Chapter 5
Power Aware Operating Systems, Compilers, and Application Software

What does a compiler have to do with power dissipation? A compiler is a piece of system software that parses a high level language, performs optimizing transformations, and finally generates code for execution on a processor. On the surface, it seems very far removed from an electrical phenomenon like power dissipation. Yet, it was not long before the two got inextricably linked. The involvement of the compiler along with the processor architecture in the design space exploration loop of application specific systems (ASIPs) might have eased the transition. In this scenario, compiler analysis can actually influence the choice of architectural parameters of the final processor. Clearly, if a low power system consisting of an application running on a processor is desired, the selected processor architecture has to work in tandem with the compiler and application programmer – an architectural feature is useless if it is not properly exploited by the code generated by a compiler or written by a programmer. Low power instruction encoding is an example optimization that features the compiler in a central role with the explicit role of reducing power. In an ASIP, the opcode decisions need not be fixed, and could be tuned to the application. Since the compiler has an intimate knowledge of the application, it could anticipate the transition patterns between consecutive instructions from the program layout and suggest an encoding of instructions that reduces switching power arising out of the fetch, transmission, and storage of sequences of instructions. Modern compiler designers are investigating the development of power awareness in a more direct way in general purpose processor systems, not just ASIPs. The role of the compiler and application programmer grows along with the concomitant provision of hooks and control mechanisms introduced by the hardware to support high level decision making on power-related issues.

The operating system has a very direct role in power management of a computer system, since it has the important responsibility of monitoring and controlling every system resource. Of course, the hardware resource itself might be designed to save power when conditions are favorable – for example, a memory device can shift to low power mode when it is inactive for a long period of time. However, system level power management can be more aggressive if the operating system plays an active role in addition to power efficiencies built into individual resources. For example, an individual resource may require an accurate prediction of future activity, in order to make good power management decisions. Since the operating system also assumes

P.R. Panda et al., *Power-efficient System Design*, DOI 10.1007/978-1-4419-6388-8_5,

the responsibility for task allocation on resources, it may have that vital dynamic information using which it can inform the resource whether any significant workload is likely to be scheduled on it in the near future. It is clear that aggressive power optimizations can take place when there is a meaningful collaboration between the operating system and the resources it manages.

In this chapter we study some recent work in the area of power aware operating systems, compilers, and application software. This continues to be an important research area and we can expect exciting new problems and solutions in the days ahead.

5.1 Operating System Optimizations

An operating system is very well placed to make intelligent run-time power management decisions because it is best suited to keep track of the dynamic variation of the status of the different resources under its supervision. Before studying the power optimization policies implemented by an OS, it is instructive to look at the component-wise break-up of the power dissipation on a typical computer. As expected, the total power dissipation, and the relative power dissipated in the individual components, vary depending on the benchmark/application domain.

Figure 5.1 gives a comparison of the total system power of an IBM Thinkpad R40 laptop with a 1.3 GHz processor, 256 MB memory, 40 GB hard drive, optical drives, wireless networking, and a 14.1" screen, when it is subject to workloads arising out of different classes of benchmark applications [30]. The idle system dissipates 13 W,

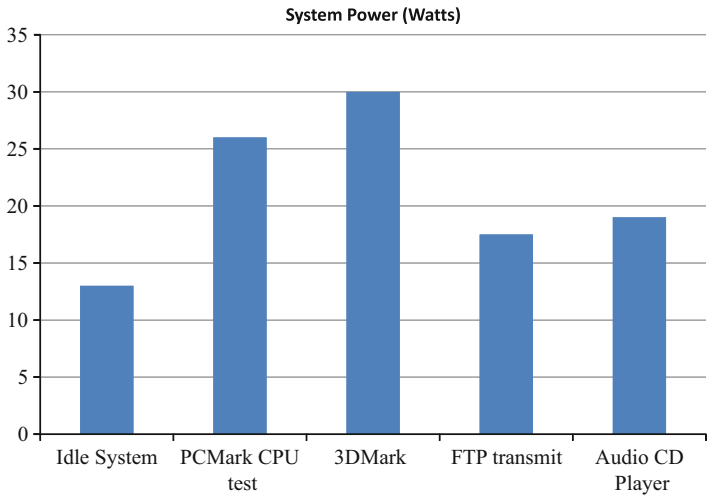


Fig. 5.1 Laptop power dissipation for different benchmarks. Idle power is 13 W. The benchmarks vary widely in their power dissipation

while the benchmarks (*3DMark* – graphics benchmark; *CPUMark* – CPU intensive workload; *Wireless FTP* – file transfer over the wireless LAN card; and *Audio CD Playback*) dissipate between 17 W and 30 W, exhibiting a wide range.

Figures 5.2, 5.3, 5.4, 5.5, and 5.6 show the component-wise break-up of the total system power dissipation for the different benchmarks and idle state. In an idle system, the LCD display consumes a relatively large fraction of the power. For the CPU-intensive PCMark suite, the CPU was, as expected, the largest consumer of

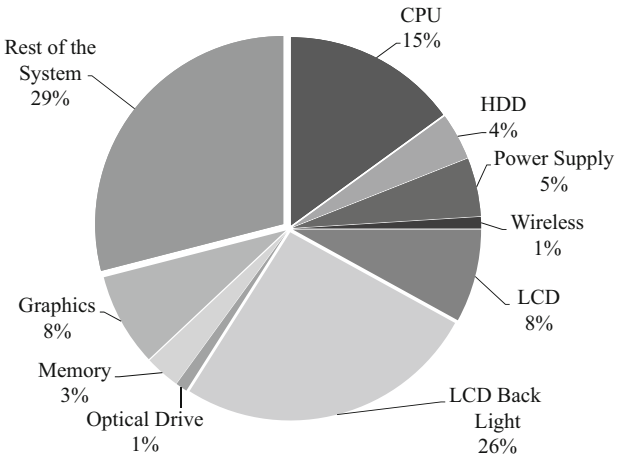


Fig. 5.2 Component-wise break-up of laptop power dissipation when the system is idle. In idle systems, the LCD display consumes a relatively large fraction (26%) of the total power

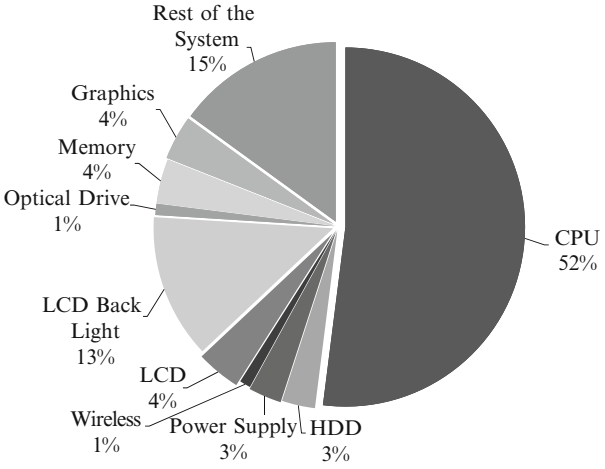


Fig. 5.3 Component-wise break-up of laptop power dissipation for CPUMark benchmark. This benchmark stresses the CPU, which consumes 52% of the total power

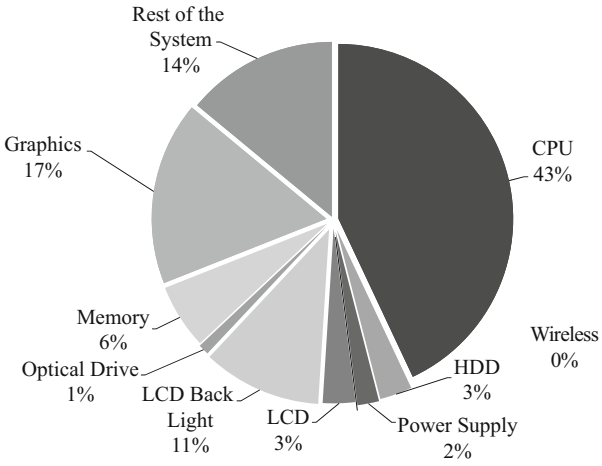


Fig. 5.4 Component-wise break-up of laptop power dissipation for 3DMark benchmark. The CPU consumes a large 43% of the system power

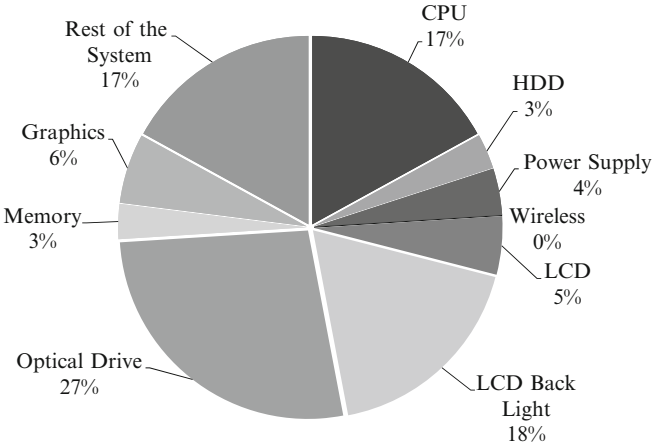


Fig. 5.5 Component-wise break-up of laptop power dissipation for Audio CD player benchmark. The optical drive was the main power consumer, accounting for 27%

power. The 3DMark suite is also CPU intensive, and the CPU power accounted for a huge 43%. FTP showed a relatively large power dissipation in the wireless card, drawing power comparable to the CPU. In the Audio CD playback, the optical drive was the main power consumer, with its power exceeding even the CPU power, because the drive was running at full speed throughout the playback period. The study shows that the power distribution among the system components depends on the type of computational and data transfer demands placed on the individual resources. The CPU is usually among the heaviest power users.

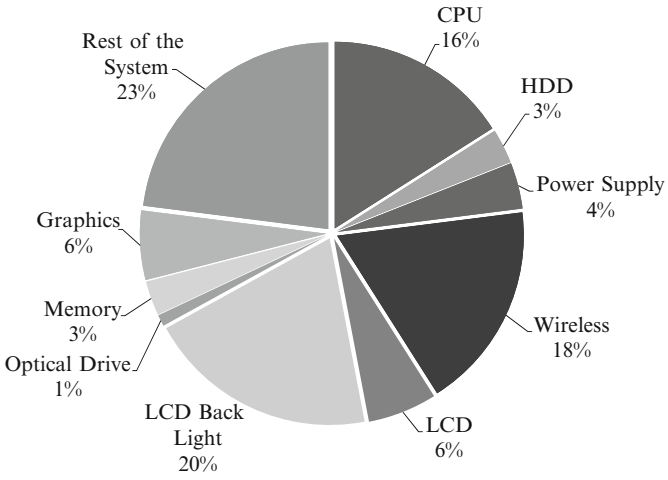


Fig. 5.6 Component-wise break-up of laptop power dissipation for wireless FTP benchmark. The wireless card dissipates 18% of the system power

Table 5.1 Hard disk drive states [30]

HDD power state	Power Consumption
Idle	0.575 W
Standby	0.173 W
Read	2.78 W
Write	2.19 W
Copy	2.29 W

Table 5.2 Power consumption variation of LCD display with brightness level [30]

LCD brightness level	Power Consumption
1	0.6 W
2	0.8 W
3	1.1 W
4	1.3 W
5	1.6 W
6	2.0 W
7	2.7 W
8	3.4 W

The significant difference in power consumption of individual components across the various benchmark applications can be attributed to the different power characteristics of the devices based on the usage pattern of the application. Tables 5.1, 5.2, 5.3, and 5.4 show the power drawn by the hard disk drive, LCD display, Wireless LAN card, and CD drive, in the different *power states* of the respective devices. The devices exhibit a significant dynamic power dissipation range, depending on the state of activity. For example, the wireless LAN card draws 22 times as much power when it is transmitting, compared to when it is idle. Similarly, there is a 7:1

Table 5.3 Wireless LAN card states [30]

Wireless LAN Card states	Power Consumption
Power saver (idle)	0.14 W
Base (idle)	1.0 W
Transmit	3.2 W at 4.2 Mb/s
Receive	2.55 W at 2.9 Mb/s

Table 5.4 Optical drive states [30]

Optical drive state	Power Consumption
Initial spin up	3.34 W
Steady spin	2.78 W
Reading data	5.31 W

power dissipation ratio between the highest and lowest brightness levels of the LCD display, and a 16:1 ratio between active and idle power for the hard disk.

5.1.1 Advanced Configuration and Power Interface (ACPI)

In a typical computing system with several input/output devices, processing units, memory devices, etc., working in unison, it is very unlikely that all of them would be active for the entire duration of system operation. For example, a modem may be active only when applications running on the system request a network access, and is inactive for the rest of the time. It is observed that a significant amount of power is dissipated during these spells of inactivity owing to the following reasons:

- With technology scaling, leakage power has become a significant portion of the total power consumption of an electronic circuit. For example, 40% of the 110W of power consumed by a 90nm Pentium 4 is actually due to the leakage current [35].
- Devices such as display, waste power in doing redundant work. Displays are designed such that the contents of the framebuffer are refreshed periodically on to the display device. Even when the system is idle and the contents of framebuffer are not expected to change, the display is periodically refreshed, resulting in waste of power.

It would appear that an ideal solution to counter leakage power would be to activate the device only when it is working. But this is not always feasible, since the switching time from *on* to *off* and vice-versa could affect system performance. Hence the devices are generally designed to work at different operating points called the *power modes* that represent trade-offs between performance and power. Depending on the requirements of the applications running on the system, the mode of device operation is dynamically selected and modified. The policies that govern the switching of operating point of a device are called the power management policies. Power management of devices could be implemented in two ways:

- in the firmware of the device and controlled by the driver of the device; and
- in the operating system.

Operating System directed Power Management (OSPM) is becoming popular in modern systems due to the following advantages over device level implementations.

- Implementation of power management in the OS makes it platform independent.
- The limitation of implementing complex power management strategies in the BIOS of the devices can be overcome.
- Algorithms common to power management of several devices can be implemented only once, thus decreasing the development cost.

Now that the power management policies are implemented in the OS, standard interfaces between the OS and device drivers are necessary for smooth operation. Advanced Configuration and Power Interface (ACPI) is the specification of a common standard for OS controlled device configuration and power management [16]. This standard was initially developed by Intel, Microsoft, and Toshiba, with Hewlett-Packard and Phoenix being involved in the later evolution.

Let us examine the ACPI standard in some detail. Figure 5.7 shows the architecture of a system using ACPI for power management. The operating system communicates with the ACPI stack through software and drivers. The ACPI layer acts as an interface between the OS and the device with the help of three main components: (i) ACPI tables; (ii) ACPI bios; and (III) ACPI registers.

ACPI tables contain definition blocks that describe the ACPI managed devices. The definition includes data and machine independent byte-code that performs device configuration and management.

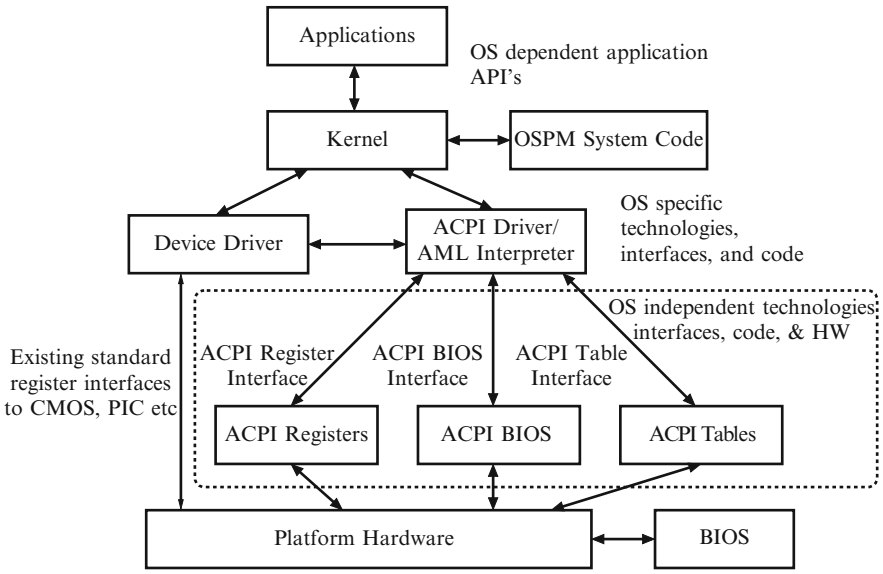


Fig. 5.7 ACPI system architecture

ACPI BIOS is responsible for low-level management operations of the device. It contains code to assist in booting the system and switch the operation mode of the device. Different power modes of a system are described in the following section.

ACPI registers are a set of hardware registers that help in configuration and management of the device. These registers are accessed through the byte-code stored in the device-specific part of the ACPI tables.

In an ACPI based system, on power-up, the ACPI BIOS is loaded prior to the OS and the ACPI tables are loaded into memory. Since the memory requirement of these tables is much more than what a BIOS memory could accommodate, the ACPI BIOS allocates space in the physical memory of the system. When the ACPI-aware OS kernel starts its operation, it searches the BIOS memory area to obtain the address of the ACPI tables in the physical memory. All ACPI operations, excluding a few BIOS functions, are performed in the OS by interpreting the machine-independent ACPI Machine Language (AML) byte-code present in the ACPI tables.

5.1.1.1 Power Modes

ACPI defines various power states in which the entire system and also individual devices in the system could be operating. These states are enumerated in Tables 5.5, 5.6, 5.7, and 5.8.

The *global states* apply to system as a whole, and are visible to the user. The G0 state ("working") is the normal active state in which user threads are executed.

Table 5.5 Global States that define the power mode of the entire system

State	Description
G0	*Working*
G1	*Sleeping* (subdivided into states S1 to S4) • S1 – All CPU caches are flushed and CPU(s) stop executing instructions. Full context is maintained in RAM. Power to RAM and CPU(s) is maintained. • S2 – CPU is powered off. Processor context and cache context are not maintained but RAM is maintained. • S3 – Only RAM remains powered, hence all system context is lost. (commonly referred as *Standby* or *Sleep*) • S4 – Data in RAM is flushed to hard disk and the system is powered down. (referred to as *Hibernation*) Reboot is required to wake up the system.
G2(S5)	*Soft off* In this state all the devices are also powered down along with CPU and caches. Some components in the system such as mouse and keyboard remain powered to wake up the system.
G3	*Mechanical off* The system is switched off except for the real time clock in the system that is powered by a small battery.

Table 5.6 Device Power States

State	Description
D0	Fully on.
D1,D2	Intermediate device dependent power states
D3	Powered off

Table 5.7 Processor Power States

State	Description
C0	*Fully on*.
C1	*Halt*. Processor does not execute any instructions, but can instantaneously return to execution
C2	*Stop-Clock*. Maintains application-visible state, but takes longer for wake-up.
C3	*Sleep*. Processor does not keep its cache, but maintains other state.

Table 5.8 Processor Performance States

State	Description
P0	Maximum power and frequency
P1	Less than P0, voltage and frequency scaled
Pn	Less than P(n-1), voltage and frequency scaled

System power consumption in this state is generally the highest. In the G1 state ("sleeping"), user threads are not executed and key components such as display are turned off to save power. However, the system can be moved to active state in a relatively small amount of time. In the G2 state ("soft off"), the system is mostly off, but some components remain "on" so that the system can "wake up" through signals received from an I/O device such as keyboard or mouse. Power consumed in this state is very low. In G3 state ("mechanical off") the system is turned off completely and draws near zero power, except for a clock powered by a small battery.

The global sleeping state is subdivided into several further levels S1 through S4, representing a finer grain power management. In the S1 state, the CPU caches are flushed and the CPU does not execute instructions, but can be moved to an active execution quickly. In the S2 state, the CPU is powered off, so the processor and cache contexts are lost, but the RAM is maintained. In the S3 state (commonly known as "standby" or "sleep"), the rest of the chip set is turned off, but the RAM is still valid. In the S4 state (commonly known as "hibernate"), the memory data is saved into the hard disk and the system is powered down. A reboot is required to wake the system up. The S5 state coincides with the G2 state.

The *device power states* D0 through D3 in ACPI apply to I/O devices connected to the system bus. These states are defined in a very generic manner, and some devices may not feature all the four states. In the D0 state, the device is "fully on" and consumes maximum power. The D1 and D2 states are low power states that are device dependent. In D3, the device is turned off and consumes no power. Re-initialization is necessary when the OS turns the device back3 on. Examples of device power states and the power consumed in each state are given in Tables 5.1, 5.2, 5.3, and 5.4, for the laptop experiment discussed above.

The *Processor power states* C0 through C3 represent various performance-power trade-offs in the processor in the global state G0 ("working"). In the C0 state ("fully on"), the processor executes normal instructions and consumes the highest power. In the C1 state ("halt"), the processor does not execute any instruction, but can immediately return to execution. In the C2 state ("stop clock"), the processor moves to a low power state, does not execute instructions, and takes longer to return to execution. The C3 state ("sleep") offers further improvements in power savings, with caches maintaining state but disallowing snooping.

Finally, the standard also defines the device and processor performance states within the respective "fully on" states D0 and C0. State P0 represents the maximum frequency for the CPU, which translates to maximum power consumption. Other states P1, P2, P3, etc., are defined, with decreasing power and associated performance. Dynamic voltage and frequency scaling (Section 5.1.2) is typically employed in the processors and devices to achieve the different power states.

5.1.2 Dynamic Voltage and Frequency Scaling

The basic dynamic power equation $P = CV^2Af$, where C is the load capacitance, V is the operating voltage, A is aggregate activity factor, and f is the operating frequency, shows the significant leverage possible by adjusting the voltage and frequency. It shows that if we can reduce voltage by some small factor, we can reduce power by the square of that factor. Furthermore, reducing supply voltage often slows transistors such that reducing the clock frequency is also required. The benefit of this is that within a given system, scaling supply voltage down now offers the potential of a cubic reduction in power dissipation. This process of reducing both the voltage and frequency of the processor dynamically (at run time) is called Dynamic Voltage and Frequency Scaling (DVFS). It is important to note here that while DVFS may reduce the instantaneous power cubically, the reduction on the total energy dissipated is quadratic. This is because, even though programs run at lower power dissipation levels, they run for longer durations.

Fundamentally, DVFS approaches exploit slack to achieve more power-efficient execution. The workload profile of applications is far from a constant; in fact, it may be highly dynamic. As a result, the processor need not be operating at the maximum performance (maximum voltage and frequency) all the time. There may be several opportunities to temporarily slow down the processor without any noticeable or adverse effects. For example, a CPU might normally respond to a user's command by running at full speed for 0.001 seconds, then waiting idle; running at one-tenth the speed, the CPU could complete the same task in 0.01 seconds, thereby saving power and energy without generating noticeable delay. Eventually DVFS is an approach that attempts to meet the seemingly conflicting goals of a responsive and intelligent device while maximizing battery life.

One of the most important decisions in implementing DVFS is the granularity at which to perform DVFS. The finest granularity at which DVFS is limited by the time it takes to switch the voltage and frequency of the processor. DVFS implementation requires a voltage regulator that is fundamentally different from a standard voltage regulator because it must also change the operating voltage for a new clock frequency [4, 7, 9]. This and other considerations result in high transition overhead for DVFS. This overhead is typically in the range of tens of micro seconds. In particular, the Intel XScale processor has a frequency switching time of $20\mu s$ [10, 17, 18]. To be able to profitably apply DVFS, and hide the penalty of voltage regulation, the granularity at which voltage and frequency are scaled should be at least 2 to 3 orders of magnitude higher, which is in the range of milliseconds. This falls more or less in the domain of operating system scheduling granularity. Consequently, most DVFS schemes have been incorporated into the OS scheduler and task management services.

The simplest application of DVFS algorithms is a history-based scheme, where we monitor the recent history to make a prediction about the immediately future. The *Past* algorithm is a simple strategy that divides time into *intervals* [45]. In each interval, the algorithm keeps track of what the CPU utilization was, and predicts that the utilization will remain unchanged in the next interval. This assumption is in keeping with system behavior in general – drastic changes in system load are relatively rare. The utilization is compared against a pre-defined *threshold*. If the utilization is below this threshold, then the system is slowed down by lowering the voltage. If the utilization is above the threshold, then the system is sped up by selecting a higher voltage. The strategy is illustrated in Fig. 5.8, with the threshold set at 80% utilization. In Fig. 5.8(a), a 70% utilization is observed in time interval t. The Past algorithm predicts a 70% utilization for interval $t + 1$, and slows down the system by stepping down the voltage. Similarly, in Fig. 5.8(b), a 90% utilization causes Past to step up the voltage. In order to prevent switching of voltages too frequently, the threshold can instead be defined as a range of utilizations, for example, between 75-85% in our example.

The Past algorithm is very simple, and suffers from some obvious difficulties as it relies on only one data point. In the Aged Averages (AVG) algorithm, a weighted average of the utilizations in the last few intervals is used as the prediction for the next interval [14]. Using more than one interval makes the algorithm more robust against transient changes in load. This is illustrated in Fig. 5.9. Here, the utilizations at intervals t and $t - 1$ are averaged with equal weights to generate the predicted utilization for interval $t + 1$. In Fig. 5.9(a), the utilizations for intervals t and $t - 1$ are 70% and 80% respectively, giving the predicted utilization for interval $t + 1$ to be 75%. The voltage is stepped down. In Fig. 5.9(b), intervals t and $t - 1$ have utilizations 90% and 70% respectively, giving 80% as the prediction for interval $t + 1$. This leads to no change in voltage levels, treating the 90% value as a transient when it appears for the first time. If the rate is sustained (for another interval in this case), then the aged average reflects the higher load and the voltage is eventually stepped up.

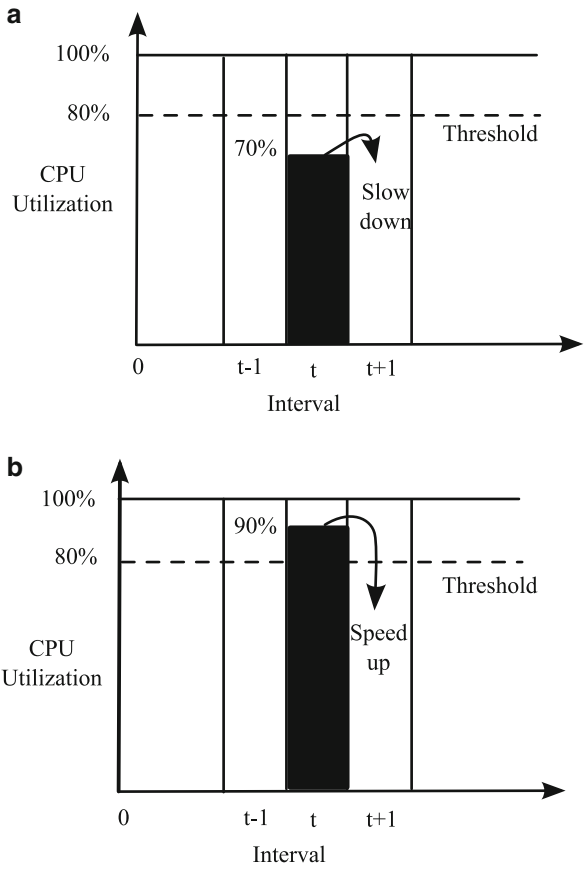


Fig. 5.8 *Past* Algorithm: predict the utilization in the next interval to equal that of the last interval (**a**) 70% utilization predicted for interval t+1. Slow down. (**b**) 90% utilization predicted for interval t+1. Speed up

To evaluate the effectiveness of the above algorithms, one can compare them with an *Oracle* algorithm that has advance knowledge of the next interval's load. Schemes such as AVG lie somewhere between the effectiveness of Past and Oracle, with the increasing effectiveness coming with the associated cost penalty of larger storage, which can be an issue in a hardware implementation. Variations of this strategy can be thought of in slightly different contexts, particularly ones involving the choice between different *power modes*: *active*, *sleep*, and *power down*. In general, we would like to move the system to power down mode upon encountering long idle periods so as to save power, but the associated penalty is that it takes a relatively large number of cycles to bring the system back to active mode. Being over-aggressive in powering down the system means high performance overheads incurred in waiting for the system to be usable again.

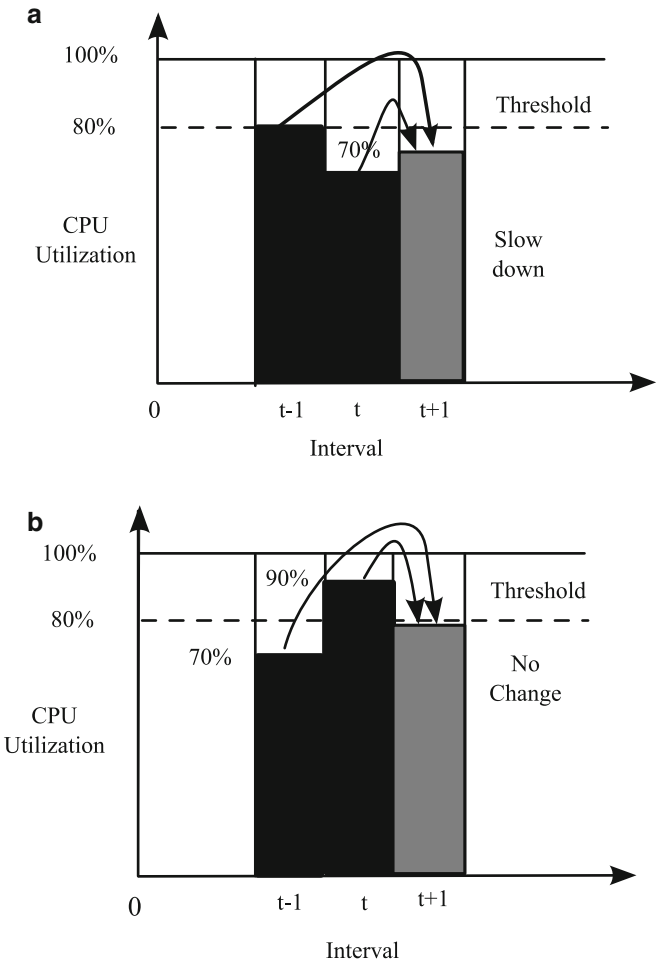


Fig. 5.9 *AVG* Algorithm: predict the utilization in the next interval to be weighted average of a few previous intervals (**a**) (70 + 80)/2 = 75% utilization predicted for interval t+1. Slow down. (**b**) (70 + 90)/2 = 80% utilization predicted for interval t+1. No change

5.1.2.1 DVFS in Real-time OS

Essentially, DVFS schemes use a simple feedback mechanism, such as detecting the amount of idle time on the processor over a period of time, and then adjust the frequency and voltage to just handle the computational load. This strategy has a simple implementation and follows the load characteristics closely, but cannot provide any timeliness guarantees and tasks may miss their execution deadlines. As an example, in an embedded camcorder controller, suppose there is a program that must react to a change in a sensor reading within a 5 ms deadline, and that it requires up to 3 ms of computation time with the processor running at the maximum

operating frequency. With a DVS algorithm that reacts only to average throughput, if the total load on the system is low, the processor would be set to operate at a low frequency, say half of the maximum, and the task, now requiring 6 ms of processor time, cannot meet its 5 ms deadline. To provide real-time guarantees, DVS must consider deadlines and periodicity of real-time tasks, requiring integration with the real-time scheduler.

Let us look at some idealized situations in *real-time systems* to understand the limits of the applicability of DVFS.

First, let us consider a situation where we have tasks $T_1, \ldots, T_n$ to be scheduled in the time interval $[0, M]$. Each task has associated with it the number of required processor cycles R_i, the arrival time $A_i \in [0, M]$, and deadline for completion $D_i \in [0, M]$. The *voltage scheduling* problem is to find the optimal speeds at which the processor should work at every time instant in $[0, M]$ so that the total energy is minimized. It is assumed that the processor speed, and consequently the voltage, can be varied continuously, and can take all real values. The R_i values are fixed constants.

An optimal voltage scheduling algorithm uses the following greedy strategy [46]. For every time interval $I = [t_1, t_2]$ in the range $[0, T]$, find the *intensity* $g(I)$ defined as:

$$g(I) = \frac{\sum_i R_i}{t_2 - t_1} \tag{5.1}$$

for all i such that $[A_i, D_i] \in [t_1, t_2]$, that is, the intensity for interval I is computed considering all tasks whose arrival and completion times lie within the interval. Since $\sum_i R_i$ represents the total work that needs to be completed in time interval I, $g(I)$ represents the minimum required average speed of the processor during time interval I. Thus, if the processor is run at speed $g(I)$ during time interval I, it will be energy-optimal for this interval (if the speed is lower, then the tasks cannot complete; if the speed is higher, then the voltage – and hence energy – must be higher). We have established the speed/voltage values for interval $I = [t_1, t_2]$. Now, we just delete the interval from consideration, and recursively solve the same problem for the smaller interval thus obtained. The arrival and completion times of the remaining tasks are adjusted to reflect the deleted interval. This strategy gives the optimal speed/voltage assignment for minimizing energy [46].

The algorithm is illustrated in Fig. 5.10, with 3 tasks T1, T2, and T3, with the arrival times and deadlines being [0,5], [2,15], and [2,25] respectively, and number of cycles R_1, R_2, and R_3 being 1, 4, and 2 respectively. The intensities for the intervals are as indicated in Fig. 5.10(a). For example, the interval $I = [0, 15]$ has two tasks T1 and T2 with arrival/completion times lying within the interval, so $g(I) = \frac{R_1+R_2}{15-0} = (1 + 4)/15 = 0.33$. Intervals not included in the list are those that cannot accommodate a single task. We select [0,15] for speed/voltage assignment first since this interval has the highest intensity. The assigned speed is 0.33. We then delete this interval, leading to a smaller problem indicated in Fig. 5.10(b). Only T3 still remains to be executed, and the arrival/completion times are as indicated in Fig. 5.10(b). Only one interval exists with intensity $\frac{R_3}{10-0} = 2/10 = 0.2$, and it

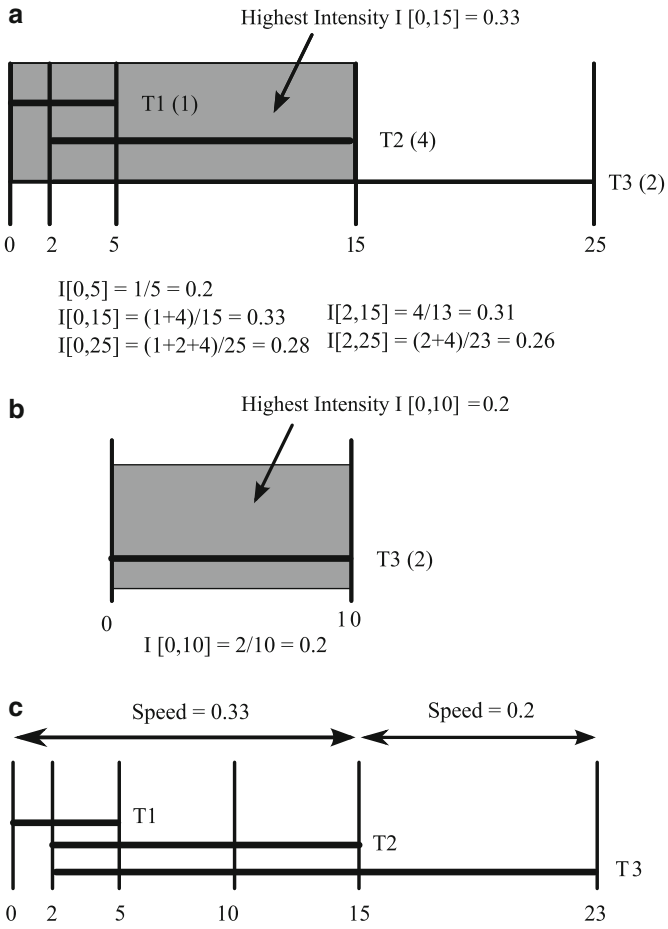


Fig. 5.10 Optimal voltage scheduling. (**a**) Interval [0,15] has the highest intensity, so it is selected first, and speed 0.33 is assigned to it. The interval is then deleted. (**b**) Interval [0,10] (corresponding to the original interval [15,25]) is selected next, with speed 0.2. (**c**) The optimal voltage schedule corresponds to the two speed settings inferred in (**a**) and (**b**)

is trivially assigned speed 0.2. The speed assignment for the complete duration is summarized in Fig. 5.10(c). The processor runs at speed 0.33 for the first 15 time units, and 0.2 for the next 10 units.

The above problem formulation assumed that it is possible to change a processor's voltage and speed to *any* desirable value. In practice, we typically have to select from a set of discrete voltage settings for a processor. Let us address the problem of selecting the optimal voltages for running a processor, given a fixed load and a time constraint [19].

Our first observation, illustrated in Fig. 5.11, is that it is always sub-optimal to complete earlier than the specified deadline. Figure 5.11(a) shows two schedules,

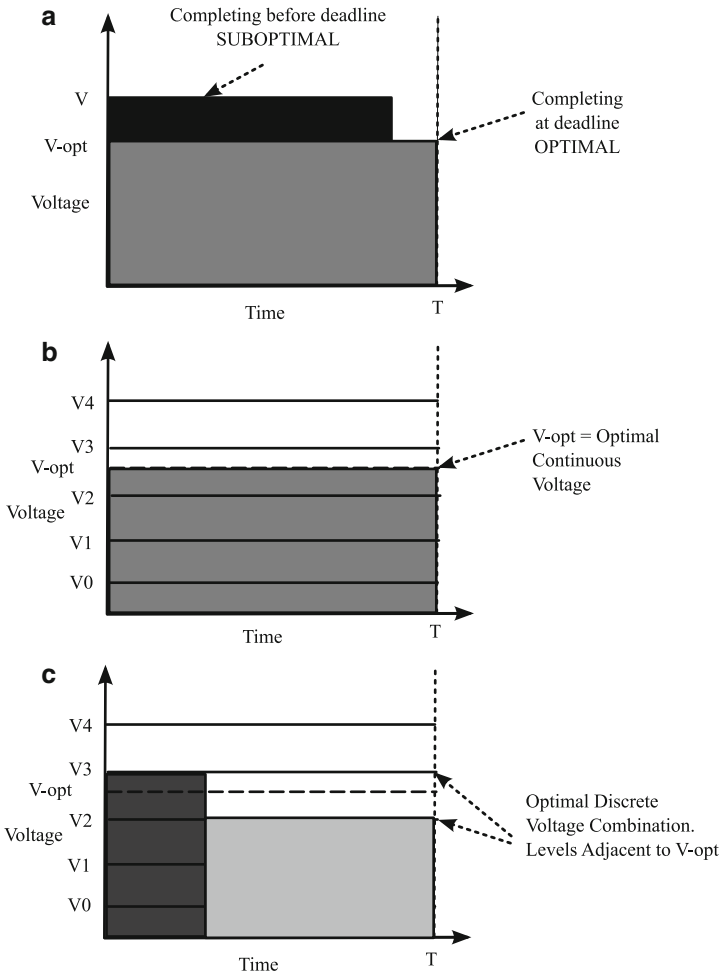


Fig. 5.11 Optimal discrete voltage scheduling with deadline T. (**a**) Energy is minimum when we select a voltage (V-opt) that allows us to complete the task exactly at the deadline. Any other voltage V leading to earlier completion is sub-optimal. (**b**) Discrete voltages available are: V0, V1, V2, V3, and V4. V-opt is not one of the available choices. (**c**) V2 and V3 are the two discrete voltage levels adjacent to V-opt. Energy is optimal when we use a combination of V2 and V3. Using any other voltage is sub-optimal

one completing before the deadline, followed by an idle period (during which the system could be assumed to dissipate zero power) with the voltage set at V; and the other completing at the deadline T. The task completing at T can progress at a lower voltage, which also increases the latency. However, the latency decreases linearly with the voltage whereas energy decreases as square of voltage. Hence, total energy is lower for the schedule completing at the deadline. In other words, average power (total latency divided by latency) is minimized for the task that utilizes all

the time available. Let the corresponding voltage be V_{opt}. This result also follows from the application of the optimal algorithm discussed earlier. However, in reality, the voltage cannot be continuously varied, and we have to select from a set of discrete choices. The situation is illustrated in Fig. 5.11(b), where the permissible discrete voltages are: V_0, V_1, V_2, V_3, and V_4. We notice that V_{opt} is not one of the available voltages, so the optimal voltage/speed setting algorithm cannot be directly applied. It can be proved that in the discrete voltage scenario the optimal voltage for the processor will be a combination of the two discrete voltages adjacent to the computed optimal voltage V_{opt} [19]. From Fig. 5.11(b), we notice that V_{opt} lies between V_2 and V_3. As shown in Fig. 5.11(c), the energy-optimal solution is to run the system at voltage V_2 for some time, and at V_3 for the remaining time. The exact durations can be easily computed. Naturally, the resulting energy will be larger than the energy of running it at the hypothetical voltage V_{opt}, but the solution is still the best possible in the discrete voltage scenario. There is no need to consider other voltages. This is true even when voltage transitions are not immediate, as assumed in this discussion, but require a fixed duration [25].

The above conceptual treatment of the real-time DVFS problem made certain idealizations and simplifications that we need to be aware of, and also, exploit appropriately in a practical aggressive DVFS strategy. First of all, the number of cycles or any other measure of *work done* in a task may not be easily computed. This may be data dependent. Worst case execution times (WCET) need to be used. Of course, there may be many situations in which the worst case execution path is not exercised. Further, the presence of a memory hierarchy makes the WCET computation very difficult, and a theoretically guaranteed WCET that takes multiple levels of cache and secondary memory in its computation may be too pessimistic to be useful for DVFS.

Secondly, power of a large processor based system is not necessarily a quadratic function of the supply voltage, and latency is not necessarily inversely proportional to the supply voltage. These results hold for a single transistor, but there are other effects in a complex circuit such as that due to memory and I/O. Since DVFS modifies the delays of only the processor and does not touch components such as memory and I/O, the latter might actually consume more energy because they are held in active state for longer. Thus, practical DVFS strategies are more based on empirical models and prediction than theoretical analysis [44].

An example DVS-based optimization is shown in Fig. 5.12. A and B represent two code sequences following a memory load. A does not depend on the result of

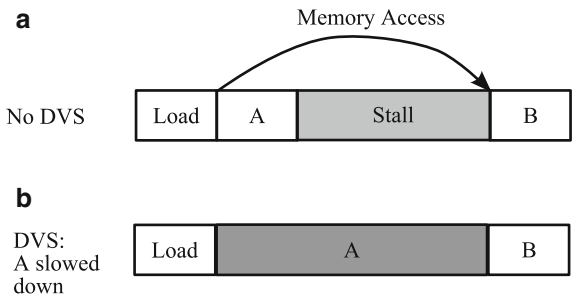


Fig. 5.12 DVS on stall due to memory access. (**a**) Lengthy stall due to load. A is not dependent on the load, so can proceed during the stall, but B cannot begin until the stall is resolved. (**b**) Anticipating the stall, we can slow down A, saving power

the load, but B does depend on it. In the situation shown in Fig. 5.12(a), A completes before the memory system has responded to the load request, leading to a system stall until the data is obtained, at which time B resumes execution. A possible resolution of this is shown in Fig. 5.12(b), where the system is aware of the expected latencies of A and the memory access. A can be slowed down by DVFS strategies to extend its execution time to be close to when the memory access is expected to complete. B resumes at its regular time, but the solution is more energy-optimal because A was executed at a lower voltage [44]. This optimization relies on reasonable estimates of the execution latencies being available, and the DVFS mechanism being able to respond with voltage/frequency switches fast enough to be useful.

One additional factor to be considered during DVFS is the accounting for leakage power. When the duration of a task is extended due to the voltage being scaled down, the dynamic energy decreases, but the leakage energy increases because the system continues to leak energy for the entire duration that it is active. Below a certain voltage/speed, the total energy actually increases [20]. The situation is shown in Fig. 5.13. The total system energy E consists of three components: (i) dynamic energy (E_d); (ii) leakage energy (E_l); and (iii) the intrinsic energy (E_{on}) that is necessary just to keep the system running. E_{on} consists of the power dissipated by system components such as analog blocks (phase locked loop and I/O) that are necessary for proper system operation.

$$E = E_d + E_l + E_{on}$$

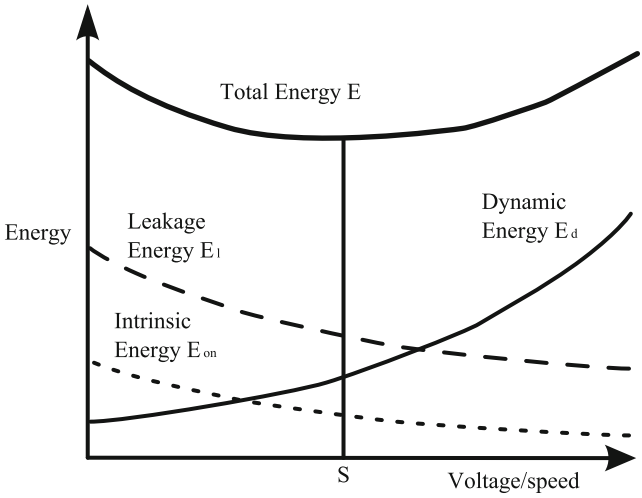


Fig. 5.13 Total system energy increases below critical voltage/speed S. At higher speed and voltage, dynamic energy dominates. At very low speed and voltage, the delay increases, so the system remains on for longer, leading to higher leakage energy

For a given time duration available for running a task, both E_l and E_{on} increase with decreasing voltage/speed; on the other hand, E_d decreases with decreasing voltage. The graph follows a 'U' shape, demonstrating a specific system-determined voltage/speed S that can be considered critical; below this level, the total energy starts increasing.

The DVFS concept is useful in the context of real-time systems where deadlines are imposed on tasks. Let us understand a straightforward application of *intra-task* DVFS – processor speeds/voltages are varied within an application so as to minimize energy. As seen earlier, the energy-optimal choice of voltage/speed is the one that causes the task to finish exactly at the deadline. However, different paths of a program will require different latencies depending on the amount of computation in the different branches, and also, as discussed earlier, the input data. A program is characterized by its worst case execution time (WCET), which, though hard to compute in general, could be obtained from user inputs on loop iteration counts, etc. Since the program path leading to the WCET must be executed within the deadline, the processor speed/voltage could be chosen such that this path completes exactly on the deadline.

Figure 5.14(a) shows a control flow graph (CFG) with each node corresponding to a basic block of code, and annotated by the delay in number of cycles required to execute it. Figure 5.14(b) shows a voltage/speed selection such that the worst case execution path A-B-D completes by the 80s deadline. However, there will be situations where this path is not executed, and the A-C-D path is taken. If the system is executed with the same voltage as in the A-B-D path, then the execution finishes by 40s, as shown in Fig. 5.14(c), and the system is idle for the remaining 40s. Instead, DVFS can be applied as soon as C starts executing, since the discrepancy between the remaining worst case execution time (10s for C + 20s for D = 30s) at the current voltage/speed, and the permitted time (70s) is known here. We can thus run C and D at a lower voltage/speed, thereby saving on the total energy, and yet meeting the task deadline Fig. 5.14(d). Although the actual decision is taken at run time, appropriate voltage scaling instructions can be inserted by the compiler in the C-branch [37].

In a *periodic* real-time system, we have a set of tasks to be executed periodically. Each task, T_i, has an associated period, P_i, and a worst-case computation time, C_i. Each task needs to complete its execution by its deadline, typically defined as the end of the period, i.e., by the next release of the task. The scheduling problem in this context is to assign the actual start times for all the tasks in such a way that all tasks meet their deadlines. Two important classical scheduling algorithms are noteworthy:

- **Earliest Deadline First (EDF).** In this strategy, we give the highest scheduling priority to the task that is constrained to complete the earliest.
- **Rate Monotonic Scheduling (RMS).** In this strategy, we give the highest scheduling priority to the task with the shortest duration.

While the EDF algorithm gives optimal results in terms of finding a valid schedule, the RMS is generally considered more practical for implementation.

The real time scheduling algorithms have to be appropriately adapted in order to accommodate DVFS possibilities. In addition to the traditional *schedulability*

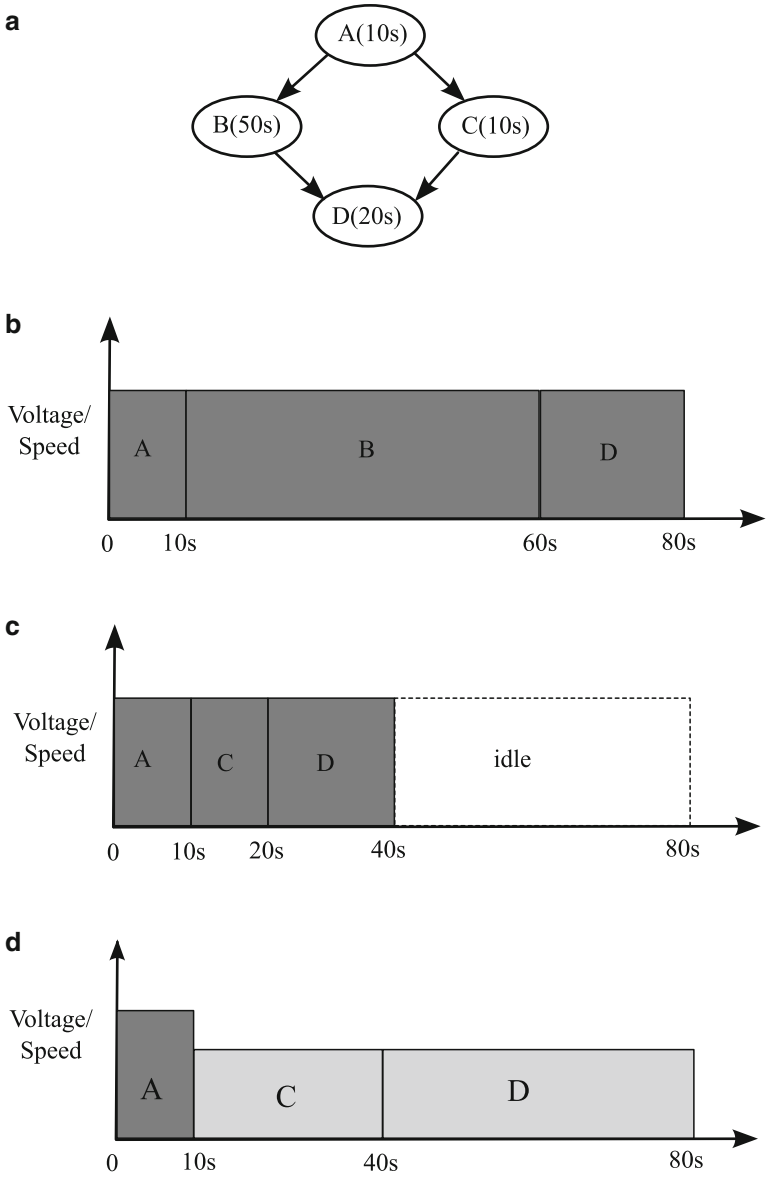


Fig. 5.14 Intra-task DVFS by lower voltage operation on slower path. (**a**) Control Flow Graph with node execution times. (**b**) Path A-B-D requires WCET = 80. (**c**) Path A-C-D completes faster. The system remains idle for 40s. No DVFS. (**d**) DVFS on the A-C-D path leads to slower execution of C and D, leading to energy saving

metric, an additional optimization criterion is total energy dissipation. For example, the necessary and sufficient schedulability test for a task set under ideal EDF scheduling requires that the sum of the worst-case utilizations be less than one, i.e.,

$$C_1/P_1 \ + \ ... \ + C_n/P_n \leq 1$$

When we apply DVFS, the operating frequency can be scaled by a factor $\alpha (0 < \alpha < 1)$, which in turn implies the worst case computation time of a task is scaled by a factor $1/\alpha$. The EDF schedulability test with frequency scaling factor α will then be:

$$C_1/P_1 \ + ... \ + C_n/P_n \leq \alpha$$

Operating frequency can then be selected as the least frequency at which the schedulability criterion is satisfied. The minimum voltage that will allow the system to operate at the required frequency is then chosen as a consequence. As shown in Fig. 5.15, this solution finds one constant operating point; the frequency and voltage do not change with time.

If each task T_i actually requires its worst-case time C_i to execute, then this result is optimal. However, in reality a task may often finish much faster than its worst

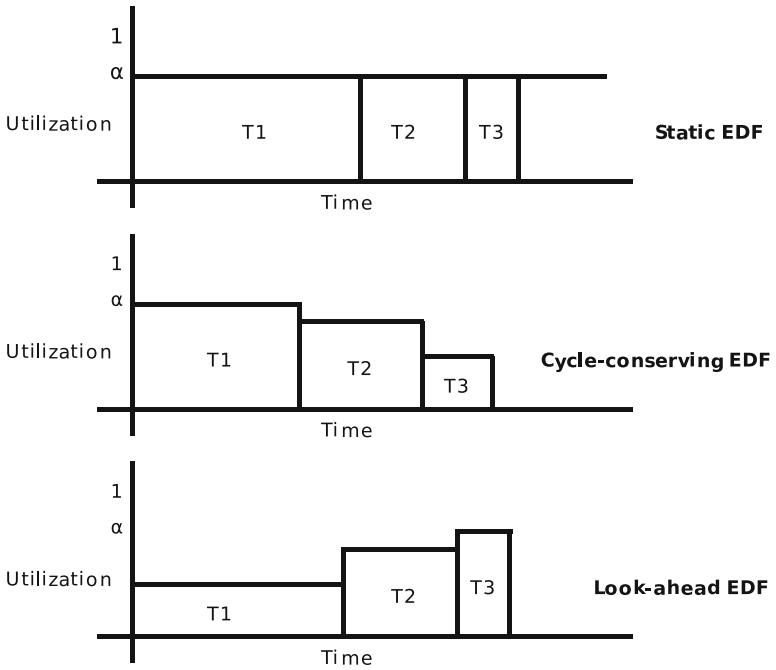


Fig. 5.15 Static EDF finds one fixed operating point at which the system must be executed to minimize power. Cycle-conserving and look-ahead schemes can change the operating point after each task. Look-ahead technique is able to exploit the difference between the actual execution time and the worst case execution time of the task

case time. Thus if c_i is the actual execution time of task T_i, and $c_i < C_i$, then this extra slack may be utilized to further slow down the system and save power. If c_i is used to compute the utilization instead of C_i, then the operating frequency obtained will result in a task set that is schedulable, at least until T_i's deadline. This is because the number of processing cycles available to other tasks between T_i's release and deadline is unchanged from the case where WCET is assumed and is actually needed. When $c_i \ll C_i$, this scheme, called *Cycle-conserving EDF* can exploit the excess slack to optimize power. Cycle-conserving EDF assumes the worst case initially, executes at a high frequency until some tasks complete, and only then reduce operating frequency and voltage. In contrast, the a look-ahead approach could defer as much work as possible, and set the operating frequency to meet the minimum work that must be done now to ensure all future deadlines are met. While this implies that high operating frequencies may be needed later on, but again, if $c_i \ll C_i$, this scheme will be advantageous. Experimental results show that the look-ahead approach is the most effective, but both the approaches can significantly reduce the energy consumption by exploiting the slack between the C_i, and c_i of the tasks.

5.1.3 I/O Device Power Management

I/O devices on computer systems such as disks and monitors dissipate a significant amount of power, and modern operating systems support various features for power management of these devices. The simplest strategy here is to monitor the activity pattern on these devices, and when the idle duration exceeds a certain threshold, move them into a low power state. For example, a laptop computer usually offers a user-configurable setting for the idle duration after which the LCD display is turned off; this saves a considerable amount of power. More sophisticated techniques in this line include activating a laptop camera periodically to monitor the surroundings; if a human face is not detected, the display could be turned off.

Simple power management mechanisms are also applicable to the hard disk. Since hard disks consume significantly lower power when in sleep mode, an idle-duration based decision to move the hard disk to sleep state is appropriate. Naturally, prediction mechanisms come handy for making the critical decision of how long we should wait before spinning a disk down. Doing it late implies wasted power but better performance. Spinning down early leads to poor performance if too many restarts are necessary, along with higher power because restart may be expensive in terms of power. Moreover, disks need not be completely spun down. Disk power management can also be performed at a finer level, where we can gradually vary the rotation speed of the disks [15].

Certain non-trivial implications of disk power management decisions ought to be kept in mind. Moving disks to sleep state implies that dirty buffers are written less often to the disk – once in several minutes instead of a few seconds – to enable the disk to stay longer in low power mode. This increases the possibility of data loss

due to power outages, where the disk does not get an opportunity to synchronize with the modified buffers. Further, frequent spin up and spin down of disks causes reliability problems and may lead to early failures. Overall, secondary storage in computer systems is a rapidly evolving area; hard disks face stiff competition from other technologies such as non-volatile memory as the latter has advantages with respect to power, weight, and noise, and is getting close with respect to cost and density.

5.2 Compiler Optimizations

Compiler optimizations targeting high performance generally also reduce average power and energy indirectly. When the optimized code generated by a compiler results in lesser number of instructions executed, it also means a smaller number of accesses to instruction memory. Since energy consumed by memory is proportional to the number of accesses, this also reduces the total energy consumed. Along the same lines, optimizations that reduce the number of accesses to data memory also reduce the total memory energy consumption. Thus, for example, all register allocation related optimizations, which increase the efficiency of register usage, are also favorable with respect to power, as it is more power-efficient to access registers than memory. This argument also generalizes to other levels of the memory hierarchy. Performance optimizations that increase the hit ratio to the L1 cache are also indirectly energy optimizations, since the L1 cache access dissipates lesser energy than an L2 cache access. The extent of performance improvement due to a compiler optimization may be different from the extent of power improvement. However, the optimizations are generally in the same direction, and if a memory related optimization improves performance, then it also reduces power and energy. However, interesting exceptions do exist – good examples being those that rely on speculative memory loads. In such cases, the access latency may be hidden by other CPU activity, but the associated energy dissipated cannot be undone. Such an optimization improves performance, but reduces energy efficiency.

Making the compiler explicitly aware of the performance/energy optimized features present in the memory subsystem increases the compilation time, but yields the power benefits without any run-time overhead and without the need of expensive hardware. While most standard compiler optimizations including constant folding and propagation, algebraic simplifications, copy propagation, common subexpression elimination, loop invariant code motion, loop transformations such as pipelining and interchange, etc. [32], are also relevant for power reduction, some others that increase the code size (such as loop unrolling and function inlining) need more careful attention. Optimizations such as unrolling and inlining increase the code size, thereby increasing the instruction memory size. Since larger memories are associated with increased access energy, these transformations may actually end up decreasing energy-efficiency.

5.2.1 Loop Transformations

Loop transformations such as *loop interchange, loop fusion, loop unrolling*, and loop tiling, which typically result in better cache performance through exploiting data reuse, also lead to improvements in power/energy by way of minimizing accesses to off-chip memory. Transformations such as unrolling cannot be indiscriminately applied because they lead to cache pollution, which affects performance; the same argument also applies to power, as we usually use cache misses as the evaluation metric.

However, other transformations such as *scalar expansion* work in the opposite direction. In scalar expansion, a global scalar variable shared across iterations that prevents parallelization, is converted into an array variable to remove the data dependency and parallelize the independent iterations. As the new array is mapped to memory (instead of possibly a register earlier), such an optimization results in a larger number of memory accesses and the associated address calculation, and consequently, worse power [23].

5.2.2 Instruction Encoding

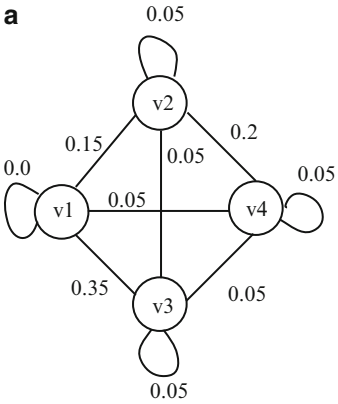
When a new instruction is fetched into the instruction register (IR), several bits of the current IR are switched. The switching activity during the instruction fetch phase is directly proportional to the number of bits switched in the IR between the successively fetched instructions. The bit changes on the opcode field can be decreased by assigning opcodes so that frequently occurring consecutive instruction pairs have a smaller Hamming Distance between their opcodes.

We can represent the instruction transition frequencies as an instruction transition graph (ITG) $G = (V, E, w)$ where V is a set of instructions, E is the set of undirected edges between all the elements in V, and w is a probability density function that maps each edge $e = (v_1, v_2) \in E$ to a real number between 0 and 1. $w(e)$ indicates the relative frequency of the instruction transitions between v_1 and v_2.

Given an instruction transition graph G, a set S of binary strings of length $\lceil log_2 |V| \rceil$, and an opcode assignment function $f : V \rightarrow S$, a power metric, the average switching in G under f can be defined as $P = \sum w(e) \times h(f(v_1), f(v_2))$, where h is a function returning the Hamming Distance between two binary strings. This is illustrated in the example shown in Fig. 5.16(b). Figure 5.16(a) shows the graph with nodes v1 to v4, each representing an instruction. The edges connecting these nodes are annotated with the instruction transition probabilities. Figure 5.16(b) shows the encoding of these instructions and Fig. 5.16(c) shows the cost incurred due to the transitions shown in Fig. 5.16(a) when the encoding shown in Fig. 5.16(b) is used.

For low-power opcode encoding, the goal is to find an optimal opcode assignment function f_{opt} that minimizes the power consumption. Standard finite state

a



b

Inst.	Opcode
v1	00
v2	01
v3	10
v4	11

c

Inst. Seq.	Hamming Dist.	Cost
v1 ↔ v2	1	1 x 0.15
v1 ↔ v3	1	1 x 0.20
v1 ↔ v4	2	2 x 0.05
v2 ↔ v3	2	2 x 0.05
v2 ↔ v4	1	1 x 0.35
v3 ↔ v4	1	1 x 0.05
v1 ↔ v1	0	0 x 0.05
v2 ↔ v2	0	0 x 0.00
v3 ↔ v3	0	0 x 0.05
v4 ↔ v4	0	0 x 0.05
	Total Cost	0.95

Fig. 5.16 An example showing the computation of the cost associated with an encoding of instructions when applied to a given Instruction Transition Graph. (**a**) Example instruction Transition Graph. (**b**) Instruction encoding. (**c**) Computation of cost

machine encoding techniques can be adapted for this purpose. Further reduction in IR switching can be effected by changing the register numbers in a generated binary to minimize the switching in register numbers in consecutive instructions.

5.2.3 Instruction Scheduling

In the instruction encoding discussed above, we assume that the instruction sequence is fixed. Instruction scheduling is the complementary optimization, where we exercise the flexibility to re-order instructions to minimize bit switching. Here, sequences of instructions can be re-scheduled where permissible to reduce transition count on the instruction register and the instruction memory data bus. Additionally, we can re-label registers in the generated instructions such that bit switching in consecutive instructions is reduced [31, 33, 39, 41].

In VLIW processors, different instructions may have varying number of operations, leading to a significant variation in the *step power* (difference in power between consecutive clock cycles) and *peak power* (maximum power dissipation during program execution). Both step power (which affects inductive noise) and peak power affect system reliability. A more balanced distribution of instructions in the schedule that avoids the extremes in terms of number of instructions in a cycle and transitions between them leads to better step power and peak power behavior. Since the instruction stream in VLIW processors is usually compressed, a reordering of the instructions within the same long word may lead to a better compression. The compression implications of different orderings can be evaluated by the compiler and the best one generated, ultimately leading to fewer I-Cache misses. Keeping in view the transition activity on the instruction bus, the instructions within the same VLIW instruction word can be re-ordered to minimize the Hamming distance from the previous instruction word. This can also be done across words, if the performance is not affected [6, 26, 36, 47].

Compared to the run-time environment, a compiler has a deeper view of the individual application being compiled, and can perform optimizations spanning a large section of code. In a hybrid VLIW/Superscalar architecture, a low-power enhancement to a superscalar processor is used, where, if the compiler is able to find efficient instruction schedules, then the low power mode is used and the circuitry for dynamic scheduling is turned off [43].

5.2.4 Dual Instruction Set Architectures

The Instruction Set Architecture (ISA) forms the interface between the hardware and software, and it is the compiler's task to convert an application expressed in high level language in terms of machine instructions. The instruction set itself has a very significant impact on the power-efficiency of program execution.

Traditionally, ISAs have been of fixed width (e.g., 32-bit SPARC, 64-bit Alpha) or variable width (e.g., x86). Fixed width ISAs give good performance at the cost of code size and variable width ISAs give good performance at the cost of added decode complexity. However, neither of the above are good choices for low power embedded processors where performance, code size, and power are critical con-

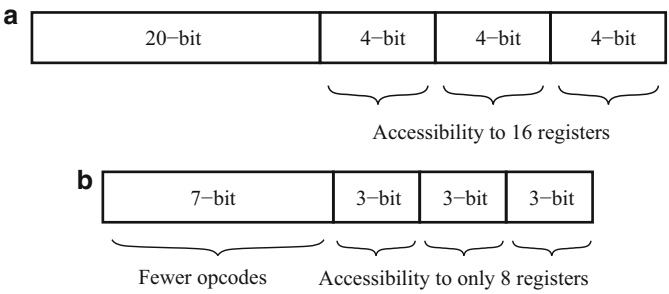


Fig. 5.17 Reduced bit-width Instruction Set Architecture or rISA is constrained due to bit-width considerations. Consequently, rISA instructions often have access to only a fraction of the register file. (**a**) 32-bit normal instruction. (**b**) 16-bit rISA instruction

straints. Dual width ISAs are a good trade-off between code size flexibility and performance, making them a good choice for embedded processors. Processors with dual width ISAs are capable of executing two different instruction sets. One is the "normal" set, which is the original instruction set, and the other is the "reduced bit-width" instruction set that encodes the most commonly used instructions using fewer bits (Fig. 5.17).

A good example of a dual-width ISA is the ARM [1] ISA with a 32-bit "normal" Instruction Set and a 16-bit Instruction Set called "Thumb". Other processors with a similar feature include the MIPS 32/16 bit TinyRISC [29], ST100 [38], and the Tangent A5 [3]. This feature is called the "**r**educed bit-width **I**nstruction **S**et **A**rchitecture" (**rISA**).

Processors with rISA feature dynamically expand (or translate) the narrow rISA instructions into corresponding normal instructions. This translation usually occurs before or during the decode stage (Fig. 5.18). Typically, each rISA instruction has an equivalent instruction in the normal instruction set. This makes translation simple and can usually be done with minimal performance penalty. As the translation engine converts rISA instructions into normal instructions, no other hardware is needed to execute rISA instructions. If the whole program can be expressed in terms of rISA instructions, then up to 50% code size reduction may be achieved. Code size reduction also implies a reduction in the number of fetch requests to the instruction memory. This results in a decrease in power and energy consumption by the instruction memory subsystem. Thus, the main advantage of rISA lies in achieving low code size and low energy consumption with minimal hardware alterations. However, compiling for rISA instructions is complicated due to several reasons:

- **Limited Instruction Set:** The rISA instruction set is tightly constrained by the instruction width. Since only 16 bits are available to encode the opcode field and the three operand fields, the rISA can encode only a small number of normal instructions. Therefore several instructions cannot be directly translated into rISA instructions.

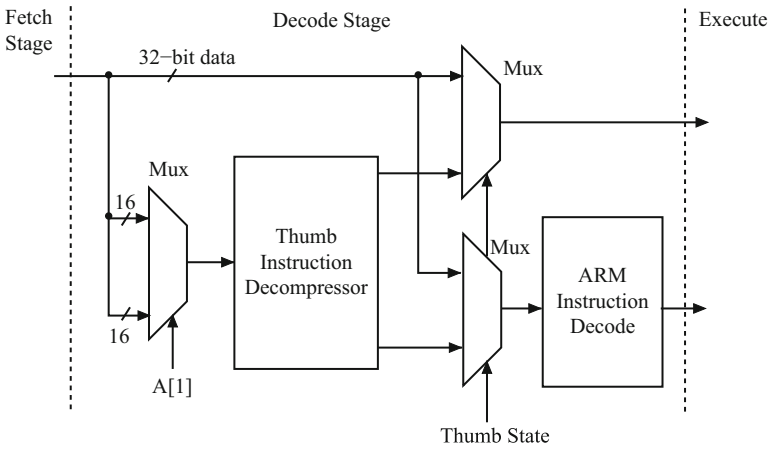


Fig. 5.18 rISA instructions are translated to normal instructions before or during decode. This allows the rest of the processor to stay unchanged

- **Access to only a fraction of registers:** The rISA instruction set, because of bit-width restrictions, encodes each operand (such as register address) using fewer number of bits. Therefore, rISA instructions can access only a small subset of registers. For example, the ARM Thumb allows access to 8 registers out of the 16 general-purpose ARM registers.
- **Limited width of immediate operands:** A severe limitation of rISA instructions is the inability to incorporate large immediate values. For example, with only 3 bits available for operands, the maximum unsigned value that can be expressed is 7.

Because of the problems mentioned above, indiscriminate conversion of normal instructions to rISA instructions may actually increase code size and power consumption, not only because a normal instruction can map to multiple rISA instructions, especially if it has large immediate operand fields, but also because of spill code since rISA instructions can access only a limited set of registers.

One of the most important decisions in a rISA compiler is the granularity at which to perform the conversion. The conversion can be performed at routine level granularity, where all the instructions in a routine can be in exactly one mode – the normal mode or the rISA mode. A routine cannot have instructions from both ISAs. Routine-level rISAization (the process of conversion from normal instructions to rISA instructions) has some drawbacks:

- First, a routine-level granularity approach misses out on the opportunity to rISAize code sections inside a routine that is deemed non profitable to rISAize. It is possible that it is not profitable to rISAize a routine as a whole, but some parts of it can be profitably rISAized. For example, in Fig. 5.19(a), Function 1 and Function 3 are found to be non-profitable to rISAize as a whole. Routine-level granularity approaches will therefore not rISAize these routines.

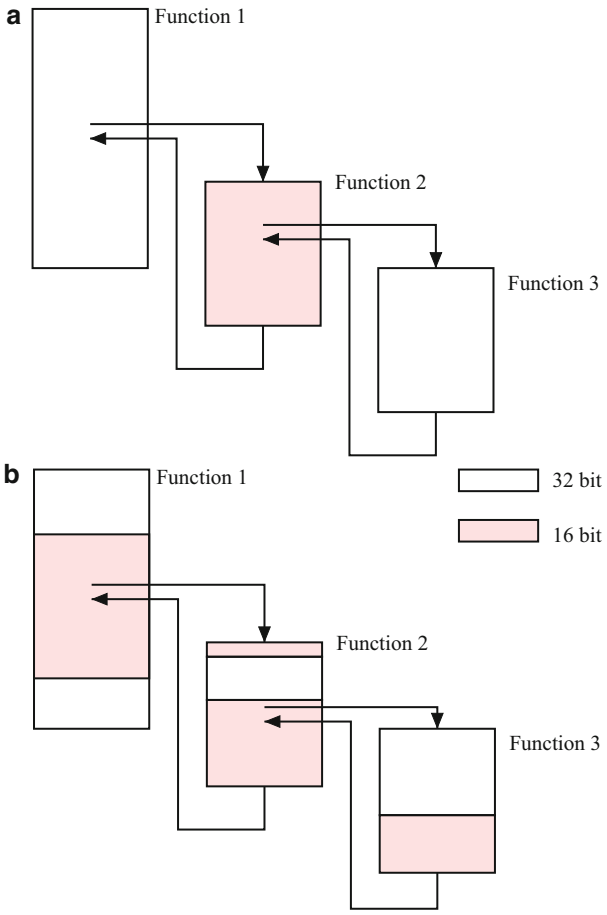


Fig. 5.19 rISAization at function level has very little overhead, but misses out on the possibility of selectively converting only the profitable regions of a function. (**a**) Routine Level Granularity. (**b**) Instruction Granularity

- Secondly, with routine-level rISAization, it is not possible to exclude from conversion some regions of code inside a routine that may incur several register spills. It is possible that excluding some pieces of code inside a profitable routine may increase the code compression achieved. For example, in Fig. 5.19(b) the instruction-level granularity approaches have the choice to exclude some regions of code inside a routine to achieve higher code compression.

Performing rISAization at instruction-level granularity alleviates both the above problems, as we can rISAize profitable portions of the application code, while excluding the non-profitable parts. However, rISAizing at instruction-level comes with its own set of challenges. Foremost is the overhead of the mode change operation: the instruction that informs the processor that the following instructions

are in the normal mode, or the rISA mode. In processors that implement routine-level conversion, this functionality can be added to the function call instruction, but instruction-level conversion requires explicit instructions. The direct implication of this is that converting only a few instructions will not be profitable, and several contiguous instructions must be converted to overcome the conversion overhead and obtain code size and power improvements. Since basic blocks are typically small, a good approach requires an inter-basic block analysis for conversion. Further, an effective approach also necessitates an associated scheme to estimate the register pressure in a code segment in order to more reliably compute the increase in the code size by rISAizing the code segment.

Experimentation with the rISAization strategy shows that rISA is a very effective code size reduction, as well as power reduction technique, and a smart compiler can consistently achieve upwards of 30% reduction in code size, and similar reduction in the power consumption of the instruction cache.

5.2.5 Instruction Set Extension

Instruction set extension is the process of adding new instructions in the processor, and adding the corresponding functional unit and control circuitry to enable the detection and execution of the new instruction, with the objective of improving the power and performance of the processor. This is specially useful in application specific processors (ASIPs), where there may be some large pieces of functionality that are used very often, and the application could benefit from performing it directly in hardware. Consider a cryptographic application using elliptic curve encryption to encode data. A processor used for this application could greatly benefit if the entire elliptic curve encryption could be performed as one single instruction, rather than as a sequence of smaller instructions. One common application for instruction set extension is the MMX extension to the x86 architecture that provides special instructions for SIMD arithmetic and string manipulation.

The procedure for extending the instruction set of a processor starts with identifying commonly occurring instruction patterns in the application set of interest, replacing them by a new instruction in the application code, adding a new hardware unit to execute the new instruction, and finally adding control logic to decode, issue, and commit the instruction.

An Instruction Set Extension or ISE typically encapsulates multiple atomic operations constituting the critical portion of the application. Execution of an ISE on a custom unit effectively migrates multiple operations from software to hardware, thus greatly accelerating the application performance. Along with performance, there are other obvious benefits of such application-specific processor customization. Because of compacting multiple operations into a single ISE, there is an overall code size reduction. Furthermore, we can expect energy reduction because fewer instructions are executed for every replacement of a large set of operations by the ISE. Such

replacement causes reduced switching activity due to reductions in the number of fetch, decode, and register store operations.

Automatic generation of ISEs is a key, and perhaps the most crucial step in automating the process of processor customization. To do this, the Control Flow Graph (CFG), and the Data Flow Graph (DFG) of the basic blocks of the application must be abstracted. DFG is a Directed Acyclic Graph (DAG) $G = (V, E)$, where the nodes V represents the instructions or external inputs/outputs and the edges E capture the data dependencies between the nodes. A cut $C \subseteq G$ can be a potential ISE if it satisfies some conditions:

- **Forbidden Operations:** Due to microarchitectural restrictions, operations of a certain type might not be allowed within the cut. For example, memory operations have been traditionally prohibited in the process of ISE generation. This is because, first of all, if memory operations are allowed in ISEs then the ISE must be combined with the load/store unit. Otherwise, the custom unit must have a new connection (that could be shared) to the memory, causing coherency issues between the data shared by the custom unit and the rest of the processor. Therefore, when searching for a cut, we have to find a maximal cut that does not contain any node that cannot be a part of the ISE.
- **Input-Output Constraints:** The custom unit will receive its operands from a register file (shown in Fig. 5.20). As a result, the number of source and destination operands of the new instruction is limited by the number of read and write ports respectively in the register file. For embedded processors with relatively fewer read/write ports on the Register File, this can be a crippling limitation.
- **Convexity Constraint:** Only convex cuts can be a candidate for ISE. In a convex cut C there exists no path from a node $u \in C$ to another node $v \in C$ through a node $w \notin C$. This is needed because scheduling policies in processors typically assume that all operands of an instruction are read before the instruction starts execution. Implementing a non-convex graph would require significant changes

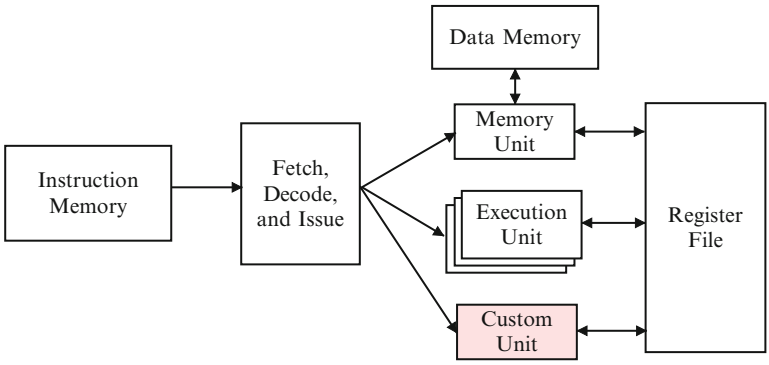


Fig. 5.20 In tightly-coupled processors, a custom unit is tightly integrated with the processor pipeline to implement instruction set extension functions

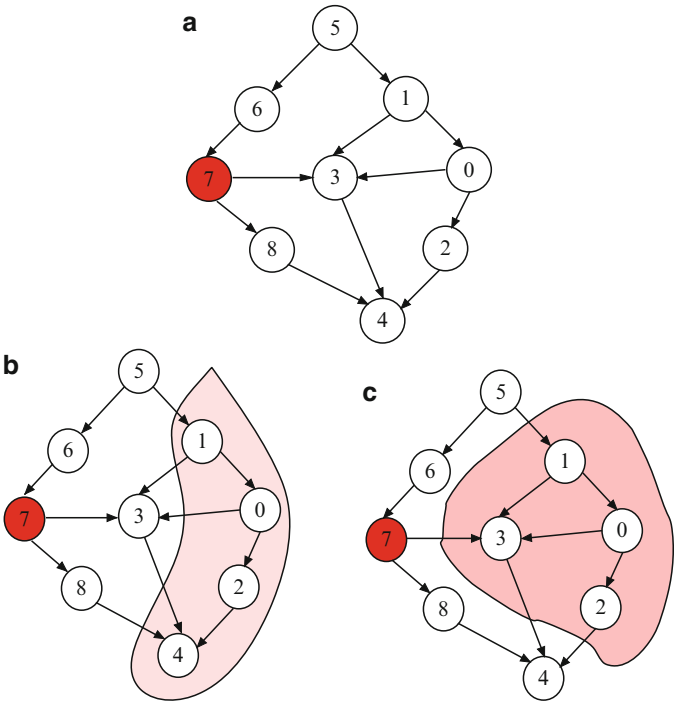


Fig. 5.21 The objective of ISE generation is to find a maximal cut that does not have forbidden functions (Shaded Nodes) and satisfies input-output and convexity constraints. (**a**) Graph with Max. Inputs = 3, Max. Outputs = 2. (**b**) Invalid Cut. (**c**) Valid Cut

in the scheduling policy. Figure 5.21(c) shows a valid cut, while Fig. 5.21(b) shows a cut that violates convexity constraints. This is because there is a path from node 1 to node 4 (both in the cut) that goes through node 3 (outside the cut).

Thus the problem of finding an ISE is to find non-overlapping cuts $C_i \subseteq G$ that satisfy the input-output, convexity, and forbidden operations constrains, and maximize the improvements in power and performance. This dual objective is tricky because on one hand, finding as large a cut as possible is beneficial, but on the other hand, the cut should be relatively small so that it is generic enough to have several instances in the application, to deliver good results. Integer Linear Programming (ILP) solutions have been developed, with the predictable behavior of generating optimal results but at the expense of too much time; they work well on small DFGs, and therefore are unable to find large cuts. On the other hand, heuristics have a hard time finding large cuts. Monte carlo and genetic solutions have also been explored. However it is difficult to define good fitness function and the termination criteria. Clustering techniques [11, 40] start with a seed node and use a guide function to select the best direction to grow the cluster. One technique prunes the candidates that do not reach a certain percentage of the best priority discovered so far, while the other prunes the directions of search that are not estimated to be worthy for growing

a candidate. ISEGEN [5] uses a graph partitioning scheme based on the Kernighan-Lin heuristic. On multimedia benchmarks and a processor with 4 read ports and 2 write ports on the Register File, an average of 50% speedup and is reported. The power savings also fall in the same range.

5.2.6 *Power Gating*

The compiler has an intimate knowledge of the processor microarchitecture. This has been exploited to develop several compiler techniques to modify the application, so that it executes in a power-efficient manner on the microarchitecture. Among various techniques proposed for leakage energy reduction at the microarchitecture level, power gating has emerged as one of the most promising approaches [8, 34]. In this technique, leakage power is reduced by shutting off the power supply to the FU during periods of inactivity (Section 3.5.3) [21].

Figure 5.22 shows the estimated energy density of different components in the ALPHA DEC 21364 processor while executing a representative susan-corners benchmark from the MiBench suite on PTScalar [27] simulator. The ALUs have the second highest energy density among all the units, next only to the integer register file. This observation is also consistent with other studies such as [12], where it is reported that compared to large modules such as secondary caches, FUs are very

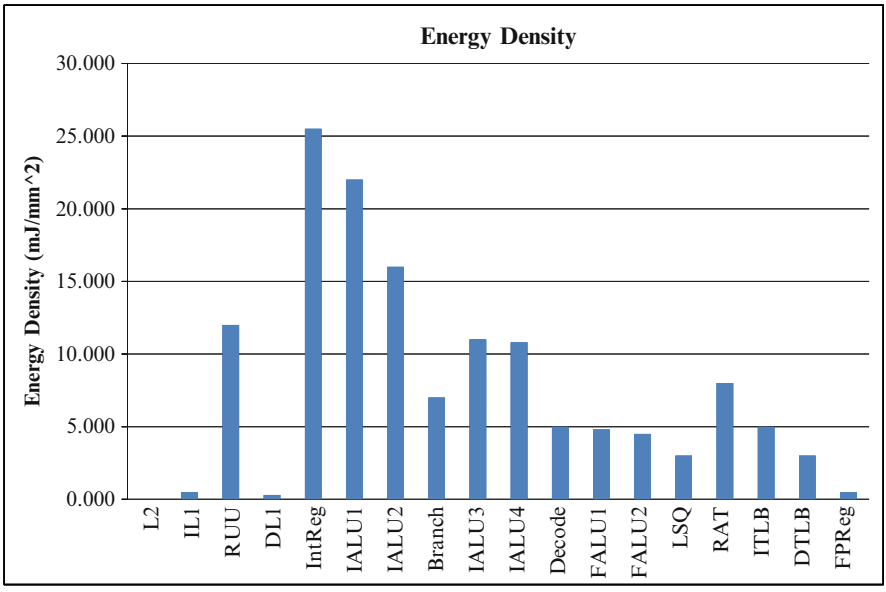


Fig. 5.22 Power gating of functional units is important, as they are typically the most important hotspots in the processor

active blocks with power densities up to twenty times higher. High power densities directly result in high temperature, which ultimately makes function units some of the highest leakage sites in the processor.

Power gating promises to be an effective approach for containing the leakage power of FUs. However, power gating large logic structures such as ALU require a large sleep transistor (see Section 2.5.5). Synthesis results at 65nm show that the delay of the sleep transistor will be about 6-10 processor cycles with a 3 GHz clock. Given this, the problem of power gating FUs translates to finding idle intervals of inactivity of the FUs, and power gating the FUs during these periods. The good news is that inherent instruction dependencies in programs ensures that we cannot use all FUs all the time. Hence, idle periods on FUs are a commonly occurring phenomenon.

One popular power gating technique is based on FU idle periods [42]. Here, the activity of FUs is monitored, and if an FU is idle for more than a threshold t_{idle} cycles, the power supply to the FU is gated off. The control circuit for power gating each FU is local and independent of other FUs. Once in a power-gated state, the FU will be woken up (power gating is disabled) when an operation is issued to it. Power gating has also been attempted in a VLIW compiler by issuing instructions to turn the FUs on or off. This is typically done at a loop-level – the number of FUs required for a loop is determined, and those not needed are turned-off. However, in order to not overheat the few active FUs, the activity is circulated among the FUs, turning them on and off in an iteration.

Use of tiny leakage sensors deployed on each FU can lead to further optimization of FU leakage power [24]. This approach attacks the power gating problem in two steps. First, it looks at the recent history of execution and determines how many FUs to keep "on". Second, it power gates the FU whose leakage is the least. Operations are issued only to the "on" FUs. Since the decision of which FUs to keep "on" is based on the leakage of the FU, it automatically considers the usage, temperature, and also the process variation effects. Because of process variations (manufacturing inaccuracies), FUs can have different base leakages. This is an exponentially growing problem as we tread towards finer dimensions in manufacturing. Leakage-aware power gating automatically considers this process variation effect, and is able to "even-out" the leakage of the FUs (Fig. 5.23).

5.2.7 Dynamic Translation and Recompilation

One traditional handicap of the compiler with respect to power optimization is that it has a limited view of the run-time environment. Since the compiler is unaware of what other tasks would be simultaneously contending for common system resources, it is difficult for it to be aggressive in its power optimizations. *Dynamic translation* and *dynamic recompilation* refer to techniques where a certain amount of code generation is actually performed by the CPU in hardware at execution time. The Transmeta Crusoe processor provided an early glimpse into such possibilities in a commercial setting [13]. A VLIW-style architecture was adopted with a view to

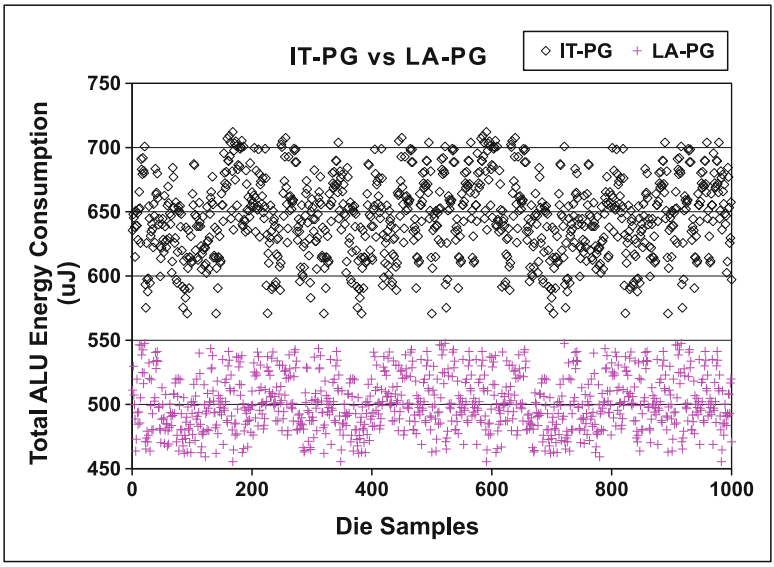


Fig. 5.23 Leakage-Aware power gating helps not only in reducing the leakage of FUs, but also helps in reducing the variation in the leakage due to process variations

reducing the power overhead of performing major tasks such as instruction reordering. Instead, a run-time software binary translator was used to generate the VLIW instructions from the original x86 code on the fly, a small sequence at a time. This still led to a significant overhead the first time the code was translated, but the resulting decoded VLIW code was cached so that future accesses to the same instruction could be read from the local memory, without the power overhead of decoding and instruction re-ordering.

The Crusoe processor was an early instance of a laptop class processor that could be run at several different voltage and frequency settings. In addition to the dynamic translation, a dynamic recompilation feature was also introduced, which would monitor the execution carefully to find frequently executed sections of code and generate optimized versions at run time. The dynamic translation and optimization feature has, since then, been implemented by several newer generation processors.

5.2.8 Compiler Optimizations Targeting Disks

Since accesses to the disk involves a significant amount of energy, making compiler optimizations disk-aware can help reduce overall system power. Both data layout and instruction transformations can benefit from knowledge of the disk subsystem. For example, data can be laid out in such a manner that in a parallel disk system, only a few disks are continuously accessed, generating the opportunity

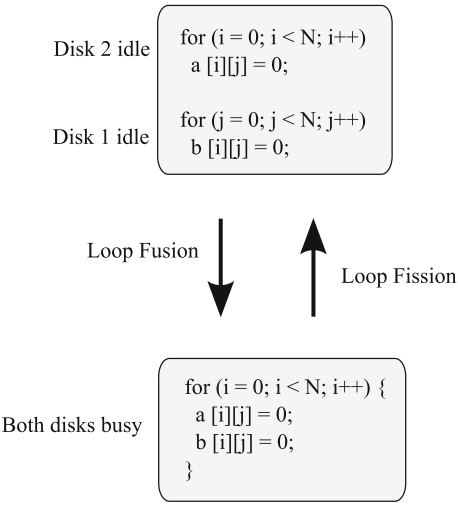


Fig. 5.24 The loop fusion transformation could be bad for disk power when data from different disks are accessed in the merged loop. In this example, arrays *a* and *b* reside on Disks 1 and 2 respectively. When the loops are split (left), Disk 2 can be powered down during the first loop and Disk 1 can be powered down during the second loop. When the loops are fused, both disks are busy throughout the merged loop

to power down the remaining ones. In this context, it is worth re-examining the implications of typical compiler optimizations – some of them work in the opposite direction in this context. An interesting observation is that *loop fusion* can be detrimental from the point of view of disk power, especially when it leads to additional arrays being accessed – and hence, more disks being activated simultaneously [22]. The reverse optimization, *loop fission*, can be beneficial using the same argument (Fig. 5.24). Note that this is in contrast to the previous observation in Section 5.2.1.

5.3 Application Software

Power awareness at the level of the hardware, operating systems, and compilers, is gradually finding its way to application software through application programming interfaces (APIs) that expose the underlying power management facilities. These APIs can be used by the programmer to pass useful information on to the operating system – specific information about the application's behavior that might not be easy to infer automatically. The converse is equally useful – knowledge provided by the operating system helps the application tune itself to the state of the system resources.

5.3.1 Application-aided Power Management

One class of hints that can be provided by an application includes task completion deadlines, expected execution times, and other measures of the estimated complexity of the task that might not be easily available statically, but could be present or

computed at run time. Such information can help the operating system make more informed power management decisions.

Exposing the state of different system resources to an application can help building systems that adapt themselves dynamically to achieve better power efficiency. An example is when there is a choice to fetch a piece of data from multiple sources – a disk and the network. If the current power modes (and associated performance penalties) of the connected devices were available to the application, a quick estimate could help decide the most appropriate device for servicing a request. If the disk is powered down, it may be cheaper in terms of both performance and energy to fetch relatively small-sized data from other networked devices [2, 44]. A general handling of this situation needs some additional intelligence. If a sequence of such small-sized requests are issued, then, beyond a certain count, it would be more energy-efficient to wake up the disk instead. A co-ordinated strategy is shown to be useful. Involving the application in the power management decision is useful here. The application, which may have knowledge about the future request patterns it will issue, can take the decision about the optimal power state of the disk. If such information is not present, then it could drop a hint to a power manager regarding what the ideal power state should have been for the device. After receiving several such *ghost hints*, the power manager can alter the power state of the device [2].

5.3.2 DVFS Under Application Control

So far, we have seen DVFS schemes being implemented either by the operating system or by the hardware itself. In both cases, the decisions have to be taken not on the basis of future requirements of the application but on the basis of past observed workload history. However, since the power-performance requirements of different applications are distinct, power management policies that are tailor made for the applications could result in improved power efficiency with minimum effect on performance [28]. A few example applications having varying nature of operation and the associated unique power management strategies are discussed below.

5.3.2.1 MPEG Video Decoder

MPEG video decoder is a soft real-time application – it needs to meet timeliness constrains, failing which, the quality of the user experience is degraded. Other applications such as DVD playback, audio players, music synthesizers, and video capture belong to the same class of soft real-time applications. These applications could be abstracted as a sequence of tasks such that each task completes within a given time. Applications in this class could use the following DVFS policy.

Consider a task among the sequence of tasks needed to be executed by the application. Let the task completion deadline be d starting from time t. If c is the CPU time needed to complete the task when the CPU is operated at maximum frequency and e is the CPU time allotted to this task before the deadline, the processor speed is calculated as shown below.

1. If $t + c > d$, the task is bound to miss the deadline even when operated at maximum frequency. Hence, we choose to operate at maximum CPU frequency.
2. If $e < c$, the CPU demand exceeds its availability and the task is bound to miss the deadline in this case also. Hence, it is best to run the processor at maximum frequency.
3. If $t + c < d$ and $e > c$, the task can be slowed down so that it completes just at the deadline. The frequency f at which the CPU is to be operated is calculated as

$$f = \frac{c}{\text{min(e,d-t)}} \times f_{\text{max}} \tag{5.2}$$

In order to compute f, the application needs to know the CPU availability e and an estimate of the processor demand c.

Estimation of CPU availability: An interface could be defined between the application and the OS such that the application receives the start and end times of, say, the previous k instances when this application was scheduled on the CPU. The average of times allotted in these previous instances can be used as an estimate for the availability in the next instance of the same application scheduled on the CPU.

Estimation of CPU demand: This could be obtained by characterizing the workload of a task in the application with respect to parameters that are expected to vary from task to task. For example, in the case of MPEG decoder, the decode time of a frame is found to be varying as a function of frame size and type of frame (frames are of three types I,B and P). Hence, a predictor could be built that computes the workload of the frame as a function of size and type of the frame. The predictor stores the observed decode times of previous N frames at full CPU frequency, to refit the prediction function to the parameters – size and type. Since each frame is annotated with a header that contains the information of frame size and type of frame, prior to decoding a frame we can extract this information and obtain the workload estimate from the prediction function.

5.3.2.2 Word Processor

This is an example of an interactive real time application. Several other applications such as games and web browsers, fall in this category of applications. These applications also consist of tasks that are to be finished within a deadline, but the tasks are initiated by an event of user interaction with the application. Hence, the workload of the applications is to be characterized as a function of these events. Since the number

of these events types is generally very large, a reasonable workload characterization is not feasible for such applications. Instead, an approach of gradually increasing the CPU frequency to satisfy the CPU demand can be employed for power management of these applications. The duration available for the task is divided into a number of sub-intervals. Processing is started at minimum CPU frequency and every time a sub-interval is crossed before task completion, the CPU frequency is scaled to next available frequency level.

5.3.2.3 Batch Compilation

Compilation using *make* is a batch application, where throughput is more important than the time taken for completion of individual tasks – in this case, a task being compilation of a program. Since it is difficult to estimate the compilation time of each program, the best strategy in this case would be to allow the end user to specify the required speed settings. For example, the user can specify the priority of the batch application to be low, and hence cause it to run in the background.

Thus we see that, different DVFS policies suit different applications, and the application programmer can contribute significantly to efficient power management depending on the power-performance characteristics of the application. As mentioned earlier, an enhanced interface is necessary through which the application can collect resource utilization statistics from the OS. Secondly, the scheduler should be modified such that per-process CPU power settings are maintained and conveyed to the underlying hardware whenever the program is scheduled for execution. Finally, the OS needs to have the ability to map the application-specific power setting to the appropriate CPU frequency supported by the hardware.

5.3.3 Output Quality Trade-offs

Often, applications have multiple choices of solutions at their disposal for a certain processing task. Different algorithms with different computational complexities could be employed for the same processing task, with different associated quality of results. Such choices could be judiciously exercised by an application when it is made aware of the status of resources in the run-time environment. For example, an MPEG encoder under power constraints could sacrifice compression efficiency by skipping some steps in encoding process. Of course, the trade-offs involved here – less energy to encode vs. more energy due to possibly larger I/O – should be properly studied before making the decision. Similarly, a video player with access to multiple versions of videos with different image sizes, could select smaller images when under energy constraints [44].

Many applications in the signal processing and graphics domain are characterized by a graceful degradation feature with respect to the bit-width of data types used for computation. Such flexibilities can be exploited by applications to continue

operation with reduced quality of output when under power and energy constraints. For example, when battery life-related constraints do not permit full-fledged processing with double precision arithmetic, an application could continue to operate by converting data to single precision and operating upon it, or by shifting to fixed point arithmetic.

5.4 Summary

Once power saving mechanisms have been incorporated into the underlying hardware, appropriate hooks need to be provided so that the software executing on the system can fully exploit them. In this chapter we covered the software components that can benefit from power awareness: the operating system, the compiler, and application software.

When the system under consideration is extended to include multiple tasks and multiple components such as the CPU, memory, I/O devices, and other resources, it is clear that the operating system emerges as an attractive entity in which to perform power management actions, since it has a good overall view of the resource usage by the different system tasks. We outlined several power management techniques including the important concept of intra-task and inter-task dynamic voltage and frequency scaling for real-time and non-real time systems.

The compiler interface is directly affected the first that is affected by the new hardware feature, since the compiler generates the code to execute on the hardware. Since a compiler has a deeper view of the program that is to ultimately execute on a processor, it can take power management decisions that may be difficult to handle at run time. We discussed different power optimization mechanisms involving the compiler: loop transformations, instruction encoding and scheduling, compilation for dual instruction architectures, instruction set extension, compiler directed power gating, and finally, disk optimizations. Finally, the application program can be made aware of the different hooks and knobs provided by the run-time environment to enable close monitoring of the state of system resources, as well as passing on crucial hints to the operating system about the state of the application.

References

1. Advanced RISC Machines Ltd: ARM7TDMI (Rev 4) Technical Reference Manual
2. Anand, M., Nightingale, E.B., Flinn, J.: Ghosts in the machine: interfaces for better power management. In: MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services, pp. 23–35 (2004). DOI http://doi.acm.org/10.1145/990064.990070
3. ARC Cores: ARCtangent-A5 Microprocessor Technical Manual
4. Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., Veidenbaum, A., Nicolau, A.: Profile-based dynamic voltage scheduling using program checkpoints. In: DATE '02: Proceedings of the conference on Design, automation and test in Europe, p. 168. IEEE Computer Society, Washington, DC, USA (2002)

5. Biswas, P., Banerjee, S., Dutt, N., Pozzi, L., Ienne, P.: ISEGEN: Generation of high-quality instruction set extensions by iterative improvement. In: DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, pp. 1246–1251. IEEE Computer Society, Washington, DC, USA (2005). DOI http://dx.doi.org/10.1109/DATE.2005.191
6. Bona, A., Sami, M., Sciuto, D., Zaccaria, V., Silvano, C., Zafalon, R.: Energy estimation and optimization of embedded vliw processors based on instruction clustering. In: DAC '02: Proceedings of the 39th conference on Design automation, pp. 886–891. New Orleans, Louisiana, USA (2002)
7. Burd, T.D., Brodersen, R.W.: Design issues for dynamic voltage scaling. In: ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design, pp. 9–14. ACM, New York, NY, USA (2000). DOI http://doi.acm.org/10.1145/344166.344181
8. Butts, J.A., Sohi, G.S.: A static power model for architects. In: Micro33, pp. 191–201 (2000). URL citeseer.ist.psu.edu/butts00static.html
9. Choi, K., Soma, R., Pedram, M.: Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. IEEE Transactions on CAD **24**(1), 18–28 (2005)
10. Clark, L.T., Hoffman, E.J., Biyani, M., Liao, Y., Strazdus, S., Morrow, M., Velarde, K.E., Yarch, M.A.: An embedded 32-b microprocessor core for low-power and high-performance applications. IEEE Journal of Solid State Circuits **36**(11), 1599–1608 (2001)
11. Clark, N., Zhong, H., Mahlke, S.: Processor acceleration through automated instruction set customization. In: In MICRO, pp. 129–140 (2003)
12. Deeney, J.: Reducing power in high-performance microprocessors. In: International Symposium on Microelectronics (2002)
13. Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., Mattson, J.: The transmeta code morphing^{TM} software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings of the international symposium on Code generation and optimization, pp. 15–24 (2003)
14. Govil, K., Chan, E., Wasserman, H.: Comparing algorithm for dynamic speed-setting of a low-power cpu. In: MOBICOM, pp. 13–25 (1995)
15. Gurumurthi, S., Sivasubramaniam, A., Kandemir, M.T., Franke, H.: Drpm: Dynamic speed control for power mangagement in server class disks. In: 30th International Symposium on Computer Architecture, pp. 169–179 (2003)
16. Hewlett-Packard, Intel, Microsoft, Phoenix Technologies Ltd., and Toshiba: Advanced Configuration and Power Interface Specification (2009)
17. Intel Corporation, http://www.intel.com/design/iio/manuals/273411.htm: Intel 80200 Processor based on Intel XScale Microarchitecture
18. Intel Corporation, http://www.intel.com/design/intelxscale/273473.htm: Intel XScale(R) Core: Developer's Manual
19. Ishihara, T., Yasuura, H.: Voltage scheduling problem for dynamically variable voltage processors. In: Proceedings of the 1998 International Symposium on Low Power Electronics and Design, 1998, Monterey, California, USA, August 10-12, 1998, pp. 197–202 (1998)
20. Jejurikar, R., Pereira, C., Gupta, R.K.: Leakage aware dynamic voltage scaling for real-time embedded systems. In: Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004, pp. 275–280 (2004)
21. Jiang, H., Marek-Sadowska, M., Nassif, S.R.: Benefits and costs of power-gating technique. In: ICCD '05: Proceedings of the 2005 International Conference on Computer Design. IEEE Computer Society, Washington, DC, USA (2005)
22. Kandemir, M., Son, S.W., Chen, G.: An evaluation of code and data optimizations in the context of disk power reduction. In: ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design, pp. 209–214. San Diego, CA, USA (2005)
23. Kandemir, M., Vijaykrishnan, N., Irwin, M.J., Ye, W.: Influence of compiler optimizations on system power. In: Proceedings of the 37th Design Automation Conference, pp. 304–307. Los Angeles, USA (2000)

24. Kim, C.H., Roy, K., Hsu, S., Krishnamurthy, R., Borkar, S.: A Process Variation Compensating Technique with an On-Die Leakage Current Sensor for nanometer Scale Dynamic Circuits. IEEE Transactions on VLSI **14**(6), 646–649 (2006)
25. Kim, T.: Application-driven low-power techniques using dynamic voltage scaling. In: 12th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006), 16-18 August 2006, Sydney, Australia, pp. 199–206 (2006)
26. Lee, C., Lee, J.K., Hwang, T., Tsai, S.C.: Compiler optimization on vliw instruction scheduling for low power. ACM Trans. Des. Autom. Electron. Syst. **8**(2), 252–268 (2003)
27. Liao, W., He, L., Lepak, K.: Ptscalar version 1.0 (2004). URL http://eda.ee.ucla.edu/PTscalar/
28. Liu, X., Shenoy, P., Corner, M.D.: Chameleon: Application-level power management. IEEE Transactions on Mobile Computing **7**(8), 995–1010 (2008). DOI http://dx.doi.org/10.1109/TMC.2007.70767
29. LSI LOGIC: TinyRISC LR4102 Microprocessor Technical Manual
30. Mahesri, A., Vardhan, V.: Power consumption breakdown on a modern laptop. In: Power-Aware Computer Systems, pp. 165–180 (2004)
31. Mehta, H., Owens, R.M., Irwin, M.J., Chen, R., Ghosh, D.: Techniques for low energy software. In: ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design, pp. 72–75. Monterey, USA (1997)
32. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufman, San Francisco, CA (1997)
33. Petrov, P., Orailoglu, A.: Compiler-based register name adjustment for low-power embedded processors. In: ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design, p. 523 (2003)
34. Powell, M., Yang, S.H., Falsafi, B., Roy, K., Vijaykumar, T.N.: Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In: ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design, pp. 90–95 (2000)
35. Sery, G., Borkar, S., De, V.: Life is cmos: why chase the life after? In: DAC '02: Proceedings of the 39th annual Design Automation Conference, pp. 78–83. ACM, New York, NY, USA (2002). DOI http://doi.acm.org/10.1145/513918.513941
36. Shao, Z., Xiao, B., Xue, C., Zhuge, Q., Sha, E.H.M.: Loop scheduling with timing and switching-activity minimization for vliw dsp. ACM Trans. Des. Autom. Electron. Syst. **11**(1), 165–185 (2006)
37. Shin, D., Kim, J., Lee, S.: Intra-task voltage scheduling for low-energy, hard real-time applications. IEEE Design & Test of Computers **18**(2), 20–30 (2001)
38. ST Microelectronics: ST100 Technical Manual
39. Su, C.L., Despain, A.M.: Cache design trade-offs for power and performance optimization: a case study. In: ISLPED '95: Proceedings of the 1995 international symposium on Low power design, pp. 63–68. ACM Press, New York, NY, USA (1995)
40. Sun, F., Ravi, S., Raghunathan, A., Jha, N.K.: Synthesis of custom processors based on extensible platforms. In: ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, pp. 641–648. ACM, New York, NY, USA (2002). DOI http://doi.acm.org/10.1145/774572.774667
41. Tomiyama, H., Ishihara, T., Inoue, A., Yasuura, H.: Instruction scheduling for power reduction in processor-based system design. In: DATE '98: Proceedings of the conference on Design, automation and test in Europe, pp. 855–860. Le Palais des Congrés de Paris, France (1998)
42. Tschanz, J.W., Narendra, S.G., Ye, Y., Bloechel, B.A., Borkar, S., De, V.: Dynamic sleep transistor and body bias for active leakage power control of microprocessors. IEEE Journal of Solid State Circuits **38** (2003)
43. Valluri, M., John, L., Hanson, H.: Exploiting compiler-generated schedules for energy savings in high-performance processors. In: ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design, pp. 414–419. ACM Press, New York, NY, USA (2003)
44. Venkatachalam, V., Franz, M.: Power reduction techniques for microprocessor systems. ACM Computing Surveys **37**(3), 195–237 (2005)

45. Weiser, M., Welch, B.B., Demers, A.J., Shenker, S.: Scheduling for reduced cpu energy. In: OSDI, pp. 13–23 (1994)
46. Yao, F.F., Demers, A.J., Shenker, S.: A scheduling model for reduced cpu energy. In: FOCS, pp. 374–382 (1995)
47. Yun, H.S., Kim, J.: Power-aware modulo scheduling for high-performance vliw processors. In: ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design, pp. 40–45. Huntington Beach, USA (2001)