System-Level Power Optimization: Techniques and Tools

LUCA BENINI
Università di Bologna
and
GIOVANNI DE MICHELI
Stanford University

This tutorial surveys design methods for energy-efficient system-level design. We consider electronic systems consisting of a hardware platform and software layers. We consider the three major constituents of hardware that consume energy, namely computation, communication, and storage units, and we review methods for reducing their energy consumption. We also study models for analyzing the energy cost of software, and methods for energy-efficient software design and compilation.

This survey is organized around three main phases of a system design: conceptualization and modeling, design and implementation, and runtime management. For each phase, we review recent techniques for energy-efficient design of both hardware and software.

Categories and Subject Descriptors: B.7.2 [**Integrated Circuits**]: Design Aids; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.1.0 [**Processor Architectures**]: General; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms: Design

1. INTRODUCTION

A system is a collection of components whose combined operations provide a useful service. Components can be heterogeneous in nature and their interaction may be regulated by some simple or complex means. Most systems are either electronic in nature (e.g., information processing systems) or contain an *embedded* electronic subsystem for monitoring and control (e.g., vehicle control). In this survey we consider electronic systems or subsystems; for the sake of simplicity, we refer to both as systems.

This work was supported in part by NSF under grant CCR-9901190, and in part by the MARCO Gigascale Research Center.
Authors' addresses: L. Benini, DEIS, Università di Bologna, Bologna, Italy; G. De Micheli, CSL, Gates Computer Science, Rm. 333, Stanford University, 353 Serra Mall, Stanford, CA 94305; email: nanni@stanford.edu.

System design consists of realizing a desired functionality while satisfying some design constraints. Broadly speaking, constraints delimit the design space and relate to the major design tradeoffs between system usefulness versus cost. System usefulness is tightly coupled with performance, i.e., the number of tasks that can be computed in a time window (system throughput) as well as the time delay to achieve a task (latency). Design cost relates to design and manufacturing costs (e.g., silicon area, testability) as well as to operational costs (e.g., power consumption, energy consumption per task, dependability).

In recent years the design tradeoff of performance versus power consumption has received much attention because (i) of the large number of mobile systems that need to provide services with the energy releasable by a battery of limited weight and size; (ii) the technical feasibility of high-performance computation due to heat extraction; (iii) concerns about the operating costs of large systems caused by electric power consumption as well as the dependability of systems operating at high temperatures because of power dissipation. (As an example, a data warehouse of an Internet service provider with 8000 servers needs 2 MW.)

Recent design methodologies and tools have addressed the problem of *energy-efficient design*, aiming to provide system realization while reducing its power dissipation. Note that energy/power optimization under performance constraints (and vice versa) is both hard to formulate and solve, when considering all degrees of freedom in system design. Thus we need to be satisfied with the notion of "power reduction" or with power minimization in a local setting. We use the term "energy-efficient design" to capture the notion of minimizing/reducing power and/or energy dissipation in system design, while providing adequate performance levels.

It is interesting to compare the evolution of goals in electronic system design with those of mechanical design, and in particular with combustion-engine design. In the beginning, achieving a working design was the engineer's goal, which was superseded by the object of achieving high-performance design. Later, energy efficiency in design was mandated by environmental and operating costs. In this respect, mechanical system design faced the problems of energy efficiency earlier, because of the larger consumption (as compared to electronic systems) of nonrenewable resources. Nevertheless, the energy consumption of electronic systems will scale up as they become more complex. Thus, economic, ecological, and ethical reasons mandate the development of energy-efficient electronic system designs.

2. SYSTEM ORGANIZATION AND SOURCES OF POWER CONSUMPTION

Typically, an electronic system consists of a hardware platform, executing system, and application software. Energy-efficient design requires reducing power dissipation in all parts of the design. When considering the hardware platform, we can distinguish three major constituents consuming significant energy: (i) *computation* units; (ii) *communication* units; and (iii)

storage units. Energy-efficient system-level design must address the reduction and balance of power consumption in all three constituents. Moreover, design decisions in a part of a system (e.g., the micro-architecture of a computing element) can affect the energy consumption in another part (e.g., memory and/or memory-processor busses).

Analyzing the hardware platform in more detail, we can distinguish integrated circuit (IC) components and components, such as *peripherals,* realized with other technologies. Peripherals may consume a significant fraction of system power. Examples of peripherals include electro-mechanical components (e.g., hard-disk drives) and electro-optical units (e.g., displays), which may consume a significant fraction of the overall power budget. Energy-efficient design of peripherals is beyond the scope of this survey; but, nevertheless, we will describe power management techniques that aim at shutting peripherals down during idle periods, thus drastically decreasing overall energy consumption.

Since software does not have a physical realization, we need appropriate models for analyzing the impact of software on hardware power consumption. Choices for software implementation (i.e., system-level software, application-level software, and their compilation into machine code) also affect the energy consumption of the three aforementioned componenets For example, software compilation affects the instructions used by computing elements, each one bearing a specific energy cost. Software storage and data access in memory affect energy balance, and data representation (i.e., encoding) affects power dissipation of communication resources (e.g., busses).

Modeling electronic systems is very important in order to abstract their characteristics and design objectives. Most textbooks, e.g., Hennessy and Patterson [1996], define *architecture* in the context of systems having one major computation engine. Thus, the *instruction set architecture* (ISA), which is the programmer-visible instruction set, provides a neat abstraction between hardware and software. Most recent electronic systems can be characterized as having several processing engines, operating concurrently and communicating with each other. Examples can be provided by chips for multimedia application, with several processing units, including a general-purpose processor core, a *digital signal processor* (DSP) core, and a microcontroller core. In general, we can think of systems as executing parallel threads of execution, some on processors (with different ISAs) and some directly in hardware. Thus the notion of *system organization*, or *macro-architecture*, is the appropriate high-level abstraction for implementation models. Conversely, we refer to the architecture of a single processor core as a *micro-architecture*. For brevity, we use the term "architecture" when its meaning is clear from the context.

System design consists of mapping a high-level functional system model onto an architecture. Energy efficient system design must be supported by a design flow that takes power consumption into account in all its steps. Tackling power dissipation in later design stages is generally an ineffective strategy because most key decisions on system organization have already

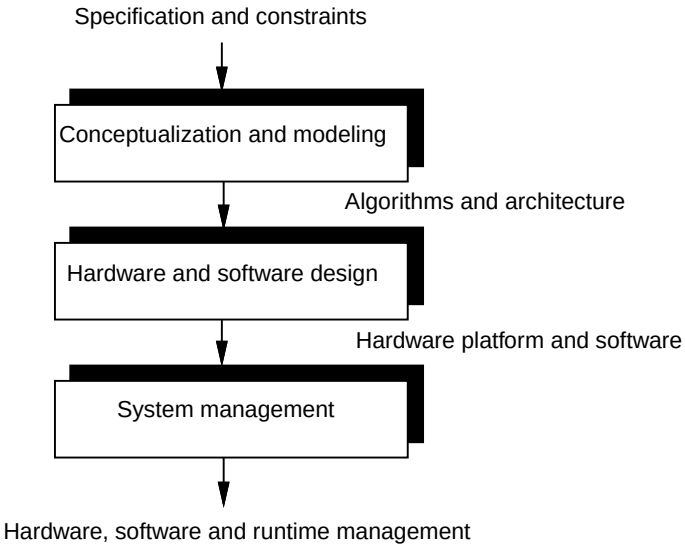


Fig. 1. Main steps in system design flow.

been taken. On the other hand, power-conscious design should not be limited to the early design stages alone because energy efficiency ultimately depends on detailed implementation issues. In this work we assume a top-down system design flow, summarized in Figure 1. We distinguish three major phases: (i) *conceptualization and modeling*; (ii) *design*; (iii) *management.*

In the first stage, starting from a system-level specification, the system architecture is outlined, together with the boundary between hardware and software. Clearly, at this stage, the designer has a very abstract view of the system. The most abstract representation of a system is the function it performs. The choice of the algorithm for performing a function (whether implemented in hardware or software) affects system performance and power consumption. In addition, the choice of the physical parameters (e.g., word-width) for the implementation of an algorithm is a degree of freedom that can be used in trading-off quality of service (e.g., image quality) for energy efficiency. Moreover, algorithms can have approximate implementations, i.e., certain operations may be implemented with limited accuracy to reduce energy costs. For example, a $\cos(x)$ function can be approximated as a Taylor expansion $1 - x^2/2 + x^4/24$. Furthermore, it may be approximated as $1 - x^2/2 + x^4/32$, which is simpler to realize than the Taylor expansion because the division of the last term by 32 can be done by shift. During system conceptualization and modeling, the system architect takes the key decision on algorithms (i.e., how to obtain the specified functionality), and hardware architectures (i.e., what is the hardware support required for implementing the functionality with the selected algorithm), but he/she does not deal with implementation details, which are left to the design phase.

During system design, organizational and technological choices are performed. At this stage, we are concerned with implementing the hardware architecture sketched in the conceptualization and modeling steps. When the architecture contains programmable components, we also need to produce the (embedded) software that executes on them. Design is supported by hardware synthesis tools and software compilers. Energy efficiency can obtained by leveraging the degrees of freedom of the underlying hardware technology. Several new technologies have been developed specifically for low-power systems [Vittoz 1994; Mendl 1995]. Notable examples are (i) technologies for very low supply-voltage operations; (ii) support for multiple supply voltages on a single chip; (iii) techniques for handling dynamically-variable supply voltage and/or clock speed. Special-purpose technologies are not the only way to design low-power systems. Even within the framework of a standard implementation technology, there are ample possibilities for reducing power consumption. It is important to note that tools for energy-efficient design may leverage degrees of freedom at various levels of abstraction. For example, a hardware resource scheduler may choose components operating at different voltages, and thus exploit both the freedom in choosing the time frames for executing the resources (as allowed by data flow) as well as resources operating with different supply voltages (as allowed by the technology) and with correspondingly different delay/power characteristics.

Once a system has been realized, and is deployed in the field, system-level software manages resources and thus controls the entire hardware platform, including peripherals. The main purpose of the last design phase is to equip the system with an energy-efficient runtime support system. Lightweight operating systems, i.e., those that avoid performing functions unimportant for the specific system application, are key to energy-efficient system design. Dynamic power management techniques allow systems to adapt to time-varying workloads and to significantly reduce energy consumption.

To help classify and compare energy-efficient system design techniques, we organize our survey by following a two-dimensional taxonomy. The first dimension corresponds to the design steps of Figure 1. The second differentiates the power consumed in computation, storage, and communication. Needless to say, a complete design methodology should cover all dimensions fully. Unfortunately, the state of the art has not yet converged to a complete methodology, and many techniques in the literature are applicable only during a single step in the design flow and focus on only one of the causes of power consumption.

The rest of this paper is organized as follows. We first review system modeling and conceptualization techniques for energy-efficient system design. Second, we address hardware synthesis and software compilation for low-power consumption. Finally, we consider system management issues. Within each design phase, we describe how various power optimization techniques target computation, storage, and communication energy consumption.

3. SYSTEM CONCEPTUALIZATION AND MODELING

In this section we focus on the early stages of the design process, when the boundary between hardware and software is not yet decided. Many authors refer to this design phase as *hardware/software codesign* [Wolf 1994; De Micheli and Gupta 1997]. We take a slightly different viewpoint. In our view, the key objective in this design phase is the creation of an abstract system model that will act as a frame for refinement and detailed design. We first survey system conceptualization and modeling styles, then outline the techniques proposed to steer the conceptualization process toward low-power solutions. Our main purpose is to identify a set of guiding principles for energy-efficient system design.

3.1 Models for System Specification and Implementation

System models are representations that highlight some characteristics while abstracting away some others. Typically, system models that capture all aspects of a design are complex and less useful than feature-specific models. Design of complex systems benefits from the orthogonalization of concerns, i.e., from decomposing the design problem into subproblems that are fairly independent of each other. The utilization of different, sometimes orthogonal, models is useful for capturing a design as well as those features most important to its designers.

3.1.1 *Functional Models*. *Functional models* are system specifications addressing functionality and requirements. A major distinction is between executable and nonexecutable functional models. *Executable functional models* capture the function that the system implements and allow designers to simulate system functionality. Some executable models are detailed enough to support synthesis of hardware components or software compilation; for example, models in *hardware description languages* (HDLs) such as Verilog HDL, or VHDL, and in programming languages such as C or C++.

Nonexecutable functional models highlight specific system features, while abstracting away some functional information. For example, the *task graph model* abstracts system functionality into a set of tasks represented as nodes in a graph, and represents functional dependencies among tasks with graph edges. The task graph is a nonexecutable functional model that emphasizes communication and concurrency between system tasks. Edge and node labelling are used to enrich the semantics of this model. For instance, node labels are used to represent communication bandwidth requirements, while node labels may store certain task computational requirements. Nonexecutable models are used when functionality is extremely complex or incompletely specified, which helps system architects to focus on just some facets of a complex system design.

Example 3.1 Consider a simple system that performs decimation, scaling, and mixing of two data streams. The executable functional model for such a system, in the C language, is given in Figure 2(a). A task graph

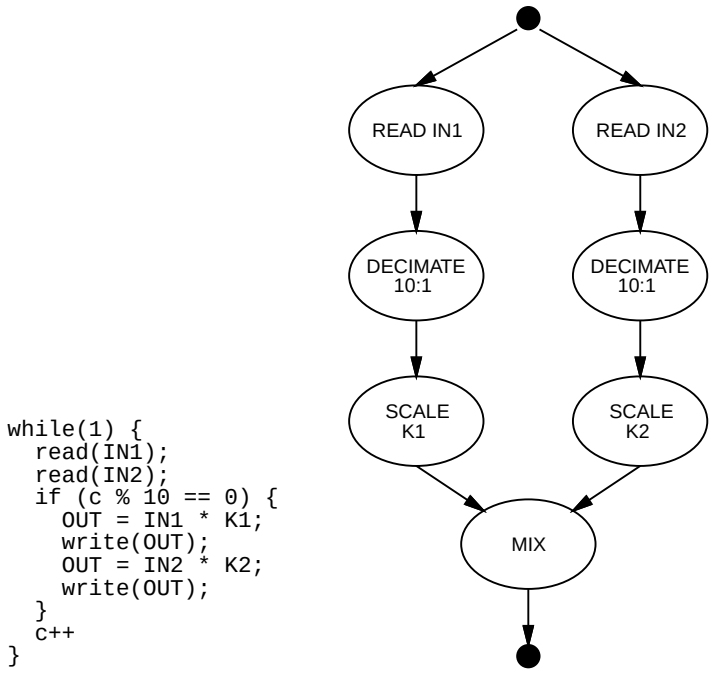


Fig. 2. (a)An executable functional model (in C); (b) a nonexecutable functional model (a task graph).

nonexecutable model is shown in Figure 2(b). While the C specification fully expresses the functionality of the system, the task graph abstracts away functionality, but provides information on concurrency (nodes with no directed path between them can be executed concurrently) and on communication (the edges represent the flow of data between tasks).

Currently, there is no widespread agreement on specification forms for system design; hybrid styles are often adopted (e.g., task graphs with executable functional models for each task). These forms are useful in bringing many different aspects of the system together, and in helping designers to keep them consistent during refinement and optimization. The reader is referred to Gajski et al. [1994] and Lee and Sangiovanni-Vincentelli [1998] for recent surveys on system specification styles and languages. Energy-efficient system design requires both executable and nonexecutable functional models.

3.1.2 *Implementation Models*. *Implementation models* are used to describe the target realizations for systems. To cope with the increasingly larger complexity of electronic systems, such models need to have specific properties, including modularity, component-orientation, and hierarchy. Implementation models include structural models that describe systems as an assembly of components. Structural models are often used as intermedi-

ate representations within the design trajectory, from functional models to detailed implementation models (e.g., logic-level, mask-level models). On the other hand, structural models are sometimes used to capture design partitions among components that are an integral part of the specifications themselves.

Some implementation models blend structural information with functional and operational information. In Section 3.4 we present two implementation models that are relevant to energy-efficient system modeling and design: the *spreadsheet* model [Lidski and Rabaey 1996] and the *power state machine* model [Benini et al. 1998c]. The former expresses a combination of components and evaluates the overall energy budget. The latter captures the power consumption of systems and their constituents as they evolve through a sequence of operational states.

3.1.3 *System Models and Design*. Depending on the targeted application, functional and implementation models are exploited in different ways. We distinguish between special-purpose and general-purpose systems. *Special-purpose systems* are designed with a specific complex application in mind, e.g., MPEG2 compression, ABS brake system control. In this case the system design task can be stated in simple terms: finding a mapping from a system function onto a macro-architecture. In practice, most systems are realized using libraries of complex components such as processor cores. Thus, the first steps in system design consist of finding a suitable partition of the system specification into processors, memories, and application-specific hardware units. The choice of a processor for realizing a part of the system functionality means that the subsystem functionality will be compiled into software machine code for the corresponding processor. Conversely, the choice of application-specific hardware units implies that the functionality will be mapped into silicon primitives using synthesis tools. Overall, the allocation of system functions to processors, implying a hardware/software partition, is a complex task [De Micheli and Gupta 1997]. Whereas tools and methodologies for computer-aided hardware/software partitioning have been proposed, it is still often done by the designers themselves We comment on methods for computer-aided hardware/software partitioning for low power in Section 3.3.

General-purpose systems are developed with no single application in mind, but with a flexible platform for a number of different applications; for example, personal digital assistants and portable computers. Functional models have little relevance for these systems, where the starting point is an architectural model and system design focuses on selecting the mix of components that best match design targets and constraints. Design of general-purpose systems emphasizes reuse of architectural templates and component families, in order to provide support to legacy applications and reduce time-to-market. The relationships among types of systems, models, and design flows are summarized in Figure 3.

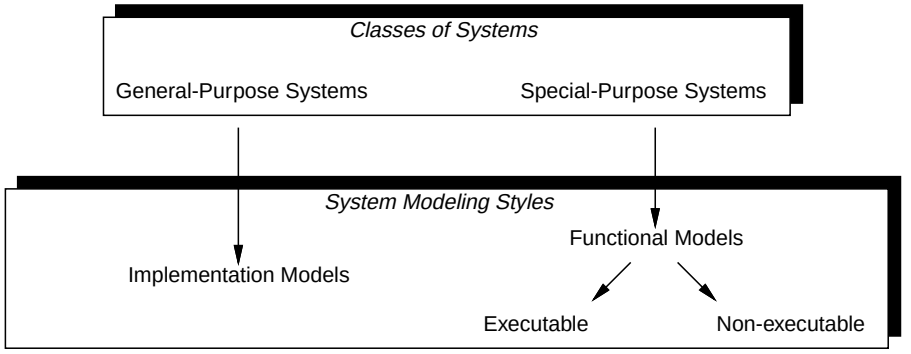


Fig. 3. Taxonomy of system conceptualization and modeling styles.

In the next sections we survey energy-efficient design techniques, starting from executable, nonexecutable, functional, and implementation models.

3.2 Energy-Efficient Design from Executable Functional Models

System specifications can be done using mathematical formalisms, i.e., pure functional specifications that describe a mapping between inputs and outputs. For example, in a GPS receiver, the inputs are signals from the GPS satellites, the outputs are spatial coordinates, mapping is implicitly expressed by the solution of a set of linear equations where the coefficients of the unknowns are derived from the input data. Pure functional specifications are quite uncommon in practice, and functionality is usually specified through procedural executable models. The widespread use of procedural models stems from the fact that both HDLs and commonly used programming languages (e.g., C, C++, Java) have a procedural semantics.

Broadly speaking, the generation of procedural models is a way of providing a "solution recipe," i.e., an algorithm. Unfortunately, procedural models are usually fairly biased, since multiple algorithms can solve a mathematical problem. As an example, a *sorting* function can be done by different algorithms. Search for the "best" algorithm is typically a very hard problem due to the size of the search space. Nevertheless, different procedures require more/less computation, storage, and communication. Thus, according to the relative energy costs of these operations, and typically to the corresponding performance, one procedure may be better than another.

Algorithm selection. To limit the search space and allow some form of algorithmic exploration, some approaches take a hybrid functional-procedural approach based on *algorithm selection* [Potkonjak and Rabaey 1999]. Procedural specifications usually contain function calls, which can be seen as pure functional views of subtasks in a complex computation. Algorithm selection assumes the existence of a library with multiple algorithms for computation of some common functions. Furthermore, it is assumed that

the energy consumption and performance of the library elements on a given target architecture can be characterized beforehand. For each function call in the procedural specification, we can select the algorithm that minimizes power consumption while satisfying performance constraints.

Example 3.2 In their study of algorithm selection for low power, Potkonjak and Rabaey [1999] considered several different algorithms for discrete cosine transform (DCT) computation. As a target architecture, they considered fully application-specific hardware, synthesized with a high-level synthesis system. For each algorithm, a power-performance tradeoff curve was obtained by synthesizing a circuit with different supply voltages.

After library characterization, system optimization based on algorithm selection is performed. For each call to the DCT function in a complex procedural specification, a heuristic optimization procedure selects the DCT implementation and the supply voltage for the corresponding circuit that optimally trades-off performance for power.

Algorithm selection may also be made from an object-oriented viewpoint, in which algorithms are seen as methods associated with objects. Hence, selecting an object or, in traditional terms, an abstract data type, implies algorithmic selection as well as data structure selection. This approach is taken by Wuytack et al. [1997] and Da Silva et al. [1998], where it is demonstratd that abstract data type selection has a major impact on power dissipation. At a lower level of abstraction, Sacha and Irwin [1998] and Winzker [1998] show that number representation is also an important algorithmic design choice for low-energy DSP systems.

The algorithmic-selection approach has several shortcomings. First, it can be applied only to general-purpose primitives, whose functionality is well known, and for which many functionally-equivalent algorithms have been developed. Second, it neglects the impact of the interaction between the hardware implementing the function and that executing the remaining computations. Hence, precharacterizing the power consumption of the function in isolation may be inaccurate. These issues must be addressed by techniques for energy-efficient design, starting from a generic executable functional model, which is outlined next.

Algorithm computational energy. The semantic of a generic functional specification can be expressed by hierarchical control-data flow graphs (CDFGs), where nodes represent elementary operations, edges represent control and data dependencies among operations, and nested procedure calls correspond to transitions between successive hierarchy levels [Aho et al. 1988]. The CDFG contains complete information on the computation performed by an algorithm, seen as data manipulation and control flow. Hence, it is conceivable to be able to assess, through CDFG analysis, the energy cost spent by computation. This approach has been taken by some researchers [Chau and Powell 1992 ; Tiwari et al. 1996].

To estimate computational energy by CDFG analysis, we need first to characterize elementary operations with a computational energy metric,

then we need compositional rules to compute the total cost of a complex CDFG containing many elementary operations. Energy estimates of elementary operations can be obtained by assuming an implementation style and by extracting cost per operation through characterization experiments. Composition rules also depend on the implementation style, and account for the impact that the execution of an operation may have on the energy dissipated in the execution of other operations later in time. Cost per operation and composition rules have been developed for both processors [Tiwari et al. 1996] and custom units [Chau and Powell 1992].

Example 3.3 An early example of computational energy estimation is the power factor analysis (PFA), by Chau and Powell [1992]. The PFA technique targets pure data-flow computations. Elementary operations are the basic arithmetic operations. Operational energy is obtained by simulating arithmetic circuits with random white noise data and by computing average energy per evaluation. Total energy is computed by simply summing the energy of all operations. Using this simple model, it is possible to formulate several algorithms to minimize computational power. For instance, data-flow transformations based on algebraic manipulation, such as those proposed by Chandrakasan et al. [1995], can be driven by the PFA metric.

Algorithm communication and storage energy. Unfortunately, procedural specifications hide storage and communication costs, which are implicit in the representation. Therefore, a simple algorithm may require large storage, and thus efficiency of realization cannot simply be related to computational energy, especially when the energy cost of storage and data transfer is significant. Several researchers have observed that storage and communication requirements are related to the *locality* of a computation [Mehra et al. 1996; 1997; Catthoor et al. 1994]. Lack of locality in a highly sequential control flow implies that results computed very early are needed much later in time. Data variables have a long lifetime, thereby increasing storage requirements. Similarly, a highly parallel computation with significant communication between parallel threads is likely to require a complex and power-hungry communication infrastructure.

Estimating algorithmic locality from a CDFG is a difficult task because this information is not explicitly available. Nevertheless, a few locality analysis procedures have been developed for data-dominated algorithms, i.e., pure data-flow specifications. Mehra and Rabaey proposed an approach based on the analysis of the connectivity of a data-flow specification [Mehra et al. 1996]. Currently, however, locality analysis for general CDFGs is still an open issue.

Example 3.4 The technique developed by Mehra et al. [1996] to assess algorithm locality transforms a given DFG into a weighted graph, where edge weights are assigned through several heuristics that take into account the amount of data transferred on each edge of the original DFG. An optimal linear placement for the graph is then computed using Lagrangian

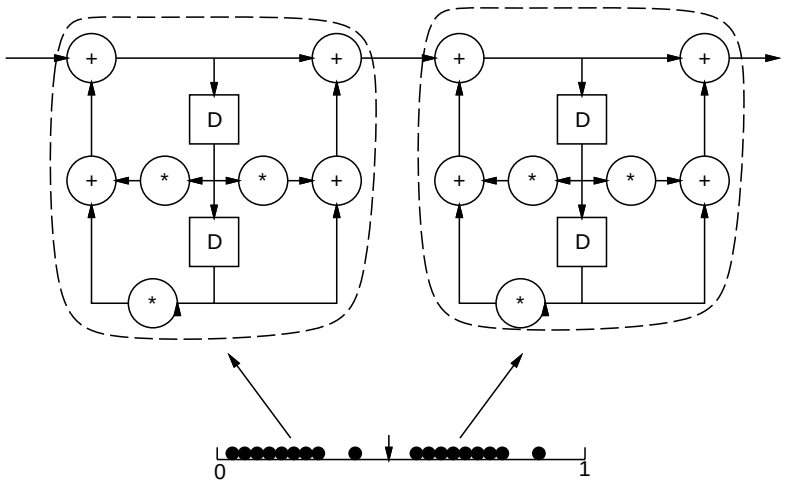


Fig. 4. Locality analysis for pure data-flow graphs.

minimization. Finally, the locality of the algorithms is assessed by detecting clusters in linear placement. Algorithms with poor locality have few and unbalanced clusters. As an example, consider the data flow of a fourth-order IIR filter, as shown at the top of Figure 4. The optimal linear placement of the corresponding weighted graph is shown at the bottom of the figure. Notice that linear placement has two natural clusters, which suggest that the algorithms has good locality.

In Mehra's approach, locality analysis is directly exploited to create loosely-connected clusters of operations that are mapped to different dedicated hardware units. This partitioning technique reduces the power consumed in the global interconnects of the hardware implementation of the initial DFG. In the example of Figure 4, the natural clusters of the data-flow graph, inferred from locality analysis, are shown by dashed lines.

It is important to observe that the relative energy cost of computation, communication, and storage is strongly dependent on the target architecture and technology. Consider for instance the relative energy costs of storage and computation. In traditional technologies, large DRAM memories cannot be integrated on the same die with logic circuits. Hence, the energy cost for storing data in large background memories involves off-chip communication, which is much more power-consuming than on-chip computation. This observation is the basic motivation behind the large body of work developed at IMEC [Catthoor et al. 1994], which focuses on algorithmic transformations for reducing memory requirements and rationalizing memory accesses. In contrast, recently developed technology, known as *embedded DRAMs*, enables the integration of large DRAMs with digital logic circuits on the same die. In this technology the balance may shift, and computation may have energy costs similar to memory access.

Example 3.5 Consider two well-known video decompression algorithms, namely MPEG decoding and vector quantization. Roughly speaking, MPEG

decoding is based on computationally intensive inverse cosine transform (IDCT) and motion compensation procedures. But vector quantization decompression is based on lookups in a code decompression table; hence it is a memory-intensive approach. Meng et al. [1995] designed a low-power video decoder chip based on vector quantization. They claimed that vector quantization is very power efficient because decompression is simply based on a memory lookup, and does not require extensive computations, as in the case of MPEG decoding. This conclusion is true only if the decompression memory is on-chip. Indeed, this was the case in the design of Meng et al. If decompression had required off-chip memory access, MPEG decoding would probably have been more power-efficient.

Computational kernels. In pure, nonhierarchical data-flow computation, every elementary operation is executed the same number of times. This is a very restrictive semantic, even for data-intensive applications, since almost every nontrivial data processing algorithm involves some form of control flow and repetitive computation. Hence, most algorithms have loops, branches, and procedure calls that create a nonuniform distribution for the number of times each operation is executed. The well-known empirical "law" that states that most execution time is spent on a small fraction of a program is an equivalent formulation of the same concept. Such nonuniformity is very relevant for energy-efficient system design.

We call *computational kernels* [Wan et al. 1998 ; Henkel 1999] the inner loops of a computation, where the most time is spent during execution. Profiling an algorithm execution flow under "typical" input streams can easily detect computational kernels. To substantially reduce power consumption, each computational kernel is optimized as a stand-alone application and implemented on dedicated hardware that interfaces with the less frequently executed sections of the algorithm whenever the flow of control transitions into and out of the kernel. During system operation, when the computation is within a computational kernel, only the kernel processor is active (and dissipates power), while the rest of the system can be shut down. Otherwise, the kernel processor is disabled. In other words, kernel extraction forces mutual exclusiveness in hardware by limiting hardware sharing. For this reason, energy efficiency is usually obtained at the expense of silicon area and, in some cases, of marginal delay increases. These overheads should be taken into account when evaluating the potential of this approach.

The potential of computational kernels has been exploited by several optimization techniques at low levels of abstraction [Benini and De Micheli 1997].Research on system-level computational kernels is still in its infancy. Two techniques with strong similarities have been proposed by Wan et al. [1998] and by Henkel [1999]. Both assume a system architectural template containing a general-purpose processor, and one or more application-specific processor which are synthesized when needed. Profiling data is collected on a system-level executable model. From profiling, the main computational kernels are extracted. The kernels are them synthesized in

hardware as application-specific processors. Increased energy efficiency is claimed based on the observation that performing a computation on dedicated hardware is usually one or two orders of magnitude more power efficient than using a general-purpose processor.

Example 3.6 MPEG audio decompression is well-suited for extraction of computational kernels. Profiling analysis has revealed that for almost any software implementation of the MPEG standard, most of the execution time is spent in two procedures: the modified inverse discrete cosine transform (MIDCT), and the sub-band synthesis filter. These procedures have little control flow, and mostly perform data manipulation (digital filtering). Hence, they are ideal candidates for a dedicated hardware implementation.

Notice that most current general-purpose and digital signal processors can easily perform MPEG decompression in real time; hence, from a performance viewpoint, mapping computational kernels in hardware is not needed. However, we can still exploit the energy efficiency of custom-hardware implementations of MIDCT and subband synthesis to reduce power dissipation.

Approximate processing. In many cases system functionality is not specified in a succinct fashion—fuzzy and flexible specifications are very common in the area of human sensory interfaces. Consider for instance a video interface such as wireless video phone. Even though we can provide a succinct specification on video quality (such as image resolution), it may impose excessively tight constraints on the design. Human users may be satisfied with low video quality, for instance when watching a television show, but they may require very high quality when reading a page on screen.

A few techniques take an aggressive approach to power reduction for applications where a well-controlled amount of noise can be tolerated. The key idea here is that power dissipation can be drastically reduced by allowing some inaccuracies in the computation [Vittoz 1994; Ludwig et al. 1996; Nawab et al. 1997; Hedge and Shanbhag 1998; Flinn and Satyanarayanan 1999]. Approximate computation algorithms can adapt this quality to power constrains and user requirements.

Example 3.7 An example application of approximate computation is digital filtering. By dynamically controlling the bit width of the processed data, it is possible to trade-off quantization noise for energy. In the finite-impulse response (FIR) filter architecture proposed by Ludwig et al. [1996], it is possible to dynamically change the bit width. The logic in the fanin and fanout of unused bit lines is disabled and does not consume power. Clearly, reduced bit-width computation is inaccurate, and the quality of the output is affected by quantization noise. In some applications, and for some operating conditions, this is tolerable.

3.3 Energy-Efficient Design from NonExecutable Functional Models

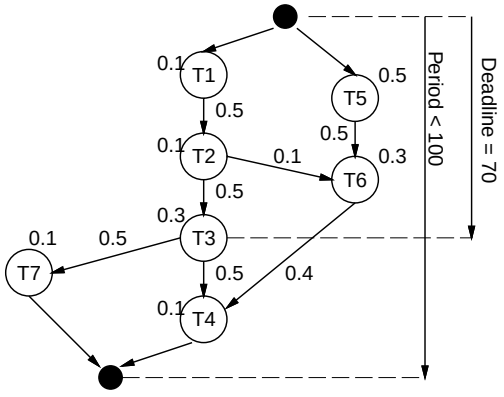
Many systems are not designed starting from executable functional models. In the early steps of system design, executable models may not be available, either because the specification is provided in natural language or because the definition of system characteristics is flexible and evolves as the design progresses. Furthermore, a complete description of system functionality is often extremely complex, and some high-level design issues must be addressed much before a complete executable functional model is available. As a result, many systems are designed without an executable specification.

Even when an executable specification does exist, designers may want to deal with a simplified, abstract view of the system, where only some specific characteristics are considered in detail, while many others are abstracted away. Thanks to the complexity reduction provided by abstraction, it is possible to rapidly explore the design space, at the price of an increase in the uncertainty of results.

A very common abstract system model, which still contains some information on functionality, but eliminates a lot of the detail needed in an executable specification, is the *task graph*, described in Section 3.1.1. Task graphs have been used in a variety of design environments much before energy efficiency became a primary concern. Hence, the main challenge in using this model for low-power design is in enhancing its semantics to include information on power dissipation.

Several authors proposed techniques that start from a task graph specification for minimizing the power consumption at the system level [Brown et al. 1997; Kirovski and Potkonjak 1997; Dave et al. 1999; Hong et al. 1998a; 1998b; 1998c; Ishihara and Yasuura 1998a; Qu and Potkonjak 1998]. These techniques attempt to provide a solution for variations of the following design problem: Given a task graph specification and an architectural template for system implementation with several functional units, find the mapping of tasks to functional units that minimizes energy while respecting performance constraints. Notice that we use the term "mapping" to denote both the binding of a task to a given computational resource and the scheduling of multiple tasks to the same resource.

The target architectural templates have considerable flexibility: processing elements, interconnect, and memory architecture are application-specific; more particularly, processing elements can be core processors, custom hardware, or programmable logics (FPGAs). Memory size and type (SRAM, DRAM, Cache, etc.) can be selected. Communication is supported through system busses. In some cases, supply voltage is an additional degree of freedom. In principle all sources of power dissipation can be addressed at this level, although the key issue is how to reliably estimate the power consumed by a given mapping. Needless to say, at this level of abstraction, only rough estimates can be performed to help designers compare design alternatives, but these estimates should never be used for design validation



Power	PE1	PE2
T1	10	20
T2	15	30
T3	10	20
T4	11	7
T5	32	41
T6	7	10
T7	12	22

Link	Pow	Speed
L1	10*BW	40*BW
L2	20*BW	30*BW

Time	PE1	PE2
T1	5	8
T2	7	12
T3	4	7
T4	8	5
T5	20	35
T6	2	3
T7	11	17

Mem	Pow	Speed
M1	50*S	12*S
M2	60*S	15*S

PE Allocation, Binding, Scheduling:

PE1: {T1–>T2–>T3–>T7–>T4}
PE2: {T5–>T6}

Mem Allocation

M1: {T5, T6}
M2:{T1 ,T2, T3, T4, T7}

Communication

PE1<–>PE2: L1

Fig. 5. Power estimation for nonexecutable functional models: Task graph.

(where only small errors with respect to the final implementation are acceptable).

In the approach proposed by Kirovski and Potkonjak [1997] and Dave et al. [1999], it is assumed that the power consumed by each task when mapped to each compatible class of processing elements is known or is obtained by precharacterization. It is also assumed that memory usage and communication requirements for each task are known beforehand. Given a mapping of a task graph into hardware, the total power consumed by functional units is computed summing the power consumed by each task when executing on the functional unit specified by the mapping. Task power consumption for storage is computed by multiplying the expected number of accesses of each task by the power-per-access of the memory that stores the task's data set. Communication power is computed by multiplying the number of data transfers initiated by the task by the cost-per-data transfer of the system bus used for communication.

Example 3.8 Consider the task graph shown in Figure 5. Two performance constraints are specified. The execution period of the entire task graph must be less than 100 time units; furthermore, execution of task T3 must terminate by time 70. Tasks (i.e., nodes) are labeled with memory usage. Edges are marked with communication bandwidths. We have two types of processing elements, PE1 and PE2 (e.g., two different core processors), two types of memories, M1 and M2 (e.g., SRAM and DRAM), and two types of communication channels, L1 and L2 (two different busses). The execution time and power (energy) consumption of each task on each processor is shown in the tables labeled "Time" and "Power." The energy and time costs of communication are a function of bandwidth, and are collected in the table labeled "Link." The energy and time costs for memory are a function of memory size, and are shown in the table marked "Mem."

An allocation of processing elements, with binding and scheduling, is shown at the bottom of Figure 5. Memory and communication channel allocation is shown as well. With this information, we can compute performance and power for the task graph. Let us consider computation energy first:

$$E_{PE} = E_{PE1} + E_{PE2} = (10 + 15 + 10 + 11 + 12) + (41 + 10) = 109$$

Observe that computation energy for a shared processor is computed by simply summing the energies required by all tasks running on it. Storage energy is computed by first calculating total memory size for each memory type and then multiplying it by its energy-per-size value (from table "Mem"):

$$E_{Mem} = E_{M1} + E_{M2} = (0.1 + 0.1 + 0.3 + 0.1 + 0.1) \cdot 60 + (0.5 + 0.3) \cdot 50 = 88$$

Finally, communication energy is obtained by first computing the total communication bandwidth between the two PEs and then multiplying it for the energy-per-bandwidth of the selected communication link:

$$E_{Com} = E_{PE1 \rightarrow PE2} + E_{PE2 \rightarrow PE1} = (0.1 + 0.4) \cdot 20 = 10$$

The total energy is $E = E_{PE} + E_{Mem} + E_{Com} = 109 + 88 + 10 = 206$. Speed is computed in a similar fashion.

The main issues in power estimation (and, as a consequence, optimization) for nonexecutable models are the need for exhaustive precharacterization and loss of accuracy caused by lack of information on the effects caused by hardware sharing. Precharacterization requires estimating the power consumed by each task on every possible hardware implementation, and for every possible voltage supply (if multiple supply voltages are available). This process can be extremely onerous for large task graphs and for large libraries of processors. Furthermore, estimating the power consumed by a task's implementation on dedicated hardware is far from trivial, since such an implementation may be unavailable in the early phases of system design.

Whenever a hardware resource is shared among several processes, it is generally very hard to provide an accurate estimate of the power cost of sharing without having functional information. Consider for instance a shared processor core with an on-chip cache. If two processes share a processor, the total power consumption is likely to be substantially larger than the sum of the power consumed by the two processes running as a single task. The overhead is caused by task-scheduling support (e.g., context switching, preemption, synchronization on mutually exclusive resources) and by cache locality disruption caused by the different work sets of the two processes. Overhead estimates depend strongly on functional information, which is not available when dealing with abstract task graphs.

	#Comp	Vdd	I_{IDLE}	I_{ON}	%idle	%on	I (mA)
Processor	1	3.3	0.5	50	0.3	0.7	35.15
DRAM	1	3.3	0.1	12	0.3	0.7	8.43
FLASH	5	3.3	0.0	9	0.3	0.7	31.5
IR	1	3.3	0.0	64	0.95	0.05	3.2
RTC	1	3.3	0.0	0.1	0	1	0.1
DC–DC	1	3.3	0.1	5.5	0.01	0.99	5.44
						Total	**83.82**

Fig. 6. Spreadsheet power model for an electronic agenda.

3.4 Energy-Efficient Design from Implementation Models

Many system designs are upgrades from older products and bear strong backward compatibility constraints. Such design often exploit commodity components connected through standard interfaces (e.g., PCI or USB busses). Thus, system modeling may be constrained from the beginning to using specific parts and interconnection topologies. Implementation models are useful in reasoning about different implementation options and the corresponding energy impact.

The spreadsheet model. In this design flow, executable functional models are never available, and rapid prototyping on breadboards is a common technique to obtain running hardware for measurement and test purposes. Unfortunately, prototyping is a time-consuming process, and designers need to have first-cut estimates of power consumption early in the design flow. To support this requirement, a few component-based power estimation approaches have been developed. The tool of choice for back-of-the-envelope power estimation is a standard spreadsheet. To estimate the impact of a component on the power budget, its dissipation is extracted from the data sheets and entered in the spreadsheet. Total power is obtained by simply summing over all components. This technique, which is very straightforward, is widely used and is often the only option available to designers.

Powerplay, developed by Lidsky and coauthors [Lidski and Rabaey 1996] is a web-based spreadsheet that offers the opportunity for component exploration in a distributed environment. This tool offers a library of power models at several levels of accuracy, starting from a constant power value up to complex activity-sensitive models that require a fair amount of input information from the designer.

Example 3.9 Consider a simple general-purpose system, such as an electronic agenda. The system is built around a microcontroller with 1MB of RAM memory. 2MB of flash memory is used to store firmware. Additional 8MB of flash memory is used to store user data. A PC interface is available through an IR link (implemented by a single integrated component).

The main user interface is through an LC display, with a touch-sensitive surface that can be written with a stylus. The system also contains a real-time clock chip with a quartz oscillator, and a stabilized DC-DC converter to power the components. All components are powered at 3.3 volts.

The power consumption of all components is taken from data sheets and collected in a spreadsheet, shown in Figure 6. For each component, the worksheet reports (i) component count, (ii) voltage supply value, (iii) current absorbed when idle, (iv) current absorbed when active, (v) percentage idle time, (vi) percentage active time, and (vii) average current absorbed. The average current (in mA) is computed as $I = N_{components} \cdot (I_{idle} \cdot frac_{idle} + I_{on} \cdot frac_{on})$. The total current absorbed is reported in the lower right-hand corner. To compute average power consumption, we multiply I by V_{dd}. We obtain $P_{avg} = 276.6\text{mW}$.

The main limitation of spreadsheets is that they do not model interactions among components. Also, spreadsheets require substantial user intervention to take into account component utilization. Consider for instance the power consumed by memory, which is strongly dependent on access frequency. In a spreadsheet model, the designer must estimate expected memory utilization and use it to scale memory power. In other words, whenever power dissipation depends on the interaction of a component with the rest of the system, the spreadsheet model falls short.

Power state machines. To mitigate the shortcomings of spreadsheets, Benini et al. [1998c] developed an abstract state-based model for system components, called a *power state machine* (PSM). In a power state machine, states represent modes of operation, while arcs represent legal transitions between modes of operation. States are labeled with power dissipation values, transitions are labeled with *triggering events*, *energy costs*, and *transition times*. A triggering event is an external stimulus that forces a state transition.

A system description of this model is a structural network of system components. A power state machine and, optionally, a functional model are specified for each system component. The functional model is an abstract specification of component behavior, which specifies how to respond to external events, possibly generating new events. A system description can be simulated. An external event source, representing the environment, drives the model. Clearly, this abstract simulation bears little resemblance to a functional simulation, but it offers the opportunity of tracking the power states of system components.

When all state machines are single state and no events are generated, the PSM model becomes equivalent to the spreadsheet model. However, PSMs offer the possibility of (i) studying how the system reacts to different workloads; (ii) modeling interactions between components; and (iii) analyzing the effects of power management. The main limitation is that it requires more complex component models than a spreadsheet, and some

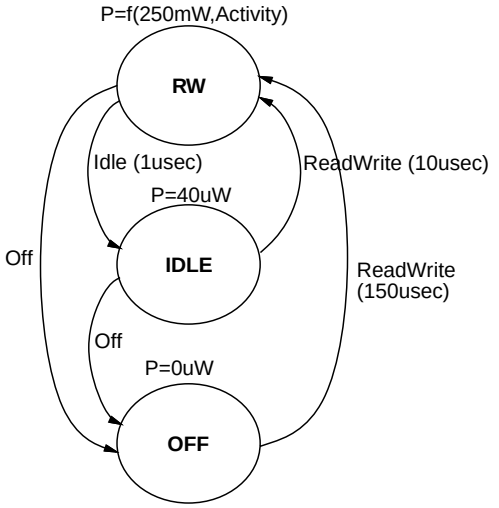


Fig. 7. Power-finite state machines for a memory component.

effort on the designer's part in describing the abstract specification of component behavior.

Example 3.10 The power state machine for a memory component is shown in Figure 7. There are three power states, namely "read/write" (RW), "idle," and "off." Each state is characterized by a power consumption level. Notice that in the active state, power is a function of memory activity (i.e., read/write traffic into memory). Edges are marked with external events that, when asserted, force the transitions. Edges are also labeled with transition times When the transition time is not reported, it is assumed to be instantaneous.

4. SYSTEM DESIGN

System modeling and conceptualization yield a hardware/software partition as well as a macro-architectural template and a set of procedures. *System design* consists of refining the hardware model of this template into computation, memory, and communication units. Different design flows, often based on computer-aided synthesis, support the mapping of these units into low-level detailed models ready for manufacture [De Micheli 1994]. When the template, as is most often the case, provides for programmable (core) processors to execute procedures, then software must be developed (or synthesized) and then compiled for the target cores.

The balance between software and hardware can vary widely, depending on the application and on the macro-architecture. Even though dedicated hardware is more energy efficient than software running on core processors, the latter has many compensating advantages, such as flexibility, ease of late debugging, low-cost, and fast design time. For these reasons,

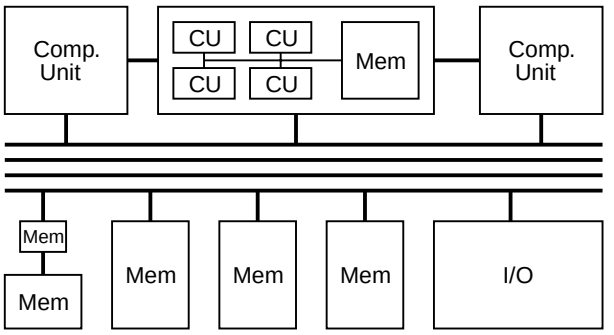


Fig. 8. Generic template for system hardware architecture.

core-based solutions are also present in energy-efficient system design. Nevertheless, most systems require some amount of hardware design.

This part of the tutorial is dedicated to the analysis of system design issues. We assume that conceptual analysis and modeling has been carried out. Therefore, the boundary between hardware and software and the macro-architectural template is set. In the next section, we focus on design of system hardware, with special emphasis on synthesis-based approaches. We consider only high-level system design techniques, and refer the reader to Macii et al. [1998]; Raghunathan et al. [1998]; Chandrakasan and Brodersen [1995]; and Rabaey and Pedram [1996] for a comparative analysis of methods for energy-efficient chip design (i.e., micro-architectural synthesis, logic synthesis, and physical design of ICs). We then report in Section 4.2 on software design and compilation. Needless to say, almost every conceivable system design flow entails both hardware and software synthesis, and it is often difficult to separate them.

4.1 Hardware Synthesis

The outcome of system modeling and conceptualization is a hardware template, which is the starting point for hardware design. Most systems can be described as instances of a basic template, shown in Figure 8. The figure depicts a heterogeneous hardware architecture containing several computation units, memory, and interconnects. Generality can be increased by allowing computation units to be hierarchical. Data processing and control are carried out by computation units, whose micro-architecture can range from fully application-specific to general-purpose. At one extreme, we have a dedicated hardware processor that performs a single function, with limited or no flexibility. At the other extreme, we just instantiate a core processor that can be programmed to perform any function. Several degrees of customization exist in between, depending on the granularity of the blocks employed for building the micro-architecture. We survey techniques for energy-efficient design and synthesis of computation units in Section 4.1.1.

Storage resources are usually organized as a *memory hierarchy*. Memories at low hierarchy levels are small, fast, energy efficient, and dedicated

to a single processor, or shared among a small number of them. Memory at high levels of hierarchy are large, relatively slow, and power hungry. The motivation for hierarchical memory organization is exploitation of temporal and spatial storage locality. Depending on the specific design flow, the degrees of freedom in memory hierarchy design can vary widely. At one extreme, we have a fixed hierarchy, where the only design parameter that can be controlled is memory size for some levels of the hierarchy. At the other extreme, we have a fully-flexible memory hierarchy. Low-power memory design techniques are surveyed in Section 4.1.2.

Communication channels are also hierarchical. Local communication has short range, and involves a reduced number of terminals (the most common case is two). Thus, it is fast and power efficient. Long-range global communication requires complex protocols, it is relatively slow and power inefficient. Nevertheless, it is required for supporting nonlocal information transfer. For simple macro-architectures, a single-level communication hierarchy is often employed. In complex architectures, communication resources can be organized in a hierarchical fashion. Energy-efficient communication channel design is analyzed in Section 4.1.3.

Even though we analyze computation, storage, and communication separately, it should be clear that it is impossible to design well-balanced systems by focusing on one of these areas only, and completely disregarding the others. Ideally, we would like to tackle all design issues at the same time, and achieve a globally optimum design. In most practical cases, however design quality benefits from focusing on just a few facets of the power optimization problem at one time, while keeping the remaining issues in the background. A well-balanced design is obtained by adopting a design flow where design challenges are prioritized and solved in sequence, possibly with iterations.

4.1.1 *Synthesis of Computation Units*. As mentioned above, data-processing units in an electronic system can be customized at different granularity levels. At the coarser granularity, we have processor cores. Cores are very successful in embedded system design because they reduce design turn-around time by moving all system customization tasks into the software domain. RISC processors are becoming very successful cores [Segars et al. 1995; Hasegawa et al. [1995], together with specialized processors for digital signal processing [Lee et al. 1997b; Mutoh et al. 1996; Verbauwhede and Touriguian 1998].

Cores are easy to use and very flexible, and sometimes they are necessary. However, they are not power efficient when compared with custom hardware solutions. Consider, for instance, a very simple computation such as FIR filtering. A core processor can easily implement a FIR filter (by performing a sequence of multiplications and additions). On the other hand, a custom hardware architecture for FIR filtering can be created with just an adder, a multiplier, and a few registers. The energies (or, equivalently, energy-delay products) of these two architectures can easily be more than two orders of magnitude apart (obviously, in favor of the custom

architecture). This is just an example of a fairly general principle, which can be summarized as an "energy-efficiency versus flexibility" tradeoff [Nishitani 1999]. Any micro-architecture of a computation unit can be seen as a point on a tradeoff curve of energy-efficiency versus flexibility, intended to be reconfigured for implementing new computational tasks. We explore this curve, starting from application-specific units and moving toward more flexible, and less power-efficient architectures.

Application-specific units. This solution is biased towards maximum power efficiency, at the price of reduced flexibility. Typically, design starts from an initial specification that is executable and given in a high-level language, with constraints on performance, power, and area. The outcome is a circuit that can perform only the specified function. The most straightforward design flow for these units is based on hand-crafted translation of the executable specification into a register-transfer level (RTL) description, followed by hand-design of the data path and RTL synthesis of the control path. To reduce energy consumption, designers can leverage a large number of low-power RTL, logic-level and physical design techniques, which have been surveyed in several papers: [Pedram and Vaishnav 1997; Pedram 1996] and monographs [Chandrakasan and Brodersen 1995; Rabaey and Pedram 1996; Benini and De Micheli 1997].

An alternative design flow relies on automatic synthesis of the custom computation unit starting from the executable specification. This approach is often called *behavioral synthesis* (or, more generically, *high-level synthesis*). In essence, the main difference between RTL synthesis and behavioral synthesis is that in the latter the order in the time for executing elementary operations (scheduling) is not taken from the specification. In other words, as long as the external performance constraints (latency, throughput, and synchronization) on the specification are met, the hardware synthesizer is free to alter the computation flow and to schedule elementary operations in time. Behavioral synthesis has been studied for more than fifteen years, and its basic theory is described in several textbooks [De Micheli 1994; Gajski et al. 1992].

Behavioral synthesis of energy-efficient circuits has been intensively studied as well. The interested reader is referred to the survey by Macii et al. [1998] and the monograph by Raghunathan et al. [1998] for an in-depth analysis of algorithms and tools, as well as an extensive bibliography. In this section we focus only on the distinctive characteristics of behavioral synthesis for low power. The optimization target is dynamic power consumption, which is the product of four factors, namely: (i) clock frequency f_{CLK}; (ii) the square of the supply voltage V_{dd}^2; (iii) the load capacitance C_L; and (iv) the average switching activity.

Probably the most direct way to reduce power is to scale down V_{dd}. Unfortunately, CMOS circuits get slower as V_{dd} decreases because CMOS transistors have smaller overdrive when they are on. Under these conditions, a good approach to reduce power consumption is to make a circuit faster than its performance constraint, then decrease the power supply

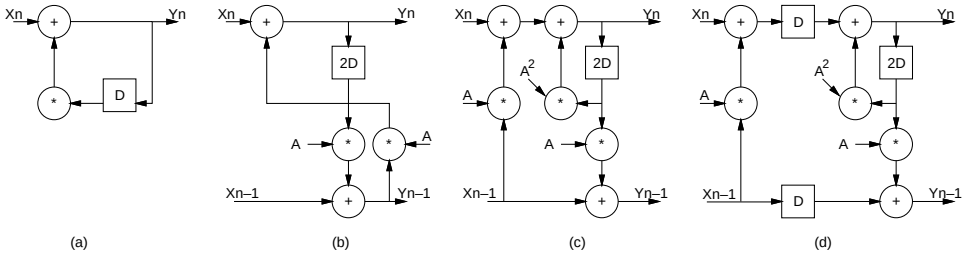


Fig. 9. Example of power-driven voltage scaling.

(obtaining a quadratic decrease in power) until the constraint is matched again. This *power-driven voltage scaling* approach was studied in depth by Chandrakasan et al. [1995] and Chandrakasan and Brodersen [1995]. One of its most interesting features is that it leverages well-established techniques for high-performance computation, such as parallelization, retiming, and pipelining to design energy-efficient circuits.

Example 4.1 An example of power optimization based on performance enhancement followed by voltage scaling is shown in Figure 9. The unoptimized specification is the data flow in Figure 9(a). Let us assume for simplicity that addition and multiplication by a constant take one time unit each. Thus, the critical path of the computation has length 2. First, loop unrolling is applied. The data-flow graph, unrolled once, is shown in Figure 9(b). The main purpose of this step is to enable speedup transformations. In Figure 9(c), constant propagation has been applied. Notice that both the number of operations performed every cycle and the critical path have increased with respect to (a). However, if we apply pipelining to the transformed data-flow graph, we reduce the critical path back to 2, but now we are processing two samples in parallel. The throughput has doubled. Hence, we can now scale down the voltage supply until delay has doubled, and we obtain a low-power implementation with the same throughput as the original specification. Voltage has been reduced from 5V to 2.9V, and power is reduced by a factor of 2.

The main limitation of power-driven voltage scaling is that it becomes less effective as technology scales down. In fact, fundamental reliability concerns are driving voltage supplies in modern submicron technologies to lower levels with each generation, reducing the headroom available for voltage scaling. Another issue is that most speedup techniques rapidly reach a region of diminishing returns when they are applied very aggressively. In other words, it may be easy to obtain good power saving by speeding up and scaling down voltage supply for circuits that are not optimized for performance, but further speeding up circuits that have already been optimized for performance is a challenging task, and may impose such complexity overhead as to be useless for power reduction.

Let us consider, for instance, two common speedup techniques: pipelining and parallelization. The point of diminishing returns for pipelining is reached due to pipeline register overhead. High-performance circuits are

often deeply pipelined, and the levels of logic between stages are in the 15 to 20 range. In these deep pipelines, a significant fraction of the cycle time budget is taken by register setup and hold times and by clock skew. Even if we disregard timing issues, we cannot arbitrarily increase pipeline depth because registers would occupy excessive area and load the clock too much.

Parallelization (i.e., the speedup technique that duplicates part of the data path to allow more than one computation at the same time) hits the region of diminishing returns even earlier than pipelining, due to duplication of data path structures, imposes a large area overhead and increases wiring by a substantial amount.

Furthermore, pipelining and parallelization are limited by the presence of control-flow constraints such as loops and conditionals. To speedup computation past control-flow bottlenecks, we need speculation (i.e., carry out a computation assuming that conditionals will follow the most probable path) or predication (i.e., perform the computation under both the exits of a conditional). These techniques are power inefficient because they increase the amount of redundant computation.

The basic power-driven voltage scaling can be extended by allowing multiple supply voltages on a single die. Various behavioral synthesis techniques have been proposed to deal with multiple supply voltages [Chang and Pedram 1997; Raje and Sarrafzadeh 1995; Johnson and Roy 1996]. The key idea in these approaches is to save power in noncritical functional units by powering them with a down-scaled supply voltage. In this way delays are equalized, throughput is unchanged, but overall power is reduced. Real-life implementation of multisupply circuits poses nontrivial technological challenges (multiple power distribution grids, level-conversion circuits, DC-DC conversion on chip). Nevertheless, the technological viability of multisupply circuits has been proven on silicon [Usami et al. 1998b]. Hence, computer-aided design tools supporting this design style may gain acceptance in the design community.

Alternative approaches to power reduction have tackled all other dependencies of dynamic power, namely, clock frequency, load capacitance, and reducing switching activity. Frequency reduction has sometimes been dismissed in the literature as ineffective because it reduces power, but not energy [Chandrakasan and Brodersen 1995] (a system clocked at a low frequency performs less "computational work" in a time unit). This conclusion is drawn under two implicit assumptions: (i) the entire system is clocked by a single clock; and (ii) the goal is to minimize average energy. If we invalidate one of these assumptions, frequency reduction can be an effective technique.

Let us consider the first assumption. If we allow multiple clock domains on a single system, we can clock noncritical subsystems at slower frequencies, thereby saving significant power without compromising overall system performance. Systems with multiple clock domains (also known as *globally asynchronous locally synchronous* or GALS), clocked at different rates, are becoming viable [Brunvand et al. 1999], pushed by the ever-increasing cost of global clock distribution grids and by the fundamental limitation of

global synchronicity posed by the finite propagation speed of electric signals [Dally and Poulton 1998]. The potential of multifrequency clocks for energy-efficient computation is still largely unexplored, with a few notable exceptions. In the work of Chung et al. [1995], a clock-distribution technique is proposed where a low-frequency clock is propagated on long wires and multiplied locally. In the work by Hemani et al. [1999], a few single-clock industrial designs are transformed in GALS, reporting power savings up to 70%.

Clock frequency can be varied over time as well. One example of dynamic clock speed setting is *clock-gating*. If a system component is idle (i.e., it is not performing any useful work), we can set its clock frequency to zero, and nullify dynamic power consumption. Clock-gating is widely applied in real-life circuits [Gonzalez and Horowitz 1996; Tiwari et al. 1998; Benini and De Micheli 1997], and it is often presented as a technique for reducing switching activity [Benini and De Micheli 1997]. In fact, clock frequency reduction has the beneficial side effect of reducing switching activity. Clearly, clock-gating is an extreme case, but we can envision applications where clock speed is dynamically reduced or increased depending on system workload. Dynamic power management techniques, including dynamic clock speed setting, are surveyed in Section 5.1.2.

Regarding the second assumption, several authors have observed that in portable systems the ultimate target is not average energy reduction, but battery lifetime extension. If we model batteries as large ideal capacitors, then energy reduction translates directly into longer lifetime. Unfortunately, the ideal capacitor model is inaccurate for real-life batteries. More specifically, several authors [Martin and Sewiorek 1996; Wolfe 1996; Pedram and Wu 1999] have shown that the amount of charge that can be drawn from a battery depends not only on its capacity (which is a function of battery mass and chemistry), but also on the rate of discharge. In first approximation, effective capacity decreases with an increasing discharge rate, hence extracting charge from a battery at a slow rate can maximize the amount of computational work performed during a battery's lifetime.

Example 4.2 Martin and Sewiorek [1996] analyzed the effect of battery discharge rate on capacity, moving from the empirical formula $C = K/I\psi$, where K is a constant determined by battery physical design and chemistry; I is the average discharge current and ψ is a fitting coefficient. If ψ is zero, then the battery is ideal (constant capacity). For real-life batteries, ψ ranges between 0.1 and 0.7, depending on battery type. Martin showed that, when considering this relationship, reducing clock frequency (and, consequencetly, I) can increase the total computational work that can be performed over a battery's lifetime.

Power can be reduced by reducing average load capacitance. This problem has been studied in depth, and it is usually tackled at the level of physical-design abstraction [Pedram and Vaishnav 1997; Pedram 1996], which is outside the scope of this survey. At the system level, we can

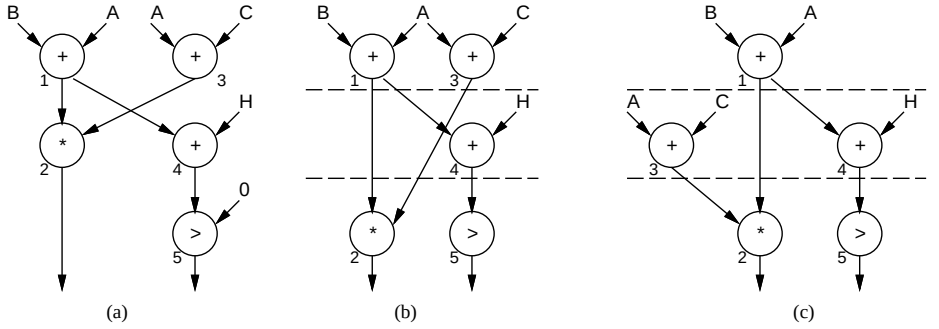


Fig. 10. An example of scheduling and binding for reduced switching activity.

observe that there is a tight relationship between locality and average load capacitance. To understand this statement, observe that an architecture with good locality minimizes global communication, which is usually carried over long global wires with high capacitance load. This observation is the basis of the work by Mehra et al. [1996; 1997] on high-level partitioning for wire-load reduction.

Finally, we consider techniques for reducing average switching activity. At first this seems the most straightforward way to achieve this goal is to maximally reduce the number of basic operations. Srivastava and Potkonjak [1996] proposed a technique for transforming the data-flow graph of linear computation in such a way that the number of operations is minimized. This technique was later extended by Hong et al. [1997] to general data flows. It is important to see that minimizing the number of operations does not guarantee minimum power, since in general the power dissipated in executing an elementary operation depends on the switching activity of the operands.

Commonly-used arithmetic units dissipate more power when their inputs have high switching activity. We can reduce switching by increasing the correlation between successive patterns at the input of a functional unit. A few CDFG transformation techniques for facilitating the identification of sequences of operations with low switching activity have been presented in the past [Chandrakasan et al. 1995; Kim and Choi 1997; Guerra et al. 1998]. After CDFG transformations, behavioral synthesis transforms the CDFG into an RTL netlist. All behavioral synthesis steps (allocation, scheduling, and binding) have been modified to take into account switching activity (see Macii et al. [1998] for a survey), and behavioral synthesis of circuits with low switching activity is an area of active research. A prerequisite for the applicability of these approaches is some form of knowledge on input statistics for the hardware unit that is being synthesized.

Example 4.3 Consider the simple data-flow graph shown in Figure 10(a). The constraint on execution time is 3 clock cycles, the resource constrains are 2 adders, 1 multiplier, and one comparator. Figure 10(b) and (c) show two schedules compatible with resource constraints. Observe that

additions 1 and 3 share one of the operands. If we perform both additions with the same adder, its average switching per operation will be low, because one of the operands remains the same, and will dissipate less power than when performing two additions with uncorrelated inputs. In the schedule in Figure 10(b), however, it is not possible to implement additions 1 and 3 with the same adder, because the two operations are scheduled in the same control step. On the other hand, the schedule in Figure 10(c) does allow sharing, and it leads to a more energy-efficient implementation.

This simple example shows that in order to minimize power, allocation, scheduling, and binding algorithms must work synergistically. It is mainly for this reason that most behavioral synthesis approaches in the literature are based on iterative improvement heuristics [Chandrakasan et al. 1995; Kumar et al. 1995; San Martin and Knight 1996; Raghunathan and Jha 1997]. The main challenge in implementing effective optimization tools based on iterative improvement is estimating the power cost metric. Estimating power at the behavioral level is still an open problem for many aspects of implementation. Probably one of the most pressing issues is the mismatch between zero-delay switching activity and actual switching activity that may lead to macroscopic underestimates of power consumption [Raghunathan et al. 1999].

Another class of techniques for energy-efficient behavioral synthesis exploits control flow and locality of computation. Almost any real-life application does contain some control flow. Control flow forces mutual exclusion in computation and unequal execution frequencies that can be exploited to reduce power consumption. If we evaluate a conditional before we start the execution of its branches, we can easily detect idleness in functional units that execute operations in the branch that is not taken only. When idleness is detected early enough, we can eliminate switching activity by freezing (with clock gating or similar techniques) the idle units [Monteiro et al. 1996]. We can push this concept even further. If by profiling and control-flow analysis, we can determine the most frequently executed sections of a specification (i.e., the computational kernels), we can optimize the circuit for the common cases, provided that, at execution time, we can detect the occurrence of the common cases early enough [Lakshminarayana et al. 1999].

Example 4.4 Consider the simple code fragment shown in Figure 11(a). Assume that the profiling analysis has revealed that most of the time the value of c is 2. We can then transform the original code into the equivalent form shown in Figure 11(b). At a first sight, the second form is less compact and redundant. However, observe that in the new specification the branch of the conditional that executes more frequently contains just a shift operation that is much more power efficient than multiplication and subtraction, as needed in the infrequent path. Hence, by testing the condition $c == 2$, we can disable the circuitry implementing the general computation, and just perform the power-efficient common-case computation.

```
while (c > 0) {
  A = A * c;
  c--;
}
```

(a)

```
if (c == 2)  A = A << 1
else {
  while (c > 0) {
    A = A * c--;
    c--;
  }
}
```

(b)

Fig. 11. Common-case computation and exploitation of mutual exclusion.

Optimization that is based on mutual exclusion and the common case often has area overhead because both methods limit hardware sharing. Hence, they can be counterproductive if the circuit operates with input patterns that completely change the probability distribution of the statements in the specification.

Application-specific processors. Even though dedicated computation units are very energy efficient, they completely lack flexibility. In many system design environments, flexibility is a primary requirement, either because initial specifications are incomplete or because they change over time. Also, many systems are designed to be reprogrammable in the field. Computation units for these systems must be programmable to some degree.

When flexibility is a primary concern, a stored program processor is the micro-architecture of choice. Besides flexibility, processors offer a few distinctive advantages. First, their behavior (i.e., the execution of a sequence of instructions) is well-understood and matches the procedural specification style very well. Second, they offer a clean hardware abstraction (the instruction set architecture, ISA for short) to the designer. Third, they leverage well-established compilation and debugging technologies. Finally, their interfaces with the rest of the system (i.e., other processors and memories) are often standardized, thereby maximizing opportunities for reuse.

From the energy-efficiency viewpoint, processors suffer from three main limitations. First, they have an intrinsic power overhead for instruction fetch and decoding, which is not a major issue for computation units with hardwired control. Second, they tend to perform computations as a sequence of instruction executions, with the power overhead mentioned above, and cannot take full advantage of algorithmic parallelism. Finally, they can perform only a limited number of elementary operations, as specified by their ISA. Thus, they must reduce any complex computation to a sequence of elementary operations.

Interestingly, these limitations not only affect power adversely, but also decrease performance. Hardware architects designing general-purpose processors have struggled for many years with the performance limitations of basic processor architecture, and have developed many advanced architectures with enhanced parallelism. When designing a processor for a specific

application, energy (and performance) optimization can leverage the knowledge of the target application to obtain a highly optimized specialized processor. This observation has lead to the development of a number of techniques for *application-specific instruction-set processor* (ASIP) synthesis [Goossens et al. 1997].

ASIPs are a compromise solution between fixed processor cores and dedicated functional units. To a large degree, they maintain the flexibility of general-purpose processors, but are tailored to a specific application. When they run their target application, they are substantially more power efficient than a general-purpose core. Traditionally, ASIPs have been developed for performance and silicon area; research on ASIPs for low power has recently just started.

Instruction subsetting proposed by Dougherty et al. [1998] aims at reducing instruction decoding and micro-architectural complexity by reducing the number of instructions supported by an ASIP. The basic procedure for instruction subsetting can be summarized as follows: The target application is first compiled for a *complete* ISA, then the executable code is profiled. Instructions that are never used or instructions that can be substituted by others with a small performance impact are dropped from the application-specific ISA. Then, the processor with a reduced ISA is synthesized, and the code.

Another technique, proposed by Conte et al. [1995] exploits profiling information to tailor a parameterized superscalar processor architecture to a single application (or an application mix). Parameters such as number and type of functional units and register file size can be explored and optimized. The optimization loop is based on iterative simulation of a micro-architecture model under the expected workload. Notice that all architectures that can be generated by the optimization have the same ISA, hence the executable code does not need to be recompiled. A similar technique is proposed by Kin et al. [1999] for exploring the design of application-specific VLIW machines. In this case, a retargetable compiler is customized whenever a new architecture is examined, and the code must be recompiled before instruction-level simulation.

As a concluding remark for this section, we want to stress the fact that both application-specific unit synthesis and general-purpose processor optimization for low power are at a much more advanced research stage than application-specific processor optimization. Commercial tool support for power optimization of computation units is still restricted to low levels of abstraction (gate-level and, to some degree, register-transfer level). We believe that many interesting research opportunities exist in the area of application-specific energy-efficient processor synthesis.

Core processors. Core processors are widely used as computation units in both embedded and general-purpose systems. Even though processors are inherently less power efficient than specialized units, they can be optimized for low power. Energy-efficient processor design is covered in detail in the papers by Gonzalez and Horowitz [1996] and by Burd and

Brodersen [1996]. We briefly mention a few basic ideas that have been widely applied in designing low-power processors.

Probably the most commonly used power-reduction technique for processors is aggressive voltage scaling. Many processor families have "low-power" versions with reduced supply voltage [Gary et al. 1994; Debnath et al. 1995; Hasegawa et al. 1995; Furber 1997; Segars et al. 1995; Gowan et al. 1998]. Needless to say, supply voltage down-scaling often requires some adjustments in (i) device technology (e.g., lower threshold voltage); (ii) circuit design (e.g., transistor stack lowering); and (iii) micro-architecture (e.g., critical path redesign). Lowering supply voltage does not give a durable competitive advantage because most processor vendors tend to align themselves on the same technology and supply voltage. In contrast, a good low-power micro-architecture design can provide a durable competitive advantage across many technology generations. The energy efficiency of a micro-architecture can be enhanced by avoiding useless switching activity in idle units (e.g. a floating point unit when the processor is executing integer code). Power waste in idle units is reduced by clock gating and operand isolation.

High performance in processor design is often achieved at a substantial price in power. For instance, speculative techniques (in execution and memory access) [Hennessy and Patterson 1996] are often detrimental energy-wise because they do not boost performance enough to compensate for the increased power dissipation. Hence, some low-power processor designers have opted for relatively simple micro-architectures that do not provide top performance, but are more energy-efficient than their high-performance counterparts [Segars et al. 1995]. Care must be taken, however, in considering the target application for a processor. In many domains, energy (i.e., the power-delay product) is not an acceptable metric because the system needs fast response time. In these cases the energy-delay product is a better metric for comparing processors. Gonzalez and Horowitz showed that most low-power processors are not better than high-performance CPUs when comparing their energy delay products.

Ideally, we would like to reduce power and boost performance at the same time. Specialized instructions are often proposed as a power and performance enhancement technique. The basic idea is to provide a few specialized instructions and the required architectural supports that allow the processor to execute very efficiently under some specific workload. Subword parallel instructions [Ishihara and Yasuura 1998b], special addressing modes [Kalambur and Irwin 1997], and native multiply-accumulate instructions [Litch and Slaton 1998] are just a few examples of domain-specific instructions that can reduce power and increase performance when the processor is executing data-dominated applications. Digital signal processors [Mutoh et al. 1996; Lee et al. 1997b; Verbauwhede and Touriguian 1998] carry this concept to the extreme, providing highly specialized instruction sets. The main problem with this approach is that it is very hard to design compilers that fully exploit the potential of special-

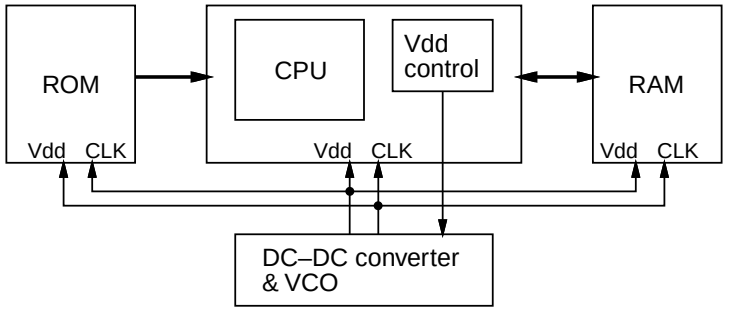


Fig. 12. A variable-voltage processor-based architecture.

ized instructions. In many cases, slow and error-prone hand-coding in assembly is required.

An emerging low-power processor design technique is based on dynamically variable voltage supply [Nielsen et al. 1994; Suzuki et al. 1997; Ishihara and Yasuura 1998b]. A variable-voltage processor can dynamically adapt performance to the workload. The basic variable-voltage architecture is shown in Figure 12. When the supply voltage is lowered, the processor clock must be slowed down, hence we need a variable frequency-clock generator that tracks the changes in circuit speed with voltage supply. This is done by employing an oscillator whose period is proportional to the propagation of a dummy chain of gates that replicates the critical path of the processor. With a variable-voltage processor we can in principle implement just-in-time computation, i.e., lower the voltage until the processor is barely fast enough to meet performance constraints. In practice, variable-voltage processors are faced with many design and validation issues such as power-efficient variable-voltage supply design [Sratakos et al. 1994; Gutnik and Chandrakasan 1997]; precise and fast voltage control and clock tracking [Wei and Horowitz 1996; Namgoong et al. 1997], and power grid reliability.

4.1.2 *Design of Memory Subsystems*. Storage is required to support computation. In the early days of digital computing, researchers focused on memory size optimization. Memory was expensive, and memory space was a scarce resource. The fast pace of semiconductor technology, which has increased the level of integration exponentially with time, has completely changed this picture. Nowadays, the cost per memory bit is extremely low, and sheer memory size is rarely the main issue. Memory performance and power are now the key challenges in system design. The simplest memory architecture, the *flat memory*, assumes that every datum is stored in a single, large memory. Memory accesses become slower and consume more power with increasing memory size. Hence, memory power and access time dominate total power and performance for computations with large storage requirements, and memory becomes the main bottleneck [Lidsky and Rabaey 1994; Catthoor et al. 1998a].

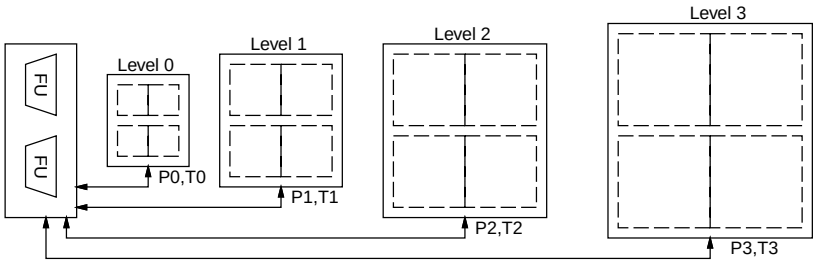


Fig. 13. A generic hierarchical memory model.

One obvious remedy to the memory bottleneck problem is to reduce the storage requirements of the target applications. During system conceptualization and algorithmic design, the *principle of temporal locality* can be very helpful in reducing memory requirements. To improve locality, the results of a computation, should be "consumed" by subsequent computations as soon as possible, thereby reducing the need for temporary storage. Other memory-reduction techniques strive to find efficient data representations that reduce the amount of inessential information stored in memory. Data compression is probably the best-known example of this approach.

Unfortunately, storage reduction techniques cannot completely remove memory bottlenecks because they also try to optimize power and performance indirectly by reducing memory size. As a matter of fact, memory size requirements have increased steadily. Designers have tackled memory bottlenecks following two pathways: (i) power-efficient technology and circuit design; (ii) advanced memory architectures that overcome the scalability limitation of flat memory. Technology and circuit design are outside the scope of this survey, the interested reader is referred to the survey by Itoh et al. [1995]). We focus on memory architectures.

All advanced memory organizations rely on the concept of *memory hierarchy*. A large number of excellent textbooks have been written on organizing memory, which has become a fundamental part of any system macro-architecture [Hennessy and Patterson 1996]. In this section we present a general hierarchical memory model that will be used as a frame for analyzing a number of memory optimizations for low power.

A hierarchical memory model. The generic memory model used in the following analysis is shown in Figure 13. Various memory organizations can be seen as specializations of the structure in Figure 13. The model is hierarchical. Low hierarchy levels are made of small memories, close to computation units, and tightly coupled with them. High hierarchy levels are made of increasingly large memories, far from computation units, and shared. The generic terms "close" and "far", imply a notion of distance. Roughly speaking, the *distance* between a computation unit and a memory hierarchy level represent the effort needed to fetch (or store) a given amount of data (say, a byte) from (to) the memory. Effort can be expressed in units of time—if we are interested in performance—or in terms of

energy. The labels in Figure 13 represent the effort associated with each memory level, which includes not only the cost for memory access, but also the cost for transferring the data through the hierarchy.

Each memory level can be partitioned into several independent blocks (called *banks*). This partition may help to reduce the cost of a memory access in a given level. On the other hand, memory sub-banking increases addressing complexity and has a sizable area overhead. Both these factors reduce the power savings that can be achieved by memory partitioning. Finding an optimal memory partition in every memory level is another facet of the memory-optimization problem.

Example 4.5 As a concrete example of a memory architecture that can be modeled with the template of Figure 13 is shown in Figure 14, taken from the paper by Ko and Balsara [1998]. The hierarchy has four levels. Three levels of cache are on the same chip as well as the execution units. Four-way associativity is assumed for all caches. To get high cache-hit rates, the line sizes at upper levels are bigger than those at lower levels. L0 has 16 bytes lines, L1 has 32 bytes, and L3 has 64 bytes. Similarly, cache size increases with level. L0 ranges from 1 to 16 KB, L1 from 4 to 64 KB, and L2 from 16 to 1024 KB. It is assumed that caches are fully inclusive (i.e., every location contained into a low-level cache is contained in all higher-level caches).

The last level of memory hierarchy is off-chip DRAM, organized in banks, with up to 1024 MB for each bank when the bank is fully populated. The DRAM access is routed through a memory control/buffer that generates row/column address strobes (RAS/CAS), addresses, and controls sequencing for burst access. To allow nonblocking external memory access, data from DRAM is routed through a data-path control buffer that signals the processor when the data from DRAM is available. Average power for accessing a L0 cache at 100MHz is approximatively 150mW; power for L1 cache access is 300mW; and power for L2 access is 700mW. Average power of a burst transaction to external DRAM is 12.71W, which is more than two orders of magnitude larger than the power for accessing L0 cache.

The main purpose of energy-efficient design is to minimize overall energy cost for accessing memory within performance and memory size constraints. Hierarchical organizations derived from the generic template of Figure 13 reduce memory power by exploiting the nonuniformities in access frequencies. In other words, most applications access a relatively small area in memory with high frequency, while most locations are accessed a small number of times. In a hierarchical memory, frequently-accessed locations should be placed in low hierarchy levels, thereby minimizing average cost per access.

Approaches to memory optimization in the literature can be grouped into three classes: *Memory hierarchy design* belongs to the first class. It assumes a given dynamic trace of memory access, obtained by profiling an application, and produces a customized memory hierarchy. The second class of techniques, called *computation transformations for memory optimi-*

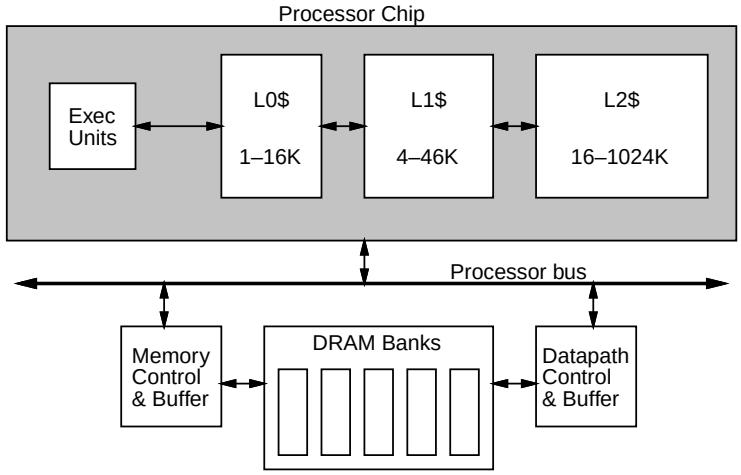


Fig. 14. A four-level hierarchical memory model with on-chip caches and off-chip DRAM.

zation, assumes a fixed memory hierarchy and tries to modify the storage requirements and access patterns of the target computation to optimally match the given hierarchy. Finally, the third approach, called *synergic memory and computation optimization* is, in principle, the most powerful, since it tries to concurrently optimize memory access patterns and memory architecture.

Memory hierarchy design. Several authors have analyzed the power dissipation of different memory architectures for a given application mix. These studies specify a core processor and an application (or an application mix), and they explore the memory hierarchy design space to find the organization that best matches the processor and application. Su and Despain [1995]; Kamble and Ghose [1997]; Ko and Balsara [1998]; Bahar et al. [1998]; Shiue and Chakrabarti [1999] focus on cache memories. Zyuban and Kogge [1998] study register files; Coumeri and Thomas [1998] analyze embedded SRAMs; Juan et al. [1997] study translation look-aside buffers.

The design space is usually parameterized and discretized to allow exhaustive or near-exhaustive search. Most research efforts in this area postulate a memory hierarchy with one or more levels of caching. A finite number of cache sizes and cache organization options are considered (e.g., degree of associativity, cache replacement policy, cache sub-banking). The best memory organization is obtained by simulating the workload for all possible alternative architectures.

Example 4.6 The memory organization options for a two-level memory hierarchy (on-chip cache and off-chip main memory) explored in the paper by Shiue and Chakrabarti [1999] are (i) cache size, ranging from 16 bytes to 8KB, in powers of two; (ii) cache line size from 4 to 32, in powers of two; (iii) associativity (1, 2, 4, and 8); (iv) off-chip memory size, from 2Mbit SRAM to 16Mbit SRAM.

The main limitation of the explorative approach is that it requires extensive data collection, and this only provides *a posteriori* insight. In order to limit the number of simulations, only a relatively small set of architectures can be tested and compared. This approach is best suited for general-purpose systems, where the degrees of freedom on the macro-architecture are reduced and where the uncertainty of the application mix imposes the need for robust memory architectures that adapt fairly well to different workloads. Caches provide hardware-supported capability for dynamically adapting to time-varying program work sets, but at the price of increased cost-per-access.

When comparing time and energy per access in memory hierarchy, we observe that these two cost metrics often behave similarly, namely, they both increase in large increments as we move from low to high hierarchy levels. Hence, we may conclude that a memory architecture that performs well also has low power consumption, and optimizing memory performance implies power optimization. Unfortunately, this conclusion is often incorrect for two main reasons. First, even though both power and performance increase with memory size and memory hierarchy levels, they are not guaranteed to increase in the same way. Second, performance is a worst-case quantity (i.e., intensive), while power is an average-case quantity (i.e., extensive). Thus, memory performance can be improved by removing a memory bottleneck on a critical computation, but this may be harmful for power consumption, since we need to consider the impact of a new memory architecture on all memory accesses, not only the critical ones.

Example 4.7 Shiue and Chakrabarti [1999] explored cache organization for maximum speed and minimum energy for MPEG decoding (within the design space bounds described in the previous example). Exhaustive exploration resulted in an energy-optimal cache organization with a cache size of 64 bytes, a line size of 4 bytes, and an 8-way set associative. Note that this is a very small memory size , almost fully associative (only two lines). For this organization, total memory energy is 293 μJ and execution time is 142,000 cycles. In contrast, the best performance is achieved with a cache size of 512 bytes, line size of 16 bytes, and an 8-way set associative. Notice that this cache is substantially larger than the energy-optimal one. In this case execution time is reduced to 121,000 cycles, but energy becomes 1,110 μJ.

A couple of interesting observations can be drawn form this result. First, the second cache dominates the first for size, line size, and associativity, hence it has a larger hit rate. This is consistent with the fact that performance strongly depends on the miss rate. On the other hand, if external memory access power is not too large with respect to cache access (as in this case), some of the hit rate can be traded-off for decreased cache energy. This justifies the fact that a small cache with a large miss rate is more power efficient than a large cache with a smaller miss rate.

Within each memory hierarchy level, power can be reduced by memory-partitioning techniques [Farrahi et al. 1995; Farrahi and Sarrafzadeh

1995]. The principle in partitioning memory is to sub-divide the address space into many blocks and to map blocks to different physical memory banks that can be i enabled and disabled ndependently. Power for memory access is reduced when memory banks are small. On the other hand, an excessively large number of small banks is highly area inefficient, and imposes a severe wiring overhead, which tends to increase communication power. For this reason, the number of memory banks should always be constrained.

When the computation engine is a processor, we need to consider the consumption of instruction memory power. The cost of instruction memory fetches is not negligible. During program execution, the processor needs to be fed with at least one instruction per clock cycle. Parallel processors, such as superscalar or very-long-instruction-word cores, fetch more than one instruction per clock cycle. Several energy optimization techniques have been developed for instruction memories. *Predecoded instruction buffers* and *loop caches* were studied in detail [Bajwa et al. 1997; Kin et al. 1997]. These techniques exploit the strong locality of reference for instruction flow. In most programs, a large fraction of execution time is spent in a very small section of executable code, namely in a few critical loops. Predecoded instruction buffers store instructions in critical loops in a predecoded fashion, thereby decreasing both fetch and decode power. Loop caches store most-frequently-executed instructions and can bypass even the first-level cache. Compared with predecoded instruction buffers, loop caches are less energy efficient for programs with very high locality, but are more flexible.

The instruction memory bandwidth (and power consumption) can be reduced by instruction compression techniques [Yoshida et al. 1997; Liao et al. 1998; Lekatsas and Wolf 1998; Benini et al. 1999a] that aim at reducing the large amount of redundancy in instruction streams. Several schemes were proposed to store compressed instructions in main memory and decompress them on-the-fly before execution (or when they are stored in the instruction cache). All these techniques trade-off the aggressiveness of the compression algorithm for the speed and power consumption of a hardware decompressor. Probably the best-known instruction compression approach is the "Thumb" instruction set of the ARM microprocessor family [Segars et al. 1995]. ARM cores can be programmed using a reduced set of 16-bit instructions (an alternative to the standard 32-bit RISC instructions) that reduce required instruction memory occupation and bandwidth by a factor of 2.

Computation transformations for memory optimization. If we assume a fixed memory hierarchy, optimization of memory energy consumption can only be carried out by modifying the memory accesses required to perform a computation. In processor-based architectures, this task relies on source code optimization techniques, which are surveyed in Section 4.2. At an even higher level of abstraction than source code, memory power can be minimized by judicious selection of data structures [Wuytack et al. 1997; Da Silva et al. 1998]. To allow a systematic exploration of different

data-structures, a library-based approach is adopted. A set of standard data structures suitable for a class of applications are developed and collected in a software library. For a given application, the best data structure is selected by exploring the library.

In application-specific computation units, memory power can be reduced by careful assignment of variables (i.e., intermediate results of computations) to memory elements. This task is known as *register allocation* in behavioral synthesis jargon, since traditional behavioral synthesis algorithms assumed that variables are stored exclusively in registers.

Gebotys [1997] proposed a power-minimization approach to simultaneous register and memory allocation in behavioral synthesis. The allocation problem is formulated as a minimum-cost network flow that can be solved in polynomial time. The rationale behind this approach is to map variables that have nonoverlapping lifetimes but, at the same time, similar values (i.e., a small average number of bit differences to the same register). The techniques by Gebotys are only applicable to pure data-flow graphs; they were extended by Zhang et al. [1999] to general CDFGs.

The register and memory allocation techniques proposed by Gebotys and Zhang are applied after scheduling. Hence, the order in time of operations and the lifetime of variables have already been decided. Clearly, scheduling can impact variable lifetimes and, as a consequence, memory requirements. Scheduling for reducing memory traffic in behavioral synthesis was studied by Saied and Chakrabarti [1996], who describe two scheduling schemes under fixed hardware resource constraints that reduce the number of memory accesses by minimizing the number of intermediate variables that need to be stored.

Another degree of freedom that can be exploited for minimizing memory power during behavioral synthesis is the allocation of arrays. Large arrays are usually mapped mapped to off-chip memories. Panda and Dutt [1999] proposed a power-per-access reduction technique that minimizes transition count on memory address busses. Arrays are mapped to memory trying to exploit regularity and spatial locality in memory accesses. The authors describe array-mapping strategies for two memory architectures: (i) a single off-chip memory; (ii) multiple memory modules drawn from a library. In the first case, they formulate a heuristic that reduces access power for each behavioral array. For mapping into multiple memory modules, the problem is partitioned into three logical-to-physical memory mapping subtasks.

Memory mapping for special multibanked memory architectures in digital signal processors has been studied by Lee et al. [1997a]. The technique proposed by Lee exploits the power efficiency of a specialized instruction of a DSP architecture that can fetch two data in the same cycles from two parallel RAM banks. The beneficial impact of this instruction is maximized by mapping independent arrays that can be accessed in parallel, in a mutually exclusive fashion, to the two RAM banks.

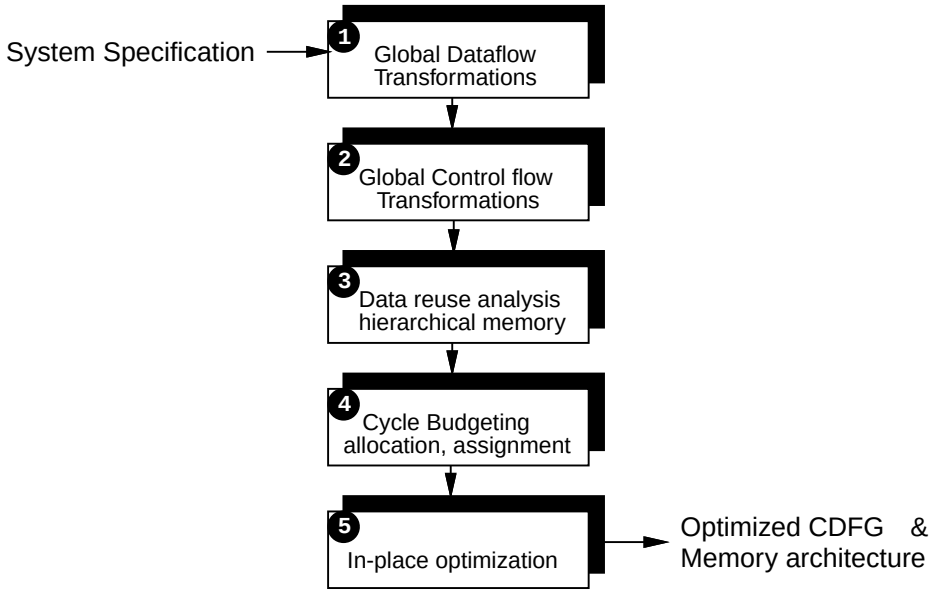


Fig. 15. Optimization flow in the DTSE methodology.

Synergistic memory and computation optimization. It is intuitively clear that the best memory optimization results can be obtained by synergistically optimizing the structure of a computation and the memory architecture that supports it. The synergistic approach is adopted by Catthoor and coworkers at IMEC, who focus on data memory optimization for data-intensive embedded applications (such as networking and multimedia) [Catthoor et al. 1994; Nachtergaele et al. 1998; De Greef et al. 1998; Wuytack et al. 1998; Kulkarni et al. 1998]. The techniques proposed by this research group are integrated into a custom memory management methodology called *data transfer and storage exploration* (DTSE) [Catthoor et al. 1998b; 1998a]. The fundamental premise for the application of this methodology is that in data-dominated applications the power consumption related to memory transfer and storage dominates overall system power. Hence, memory optimization should be the top priority, starting with system specification and moving down the design flow.

DTSE methodology is summarized in the diagram of Figure 15, which is a simplified version of the flow described in Catthoor et al. [1998a]. The emphasis on memory optimization drives the choice of an *applicative* specification style, where arrays are specified and manipulated in an abstract fashion and pointers are not allowed. In contrast, traditional procedural constructs tend to severely limit the degrees of freedom in memory exploration. For instance, specifications using memory pointers are notoriously hard to analyze and optimize because of the pointer aliasing problem [Aho et al. 1988]. The starting point of the DTSE flow is an executable specification with accesses to multidimensional array structures (called *signals* in Catthoor et al. [1998a]) with a single thread of

control. The output is a memory hierarchy and address-generation logic, together with an optimized executable specification where memory accesses are optimized for memory architecture. The optimized specification can then be fed to behavioral synthesis (when targeting application-specific computational units), or to software compilation.

The first step of DTSE applies a set of *global data-flow transformations* to the initial specification—for example, advanced substitutions across branches, computation reordering based on associative or distributive properties, and replacing storage with recomputation. This step is based on direct designer intervention, and no automatic support is available. In the second step, *global loop and control-flow transformations* are applied. This step is partially automated: the designer can choose from a set of loop transformations and apply them on selected loops. Basic supported transformations include loop interchange, reversal, splitting, and merging [Muchnick 1997].

While the first and second steps operate exclusively on the specification, the third step, *data-reuse analysis for hierarchical memory* begins to define memory hierarchy. The memory transfers generated by the specification are analyzed, clustered, and partitioned over several hierarchical memory levels. If multiple copies of some data need to be stored (for instance, if low-level memories are buffers that need to be reused), the required data copy instructions are inserted automatically. The main purpose of this step is to find a memory organization that optimally trades-off data replication and copying with good storage locality for frequently accessed memory locations.

Once the memory hierarchy is shaped, the DTSE flow proceeds to create a detailed memory architecture (and the assignment of data to memory) to minimize memory costs while guaranteeing performance (i.e., a cycle budget for the execution). This step consists of three substeps, namely *storage cycle distribution*, *memory allocation*, and *memory assignment*. It includes estimates of memory cost, speed, and power. The last step in DTSE is *in-place optimization* that carefully analyzes and optimizes access patterns in multiple nested loops to maximally reduce memory traffic.

It is important to stress that both memory architecture and executable specifications are not fully designed after the application of DTSE, but they are ready for hardware synthesis and/or software compilation. The main purpose of the methodology is to provide a specification that, when synthesized into silicon, is very energy efficient (as far as memory power is concerned).

Example 4.8 As an example of DTSE flow, consider the optimization of the simple specification shown in Figure 16(a), which implements FIR filtering of a finite sequence. The input sequence has length N and is stored in array $v[i]$, the filter coefficients are stored in array $f[i]$ (of size $M \ll N$), and the output sequence is stored in array $r[i]$. Notice that the last elements of $r[i]$ are computed, "wrapping around" the input sequence. To implement the wrap-around, the $v[i]$ array is extended to size $N + M - 1$,

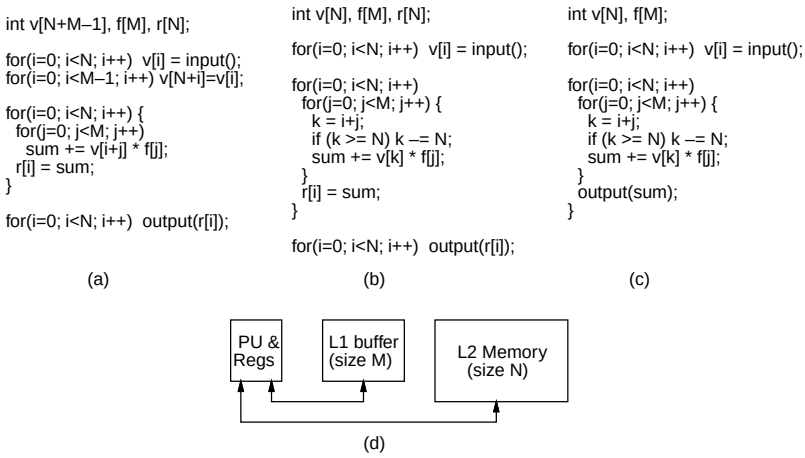


Fig. 16. Example of an application of DTSE methodology.

and its first $M - 1$ elements are copied into the $M - 1$ additional elements at the end of the array.

A simple global data-flow transformation applied to the initial specification is shown in Figure 16(b). Instead of enlarging the array, a conditional test is applied on the index. When index k becomes larger than the array boundary, it is "wrapped around." As a result, memory use is reduced by $M - 1$. This is an example of how additional computation can be traded-off for reduced memory use. An example of global control-flow transformation is shown in Figure 16(b). We observe that the only use of array $r[i]$ is the last loop. Hence, the array and the loop can be removed. This step reduces memory usage by N. In data reuse analysis, we observe that each element array $f[i]$ is accessed N times, hence it is a good candidate for storing in a small and power-efficient access buffer. For this reason, a three-level memory hierarchy is instantiated as shown in Figure 15(d), with a large background memory for storing $v[i]$, a small buffer for storing $f[i]$, and registers within the execution unit to store scalar variables. Finally, in the in-place optimization step, the access patterns to the arrays are analyzed. It is observed that, at any given time during execution, only M elements of $v[i]$ are needed. After an element of $v[i]$ has been used N times, it can be discarded. Hence, we do not need to read all $v[i]$ in advance and to store it in memory. We can further reduce $L2$ memory size by $N - M$.

DTSE is a very complete methodology proven to be effective in a number of case studies. However, it is still unclear how much of DTSE can be automated effectively. Many transformations in DTSE appear to be more hand-crafted code optimizations than algorithmic power-reduction techniques. Two approaches to automated memory optimization were proposed by Li and Henkel [1998] and by Kirovski et al. [1998], consisting of a simple architecture based on a single core processor, one level of data, an instruction cache, and main memory. Given an executable memory, optimization is

performed in two steps. In the first step, a restricted set of code transformations is applied to the executable specification, creating a pool of possible solutions. A number of memory architectures are evaluated for each possible solution. The most power-efficient optimized code and memory architecture pairs are then selected. The selection process is based on heuristic search strategies, such as steepest descent or genetic algorithms. To improve convergence, the design space is pruned using a set of heuristics based on dominance relationships and constraints. In contrast to DTSE, these approaches explore a much reduced design space, but are fully automated.

Furthermore, DTSE targets data memory exclusively. The power consumed by instruction memory, when the computation is performed by a processor, can also benefit from synergic approaches that join specialized memory architectures with program optimization. An interesting technique for instruction memory power optimization was proposed by Panwar and Rennels [1995], and later improved by Bellas et al. [1998]. This technique aims at improving the effectiveness of instruction buffers. To optimally exploit the buffer, most frequently executed basic blocks are placed into memory locations that are statically mapped into the buffer address space. This requires code reordering and the insertion of some unconditional branches in the original code. A similar basic block reordering technique, targeting improved instruction cache locality, was proposed by Kirovski et al. [1998].

4.1.3 *Design of Communication Resources*. Communication among system components is always required to carry out any meaningful task. Communication energy is a fundamental and unavoidable part of the overall energy budget of a system. The main purpose of this section is to analyze the energy costs of communication and survey techniques for energy-efficient design. Consistent with previous sections, we focus on electronic systems where all components are close to each other (on the same chip or board).

One of the best proofs of the fast progress of semiconductor technology is that computation speed (which is tightly related to device speed) has increased at an impressive rate, but communication speed has not scaled accordingly. In other words, the relative impact of communication on overall system performance has steadily increased over time [Dally and Poulton 1998]. The same holds for communication energy, which is taking an increasingly large fraction of the power budget. To tackle these fundamental issues, VLSI engineers and researchers have started to leverage technologies developed for distributed systems, where the cost of communication has always been the central issue. One of the most widespread abstractions employed in the analysis and design of distributed systems and communications engineering is the *protocol stack*.

A simplified protocol stack for a generic communication system [Agrawal 1998] is shown in Figure 17. Abstraction decreases from top to bottom. Lower layers offer an abstract view of the communication channel to the

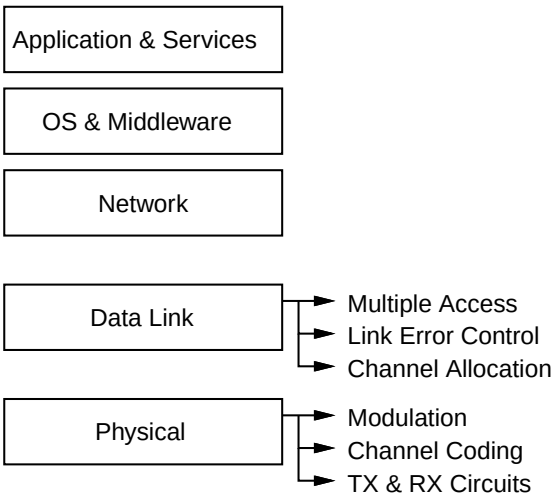


Fig. 17. The communication protocol stack.

layers above. In this way design complexity within each layer is reduced. In distributed wireless systems, communication power consumption has been tackled at all the levels of the stack (for recent surveys on this topic, see Agrawal [1998] and Bambos [1998]). In current electronic VLSI systems, most efforts in energy-efficient communication channel design have focused on the bottom layers of the stack, namely the *physical* and *link* layers. Thus, we narrow our analysis to techniques within these stack layers. It is important however to acknowledge that VLSI designers will soon have to deal with the stack levels above the link layer. We believe that, for communications, future VLSI systems will increasingly resemble today's distributed systems.

The physical layer of the protocol stack deals with the physical nature of the communication channels, and the circuit technology needed to send and receive information over them. In VLSI systems, communication is carried over interconnect wires. Transmitters and receivers are implemented in the same technology used for logic circuits. The design of transmitters and receivers that matches the characteristics of interconnect wires is one of the key physical layer challenges. This problem has been studied in detail, with emphasis on achieving maximum performance [Dally and Poulton 1998]. Additional issues tackled in the physical layer are modulation and channel coding. The modulation and channel-coding schemes traditionally used in VLSI were straightforward. Rail-to-rail binary signals from computation or storage units were directly transmitted on the communication bus. But recently this situation has changed.

In the data link layer, the physical nature of the channel is abstracted away, as well as the architecture of transmitter and receiver circuits. The issues addressed in this layer are error control through coding and management of channel resources when multiple transmitters and receivers must communicate over it (channel allocation and multiple access control). In

traditional VLSI systems, communication was reliable, and error control was not considered a top priority. In constrast, channel allocation and multiple access control have been studied in detail for shared system busses, where several computations and storage must communicate over the same channel [Duato et al. 1997]. However, most research and development efforts have focused on performance.

We survey recent work on power-efficient communication, moving from the physical layer up to the data link layer. We employ a simple model for estimating the power dissipated in communication. We assume that communication is carried over a set of metal wires (a *bus*). Consistent with previous sections, we assume that the energy dissipated in communication is mainly dynamic. The total capacitance of the wires supporting communication is fixed, and it is assumed to be much larger (two to three orders of magnitude) than the average capacitance of local wires. Furthermore, communication throughput and latency are tightly constrained. Communication power can therefore be reduced by either scaling down the voltage swing or the average number of signal transitions.

Swing reduction. At the physical level, we can lower power consumption by reducing the voltage swing on the high-capacitance wires of the bus. By lowering the voltage swing, the corresponding power dissipation decreases quadratically. The trend in reducing voltage levels is fundamentally limited by noise margins as well as component compatibility. However, in current CMOS technology, sizable power savings can be achieved by reduced-swing signalling without compromising communication reliability. Single-ended communication schemes can transmit information reliably with voltage swings in the neighborhood of $0.7V$. For even smaller voltage swings, differential signalling is required. CMOS circuits for low swing, single-ended signalling are surveyed by Zhang and Rabaey [1998]. Differential signalling techniques are studied in detail in the textbook by Dally and Poulton [1998].

It is important to note that low-swing signalling is beneficial for performance and because it takes less time for a finite-slope signal to complete a small swing than a large swing. Hence, both performance and energy optimization push towards low-swing signalling. The tradeoff is against reliability, which will ultimately limit swing reductions. However, there are techniques, based on redundant coding, that can greatly help in mitigating reliability concerns. Coding for error control has been a major research area in wireless telecommunications, where the channel is extremely unreliable. Hence, VLSI designers can tap into a well-developed field to look for solutions to reliability issues in communications. Preliminary studies of the energy-reliability tradeoff were performed by Hedge and Shanbhag from an information-theoretical viewpoint [Hedge and Shanbhag 1998]. Their analysis clearly indicates the potential for error tolerance via error-control coding in achieving low-energy operations. Future low-energy signalling schemes may allow some nonnegligible error rates in the physical layer, and then provide robust error-control coding techniques at the data link layer.

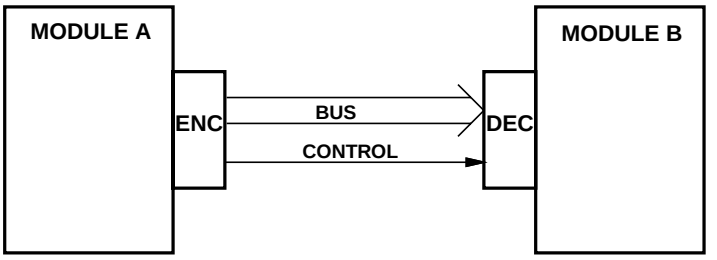


Fig. 18. Bus encoding: One-directional communication.

Data encoding. Bus data communication in VLSI circuits has traditionally adopted simple signal encoding and modulation techniques for communication. The transfer function of the encoder and modulator was implicitly assumed to be unitary. In other words, no modulation and encoding were applied to the binary data before sending it on the bus. Low-energy communication techniques have eliminated this implicit assumption by introducing the concept of *signal coding for low power*.

The basic idea behind these approaches is to encode the binary data sent through the communication channel to minimize its average switching activity, which is proportional to dynamic power consumption. Ramprasad et al. [1998] studied data encoding for the minimum switching activity problem and obtained upper and lower bounds on transition activity reduction for any encoding algorithm. This important result can be summarized as follows: the savings obtainable by encoding depend on the entropy rate of the data source and on the amount of redundancy in the code. The higher the entropy rate, the lower the energy savings that can be obtained by encoding from a given code redundancy.

Even though the work by Ramprasad et al. provides a theoretical framework for analyzing encoding algorithms, it does not provide general techniques for obtaining effective encoders and decoders. It is also important to note that the complexity and energy costs of encoding and decoding circuits must be taken into account when evaluating an encoding scheme.

Several authors have proposed low-transition activity encoding and decoding schemes. To illustrate the characteristics of these schemes, we consider, in the sequel, a point-to-point one-directional bus connecting two modules (e.g., a processor and its memory), as shown in Figure 18. Data from the source module is encoded, transmitted on the bus, and decoded at the destination. An instance of this problem, as shown in Figure 19, is the address bus for the processor/memory system. The techniques described here have wider applicability, e.g., for data busses connecting arbitrary computational units.

In the sequel, we assume that busses have relatively large parasitic capacitance, so that the energy dissipated in data transfers is significant, and dominates the energy required to encode/decode the signals at both ends. This assumption has been verified on practical circuits for most of the

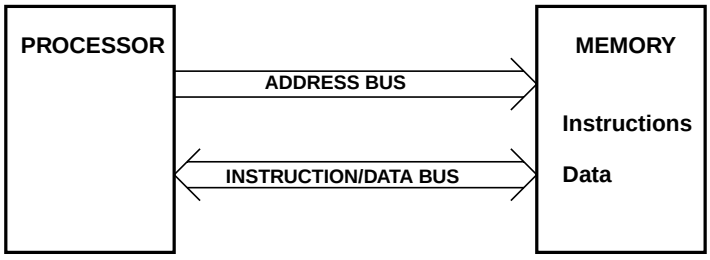


Fig. 19. Simple example of processor/memory subsystem with a Von Neumann architecture.

schemes described below, and must be validated whenever a new encoding scheme is considered.

Encoding for random white noise data. A few encoding schemes have been studied starting from the assumption that the data sent on the bus is random white noise (RWN), i.e., it has the maximum entropy rate. By this assumption, it is possible to just focus on how to exploit redundancy to decrease switching activity, since all irredundant codes have the same switching activity (this result is proved by Ramprasad et al. [1998]). Data is formatted in words of equal width, and a single word is transmitted every clock cycle.

Encoding schemes are based on the notion of adding *redundant* control wires to the bus. This can be seen as extending the word's width by one or more redundant bits. These bits inform the receiver about how the data was encoded before the transmission (see Figure 18). Low energy encodings exploit the correlation between the word currently being sent and the previously transmitted one. The rationale is that energy consumption is related to the number of switching lines, i.e., to the the Hamming distance between the words. Thus, transmitting identical words will consume no power, but alternating a word and its complement will produce the largest power dissipation, since all bus lines would be switching.

A conceptually simple and powerful encoding scheme, called *bus invert* (BI), was proposed by Stan and Burleson [1995]. To reduce switching, the transmitter computes the Hamming distance between the word to be sent and the previously transmitted one. If the distance is larger than half the word width, the word to be transmitted is inverted, i.e., complemented. An additional wire carries the *bus invert* information, which is used at the receiver's end to restore the data.

This encoding scheme has some interesting properties. First, the worst-case number of transitions of an n-bit bus is $n/2$ at each time frame. Second, if we assume that data is uniformly randomly distributed, it is possible to show that the average number of transitions with this code is lower than that of any other encoding scheme with just one redundant line [Stan and Burleson 1995].

An unfortunate property of the 1-bit redundant bus-invert code is that the average number of transitions per line increases as the bus gets wider, and asymptotically converges to 0.5, which is also the average switching

Table I. Comparing Code Efficiency To BI Code

#line	mode	AT	AT/line	AP	MT	MT/line	MP
2	unencoded	1	0.5	100%	2	1	100%
2	1-invert	0.75	0.375	75%	1	0.5	50%
8	unencoded	4	0.5	100%	8	1	100%
8	1-invert	3.27	0.409	81.8%	4	0.5	50%
8	4-invert	3	0.375	75%	4	0.5	50%
16	unencoded	8	0.5	100%	16	1	100%
16	1-invert	6.83	0.427	85.4%	8	0.5	50%

Legend: AT = Average Transition per time frame.
AP = Average normalized Power dissipation per time frame.
MT = Maximum Transition per time frame.
MP = Maximum normalized Power dissipation per time frame

per line of an unencoded bus (see Table I). Moreover, the average number of transitions per line is already close to 0.5 for 32-bit busses. Thus, this encoding provides small energy saving for busses of typical width.

A solution to this problem is to partition the bus into fields, and to use bus inversion in each field independently. If a word is partitioned in m fields, then m control lines are needed. Whereas this scheme can be much more energy efficient as compared to 1-bit bus invert, m-bit bus invert is no longer the best among m-redundant codes. Nevertheless, it is conceptually simpler than other encoding schemes based on redundancy, and thus its implementation overhead (in terms of power) is small.

Extensions to the bus-invert encoding approach include the use of limited-weight codes and transition signalling. A k-limited-weight code is a code having at most k 1's per word. This can be achieved by adding appropriate redundant lines [Stan and Burleson 1997]. Such codes are useful in conjunction with transition signalling, i.e., with schemes where 1's are transmitted as a 0-1 (or 1-0) transition and 0's by the lack of a transition. Thus, a k-limited-weight code would guarantee at most k transitions per time frame (if we neglect the transitions on the redundant lines).

Exploiting spatio-temporal correlations. Even though the random white noise data model is useful for developing redundant codes with good worst-case behavior, in many practical cases data words have significant spatio-temporal correlation. From an information-theoretic viewpoint, this means that the data source is not maximum entropy. This fact can be profitably exploited by advanced encoding schemes that outperform codes developed under the RWN model [Stan and Burleson 1997], even without adding any redundant bus line.

A typical example of highly correlated data streams is the address stream in processor/memory systems. Addresses show a high degree of sequentiality. This is typical for instruction addresses (within basic blocks) and for data addresses (when data is organized in arrays). Therefore, in the limiting case of addressing a stream of data with consecutive addresses,

Gray coding is beneficial [Su and Despain 1995] because the Hamming distance between any pair of consecutive words is one, and thus the transitions on the address bus are minimized.

By using Gray encoding, instruction addresses need to be converted to Gray addresses before being sent off to the bus. The conversion is necessary because offset addition and arithmetic address manipulation is best done with standard binary encoding [Mehta et al. 1996]. Moreover, address increments depend on the word width n. Since most processors are byte-addressable, consecutive words require an increment by $n/8$, e.g., by 4 (8) for 32-bit (64-bit) processors. Thus the actual encoding of interest is a partitioned code, whose most significant field is Gray encoded and whose least significant field has $\lceil \log n/8 \rceil$ bits.

Musoll et al. [1998] proposed a different partitioned code for addresses, which exploits the locality of reference. Namely, most software programs favor *working zones* of their address space. The proposed approach partitions the address into an offset within a working zone and an identifier of the current working zone. In addition, a bit is used to denote a hit or a miss of the working zone. When there is a miss, the full address is transmitted through the bus. In the case of a hit, the bus is used to transmit the offset (using 1-hot encoding and transition signalling) and additional lines are used to send the identifier of the working zone (using binary encoding).

The $T0$ code [Benini et al. 1997] uses one redundant line to denote when an address is consecutive to the previously-transmitted one. In this case the transmitter does not need to transmit the address, and freezes the information on the bus, thus avoiding any switching. The receiver updates the previous address. When the address to be sent is not consecutive, it is transmitted *tout court*, and the redundant line is deasserted to inform the receiver to accept the address as is. When transmitting a sequence of consecutive addresses, this encoding requires no transition on the bus, as compared to the single transition (per transmitted word) of the Gray code.

It is interesting to note the complementarity and similarity of BI and T0 codes. In its simplest implementation, BI uses one redundant line as T0 does. Such a line is asserted when the transmitted word needs to be either complemented or incremented at the receiving end, while it is deasserted when the transmitted word should be preserved as is. This observation justifies the combination of BI and T0 codes for busses that transmit both data and addresses. For such shared busses, two redundant lines can be used to require complementation or increment at the receiving end.

Some processors, such as those in the MIPS family, have time-multiplexed address busses for both instructions and data addresses. The two address streams may have different characteristics: instructions are likely to be highly sequential and data may have few in-sequence patterns. An efficient encoding scheme for multiplexed address busses is to use two redundant signals, one of which is already present in the bus interface, to demultiplex the bus on the receiving side. When this signal denotes an incoming instruction address stream, the receiver expects such addresses

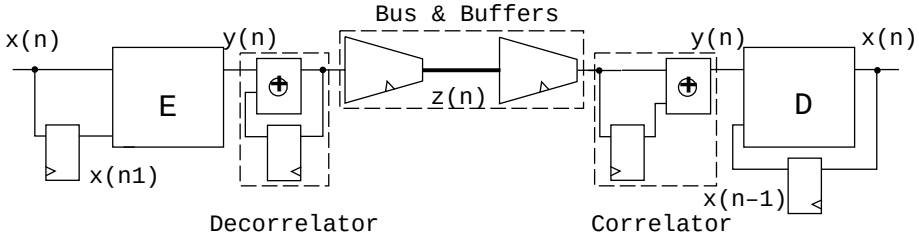


Fig. 20. Advanced encoder-decoder architecture template.

encoded with the T0 code, controlled by the second redundant signal. In a similar vein, the signal denoting the incoming stream can be used to switch from T0 to the BI code [Benini et al. 1998a].

The highly sequential nature of addresses is just one simple example of spatio-temporal correlation on communication busses. Several encoding schemes have been proposed for dealing with much more general correlations [Benini et al. 1998b; 1999a; Ramprasad et al. 1998]. The basic assumption in these approaches is that data stream statistics can be collected and analyzed at design time.

A complete statistical characterization of the data stream is a discrete function $f_K(w_1, w_2, \ldots, w_K) \rightarrow [0,1]$, where the input domain is the set of K-tuples of input symbols to be transmitted, and the output value is the probability of each K-tuple. Even though it is in principle possible to construct function f_K for an arbitrary value of K by examining the input stream, in practice the complexity of the data-collection process grows very rapidly with K. Thus, f_K is constructed for $K = 1$ (first-order statistic) and $K = 2$ (second order, or pairwise, statistic). Given f_1 or f_2, we wish to build an encoder and decoder pair that minimizes average switching activity. These circuits can be described as specializations of a basic template, shown in Figure 20.

The encoder takes as inputs $I + 1$ consecutive symbols (i.e., unencoded words) $x(n)$, $x(n - 1)$, $\ldots$, $x(n - I)$ from the source, and produces an encoded word. The decoder takes as input one encoded word and I previously decoded words, and outputs the original unencoded symbol $x(n)$. A decorrelator-correlator pair (DECOR) can optionally be inserted between the encoder-decoder pair and the bus, as shown in Figure 20. This circuit has as its only purpose mapping the ones in $y(n)$ into transitions on the bus lines, and vice versa. It is useful because it translates the problem of minimizing transitions into the problem of minimizing the number of ones.

Beach code, described by Benini et al. [1998b] is algorithmically constructed starting from second-order statistics. Its encoder takes one input word $x(n)$ (i.e., $I = 0$), produces one output word, and does not use a DECOR. The basic rationale in Beach code construction is to assign codes with small Hamming distance to data words that are likely to be sent on the bus in two successive clock cycles. The code was tested on the processor-

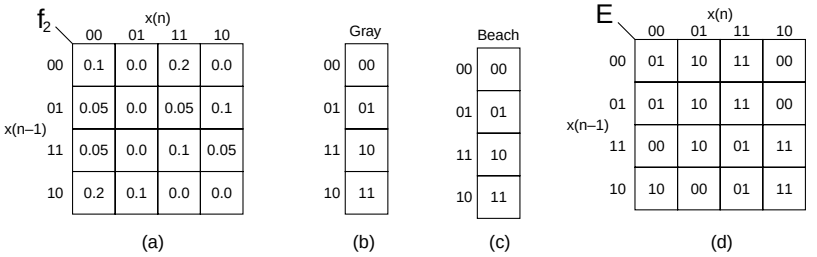


Fig. 21. Examples of low transition encodings.

memory bus of a MIPS R4000 core (in single user mode). It decreased average switching activity by 40%, on average.

Several encoding schemes with $I = 1$ (i.e., exploiting both $x(n)$ and $x(n - 1)$ to decide the value of $y(n)$) were investigated by Ramprasad et al. [1998]. A procedure for building optimum $I = 1$ codes for a given f_2 statistic is described by Benini et al. [1999a]. Extensive tests on these codes indicate that they can be much more effective than any other low-transition codes. Average transition activity reductions exceeding 80% have been reported.

It is important to acknowledge, however, that reducing transition activity does not directly map into power savings. The power consumed by encoder and decoder has to be taken into account. For large bus widths, the power of the encoder and decoder may scale up more than linearly, thereby reducing power savings. To mitigate this problem, clustering techniques have been proposed [Benini et al. 1998b; 1999a] that divide the wide bus into several smaller clusters. Minimum-transition encoder and decoder pairs are then built for each cluster. This approach trades-off some of the theoretically achievable reduction in transition activity for reduced encoder-decoder complexity (and power).

Example 4.9 Consider the 2-bit data bus with the second-order statistic f_2 shown in Figure 21(a). The average transition activity of the original binary code is 1.25. If we apply Gray encoding on the data, as shown in Figure 21(b), average switching is reduced to 1.05. The same result is obtained by constructing the Beach code, shown in Figure 21(c), for the given f_2. However, if we build the optimal code according to the procedure described by Benini et al. [1999a], we obtain an average transition activity 0.45. The encoding function is shown in Figure 21(d). Note that the encoding function takes as inputs two consecutive values of the original downstream. Hence the encoding function has four inputs, and its implementation is expected to be more expensive than the Gray or Beach encoders. This example shows that the encoder architecture of Figure 20 can achieve much better reduction in transition activity than encoders that observe only one data word at a time. On the other hand, the complexity of the encoder and the decoder (and their power dissipation) grows.

The main limitation of approaches that assume known data stream statistics is that it is not always possible to collect f_K in advance. If f_K is not available, then there is not enough information to build the encoder and decoder. To address this limitation, an adaptive encoding approach was proposed by Benini et al. [1999a]. Adaptive encoding is less effective than the ones previously described in reducing transition activity, but it can be applied to data streams with unknown statistics. The basic principle of adaptive encoding is to "learn" input statistics online and to automatically select the code that reduces switching. Another adaptive encoding technique was proposed by Komatsu et al. [1999]. This scheme relies on the presence of a "code table" in the encoder and decoder that stores the most frequently communicated data words. The code table can be periodically updated, thereby guaranteeing online adaptation.

Arbitration protocols. Up to now, we have only considered point-to-point busses with a single bus master. Many practical bus organizations support multiple masters and multiple independent receivers. Protocols for bus sharing are designed in the data link layer of the protocol stack. Techniques for bus access control are often called *arbitration protocols*. Arbitration for low power is still an open issue. The work by Dasgupta and Karri [1998] on scheduling and binding of data transfer on a shared bus shows that bus power consumption can be reduced if highly correlated data streams are scheduled consecutively on the bus.

Givargis and Vahid [1998] investigated the impact of different types of data transfer on bus power consumption. They considered *equal split* data transfers, where long data words are split in many consecutive short subwords and sent consecutively over a narrow bus. These research results confirm that link-layer media access protocols do impact power. However, no complete investigation has been carried out for complex, realistic bus protocols.

Bus design. Bus power can be reduced not only by lowering switching activity, but also by reducing the capacitance that needs to be switched. This goal can be achieved either by minimizing bus length with careful module placement and bus routing [Pedram and Vaishnav 1997], or by building a *partitioned* and *hierarchical* bus. Several interconnect architectures were analyzed by Zhang and Rabaey [1998]. The emphasis of this work is on reconfigurable systems where heterogeneous computation units are connected together through an reconfigurable network. *Global interconnect networks* such as the crossbar (i.e., a network with a dedicated connection between each source and each destination), multistage interconnects (e.g., Omega networks [Duato et al. 1997]), and multibus networks are considered. These architectures provide routes with identical costs for all connections between any pair of modules, but they are not power efficient because they do not allow exploitation of locality in data transfers in order to reduce energy consumption.

Local interconnection networks are asymmetric, in the sense that they provide local low-cost connection for modules that are close together, while using global connection for communications among topologically remote modules. Network architectures in this class are the generalized mesh, and several flavors of hierarchical interconnect networks (for details, see Zhang and Rabaey [1998].) The authors conclude that hierarchical generalized meshes are the most power efficient because they have enough flexibility to support remote data transfers, but they also provide abundant bandwidth for low-power short-range interconnects for local communication.

Bus segmentation is a technique for automatically transforming a long, heavily loaded global bus into a partitioned multistage network by inserting pass transistors on the bus lines to separate various local busses, called segments [Chen et al. 1999]. This transformations can reduce power by as much as 60% compared to a global unpartitioned bus.

4.2 Software Analysis and Compilation

Systems have several software layers running on the hardware platform. Here we consider application software programs, and defer issues related to runtime system software to Section 5. Application software is typically written in programming languages (e.g., C, C++, Java) and then compiled into machine code for specific instruction-set micro-architectures.

Interesting metrics for power consumptions are the energy required by a program to perform a batch (one-time) job, as well as average power consumption for interactive applications. Performance metrics are latency (i.e., user waiting time) for batch jobs and average response time for interactive programs.

Software does not consume energy *per se*, but the execution and storage of software requires energy consumption by the underlying hardware. Software execution corresponds to performing operations on hardware, as well as accessing and storing data. Thus software execution involves power dissipation for computation, storage, and communication. Moreover, storage of computer programs in semiconductor memories requires energy (refresh of DRAMs, static power for SRAMs). The energy budget for storing programs is typically small (with the choice of appropriate components) and predictable at design time. Hence we concentrate on energy consumption of software during its execution. Nevertheless, it is important to remember that reducing the size of programs, which is the usual goal in compilation, correlates with reducing their energy storage costs. Additional reduction of code size can be achieved by means of compression techniques, described in Section 4.1.2.

The energy cost of executing a program depends on its machine code and on the corresponding micro-architecture—if we exclude the intervention of the operating system in the execution (e.g., swapping). Thus, for any given micro-architecture, energy cost is tied to machine code. Since the machine code is derived from the source code from compilation, it is the compilation process itself that affects energy consumption. It is interesting to note that

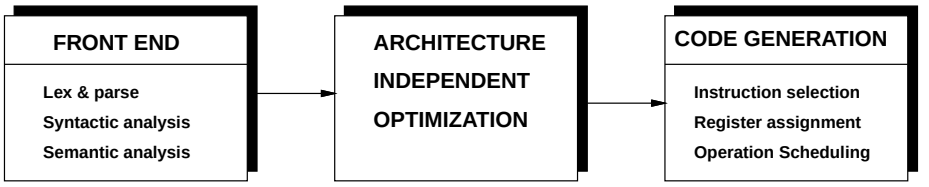


Fig. 22. Grouping major tasks in a software compiler.

the style of the software source program (for any given function) affects energy cost. This issue is addressed in Section 4.2.3. It is conceivable to think that the front-end of a compiler could handle source/source transformations, and thus the program transformations in Section 4.2.3 could be automated and embedded in a compiler. But this is not the case with current software technologies.

On the basis of the aforementioned arguments, the energy cost of machine code can be affected by the back-end of software compilation (see Figure 22 for a typical flow inside a software compiler). In particular, energy is affected by the type, number, and order of operations and by the means of storing data, e.g., locality (registers vs. memory arrays), addressing, and order.

In the sequel, we first address the energy/performance analysis of software and then leverage this information to show how energy-efficient compilation is performed.

4.2.1 *Software Analysis*. There are two avenues to take in exploring the energy costs of programs: simulation and experimentation, which have different goals.

The first involves simulating the execution of software programs on a given architecture. Simulation is used to achieve an overall assessment of the energy/performance of a software program. It requires models, possibly of different types, for the hardware platform. When high-level models are used, such as instruction set and/or bus functional models, energy/performance models for instructions and/or bus transactions are needed. When detailed electrical models are used, energy/performance can be derived from the electrical properties of the semiconductor process. Nevertheless, this approach is too computationally expensive, since execution of a software program involves a very large amount of electrical switching. When models are not available, or when an accurate measure of energy is required for a given operation in an ISA, experimental measurements can be used, as described below. The goal of experimental measurement is to derive empirical models for operation execution and bus transactions.

The search for appropriate energy/performance models is an area of active research. The overall objective is to abstract the cost of operations and memory access, while considering the different memory structures. Modeling the cost of each operation is the easiest problem, and can be done by means of detailed electrical simulation of the corresponding hardware resources. Modeling memory access has been addressed by various re-

searchers. Accurate energy consumption cache models are presented by Kamble and Ghose [1997]. The cache models rely on knowledge of the capacitance of each portion of cache design, stochastic distributions for signal values, and the runtime statistics for hit/miss and read/write counts. Jouppi and Wilton [1996] designed CACTI - an enhanced cache access and cycle time model based on resistance and capacitance values derived from the technology files and the cache netlist. RAM energy consumption and performance models based on technology parameters and the netlists are described by Itoh et al. [1995]. These models require knowledge of the internal structure and implementation of commodity components.

There are also a few tools [Li and Henkel 1998; Kapoor 1998] that estimate the energy consumption of software, caches and off-chip memory for *system on a chip* (SOC) design. Within these tools, performance and energy consumption of each component can be analyzed separately. The final system's energy consumption is obtained by summing the results of each analysis. Energy consumption models used by these approaches require detailed knowledge of the internal structure and implementation of the components, and as such are not applicable to designs based on commodity parts. In addition, it is difficult to estimate their accuracy, since no comparison is given of the simulation results with the hardware measurements.

Example 4.10 Šimunić et al. [1999a] developed energy models for systems using processors of the ARM family. Such models are used in conjunction with the cycle-accurate instruction-level simulator for the ARM processor family, called the ARMulator [Advanced RISC Machines Ltd (ARM) 1996]. With these power models (addressing memories, caches, interconnect, DC-DC converters, and energy source), as well as with application software (written in C) that is cross-compiled and loaded in specified locations of the system memory model, the ARMulator can accurately measure the execution of software code.

The experimental approach to energy measurement was pioneered by Tiwari et al. [1994], who measured the current consumption of some processors (e.g., an Intel 486DX2, a Fujitsu SparcLite 934), while executing loops with a specific instruction. This measure is then repeated for each instruction. Each iteration contains enough instructions to make the loop jump overhead negligible, but not too many, in order to avoid cache miss effects. A typical size for the loop body is 200 instructions. A first set of measurements yields the *base cost* of each operation. Sample results of base cost measurements are shown in Table II. A second set of measurements targets *interinstruction effects*, to take into account the effect of issuing an instruction in the state set by the previous ones. Interinstruction effects are measured by analyzing streams of instruction pairs. The current absorbed by each pair is larger than the sum of the currents absorbed while executing the corresponding single instruction: the difference can be tabulated for different pairs of instructions. These differences tend to be uniform for most pairs, and thus can be assimilated to a constant, repre-

Table II. Some Base Costs for Intel 486DX2

	Computation	Current (mA)	Cycles
1	NOP	257.7	1
2	MOV DX, BX	302.4	1
3	MOV DX, [BX	428.3	1
4	MOV DX, [BX][DI]	409.0	2
5	MOV [BX] DX	521.7	1
6	MOV [BX] [DI] DX	451.7	2

senting the circuit state overhead. The energy cost of an instruction (pair) is obtained by multiplying the measured current by the supply voltage and by the number of cycles, and dividing it by the operating frequency.

Nonideal instruction execution (e.g., pipeline stalls) is modeled by measuring the additional current consumption caused by the combination of instructions that cause the event of interest to occur.

Example 4.11 Wan et al. [1998] extend the StrongARM processor model with base current costs for each instruction. The average power consumption for most of the instructions is 200mW measured at 170MHz. Load and store instructions required 260mW each. Nonideal effects, such as stalls due to register dependencies and cache effects, are not considered by Wan. If all effects are measured, the total power consumed per instruction matches the data sheets. Because the difference in energy per instruction is minimal, it is expected that the average power consumption value from the data sheets is on the same level of accuracy as the instruction-level model.

4.2.2 *Software Compilation*. Software compilation is the object of extensive research [Aho et al. 1988; Muchnick 1997]. The design of an embedded system running dedicated software has brought a renewed interest in compilation, especially due to the desire for high-quality code (fast and energy efficient), possibly at the expense of longer compilation time (which is tolerable for embedded systems running code compiled by the manufacturer).

Most software compilers consists of three layers: the front-end, the machine-independent optimization, and the back-end (see Figure 22). The front-end is responsible for parsing and performing syntax and semantic analysis, as well as for generating an intermediate form, which is the object of many machine-independent optimizations [Aho et al. 1988]. The back-end is specific to the hardware architecture, and it is often called the *code generator* or *codegen*. Typically, energy-efficient compilation is done by introducing specific transformations in the back-end because they are directly related to the underlying architecture. Nevertheless, some machine-independent optimizations can be useful in general to reduce energy consumption [Mehta et al. 1997]. An example is selective loop unrolling, which reduces the loop overhead, but is effective if the loop is short enough. Another example is software pipelining that decreases the number of stalls by fetching instructions from different iterations; a third is removing tail recursion, which eliminates the stack overhead.

The main tasks of a code generator are instruction selection, register allocation, and scheduling. Instruction selection is the task of choosing instructions, each performing a fragment of the computation. Register allocation is allocating data to registers—when all registers are in use, data is *spilled* to main memory. Spills are usually undesirable, due to performance and energy overhead in saving temporary information in main memory. Instruction scheduling is ordering instructions in a linear sequence. When considering compilation for general-purpose microprocessors, instruction selection and register allocation are often achieved by dynamic programming algorithms [Aho et al. 1988], which also generate the order of instructions. When considering compilers for application-specific architectures (e.g., DSPs), the compiler back-end is often more complex due to irregular structures such as inhomogeneous register sets and connections. As a result, instruction selection, register allocation, and scheduling are intertwined problems that are much harder to solve [Goossens et al. 1997].

The traditional compiler goal is to speed up the execution of the generated code by reducing code size (which correlates with latency in execution time) and minimizing spills. Interestingly enough, executing machine code of minimum size consumes minimum energy if we neglect the interaction with memory and assume a uniform energy cost for each instruction.

Energy-efficient compilation strives to achieve machine code that requires less energy, compared to a performance-driven traditional compiler, by leveraging the disuniformity in instruction energy costs and the energy costs for storage in registers and in main memory, due to addressing and address decoding. Nevertheless, results are sometimes contradictory. Whereas for some architectures, energy-efficient compilation gives a competitive advantage compared to traditional compilation, for some others the most compact code is also the most economical in terms of energy, thus obviating the need for specific low-power compilers. In summary, this area is still an open field for research.

Consider first energy-efficient compilation that exploits instruction selection. This idea was proposed by Tiwari et al. [1994], and tied to software analysis and determination of base costs for operations. Tiwari et al. argue that accessing registers is much less energy consuming than accessing memory, and thus a reduction of memory operands is highly beneficial. Moreover, the resulting code also runs faster. Tiwari et al. propose an instruction selection algorithm based on the classical dynamic programming tree cover [Aho et al. 1988], where instruction weights are the energy costs. Experimental results show that this algorithm yields results similar to the traditional algorithm, thus supporting the hypothesis that the shortest code is the least-energy code when storage side-effects are neglected.

Instruction scheduling is an enumeration of the instructions consistent with the partial order induced by data and control flow dependencies. Instruction reordering for low energy can be done by exploiting the degrees of freedom allowed by the partial order. Instruction reordering may have several beneficial effects, including reduction of interinstruction effects

[Tiwari et al. 1996], as well as switching on the instruction bus [Su et al. 1994] and/or in some hardware circuits, such as the instruction decoder.

Su et al. [1994] proposed a technique called *cold scheduling*, which aims at ordering instructions to reduce interinstruction effects. In their model, interinstruction effects are dominated by switching on the instruction bus internal to a processor and by the corresponding power dissipation in the processor's control circuit. Given op-codes for the instructions, each pair of consecutive instructions requires as many bit lines to switch as the Hamming distance between the respective op-codes. The cold scheduling algorithm belongs to the family of list schedulers [De Micheli 1994]. At each step of the algorithm, all instructions that can be scheduled next are placed on a *ready* list. The priority for scheduling an instruction is inversely proportional to the Hamming distance from the currently scheduled instruction, thus locally minimizing interinstruction energy consumption on the instruction bus. Su et al. [1994] reported a reduction in overall bit switching in the range from 20% to 30%.

Tomiyama et al. [1998] generalized this scheduling problem to processor/memory systems. Their technique aims at reducing transitions on the data bus between an on-chip cache and off-chip main memory (when instruction cache misses occur), and thus at reducing the power consumed by the off-chip drivers and on the communication bus. A low-power compiler was designed that schedules instructions within basic blocks, so that bus switching is reduced. The scheduler searches for linear orders of instruction consistent with data-flow constraints, using a graph-based data structure annotated with the cost of a solution. The algorithm performs a pseudo-exhaustive search of the solution space, with the help of a pruning mechanism that avoids a redundant solution (by hashing subtrees) and by heuristically limiting the number of subtrees. Experimental results show a reduction of bus activity up to 28%.

Register assignment aims at the best utilization of available registers by reducing spills to main memory. Moreover, registers can be labeled during the compilation phase, and register assignment can be done with the goal of reducing switching in the instruction register as well as in register decoders [Mehta et al. 1997]. Again, the idea is to reduce the Hamming distance between pairs of consecutive register accesses. When comparing this approach to cold scheduling, note that the instruction order is now fixed, but the register labels can be changed. Mehta et al. [1997] proposed an algorithm that improves upon an initial register labeling by greedily swapping labels, until no further switching reduction is allowed. Experimental results show an improvement ranging from 4.2 % to 9.8%.

When considering embedded system applications, the memory/processor bandwidth can be reduced by recoding the relevant instructions and compressing the corresponding object code. As an example, if an embedded software program requires only a subset of instructions, then these instructions can be recoded (possibly with minimum-length code). This scheme requires instruction decompression ahead of the processor (core). An encoding

scheme proposed by Yoshida et al. [1997] is applied to an ARM 610 processor core.

4.2.3 *Writing Energy-Efficient Software Programs.* Software programs capture the system functionality to be executed by processors. It is often the case that programs that properly emulate the system functionality display poor performance and/or low energy efficiency. Therefore, such programs need to be rewritten. It is desirable that designers use software programs to model system functions and are energy efficient at the same time. The difficulty stems from the fact that (i) procedural languages allow designers to represent functions in many different ways; and (ii) software programs are sometimes written (or ported) without the knowledge of the target processor or of its specifics.

At present, there are very few guidelines for writing efficient software code. On the other hand, experiments show that, for some processors, some particular software writing styles are more efficient [Šimunić et al. 1999b] than others. To be specific, we next consider a case study.

Example 4.12 We now summarize some considerations for writing software for the ARM processor family in the form of tips to the programmer. Details are reported in Šimunić et al. [1999b], where energy savings are measured by comparing fragments of software code, using a specific style, against fragments coded with a generic style.

—**Integer division and modulo operation.** The ARM compiler uses a shift operation for modulo 2 division, since it is much more efficient than the standard division operation. In modulo 2 division, unsigned numbers should be used whenever possible, as the unsigned implementation is more efficient than the signed version (which requires sign extension correction on the shift operation).

—**Conditional execution.** All ARM instructions can be guarded; called conditionalized, in jargon. Conditionalizing is done in two steps. Firsts a few compare instructions set the compare codes; these instructions are then followed by the standard ARM instructions with their flag fields set, so that their execution proceeds only if the preset condition is true.

—**Boolean expressions.** A more energy-efficient way to check if a variable is within some range is to use the ability of the ARM compiler to conditionalize the arithmetic function.

—**Switch statement vs. table lookup.** Table lookup is more energy efficient than the switch statement when switch statement codes are more than half of the range of the possible labels.

—**Register allocation.** A compiler cannot usually assign local variables to a register if their addresses are passed to other functions. If the copy of the variable is made and the address of the copy is used instead, then a variable can be placed in the register, thus saving memory access. If

global variables are used, it is beneficial to make a local copy so that they can be assigned to registers.

—**Variable types.** The most energy-efficient variable type for the ARM processor is integer. The compiler by default uses 32 bits in each assignment, so when either short of char are used, sign or zero extension is needed, costing at most two extra instructions as compared to ints.

—**Function design.** By far the largest savings are possible with good function design. Function call overhead on ARM is four cycles. Function arguments are usually passed on the stack, but when there are four or fewer arguments, they can be passed in registers. Upon return from a function, structures up to four words can be passed through registers to the caller. When the return from one function calls another, the compiler can convert that call to branch to another function.

5. SYSTEM MANAGEMENT

We now consider digital systems during their operation. Typically, application software programs are monitored and controlled by an *operating system* (OS), which coordinates the various tasks, and thus plays an important role in achieving energy efficiency. Nevertheless, typical operating systems, like Unix, Windows, and MacOS, were not designed originally with energy efficiency in mind. Some support for low-power execution was introduced only recently [Intel, Microsoft, and Toshiba 1996; Microsoft 1997].

A first set of considerations revolve around the selection of an operating system and its size. The choice is often dictated by compatibility requirements. The size of the OS (and of its kernel) affect memory sizing and its traffic. Thus, there is an overall energy cost associated with the operating system's complexity.

Consider OSs in three different classes of systems: (i) general-purpose computing systems, e.g., portable computers; (ii) systems dedicated to an application, e.g., portable wireless terminals; (iii) systems operating under real-time constraints, e.g., vehicle operation controllers.

In the first case, operating systems provide a variety of services to the user. Nevertheless, it is important for the user to be able to customize the OS to his/her usage profile. In dedicated systems, the OS should support only the required functions. Lightweight, modular, and subsettable OSs are mandatory. Also, in this case, compatibility makes the choice of the OS (or of its subsetting) problematic. Real-time systems use specific operating systems to insure satisfaction of real-time constraints, which often relate to performance specs. These operating systems are typically simpler (compared to general-purpose OSs), and centered around a real-time scheduler. They are usually power aware to some extent.

A second and more important set of considerations relate to the heart of any OS, which is the task scheduler and should be energy aware. A scheduler usually determines the set of start times for each task, with the

goal of optimizing a cost function related to the completion time of all tasks, and to satisfy real-time constraints, if applicable. Since tasks are associated with resources having specific energy models, the scheduler can exploit this information to reduce run-time power consumption.

Recent research has followed two avenues. One relates to the design of schedulers that are better aware of processes, their needs, and their usefulness. For example, Lorch and Smith [1997] suggest heuristics to (i) avoid running processes that are still blocked and waiting on an event; (ii) delaying processes that execute without producing output or signalling useful activity; (iii) delaying the frequency of periodic processes that are not likely to produce useful services.

The second avenue deals with the task-scheduling itself, in presence of a time-varying workload. These techniques fall under the umbrella of dynamic power management, and are described in detail next. Specific scheduler implementations that exploit variable voltage and frequency processing units are described in Section 5.1.3.

5.1 Dynamic Power Management

Dynamic power management (DPM) is a design methodology that dynamically reconfigures an electronic system to provide the requested services and performance levels with a minimum number of active components or a minimum load on such components [Lorch and Smith 1998; Benini and De Micheli 1997]. DPM encompasses a set of techniques that achieve energy-efficient computation by selectively turning off (or reducing the performance of) system components when they are *idle* (or partially unexploited). DPM is used in various forms in most portable (and some stationary) electronic designs; yet its application is sometimes primitive because its full potential is still unexplored and because the complexity of interfacing heterogeneous components has limited designers to simple solutions.

The fundamental premise for the applicability of DPM is that systems (and their components) experience nonuniform workloads during operation time. Such an assumption is valid for most systems, both when considered in isolation and when internetworked. A second assumption of DPM is that it is possible to predict, with a certain degree of confidence, the fluctuations of workload. Workload observation and prediction should not consume significant energy.

Dynamic power managers can have different embodiments, according to the level (e.g., component, system, network) where DPM is applied and to the physical realization style (e.g., timer, hard-wired controller, software routine). Typically, a *power manager* (PM) implements a control procedure based on some observations and/or assumptions about the workload (see Figure 23). The control procedure is often called a *policy*. An example of a simple policy, ubiquitously used for laptops and palmtops, is the *timeout* policy, which shuts down a component after a fixed inactivity time, under the assumption that it is highly likely that a component will remain idle if it has been idle for the timeout time.

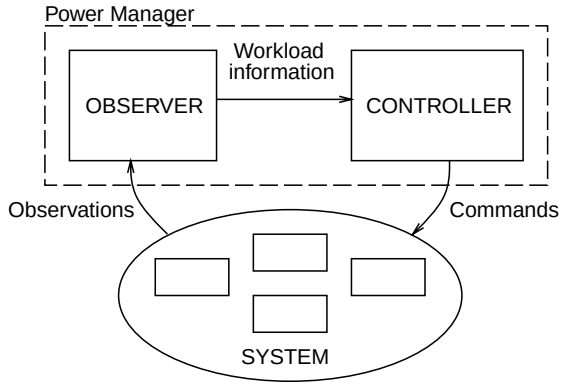


Fig. 23. Abstract view of a system-level power manager.

5.1.1 *Modeling Power-Managed Components*. We model a power-managed system as a set of interacting *power-manageable components* (PMCs) controlled by the power manager. We view PMCs as *black boxes*. The fundamental characteristic of a PMC is availability of multiple *modes of operation* that span the power-performance tradeoff. Examples of PMCs are processor cores (e.g., the ARM SA-1100 that has three main states), hard disk drives (e.g., the IBM Travelstar that has nine states), and displays.

An important characteristic of real-life PMCs is that transitions between modes of operation have a cost. In many cases, the cost is in terms of delay or performance loss. If a transition is not instantaneous and the component is not operational during a transition, performance is lost whenever a transition is initiated. Transition cost depends on PMC implementation: for example, restarting the clock and restoring the context in a SA-1100 requires 160ms.

In most practical instances, we can model a PMC by a *power state machine* (PSM), as illustrated in Section 3.4. States are the various modes of operation that span the tradeoff between performance and power. State transitions have a power and delay cost. In general, low-power states have lower performance and larger transition latency than states with higher power. This simple abstract model holds for many single-chip components, such as processors [Gary et al. 1994] and memories [Advanced Micro Devices 1998]; as well as for devices such as disk drives [Harris et al. 1996]; wireless network interfaces [Stemm and Katz 1997]; and displays [Harris et al. 1996] that are more heterogeneous and complex than a single chip.

Example 5.1 The StrongARM SA-1100 processor [Intel 1998] is an example of PMC, and has three modes of operation: `run`, `idle`, and `sleep`. `Run` mode is the normal operating mode of the SA-1100: every on-chip resource is functional. The chip enters `run` mode after successful power-up and reset. `Idle` mode allows a software application to stop the CPU when not in use, while continuing to monitor interrupt requests on or off-chip. In idle mode, the CPU can be brought back to run mode quickly when an interrupt occurs. `Sleep` mode offers the greatest power savings, and

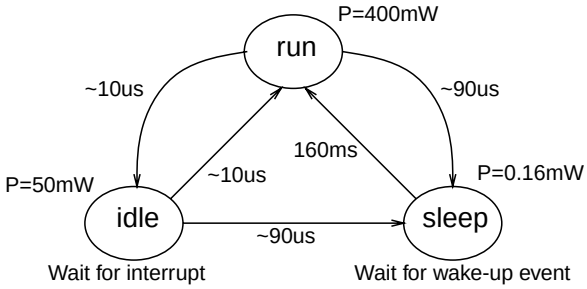


Fig. 24. Power state machine for the StrongARM SA-1100 processor.

consequently the lowest level of available functionality. In the transition from `run` or `idle`, the SA-1100 performs an orderly shutdown of on-chip activity. In a transition from `sleep` to any other state, the chip steps through a rather complex wake-up sequence before it can resume normal activity.

The PSM model of the StrongARM SA-1100 is shown in Figure 24. States are marked with power dissipation and performance values, edges are marked with transition times. The power consumed during transitions is approximatively equal to that in `run` mode. Notice that both `idle` and `sleep` have null performance, but the time for exiting `sleep` is much longer than that for exiting `idle` (10 μs versus 160 ms). On the other hand, the power consumed by the chip in `sleep` mode (0.16mW) is much smaller than that in `idle` (50mW).

5.1.2 *Dynamic Power-Management Techniques*. Putting a PMC into an inactive state causes a period of inactivity whose duration T_n is the sum of the actual time spent in the target state and the time spent to enter and exit it. We define the *break-even time* (denoted T_{BE}) as the minimum inactivity time required to compensate the cost of shutting down a component. The break-even time T_{BE} is inferred directly from the power-state machine of a PMC.

If $T_n < T_{BE}$, either there is not enough time to enter and exit the inactive state or the power saved when in the inactive state does not amortize the additional power consumption typically required to turn on the component. Intuitively, DPM aims at exploiting idleness to transition a component to an inactive low-power state. If no performance loss is tolerated, the length of the workload's idle periods is an upper bound for the inactivity time of the resource. On the other hand, if some performance loss is tolerated, inactivity times may be longer than idle periods.

An analysis of the break-even times of the system's PMCs in conjunction with the workload statistics can measure the usefulness of applying DPM. On the other hand, when designing components for power-managed systems, workload statistics can yield useful information on the transition times and power levels required for the component to be useful.

When designing a power-managed system, the central problem is to determine the policy that the PM will implement. Several approaches have been followed, which are described next.

Predictive techniques. In most real-world systems there is little knowledge of future input events, and DPM decisions have to be taken based on uncertain predictions. The rationale in all predictive techniques is that of exploiting the correlation between the past history of the workload and its near future, in order to make reliable predictions about future events. We denote by p the future event that we want to predict. We denote by o the past event whose occurrence is used to make predictions on p. For the purpose of DPM, we are interested in predicting idle periods long enough to go to sleep; that is, $p = \{T_{idle} > T_{BE}\}$.

Example 5.2 The most common predictive PM policy is the *fixed timeout* that uses the elapsed idle time as observed event ($o = \{T_{idle} > T_{TO}\}$) used to predict the total duration of the current idle period ($p = \{T_{idle} > T_{TO} + T_{BE}\}$). The policy can be summarized as follows: when an idle period begins, a timer is started with duration T_{TO}. If after T_{TO} the system is still idle, then the PM forces the transition to the `off` state. The system remains off until it receives a request from the environment that signals the end of the idle period. The critical design decision is obviously the choice of the timeout value T_{TO}.

When it is possible to chose as timeout the break-even time of the component, the corresponding policy has an important property: the energy consumption is at worse twice the energy consumed by an ideal policy (computed off-line) [Karlin et al. 1994]. The rationale of this strong result is related to the fact that the worse case happens for workloads with repeated idle periods of length $T_{idle} = T_{BE}$ separated by pointwise activity.

Timeouts have two main advantages: they are general and simple to implement. Unfortunately, large timeouts cause a large number of under-predictions, that represent missed opportunity of saving power, and a sizable amount of power is wasted waiting for the timeout to expire. Moreover there is a performance penalty upon wakeup.

Predictive shut-down policies [Golding et al. 1996; Srivastava et al. 1996] improve upon timeouts by taking decisions as soon as a new idle period starts, based on the observation of past idle and busy periods.

Example 5.3 Two predictive shut-down schemes were proposed by Srivastava et al. [1996]. In the first scheme, a nonlinear regression equation is obtained from the past history: it yields a predicted idle time T_{pred}. If $T_{pred} > T_{BE}$, the system is immediately shut down as soon as it becomes idle. The format of the nonlinear regression is decided heuristically, while the fitting coefficients can be computed with standard techniques. The main limitations of this approach are (i) there is no automatic way to decide the type of regression equation; (ii) offline data collection and analysis are required to construct and fit the regression model.

The second approach proposed by Srivastava et al. [1996] is based on a *threshold*. The duration of the busy period immediately preceding the current idle period is observed. If $o = \{T_{active}^{n-1} < T_{Thr}\}$, the idle period is assumed to be larger than T_{BE} and the system is shut down. The rationale is that, for the class of systems considered by Srivastava et al. (i.e., interactive graphic terminals), short active periods are often followed by long idle periods. Clearly, the choice of T_{Thr} is critical. Careful analysis of the scatter plot of T_{idle} versus T_{active} is required to set it to a correct value, hence this method is inherently offline (i.e., based on extensive data collection and analysis). Furthermore, the method is not applicable if the scatter plot is not L-shaped.

The DPM strategy proposed by Hwang and Wu [1997] addresses another limitation of timeout policies, namely the performance penalty that is always paid on wakeup. To reduce this cost, the power manager performs *predictive wakeup* when the predicted idle time expires, even if no new requests have arrived. This choice may increase power dissipation if T_{idle} has been under-predicted, but decreases the delay for servicing the first incoming request after an idle period.

Adaptive techniques. Since the optimality of DPM strategies depends on the workload statistics, static predictive techniques are all ineffective (i.e., suboptimal) when the workload is either unknown *a priori*, or nonstationary. So some form of adaptation is required. For timeouts the only parameter to be adjusted is timer duration, for history-based predictors even the type of observed events could in principle be adapted to the workload.

Several adaptive predictive techniques were proposed to deal with nonstationary workloads. In the work by Krishnan et al. [1995], a set of timeout values is maintained, and each timeout is associated with an index indicating how successful it will be. The policy chooses, at each idle time, the timeout that will perform best among the set of available ones. Another policy, presented by Helmbold et al. [1996], also keeps a list of candidate timeouts, and assigns a weight to each based on how well it will perform relative to an optimum offline strategy for past requests. The actual timeout is obtained as a weighted average of all candidates with their weight. Another approach, introduced by Douglis et al. [1995] is to keep only one timeout value and to increase it when it is causing too many shutdowns. The timeout is decreased when more shutdowns can be tolerated. Several predictive policies are surveyed and classified in Douglis' paper.

Example 5.4 The shutdown policy proposed by Hwang and Wu [1997] is capable of online adaptation, since the predicted idle time T_{pred}^{n} is obtained as a weighted sum of the last idle period T_{idle}^{n-1} and the last prediction T_{pred}^{n-1}. Namely, $T_{pred}^{n} = aT_{idle}^{n-1} + (1 - a)T_{pred}^{n-1}$. The impact of under-prediction is mitigated by employing a timeout scheme to periodically reevaluate T_{pred} if

the system is idle and has not been shut down. The impact of overprediction is reduced by imposing a saturation condition on predictions: $T^{n}_{pred} < C_{max} T^{n-1}_{pred}$.

Stochastic control. Policy optimization is a problem under uncertainty. Predictive approaches address workload uncertainty, but they assume deterministic response and transition times for the system. However, the system model for policy optimization is very abstract, and abstraction introduces uncertainty. Hence, it is more appropriate to assume a stochastic model for the system as well. Moreover, predictive algorithms are based on a two-state system model, while real-life systems have multiple power states. Policy optimization involves not only the choice of *when* to perform state transitions, but also the choice of *which* transition should be performed. Furthermore, predictive algorithms are heuristic, and their optimality can only be gauged through comparative simulation. Parameter tuning for these algorithms can be very hard if many parameters are involved. Finally, predictive algorithms are geared toward power minimization, and cannot finely control performance penalty.

The stochastic control approach addresses the generality and optimality issues outlined above. Rather than trying to eliminate uncertainty by prediction, it formulates policy optimization as an optimization problem under uncertainty. More specifically [Benini et al. 1999b], power-management optimization has been studied within the framework of *controlled Markov processes* [Ross 1997; Puterman 1994]. In this flavor of stochastic optimization, it is assumed that the system and the workload can be modeled as Markov chains. Under this assumption, it is possible to: (i) model the uncertainty in system power consumption and response (transition) times; (ii) model complex systems with many power states, buffers, queues, and so on; (iii) compute power-management policies that are globally optimum; and (iv) explore tradeoffs between power and performance in a controlled fashion.

When using Markov models, the problem of finding a minimum-power policy that meets given performance constraints can be cast as a linear program (LP). The solution of the LP produces a *stationary, randomized* policy. Such a policy is a nondeterministic function which, given a present system state, associates a probability with each command. The command to be issued is selected by a random trial on the basis of state-dependent probabilities. It can be shown [Puterman 1994] that the policy computed by LP is *globally optimum*. Furthermore, LP can be solved in polynomial time in the number of variables. Hence, policy optimization for Markov processes is exact and computationally efficient.

Example 5.5 A simple Markov model for a power-managed system [Benini et al. 1999b] is shown in Figure 25. The workload is modeled by a two-state Markov chain with two states: 0 (no request) and 1 (a request). The transition probabilities between states are represented as edge weights in Figure 25(a). The chain models a "bursty" workload. There is a high

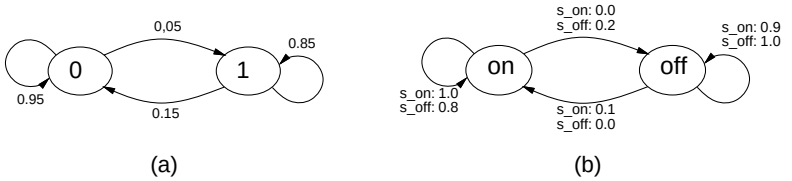


Fig. 25. A Markov model of a power-managed system and its environment.

probability (0.85) of receiving a request during period $n + 1$ if a request is received during period n, and the mean duration of a stream of requests is equal to $1/0.15 = 6.67$ periods.

The PMC model has two states as well, namely {on, off}. State transitions are controlled by two commands that can be issued by the power manager. The commands are, respectively, s_on and s_off, with the intuitive meaning of *"switch on"* and *"switch off"*. When a command is issued, the PMC will move to a new state in the next period with a probability dependent on the command and on the departure and arrival states. The Markov chain model of the PMC is shown in Figure 25(b). Edge weights represent transition probabilities. Note that their values depend on the command issued by the power manager. A power-management policy can be represented as a table that associates a command with each pair of states. For instance, a simple deterministic policy is f : {(0, on) → s_off, (1, on) → s_on, (0, off) → s_off, (1, off) → s_on}.

Stochastic control based on Markov models has several advantages over predictive techniques. First, it captures the global view of the system, thus allowing the designer to search for a global optimum that possibly exploits multiple inactive states of multiple interacting resources. Second, it enables the exact solution (in polynomial time) of the performance-constrained power optimization problem. Third, it exploits the strength and optimality of randomized policies.

One limitation of the stochastic optimization technique described above is that it assumes complete *a priori* knowledge of the system and its workload statistics. Even though it is generally possible to construct a model for the system once for all, the workload is generally much harder to characterize in advance. Furthermore, workloads are often nonstationary.

Example 5.6 An adaptive extension of the static stochastic optimization approach was presented by Chung et al. [1999]. Adaptation is based on three simple concepts: *policy precharacterization*, *parameter learning*, and *policy interpolation*. A simple, two-parameters Markov model for the workload is assumed, but the value of the two parameters is initially unknown.

Policy precharacterization constructs a two-dimensional table addressed by values of the two parameters. The table element uniquely identified by a pair of parameters contains the optimal policy for the system under the workload uniquely identified by the pair. The table is filled by computing optimum policies under different workloads. During system operation,

parameter learning is performed online. Short-term averaging techniques are employed to obtain run-time estimates of workload parameters based on past history. The parameter values estimated by learning are then used for addressing the lookup table and obtain the power-management policy. Clearly, in many cases the estimated parameter values do not correspond exactly to values sampled in the table. If this is the case, *policy interpolation* is employed to obtain a policy as a combination of the policies in table locations corresponding to parameter values close to the estimated ones.

Experimental results reported by Chung et al. [1999] indicate that adaptive techniques are advantageous, even in the stochastic optimization framework. Simulations of power-managed systems under highly nonstationary workloads show that the adaptive technique performs nearly as well as the ideal policy computed offline, assuming perfect knowledge of workload parameters over time.

5.1.3 *Implementation of Dynamic Power Management*. Dynamic power management can be implemented in different ways. For clocked hardware units, energy can be saved by reducing the clock frequency (and in the limit by stopping the clock), or by reducing the supply voltage (and in the limit by powering off a component). For example, clock-gating is widely used for controlling digital components [Benini and De Micheli 1997; Usami et al. 1998a].

Power shutdown to a component is a radical solution that eliminates all sources of power dissipation (including leakage). Moreover, it is widely applicable to all kinds of electronic components, i.e., digital and analog units, sensors, and transducers. A major disadvantage is the wake-up recovery time, which is typically higher than in clock-gating because the component's operation must be reinitialized.

Typical electronic systems are software-programmable, and a majority have an operating system ranging from a simple run-time scheduler or real-time operating system (RTOS) (for embedded applications) to a full-fledged operating system (as in the case of personal computers or workstations).

There are several reasons for migrating the power manager to software. Software power managers are easy to write and to reconfigure. In most cases, the designer cannot, or does not want to, interfere with and modify the underlying hardware platform. DPM implementations are still a novel art, and experimentation with software is easier than with hardware.

In general, the operating system is the software layer where the dynamic power-management policy can be implemented best. *OS-based power management* (OSPM) has the advantage that the power/performance dynamic control is performed by the software layer (i.e., the OS) that manages the computational, storage, and I/O tasks of the system.

Recent initiatives to handle system-level power management include Microsoft's *OnNow* initiative [Microsoft 1997] and the *Advanced Configuration and Power Interface* (ACPI) standard proposed by Intel, Microsoft, and Toshiba [1996]. The former supports the implementation of OSPM and

targets the design of personal computers with improved usability through innovative OS design. The latter simplifies the codesign of OSPM by providing an interface standard to control system resources. On the other hand, the aforementioned standards do not provide procedures for optimal control of power-managed system. Using ACPI, several policies have been implemented and tested on both a desktop and a laptop computer. A comparative evaluation of results is reported in Lu et al. [2000].

Recent research has addressed the design of task schedulers within OSs that take advantage of multiple frequency, multiple voltage components. In this case the scheduler implements a policy that sets the clock speed (and voltage) of the processor. For example, Weiser et al. [1994] developed a predictive technique that assumes that workload in a near-future time window is similar to that in a near-past window. Accordingly, this scheduler reduces the clock rate to satisfy scheduling constraints. Govil et al. [1995] perfected Weiser's scheme by proposing other predictive policies.

Scheduling for real-time systems with variable-voltage components was studied extensively in various flavors [Hong et al. 1998a; 1998b; 1998c; Ishihara and Yasuura 1998a; Pering et al. 1998; Qu and Potkonjak 1998]. An optimal schedule fixes both the time frame for the execution of a task and the supply voltage for the processor that runs the task. Shin and Choi studied a fixed priority scheduling algorithm for real-time systems with a variable voltage processor that can be shut down [Shin and Choi 1999]. Brown et al. [1997] considered the task scheduling problem form the perspective of increasing the efficiency of DPM. Consecutive tasks separated by an idle period are scheduled to execute after each other, thus eliminating the shut down and wake up costs.

Task scheduling and clock setting should be computed while using realistic battery models. Since the overall objective of DPM (for mobile systems) is to extend battery lifetime, then the charge should be extracted from the battery with a time-varying rate and the battery should be given some recovery intervals. Experiments by Martin and Sewiorek [1999] on the *Itsy* hand-held computer show that ideal clock setting policies change when considering different battery sources, such as alkaline or lithium-ion.

Even though the operating system is probably the most significant software layer for a power manager, information exchange between applications and OS can greatly help in aggressively reducing power consumption. Several authors have observed that applications should be involved in power management [Ellis 1999; Flinn and Satyanarayanan 1999]. However, the development of a communication infrastructure to support application-aware power management is still an unexplored research area.

6. CONCLUSIONS

Electronic system design aims at striking a balance between performance and energy efficiency. Designing energy-efficient systems is a multifaceted problem, due to the plurality of embodiments that a system specification may have and the variety of degrees of freedom that designers have to cope with power reduction.

We envision system design as consisting of three stages: modeling, implementation, and run-time management. Energy-efficient design must be addressed in all these phases. Moreover, it must target the sources of power consumption that can be identified as computation, communication, and storage.

Within this framework, we have attempted to cover a wide range of problems and their solutions. Our coverage is by no means exhaustive. Several problems in this framework have not been addressed yet, since this is still an area of active research. We have also limited ourselves to high-level system-level issues, and have purposely neglected chip-level design techniques at the logic, electrical, and physical design levels.

REFERENCES

ADVANCED MICRO DEVICES, 1998. AM29SLxxx low-voltage flash memories.

ADVANCED RISC MACHINES LTD., 1996. ARM software development toolkit version 2.11.

AGRAWAL, P. 1998. Energy conservation design techniques for mobile wireless VLSI systems. In *Proceedings of the Computer Society Workshop on VLSI System-Level Design* (Apr.), 34–39.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

BAHAR, R. I., ALBERA, G., AND MANNE, S. 1998. Power and performance tradeoffs using various caching strategies. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design* (ISLPED '98, Monterey, CA, Aug. 10–12), A. Chandrakasan and S. Kiaei, Eds. ACM Press, New York, NY, 64–69.

BAJWA, R. S., HIRAKI, M., KOJIMA, H., GORNY, D. J., NITTA, K., SHRIDHAR, A., SEKI, K., AND SASAKI, K. 1997. Instruction buffering to reduce power in processors for signal processing. *IEEE Trans. Very Large Scale Integr. Syst. 5*, 4, 417–424.

BAMBOS, N. 1998. Toward power-sensitive network architectures in wireless communications. *IEEE Personal Commun. 5*, 3 (June), 50–58.

BENINI, L. AND DE MICHELI, G. 1997. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Academic, Dordrecht, Netherlands.

BENINI, L., DE MICHELI, G., MACII, E., SCIUTO, D., AND SILVANO, C. 1997. Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems. In *Proceedings of the Great Lakes Symposium on VLSI* (Mar.), 77–82.

BENINI, L., DE MICHELI, G., MACII, E., SCIUTO, D., AND SILVANO, C. 1998a. Address bus encoding techniques for system-level power optimization. In *Proceedings of the Conference on Design, Automation and Test in Europe 98*, 861–866.

BENINI, L., DE MICHELI, G., MACII, E., PONCINO, M., AND QUER, S. 1998b. Power optimization of core-based systems by address bus encoding. *IEEE Trans. Very Large Scale Integr. Syst. 6*, 4, 554–562.

BENINI, L., HODGSON, R., AND SIEGEL, P. 1998c. System-level power estimation and optimization. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design* (ISLPED '98, Monterey, CA, Aug. 10–12), A. Chandrakasan and S. Kiaei, Eds. ACM Press, New York, NY, 173–178.

BENINI, L., MACII, A., MACII, E., PONCINO, M., AND SCARSI, R. 1999a. Synthesis of low-overhead interfaces for power-efficient communication over wide buses. In *Proceedings of the Conference on Design Automation* (June), 128–133.

BENINI, L., BOGLIOLO, A., PALEOLOGO, G., AND DE MICHELI, G. 1999b. Policy optimization for dynamic power management. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 18*, 6 (June), 813–833.

BROWN, J. J., CHEN, D. Z., GREENWOOD, G. W., HU, X., AND TAYLOR, R. W. 1997. Scheduling for power reduction in a real-time system. In *Proceedings of the 1997 International Symposium*

on Low Power Electronics and Design (ISLPED '97, Monterey, CA, Aug. 18–20), B. Barton, M. Pedram, A. Chandrakasan, and S. Kiaei, Eds. ACM Press, New York, NY, 84–87.

BRUNVAND, E., NOWICK, S., AND YUN, K. 1999. Practical advances in asynchronous design and in asynchronous/synchronous interfaces. In *Proceedings of the Conference on Design Automation* (June), 104–109.

BURD, T. D. AND BRODERSEN, R. W. 1996. Processor design for portable systems. *J. VLSI Signal Process. 13*, 2/3, 203–221.

CATTHOOR, F., FRANSSEN, F., WUYTACK, S., NACHTERGAELE, L., AND DE MAN, H. 1994. Global communication and memory optimizing transformations for low power systems. In *Proceedings of the International Workshop on Low Power Design*, 203–208.

CATTHOOR, F., WUYTACK, S., DE GREEF, E., BALASA, F., NACHTERGAELE, L., AND VANDECAPPELLE, A. 1998a. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic, Dordrecht, Netherlands.

CATTHOOR, F., WUYTACK, S., DE GREEF, E., FRANSSEN, F., NACHTERGAELE, L., AND DE MAN, H. 1998b. System-level transformations for low-power data transfer and storage. In *Low-Power CMOS Design*, R. Chandrakasan and R. Brodersen, Eds. IEEE Press, Piscataway, NJ.

CHANDRAKASAN, A. P., POTKONJAK, M., MEHRA, R., RABAEY, J., AND BRODERSEN, R. 1995. Optimizing power using transformations. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 14*, 1 (Jan.), 13–32.

CHANDRAKASAN, M. AND BRODERSEN, R. 1995. *Low Power Digital CMOS Design*. Kluwer Academic, Dordrecht, Netherlands.

CHANG, J.-M. AND PEDRAM, M. 1997. Energy minimization using multiple supply voltages. *IEEE Trans. Very Large Scale Integr. Syst. 5*, 4, 436–443.

CHAU, P. M. AND POWELL, S. R. 1992. Power dissipation of VLSI array processing systems. *J. VLSI Signal Process. 4*, 2/3 (May 1992), 199–212.

CHEN, J., JONE, W., WANG, J., LU, H., AND CHEN, T. 1999. Segmented bus design for low-power systems. *IEEE Trans. Very Large Scale Integr. Syst. 7*, 1 (Mar.), 25–29.

CHUNG, E., BENINI, L., BOGLIOLO, A., AND DE MICHELI, G. 1999. Dynamic power management for non-stationary service requests. In *Proceedings of the Conference on Design Automation and Test in Europe* (Mar.), 77–81.

CHUNG, J. W., KAO, D.-Y., CHENG, C.-K., AND LIN, T.-T. 1995. Optimization of power dissipation and skew sensitivity in clock buffer synthesis. In *Proceedings of the 1995 International Symposium on Low Power Design* (ISLPD-95, Dana Point, CA, Apr. 23–26), M. Pedram, R. Brodersen, and K. Keutzer, Eds. ACM Press, New York, NY, 179–184.

CONTE, T., MENEZES, K., AND SATHAYE, S. 1995. Technique to determine power-efficient, high-performance superscalar processors. In *Proceedings of the Hawaii International Conference on System Sciences* (HICSS '95, Maui, Hawaii, Jan.), IEEE Computer Society Press, Los Alamitos, CA, 534–333.

COUMERI, S. L. AND THOMAS, D. E. 1998. Memory modeling for system synthesis. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design* (ISLPED '98, Monterey, CA, Aug. 10–12), A. Chandrakasan and S. Kiaei, Eds. ACM Press, New York, NY, 179–184.

DALLY, W. J. AND POULTON, J. W. 1998. *Digital Systems Engineering*. Cambridge University Press, New York, NY.

DASGUPTA, A. AND KARRI, R. 1998. High-reliability, low-energy microarchitecture synthesis. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 17*, 12 (Dec.), 1273–1280.

DA SILVA, J. L., CATTHOOR, F., VERKEST, D., AND DE MAN, H. 1998. Power exploration for dynamic data types through virtual memory management refinement. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design* (ISLPED '98, Monterey, CA, Aug. 10–12), A. Chandrakasan and S. Kiaei, Eds. ACM Press, New York, NY, 311–316.

DAVE, B., LAKSHMINARAYANA, G., AND JHA, N. 1999. COSYN: Hardware-software co-synthesis for heterogeneous distributed embedded systems. *IEEE Trans. Very Large Scale Integr. Syst. 7*, 1 (Mar.).

DEBNATH, G., DEBNATH, K., AND FERNANDO, R. 1995. The Pentium processor-90/100, microarchitecture and low-power circuit design. In *Proceedings of the IEEE International Conference on VLSI Design* (Jan. 1995), IEEE Press, Piscataway, NJ, 185–190.

DE GREEF, E., CATTHOOR, F., AND DE MAN, H. 1998. Program transformation strategies for memory size and power reduction of pseudoregular multimedia subsystems. *IEEE Trans. Circuits Syst. Video Technol. 8*, 6 (Oct.), 719–733.

DE MICHELI, G. AND GUPTA, R. 1997. Hardware/sofware co-design. *Proc. IEEE 95*, 3 (Mar.), 349–365.

DE MICHELI, G. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., New York, NY.

DOUGLIS, F., KRISHNAN, P., AND BERSHAD, B. 1995. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, 121–137.

DOUGHERTY, W., PURSLEY, D., AND THOMAS, D. 1998. Instruction subsetting: Trading power for programmability. In *Proceedings of the Computer Society Workshop on VLSI System-Level Design* (Apr.), 42–47.

DUATO, J., YALAMANCHILI, S., AND NI, L. 1997. *Interconnection Networks. An Engineering Approach*. IEEE Computer Society Press, Los Alamitos, CA.

ELLIS, C. 1999. The case for higher-level power management. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems* (Mar.), IEEE Computer Society Press, Los Alamitos, CA, 162–167.

FARRAHI, A. H., TÉLLEZ, G. E., AND SARRAFZADEH, M. 1995. Memory segmentation to exploit sleep mode operation. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation* (DAC '95, San Francisco, CA, June 12–16), B. T. Preas, Ed. ACM Press, New York, NY, 36–41. http:www.getridofme.com

FARRAHI, A. H. AND SARRAFZADEH, M. 1995. System partitioning to maximize sleep time. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD-95, San Jose, CA, Nov. 5–9), R. Rudell, Ed. IEEE Computer Society Press, Los Alamitos, CA, 452–455.

FLINN, J. AND SATYANARAYANAN, M. 1999. Energy-aware adaptation for mobile applications. In *Proceedings of the ACM Symposium on Operating System Principles* (Dec.), ACM Press, New York, NY, 48–63.

FURBER, S. 1997. *ARM System Architecture*. Addison-Wesley Publishing Co., Inc., Redwood City, CA.

GAJSKI, D. D., DUTT, N. D., WU, A. C.-H., AND LIN, S. Y.-L. 1992. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Hingham, MA.

GAJSKI, D. D., VAHID, F., NARAYAN, S., AND GONG, J. 1994. *Specification and Design of Embedded Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ.

GARY, S. AND IPPOLITO ET AL., P. 1994. PowerPC 603, a microprocessor for portable computers. *IEEE Des. Test 11*, 4 (Winter), 14–23.

GEBOTYS, C. 1997. Low energy memory and register allocation using network flow. In *Proceedings of the 34th Conference on Design Automation* (DAC '97, Anaheim, CA, June), 435–440.

GIVARGIS, T. AND VAHID, F. 1998. Interface exploration for reduced power in core-based systems. In *International Symposium on System-Level Synthesis* (Dec.), 117–122.

GOLDING, R., BOSH, P., AND WILKES, J. 1996. Idleness is not Sloth. Hewlett-Packard, Fort Collins, CO.

GONZALEZ, R. AND HOROWITZ, M. 1996. Energy dissipation in general purpose microprocessors. *IEEE J. Solid-State Circuits 31*, 9 (Sept.), 1277–1284.

GOOSSENS, G., PAULIN, P., VAN PRAET, J., LANNEER, D., GUERTS, W., KIFLI, A., AND LIEM, C. 1997. Embedded software in real-time signal processing systems: Design technologies. *Proc. IEEE 85*, 3 (Mar.), 436–454.

GOVIL, K., CHAN, E., AND WASSERMAN, H. 1995. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *Proceedings of the First Annual International Conference on Mobile Computing and Networking* (MOBICOM '95, Berkeley, CA, Nov. 13–15), B. Awerbuch and D. Duchamp, Eds. ACM Press, New York, NY, 13–25.

GOWAN, M. K., BIRO, L. L., AND JACKSON, D. B. 1998. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the 35th Annual Conference on Design Automation* (DAC '98, San Francisco, CA, June 15–19), B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Eds, ACM Press, New York, NY, 726–731.

GUERRA, L., POTKONJAK, M., AND RABAEY, J. 1998. A methodology for guided behavioral-level optimization. In *Proceedings of the 35th Annual Conference on Design Automation* (DAC '98, San Francisco, CA, June 15–19), B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Eds, ACM Press, New York, NY, 309–314.

GUTNIK, V. AND CHANDRAKASAN, A. P. 1997. Embedded power supply for low-power DSP. *IEEE Trans. Very Large Scale Integr. Syst. 5*, 4, 425–435.

HAJJ, N. B. I., STAMOULIS, G., BELLAS, N., AND POLYCHRONOPOULOS, C. 1998. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design* (ISLPED '98, Monterey, CA, Aug. 10–12), A. Chandrakasan and S. Kiaei, Eds. ACM Press, New York, NY, 70–75.

HARRIS ET AL., E. 1995. Technology directions for portable computers. *Proc. IEEE 83*, 4 (Apr.), 636–657.

HASEGAWA, A., KAWASAKI, I., YAMADA, K., YOSHIOKA, S., YOSHIOKA, S., KAWASAKI, S., AND BISWAS, P. 1995. SH3: High code density, low power. *IEEE Micro 15*, 5 (Dec.).

HEGDE, R. AND SHANBHAG, N. R. 1998. Energy-efficiency in presence of deep submicron noise. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '98, San Jose, CA, Nov. 8–12), H. Yasuura, Ed. ACM Press, New York, NY, 228–234.

HELMBOLD, D. P., LONG, D. D. E., AND SHERROD, B. 1996. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the 2nd Annual International Conference on Mobile Computing and Networking* (MOBICOM '96, Rye, NY, Nov. 10–12), H. Ahmadi, R. H. Katz, I. F. Akyildz, and Z. J. Haas, Eds. ACM Press, New York, NY, 130–142.

HEMANI, A., MEINCKE, T., KUMAR, T., OLSSON, T., NILSSON, P., OBERG, J., ELLERVEE, P., AND LUNDQVIST, D. 1999. Lowering power consumption in clock by using globally asynchronous locally synchronous design style. In *Proceedings of the Conference on Design Automation* (June), 873–878.

HENKEL, J. 1999. A low-power hardware/software partitioning approach for core-based embedded systems. In *Proceedings of the Conference on Design Automation* (June), 122–127.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach*. 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA.

HONG, I., POTKONJAK, M., AND KARRI, R. 1997. Power optimization using divide-and-conquer techniques for minimization of the number of operations. In *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '97, San Jose, CA, Nov. 9–13), R. H. J. M. Otten and H. Yasuura, Eds. IEEE Computer Society, Washington, DC, 108–111.

HONG, I., KIROVSKI, D., QU, G., POTKONJAK, M., AND SRIVASTAVA, M. B. 1998. Power optimization of variable voltage core-based systems. In *Proceedings of the 35th Annual Conference on Design Automation* (DAC '98, San Francisco, CA, June 15–19), B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Eds, ACM Press, New York, NY, 176–181.

HONG, I., POTKONJAK, M., AND SRIVASTAVA, M. B. 1998. On-line scheduling of hard real-time tasks on variable voltage processor. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '98, San Jose, CA, Nov. 8–12), H. Yasuura, Ed. ACM Press, New York, NY, 653–656.

HONG, I., QU, G., AND POTKONJAK, M. 1998. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Proceedings of the 19th IEEE Symposium on Real-Time Systems* (Madrid, Spain, Dec.), IEEE Computer Society Press, Los Alamitos, CA, 178–187.

HWANG, C.-H. AND WU, A. C.-H. 1997. A predictive system shutdown method for energy saving of event-driven computation. In *Proceedings of the 1997 IEEE/ACM International*

Conference on Computer-Aided Design (ICCAD '97, San Jose, CA, Nov. 9–13), R. H. J. M. Otten and H. Yasuura, Eds. IEEE Computer Society, Washington, DC, 28–32.

INTEL. 1998. SA-1100 Microprocessor Technical Reference Manual. Intel Corp., Santa Clara, CA.

ISHIHARA, T. AND YASUURA, H. 1998. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design* (ISLPED '98, Monterey, CA, Aug. 10–12), A. Chandrakasan and S. Kiaei, Eds. ACM Press, New York, NY, 197–202.

ISHIHARA, T. AND YASUURA, M. 1998. Power-Pro: Programmable power management architecture. In *Proceedings of the on Asia and South Pacific Design Automation* (Feb.), 321–322.

ITOH, K., SASAKI, K., AND NAKAGOME, Y. 1995. Trends in low-power RAM circuit technologies. *Proc. IEEE 83*, 4 (Apr.), 524–543.

JOHNSON, M. AND ROY, K. 1996. Optimal selection of supply voltages and level-conversion during data-path scheduling under resource constraints [AUTHOR: This citation does NOT appearin this Proceedings. We need CORRECT info.]. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '96, San Jose, CA, Nov. 10–14), R. A. Rutenbar and R. H. J. M. Otten, Eds. IEEE Computer Society Press, Los Alamitos, CA.

JOUPPI, N. AND WILTON, N. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE J. Solid-State Circuits 31*, 5 (May), 677–688.

JUAN, T., LANG, T., AND NAVARRO, J. J. 1997. Reducing TLB power requirements. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design* (ISLPED '97, Monterey, CA, Aug. 18–20), B. Barton, M. Pedram, A. Chandrakasan, and S. Kiaei, Eds. ACM Press, New York, NY, 196–201.

KALAMBUR, A. AND IRWIN, M. J. 1997. An extended addressing mode for low power. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design* (ISLPED '97, Monterey, CA, Aug. 18–20), B. Barton, M. Pedram, A. Chandrakasan, and S. Kiaei, Eds. ACM Press, New York, NY, 208–213.

KAMBLE, M. B. AND GHOSE, K. 1997. Analytical energy dissipation models for low-power caches. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design* (ISLPED '97, Monterey, CA, Aug. 18–20), B. Barton, M. Pedram, A. Chandrakasan, and S. Kiaei, Eds. ACM Press, New York, NY, 143–148.

KAPOOR, B. 1998. Low power memory architectures for video applications. In *Proceedings of the Great Lakes Symposium on VLSI* (Feb.), 2–7.

KARLIN, A., MANASSE, M., MCGEOCH, L., AND OWICKI, S. 1994. Competitive randomized algorithms for nonuniform problems. *Algorithmica 11*, 6 (June), 542–571.

KIM, D. AND CHOI, K. 1997. Power-conscious high level synthesis using loop folding. In *Proceedings of the 34th Annual Conference on Design Automation* (DAC '97, Anaheim, CA, June 9–13), E. J. Yoffa, G. De Micheli, and J. M. Rabaey, Eds. ACM Press, New York, NY, 441–445.

KIN, J., GUPTA, M., AND MANGIONE-SMITH, W. H. 1997. The filter cache: an energy efficient memory structure. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 30, Research Triangle Park, NC, Dec. 1–3), M. Smotherman and T. Conte, Eds. IEEE Computer Society Press, Los Alamitos, CA, 184–193.

KIN, J., LEE, C., MANGIONE-SMITH, W., AND POTKONJAK, M. 1999. Power efficient mediaprocessors: Design space exploration. In *Proceedings of the Conference on Design Automation* (June), 321–326.

KIROVSKI, D., LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. 1998. Synthesis of power efficient systems-on-silicon. In *Proceedings of the Conference on Asian and South Pacific Design Automation* (Feb.), 557–562.

KIROVSKI, D. AND POTKONJAK, M. 1997. System-level synthesis of low-power hard real-time systems. In *Proceedings of the 34th Annual Conference on Design Automation* (DAC '97, Anaheim, CA, June 9–13), E. J. Yoffa, G. De Micheli, and J. M. Rabaey, Eds. ACM Press, New York, NY, 697–702.

KO, U., BALSARA, P. T., AND NANDA, A. K. 1998. Energy optimization of multilevel cache architectures for RISC and CISC processors. *IEEE Trans. Very Large Scale Integr. Syst. 6*, 2, 299–308.

KOMATSH, S., IKEDA, M., AND ASADA, K. 1999. Low power chip interface based on bus data encoding with adaptive code-book method. In *Proceedings of the Great Lakes Symposium on VLSI*, 368–371.

KRISHNAN, P., LONG, P., AND VITTER, J. 1995. Adaptive disk spindown via optimal rent-to-buy in probabilistic environments. In *Proceedings of the 12th International Conference on Machine Learning* (Lake Tahoe, CA), 322–330.

KULKARNI, C., CATTHOOR, F., AND DE MAN, H. 1998. Code transformations for low power caching in embedded multimedia processors. In *Proceedings of the First Merged IPPS/SPDP Symposium on Parallel and Distributed Processing* (IPPS/SPDP '98, Mar.), 23–26.

KUMAR, N., KATKOORI, S., RADER, L., AND VEMURI, R. 1995. Profile-driven behavioral synthesis for low-power VLSI systems. *IEEE Des. Test 12*, 3 (Fall 1995), 70-84.

LAKSHMINARAYANA, G., RAGHUNATHAN, A., KHOURI, K., JHA, N., AND DEY, S. 1999. Common-case computation: A high-level technique for power and performance optimization. In *Proceedings of the Conference on Design Automation* (June), 56–61.

LEE, M. T.-C., FUJITA, M., TIWARI, V., AND MALIK, S. 1997. Power analysis and minimization techniques for embedded DSP software. *IEEE Trans. Very Large Scale Integr. Syst. 5*, 1, 123–135.

LEE ET AL., W. 1997. A 1-V programmable DSP for wireless communications. *IEEE J. Solid-State Circuits 32*, 11 (Nov.), 1766–1776.

LEE, E. AND SANGIOVANNI-VINCENTELLI, A. 1998. A framework for comparing models of computation. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 17*, 12 (Dec.), 1217–1229.

LEKATSAS, H. AND WOLF, W. 1998. Code compression for embedded systems. In *Proceedings of the 35th Annual Conference on Design Automation* (DAC '98, San Francisco, CA, June 15–19), B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Eds, ACM Press, New York, NY, 516–521.

LI, Y. AND HENKEL, J.-R. 1998. A framework for estimation and minimizing energy dissipation of embedded HW/SW systems. In *Proceedings of the 35th Annual Conference on Design Automation* (DAC '98, San Francisco, CA, June 15–19), B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Eds, ACM Press, New York, NY, 188–193.

LIAO, S., DEVADAS, S., AND KEUTZER, K. 1998. Code density optimization for embedded DSP processors using data compression techniques. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 17*, 7 (July), 601–608.

LIDSKY, D. AND RABAEY, J. M. 1996. Early power exploration—a World Wide Web application. In *Proceedings of the 33rd Annual Conference on Design Automation* (DAC '96, Las Vegas, NV, June 3–7), T. P. Pennino and E. J. Yoffa, Eds. ACM Press, New York, NY, 27–32.

LIDSKY, D. AND RABAEY, J. 1994. Low-power design of memory intensive functions. In *Proceedings of the IEEE Symposium on Low Power Electronics* (Sept.), IEEE Computer Society Press, Los Alamitos, CA, 16–17.

LITCH, T. AND SLATON, J. 1998. Portable communications. *IEEE Micro 18*, 2 (Apr.), 48–55.

LORCH, J. AND SMITH, A. 1998. Software strategies for portable computer energy management. *IEEE Personal Commun. 5*, 3 (June).

LORCH, J. R. AND SMITH, A. J. 1997. Scheduling techniques for reducing processor energy use in MacOS. *Wireless Networks 3*, 5, 311–324.

LU, Y., CHUNG, E. Y., IMUNI, T., BENINI, L., AND DE MICHELI, G. 2000. Quantitative comparison of power management algorithms, DATE. In *Proceedings of the Conference on Design Automation and Test in Europe* (Mar.),

LUDWIG, J., NAWAB, H., AND CHANDRAKASAN, A. 1996. Low-power digital filtering using approximate processing. *IEEE J. Solid-State Circuits 31*, 3 (Mar.), 395–399.

MACII, E., PEDRAM, M., AND SOMENZI, F. 1998. High-level power modeling, estimation and optimization. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 17*, 11 (Nov.), 1061–1079.

MARTIN, T. AND SEWIOREK, D. 1999. The impact of battery capacity and memory bandwith on CPU speed-setting: A case study. In *Proceedings of the International Symposium on Low Power Electronics and Design* (June), 200–205.

MARTIN, T. AND SIEWIOREK, D. 1996. A power metric for mobile systems. In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design* (ISLPED '96, Monterey, CA, Aug 12–14), M. Horowitz, J. Rabaey, B. Barton, and M. Pedram, Eds. IEEE Press, Piscataway, NJ, 37–42.

MEHRA, R., GUERRA, L. M., AND RABAEY, J. M. 1996. Low-power architectural synthesis and the impact of exploiting locality. *J. VLSI Signal Process. 13*, 2/3, 239–258.

MEHRA, R., GUERRA, L., AND RABAEY, J. 1997. A partitioning scheme for optimizing interconnect power. *IEEE J. Solid-State Circuits 32*, 3 (Mar.), 433–443.

MEHTA, H., OWENS, R. M., AND IRWIN, M. J. 1996. Some issues in Gray code addressing. In *Proceedings of the Great Lakes Symposium on VLSI* (Ames, IA), IEEE Computer Society Press, Los Alamitos, CA, 178–180.

MEHTA, H., OWENS, R. M., IRWIN, M. J., CHEN, R., AND GHOSH, D. 1997. Techniques for low energy software. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design* (ISLPED '97, Monterey, CA, Aug. 18–20), B. Barton, M. Pedram, A. Chandrakasan, and S. Kiaei, Eds. ACM Press, New York, NY, 72–75.

MENDL, J. 1995. Low power microelectronics: Retrospect and prospect. *Proc. IEEE 83*, 4 (Apr.), 619–635.

MENG, T., GORDON, B., TSENG, E., AND HUNG, A. 1995. Portable video-on-demand in wireless communication. *Proc. IEEE 83*, 4 (Apr.), 659–690.

MICROSOFT AND TOSHIBA. 1996. Advanced configuration and power interface specification. Tech. Rep.

MICROSOFT. 1997. OnNow: the evolution of the PC platform. Microsoft Press, Redmond, WA.

MONTEIRO, J., DEVADAS, S., ASHAR, P., AND MAUSKAR, A. 1996. Scheduling techniques to enable power management. In *Proceedings of the 33rd Annual Conference on Design Automation* (DAC '96, Las Vegas, NV, June 3–7), T. P. Pennino and E. J. Yoffa, Eds. ACM Press, New York, NY, 349–352.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

MUSOLL, E., LANG, T., AND CORTADELLA, J. 1998. Working-zone encoding for reducing the energy in microprocessor address buses. *IEEE Trans. Very Large Scale Integr. Syst. 6*, 4, 568–572.

MUTOH, S., SHIGEMATSU, S., MATSUYA, Y., FUKUDA, H., KANEKO, T., AND YAMADA, J. 1996. A 1-V multithreshold-voltage CMOS digital signal processor for mobile phone applications. *IEEE J. Solid-State Circuits 31*, 11 (Nov.), 1795–1802.

NACHTERGAELE, L., MOOLENAAR, D., VANHOOF, B., CATTHOOR, F., AND DE MAN, H. 1998. System-level power optimization of video codecs on embedded cores: a systematic approach. *J. VLSI Signal Process. 18*, 2, 89–109.

NAMGOONG, W., YU, M., AND MENG, T. 1997. A high-efficiency variable-voltage CMOS dynamic DC-DC switching regulator. In *Proceedings of the IEEE International Conference on Solid-State Circuits*, IEEE Computer Society Press, Los Alamitos, CA, 380–381.

NAWAB, S. H., OPPENHEIM, A. V., CHANDRAKASAN, A. P., WINOGRAD, J. M., AND LUDWIG, J. T. 1997. Approximate signal processing. *J. VLSI Signal Process. 15*, 1-2, 177–200.

NIELSEN, L. S. AND NIESSEN, C. 1994. Low-power operation using self-timed circuits and adaptive scaling of the supply voltage. *IEEE Trans. Very Large Scale Integr. Syst. 2*, 4 (Dec. 1994), 391–397.

NISHITANI, T. 1999. Low-power architectures for programmable multimedia processors. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci. E82-A*, 2 (Feb.), 184–196.

PANDA, P. AND DUTT, N. 1999. Low-power memory mapping through reducing address bus activity. *IEEE Trans. Very Large Scale Integr. Syst. 7*, 3 (Sept.), 309–320.

PANWAR, R. AND RENNELS, D. 1995. Reducing the frequency of tag compares for low power I-cache design. In *Proceedings of the 1995 International Symposium on Low Power Design*

(ISLPD-95, Dana Point, CA, Apr. 23–26), M. Pedram, R. Brodersen, and K. Keutzer, Eds. ACM Press, New York, NY, 57–62.

PEDRAM, M. 1996. Power minimization in IC design: Principles and applications. *ACM Trans. Des. Autom. Electron. Syst. 1*, 1, 3–56.

PEDRAM, M. AND VAISHNAV, H. 1997. Power optimization in VLSI layout: A survey. *J. VLSI Signal Process. 15*, 3, 221–232.

PEDRAM, M. AND WU, Q. 1999. Design considerations for battery-powered electronics. In *Proceedings of the Conference on Design Automation* (June), 861–866.

PERING, T., BURD, T., AND BRODERSEN, R. 1998. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design* (ISLPED '98, Monterey, CA, Aug. 10–12), A. Chandrakasan and S. Kiaei, Eds. ACM Press, New York, NY, 76–81.

POTKONJAK, M. AND RABAEY, M. 1999. Algorithm selection: A quantitative optimization-intensive approach. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 18*, 5 (May), 524–532.

PUTERMAN, M. 1994. *Finite Markov Decision Processes*. John Wiley and Sons, Inc., New York, NY.

QU, G. AND POTKONJAK, M. 1998. Techniques for energy minimization of communication pipelines. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '98, San Jose, CA, Nov. 8–12), H. Yasuura, Ed. ACM Press, New York, NY, 597–600.

RABAEY, J. AND PEDRAM, M. 1996. *Low Power Design Methodologies*. Kluwer Academic, Dordrecht, Netherlands.

RAGHUNATHAN, A. AND JHA, N. 1997. SCALP: An iterative improvement-based low-power datapath synthesis algorithm. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 14*, 11 (Nov.), 1260–1277.

RAGHUNATHAN, A., JHA, N., AND DEY, S. 1998. *High-Level Power Analysis and Optimization*. Kluwer Academic, Dordrecht, Netherlands.

RAGHUNATHAN, A., DEY, S., AND JHA, N. 1999. Register transfer level power optimization with emphasis on glitch analysis and reduction. *IEEE Trans. Comput.-Aided Des. 18*, 8 (Aug.), 1114–1131.

RAJE, S. AND SARRAFZADEH, M. 1995. Variable voltage scheduling. In *Proceedings of the 1995 International Symposium on Low Power Design* (ISLPD-95, Dana Point, CA, Apr. 23–26), M. Pedram, R. Brodersen, and K. Keutzer, Eds. ACM Press, New York, NY, 9–14.

RAMPRASAD, S., SHANBHAG, N., AND HAJJ, I. 1998. Signal coding for low power: Fundamental limits and practical realizations. In *Proceedings of the IEEE International Symposium on Circuits and Systems* (ISCAS), IEEE Computer Society Press, Los Alamitos, CA.

ROSS, S. 1997. *Introduction to Probability Models*. Academic Press, Inc., New York, NY.

SACHA, J. AND IRWIN, M. 1998. Number representation for reducing switching capacitance in subband coding. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing* (May), 12–15.

SAIED, R. AND CHAKRABARTI, C. 1996. Scheduling for minimizing the number of memory accesses in low power applications. In *VLSI Signal Processing* 169–178.

SAN MARTIN, R. AND KNIGHT, J. 1996. Optimizing power in ASIC behavioral synthesis. *IEEE Des. Test 13*, 2, 58–70.

SEGARS, S., CLARKE, K., AND GOUDGE, L. 1995. Embedded control problems, thumb and the ARM7TDMI. *IEEE Micro 15*, 5 (Dec.), 22–30.

SHIN, Y. AND CHOI, K. 1999. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the Conference on Design Automation* (June), 134–139.

SHIUE, W. AND CHAKRABARTI, C. 1999. Memory exploration for low power, embedded systems. In *Proceedings of the Conference on Design Automation* (June), 140–145.

ŠIMUNIĆ, T., BENINI, L., AND DE MICHELI, G. 1999. Cycle-accurate simulation of energy consumption in embedded systems. In *Proceedings of the Conference on Design Automation* (June), 867–872.

ŠIMUNIÈ, T., BENINI, L., AND DE MICHELI, G. 1999. Energy-efficient design of battery-powered embedded systems. In *Proceedings of the International Symposium on Low Power Electornics and Design* (June), 212–217.

SRATAKOS, A., SANDERS, S., AND BRODERSEN, R. 1994. A low-voltage CMOS DC-DC converter for a portable battery-operated system. In *Proceedings of the IEEE Conference on Power Electronics Specialists*, IEEE Computer Society Press, Los Alamitos, CA, 619–626.

SRIVASTAVA, M. B., CHANDRAKASAN, A. P., AND BRODERSEN, R. W. 1996. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Trans. Very Large Scale Integr. Syst. 4*, 1, 42–55.

SRIVASTAVA, M. B. AND POTKONJAK, M. 1996. Power optimization in programmable processors and ASIC implementations of linear systems: Transformation-based approach. In *Proceedings of the 33rd Annual Conference on Design Automation* (DAC '96, Las Vegas, NV, June 3–7), T. P. Pennino and E. J. Yoffa, Eds. ACM Press, New York, NY, 343–348.

STAN, M. R. AND BURLESON, W. P. 1995. Bus-invert coding for low-power I/O. *IEEE Trans. Very Large Scale Integr. Syst. 3*, 1 (Mar. 1995), 49–58.

STAN, M. R. AND BURLESON, W. P. 1997. Low-power encodings for global communication in CMOS VLSI. *IEEE Trans. Very Large Scale Integr. Syst. 5*, 4, 444–455.

STEMM, M. AND KATZ, R. 1997. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci. E80-B*, 8 (Aug.), 1125–1131.

SU, C. L., TSUI, C. Y., AND DESPAIN, A. M. 1994. Saving power in the control path of embedded processors. *IEEE Des. Test 11*, 4 (Winter), 24–30.

SU, C.-L. AND DESPAIN, A. M. 1995. Cache design trade-offs for power and performance optimization: a case study. In *Proceedings of the 1995 International Symposium on Low Power Design* (ISLPD-95, Dana Point, CA, Apr. 23–26), M. Pedram, R. Brodersen, and K. Keutzer, Eds. ACM Press, New York, NY, 63–68.

SUZUKI ET AL., K. 1997. A 300 MIPS/W RISC core processor with variable supply-voltage scheme in variable threshold-voltage CMOS. In *Proceedings of the Conference on Custom Integrated Circuits* (May), 112–118.

TIWARI, V., MALIK, S., AND WOLFE, A. 1994. Power analysis of embedded software: a first step towards software power minimization. *IEEE Trans. Very Large Scale Integr. Syst. 2*, 4 (Dec. 1994), 437–445.

TIWARI, V., MALIK, S., WOLFE, A., AND LEE, M. T.-C. 1996. Instruction level power analysis and optimization of software. *J. VLSI Signal Process. 13*, 2/3, 223–238.

TIWARI, V., SINGH, D., RAJGOPAL, S., MEHTA, G., PATEL, R., AND BAEZ, F. 1998. Reducing power in high-performance microprocessors. In *Proceedings of the 35th Annual Conference on Design Automation* (DAC '98, San Francisco, CA, June 15–19), B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Eds, ACM Press, New York, NY, 732–737.

TOMIYAMA, H., ISHIHARA, T., INOUE, A., AND YASUURA, H. 1998. Instruction scheduling for power reduction in processor-based system design. In *Proceedings of the Conference on Design, Automation and Test in Europe 98*, 855–860.

USAMI, K. AND IGARASHI ET AL., M. 1998. Automated low-power technique exploiting multiple supply voltages applied to a media processor. *IEEE J. Solid-State Circuits 33*, 3 (Mar.), 463–472.

USAMI, K., IGARASHI, M., ISHIKAWA, T., KANAZAWA, M., TAKAHASHI, M., HAMADA, M., ARAKIDA, H., TERAZAWA, T., AND KURODA, T. 1998. Design methodology of ultra low-power MPEG4 codec core exploiting voltage scaling techniques. In *Proceedings of the 35th Annual Conference on Design Automation* (DAC '98, San Francisco, CA, June 15–19), B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Eds, ACM Press, New York, NY, 483–488.

VERBAUWHEDE, I. AND TOURIGUIAN, M. 1998. A low power DSP engine for wireless communications. *J. VLSI Signal Process. 18*, 2, 177–186.

VITTOZ, E. 1994. Low power microelectronics: Ways to approach the limits. In *Proceedings of the International Conference on Solid-State Circuits* (Jan.), 14–18.

WAN, M., ICHIKAWA, Y., LIDSKY, D., AND RABAEY, J. 1998. An energy conscious methodology for aarly design exploration of heterogeneous DSPs. In *Proceedings of the Conference on Custom Integrated Circuits* (May), 111–117.

WEI, G. AND HOROWITZ, M. 1996. A low power switching power supply for self-clocked systems. In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design* (ISLPED '96, Monterey, CA, Aug 12–14), M. Horowitz, J. Rabaey, B. Barton, and M. Pedram, Eds. IEEE Press, Piscataway, NJ, 313–317.

WEISER, M., WELCH, B., DEMERS, A., AND SHENKER, S. 1994. Scheduling for reduced CPU energy. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation* (Monterey, CA, May), USENIX Assoc., Berkeley, CA, 13–23.

WINZKER, M. 1998. Low-power arithmetic for the processing of video signals. *IEEE Trans. Very Large Scale Integr. Syst. 6*, 3 (Sept.).

WOLF, W. 1994. Hardware-software co-design of embedded systems. *Proc. IEEE 82*, 7 (July 1994), 967–989.

WOLFE, A. 1996. Issues for low-power CAD tools: A system-level design study. *J. Des. Autom. Embedded Syst. 1*, 4, 315–332.

WUYTACK, S., CATTHOOR, F., AND DE MAN, H. 1997. Transforming set data types to power optimal data structures. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 15*, 6 (June), 619–629.

WUYTACK, S., DIGUET, J.-P., CATTHOOR, F. V. M., AND DE MAN, H. J. 1998. Formalized methodology for data reuse exploration for low-power hierarchical memory mappings. *IEEE Trans. Very Large Scale Integr. Syst. 6*, 4, 529–537.

YOSHIDA, Y., SONG, B.-Y., OKUHATA, H., ONOYE, T., AND SHIRAKAWA, I. 1997. An object code compression approach to embedded processors. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design* (ISLPED '97, Monterey, CA, Aug. 18–20), B. Barton, M. Pedram, A. Chandrakasan, and S. Kiaei, Eds. ACM Press, New York, NY, 265–268.

ZHANG, H. AND RABAEY, J. 1998. Low-swing interconnect interface circuits. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design* (ISLPED '98, Monterey, CA, Aug. 10–12), A. Chandrakasan and S. Kiaei, Eds. ACM Press, New York, NY, 161–166.

ZHANG, Y., HU, X., AND CHEN, D. 1999. Low energy register allocation beyond basic blocks. In *Proceedings of the International Symposium on Circuits and Systems* (June), 290–293.

ZYUBAN, V. AND KOGGE, P. 1998. The energy complexity of register files. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design* (ISLPED '98, Monterey, CA, Aug. 10–12), A. Chandrakasan and S. Kiaei, Eds. ACM Press, New York, NY, 305–310.

Received: December 1999; accepted: February 2000