types of Memory. One can classify memory structures into two categories, Random Access Memories (RAM) and Read-Only-Memories (ROM). There are two kinds of RAMs, static RAMs (SRAM) and dynamic RAMs (DRAM) which differ in how they store data. SRAMs store data using flip-flops and DRAMs store each bit of data as a charge on a capacitor; thus DRAMs need to refresh their data periodically. SRAMs allow faster accesses than DRAMs but require more area and are more expensive. As a result, normally only register files, caches, and high bandwidth parts of the system are made up of SRAM cells, while main memory is made up of DRAM cells. Although these cells have slower access times than SRAMs, they contain fewer transistors and are less expensive than SRAM cells.

The techniques we will introduce are not confined to any specific type of RAM or ROM. Rather they are high-level architectural principles that apply across the spectrum of memories to the extent that the required technology is available. They attempt to reduce the energy dissipation of memory accesses in two ways, either by reducing the energy dissipated in a memory accesses, or by reducing the number of memory accesses.

3.3.2. Splitting Memories Into Smaller Subsystems. An effective way to reduce the energy that is dissipated in a memory

access is to activate only the needed memory circuits in each access. One way to expose these circuits is to partition memories into smaller, independently accessible components. This can be done as different granularities.

At the lowest granularity, one can design a main memory system that is split up into multiple banks each of which can independently transition into different power modes. Compilers and operating systems can cluster data into a minimal set of banks, allowing the other unused banks to power down and thereby save energy [Luz et al. 2002; Lebeck et al. 2000].

At a higher granularity, one can partition the separate banks of a partitioned memory into subbanks and activate only the relevant subbank in every memory access. This is a technique that has been applied to caches [Ghose and Kamble 1999]. In a normal cache access, the set-selection step involves transferring all blocks in all banks onto tag-comparison latches. Since the requested word is at a known offset in these blocks, energy can be saved by transfering only words at that offset. Cache subbanking splits the block array of each cache line into multiple banks. During set-selection, only the relevant bank from each line is active. Since a single subbank is active at a time, all subbanks of a line share the same output latches and sense amplifiers to reduce hardware complexity. Subbanking saves energy without increasing memory access time. Moreover, it is independent of program locality patterns.

3.3.3. Augmenting the Memory Hierarchy With Specialized Cache Structures. The second way to reduce energy dissipation in the memory hierarchy is to reduce the number of memory hierarchy accesses. One group of researchers [Kin et al. 1997] developed a simple but effective technique to do this. Their idea was to integrate a specialized cache into the typical memory hierarchy of a modern processor, a hierarchy that may already contain one or more levels of caches. The new cache they were proposing would sit between the processor and first level cache. It would be smaller than the first level cache and hence dissipate less energy. Yet it would be large enough to store an application's working set and filter out many memory references; only when a data request misses this cache would it require searching higher levels of cache, and the penalty for a miss would be offset by redirecting more memory references to this smaller, more energy efficient cache. This was the principle behind the *filter cache*. Though deceptively simple, this idea lies at the heart of many of the low-power cache designs used today, ranging from simple extensions of the cache hierarchy (e.g., block buffering) to the scratch pad memories used in embedded systems, and the complex trace caches used in high-end generation processors.

One simple extension is the technique of *block buffering*. This technique augments caches with small buffers to store the most recently accessed cache set. If a data item that has been requested belongs to a cache set that is already in the buffer, then one does not have to search for it in the rest of the cache.

In embedded systems, *scratch pad memories* [Panda et al. 2000; Banakar et al. 2002; Kandemir et al. 2002] have been the extension of choice. These are software controlled memories that reside on-chip. Unlike caches, the data they should contain is determined ahead of time and remains fixed while programs execute. These memories are less volatile than conventional caches and can be accessed within a single cycle. They are ideal for specialized embedded systems whose applications are often hand tuned.

General purpose processors of the latest generation sometimes rely on more complex caching techniques such as the *trace cache*. Trace caches were originally developed for performance but have been studied recently for their power benefits. Instead of storing instructions in their compiled order, a trace cache stores traces of instructions in their executed order. If an instruction sequence is already in the trace cache, then it need not be fetched from the instruction cache

but can be decoded directly from the trace cache. This saves power by reducing the number of instruction cache accesses.

In the original trace cache design, both the trace cache and the instruction cache are accessed in parallel on every clock cycle. Thus instructions are fetched from the instruction cache while the trace cache is searched for the next trace that is predicted to executed. If the trace is in the trace cache, then the instructions fetched from the instruction cache are discarded. Otherwise the program execution proceeds out of the instruction cache and simultaneously new traces are created from the instructions that are fetched.

Low-power designs for trace caches typically aim to reduce the number of instruction cache accesses. One alternative, the *dynamic direction prediction-based trace cache* [Hu et al. 2003], uses branch predition to decide where to fetch instructions from. If the branch predictor predicts the next trace with high confidence and that trace is in the trace cache, then the instructions are fetched from the trace cache rather than the instruction cache.

The *selective trace cache* [Hu et al. 2002] extends this technique with compiler support. The idea is to identify frequently executed, or hot traces, and bring these traces into the trace cache. The compiler inserts hints after every branch instruction, indicating whether it belongs to a hot trace. At runtime, these hints cause instructions to be fetched from either the trace cache or the instruction cache.

3.4. Low-Power Processor Architecture Adaptations

So far we have described the energy saving features of hardware as though they were a fixed foundation upon which programs execute. However, programs exhibit wide variations in behavior. Researchers have been developing hardware structures whose parameters can be adjusted on demand so that one can save energy by activating just the minimum hardware resources needed for the code that is executing.

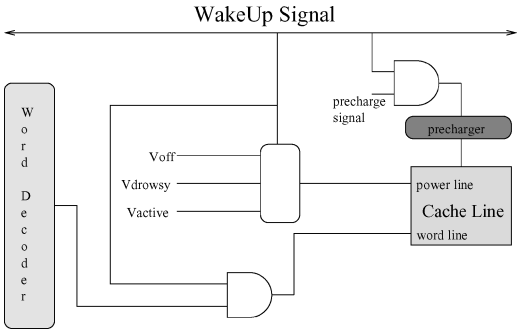

Fig. 13. Drowsy cache line.

3.4.1. Adaptive Caches. There is a wealth of literature on adaptive caches, caches whose storage elements (lines, blocks, or sets) can be selectively activated based on the application workload. One example of such a cache is the Deep-Submicron Instruction (DRI) cache [Powell et al. 2001]. This cache permits one to deactivate its individual sets on demand by gating their supply voltages. To decide what sets to activate at any given time, the cache uses a hardware profiler that monitors the application's cache-miss patterns. Whenever the cache misses exceed a threshold, the DRI cache activates previously deactivated sets. Likewise, whenever the miss rate falls below a threshold, the DRI deactivates some of these sets by inhibiting their supply voltages.

A problem with this approach is that dormant memory cells lose data and need more time to be reactivated for their next use. Thus an alternative to inhibiting their supply voltages is to reduce their voltages as low as possible without losing data. This is the aim of the *drowsy cache*, a cache whose lines can be placed in a drowsy mode where they dissipate minimal power but retain data and can be reactivated faster.

Figure 13 illustrates a typical drowsy cache line [Kim et al. 2002]. The wakeup signal chooses between voltages for the active, drowsy, and off states. It also regulates the word decoder's output to prevent data in a drowsy line from being accessed. Before a cache access, the line circuitry must be precharged. In a

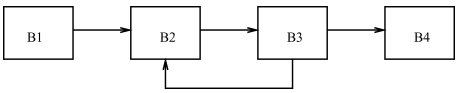

Fig. 14. Dead block elimination.

traditional cache, the precharger is continuously on. To save power, the drowsy circuitry regulates the wakeup signal with a precharge signal. The precharger becomes active only when the line is woken up.

Many heuristics have been developed for controlling these drowsy circuits. The simplest policy for controlling the drowsy cache is *cache decay* [Kaxiras et al. 2001] which simply turns off unused cache lines after a fixed interval. More sophisticated policies [Zhang et al. 2002; 2003; Hu et al. 2003] consider information about runtime program behavior. For example, Hu et al. [2003] have proposed hardware that detects when an execution moves into a hotspot and activates only those cache lines that lie within that hotspot. The main idea of their approach is to count how often different branches are taken. When a branch is taken sufficiently many times, its target address is a hotspot, and the hardware sets special bits to indicate that cache lines within that hotspot should not be shut down. Periodically, a runtime routine disables cache lines that do not have these bits set and decays the profiling data. Whenever it reactivates a cache line before its next use, it also reactivates the line that corresponds to the next subsequent memory access.

There are many other heuristics for controlling cache lines. Dead-block elimination [Kabadi et al. 2002] powers down cache lines containing basic blocks that have reached their final use. It is a compiler-directed technique that requires a static control flow analysis to identify these blocks. Consider the control flow graph in Figure 14. Since block B1 executes only once, its cache lines can power down once control reaches B2. Similarly, the lines containing B2 and B3 can power down once control reaches B4.

3.4.2. Adaptive Instruction Queues. Buyuktosunoglu et al. [2001] developed one of the first adaptive instruction issue queues, a 32-bit queue consisting of four equal size partitions that can be independently activated. Each partition consists of wakeup logic that decides when instructions are ready to execute and readout logic that dispatches ready instructions into the pipeline. At any time, only the partitions that contain the currently executing instructions are activated.

There have been many heuristics developed for configuring these queues. Some require measuring the rate at which instructions are executed per processor clock cycles, or IPC. For example, Buyuktosunoglu et al.'s heuristic [2001] periodically measures the IPC over fixed length intervals. If the IPC of the current interval is a factor smaller than that of the previous interval (indicating worse performance), then the heuristic increases the instruction queue size to increase the throughput. Otherwise it may decrease the queue size.

Bahar and Manne [2001] propose a similar heuristic targeting a simulated 8-wide issue processor that can be reconfigured as a 6-wide issue or 4-wide issue. They also measure IPC over fixed intervals but measure the floating point IPC separately from the integer IPC since the former is more critical for floating point applications. Their heuristic, like Buyuktosunoglu et al.'s [2001], compares the measured IPC with specific thresholds to determine when to downsize the issue queue.

Folegnani and Gonzalez [2001], on the other hand, use a different heuristic. They find that when a program has little parallelism, the youngest part of the issue queue (which contains the most recent instructions) contributes very little to the overall IPC in that instructions in this part are often committed late. Thus their heuristic monitors the contribution of the youngest part of this queue and deactivates this part when its contribution is minimal.

3.4.3. Algorithms for Reconfiguring Multiple Structures. In addition to this work,

researchers have also been tackling the problem of problem of reconfiguring multiple hardware units simultaneously.

One group [Hughes et al. 2001; Sasanka et al. 2002; Hughes and Adve 2004] has been developing heuristics for combining hardware adaptations with dynamic voltage scaling during multimedia applications. Their strategy is to run two algorithms while individual video frames are being processed. A global algorithm chooses the initial DVS setting and baseline hardware configuration for each frame, while a local algorithm tunes various hardware parameters (e.g., instruction window sizes) while the frame is executing to use up any remaining slack periods.

Another group [Iyer and Marculescu 2001, 2002a] has developed heuristics that adjust the pipeline width and register update unit (RUU) size for different hotspots or frequently executed program regions. To detect these hotspots, their technique counts the number of times each branch instruction is taken. When a branch is taken enough times, it is marked as a candidate branch.

To detect how often candidate branches are taken, the heuristic uses a *hotspot detection counter*. Initially the hotspot counter contains a positive integer. Each time a candidate branch is taken, the counter is decremented, and each time a noncandidate branch is taken, the counter is incremented. If the counter becomes zero, this means that the program execution is inside a hotspot.

Once the program execution is in a hotspot, this heuristic tests the energy consumption of different processor configurations at fixed length intervals. It finally chooses the most energy saving configuration as the one to use the next time the same hotspot is entered. To measure energy at runtime, the heuristic relies on hardware that monitors usage statistics of the most power hungry units and calculates the total power based on energy per access values.

Huang et al.[2000; 2003] apply hardware adaptations at the granularity of subroutines. To decide what adaptations to apply, they use offline profiling to plot an energy-delay tradeoff curve. Each point in the curve represents the energy and delay tradeoff of applying an adaptation to a subroutine. There are three regions in the curve. The first includes adaptations that save energy without impairing performance; these are the adaptations that will always be applied. The third represents adaptations that worsen both performance and energy; these are the adaptations that will never be applied. Between these two extremes are adaptations that trade performance for energy savings; some of these may be applied depending on the tolerable performance loss. The main idea is to trace the curve from the origin and apply every adaptation that one encounters until the cumulative performance loss from applying all the encountered adaptations reaches the maximum tolerable performance loss.

Ponomarev et al. [2001] develop heuristics for controlling a configurable issue queue, a configurable reorder buffer and a configurable load/store queue. Unlike the IPC based heuristics we have seen, this heuristic is occupancy-based because the occupancy of hardware structures indicates how congested these structures are, and congestion is a leading cause of performance loss. Whenever a hardware structure (e.g., instruction queue) has low occupancy, the heuristic reduces its size. In contrast, if the structure is completely full for a long time, the heuristic increases its size to avoid congestion.

Another group of researchers [Albonesi et al. 2003; Dropsho et al. 2002] explore the complex problem of configuring a processor whose adaptable components include two levels of caches, integer and floating point issue queues, load/store queues, register files, as well as a reorder buffer. To dodge the sheer explosion of possible configurations, they tune each component based only on its local usage statistics. They propose two heuristics for doing this, one for tuning caches and another for tuning buffers and register files.

The caches they consider are selective way caches; each of their multiple ways can independently be activated or

disabled. Each way has a most recently used (MRU) counter that is incremented every time that a cache searche hits the way. The cache tuning heuristic samples this counter at fixed intervals to determine the number of hits in each way and the number of overall misses. Using these access statistics, the heuristic computes the energy and performance overheads for all possible cache configurations and chooses the best configuration dynamically.

The heuristic for controlling other structures such as issue queues is occupancy based. It measures how often different components of these structures fill up with data and uses this information to decide whether to upsize or downsize the structure.

3.5. Dynamic Voltage Scaling

Dynamic voltage scaling (DVS) addresses the problem of how to modulate a processor's clock frequency and supply voltage in lockstep as programs execute. The premise is that a processor's workloads vary and that when the processor has less work, it can be slowed down without affecting performance adversely. For example, if the system has only one task and it has a workload that requires 10 cycles to execute and a deadline of 100 seconds, the processor can slow down to 1/10 cycles/sec, saving power and meeting the deadline right on time. This is presumably more efficient than running the task at full speed and idling for the remainder of the period. Though it appears straightforward, this naive picture of how DVS works is highly simplified and hides serious real-world complexities.

3.5.1. Unpredictable Nature Of Workloads. The first problem is how to predict workloads with reasonable accuracy. This requires knowing what tasks will execute at any given time and the work required for these tasks. Two issues complicate this problem. First, tasks can be preempted at arbitrary times due to user and I/O device requests. Second, it is not always possible to accurately predict the future runtime of an arbitrary algorithm (c.f., the Halting Problem [Turing 1937]). There have been ongoing efforts [Nilsen and Rygg 1995; Lim et al. 1995; Diniz 2003; Ishihara and Yasuura 1998] in the real-time systems community to accurately estimate the worst case execution times of programs. These attempts use either measurement-based prediction, static analysis, or a mixture of both. Measurement involves a *learning lag* whereby the execution times of various tasks are sampled and future execution times are extrapolated. The problem is that, when workloads are irregular, future execution times may not resemble past execution times. Microarchitectural innovations such as pipelining, hyperthreading and out-of-order execution make it difficult to predict execution times in real systems statically. Among other things, a compiler needs to know how instructions are interleaved in the pipeline, what the probabilities are that different branches are taken, and when cache misses and pipeline hazards are likely to occur. This requires modeling the pipeline and memory hierarchy. A number of researchers [Ferdinand 1997; Clements 1996] have attempted to integrate complete pipeline and cache models into compilers. It remains nontrivial to develop models that can be used efficiently in a compiler and that capture all the complexities inherent in current systems.

3.5.2. Indeterminism And Anomalies In Real Systems. Even if the DVS algorithm predicts correctly what the processor's workloads will be, determining how fast to run the processor is nontrivial. The nondeterminism in real systems removes any strict relationships between clock frequency, execution time, and power consumption. Thus most theoretical studies on voltage scaling are based on assumptions that may sound reasonable but are not guaranteed to hold in real systems.

One misconception is that total microprocessor system power is quadratic in supply voltage. Under the CMOS transistor model, the power dissipation of individual transistors are quadratic in their supply voltages, but there remains no

precise way of estimating the power dissipation of an entire system. One can construct scenarios where total system power is not quadratic in supply voltage. Martin and Siewiorek [2001] found that the StrongARM SA-1100 has two supply voltages, a 1.5V supply that powers the CPU and a 3.3V supply that powers the I/O pins. The total power dissipation is dominated by the larger supply voltage; hence, even if the smaller supply voltage were allowed to vary, it would not reduce power quadratically. Fan et al. [2003] found that, in some architectures where active memory banks dominate total system power consumption, reducing the supply voltage dramatically reduces CPU power dissipation but has a miniscule effect on total power. Thus, for full benefits, dynamic voltage scaling may need to be combined with resource hibernation to power down other energy hungry parts of the chip.

Another misconception is that it is most power efficient to run a task at the slowest constant speed that allows it to exactly meet its deadline. Several theoretical studies [Ishihara and Yasuura 1998] attempt to prove this claim. These proofs rest on idealistic assumptions such as "power is a convex linearly increasing function of frequency", assumptions that ignore how DVS affects the system as a whole. When the processor slows down, peripheral devices may remain activated longer, consuming more power. An important study of this issue was done by Fan et al. [2003] who show that, for specific DRAM architectures, the energy versus slowdown curve is "U" shaped. As the processor slows down, CPU energy decreases but the cumulative energy consumed in active memory banks increases. Thus the optimal speed is actually higher than the processor's lowest speed; any speed lower than this causes memory energy dissipation to overshadow the effects of DVS.

A third misconception is that execution time is inversely proportional to clock frequency. It is actually an open problem how slowing down the processor affects the execution time of an arbitrary application. DVS may result in nonlinearities [Buttazzo 2002]. Consider a system with

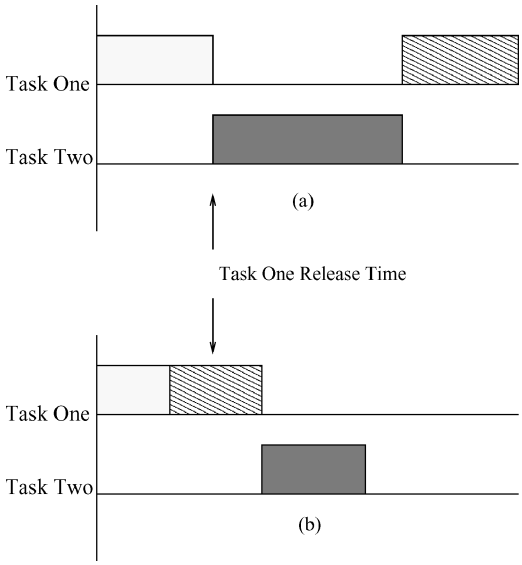

Fig. 15. Scheduling effects. When DVS is applied (a), task one takes a longer time to enter its critical section, and task two is able to preempt it right before it does. When DVS is not applied (b), task one enters its critical section sooner and thus prevents task two from preempting it until it finishes its critical section. This figure has been adopted from Buttazzo [2002].

two tasks (Figure 15). When the processor slows down (a), Task One takes longer to enter its critical section and is thus preempted by Task Two as soon as Task Two is released into the system. When the processor instead runs at maximum speed (b), Task One enters its critical section earlier and blocks Task Two until it finishes this section. This hypothetical example suggests that DVS can affect the order in which tasks are executed. But the order in which tasks execute affects the state of the cache which, in turn, affects performance. The exact relationship between the cache state and execution time is not clear. Thus, it is too simplistic to assume that execution time is inversely proportional to clock frequency.

All of these issues show that theoretical studies are insufficient for understanding how DVS will affect system state. One needs to develop an experimental approach driven by heuristics and evaluate the tradeoffs of these heuristics empirically.

Most of the existing DVS approaches can be classified as interval-based approaches, intertask approaches, or intratask approaches.

3.5.3. Interval-Based Approaches. Interval-based DVS algorithms measure how busy the processor is over some interval or intervals, estimate how busy it will be in the next interval, and adjust the processor speed accordingly. These algorithms differ in how they estimate future processor utilization. Weiser et al. [1994] developed one of the earliest interval-based algorithms. This algorithm, *Past*, periodically measures how long the processor idles. If it idles longer than a threshold, the algorithm slows it down; if it remains busy longer than a threshold, it speeds it up. Since this approach makes decisions based only on the most recent interval, it is error prone. It is also prone to thrashing since it computes a CPU speed for every interval. Govil et al. [1995] examine a number of algorithms that extend Past by considering measurements collected over larger windows of time. The Aged Averages algorithm, for example, estimates the CPU usage for the upcoming interval as a weighted average of usages measured over all previous intervals. These algorithms save more energy than Past since they base their decisions on more information. However, they all assume that workloads are regular. As we have mentioned, it is very difficult, if not impossible, to predict irregular workloads using history information alone.

3.5.4. Intertask Approaches. Intertask DVS algorithms assign different speeds for different tasks. These speeds remain fixed for the duration of each task's execution. Weissel and Bellosa [2002] have developed an intertask algorithm that uses hardware events as the basis for choosing the clock frequencies for different processes. The motivation is that, as processes execute, they generate various hardware events (e.g., cache misses, IPC) at varying rates that relate to their performance and energy dissipation. Weissel and Bellosa's technique involves monitoring these event rates using hardware performance counters and attributing them to the processes that are executing. At each context switch, the heuristic adjusts the clock frequency for the process being activated based on its previously measured event rates. To do this, the heuristic refers to a previously constructed table of event rate combinations, divided into frequency domains. Weissel and Bellosa construct this table through a detailed offline simulation where, for different combinations of event rates, they determine the lowest clock frequency that does not violate a user-specified threshold for performance loss.

Intertask DVS has two drawbacks. First, task workloads are usually unknown until tasks finish running. Thus traditional algorithms either assume perfect knowledge of these workloads or estimate future workloads based on prior workloads. When workloads are irregular, estimating them is nontrivial. Flautner et al. [2001] address this problem by classifying all workloads as interactive episodes and producer-consumer episodes and using separate DVS algorithms for each. For interactive episodes, the clock frequency used depends not only on the predicted workload but also on a perception threshold that estimates how fast the episode would have to run to be acceptable to a user. To recover from prediction error, the algorithm also includes a *panic threshold*. If a session lasts longer than this threshold, the processor immediately switches to full speed.

The second drawback of intertask approaches is that they are unaware of program structure. A program's structure may provide insight into the work required for it, insight that, in turn, may provide opportunities for techniques such as voltage scaling. Within specific program regions, for example, the processor may spend significant time waiting for memory or disk accesses to complete. During the execution of these regions, the processor can be slowed down to meet pipeline stall delays.

3.5.5. Intratask Approaches. Intratask approaches [Lee and Sakurai 2000; Lorch and Smith 2001; Gruian 2001; Yuan and Nahrstedt 2003] adjust the processor speed and voltage within tasks. Lee and Sakurai [2000] developed one of the earliest approaches. Their approach, *run-time voltage hopping*, splits each task into fixed length timeslots. The algorithm assigns each timeslot the lowest speed that allows it to complete within its preferred execution time which is measured as the worst case execution time minus the elapsed execution time up to the current timeslot. An issue not considered in this work is how to divide a task into timeslots. Moreover, the algorithm is pessimistic since tasks could finish before their worst case execution time. It is also insensitive to program structure.

There are many variations on this idea. Two operating system-level intratask algorithms are PACE [Lorch and Smith 2001] and Stochastic DVS [Gruian 2001]. These algorithms choose a speed for every cycle of a task's execution based on the probability distribution of the task workload measured over previous cycles. The difference between PACE and stochastic DVS lies in their cost functions. Stochastic DVS assumes that energy is proportional to the square of the supply voltage, while PACE assumes it is proportional to the square of the clock frequency. These simplifying assumptions are not guaranteed to hold in real systems.

Dudani et al. [2002] have also developed intratask policies at the operating system level. Their work aims to combine intratask voltage scaling with the earliest-deadline-first scheduling algorithm. Dudani et al. split into two subprograms each program that the scheduler chooses to execute. The first subprogram runs at the maximum clock frequency, while the second runs slow enough to keep the combined execution time of both subprograms below the average execution time for the whole program.

In addition to these schemes, there have been a number of intratask policies implemented at the compiler level. Shin and Kim [2001] developed one of the first of these algorithms. They noticed that programs have multiple execution paths, some more time consuming than others, and that whenever control flows away from a critical path, there are opportunities for the processor to slow down and still finish the programs within their deadlines. Based on this observation, Shin and Kim developed a tool that profiles a program offline to determine worst case and average cycles for different execution paths, and then inserting instructions to change the processor frequency at the start of different paths based on this information.

Another approach, *program checkpointing* [Azevedo et al. 2002], annotates a program with checkpoints and assigns timing constraints for executing code between checkpoints. It then profiles the program offline to determine the average number of cycles between different checkpoints. Based on this profiling information and timing constraints, it adjusts the CPU speed at each checkpoint.

Perhaps the most well known approach is by Hsu and Kremer [2003]. They profile a program offline with respect to all possible combinations of clock frequencies assigned to different regions. They then build a table describing how each combination affects execution time and power consumption. Using this table, they select the combination of regions and frequencies that saves the most power without increasing runtime beyond a threshold.

3.5.6. The Implications of Memory Bounded Code. Memory bounded applications have become an attractive target for DVS algorithms because the time for a memory access, on most commercial systems, is independent of how fast the processor is running. When frequent memory accesses dominate the program execution time, they limit how fast the program can finish executing. Because of this "memory wall", the processor can actually run slower and save a lot of energy without losing as much performance as it would if it were slowing down a compute-intensive code.

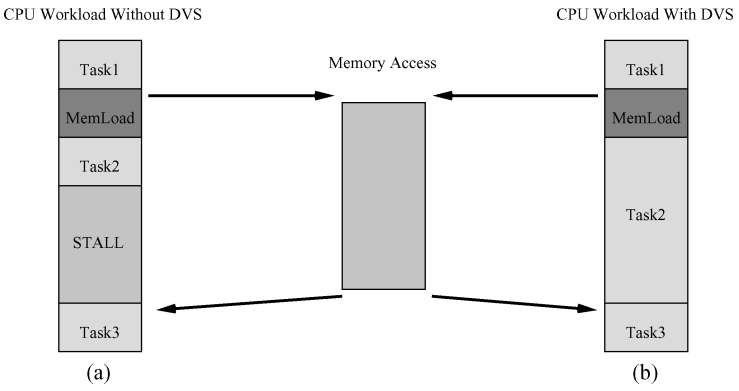

Fig. 16. Making the best of the memory wall.

Figure 16 illustrates this idea. For ease of exposition, we call a sequence of executed instructions a task. Part (a) depicts a processor executing three tasks. Task 1 is followed by a memory load instruction that causes a main memory access. While this access is taking place, task 2 is able to execute since it does not depend on the results of task 1. However task 3, which comes after task 2, depends on the result of task 1, and thus can execute only after the memory access completes. So immediately after executing task 2, the processor sits idle until the memory access finishes. What the processor could have done is illustrated in part (b). Here instead of running task 2 as fast as possible and idling until the memory access completes, the processor slows down the execution of task 2 to overlap exactly with the main memory access. This, of course, saves energy without impairing performance.

But there are many idealized assumptions at work here such as the assumption that a DVS algorithm can predict with complete accuracy a program's future behavior and switch the clock frequency without any hidden costs. In reality, things are far more complex. A fundamental problem is how to detect program regions during which the processor stalls, waiting for a memory, disk, or network access to complete. Modern processors contain counters that measure events such as cache misses, but it is difficult to extrapolate the nature of these stalls from observing these events.

Nevertheless, a growing number of researchers are addressing the DVS problem from this angle; that is, they are developing techniques for modulating the processor frequency based on memory access patterns. Because it is difficult to reason abstractly about the complex events occurring in modern processors, the research in this area has a strong experimental flavor; many of the techniques used are heuristics that are validated through detailed simulations and experiments on real systems.

One recent work is by Kondo and Nakamura [2004]. It is an interval-based approach driven by cache misses. The heuristic periodically calculates the number of outstanding cache misses and increments one of three counters depending on whether the number of outstanding misses is zero, one, or greater than one. The cost function that Kondo and Nakamura use expresses the memory boundedness of the code as a weighted sum of the three counters. In particular, the third counter which is incremented each time there are multiple outstanding misses, receives the heaviest weight. At fixed length intervals, the heuristic compares the memory boundedness, so computed, with respect to an upper and lower threshold. If the memory boundedness is greater than the upper threshold, then the heuristic decreases the frequency and voltage by one setting; otherwise if the memory boundedness is below the lower threshold, then it increases the frequency and voltage settings by one unit.

Stanley-Marbell et al. [2002] propose a similar approach, the Power Adaptation Unit. This is a finite state machine that detects the program counter values for which memory stalls often occur. When a stall is initially detected, the unit would enter a transient state where it counts the number of stall cycles, and after the stall cycles exceeded a threshold, it would mark the initial program counter value as hot. Thus the next time the program counter assumes the same value, the PAU would automatically reduce the clock frequency and voltage.

Another recent work is by Choi et al. [2004]. Their work is based on an execution time model where a program's total runtime is split into two parts, time that the program spends on-chip, and time that it spends in main memory accesses. The on-chip time can be calculated from the number of on-chip instructions, the average cycles for each of these instructions, and the processor clock frequency. The off-chip time is similarly a function of the number of off-chip (memory reference) instructions, the average cycles per off-chip instruction, and the memory clock frequency.

Using this model, Choi et al. [2004] derive a formula for the lowest clock frequency that would keep the performance loss within a tolerable threshold. They find that the key to computing this formula is determining the average number of cycles for an on-chip instruction CPU_{avg}^{on}; this essential information determines the ratio of on-chip to off-chip instructions. They also found that the average cycles per instruction (CPI_{avg}) is a linear function of the average stall cycles per instruction (SPI_{avg}). This reduced their problem to calculating SPI_{avg}. To calculate this, they used data-cache miss statistics provided by hardware counters. Reasoning that the cycles spent in stalls increase with respect to the number of cache misses, Choi et al. built a table that associated different cache miss ranges with different values of SPI_{avg}; this would allow their heuristic to calculate in real time the workload decomposition and appropriate clock frequency.

Unlike the previous approaches which attempt to monitor memory boundedness, a recent technique by Hsu and Feng [2004] attempts to monitor *CPU boundedness*. Their algorithm periodically measures the rate at which instructions are executed (expressed in million instructions per second) to determine how compute-intensive the workload is. The authors find that this approach is as accurate as other approaches that measure memory boundedness, but may be easier to implement because of its simpler cost model.

3.5.7. Dynamic Voltage Scaling In Multiple Clock Domain Architectures. A *Globally Asynchronous, Locally Synchronous* (GALS) chip is split into multiple domains, each of which has its own local clock. Each domain is synchronous with respect to its clock, but the different domains are mutually asynchronous in that they may run at different clock frequencies. This design has three advantages. First, the clocks that power different domains are able to distribute their signals over smaller areas, thus reducing clock skew. Second, the effects of changing the clock frequency are felt less outside the given domain. This is an important advantage that GALS has over conventional CPUs. When a conventional CPU scales its clock frequency, all of the hardware structures that receive the clock signal slow down causing widespread performance loss. In GALS, on the other hand, one can choose to slow down some parts of the circuit, while allowing others to operate at their maximum frequencies. This creates more opportunities for saving energy. In compute bound applications, for example, a GALS system can keep the critical paths of the circuit running as fast as possible but slow down other parts of the circuit.

Nevertheless, it is nontrivial to create voltage scaling algorithms for GALS processors. First, these processors require special mechanisms allowing the different domains to communicate and synchronize. Second, it is unclear how to split the processor into domains. Because communication energy can be large, domains must be

chosen so that there is minimal communication between them. One possibility is to cluster similar functionalities into the same domain. In a GALS processor developed by Semeraro et al. [2002], all the integer execution units are in the same domain. If instead the integer issue queue were in a different domain than the integer ALU, these two units would waste more energy communicating since they are often accessed together.

Another factor that makes it hard to design algorithms is that one needs to consider possible interdependencies between domains. For example, if one domain is slowed down, data may take more time to move through it and reach other domains. This may impact the overall congestion in the circuit and affect performance. A DVS algorithm would need to be aware of these global effects.

Numerous researchers have tackled the complex problem of controlling GALS systems [Magklis et al. 2003; Marculescu 2004; Dropsho et al. 2004; Semeraro et al. 2002; Iyer and Marculescu 2002b; Magklis et al. 2003]. We follow with two recent examples, the first being an interval-based heuristic and the second being a compiler-based heuristic.

Semeraro et al. [2002] developed one of the first runtime algorithms for dynamic voltage scaling in a GALS system. They observed that each domain has an instruction issue queue that indicates how fast instructions are flowing through the pipeline, thus giving insight into the workload. They propose to monitor the issue queue utilization over fixed length intervals and to respond swiftly to sudden changes by rapidly increasing or decreasing the domain's clock frequency, but to decrease the clock frequency very gradually when there are little observable changes in the workload.

One of the first compiler-based algorithms for controlling GALS systems is by Magklis et al. [2003]. Their main algorithm is called the *shaker* algorithm due to its resemblance to a salt shaker. It repeatedly traverses the Directed Acyclic Graph (DAG) representation of a program from root to leaves and from leaves to root, searching for operations whose energy dissipation exceeds a threshold. It stretches out the workload of such operations until the energy dissipation is below a given threshold, repeating this process until either no more operations dissipate excessive energy or until all of the operations have used up their slack. The information provided by this algorithm would be used by a compiler to select clock frequencies for different GALS domains in different program regions.

3.6. Resource Hibernation

Though switching activity causes power consumption, computer components consume power even when idle. *Resource hibernation* techniques power down components during idle periods. A variety of heuristics have been developed to manage the power settings of different devices such as disk drives, network interfaces, and displays. We now give examples of these techniques.

3.6.1. Disk Drives. Disk drives can account for a large percentage of a system's total energy dissipation and many techniques have been developed to attack this problem [Gurumurthi et al. 2003; Li et al. 2004]. Most of the power loss stems from the rotating platter during disk activity and idle periods. To save energy, operating systems tend to pause disk rotation after a fixed period of inactivity and restart it during the next disk access. The decision of when to spin down an idle disk involves tradeoffs between power and performance. High idle time thresholds lead to more energy loss but better performance since the disk need not restart to process requests. Low thresholds allow a disk to spin down sooner when idle but increase the probability of an immediate restart to process a spontaneous request. These immediate restarts, called *bumps*, waste energy and cause delays.

One of the techniques used at the operating system level is *Predictive Dynamic Threshold Adjustment*. This technique adjusts at runtime the idleness threshold, or acceptable window of time before an idle

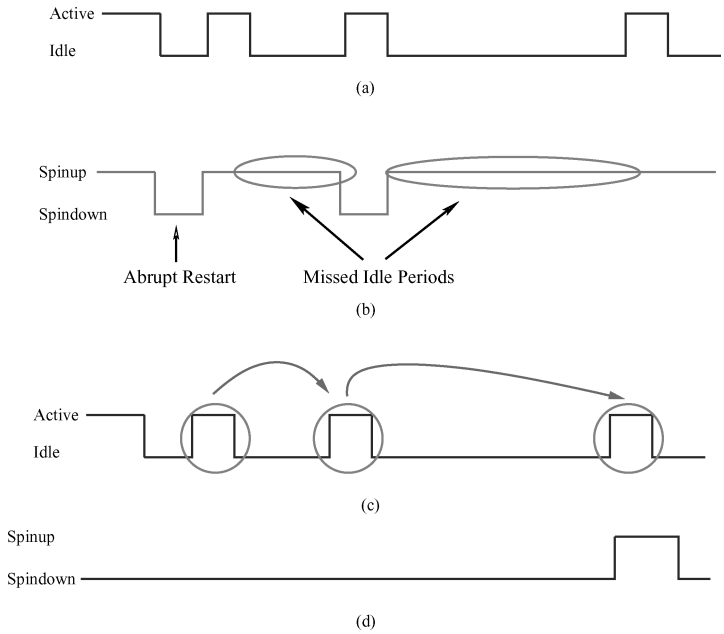

Fig. 17. Predictive disk management.

disk can be spun down. If a disk immediately spins up after spinning down, this technique increases the idleness threshold. Otherwise, it decreases the threshold. This is an example of a technique that uses past disk usage information to predict future usage patterns. These techniques are less effective at dealing with irregular workloads, as Figure 17 illustrates. Part (a) depicts a disk usage pattern in which active phases are abruptly followed by idle phases, and the idle phases keep getting longer. Dynamic threshold adjustment (b) abruptly restarts the disk everytime it spins it down and keeps the disk activated during long idle phases. To improve on this technique, an operating system can cluster disk requests (c), thereby lengthening idle periods (d) when it can spin down the disk. To create opportunities for clustering, the algorithm of Papathanasiou and Scott [2002] delays processing nonurgent disk requests and reduces the number of remaining requests by aggressively prefetching disk data into memory. The delayed requests accumulate in a queue for later processing in a more voluminous transaction. Thus, the disk may remain uninterrupted in a low-power, inactive state for longer periods. However, dependencies between disk requests can sometimes make it impossible to apply clustering.

Moreover, idle periods for spinning down the disk are not always available. For example, large scale servers are continuously processing heavy workloads. Gurumurthi et al. [2003] have recently investigated an alternative strategy for server environments called Dynamic RPM control (DRPM). Instead of spinning down the disk fully, DRPM modulates the rotational speed (in rotations per minute) according to workload demands, thereby eliminating the overheads of transitioning between low power and fully active states. As part of their study, Gurumurthi et al. develop power models for a disk drive that supports 15 different RPM levels. Using these models they propose a simple heuristic for RPM modulation. The heuristic involves some cooperation between the disks and a controller that oversees the disks. The controller initially defines a range of RPM levels for each disk; the disks modulate their RPMs within this

range based on their demands. They periodically monitor their input queues, and whenever the number of requests exceeds a threshold, they increase their RPM settings by 1. Meanwhile, the controller oversees the response times in the system. When the response time is above a threshold, this is a sign of performance degradation, and the controller immediately requests the disks to switch to full power. When on the other hand, the response times are sufficiently low, this is a sign that the disks could be slowed down further, and the controller increases the minimum RPM threshold setting.

3.6.2. Network Interfaces. Network interfaces such as wireless cards present a unique challenge for resource hibernation techniques because the naive solution of shutting them off would disconnect hosts from other devices with which they may have to communicate. Thus power management strategies also include protocols for synchronizing the communication of these devices with other devices in the network. Kravets et al. [1998] have developed a heuristic for doing this at the operating system level. One of its key features is that it uses timers to track the idleness of devices. If a device should be transmitting but its idle timer expires, it enters a listening mode or goes to sleep. To decide how long devices should remain idle before shifting into listening mode, Kravets et al. apply an adaptive threshold algorithm. When the algorithm detects communication activity, it decreases the acceptable hibernation time. When it detects idle periods, it increases the time.

Threshold-based schemes such as this one allow a network card to remain idle for an extended length of time before shutting down. Hom and Kremer [2003] have argued that compilers can shut down a card sooner using a model of future program behavior. Their compiler-based hibernation technique targets a simplified scenario where a mobile device's virtual memory resides on a server. Page faults require activating the device's wireless card so that pages can be sent from the server to the device. The technique identifies program regions where the card should hybernate based on the nature of array accesses in each region. If an accessed array is not in local memory, the card is activated to transfer it from the remote server. To determine what arrays are in memory during each region's execution, the compiler simulates a least-recently-used memory bank.

3.6.3. Displays. Displays are a major source of power consumption. Normally, an operating system dims the display after a sufficiently long interval with no user response. This heuristic may not coincide with a user's intentions.

Two examples of more advanced techniques that manage displays based on user activity are *face-off* and *zoned backlighting*. They have yet to be implemented in commercial systems.

Face-off [Dalton and Ellis 2003] is an experimental face recognition system on an IBM T21 Thinkpad laptop running Red Hat Linux. It periodically photographs the monitor's perspective and detects a face through large regions of skin color. It then controls the display using the ACPI interface. Its developers tested it on a long network transfer during which the user looked away from the screen. Face-off saved more power than the default display manager. A drawback is the repetitive polling for face detection. For future work, the authors suggest detecting a user's presence through low-power heat microsensors. These sensors can trigger the face recognition algorithm when they detect a user.

Zoned backlighting [Flinn and Satyanarayanan 1999] targets future display systems that will allow software to selectively adjust the brightness of different regions of the screen based on usage patterns and battery drain. Researchers at Carnegie Mellon have proposed ideas for exploiting these displays. The underlying assumption is that users are interested in only a small part of the screen. One idea is to make foreground windows with keyboard and mouse focus brighter than

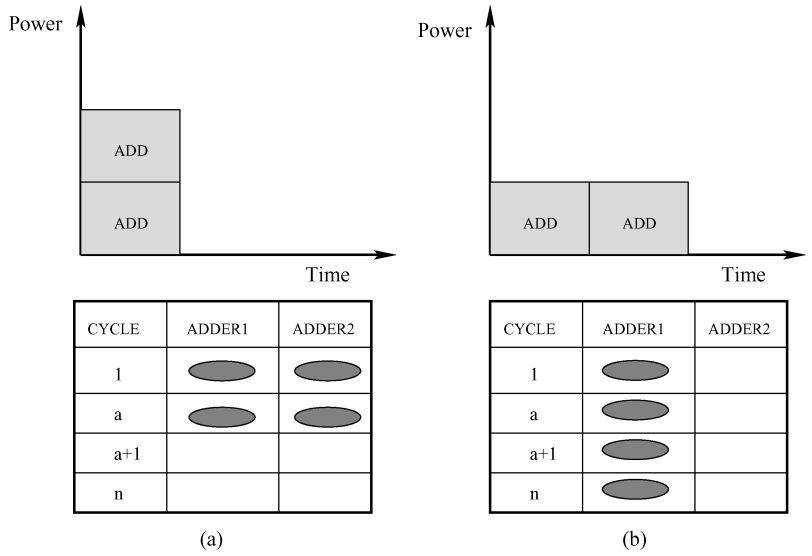

Fig. 18. Performance versus power.

back-ground windows. Another is to scale images to allow most of the screen to dim when the battery is low.

3.7. Compiler-Level Power Management

3.7.1. What Compilers Can Do. There are many ways a compiler can help reduce power regardless of whether a processor explicitly supports software-controlled power reduction. Aside from generating code that reconfigures hardware units or activates power reduction mechanisms that we have seen, compilers can apply common performance-oriented optimizations that also save energy by reducing the execution time, optimizations such as Common Subexpression Elimination, Partial Redundancy Elimination, and Strength Reduction. However, some performance optimizations increase code size or parallelism, sometimes increasing resource usage and peak power dissipation. Examples include Loop Unrolling and Software Pipelining. Researchers [Givargis et al. 2001] have developed models for relating performance and power, but these models are relative to specific architectures. There is no fixed relationship between performance and power across all architectures and applications. Figure 18 compares the power dissipation and execution time of doing two additions in parallel (as in a VLIW architecture) (a) and doing them in succession (b). Doing two additions in parallel activates twice as many adders over a shorter period. It induces higher peak resource usage but fewer execution cycles and is better for performance. To determine which scenario saves more energy, one needs more information such as the peak power increase in scenario (a), the execution cycle increase in scenario (b), and the total energy dissipation per cycle for (a) and (b). For ease of illustration, we have depicted the peak power in (a) to be twice that of (b), but all of these parameters may vary with respect to the architecture.

Compilers can reduce memory accesses to reduce energy dissipated in these accesses. One way to reduce memory accesses is to eliminate redundant load and store operations. Another is to keep data as close as possible to the processor, preferably in the registers and lower-level caches, using aggressive register allocation techniques and optimizations improving cache usage. Several loop transformations alter data traversal patterns to make better use of the cache. Examples include Loop Interchange, Loop Tiling, and Loop

Fusion. The assignment of data to memory locations also influences how long data remains in the cache. Though all these techniques reduce power consumption along one or more dimensions, they might be suboptimal along others. For example, to exploit the full bandwidth a banked memory architecture provides, one may need to disperse data across multiple memory banks at the expense of cache locality and energy.

3.7.2. Remote Compilation and Remote Execution. A new area of research for compilers involves compiling and executing applications jointly between mobile devices and more powerful servers to reduce execution time and increase battery life. This is an area frought with challenges such as the decision of what program sections to compile or execute remotely. Other issues include application partitioning between multiple servers and handhelds, application migration, and fault tolerance. Though researchers have yet to address all these issues, a number of studies provide valuable insights into the benefits of partitioning.

Recent studies of low-power Java application partitioning are by Palm et al. [2002] and Chen et al. [2003]. Palm et al. compare the energy costs of remote and local compilation and optimization. They examine four scenarios in which a handheld downloads code from a server and executes it locally. In the first two scenarios, the handheld downloads bytecode, compiles it, and executes it. In the remaining scenarios, the server compiles the bytecode and the handheld downloads the native codestream. Palm et al. find that, when a method's compilation time exceeds the execution time, remote compilation is favorable to local compilation. However, if no optimizations are used, remote compilation can be more expensive due to the cost of downloading code from the server. Overall, the largest energy savings are obtained by compiling and optimizing code remotely but sending results to the handheld exactly when needed to minimize idle time.

While Palm et al. [2002] restrict their studies to local and remote compilation, Chen et al. [2003] also examine the benefits of executing compiled Java code remotely. Their testbed consists of a 750MHz SPARC workstation server and a client handheld running a microSPARC-IIep embedded processor. The programmer initially annotates a set of candidate methods that the client may execute remotely. Before calling any of these methods, the client decides where to compile and execute the method based on computation and communication costs. It also selects optimization levels for compilation. If remote execution is favorable, the client transfers all method data to the server via the object serialization interface. The server uses reflection to invoke the method and serialization to send back the results. Chen et al. evaluate different compilation and execution strategies in this framework through simulation and analytical models. They consider seven partitioning strategies, five of which are static and two of which are dynamic. While the static strategies fix the partitioning decision for all methods of a program, the dynamic ones decide the best strategy for each method at call time. Chen et al. find the best static strategy to vary depending on input size and communication channel quality. When the quality is low, the client's network interface must send data at a higher transmission power to reach the server. For large inputs and good channel quality, static remote execution prevails over other static strategies. Overall, Chen et al. find the highest energy savings to be achieved by a dynamic strategy that decides where to compile and execute each method.

The Proxy Virtual Machine [Venkatachalam et al. 2003] reduces the energy consumption of mobile devices by combining application partitioning with dynamic optimization. The framework (Figure 19) positions a powerful server infrastructure, the proxy, between mobile devices and the Internet. The proxy includes a just-in-time compiler and bytecode translator. A high bandwidth, low-latency secure wireless connection mediates communication between the proxy and mobile devices in

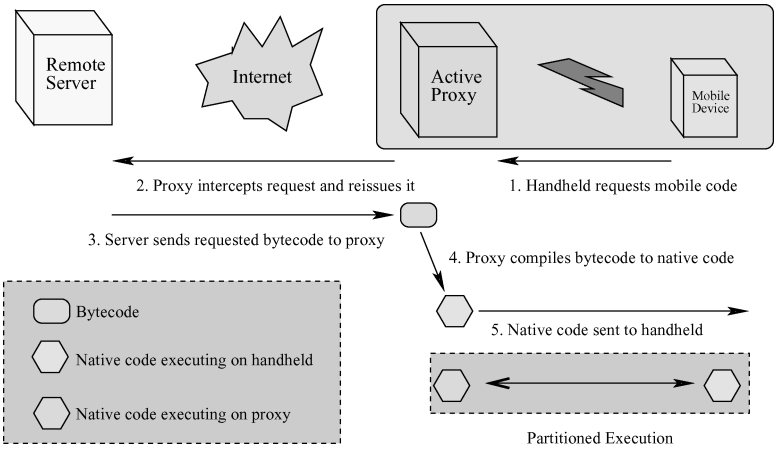

Fig. 19. The proxy VM framework.

the vicinity. Users can request Internet applications as they normally would through a Web browser. The proxy intercepts these requests and reissues them to a remote Internet server. The server sends the proxy the application in mobile code format. The proxy verifies the application, compiles it to native code, and sends part of the generated code to the target. The remainder of the code executes on the proxy itself to significantly reduce energy consumption on the mobile device.

3.7.3. The Limitations of Statically Optimizing Compilers. The drawback of compiler-based approaches is that a compiler's view is usually limited to the programs it is compiling. This raises two problems. First, compilers have incomplete information about how a program will actually behave, information such as the control flow paths and loop iterations that will be executed. Hence, statically optimizing compilers often rely on profiling data that is collected from a program prior *to execution*, to determine what optimizations should be applied in different program regions. A program's actual runtime behavior may diverge from its behavior in these "simulation runs", thus leading to suboptimal decisions.

The second problem is that compiler optimizations tend to treat programs as if they exist in a vacuum. While this may be ideal for specialized embedded systems where the set of programs that will execute is determined ahead of time and where execution behavior is mostly predictable, real systems tend to be more complex. The events occurring within them continuously change and compete for processor and memory resources. For example, context switches abruptly shift execution from a region of one program to a completely different region of another program. If a compiler had generated code so that the two regions put a device into different power mode settings, then there will be an overhead to transition between these settings when execution flows from one region to another.

3.7.4. Dynamic Compilation. Dynamic compilation addresses some of these problems by introducing a feedback loop. A program is compiled but is then also monitored as it executes. As the behavior of the program changes, possibly along with other changes in the runtime environment (e.g., resource levels), the program is recompiled to adapt to these changes. Since the program is continuously recompiled in response to runtime feedback, its code is of a significantly higher quality than it would have been if it had been generated by a static compiler.

There are a number of scenarios where continuous compilation can be effective.

For example, as battery capacity decreases, a continuous compiler can apply more aggressive transformations that trade the quality of data output for reduced power consumption. (For example, a compiler can replace expensive floating point operations with integer operations, or generate code that dims the display.) However, there are tradeoffs. For example, the cost function for a dynamic compiler needs to weigh the overhead of recompiling a program with the energy that can be saved. Moreover, there is the issue of how to profile an executing program in a minimally invasive, low overhead way.

While there is a large body of work in dynamic compilation for performance [Kistler and Franz 2001; Kistler and Franz 2003], dynamic recompilation for power is still a young field with many open problems. One of the earliest commercial dynamic compilation systems is Transmeta's Crusoe processor [Transmeta Corporation 2003; Transmeta Corporation 2001] which we will discuss in Section 3.10. One of the earliest academic studies of power-aware dynamic recompilation was done by Unnikrishnan et al. [2002]. Their idea is to instrument critical program regions with sensitivity lists that detect changes in various energy budgets and hot-swap optimized versions of these regions with the original versions based on these changes. Each optimized version is precomputed ahead of time, using offline energy estimation techniques. The authors target a banked memory architecture and focus on loop transformations such as tiling, unrolling, and interchange. Their implementation is based on Dyninst, a tool that allows program regions to be patched at runtime.

Though dynamic compilation for power is still in its infancy, there is a growing body of work in runtime power monitoring. The work that is most relevant for dynamic compilation is that of Isci and Martonosi [2003]. These researchers have developed techniques for profiling the power behavior of programs at runtime using information from hardware performance counters. They target a Pentium 4 processor which allows one to sample 18 different performance counter events simultaneously. For each of the core components of this processor Isci and Martonosi have developed power models that relate their power consumption to their usage statistics. In their actual setup, they take two kinds of power readings in parallel. One is based on current measurements and the other is based on the performance counters. These readings give the total power as well as a breakdown of power consumed among the various components. Isci and Martonosi find that their performance counter-based approach is nearly as accurate as the current measurements and is also able to distinguish between different phases in program execution.

3.8. Application-Level Power Management

Researchers have been exploring how to give applications a larger role in power management decisions. Most of the recent work has two goals. The first is to develop techniques that enable applications to adapt to their runtime environment. The second is to develop interfaces allowing applications to provide hints to lower layers of the stack (i.e., operating systems, hardware) and likewise exposing useful information from the lower layers to applications. Although these are two separate goals, the techniques for achieving them can overlap.

3.8.1. Application Transformations and Adaptations. As a starting point, some researchers [Tan et al. 2003] have adopted an "architecture-centric" view of applications that allows for some high-level transformations. These researchers claim that an application's architecture consists of fundamental elements that comprise all applications, namely processes, event handlers, and communication mechanisms. Power is consumed when these various elements interact during activities such as context switches and interprocess communication.

They propose a low-power application design methodology. Initially, an application is represented as a *software architecture graph* (SAG) [Tan et al. 2003]

which captures how it has been built out of processes and events. It is then run through a simulator to measure its base energy consumption which is then compared to its energy consumption when transformations are applied to the SAG, transformations that include merging processes to reduce interprocess communication, replacing expensive IPC mechanisms by cheaper mechanisms, and migrating computations between processes.

To determine the optimal set of transformations, the researchers use a greedy approach. They keep applying transformations that reduce energy until no more transformations remain to be applied. This results in an optimized version of the same application that has a more energy efficient mix of processes, events, and communication mechanisms.

Sachs et al. [2003] have explored a different kind of adaptation that involves trading the accuracy of computations for reduced energy consumption. They propose a video encoder that allows one to vary its compression efficiency by selectively skipping the motion search and discrete cosine transform phases of the encoding algorithm. The extent to which it skips these phases depends on parameters chosen by the adaptation algorithm. The target architecture is a processor with support for DVS and architectural adaptations (e.g., configurable caches). Two algorithms work side by side. One algorithm tunes the hardware parameters at the start of each frame, while another tunes the parameters of the video encoder while a frame is being processed.

Yet another form of adaptation involves reducing energy consumption by trading off fidelity or the quality of data presented to users. This can take many forms. One example of a fidelity reduction system is Odyssey [Flinn and Satyanarayanan 1999]. Odyssey is an operating system and execution framework for multimedia and Web applications. It continuously monitors the resources used by applications and alerts applications whose resources fall below requested levels. In turn, the applications lower their quality of service until resources are plentiful. Originally, Odyssey supported four applications, a video player, speech recognizer, map viewer, and Web browser. The video player, for instance, would download images from a server storing multiple copies at different compression levels. When bandwidth is low, it would download the compressed versions of images to transfer less data over the network. As battery resources become scarce, it would also reduce the size of the images to conserve energy.

One example of an operating system that assists application adaptation is ECOSystem [Zeng et al. 2002]. It treats energy as a commodity assigned to the different resources in a system. The idea is to allow resources to operate so long as they have enough of the commodity. Thus resource management in ECOSystem relies on the notion of "currentcy", an abstraction that models the power resource as a monetary unit. Essentially, the different resources in a computer (e.g., ALU, memory) have prices for using them, and applications pay these prices with currentcy. Periodically, the operating system distributes a fixed amount of currentcy among applications, using the desired battery drain rate as the basis for deciding on how much to distribute. Applications can use specific resources only if they can pay for them. As an application executes, it expends its currentcy. The application is interrupted once it depletes its currentcy.

3.8.2. Application Hints. In one of the earliest works on application hints, Pereira et al. [2002] propose a software architecture containing two layers of API calls, one allowing applications to communicate with the operating system, and the other allowing the operating system to communicate with the underlying hardware. The API layer that is exposed to applications allows applications to drop hints to the operating system such as the start times, deadlines, and expected execution times of tasks. Using these hints provided by the application, the operating system will have a better understanding of the possible future behavior of programs and

will thus be able to make better DVS decisions.

Similar to this is the work by Anand et al. [2004] which considers a setting where applications adapt by deciding when to access different I/O devices based on the relative costs of accessing these devices. For example, it may be expensive to access data resident on a disk if the disk is powered down and would take a long time to restart. In such cases, an application may choose instead to access the data through the network. To aid in these adaptations, Anand et al. propose an API layer that gives applications control over all I/O devices but abstracts away the details of these devices. The API functions as a giant regulating dial that applications can tune to specify their desired power/performance tradeoffs and which transparently manages all the devices to achieve these tradeoffs. The API also give applications more fine-grained control over devices they may want to access. One of its unique features is that it allows applications to issue ghost hints: if an application chose not to access a device that was deactivated but would have saved more energy by accessing that device if it had been activated, then it can send the device a message to this effect. After a few of these ghost hints, the device automatically becomes activated, anticipating that the application will need to use it.

Heath et al. [2004] propose similar work to increase opportunities for spinning down idle hard disks and saving energy. In their framework, a compiler performs various code transformations to cluster disk accesses and then inserts hints that tell the operating system how long these clustered disk accesses will take. The operating system is then able to serve these disk requests in a single batch and power down the disk for longer periods of time.

3.9. Cross-Layer Adaptations

Because power consumption depends on decisions spanning the entire stack from transistors to applications, a research question that is becoming increasingly popular is how to develop holistic approaches that integrate information from multiple levels (e.g., compiler, OS, hardware) into power management decisions. There are already a number of systems being developed to explore cross-layer adaptations. We give four examples of such systems.

Forge [Mohapatra et al. 2003] is an integrated power management framework for networked multimedia applications. Forge targets a typical scenario where users request video streams for their handheld devices and these requests are filtered by a remote proxy that transcodes the streams and transmits them to the users at the most energy efficient QoS levels.

Forge integrates multiple levels of adaptations. The hardware level contains a frequency and voltage scaling interface, a network card interface (allowing the network card to be powered down when idle), as well as interfaces allowing various architectural parameters (e.g., cache ways, register file sizes) to be tuned. A level above hardware are the operating system and compiler, which control the architectural knobs, and a level above the operating system is a distributed middleware frame-work, part of which resides on the handheld devices and part of which resides on the remote proxy. The middleware on each handheld device monitors the energy statistics of the device (through the OS), and sends feedback to the middleware on the proxy. The middleware on the proxy uses this feedback to decide how to regulate network traffic and at what QoS levels to stream requested video feeds.

Different heuristics are used for different adaptations. To determine the optimal cache parameters for a video feed, one simulates the feed offline at each QoS level for all possible variations of cache size and associativity and determines the most energy optimal configuration. This configuration is automatically chosen when the feed executes. The cache configuration, in turn, affects how long it takes to decode each video frame. If the decoding time is less than the frame's deadline, then the DVS algorithm slows down the frame to

exactly meet the deadline. Meanwhile on the proxy, the middleware layer continually receives requests for video feeds and information about the resource levels on various devices. When it receives a request from a user on a handheld device, it checks whether there is enough remaining energy on the device allowing it to receive the video feed at an acceptable quality. If there is, then it transmits the feed at the highest quality that does not exceed the energy constraints of the device. Otherwise, it rejects the user's request or sends the feed at a lower quality. When the proxy sends a video feedback back to the users, it transmits the feed in bursts of packets, allowing the users' network cards to be shut down over longer idle periods to save energy.

Grace [Yuan and Nahrstedt 2003] is another cross-layer adaptation framework that attempts to integrate dynamic voltage scaling, power-aware task scheduling, and QoS setting. Its target applications are real-time multimedia tasks with fixed periods and deadlines. Grace applies two levels of adaptations, global and local. Global adaptations respond to larger resource variations and are applied less frequently because they are more expensive. These adaptations are controlled by a central coordinator that continuously monitors battery levels and application workloads and responds to widespread changes in these variables. Local adaptations, on the other hand, respond to smaller workload variations within a task. These adaptations are controlled by local adaptors. There are three local adaptors, one to set the CPU frequency, another to schedule tasks, and a third to adapt QoS parameters. Major changes in the runtime environment, such as low battery levels or wide variations in workload, trigger the central coordinator to decide upon a new set of global adaptations. The coordinator chooses the most energy-optimal adaptations by solving a constrained optimization problem. It then broadcasts its decision to the local adaptors which implement these changes but are free to adjust these initial settings within the execution of specific tasks.

Another multilayer framework for multimedia applications has been developed by Fei et al. [2004]. Their framework consists of four layers divided into user space and system space. The system space consists of the target platform and the operating system running on top of it. Directly above the operating system in user space is a middleware layer that decides how to apply adaptations, guided by information it receives from the other layers. All applications execute on top of the middleware coordinator which is responsible for deciding when to admit different applications into the system and how to select their QoS levels. When applications first enter the system, they store their meta-information inside the middleware layer. This includes information about the QoS levels they offer and the energy consumption for each of these levels. The middleware layer, in turn, monitors battery levels using counters inside the operating system. Based on the battery levels, the middleware decides on the most energy efficient QoS setting for each application. It then admits applications into the system according to user-defined priorities.

A fourth example is the work of AbouGhazaleh et al. [2003] which explores how compilers and operating systems can interact to save energy. This work targets real-time applications with fixed deadlines and worst-case execution times. In this approach, the compiler instruments specific program regions with code that saves the remaining worst-case execution cycles into a register. The operating system periodically reads this register and adjusts the speed of the processor so that the task finishes by its worst-case execution time.

3.10. Commercial Systems

To understand what power management techniques industry is adopting, we examine the low-power techniques used in four widely used processors, the Pentium 4, Pentium M, the PXA27x, and Transmeta Crusoe. We then discuss three power management strategies by IBM, ARM, and National Semiconductor that are rapidly gaining in importance.

3.10.1. The Pentium 4 Processor. Though its goal is high performance, the Pentium 4 processor also contains features to manage its power consumption [Gunther et al. 2001; Hinton et al. 2004]. One of Intel's goals in including these features was to prevent the Pentium's internal temperatures from becoming dangerously high due to the increased power dissipation. To meet this goal, the designers included a thermal detection and response mechanism which inhibits the processor clock whenever the observed temperature exceeds a safe threshold. To monitor temperature variations, the processor features a diode-based thermal sensor. The sensor sits at the hottest area of the chip and measures temperature via voltage drops across the diode. Whenever the temperature increases into a danger zone, the sensor issues a STOPCLOCK request, causing the main clock signal to be inhibited from reaching most of the processor until the temperature is no longer in the danger zone. This is to guarantee response time while the chip is cooling. The temperature sensor also provides temperature information to higher levels of software through output signals (some of which are in registers), allowing the compiler or operating system, for instance, to activate other techniques in response to the high temperatures.

The Pentium 4 also supports the low-power operating states defined by the Advanced Configuration and Power Interface (ACPI) specification, allowing software to control the processor power modes. In particular, it features a model-specific register allowing software to influence the processor clock. In addition to stopping the clock, the Pentium 4 features the Intel Basic Speedstep Technology which allows two settings for the processor clock frequency and voltage, a high setting for performance and a low setting for power. The high setting is normally used when the computer is connected to a wall outlet, and the low setting is normally used when the computer is running on batteries as in the case of a laptop.

3.10.2. The Pentium M Processor. The Pentium M is the fruit of Intel's efforts to bring the Pentium 4 to the mobile Domain. It carefully balances performance enhancing features with several power-saving features that increase the battery lifetime [Genossar and Shamir 2003; Gochman et al. 2003]. It uses three main strategies to reduce dynamic power consumption: reducing the total instructions and micro-operations executed, reducing the switching activity in the circuit, and reducing the energy dissipated per transistor switch.

To save energy, the Pentium M integrates several techniques that reduce the total switching activity. These include hardware for predicting idle units and inhibiting their clock signals, buses whose components are activated only when data needs to be transferred, and a technique called *execution stacking* which clusters units that perform similar functions into similar regions so that the processor can selectively activate the parts of the circuit that will be needed by an instruction.

To reduce the static power dissipation, the Pentium M incorporates low leakage transistors in the caches. To further reduce both dynamic and leakage energy throughout the processor, the Pentium M supports an enhanced version of the Intel SpeedStep technology, which unlike its predecessor, allows the processor to transition between 6 different frequency and voltage settings.

3.10.3. The Intel PXA27x Processors. The Intel PXA27x processors used in many wireless handheld devices feature the Intel Wireless Speedstep power manager [Intel Corporation 2004]. This power manager uses memory boundedness as the criterion for deciding how to manage the processor's power modes. It receives information from an idle profiler which monitors the system idle thread and a performance profiler that monitors the processor's built-in performance counters to gather statistics pertaining to cache and TLB misses, instructions executed, and

processor stalls. Using this information, the power manager determines how memory bounded the workload is and decides how to adjust the processor's power modes as well as its clock frequency and voltage.

3.10.4. The Transmeta Crusoe Processor. Transmeta's Crusoe processor [Transmeta Corporation 2001, 2003] saves power not only by means of its LongRun Power Manager but also through its overall design which shifts many of the complexities in instruction execution from hardware into software. The Crusoe relies on a *code morphing engine*, a layer of software that interprets or compiles x86 programs into the processor's native VLIW instructions, saving the generated code in a *Translation Cache* so that they can be reused. This layer of software is also responsible for monitoring executing programs for hotspots and reoptimizing code on the fly. As a result, features that are typically implemented in hardware (e.g., out-of-order execution) are instead implemented in software. This reduces the total on-chip real estate and its accompanying power dissipation. To further reduce the power dissipation, Crusoe's LongRun modulates the clock frequency and voltage according to workload demands. It identifies idle periods of the operating system and scales the clock frequency and voltage accordingly.

Transmeta's later generation Efficeon processor contains enhanced versions of the code morphing engine and the LongRun Power Manager. Some news articles claim that the new LongRun includes techniques to reduce leakage, but the details are proprietary.

3.10.5. IBM Dynamic Power Management. IBM has developed an extensible operating system module [Brock and Rajamani 2003] that allows power management policies to be developed for a wide range of embedded devices. For portability, this module abstracts away from the underlying architecture and provides a simple interface allowing designers to specify device-specific parameters (e.g., available clock frequencies) and to integrate their own policies in a "plug-and-play" manner. The main idea is to model the underlying hardware as a state machine composed of different *operating points* and *operating states*. The operating points are clock frequency and voltage settings, while the operating states are power mode settings such as active, idle and deep sleep. Designers specify the operating points and states for their target architecture, and using this abstraction, they write policies for transitioning between the states. IBM has experimented with a wide range of policies for the PowerPC 405LP, from simple policies that adjust the frequency based on whether the processor is idle or active, to more complex policies that integrate information about application workloads and deadlines.

3.10.6. Powerwise and Intelligent Energy Management. National Semiconductor and ARM Inc. have collaborated to develop a single end-to-end energy management system for embedded processors. Their system brings together two technologies, ARM's Intelligent Energy Management Software (IEM), which modulates the processor speed based on workloads, and National Semiconductor's Powerwise Adaptive Voltage Scaling (AVS), which dynamically adjusts the supply voltage for a given clock frequency based on variations in temperature and other ambient system effects.

The IEM software consists of a three-level decision hierarchy of policies for deciding how fast the processor should run. At the bottom is a baseline algorithm that selects the optimal frequency based on estimating future task workloads from past workloads measured over a sliding history window. At the top is an algorithm more suited for interactive and media applications, which considers how fast these applications would need to run to deliver smooth performance to users. Between these two layers is an interface allowing applications to communicate their

workload requirements directly. Each of these three layers combines its performance prediction with a confidence rating, which indicates how to compare its decision with that of other layers. Whenever there is a context switch, the IEM software analyzes the decisions and confidence ratings of each layer to finally decide how fast to run the processor.

The Powerwise Adaptive Voltage Scaling technology [Dhar et al. 2002] aims to save more energy than conventional voltage scaled systems by choosing the minimum acceptable voltage for a given clock frequency. Conventional DVS processors determine the appropriate voltage for a clock frequency by referring to a table that is developed offline using a worst-case model of ambient hardware characteristics. As a result, the voltages chosen are often higher than they need to be. Powerwise gets around this problem by adding a feedback loop. It continually monitors variations in temperature and other ambient system effects through performance counters and uses this information to dynamically adjust the supply voltage for each clock frequency. This results in an additional 45% energy savings over conventional voltage scaling.

3.11. A Glimpse Into Emerging Radical Technologies

Imagine a laptop that runs on hydrogen fuel or a mobile phone that is powered by the same type of engines that propel gigantic aircraft. Such ideas may sound far fetched, but they are exactly the kinds of innovations that some researchers claim will eventually solve the energy problem. We close our survey by briefly discussing two of the more radical technologies that are likely to gain in importance over the next decade. The focus of these techniques is on improving *energy efficiency*. The techniques include fuel cells and MEMS systems.

3.11.1. Fuel Cells. Fuel cells [Dyer 2004] are being developed to replace the batteries used in mobile devices. Batteries have a limited capacity. Once depleted, they must be discarded unless they are rechargeable, and recharging a battery can take several hours and even rechargeable batteries eventually die. Moreover, battery technology has been advancing slowly. Building more efficient batteries still means building bigger batteries, and bigger and heavier batteries are not suited for small mobile devices.

Fuel cells are similar to batteries in that they generate electricity by means of a chemical reaction but, unlike batteries, fuel cells can in principle supply energy indefinitely. The main components of a fuel cell are an anode, a cathode, a membrane separating the anode from the cathode, and a link to transfer the generated electricity. The fuel enters the anode, where it reacts with a catalyst and splits into protons and electrons. The protons diffuse through the membrane, while the electrons are forced to travel through the link generating electricity. When the protons reach the cathode, they combine with Oxygen in the air to produce water and heat as by-products. If the fuel cell uses a water-diluted fuel (as some do), then this waste water can be recycled back to the anode.

Fuel cells have a number of advantages. One advantage is that fuels (e.g., hydrogen) are abundantly available from a wide variety of natural resources, and many offer energy densities high enough to allow portable devices to run far longer than they do on batteries. Another advantage is that refueling is significantly faster than recharging a battery. In some cases, it merely involves spraying more fuel into a tank. A third advantage is that there is no limit to how many times a fuel cell can be refueled. As long as it contains fuel, it generates electricity.

Several companies are aggressively developing fuel cell technologies for portable devices including Micro Fuel Cell Inc., NEC, Toshiba, Medis, Panasonic, Motorola, Samsung, and Neah Systems. Already a number of prototype cells have emerged as well as prototype mobile computers powered by hydrogen or alcohol-based fuels. Despite these rapid advances, some industry analysts estimate that it

will take another 5-10 years for fuel cells to become commonplace in mobile devices.

Fuel cells also have several drawbacks. First, they can get very hot (e.g., 500-1000 celsius). Second, the metallic materials and mechanical components that they are composed of can be quite expensive. Third, fuel cells are flammable. In particular, fuel-powered devices will require safety measures as well as more flexible laws allowing them inside airplanes.

3.11.2. MEMS Systems. Microelectrical and Mechanical Systems (MEMS) are miniature versions of large scale devices commonly used to convert mechanical energy into electrical energy. Researchers at MIT and the Georgia Institute of Technology [Epstein 2004] are exploring a radical way of using them to solve the energy problem. They are developing prototype millimeter scale versions of the gigantic gas turbine engines that power airplanes and drive electrical generators. These microengines, they claim, will give mobile computers unprecedented amounts of untethered lifetime.

The microengines work using similar principles as their large scale counterparts. They suck air into a compressor and ignite it with fuel. The compressed air then spins a set of turbines that are connected to a generator to generate electrical power. The fuels used could be hydrogen, diesel based, or more energy-dense solid fuels.

Made from layers of silicon wafers, these tiny engines are supposed to output the same levels of electrical power per unit of fuel as their large scale counterparts. Their proponents claim they have two advantages. First, they can output far more power using less fuel than fuel cells or batteries alone. In fact, the ones under development are expected to output 10 to 100 Watts of power almost effortlessly and keep mobile devices powered for days. Moreover, as a result of their high energy density, these engines would require less space than either fuel cells or batteries.

This technology, according to researchers, is likely to be commercialized over the next four years. However, it is too early to tell whether it will in fact replace batteries and fuel cells. One problem is that jet engines produce hot exhaust gases that could raise chip temperatures to dangerous levels, possibly requiring new materials for a chip to withstand these temperatures. Other issues include flammability and the power dissipation of the rotating turbines.

4. CONCLUSION

Power and energy management has grown into a multifaceted effort that brings together researchers from such diverse areas as physics, mechanical engineering, electrical engineering, design automation, logic and high-level synthesis, computer architecture, operating systems, compiler design, and application development. We have examined how the power problem arises and how the problem has been addressed along multiple levels ranging from transistors to applications. We have also surveyed major commercial power management technologies and provided a glimpse into some emerging technologies. We conclude by noting that the field is still active, and that researchers are continually developing new algorithms and heuristics along each level as well as exploring how to integrate algorithms from multiple levels. Given the wide variety of microarchitectural and software techniques available today and the astoundingly large number of techniques that will be available in the future, it is highly likely that we will overcome the limits imposed by high power consumption and continue to build processors offering greater levels of performance and versatility. However, only time will tell which approaches will ultimately succeed in solving the power problem.

REFERENCES

AbouGhazaleh, N., Childers, B., Mosse, D., Melhem, R., and Craven, M. 2003. Energy management for real-time embedded applications with compiler support. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, 284–293.

ALBONESI, D., DROPSHO, S., DWARKADAS, S., FRIEDMAN, E., HUANG, M., KURSUN, V., MAGKLIS, G., SCOTT, M., SEMERARO, G., BOSE, P., BUYUKTOSUNOGLU, A., COOK, P., AND SCHUSTER, S. 2003. Dynamically tuning processor resources with adaptive processing. *IEEE Computer Magazine 36*, 12, 49–58.

ANAND, M., NIGHTINGALE, E., AND FLINN, J. 2004. Ghosts in the machine: Interfaces for better power management. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*. 23–35.

AZEVEDO, A., ISSENIN, I., CORNEA, R., GUPTA, R., DUTT, N., VEIDENBAUM, A., AND NICOLAU, A. 2002. Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 168–175.

BAHAR, R. I. AND MANNE, S. 2001. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. ACM Press, 218–229.

BANAKAR, R., STEINKE, S., LEE, B.-S., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*. 73–78.

BANERJEE, K. AND MEHROTRA, A. 2001. Global interconnect warming. *IEEE Circuits and Devices Magazine 17*, 5, 16–32.

BISHOP, B. AND IRWIN, M. J. 1999. Databus charge recovery: practical considerations. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 85–87.

BROCK, B. AND RAJAMANI, K. 2003. Dynamic power management for embedded systems. In *Proceedings of the IEEE International SOC Conference*. 416–419.

BUTTAZZO, G. C. 2002. Scalable applications for energy-aware processors. In *Proceedings of the 2nd International Conference On Embedded Software*. Springer-Verlag, 153–165.

BUTTS, J. A. AND SOHI, G. S. 2000. A static power model for architects. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. Monterey, CA. 191–201.

BUYUKTOSUNOGLU, A., SCHUSTER, S., BROOKS, D., BOSE, P., COOK, P. W., AND ALBONESI, D. 2001. An adaptive issue queue for reduced power at high performance. In *Proceedings of the 1st International Workshop on Power-Aware Computer Systems*. 25–39.

CALHOUN, B. H., HONORE, F. A., AND CHANDRAKASAN, A. 2003. Design methodology for fine-grained leakage control in MTCMOS. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 104–109.

CHEN, D., CONG, J., LI, F., AND HE, L. 2004. Low-power technology mapping for FPGA architectures with dual supply voltages. In *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*. ACM Press, 109–117.

CHEN, G., KANG, B., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. 2003. Energy-aware compilation and execution in java-enabled mobile devices. In *Proceedings of the 17th Parallel and Distributed Processing Symposium*. IEEE Press, 34a.

CHOI, K., SOMA, R., AND PEDRAM, M. 2004. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 174–179.

CLEMENTS, P. C. 1996. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*. IEEE Computer Society, 16–25.

DALLY, W. J. AND TOWLES, B. 2001. Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th Conference on Design Automation*. ACM Press, 684–689.

DALTON, A. B. AND ELLIS, C. S. 2003. Sensing user intention and context for energy management. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*. 151–156.

DE, V. AND BORKAR, S. 1999. Technology and design challenges for low power and high performance. In *Proceedings of the International Symposium on Low Power Electronics and Design ISLPED'99* . ACM Press, 163–168.

DHAR, S., MAKSIMOVIC, D., AND KRANZEN, B. 2002. Closed-loop adaptive voltage scaling controller for standard-cell asics. In *Proceedings of the International Symposium on Low Power Electronics and Design ISLPED'02*. ACM Press, 103–107.

DINIZ, P. C. 2003. A compiler approach to performance prediction using empirical-based modeling. In *International Conference On Computational Science*. 916–925.

DROPSHO, S., BUYUKTOSUNOGLU, A., BALASUBRAMONIAN, R., ALBONESI, D. H., DWARKADAS, S., SEMERARO, G., MAGKLIS, G., AND SCOTT, M. L. 2002. Integrating adaptive on-chip storage structures for reduced dynamic power. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 141–152.

DROPSHO, S., SEMERARO, G., ALBONESI, D. H., MAGKLIS, G., AND SCOTT, M. L. 2004. Dynamically trading frequency for complexity in a gals microprocessor. In *Proceedings of the 37th International Symposium on Microarchitecture*. IEEE Computer Society, 157–168.

DUDANI, A., MUELLER, F., AND ZHU, Y. 2002. Energy-conserving feedback EDF scheduling for embedded systems with real-time constraints. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, 213–222.

DYER, C. 2004. Fuel cells and portable electronics. In *Symposium On VLSI Circuits Digest of Technical Papers* (2004). 124–127.

EBERGEN, J., GAINSLEY, J., AND CUNNINGHAM, P. 2004. Transistor sizing: How to control the speed and energy consumption of a circuit. In *the 10th International Symposium on Asynchronous Circuits and Systems*. 51–61.

EPSTEIN, A. 2004. Millimeter-scale, microelectromechanical systems gas turbine engines. *J. Eng. Gas Turb. Power 126*, 205–226.

ERNST, D., KIM, N., DAS, S., PANT, S., RAO, R., PHAM, T., ZIESLER, C., BLAAUW, D., AUSTIN, T., FLAUTNER, K., AND MUDGE, T. 2003. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 7–18.

FAN, X., ELLIS, C. S., AND LEBECK, A. R. 2003. Synergy between power-aware memory systems and processor voltage scaling. In *Proceedings of the Workshop on Power-Aware Computer Systems*. 164–179.

FEI, Y., ZHONG, L., AND JHA, N. 2004. An energy-aware framework for coordinated dynamic software management in mobile computers. In *Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*. 306–317.

FERDINAND, C. 1997. Cache behavior prediction for real time systems. Ph.D. thesis, Universität des Saarlandes.

FLAUTNER, K., REINHARDT, S., AND MUDGE, T. 2001. Automatic performance setting for dynamic voltage scaling. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*. ACM Press, 260–271.

FLINN, J. AND SATYANARAYANAN, M. 1999. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. ACM Press, 48–63.

FOLEGNANI, D. AND GONZALEZ, A. 2001. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. ACM Press, 230–239.

GAO, F. AND HAYES, J. P. 2003. ILP-based optimization of sequential circuits for low power. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 140–145.

GENOSSAR, D. AND SHAMIR, N. 2003. Intel Pentium M processor power estimation, budgeting, optimization, and validation. *Intel. Tech. J. 7*, 2, 44–49.

GHOSE, K. AND KAMBLE, M. B. 1999. Reducing power in superscalar processor caches using subbanking, multiple line buffers, and bit-line segmentation. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 70–75.

GIVARGIS, T., VAHID, F., AND HENKEL, J. 2001. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 25–30.

GOCHMAN, S., RONEN, R., ANATI, I., BERKOVITS, A., KURTS, T., NAVEH, A., SAEED, A., SPERBER, Z., AND VALENTINE, R. 2003. The Intel Pentium M processor: Microarchitecture and performance. *Intel. Tech. J. 7*, 2, 21–59.

GOMORY, R. E. AND HU, T. C. 1961. Multi-terminal network flows. *J. SIAM 9*, 4, 551–569.

GOVIL, K., CHAN, E., AND WASSERMAN, H. 1995. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*. ACM Press, 13–25.

GRUIAN, F. 2001. Hard real-time scheduling for low-energy using stochastic data and DVS processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 46–51.

GUNTHER, S., BINNS, F., CARMEAN, D., AND HALL, J. 2001. Managing the impact of increasing microprocessor power consumption. *Intel Tech. J.*

GURUMURTHI, S., SIVASUBRAMANIAM, A., KANDEMIR, M., AND FRANKE, H. 2003. DRPM: Dynamic speed control for power management in server class disks. *SIGARCH Comput. Architect. News 31*, 2, 169–181.

HEATH, T., PINHEIRO, E., HOM, J., KREMER, U., AND BIANCHINI, R. 2004. Code transformations for energy-efficient device management. *IEEE Trans. Comput. 53*, 8, 974–987.

HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., AND ROUSSEL, P. 2004. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Tech. J. 8*, 1, 1–17.

HO, Y.-T. AND HWANG, T.-T. 2004. Low power design using dual threshold voltage. In *Proceedings of the Conference on Asia South Pacific Design Automation* IEEE Press, (Piscataway, NJ,) 205–208.

HOM, J. AND KREMER, U. 2003. Energy management of virtual memory on diskless devices. In *Compilers and Operatings Systems for Low Power.* Kluwer Academic Publishers, Norwell, MA. 95–113.

HOSSAIN, R., ZHENG, M. AND ALBICKI, A. 1996. Reducing power dissipation in CMOS circuits by signal probability based transistor reording. In *IEEE Trans. Comput.-Aided Design Integrated Circuits Syst. 15*, 3, 361–368.

HSU, C. AND FENG, W. 2004. Effective dynamic voltage scaling through cpu-boundedness detection. In *Workshop on Power Aware Computing Systems*.

HSU, C.-H. AND KREMER, U. 2003. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In

Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM Press, 38–48.

HU, J., IRWIN, M., VIJAYKRISHNAN, N., AND KANDEMIR, M. 2002. Selective trace cache: A low power and high performance fetch mechanism. Tech. Rep. CSE-02-016, Department of Computer Science and Engineering, The Pennsylvania State University (Oct.).

HU, J., VIJAYKRISHNAN, N., IRWIN, M., AND KANDEMIR, M. 2003. Using dynamic branch behavior for power-efficient instruction fetch. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. 127–132.

HU, J. S., NADGIR, A., VIJAYKRISHNAN, N., IRWIN, M. J., AND KANDEMIR, M. 2003. Exploiting program hotspots and code sequentiality for instruction cache leakage management. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 402–407.

HUANG, M., RENAU, J., YOO, S.-M., AND TORRELLAS, J. 2000. A framework for dynamic energy efficiency and temperature management. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. ACM Press, 202–213.

HUANG, M. C., RENAU, J., AND TORRELLAS, J. 2003. Positional adaptation of processors: application to energy reduction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture ISCA '03*. ACM Press, 157–168.

HUGHES, C. J. AND ADVE, S. V. 2004. A formal approach to frequent energy adaptations for multimedia applications. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. IEEE Computer Society, 138.

HUGHES, C. J., SRINIVASAN, J., AND ADVE, S. V. 2001. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'34)*. IEEE Computer Society, 250–261.

Intel Corporation. 2004. *Wireless Intel Speedstep Power Manager*. Intel Corporation.

ISCI, C. AND MARTONOSI, M. 2003. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'36)*. IEEE Computer Society, 93–104.

ISHIHARA, T. AND YASUURA, H. 1998. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 197–202.

ITRSRoadmap. The International Technology Roadmap for Semiconductors. Available at http://public.itrs.net.

IYER, A. AND MARCULESCU, D. 2001. Power aware microarchitecture resource scaling. In *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE Press, 190–196.

IYER, A. AND MARCULESCU, D. 2002a. Microarchitecture-level power management. *IEEE Trans. VLSI Syst. 10*, 3, 230–239.

IYER, A. AND MARCULESCU, D. 2002b. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. ACM Press, 379–386.

JONE, W.-B., WANG, J. S., LU, H.-I., HSU, I. P., AND CHEN, J.-Y. 2003. Design theory and implementation for low-power segmented bus systems. *ACM Trans. Design Autom. Electr. Syst. 8*, 1, 38–54.

KABADI, M., KANNAN, N., CHIDAMBARAM, P., NARAYANAN, S., SUBRAMANIAN, M., AND PARTHASARATHI, R. 2002. Dead-block elimination in cache: A mechanism to reduce i-cache power consumption in high performance microprocessors. In *Proceedings of the International Conference on High Performance Computing*. Springer Verlag, 79–88.

KANDEMIR, M., RAMANUJAM, J., AND CHOUDHARY, A. 2002. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Proceedings of the 39th Conference on Design Automation*. ACM Press, 219–224.

KAXIRAS, S., HU, Z., AND MARTONOSI, M. 2001. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. ACM Press, 240–251.

KIM, C. AND ROY, K. 2002. Dynamic Vth scaling scheme for active leakage power reduction. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. IEEE Computer Society, 0163–0167.

KIM, N. S., FLAUTNER, K., BLAAUW, D., AND MUDGE, T. 2002. Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society Press, 219–230.

KIN, J., GUPTA, M., AND MANGIONE-SMITH, W. H. 1997. The filter cache: An energy efficient memory structure. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, 184–193.

KISTLER, T. AND FRANZ, M. 2001. Continuous program optimization: design and evaluation. *IEEE Trans. Comput. 50*, 6, 549–566.

KISTLER, T. AND FRANZ, M. 2003. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst. 25*, 4, 500–548.

KOBAYASHI AND SAKURAI. 1994. Self-adjusting threshold voltage scheme (SATS) for low-voltage high-speed operation. In *Proceedings of the IEEE Custom Integrated Circuits Conference*. 271–274.

KONDO, M. AND NAKAMURA, H. 2004. Dynamic processor throttling for power efficient computations. In *Workshop on Power Aware Computing Systems*.

KONG, B., KIM, S., AND JUN, Y. 2001. Conditional-capture flip-flop for statistical power reduction. *IEEE J. Solid State Circuits 36*, 8, 1263–1271.

KRANE, R., PARSONS, J., AND BAR-COHEN, A. 1988. Design of a candidate thermal control system for a cryogenically cooled computer. *IEEE Trans. Components, Hybrids, Manufact. Techn. 11*, 4, 545–556.

KRAVETS, R. AND KRISHNAN, P. 1998. Power management techniques for mobile communication. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*. ACM Press, 157–168.

KURSUN, E., GHIASI, S., AND SARRAFZADEH, M. 2004. Transistor level budgeting for power optimization. In *Proceedings of the 5th International Symposium on Quality Electronic Design*. 116–121.

LEBECK, A. R., FAN, X., ZENG, H., AND ELLIS, C. 2000. Power aware page allocation. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (2000). ACM Press, 105–116.

LEE, S. AND SAKURAI, T. 2000. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37th Conference on Design Automation*. ACM Press, 806–809.

LI, H., KATKOORI, S., AND MAK, W.-K. 2004. Power minimization algorithms for LUT-based FPGA technology mapping. *ACM Trans. Design Autom. Electr. Syst. 9*, 1, 33–51.

LI, X., LI, Z., DAVID, F., ZHOU, P., ZHOU, Y., ADVE, S., AND KUMAR, S. 2004. Performance directed energy management for main memory and disks. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*. ACM Press, New York, NY, 271–283.

LIM, S.-S., BAE, Y. H., JANG, G. T., RHEE, B.-D., MIN, S. L., PARK, C. Y., SHIN, H., PARK, K., MOON, S.-M., AND KIM, C. S. 1995. An accurate worst case timing analysis for RISC processors. *IEEE Trans. Softw. Eng. 21*, 7, 593–604.

LIU, M., WANG, W.-S., AND ORSHANSKY, M. 2004. Leakage power reduction by dual-vth designs under probabilistic analysis of vth variation. In *Proceedings of the International Symposium on Low Power Electronics and Design* ACM Press, New York, NY, 2–7.

LLOPIS, R. AND SACHDEV, M. 1996. Low power, testable dual edge triggered flip-flops. In *Proceedings of the International Symposium on Low Power Electronics and Design*. IEEE Press, 341–345.

LORCH, J. AND SMITH, A. 2001. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM Press, 50–61.

LUZ, V. D. L., KANDEMIR, M., AND KOLCU, I. 2002. Automatic data migration for reducing energy consumption in multi-bank memory systems. In *Proceedings of the 39th Conference on Design Automation*. ACM Press, 213–218.

LYUBOSLAVSKY, V., BISHOP, B., NARAYANAN, V., AND IRWIN, M. J. 2000. Design of databus charge recovery mechanism. In *Proceedings of the International Conference on ASIC*. ACM Press, 283–287.

MAGKLIS, G., SCOTT, M. L., SEMERARO, G., ALBONESI, D. H., AND DROPSHO, S. 2003. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ACM Press, 14–27.

MAGKLIS, G., SEMERARO, G., ALBONESI, D., DROPSHO, S., DWARKADAS, S., AND SCOTT, M. 2003. Dynamic frequency and voltage scaling for a multiple-clock-domain microprocessor. *IEEE Micro 23*, 6, 62–68.

MARCULESCU, D. 2004. Application adaptive energy efficient clustered architectures. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 344–349.

MARTIN, T. AND SIEWIOREK, D. 2001. Nonideal battery and main memory effects on cpu speed-setting for low power. *IEEE Tran. (VLSI) Syst. 9*, 1, 29–34.

MENG. Y., SHERWOOD, T., AND KASTNER, R. 2005. Exploring the limits of leakage power reduction in caches. *ACM Trans. Architecture Code Optimiz.*, 1, 221–246.

MOHAPATRA, S., CORNEA, R., DUTT, N., NICOLAU, A., AND VENKATASUBRAMANIAN, N. 2003. Integrated power management for video streaming to mobile handheld devices. In *Proceedings of the 11th ACM International Conference on Multimedia*. ACM Press, 582–591.

NEDOVIC, N., ALEKSIC, M., AND OKLOBDZIJA, V. 2001. Conditional techniques for low power consumption flip-flops. In *Proceedings of the 8th International Conference on Electronics, Circuits, and Systems*. 803–806.

NILSEN, K. D. AND RYGG, B. 1995. Worst-case execution time analysis on modern processors. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*. ACM Press, 20–30.

PALM, J., LEE, H., DIWAN, A., AND MOSS, J. E. B. 2002. When to use a compilation service? In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, 194–203.

PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 2000. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Design Autom. Electr. Syst. 5*, 3, 682–704.

PAPATHANASIOU, A. E. AND SCOTT, M. L. 2002. Increasing disk burstiness for energy efficiency. Technical Report 792 (November), Department of Computer Science, University of Rochester (Nov.).

PATEL, K. N. AND MARKOV, I. L. 2003. Error-correction and crosstalk avoidance in dsm busses. In *Proceedings of the International Workshop on System-Level Interconnect Prediction*. ACM Press, 9–14.

PENZES, P., NYSTROM, M., AND MARTIN, A. 2002. Transistor sizing of energy-delay-efficient circuits. Tech. Rep. 2002003, Department of Computer Science, California Institute of Technology.

PEREIRA, C., GUPTA, R., AND SRIVASTAVA, M. 2002. PASA: A software architecture for building power aware embedded systems. In *Proceedings of the IEEE CAS Workshop on Wireless Communication and Networking*.

POLLACK, F. 1999. New microarchitecture challenges in the coming generations of CMOS process technologies. *International Symposium on Microarchitecture*.

PONOMAREV, D., KUCUK, G., AND GHOSE, K. 2001. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, 90–101.

POWELL, M., YANG, S.-H., FALSAFI, B., ROY, K., AND VIJAYKUMAR, T. N. 2001. Reducing leakage in a high-performance deep-submicron instruction cache. *IEEE Trans. VLSI Syst. 9*, 1, 77–90.

RUTENBAR, R. A., CARLEY, L. R., ZAFALON, R., AND DRAGONE, N. 2001. Low-power technology mapping for mixed-swing logic. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 291–294.

SACHS, D., ADVE, S., AND JONES, D. 2003. Cross-layer adaptive video coding to reduce energy on general purpose processors. In *Proceedings of the International Conference on Image Processing*. 109–112.

SASANKA, R., HUGHES, C. J., AND ADVE, S. V. 2002. Joint local and global hardware adaptations for energy. *SIGARCH Computer Architecture News 30*, 5, 144–155.

SCHMIDT, R. AND NOTOHARDJONO, B. 2002. High-end server low-temperature cooling. *IBM J. Res. Devel. 46*, 6, 739–751.

SEMERARO, G., ALBONESI, D. H., DROPSHO, S. G., MAGKLIS, G., DWARKADAS, S., AND SCOTT, M. L. 2002. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society Press, 356–367.

SEMERARO, G., MAGKLIS, G., BALASUBRAMONIAN, R., ALBONESI, D. H., DWARKADAS, S., AND SCOTT, M. L. 2002. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 29–40.

SETA, K., HARA, H., KURODA, T., KAKUMU, M., AND SAKURAI, T. 1995. 50% active-power saving without speed degradation using standby power reduction (SPR) circuit. In *Proceedings of the IEEE International Solid-State Conference*. IEEE Press, 318–319.

SGROI, M., SHEETS, M., MIHAL, A., KEUTZER, K., MALIK, S., RABAEY, J., AND SANGIOVANNI-VENCENTELLI, A. 2001. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proceedings of the 38th Conference on Design Automation*. ACM Press, 667–672.

SHIN, D., KIM, J., AND LEE, S. 2001. Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of the 38th Conference on Design Automation*. ACM Press, 438–443.

STAN, M. AND BURLESON, W. 1995. Bus-invert coding for low-power i/o. *IEEE Trans. VLSI*, 49–58.

STANLEY-MARBELL, P., HSIAO, M., AND KREMER, U. 2002. A Hardware Architecture for Dynamic Performance and Energy Adaptation. In *Proceedings of the Workshop on Power-Aware Computer Systems*. 33–52.

STROLLO, A., NAPOLI, E., AND CARO, D. D. 2000. New clock-gating techniques for low-power flip-flops. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 114–119.

SULTANIA, A., SYLVESTER, D., AND SAPATNEKAR, S. 2004. Transistor and pin reordering for gate oxide leakage reduction in dual Tox circuits. In *IEEE International Conference on Computer Design*. 228–233.

SYLVESTER, D. AND KEUTZER, K. 1998. Getting to the bottom of deep submicron. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. ACM Press, 203–211.

TAN, T., RAGHUNATHAN, A., AND JHA, N. 2003. Software architectural transformations: A new approach to low energy embedded software. In *Design, Automation and Test in Europe*. 1046–1051.

TAYLOR, C. N., DEY, S., AND ZHAO, Y. 2001. Modeling and minimization of interconnect energy dissipation in nanometer technologies. In *Proceedings of the 38th Conference on Design Automation*. ACM Press, 754–757.

Transmeta Corporation. 2001. *LongRun Power Management: Dynamic Power Management for Crusoe Processors*. Transmeta Corporation.

Transmeta Corporation. 2003. *Crusoe Processor Product Brief: Model TM5800*. Transmeta Corporation.

TURING, A. 1937. Computability and lambda-definability. *J. Symbolic Logic 2*, 4, 153–163.

UNNIKRISHNAN, P., CHEN, G., KANDEMIR, M., AND MUDGETT, D. R. 2002. Dynamic compilation for energy adaptation. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. ACM Press, 158–163.

VENKATACHALAM, V., WANG, L., GAL, A., PROBST, C., AND FRANZ, M. 2003. Proxyvm: A network-based compilation infrastructure for resource-constrained devices. Technical Report 03-13, University of California, Irvine.

VICTOR, B. AND KEUTZER, K. 2001. Bus encoding to prevent crosstalk delay. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 57–63.

WANG, H., PEH, L.-S., AND MALIK, S. 2003. Power-driven design of router microarchitectures in on-chip networks. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'36)*. IEEE Computer Society, 105–116.

WEI, L., CHEN, Z., JOHNSON, M., ROY, K., AND DE, V. 1998. Design and optimization of low voltage high performance dual threshold CMOS circuits. In *Proceedings of the 35th Annual Conference on Design Automation*. ACM Press, 489–494.

WEISER, M., WELCH, B., DEMERS, A. J., AND SHENKER, S. 1994. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*. 13–23.

WEISSEL, A. AND BELLOSA, F. 2002. Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM Press, 238–246.

WON, H.-S., KIM, K.-S., JEONG, K.-O., PARK, K.-T., CHOI, K.-M., AND KONG, J.-T. 2003. An MTCMOS design methodology and its application to mobile computing. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*. ACM Press, 110–115.

YUAN, W. AND NAHRSTEDT, K. 2003. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 149–163.

ZENG, H., ELLIS, C. S., LEBECK, A. R., AND VAHDAT, A. 2002. Ecosystem: managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 123–132.

ZHANG, H. AND RABAEY, J. 1998. Low-swing interconnect interface circuits. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 161–166.

ZHANG, H., WAN, M., GEORGE, V., AND RABAEY, J. 2001. Interconnect architecture exploration for low-energy reconfigurable single-chip dsps. In *Proceedings of the International Symposium on Systems Synthesis*. ACM Press, 33–38.

ZHANG, W., HU, J. S., DEGALAHAL, V., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. J. 2002. Compiler-directed instruction cache leakage optimization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society Press, 208–218.

ZHANG, W., KARAKOY, M., KANDEMIR, M., AND CHEN, G. 2003. A compiler approach for reducing data cache energy. In *Proceedings of the 17th Annual International Conference on Supercomputing*. ACM Press, 76–85.

ZHAO, P., DARWSIH, T., AND BAYOUMI, M. 2004. High-performance and low-power conditional discharge flip-flop. *IEEE Trans. VLSI Syst. 12*, 5, 477–484.

Received December 2003; revised February and June 2005; accepted September 2005