Chapter #16

Energy-efficient system-level design

Luca Benini [1] and Giovanni De Micheli [2]
1 Università di Bologna Bologna – Italy

2 CSL - Stanford University Stanford - CA - USA

Abstract: The complexity of current and future integrated systems requires a paradigm shift towards component-based design technologies that enable the integration of large computational cores, memory hierarchies and communication channel as well as system and application software onto a single chip. Moving from a set of case studies, we give an overview of energy-efficient system-level design, emphasizing a component-based approach.

Key words: Embedded systems, Memory hierarchy, Network-on-Chip, Chip Multiprocessor, System software, Application software, Power management

1. INTRODUCTION

A system is a collection of components whose combined operation provides a useful service. We consider specifically *systems on chips* (SoCs). Such systems consist of hardware components integrated on a single chip and various software layers. Hardware components are macro-cells that provide information processing, storage, and interfacing. Software components are programs that realize system and application functions.

When analyzing current SoC designs, it is apparent that systems are described and realized as collections of components. Indeed, to date, there is limited use of behavioral synthesis at the system level. System implementation by component interconnection allows designers to realize complex functions while leveraging existing units and/or design technologies, such as synthesis, on components whose size is much smaller than the system itself.

Sometimes, system specifications are required to fit into specific interconnections of components called *hardware platforms*. Thus, a hardware platform, which is a restriction of the design space, may facilitate system realization because it reduces the number of design options and fosters the use and reuse of standard components. Expertise with designing systems on a known platform is also a decisive factor in reducing design time and in increasing designers' confidence in success.

System design consists of realizing a desired functionality while satisfying some design constraints. Broadly speaking, constraints limit the design space and relate to the major design trade-off between *quality of service* (QoS) versus cost. QoS is closely related to performance, i.e., the number of tasks that can be computed in a time window (system throughput), as well as the time delay to complete a task (latency). QoS relates also to the system *dependability*, i.e., to a class of specific system figures (e.g., reliability, availability, safety) that measure the ability of the system to deliver a service correctly, within a given time window and at any time. Design cost relates to design and manufacturing costs (e.g., silicon area, testability) as well as to operation costs (e.g., power consumption, energy consumption per task).

In recent years, the design trade-off of performance versus power consumption has received large attention because of: (i) 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 because of heat extraction, and (iii) concerns about operating costs caused by electric power consumption in large systems and the dependability of systems operating at high temperatures because of power dissipation. Dependability measures will be extremely relevant in the near future because of the use of SoCs in safety-critical applications (e.g., vehicular technologies) and in devices that connect humans with services (e.g., portable terminals used to manage finances and working activities).

Recent design methodologies and tools have been addressing the problem of *energy-efficient design*, aiming at providing a high-performance realization while reducing its power dissipation. Most of these techniques, as described in the previous chapters, address system components design. The objective of this chapter is to describe current techniques that address system-level design.

2. SYSTEMS ON CHIPS AND THEIR DESIGN

We attempt to characterize SOC designs based on trends and technologies. Electronic systems are best implemented on a single chip because input-output pins are a scarce resource, and because on-chip interconnect is faster and more reliable while overall cost is usually smaller. At present, it is possible to integrate opto-electronic units on chip (e.g., charge-coupled device cameras) and mechanical elements (e.g., accelerometers) even though systems with such components go beyond the scope of this chapter. In some domains, e.g., digital telephony, there is a definite trend to cluster all electronics of a product on a single die.

Current near-future electronic technologies provide designers with an increasingly larger number of transistors per chip. Standard, CMOS silicon-based technologies with feature size around 100nm are considered here. Such technologies support half a billion transistor chips of a few square centimeters in size, according to the *international technology semiconductor roadmap* (ITRS). As device sizes will further shrink to 50nm by the end of the decade, chips will accommodate up to four billion transistors. Whereas the increased amount of active devices will support increasingly more complex design, chip power dissipation will be capped around 175W because of packaging limitations and costs. Thus, the computing potential is limited by energy efficiency.

At the same time, the design of large (i.e., billion transistor) chips will be limited by the ability of humans and *computer-aided design* (CAD) tools to tame their complexity. The million-transistor chip frontier was overcome by using semi-custom technologies and cell libraries in the 1990s. Billion-transistor chips will be designed with methodologies that limit design options and leverage both libraries of very large scale components and generators of embedded memory arrays.

Such library components are typically processors, controllers, and complex functional units (e.g., MPEG macro-cells). System designers will accept such components as basic building blocks as they are used to accepting NAND and NOR gates without questioning their layout. At the same time, successful component providers are expected to design reliable and flexible units that can interact with others under varying operating conditions and modes. Post-design, possibly *in situ* software (or programmable hardware) configuration of these components, will play a major role in achieving versatile components.

When observing any SoC layout, it is simple to recognize large memory arrays. The ability to realize various types of embedded memories on chip and the interspersion of storage and computing units are key to achieving high-performance. The layout of embedded memory arrays is automatically generated by physical synthesis tools and can be tailored in size, aspect ratio, and speed.

The distinguishing features of the upcoming SoCs relate directly to the features and opportunities offered by semiconductor technology. Namely, SoCs will display many processing elements (i.e., cores) and memory arrays. Multi-processing will be the underlying characteristic of such chips. Thus SoC technology will provide for both the implementation of multi-processing computing systems and application-specific functions. The latter class of systems is likely to be large and will be the driving force for SoC technology. Indeed, embedded systems will be realized by SoCs realizing a specific function, e.g., vehicular control, processing for wireless communication, etc. Application specific SoCs will be characterized by the presence of processing units running embedded software (and thus emulating hardware functions) and by asymmetric structures, due to the diversity of functions realized by the processing elements and their different storage requirements.

The presence of several, possibly application-specific, on-chip storage arrays presents both an opportunity and a design challenge. Indeed, the use of hierarchical storage that exploits spatial and temporal locality by interspersing processing elements and storage arrays is key to achieving high throughput with low latency [57, 70, 75]. The sizing and synthesis of embedded storage arrays poses new challenges, because the effectiveness of multi-processing is often limited by the ability to transfer and store information. SoCs will generate large data traffic on chip; the energy spent to process data is likely to be dwarfed by the energy spent to move and store data. Thus, the design of the on-chip communication and storage systems will be key in determining the energy/performance trade-off points of an implementation.

The use of processing cores will force system designers to treat them as black boxes and renounce the detailed tuning of their performance/energy parameters. Nevertheless, many processing elements are designed to operate at different service levels and energy consumption, e.g., by controlling their operation frequency and voltage. Thus system designers will be concerned with run-time power management issues, rather than with processing element design.

As a result, the challenging issues in system-level design relate to designing the storage components and the interconnect network of SoCs. At the same time, designers must conceive run-time environments that manage processing elements, memory, and on-chip network to provide for the workload-dependent operating conditions, which yield the desired quality of service with minimal energy consumption. In other words, SoCs will require dedicated operating systems that provide for power management.

The overall design of system and application software is crucial for achieving the overall performance and energy objectives. Indeed, while software does not consume power per se, the software execution causes energy consumption by processing elements, storage arrays, and interconnect network. It is well known that software design for a SoC is at least as demanding as hardware design. For this reason, software design issues will be covered in this chapter.

The remaining of this chapter is organized as follows. First a set of recent SoC examples is considered to motivate this survey. Next the storage array and interconnect network design on chip is address. The chapter concludes with a survey of software design techniques, for both system and application software.

3. SOC CASE STUDIES

This section analyzes three SoC designs from an energy-centric perspective. It is organized in order of tightening power and cost constraints, starting from a 3D graphics engine for game consoles, moving to a MPEG4 encoder-decoder for 3G wireless terminals, and concluding with an audio recorder for low-end consumer applications. Clearly, this survey gives a very partial view of an extremely variegated landscape, but its purpose is to focus on the key design challenges in power-constrained integrated system design and to enucleate system design guidelines that have lead to successful industrial implementations.

3.1 Emotion Engine

The *Emotion Engine* [78, 41] was designed by Sony and Toshiba to support 3-D graphics for the PlayStation 2 game console. From a functional viewpoint, the design objective was to enable real-time synthesis of realistic animated scenes in three dimensions. To achieve the desired degree of realism, physical modeling of objects and their interactions, as well as 3-D geometry, transformation are required. Power budget constraints are essentially set by cost considerations: the shelf price of a game console should be lower than 500$, thus ruling out expensive packaging and cooling. Furthermore, game consoles should be characterized by the low cost of ownership, robustness with respect to a wide variety of operating conditions, and minimal maintenance. All of these requirements conflict with high power dissipation. These challenges were met by following two fundamental design guidelines: (i) integration of most of the critical communication, storage, and computation on a single SoC, and (ii) architectural specialization for a specific class of applications.

The architecture of the Emotion Engine is depicted in *Figure 1*. The system integrates three independent processing cores and a few smaller I/O controllers and specialized coprocessors. The main CPU, the master controller, is a superscalar RISC processor with a floating-point coprocessor. The other two cores are floating-point vector processing units. The first vector unit, VPU0, performs physical modeling computations, while the second, VPU1, is dedicated to 3-D geometry computation. These two functions are allocated to two different vector units because their schedules are conflicting. Physical modeling is performed under the control of the main CPU, and it is scheduled quite irregularly and unpredictably. In contrast, geometry computations are performed in response to requests from the rendering engine, which are spaced in equal time increments.

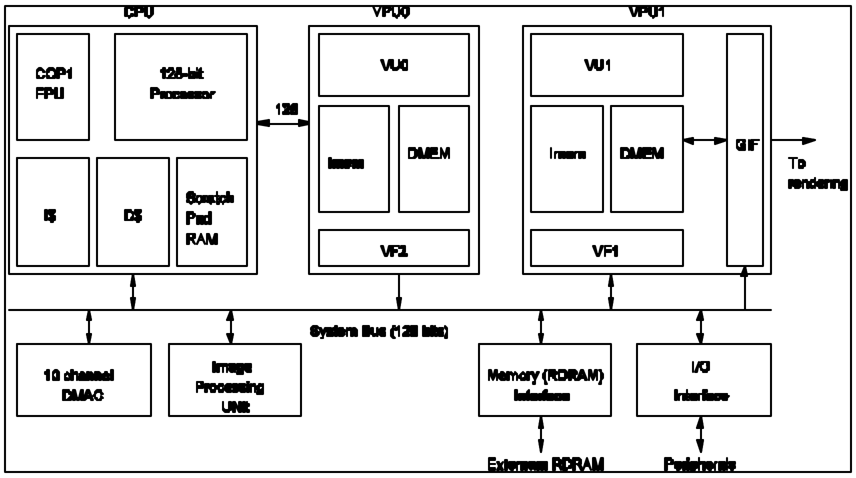

Figure 1: Architecture of the Emotion Engine

The main CPU is a two-way superscalar RISC core implementing the MIPS III instruction set, plus 107 new SIMD multimedia instructions. The core has 32 128-bit registers and two 64-bit integer units. Instruction and data caches are two-way set associative, 16-KB and 8-KB, with one-cycle access. Local data storage is also supported by a 16-KB scratch-pad RAM (one-cycle access). The vector units VPU0 and VPU1 have similar micro-architectures. However, VPU0 works as a coprocessor of the main CPU, while VPU1 operates independently. The vector units have a four-way SIMD organization. Instruction memory is 64-bits wide and its size is 16-KB (for VPU1, 4-KB for VPU0). To provide single-cycle data feed to the floating-point units, four pipelined buffers are instantiated within the VPUs. The quad-buffer appears as a 16-KB (for VPU1, 4-KB for VPU0), 4-ported memory.

Communication is critical for system performance. VPU1 works independently from the processor and produces a very large amount of data for the external rendering engine. Therefore, there is a dedicated connec tion and I/O port between VPU1 and the rendering engine. In contrast, VPU0 receives data from the CPU (as a coprocessor). For this reason data transfer from/to the unit is stored in the CPU's scratch-pad memory and transferred to VPU0 via DMA on a shared, 128-bit interconnection bus. The bus supports transfers among the three main processors, the coprocessors, and I/O blocks (e.g., for interfacing with high-bandwidth RDRAM).

The Emotion Engine was fabricated in 0.25 ?m technology with 0.18 ?m drawn gate le ngth for improved switching speed. The CPU and the VPUs are clocked at 250MHz. External interfaces are clocked at 125 MHz. Die size is 17 ? 14.1 mm^2. The chip contains 10.5 million transistors. The chip can sustain 5 GFLOPs. With power supply V_{DD} = 1.8 V, the power consumption is 15 W. Clearly, such a power consumption is not adequate for portable, battery-operated equipment; however it is much lower than that of a general-purpose microprocessor with similar FP performance (in the same technology).

The energy efficiency of the Emotion Engine stems form several factors. First it contains many fast SRAM memories, providing adequate bandwidth for localized data transfers but not at the high energy cost implied by cache memories. On the contrary, instruction and data caches have been kept small, and it is up to the programmer to develop tight inner loops that minimize misses. Second, the architecture provides an extremely aggressive degree of parallelism without pushing the envelope for maximum clock speed. Privileging parallelism with respect to sheer speed is a well-known low-power design technique [10]. Third, parallelism is explicit in hardware and software (the various CPUs have well-defined tasks), and it is not compromised by centralized hardware structures that impose unacceptable global communication overhead. The only global communication channel (the on-chip system bus) is bypassed by dedicated ports for high-bandwidth point-to-point communication (e.g., between VPU1 and the rendering hardware). Finally, the SoC contains many specialized coprocessors for common functions (e.g., MPEG2 video decoding), which unloads the processors and achieves very high energy efficiency and locality. Specialization is also fruitfully exploited in the micro-architecture of the programmable processors, which natively support a large number of application-specific instructions.

3.2 MPEG4 Core

In contrast with the Emotion Engine, the MPEG4 video codec SoC described by Takahashi et al. [80] has been developed specifically for the highly power-constrained mobile communications market. Baseband processing for a multimedia-enabled 3G wireless terminal encompasses several complex tasks that can, in principle, be implemented by multiple ICs. However, it is hard to combine many chips within the small body of a mobile terminal, and, more importantly, the high-bandwidth I/O interfaces among the various ICs would lead to excessive power consumption. For this reason, Takahashi et al. opted for an SoC solution that integrates most of the digital baseband functionality. The SoC implements a video codec, a speech codec or an audio decoder, and multiplexing and de-multiplexing between multiple video and speech/audio streams.

Video processing is characterized by large data streams from/to memory, and memory space requirements are significant. For this reason, the MPEG4 video codec has been implemented in an embedded-DRAM process. The abstracted block diagram on the SoC is shown in *Figure 2*. The chip contains 16-Mb embedded DRAM and three signal processing cores: a video core, a speech/audio core, and a stream-multiplexing core. Several peripheral interfaces (camera, display, audio, and an external CPU host for configuration) are also implemented on-chip.

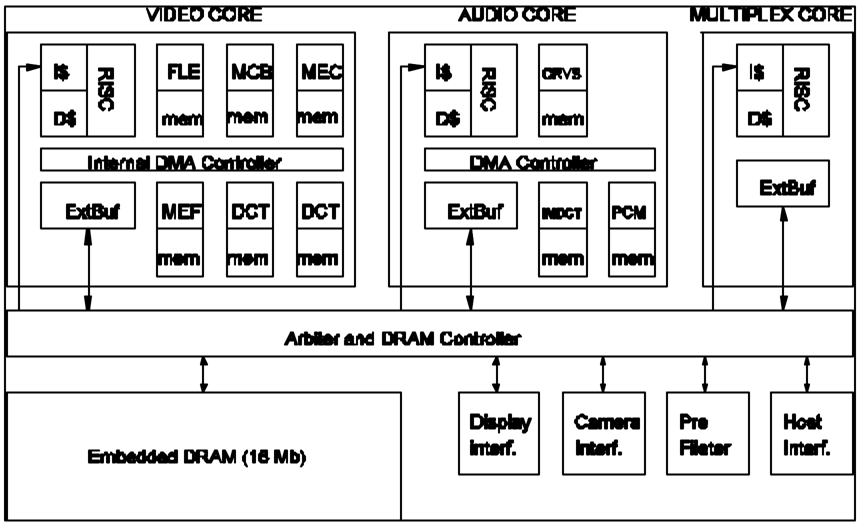

Figure 2: Architecture of the Decoder.

Each of the major signal processing cores contains a 16-bit RISC processor and dedicated hardware accelerators. The system is a three-way asymmetric on-chip multiprocessor. Data transfers among the three processors are performed via the DRAM. A virtual FIFO is configured on the DRAM for each processor pair. The size of the FIFOs can be changed by the firmware of each core. The communication network is organized as a set of point-to-point channels between processors and DRAM. An arbitration unit regulates access to the DRAM, based on DMA. Most of the traffic on the channels is caused by cache and local memory refills issued by the three processing cores. Communication among processors is sporadic.

The video processing core of the SoC contains a multimedia-enhanced RISC processor with a 4-Kb direct mapped instruction cache and a 8-Kb data cache. The video processor also includes several custom coprocessors: 2 DCT coprocessors, a motion compensation block, two motion estimation blocks, and a filter block. All hardware accelerators have local SRAM buffers for limiting the number of accesses to the shared DRAM. The total SRAM memory size is 5.3 Kb. The video processing core supports concurrent execution in real time of one encoding thread and up to four decoding threads. The audio core has a similar organization. It also contains an RISC processor with caches, but it includes different coprocessors. The multiplexing core contains a RISC processor and a network interface block, and it handles tasks without the need for hardware accelerators.

The MPEG4 core targets battery-powered portable terminals, hence, it has been optimized for low power consumption at the architectural, circuit, and technology level. Idle power reduction was a primary concern. Therefore, clock gating is adopted throughout the chip; the local clock is automatically stopped whenever processors or hardware accelerators are idle. Shutdown is also supported at a coarser granularity: all RISC processors support sleep instructions for explicit, software-controlled shutdown, with interrupt-based wake-up. Active power minimization is tackled primarily through the introduction of embedded DRAM, which drastically reduces IO, bus, and memory access energy. Memory tailoring reduces power by 20% with respect to a commodity-DRAM solution. Page and word size have been chosen to minimize redundant data fetch and transfer, and specialized access modes have been defined to improve latency and throughput.

To further reduce power, the SoC was designed in a 0.25 ? m *variable-threshold CMOS* technology with V_{DD}=2.5 V. In active mode, the threshold voltage of transistors is 0.55 V. In standby mode it is raised through body-bias to 0.65 V to reduce leakage. The chip contains 20.5 million transistors, chip area is 10.84 ? 10.84 mm^2. The 16-Mb embedded DRAM occupies roughly 40% of the chip. The chip consumes 260 mW at 60 MHz. Compared to a previous design, with external commodity DRAM and separate video and audio processing chips, power is reduced by roughly a factor of four.

Comparing the MPEG4 core with the Emotion Engine from a power viewpoint, one notices that the second SoC consumes roughly 60 times less power than the first one at a comparable integration level. The differences in speed and voltage supply account for a difference in power consumption of, roughly, a factor of 2, which becomes a factor of 4 if one discounts area (i.e., focuses on power density). The residual 15 ? difference is due to the different transistor usage (the MPEG4 core is dom inated by embedded DRAM, which has low power density), and to architecture, circuit, and technology optimizations. This straightforward comparison convincingly demonstrates the impact of power -aware system design techniques and the impressive flexibility of CMOS technology.

3.3 Single -chip Voice Recorder

Digital audio is a large market where system cost constraints are extremely tight. For this reason, several companies are actively pursuing single-chip solutions based on embedded memory for the on-chip storage of sound samples [84, 43]. The main challenges are the cost per unit area of semiconductor memory, and the power dissipation of the chip, which should be as low as possible to reduce the cost of batteries (e.g., primary Lithium vs. rechargeable Li-Ion).

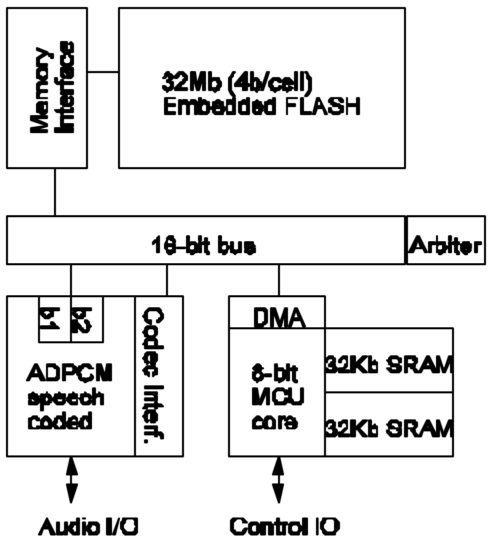

Figure 3: Architecture of the voice recorder.

The single-chip voice recorder and player developed by Borgatti and coauthors [14] stores recorded audio samples on embedded FLASH memory. The chip was originally implemented in 0.5 ?m technology with 3.0 V supply, and it is a typical example of an SoC designed for a single application. The main building blocks (*Figure 3*) are: a microcontroller unit (MCU), a speech coder and decoder, and an embedded FLASH memory. A distinguishing feature of the system is the use of a multi-level storage scheme to increase the speech recording capacity of the FLASH. Speech samples are first digitized then compressed with a simple waveform coding technique (adaptive-differential pulse-code modulation) and finally stored in FLASH memory, 4-bits per cell.

A 4-bits per cell density requires 16 different thresholds for the FLASH cells. Accurate threshold programming and readout requires mixed-signal circuitry in the memory write and read paths. The embedded FLASH macro contains 8 Mcells. It is divided into 128 sectors that can be independently erased. Each sector contains 64-K cells, which can store 32 Kbytes in multilevel mode. Memory read is performed though an 8-bit, two-step analog-to-digital converter.

Besides the multilevel FLASH memory, the other main components of the SoC are the 8-bit MCU, the ADCPM speech codec, and the 16-bit on-chip bus. The core interfaces to two 32kB embedded RAM blocks (one for storing data and the other for executable code and data). The two blocks are split into 16 selectively accessed RAM modules to reduce power consumption. The executable code is downloaded to program RAM from dedicated sectors of the FLASH macro though 16-bit DMA transfers on the on-chip bus. A few code blocks (startup code, download control code, and other low-level functions) are stored in a small ROM module (4-kB).

The speech codec is a custom datapath block implementing ITU-T G.726 compression (ADPCM). Its input/output ports are in PCM format for directly interfacing to a microphone and a loudspeaker. At a clock speed of 128 kHz, a telephone-quality speech signal can be compressed at one of four selectable bit rates (16-40 kB/s). The compressed audio stream is packed in blocks of 1 kB using two on-chip RAM buffers (in a two-phase fashion). This organization guarantees that samples can be transferred to FLASH in blocks, at a much higher burst rate than the sample rate.

The on-chip bus is synchronous and 16-bits wide, and it supports multiple masters and interrupts. A bus arbiter manages mutual exclusion and resolves access conflicts. A static priority order is assigned to all bus masters at initialization time, but it can be modified through a set of dedicated signals. The on-chip bus can be clocked at different speeds (configured through a software accessible register). Each block is clocked at a different speed by a dedicated clock. All clocks are obtained by dividing an externally provided 16-32 MHz clock. Clock gating was used extensively to reduce the power consumption of idle sub-circuits.

The chip is fabricated in a 3 V 0.5 ?m common-ground NOR embedded FLASH process. The chip area is 15 ? 15 mm^2, and it has only 26 logically active pins. Standby power is less than 1mW. Peak power during recordin g is 150 mWand 110 mW during play. The average power increases with higher bit rates, but it is generally much smaller than peak power (e.g., 75mW for recording at 24 kbps).

The single-chip recorder demonstrates power minimization principles that have not been fully exploited in the SoCs examined in the previous subsections. The use of application-specific processing units is pushed one step further. Here, the programmable processor has only control and coordination functions. All computationally expensive data processing is farmed off to a specialized datapath block. An additional quantum leap in energy efficiency is provided by mixed-signal or analog implementation of key functional blocks. In this chip, analog circuits are used to support 16-bit per cell programming density in the embedded FLASH memory. The 16-fold density increase for embedded memory represents a winning point from the energy viewpoint as well.

4. DESIGN OF MEMORY SYSTEMS

The SoCs analyzed in the previous section demonstrate that today's integrated systems contain a significant amount of storage arrays. In many cases the fraction of silicon real estate devoted to memory is dominant, and the power spent in accessing memories dictates the overall chip power consumption. The general trend in SoC integration is toward increasing embedded memory content [56]. It is reported that, on average, 50% of the transistors in an SoC designed in 2001 are instantiated within memory arrays. This percentage is expected to grow to 70% by 2003 [29]. In view of this trend it is obvious that energy-efficient memory system design is a critical issue.

The simplest memory organization, the *flat* memory, assumes that data is stored in a single, large array. Even in such a simplistic setting, sizing memory arrays is not trivial. Undersized memories penalize system performance, while oversized memories cost in terms of silicon area as well as performance and power, because access time and power increase monotonically with memory size [51, 17].

The most obvious way to alleviate memory bottlenecks is to reduce the storage requirements of the target application. To this goal, designers can reduce memory requirements by exploiting the principle of temporal locality, i.e., trying to reuse the results of a computation as soon as possible, in order to reduce the need for temporary storage. Other memory-reduction techniques aim at finding efficient data representations that reduce the amount of unused information stored in memory. Storage reduction techniques cannot completely remove memory bottlenecks, mainly because they try to optimize power and performance indirectly as a by-product of the reduction of memory size. As a matter of fact, memory size requirements of system applications have steadily increased over time.

From the hardware design viewpoint, memory power reduction has been pursued mainly through technology and circuit design and through a number of architectural optimizations. While technology and circuit techniques are reviewed in detail in previous chapters, architectural optimizations, which rely on the idea of overcoming the scalability limitation intrinsic of flat memories, are focused on here. Indeed, hierarchical memories allow the designer to exploit the spatial locality of reference by clustering related information into the same (or adjacent) arrays.

4.1 On-chip Memory Hierarchy

The concept of a *memory hierarchy*, conceptually depicted in *Figure 4*, is at the basis of most on-chip memory optimization approaches. Lower levels in the hierarchy are made of small memories, tightly coupled with processing units. Higher hierarchy levels are made of increasingly larger memories, placed relatively far from computation units, and possibly shared.

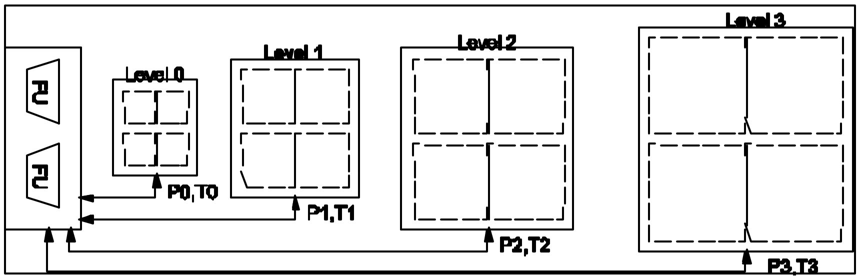

Figure 4: A generic hierarchical memory model.

When looking at the hierarchical structure of computational and storage nodes, the distance between a computation unit and a storage array represents the effort needed to fetch (or store) a data unit from (to) the memory. The main objective of energy-efficient memory design is to minimize the overall energy cost for accessing memory within performance and memory size constraints. Hierarchical organizations reduce memory power by exploiting non-uniformity (or locality) in access.

Memory optimization techniques can be classified into three categories:

?? *Memory hierarchy design.* Given a dynamic trace of memory accesses, obtained by profiling an application, derive a customized memory hierarchy.
?? *Computation transformation.* Given a fixed memory hierarchy, modify the storage requirements and access patterns of the target computation to optimally match the given hierarchy.
?? *Synergistic memory and computation optimization*. Concurrently optimize memory access patterns and memory architecture.

Memory-hierarchy design is considered next. Computation transformations are software-oriented techniques (see Section 5). For a comprehensive survey of the topic, with special emphasis on synergistic techniques, refer to [10, 66].

When comparing time and energy per access in a memory hierarchy, one can observe that they both increase with the move from low to high hierarchy levels. One may be led to conclude that a low-latency memory architecture will also be a low-power architecture and that memory performance optimization implies power optimization. This conclusion is often incorrect for three main reasons. First, even though both power and performance increase with memory size and memory hierarchy levels, they do not increase by the same amount. 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, the impact of a new memory architecture on all memory accesses, not only the critical ones, needs to be considered. Third, several circuit-level techniques actually trade shorter access time for higher power (and vice versa) at a constant memory size. The following example, taken from [74], demonstrates how energy and performance can be contrasting quantities.

Example 1 *The memory organization options for a two-level memory hierarchy (on-chip cache and off-chip main memory) explored in [74] are the following: (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); and (iv) off-chip memory size, from 2Mbit SRAM, to 16Mbit SRAM.*

The exhaustive exploration of the cache organization for minimum energy for an MPEG decoding application results in an energy-optimal cache organization with cache size 64 bytes, line size 4 bytes, 8-way set associative. Notice that this is a very small memory, almost fully associative (only two lines). For this organization, the total memory energy is 293 ?J, and the execution time is 142,000 cycles. In contrast, exploration for maximum performance yields a cache size of 512 bytes, a line size of 16 bytes, and is 8-way set associative. Notice that this cache is substantially larger than the energy-optimal one. In this case, the execution time is reduced to 121,000 cycles, but the energy becomes 1,110 ?J.

One observes that the second cache dominates the first one for size, line size, and associativity; hence, it has the larger hit rate. This is consistent with the fact that performance strongly depends on 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 hit rate can be traded 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.

The example shows that energy cannot generally be reduced as a byproduct of performance optimization. On the other hand, architectural solutions originally devised for performance optimization are often beneficial in terms of energy. Generally, when locality of access is improved, both performance and energy tend to improve. This fact is heavily exploited in software optimization techniques.

4.2 Explorative Techniques

Several recently proposed memory optimization techniques are explorative. They exploit the fact that the memory design space can usually be parameterized and discretized, to allow for an exhaustive or near-exhaustive search. Most approaches assume a memory hierarchy with one or more levels of caching and, in some cases, an off-chip memory. A finite number of cache sizes and cache organization options are considered (e.g., degree of associativity, line size, cache replacement policy, as well as different off-chip memory alternatives--number of ports, available memory cuts). The best memory organization is obtained by simulating the workload for all possible alternative architectures. The various approaches mainly differ in the number of hierarchy levels that are covered by the exploration or the number of available dimensions in the design space. Su and Despain [77], Kamble and Ghose [37], Ko and Balsara [42], Bahar *at al*. [4], and Shiue and Chakrabarti [74] focus on cache memories. Zyuban and Kogge

[94] study register files; Coumeri and Thomas [21] analyze embedded SRAMs; Juan *et al*. [44] study translation look-aside buffers.

Example 1 has shown an instance of a typical design space and the result of the relative exploration. An advantage of explorative techniques is that they allow for concurrent evaluation of multiple cost functions such as performance and area. The main limitation of the explorative approach is that it requires extensive data collection, which provides a posteriori insight. In order to limit the number of simulations, only a relatively small set of architectures can be tested and compared.

4.3 Memory Partitioning

Within a hierarchy level, power can be reduced by memory partitioning. The principle of memory partitioning is to sub-divide the address space and to map blocks to different physical memory banks that can be independently enabled and disabled. Arbitrary fine partitioning is prevented due to the fact that a large number of small banks is area inefficient and imposes a severe wiring overhead, which tends to increase communication power and performance.

Partitioning techniques can be applied at all hierarchy levels, from register files to off-chip memories. Another aspect is the “type” of partitioning, such as physical or logic partitioning. *Physical* partitioning strictly maps the address space onto different, non-overlapping memory blocks. *Logic* partitioning exploits some redundancy in the various blocks of the partition, with the possibility of addresses that are stored several times in the same level of hierarchy.

A physically-partitioned memory is energy-efficient mainly for two reasons. First, if accesses have high spatial and/or temporal locality, individual memory banks are accessed in bursts. Burst access to a single bank is desirable because idle times for all other banks are long, thereby amortizing the cost of shutdown [28]. Second, energy is saved because every access is on a small bank as opposed to a single large memory [77]. For embedded systems designed with a single application target, application profiling can be exploited to derive a tailored memory partition, where small memory banks are tightly fitted on highly-accessed address ranges, while “colder” regions of the address space can be mapped onto large banks. Clearly, such a non-uniform memory partitioning strategy can out perform equi-partition when access profiles are highly non-uniform and are known at design time [56].

Logic partitioning was proposed by Gonzalez *et al.* [30], where the on-chip cache is split into a *spatial* and into a *temporal* cache to store data with high spatial and temporal correlation, respectively. This approach relies on a dynamic prediction mechanism that can be realized without modification to the application code by means of a prediction buffer.

A similar idea is proposed by Milutinovic *et al*. [61], where a split spatial/temporal cache with different line sizes is used. Grun *at al.* [32] exploit this idea in the context of embedded systems for energy optimization. Data are statically mapped to the either cache, using the high predictability of the access profiles for embedded applications, and thus avoiding the hardware overhead of the buffer. Depending on the application, data might be duplicated and thus be mapped to both caches. Another class of logic partitioning techniques falls within the generic scheme of *Figure 5*. Buffers are put along the I-cache and/or the D-cache, to realize some form of cache parallelization. Such schemes can be regarded as a partitioning solution because the buffers and the caches are actually part of the same level of hierarchy.

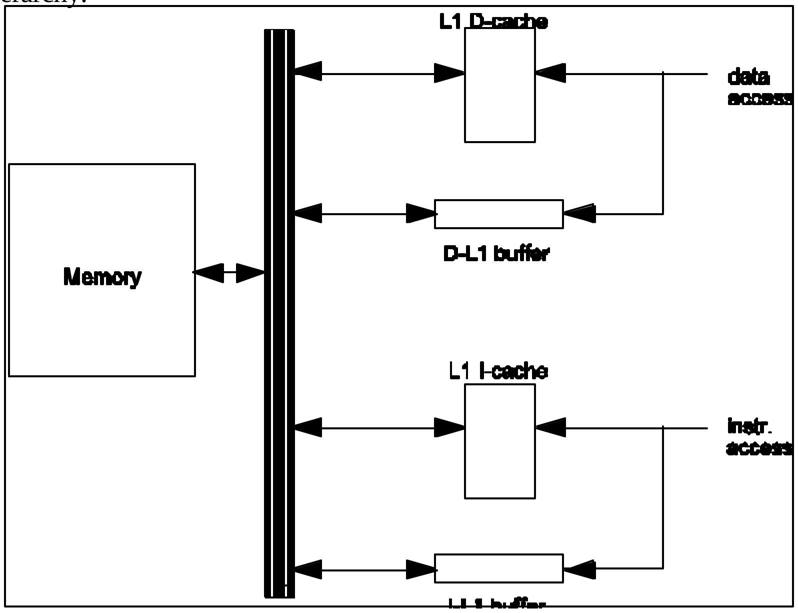

Figure 5: Using buffers together with caches.

4.4 Extending the Memory Hierarchy

Memory partitioning extends the “width” of the memory hierarchy by splitting, with or without replication, a given hierarchy level. An alternative possibility is offered by modifying its “depth”, i.e., the number of hierarchy levels. This option does not just imply the straightforward addition of extra levels of caching.

A first class of techniques is based on the insertion of “ad-hoc” memories between existing hierarchy levels. This approach is particularly useful for instruction memory, where access locality is very high. Pre-decoded instruction buffers [6] store instructions in critical loops in a pre-decoded fashion, thereby decreasing both fetch and decode energy. Loop caches [40] store the most frequently executed instructions (typically contained in small loops) and can bypass even the first-level cache. Notice that these additional memories would not be useful for performance if the first-level cache can be accessed in a single cycle. On the contrary, performance can be slightly worsened because the access time for the loop cache is on the critical path of the memory system.

Another approach is based on the replacement of one or more levels of caches with more energy-efficient memory structures. Such structures are usually called *scratch-pad* buffers and are used to store a portion of the off-chip memory, in an explicit fashion. In contrast with caches, reads and writes to the scratch-pad memory are controlled explicitly by the programmer. Clearly, allocation of data to the scratch pad should be driven by profiling and statistics collection. These techniques are particularly effective in application-specific systems, which run an application mix whose memory profiles can be studied a priori, thus providing intuitive candidates for the addresses to be put into the buffer. The work by Panda *et al*. [63, 64] is probably the most comprehensive effort in this area [64].

4.5 Bandwidth Optimization

When the memory architecture is hierarchical, memory transfers become a critical facet of memory optimization. From a performance viewpoint, both memory latency and bandwidth are critical design metrics [35]. From an energy viewpoint, memory bandwidth is much more critical than latency. Optimizing memory bandwidth implies reducing the average number of bits that are transferred across the boundary between two hierarchy levels in a time unit. It has been pointed out [16] that memory bandwidth is becoming more and more important as a metric for modern systems, because of the increased instruction-level parallelism generated by superscalar or VLIW processors and because of the density of integration that allows shorter latencies. Unlike latency, bandwidth is an average-case quantity. Well-known latency-reduction techniques, such as prefetching, are inefficient in terms of bandwidth (and energy).

As an example of bandwidth optimization, the work by Burger *et al.* [15, 16] introduces several variants of traffic-efficient caches that reduce unnecessary memory traffic by the clever choice of associativity, block size, or replacement policy, as well as clever fetch strategies fetches. These solutions do not necessarily improve worst-case latency but result in reduced read and writes across different memory hierarchy levels, thus reducing energy as well.

Another important class of bandwidth optimization techniques is based on the compression of the information passed between hierarchy levels. These techniques aim at reducing the large amount of redundancy in instruction streams by storing compressed instructions in the main memory and decompressing them on the fly before execution. Compression finds widespread application in wireless networking, where channel bandwidth is severely limited. In memory compression, the constraints on the speed and hardware complex ity of the compressor and decompressor are much tighter than in macroscopic networks. Furthermore, memory transfers usually have very fine granularity (they rarely exceed a few tens of bytes). Therefore, the achieved compression ratios are usually quite lo w, but compression speed is very high. Hardware-assisted compression has been applied mainly to instruction memory, [89, 50, 49, 9] and, more recently, to data memory [11]. A comprehensive survey of memory compression techniques can be found in [47].

5. DESIGN OF INTERCONNECT NETWORKS

As technology improves and device sizes scale down, the energy spent on processing and storage components decreases. On the other hand, the energy for global communication does not scale down. On the contrary, projections based on current delay optimization techniques for global wires [79] show that global communication on chip will require increasingly higher energy consumption.

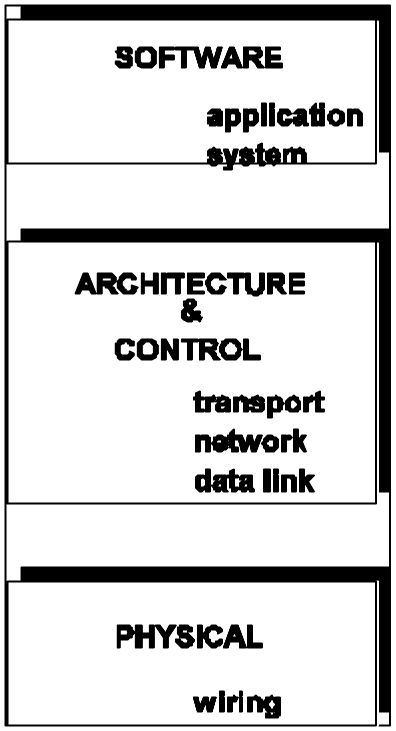

Figure 6: The on-chip network stack

The chip interconnect has to be considered and designed as an on-chip network, called a *micro-network* [8]. As for general network design, a layered abstraction of the micro-network (shown in *Figure 6*) can help us analyze the design problems and find energy-efficient communication solutions. Next, micro-network layers are considered in a bottom-up fashion. First, the problems due to the physical propagation of signals on chip are analyzed. Then general issues related to network architectures and control protocols are considered. Protocols are considered independently from their implementation, from the physical to the transport layers. The discussion of higher-level layers is postponed until Section 5. Last, we close this section by considering techniques for energy-efficient communication on micro-networks.

5.1 Signal transmission on chip

Global wires are the physical implementation of on-chip communication channels. Physical-layer signaling techniques for lossy transmission lines have been studied for a long time by high-speed board designers and microwave engineers [5, 24].

Traditional rail-to-rail voltage signaling with capacitive termination, as used today for on-chip communication, is definitely not well-suited for high-speed, low-energy communication on future global interconnects [24]. Reduced swing, current-mode transmission, as used in some processor-memory systems, can significantly reduce communication power dissipation while preserving speed of data communication.

Nevertheless, as technology trends lead us to use smaller voltage swings and capacitances, error probabilities will rise. Thus the trend toward faster and lower-power communication may decrease reliability as an unfortunate side effect. Reliability bounds can be derived from theoretical (entropic) considerations [34] and measured by experiments on real circuits as voltages scale.

A paradigm shift is needed to address the aforementioned challenges. Current design styles consider wiring-related effects as undesirable parasitics and try to reduce or cancel them by specific and detailed physical design techniques. It is important to realize that a well-balanced approach should not over-design wires so that their behavior approaches an ideal one because the corresponding cost in performance, energy-efficiency and modularity may be too high. Physical-layer design should find a compromise between competing quality metrics and provide a clean and complete abstraction of channel characteristics to micro-network layers above.

5.2 Network architectures and control protocols

Due to the limitations at the physical level and to the high bandwidth requirement, it is likely that SoC design will use network architectures similar to those used for multi-processors. Whereas *shared medium* (e.g., bus-based) communication dominates today's chip designs, scalability reasons make it reasonable to believe that more general network topologies will be used in the future. In this perspective, micro-network design entails the specification of *network architectures* and *control protocols* [27]. The architecture specifies the topology and physical organization of the interconnection network, while the protocols specify how to use network resources during system operation.

The *data-link layer* abstracts the physical layer as an unreliable digital link, where the probability of bit errors is non null (and increasing as technology scales down). Furthermore, reliability can be traded for energy [34, 12]. The main purpose of data-link protocols is to increase the reliability of the link up to a minimum required level, under the assumption that the physical layer by itself is not sufficiently reliable.

An additional source of errors is contention in shared-medium networks. Contention resolution is fundamentally a non-deterministic process because it requires synchronization of a distributed system, and for this reason it can be seen as an additional noise source. In general, non-determinism can be virtually eliminated at the price of some performance penalty. For instance, centralized bus arbitration in a synchronous bus eliminates contention-induced errors, at the price of a substantial performance penalty caused by the slow bus clock and by bus request/release cycles.

Future high-performance shared-medium on-chip micro-networks may evolve in the same direction as high-speed local area networks, where contention for a shared communication channel can cause errors, because two or more transmitters are allowed to send data on a shared medium concurrently. In this case, provisions must be made for dealing with contention-induced errors.

An effective way to deal with errors in communication is to *packetize* data. If data is sent on an unreliable channel in packets, error containment and recovery is easier because the effect of the errors is contained by packet boundaries, and error recovery can be carried out on a packet-by-packet basis. At the data-link layer, error correction can be achieved by using standard *error-correcting codes* (ECC) that add redundancy to the transferred information. Error correction can be complemented by several packet-based error detection and recovery protocols. Several parameters in these protocols (e.g., packet size, number of outstanding packets, etc.) can be adjusted depending on the goal to achieve maximum performance at a specified residual error probability and/or within given energy consumption bounds. At the relatively low noise levels typical of on-chip communication, recent research results [12] indicate that error recovery is more energy-efficient than forward error correction, but it increases the variance in communication latency.

At the *network layer*, packetized data transmission can be customized by choosing switching or routing algorithms. The former, (e.g., *circuit*, *packet*, and *cut-through* switching), establishes the type of connection while the latter determines the path followed by a message through the network to its final destination. Switching and routing for on-chip micro-networks affect the performance and energy consumption heavily. Future approaches will most likely emphasize speed and the decentralization of routing decisions [1]. Robustness and fault tolerance will also be highly desirable.

At the *transport layer*, algorithms deal with the decomposition of messages into packets at the source and their assembly at the destination. Packetization granularity is a critical design decision, because the behavior of most network control algorithms is very sensitive to packet size. Packet size can be application-specific in SoCs, as opposed to general networks. In general, flow control and negotiation can be based on either deterministic or statistical procedures. Deterministic approaches ensure that traffic meets specifications and provide hard bounds on delays or message losses. The main disadvantage of deterministic techniques is that they are based on worst cases, and they generally lead to significant under-utilization of network resources. Statistical techniques are more efficient in terms of utilization, but they cannot provide worst-case guarantees. Similarly, from an energy viewpoint, deterministic schemes are expected to be more inefficient than statistical schemes because of their implicit worst-case assumptions.

5.3 Energy-efficient design: techniques and examples

This section delves into a few specific instances of energy-efficient micro-network design problems. In most cases, specific solutions that have been proposed in the literature are also outlined, although it should be clear that many design issues are open and significant progress in this area is expected in the near future.

5.3.1 Physical Layer At the physical layer, low-swing signaling is actively investigated to reduce communication energy on global interconnects [92]. In the case of a simple CMOS driver, low-swing signaling is achieved by lowering the driver's supply voltage V_{dd}. This implies a quadratic dynamic-power reduction (because $P_{dyn} = K\ V_{dd}^2$). Unfortunately, swing reduction at the transmitter complicates the receiver's design. Increased sensitivity and noise immunity are required to guarantee reliable data reception. Differential receivers have superior sensitivity and robustness, but they require doubling the bus width. To reduce the overhead, pseudo-differential schemes have been proposed, where a *reference* signal is shared among several bus lines and receivers, and incoming data is compared against the reference in each receiver. Pseudo-differential signaling reduces the number of signal transitions, but it has reduced noise margins with respect to fully-differential signaling. Thus, reduced switching activity is counterbalanced by higher swings, and determining the minimum-energy solution requires careful circuit-level analysis.

Another key physical-layer issue is synchronization. Traditional on-chip communication has been based on the synchronous assumption, which implies the presence of global synchronization signals (*i.e.*, clocks) that define data sampling instants throughout the chip. Unfortunately, clocks are extremely energy-inefficient, and it is a well-known fact that they are responsible for a significant fraction of the power budget in digital integrated systems. Thus, postulating global synchronization when designing on-chip micro-networks is not an optimal choice from the energy viewpoint. Alternative on-chip synchronization protocols that do not require the presence of a global clock have been proposed in the past [93,7] but their effectiveness has not been studied in detail from the energy viewpoint.

5.3.2 Data-link layer At the data-link layer, a key challenge is to achieve the specified communication reliability level with minimum energy expense. Several error recovery mechanisms developed for macroscopic networks can be deployed in on-chip micro-networks, but their energy efficiency should be carefully assessed in this context. As a practical example, consider two alternative reliability-enhancement techniques: *error-correcting codes* and *error-detecting codes with retransmission*. Both approaches are based on transmitting redundant information over the data link, but error-correction is generally more demanding than error detection in terms of redundancy and decoding complexity. Hence, we can expect error-correcting transmission to be more power-hungry in the error-free case. However, when an error arises, error-detecting schemes require retransmission of the corrupted data. Depending on the network architecture, retransmission can be very costly in terms of energy (and performance).

Clearly, the trade-off between the increased cost of error correction and the energy penalty of retransmission should be carefully explored when designing energy-efficient micro-networks [34]. Either scheme may be optimal, depending on system constraints and on physical channel characteristics. Automatic design space exploration could be very beneficial in this area.

Bertozzi et al. [12] considered *error-resilient codes* for 32-bit buses. Namely, they consider Hamming encoding/decoding schemes that support single-error correction, double-error detection, and (non-exhaustive) multi-error detection. The physical overhead of these schemes is 6 or 7 additional bus lines plus the encoders and decoders. When error is detected and not corrected, data retransmission occurs. When error is not detected, the system has a catastrophic failure. For a given reliability specification of mean time to failure (MTTF) - ranging from 10 years to a few milliseconds - it is possible to determine the average energy per useful bit that is transmitted under various hypotheses. Such hypotheses include wiring length, and thus the ratio of energy spent on wires over the energy spent in coding, and voltage swings. In particular, for long MTTF (10^{15} sec) and wires (5 pF), error detection with retransmission is more energy-efficient than forward error correction, mainly for two reasons. First, for the same level of redundancy, error detection is more robust than error correction; hence, the signal-to-noise ratio can be lowered more aggressively. Second, the error-detecting decoder is simpler and consumes less power than the error-correcting decoder. These two advantages overcome retransmission costs, which are sizable, but they are incurred under the relatively rare occurrence of transmission errors.

In case of shared-medium network links (such as busses), the media-access-control function of the data link layer is also critical for energy efficiency. Currently, centralized time-division multiplexing schemes (also called centralized arbitration) are widely adopted [3, 20, 86]. In these schemes, a single arbiter circuit decides which transmitter accesses the bus for every time slot. Unfortunately, the poor scalability of centralized arbitration indicates that this approach is likely to be energy-inefficient as micro-network complexity scales up. In fact, the energy cost of communicating with the arbiter and the hardware complexity of the arbiter itself scale up more than linearly with the number of bus masters.

Distributed arbitration schemes as well as alternative multiplexing approaches, such as code division multiplexing, have been extensively adopted in shared-medium macroscopic networks and are actively being investigated for on-chip communication [90]. However, research in this area is just burgeoning, and significant work is needed to develop energy-aware media-access-control for future micro-networks.

5.3.3 Network layer Network architecture heavily influences communication energy. As hinted in the previous section, shared-medium networks (busses) are currently the most common choice, but it is intuitively clear that busses are not energy-efficient as network size scales up [33]. In bus-based communication, data is always broadcasted from one transmitter to all possible receivers, while in most cases messages are destined to only one receiver or a small group of receivers. Bus contention, with the related arbitration overhead, further contributes to the energy overhead.

Preliminary studies on energy-efficient on-chip communication indicate that hierarchical and heterogeneous architectures are much more energy-efficient than busses [68, 93]. In their work, Zhang *et al*. [93] develop a *hierarchical generalized mesh* where network nodes with a high communication bandwidth requirement are clustered and connected through a programmable generalized mesh consisting of several short communication channels joined by programmable switches. Clusters are then connected through a generalized mesh of global long communication channels. Clearly such architecture is heterogeneous because the energy cost of intra-cluster communication is much smaller than that of inter-cluster communication. While the work of Zhang *et al*. demonstrates that power can be saved by optimizing network architecture, many network design issues are still open, and tools and algorithms are needed to explore the design space and to tailor network architecture to specific applications or classes of applications.

Network architecture is only one facet of network layer design, the other major facet being network control. A critical issue in this area is the choice of a switching scheme for indirect network architectures. From the energy viewpoint, the tradeoff is between the cost of setting up a circuit-switched connection once for all and the overhead for switching packets throughout the entire communication time on a packet-based connection. In the former case the network control overhead is “lumped” and incurred once, while in the latter case, it is distributed over many small contributions, one for each packet. When communication flow between network nodes is extremely persistent and stationary, circuit-switched schemes are likely to be preferable, while packet-switched schemes should be more energy-efficient for irregular and non-stationary communication patterns. Needless to say, circuit switching and packet switching are just two extremes of a spectrum, with many hybrid solutions in between [85].

5.3.4 Transport layer Above the network layer, the communication abstraction is an end-to-end connection. The transport layer is concerned with optimizing the use of network resources and providing a requested quality of service. Clearly, energy can be seen as a network resource or a component in a quality of service metric. An example of a transport-layer design issue is the choice between connection-oriented and connectionless protocols. Energy efficiency can be heavily impacted by this decision. In fact, connection-oriented protocols can be energy inefficient under heavy traffic conditions because they tend to increase the number of re-transmissions. On the other hand, out-of-order data delivery may imply additional work at the receiver, which causes additional energy consumption. Thus, communication energy should be balanced against computation energy at destination nodes.

Another transport-layer task with far-reaching implications on energy is flow control. When many transmitters compete for limited communication resources, the network becomes congested, and the cost per transmitted bit increases because of increased contention and contention resolution overhead. Flow control can mitigate the effect of congestion by regulating the amount of data that enters the network at the price of some throughput penalty. Energy reduction by flow control has been extensively studied for wireless networks [85, 67], but it is an unexplored research area for on-chip micro-networks.

6. SOFTWARE

Systems have several software layers running on top of the hardware. Both system and application software programs are considered here.

Software does not consume energy per se, but it is the execution and storage of software that requires energy consumption by the underlying hardware. Software execution corresponds to performing operations on hardware, as well as storing and transferring data. Thus software execution involves power dissipation for computation, storage, and communication. Moreover, storage of computer programs in semiconductor memories requires energy (e.g., 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. Nevertheless, reducing the size of the stored programs is beneficial. This can be achieved by compilation (see Section 5.2.2) and code compression. In the latter case, the compiled instruction stream is compressed before storage. At run time, the instruction stream is decompressed on the fly. Besides reducing the storage requirements, instruction compression reduces the data traffic between memory and processor and the corresponding energy cost. (See also Section 3.5.) Several approaches have been devised to reduce instruction fetch-and-store overhead, as surveyed in [56]. The following subsections focus mainly on system-level design techniques to reduce the power consumption associated with the execution of software.

6.1 System software

The notion of *operating system* (OS) is generalized to capture the system programs that provide support for the operation of SoCs. Note that the system support software in current SoCs usually consists of ad hoc routines, designed for a specific integrated core processor, under the assumption that a processor provides global, centralized control for the system. In future SoCs, the prevailing paradigm will be peer-to-peer interaction among several, possibly heterogeneous, processing elements. Thus, system software will be designed as a modular distributed system. Each programmable component will be provided with system software to support its own operation, to manage its communication with the communication infrastructure, and to interact effectively with the system software of the neighboring components.

Seamless composition of components around the micro-network will require the system software to be configurable according to the requirements of the network. Configuration of system software may be achieved in various ways, ranging from manual adaptation to automatic configuration. At one end of the spectrum, software optimization and compactness are privileged; at the other end, design ease and time are favored. With this vision, on-chip communication protocols should be programmable at the system software level, to adapt the underlying layers (e.g., transport) to the characteristics of the components.

Let us now consider the broad objectives of system software. For most SoCs, which are dedicated to some specific application, the goal of system software is to provide the required quality of service within the physical constraints. Consider, for example, an SoC for a wireless mobile video terminal. Quality of service relates to the video quality, which implies specific performance levels of the computation and storage elements as well as of the micro-network. Constraints relate to the strength and S/N ratio of the radio-frequency signal and to the energy available in the battery. Thus, the major task of system software is to provide high performance by orchestrating the information processing within the service stations and providing the "best" information flow. Moreover, this task should be achieved while keeping energy consumption to a minimum.

The system software provides us with an abstraction of the underlying hardware platform. In a nutshell, one can view the system as a queuing network of service stations. Each service station models a computational or storage unit, while the queuing network abstracts the micro-network. Moreover, one can assume that:

?? Each service station can operate at various service levels, providing corresponding performance and energy consumption levels. This abstracts the physical implementation of components with adjustable voltage and/or frequency levels, as well as with the ability to disable their functions in full or in part.

?? The information flow between the various units can be controlled by the system software to provide the appropriate quality of service. This entails controlling the routing of the information, the local buffering into storage arrays, and the rate of the information flow.

In other words, the system software must support the *dynamic power management* (DPM) of its components as well as *dynamic information-flow management*.

6.1.1 Dynamic Power Management *Dynamic power management* (DPM) is a feature of the run-time environment of an electronic system that dynamically reconfigures it to provide the requested services and performance levels with a minimum number of active components or a minimum activity level on such components [9]. 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 often realized by throttling the frequency of processor operation (and possibly stopping the clock) and/or reducing the power supply voltage. *Dynamic frequency scaling* (DFS) and *dynamic voltage scaling* (DVS) are the terms commonly used to denote power management over a range of values. Typically, DVS is used in conjunction with DFS since reduced voltage operation requires lower operating frequencies, while the converse is not true.

The fundamental premise for the applicability of DPM is that systems (and their components) experience non-uniform workloads during operation time. Such an assumption is valid for most systems, both when considered in isolation and when inter-networked. 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.

Designing power-managed systems encompasses several tasks, including the selection of power-manageable components with appropriate characteristics, determining the power management *policy* [9], and implementing the policy at an appropriate level of system software. DPM was described in a previous Chapter. This chapter considers only the relations between DPM policy implementation and system software.

A power management policy is an algorithm that observes requests and states of one or more components and issues commands related to frequency and voltage settings. This chapter also considers the limiting cases of turning on/off the clock and/or the power supply to a component. Whereas policies can be implemented in hardware (as a part of the control-unit of a component), software implementations achieve much greater flexibility and ease of integration. Thus a policy can be seen as a program that is executed at run-time by the system software.

The simplest implementation of a policy is by a *filter driver*, i.e., by a program attached to the software driver of a specific component. The driver monitors the traffic to/from the component and has access to the component state. Nevertheless, the driver has a limited view of other components. Thus such an implementation of power management may suffer from excessive locality.

Power management policies can be implemented in system kernels and be tightly coupled to process management. Indeed, process management has knowledge of currently-executing tasks and tasks coming up for execution. Process managers also know which components (devices) are needed by each task. Thus, policy implementation at this level of system software enjoys both a global view and an outlook of the system operation in the near future. Predictive component wake-up is possible with the knowledge of upcoming tasks and required components.

The system software can be designed to improve the effectiveness of power management. Power management exploits idle times of components. The system software scheduler can sequence tasks for execution with the additional goal of clustering component operation, thus achieving fewer but longer idle periods. Experiments with implementing DPM policies at different levels of system software [55] have shown increasing energy savings as the policies have deeper interaction with the system software functions.

6.1.2 Information-flow management Dynamic information-flow management relates to configuring the micro-network and its bandwidth to satisfy the information flow requirements. This problem is tightly related to DPM and can be seen as an application of DPM to the micro-network instead of to components. Again, policies implemented at the system software layer request either specific protocols or parameters at the lower layers to achieve the appropriate information flow, using the least amount of resources and energy.

An example of information-flow management is provided by the Maia processor [91], which combines an ARM8 processor core with 21 satellite units, including processing and storage units. The ARM8 processor configures the memory-mapped satellites using a 32bit configuration bus, and communicates data with satellites using two pairs of I/O interface ports and direct memory read/writes. Connections between satellites are through a 2-level hierarchical mesh-structured reconfigurable network. Dynamic voltage scaling is applied to the ARM8 core to increase energy efficiency.

With this approach, the micro-network can be configured before running specific applications and tailored to these applications. Thus, application programs can be spatially distributed and achieve an energy savings of one order of magnitude as compared to a DSP processor with the same performance level. Such savings are due to the ability of Maia to reconfigure itself to best match the applications, to activate satellites only when data is present, and to operate at dynamically varying rates.

6.2 Application software

The energy cost of executing a program depends on its machine code and on the corresponding micro-architecture, if one excludes the intervention of the operating system in the execution (e.g., swapping). Thus, for any given micro-architecture, the energy cost is tied to the machine code.

There are two important problems of interest: software *design* and software *compilation*. Software design affects energy consumption because the style of the software source program (for any given function) affects the energy cost. For example, the probability of swapping depends on appropriate array dimensioning while considering the hardware storage resources. As a second example, the use of specific constructs, such as guarded instructions instead of branching constructs for the ARM architecture [10], may significantly reduce the energy cost. Several efforts have addressed the problem of automatically re-writing software programs to increase their efficiency. Other efforts have addressed the generation of energy-efficient software from high-level specification. We call these techniques *software synthesis*.

Eventually, since the machine code is derived from the source code from compilation, it is the compilation process itself that affects the energy consumption. It is important to note that most compilers were written for achieving high-performing code with short compilation time. The design of an embedded system running dedicated software has brought a renewed interest in compilation, especially because of the desire of achieving high-quality code (i.e., fast, energy efficient) possibly at the expense of longer compilation time (which is tolerable for embedded systems running code compiled by the manufacturer).

For both software synthesis and compilation it is important to define the metrics of interest well. Typically, the performance (e.g., latency) and energy of a given program can be evaluated in the *worst* or *average* case. Worst-case latency analysis is relevant to real-time software design when hard timing constraints are specified. In general, average latency and average energy consumption are of interest. Average measures require the knowledge of the environment, i.e., the distribution of program inputs, which eventually affect the branches taken and the number of iterations. When such information is unavailable, meaningful average measures are impossible to achieve.

To avoid this problem, some authors have measured the performance and energy on the basic blocks, thus avoiding the effects of branching and iteration. It is often the case that instructions can be grouped into two classes. Instructions with no memory access tend to have similar energy cost and execute in a single cycle. Instructions with memory access have higher latency and energy cost. With these assumptions, reducing code size and reducing memory accesses (e.g., spills) achieves the fastest and most energy-efficient code. Nevertheless this argument breaks down when instructions (with no memory access) have non-uniform energy cost even though experimental results do not show significant variation between compilation for low latency and for low energy.

It is very important to stress that system design requires the coordination of various hardware and software components. Thus, evaluation of software programs cannot be done in isolation. Profiling techniques can and must be used to determine the frequency distribution of the values of the input to software programs and subprograms. Such information is of paramount importance for achieving application software that is energy efficient in the specific environment where it will be executed. It is also interesting to note that, given a specific environment profile, the software can be restructured so that lower energy consumption can be achieved at the price of slightly higher latency. In general, the quest for maximum performance pushes toward the speculative execution and aggressive exploitation of all hardware resources available in the system. In contrast, energy efficiency requires a more conservative approach, which limits speculation and reduces the amount of redundant work that can be tolerated for a marginal performance increase [58].

6.2.1 Software synthesis Software synthesis is a term used with different connotations. In the present context, software synthesis is an automated procedure that generates source code that can be compiled. Whereas source code programs can be synthesized from different starting points, source code synthesis from programs written in the same programming language are considered here. Software synthesis is often needed because the energy consumption of executing a program depends on the style and constructs used. Optimizing compilers are biased by the starting source code to be compiled. Recall that programs are often written with only functionality and/or performance in mind, and rarely with concerns for energy consumption. Moreover, it is common practice to use legacy code for embedded applications, sometimes with high-energy penalties. Nevertheless, it is conceivable to view this type of software synthesis as pre-processing for compilation with specific goals.

Source-level transformations. Recently several researchers have proposed source-to-source transformations to improve software code quality, and in particular energy consumption. Some transformations are directed toward using storage arrays more efficiently [17,65]. Others exploit the notion of *value locality*. Value locality is defined as the likelihood of a previously-seen value recurring repeatedly within a physical or logical storage location [52]. With value locality information, the computational cost of a program can be reduced by reusing previous computations.

Researchers have shown that value locality can be exploited in various ways depending on the target system architecture. In [46], common-case specialization was proposed for hardware synthesis using loop unrolling and algebraic reduction techniques. In [52, 48], value prediction was proposed to reduce the load/store operations with the modification of a general purpose microprocessor. Some authors [72] considered *redundant computation*, i.e., performing the same computation for the same operand value. Redundant computation can be avoided by reusing results from a *result cache*. Unfortunately, some of these techniques are architecture dependent, and thus cannot be used within a general-purpose software synthesis utility.

Next a family of techniques for source code optimization, based on specialization of programs and data, is considered. Program specialization encodes the results of previous computations in a *residual program*, while data specialization encodes these results in the data structures like caches [18]. Program specialization is more aggressive in the sense that it optimizes even the control flow, but it can lead to a code explosion problem due to over-specialization. For example, code explosion can occur when a loop is unrolled and the number of iterations is large. Furthermore, code explosion can degrade the performance of the specialized program due to increased instruction cache misses.

On the other hand, data specialization is much less sensitive to code explosion because the previous computation results are stored in a data structure that requires less memory than the textual representation of program specialization. However, this technique should be carefully applied such that the cached previous computations are expensive enough to amortize the cache access overhead. The cache can also be implemented in hardware to amortize the cache access overhead [72].

A specific instance of program specialization was proposed by Chung et al. [19]. In this approach, the computational effort of a source code program is estimated with both *value* and *execution-frequency* profiling. The most effective specializations are automatically searched and identified, and the code is transformed through *partial evaluation*. Experimental results show that this technique improves both energy consumption and performance of the source code up to more than a factor of two and in average about 35% over the original program.

Example 2 *Consider the source code in Figure 7 (a), and the first call of procedure* `foo` *in procedure* `main`*. If the first parameter* `a` *were* `0` *for all cases, this procedure could be reduced to procedure* `sp_foo` *by partial evaluation, as shown in Figure 7 (b).*

In reality, the value of parameter a *is not always* 0, *and the call to procedure* foo *cannot be substituted by procedure* sp_foo. *Instead, it can be replaced by a branching statement that selects an appropriate procedure call, depending on the result of the common value detection (CVD). The CVD procedure is named* cvd_foo *in Figure 7 (b). This is called transformation step source code alternation. Its effectiveness depends on the frequency with which* a *takes the common value* 0.

```
main () {
   int i, a, b, c[100], d[200], e, result = 0;
   ...............
   ...............
   result = foo(a, 100, c);
   for (i = 0; i < 10; i++) {
      result += foo(i, 100, c);
      result += foo(b, e, d);
      result += foo(b, 200, d);
    }
}
int foo(int fa, int fb, int *fc) {
   int i, sum = 0;
   for (i = 0; i < fb; i++)
   for(j = 0; j < fb/2; j++)
      sum += fa * fc[i];
    return sum;
}
```

(a) Original program

```
main () {
   int i, a, b, c[100], d[200], e, result = 0;
   ...............
   ...............
   if (cvd_foo(a)) result += sp_foo(b);
   else result += foo(a, 100, c);
   for (i = 0; i < 10; i++) {
      result += foo(i, 100, c);
      result += foo(b, e, d);
      result += foo(b, 200, d);
    }
}
int foo(int fa, int fb, int *fc) {
   int i, sum = 0;
   for (i = 0; i < fb; i++)
      for(j = 0; j < fb/2; j++)
         sum += fa * fc[i];
    return sum;
}
int sp_foo(int *c) {
   return 0;
}
int cvd_foo(int a) {
   if (a == 0) return 1;
   return 0;
}
```

(b) New specialized program

Figure 7: Example of source code alternation

`Software libraries.` Software engineers working on embedded systems use often software libraries, like those developed by standards groups (e.g, MPEG) or by system companies (e.g., Intel's multimedia library for the SA-1110 and TI's library for the TI'54x DSP.) Embedded operating systems typically provide a choice from a number of math and other libraries [22]. When a set of pre-optimized libraries is available, the designer has to choose the elements that perform best for a given section of the code. Such a manual optimization is error-prone and should be replaced by automated library insertion techniques that can be seen as part of software synthesis.

For example, consider a section of code that calls the `log` function. The library may contain four different software implementations: double, float, fixed point using simple bit manipulation algorithm [23, 71], and fixed point using polynomial expansion. Each implementation has a different accuracy, performance, and energy trade-off.

Thus, the automation of the use of software libraries entails two major tasks. First, characterize the library element implementations in terms of the criteria of interest. This can be achieved by analyzing the corresponding instruction flow for a given architecture. Second, recognize the sections of code that can be replaced effectively by library elements.

In the case of computation-intensive basic blocks of data-flows, code manipulation techniques based on symbolic algebra have shown to be effective in both optimizing the computation by reshaping the data flow and in performing the automatic mapping to library elements. Moreover, these tasks can be fully automated. These methods are based on the premise that in several application domains (e.g., multimedia) computation can be reduced to the evaluation of polynomials with fixed-point precision. The loss in accuracy is usually compensated by faster evaluation and lower energy consumption. Next, polynomials can be algebraically manipulated using symbolic techniques, similar to those used by tools such as Maple. Polynomial representations of computation can be also decomposed into sequences of operations to be performed by software library elements or elementary instructions. Such a decomposition can be driven by energy and/or performance minimization goals. Recent experiments have shown large energy gains on applications such as MP3 decoding [69].

6.2.2 Software Compilation Most software compilers consist of three layers: the front-end, the machine-independent optimization, and the back-end. 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 [2]. The back-end is specific to the hardware architecture, and it is often called *code generator* or *codegen*. Typically, energy-efficient compilation is performed 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 [60]. 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, which decreases the number of stalls by fetching instructions from different iterations. A third example 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 the task of allocating data to registers; when all registers are in use, data is *spilled* to the main memory. Spills are usually undesirable because of the performance and energy overhead of saving temporary information in the 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 [2], which also generate the order of the instructions. When considering compilers for application-specific architectures (e.g., DSPs), the compiler back-end is often more complex, because of 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 [31].

Energy-efficient compilation-exploiting instruction selection was proposed by Tiwari et al. [81] and tied to software analysis and determination of base costs for operations. Tiwari proposed an instruction selection algorithm based on the classical dynamic programming tree cover [2] where instruction weights are the energy costs. Experimental results showed that this algorithm yields results similar to the traditional algorithm because energy weights do not differ much in practice.

Instruction scheduling is an enumeration of the instructions consistent with the partial order induced by data and control flow dependencies. Instruction re-ordering for low-energy can be done by exploiting the degrees of freedom allowed by the partial order. Instruction re-ordering may have several beneficial effects, including reduction of inter-instruction effects [82, 53] as well as switching on the instruction bus [76] and/or in some hardware circuits, such as the instruction decoder.

Su et al. [76] proposed a technique called *cold scheduling*, which aims at ordering the instructions to reduce the inter-instruction effects. In their model, the inter-instruction effects were dominated by the switching on the internal instruction bus of 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 [25]. 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 minimizing locally the inter-instruction energy consumption on the instruction bus. Su [76] reported a reduction in overall bit switching in the range of 20 to 30%.

Register assignment aims at utilizing the available registers most effectively by reducing spills to main memory. Moreover, a register can be labeled during the compilation phase, and register assignment can be performed with the objective of reducing the switching in the instruction register as well as in the register decoders [60]. Again, the idea is to reduce the Hamming distance between pairs of consecutive register accesses. When comparing this approach to cold scheduling, note that now the instruction order is fixed, but the register labels can be changed. Metha et al. [60] proposed an algorithm that improves upon an initial register labeling by greedily swapping labels until no further switching reduction is allowed. Experimental results showed an improvement ranging from 4.2% to 9.8%.

Registers are only the last level of a memory hierarchy, which usually contains caches, buffers, multi-banked memories, etc. Compilers can have a large impact on energy consumption by optimizing not only register accesses but all kinds of memory traffic patterns as well. Many compiler transformations have limited scope, and they are not very effective in reducing memory power outside the register file. However, some restricted classes of programming constructs (namely, loop nets with data-independent iterations) can be transformed and optimized by the compiler in a very aggressive fashion. The theory and practice of loop transformations was intensely explored by parallelizing and high-performance compilers in the past [88], and it is being revisited from a memory energy minimization viewpoint with promising results [38, 39, 65]. These techniques are likely to have greater impact on SoCs because they have very heterogeneous memory architectures, and they often expose memory transfers to the programmer, as outlined in the case studies (this is rarely done in general-purpose processors).

6.2.3 Application software and power management The quest for very low energy software cost leads to the crafting and tuning of very specific application programs. Thus, a reasonable question is–why not let the application programs finely control the service levels and energy cost of the underlying hardware components? There are typically two objections to such an approach. First, application software should be independent of the hardware platform for portability reasons. Second, system software typically supports multiple tasks. When a task controls the hardware, unfair resource utilization and deadlocks may become serious problems.

For these reasons, it has been suggested [54] that application programs contain system calls that request the system software to control a hardware component, e.g., by turning it on or shutting it down, or by requesting a specific frequency and/or voltage setting. The request can be accepted or denied by the operating system, which has access to the task schedule information and to the operating levels of the components. The advantage of this approach is that OS-based power management is enhanced by receiving detailed service request information from applications and thus is in a position to make better decisions.

Another approach is to let the compiler extract the power management requests directly from the application programs at compile time. This is performed by an analysis of the code. Compiler-directed power management has been investigated for variable-voltage, variable-speed systems. A compiler can analyze the control-data flow graph of a program to find paths where execution time is much shorter than the worst-case. It can then insert voltage downscaling directives at the entry points of such paths, thereby slowing down the processor (and saving energy) only when there is sufficient slack [73].

7. CONCLUSIONS

This concluding chapter has surveyed some of the challenges in achieving energy-efficient system-level design, with specific emphasis on SoC implementation.

Digital systems with very low energy consumption require the use of components that exploit all features of the underlying technologies (as described in the previous chapters) and the realization of an effective interconnection of such components. Network technologies will play a major role in the design of future SoCs, as the communication among components will be realized as a network on chip. Micro-network architectural choices and control protocol design will be key in achieving high performance and low-energy consumption.

A large, maybe dominant, effort in SoC design is spent in writing software, because the operation of programmable components can be tailored to specific needs by means of embedded software. System software must be designed to orchestrate the concurrent operation of on-chip components and network. Dynamic power management and information-flow management are implemented at the system software level, thus adding to the complexity of its design. Eventually, application software design, synthesis, and compilation will be crucial tasks in realizing low-energy implementations.

Because of the key challenges presented in this book, SoC design technologies will remain a central engineering problem, deserving large human and financial resources for research and development.

References

[1] B. Ackland et al., "A Single Chip, 1.6-Billion, 16-b MAC/s Multiprocessor DSP," *IEEE Journal of Solid-State Circuits,* vol. 35, no. 3, March 2000.

[2] A. Aho, R. Sethi, J. Ullman, *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1988.

[3] P. Aldworth, "System-on-a-Chip Bus Architecture for Embedded Applications," *IEEE International Conference on Computer Design*, pp. 297-298, Nov. 1999.

[4] R. Bahar, G. Albera, S. Manne, "Power and Performance Tradeoffs Using Various Caching Strategies," *ACM/IEEE International Symposium on Low Lower Electronics and Design*, pp. 64-69, Aug. 1998.

[5] H. Bakoglu, Circuits, *Interconnections, and Packaging for VLSI*, Addison-Wesley, 1990

[6] R. Bajwa, M. Hiraki, H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Seki, K. Sasaki, "Instruction Buffering to Reduce Power in Processors for Signal Processing," *IEEE Transactions on VLSI Systems,* vol. 5, no. 4, pp. 417-424, Dec. 1998.

[7] W. Bainbridge, S. Furber, "Delay insensitive system-on-chip interconnect using 1-of-4 data encoding," *IEEE International Symposium on synchronous Circuits and Systems*, pp. 118-126, 2001.

[8] L. Benini and G. De Micheli, ``Networks on Chip: A New SoC Paradigm," *IEEE Computers*, January 2002, pp. 70-78.

[9] L. Benini, A. Bogliolo, G. De Micheli, "A Survey of Design Techniques for System-Level Dynamic Power Management," *IEEE Transactions on Very Large-Scale Integration Systems*, vol. 8, no. 3, pp. 299-316, June 2000.

[10] L. Benini, G. De Micheli, "System-Level Power Optimization: Techniques and Tools," *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 2, pp. 115-192, April 2000.

[11] L. Benini, D. Bruni, A. Macii, E. Macii, "Hardware-Assisted Data Compression for Energy Minimization in Systems with Embedded Processors," *IEEE Design and Test in Europe*, pp. 449-453, March. 2002.

[12] D.Bertozzi, L. Benini and G. De Micheli, "Low-Power Error-Resilient Encoding for On-Chip Data Busses," *IEEE Design and Test in Europe*, pp. 102-109, March 2002.

[13] D. Bertsekas, R. Gallager, *Data Networks*. Prentice Hall, 1991.

[14] M. Borgatti et al.,"A 64-Min Single-Chip Voice Recorder/Player Using Embedded 4-b/cell FLASH Memory," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 3, pp. 516-521, March. 2001.

[15] D.Burger, J. Goodman, A. Kagle, "Limited Bandwidth to Affect Processor Design," IEEE Micro, vol. 17, no. 6, November/December 1997.

[16] D. C. Burger, Hardware Techniques to Improve the Performance of the Processor/Memory Interface, Ph.D. Dissertation, University of Wisconsin-Madison, 1998.

[17] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*, Kluwer, 1998

[18] S. Chirokoff and C. Consel, "Combining Program and Data Specialization", *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, pp.45-59, San Antonio, Texas, USA, January 1999

[19] E.Y.Chung, L. Benini and G. De Micheli,"Automatic Source Code Specialization for Energy Reduction," *ISLPED, IEEE Symposium on Low Power Electronics and Design*, 2000, pp. 80-83.

[20] B. Cordan, "An efficient bus architecture for system-on-chip design," *IEEE Custom Integrated Circuits Conference*, pp. 623-626, 1999.

[21] S. Coumeri, D. Thomas, "Memory Modeling for System Synthesis," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 179-184, Aug. 1998.

[22] J.Crenshaw *math Toolkit for Real-Time Programming*, CMP Books, kansas, 2000.

[23] Cygnus Solutions, *eCOS reference Manual*, 1999

[24] W. Dally and J. Poulton, *Digital Systems Engineering*, Cambridge University Press, 1998.

[25] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[26] D. Ditzel, "Transmeta's Crusoe: Cool Chips for Mobile Computing", Hot Chips Symposium

[27] J. Duato, S. Yalamanchili, L. Ni, *Interconnection Networks: an Engineering Approach*. IEEE Computer Society Press, 1997.

[28] A. Farrahi, G. Tellez, M. Sarrafzadeh, "Memory Segmentation to Exploit Sleep Mode Operation," *ACM/IEEE Design Automation Conference*, pp. 36-41, June 1995.

[29] Gartner, Inc., Final 2000 Worldwide Semiconductor Market Share, 2000.

[30] A. Gonzalez, C. Aliagas, M. Valero, "A Data-Cache with Multiple Caching Strategies Tuned to Different Types of Locality," *ACM International Conference on Supercomputing*, pp. 338--347, July 1995.

[31] G. Goossens, P. Paulin, J. Van Praet, D. Lanneer, W.Guerts, A. Kifli and C.Liem, "Embedded Software in Real-Time Signal Processing Systems: Design Technologies," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 436--54, March 1997.

[32] P. Grun, N. Dutt, A. Nicolau, "Access Pattern Based Local Memory Customization for Low-Power Embedded Systems," *Design Automation and Test in Europe*, pp. 778--784, March 2001.

[33] P. Guerrier, A. Grenier, "A generic architecture for on-chip packet-switched interconnections," *Design Automation and Test in Europe Conference*, pp. 250-256, 2000.

[34] R. Hegde, N. Shanbhag, "Toward achieving energy efficiency in presence of deep submicron noise," *IEEE Transactions on VLSI Sys*tems, pp. 379--391, vol. 8, no. 4, August 2000.

[35] J. Hennessy, D. Patterson, *Computer Architecture - A Quantitative Approach*, II Edition, Morgan Kaufmann Publishers, 1996.

[36] R. Ho, K. Mai, M. Horowitz, "The Future of wires," *Proceedings of the IEEE*, January 2001.

[37] M. Kamble, K. Ghose, "Analytical Energy Dissipation Models for Low-Power Caches," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 143-148, August 1997.

[38] M. Kandemir, M. Vijaykrishnan, M. Irwin, W. Ye, "Influence of compiler optimizations on system power, "*IEEE Transactions on VLSI Systems*, vol. 9, no. 6, pp. 801-804, Dec. 2001.

[39] H. Kim, M. Irwin, N. Vijaykrishnan, M. Kandemir, "Effect of compiler optimizations on memory energy," *IEEE Workshop on Signal Processing Systems*, pp. 663-672, 2000.

[40] J. Kin, M. Gupta, W. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *IEEE/ACM International Symposium on Microarchitecture*, pp. 184-193, Dec. 1997.

[41] A. Kunimatsu et al., "Vector Unit Architecture for Emotion Synthesis," *IEEE Micro*, vol. 20, no. 2, pp. 40-47, March-April 2000.

[42] U. Ko, P. Balsara, A. Nanda, "Energy Optimization of Multilevel Cache Architectures for RISC and CISC Processors," *IEEE Transactions on VLSI Systems*, vol. 6, no. 2, pp. 299-308, June 1998.

[43] G. Jackson et al., "An Analog Record, Playback and Processing System on a Chip for Mobile Communications Devices," *IEEE Custom Integrated Circuits Conference*, pp. 99-102, San Diego, CA, May 1999.

[44] T. Juan, T. Lang, J. Navarro, "Reducing TLB Power Requirements," *ACM/IEEE International Symposium on Low Power Electronics and Design,* pp. 196-201, August 1997.

[45] K. Lahiri, A. Raghunathan, G. Lakshminarayana, S. Dey, "Communication architecture tuners: a methodology for the design of high-performance communication architectures for systems-on-chip," *IEEE/ACM Design Automation Conference*, pp. 513--518, 2000.

[46] G. Lakshminarayana, A. Raghunathan, K. Khouri, K. Jha, and S. Dey, "Common-Case Computation: A High-Level Technique for Power and Performance Optimization", *Design Automation Conference*, pp.56-61, 1999

[47] C. Lefurgy, *Efficient Execution of Compressed Programs*, Doctoral Dissertation, Dept. of CS and Eng., University of Michigan, 2000.

[48] K. Lepak and M. Lipasti, "On the value locality of store instructions", *ISCA*, pp. 182-191, 2000

[49] H. Lekatsas, W. Wolf, "Code Compression for Low Power Embedded Systems," *ACM/IEEE Design Automation Conference*, pp. 294–299, June 2000.

[50] S. Liao, S. Devadas, K. Keutzer, "Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques," *IEEE Transactions on CAD/ICAS*, vol. 17, no. 7, pp. 601–608, July 1998.

[51] D. Lidsky, J. Rabaey, "Low-Power Design of Memory Intensive Functions," *IEEE Symposium on Low Power Electronics*, San Diego, CA, pp. 16-17, September 1994.

[52] M. Lipasti, C. Wilkerson, and J. Shen, "Value Locality and Load Value Prediction", *ASPLOS*, pp.138-147, 1996

[53] M. Lorenz, R. Leupers, P. Marwedel, T. Drager, G. Fettweis, “Low-energy DPS code generation using a genetic algorithm,” *IEEE International Conference on Computer Design*, pp. 431-437, Sept. 2001.

[54] Y. Lu, L. Benini and G. De Micheli, “Requester-Aware Power Reduction,” *ISSS, International System Synthesis Symposium*, 2000, pp. 18-23.

[55] Y. Lu, L. Benini and G. De Micheli, “Power Aware Operating Systems for Interacting Systems,” *IEEE Transactions on VLSI*, April 2002.

[56] A. Macii, L. Benini, M. Poncino, *Memory Design Techniques for Low Energy Embedded Systems*, Kluwer, 2002.

[57] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, M. Horowitz, “Smart Memories: a modular reconfigurable architecture,” *IEEE International Symposium on Computer Architecture*, pp. 161-171, June 2000.

[58] S. Manne, A. Klauser, D. Grunwald, “Pipeline gating: speculation control for energy reduction,” *International Symposium on Computer Architecture*, pp. 122-131, July 1998.

[59] H. Mehta, R. M. Owens, M. J. Irwin, “Some Issues in Gray Code Addressing,” *Great Lakes Symposium on VLSI*, pp. 178–180, March 1996.

[60] H. Mehta, R. Owens, M. Irwin, R. Chen, D. Ghosh, “Techniques for Low Energy Software,” *International Symposium on Low Power Electronics and Design*, pp. 72-75, Aug 1997.

[61] V. Milutinovic, B. Markovic, M. Tomasevic, M. Tremblay, “A new cache architecture concept: The Split Temporal/Spatial Cache,” *IEEE Mediterranean Electrotechnical Conference*, pp. 1108-1111, March 1996.

[62] J. Montanaro et al., “A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor,” *IEEE Journal of Solid-State Circuits*, vol. 31, no. 11, pp. 1703--1714, Nov. 1996.

[63] P. Panda, N. Dutt, *Memory Issues in Embedded Systems-on-Chip Optimization and Exploration*, Kluwer, 1999.

[64] P. Panda, N. Dutt, A. Nicolau, “On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems”, *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 3, pp. 682–704, July 2001.

[65] R. Panda et al., “Data memory organization and optimization in application-specific systems,” *IEEE Design \& Test of Computers*, vol. 18, no. 3, pp. 56-68, May-June 2001.

[66] P. R. Panda, F. Catthor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni,A. Vandecappelle, P. G. Kjeldsberg, “Data and Memory Optimization Techniques for Embedded Systems”, *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 2, pp. 149-206, April 2001.

[67] I. Papadimitriou, M. Paterakis, “Energy-conserving access protocols for transmitting data in unicast and broadcast mode,” *International Symposium on Personal, Indoor and Mobile Radio Communication*, pp. 416–420, 2000.

[68] C. Patel, S. Chai, S. Yalamanchili, D. Shimmel, “Power constrained design of multiprocessor interconnection networks,” *IEEE International Conference on Computer Design*, pp. 408-416, 1997.

[69] A. Peymandoust, T. Simunic and G. De Micheli, “Complex Library Mapping for Embedded Software using Symbolic Algebra,” *DAC, Design Automation Conference*, 2002.

[70] D. Patterson, et al., “A Case for intelligent RAM,” *IEEE Micro*, vol. 17, no. 2, pp. 34-44, March-April 1997.

[71] Redhat, *Linux-ARM math Library Reference Manual*

[72] S.E. Richardson, “Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation”, *Tech. report, Sun Microsystems Laboratories*, 1992

[73] D. Shin, J. Kim, "A profile-based energy-efficient intra-task voltage scheduling algorithm for hard real-time applications," *IEEE International Symposium on Low-Power Electronics and Design*, pp. 271-274, Aug.2001.

[74] W. Shiue, C. Chakrabarti, "Memory Exploration for Low Power, Embedded Systems," *DAC-36: ACM/IEEE Design Automation Conference*, pp. 140-145, June 1999.

[75] A. Shubat, "Moving the market to embedded memory," *IEEE Design & Test of Computers*, vol. 18, no. 3, pp. 16-27, May-June 2001.

[76] C. Su, C. Tsui, A. Despain, "Saving Power in the Control Path of Embedded Processors," *IEEE Design and Test of Computers*, vol. 11, no. 4, pp. 24--30, Winter 1994.

[77] C. L. Su, A. Despain, "Cache Design Trade-Offs for Power and Performance Optimization: A Case Study," *ACM/IEEE International Symposium on Low Power Design*, pp. 63-68, April 1995.

[78] M. Suzuoki et al., "A Microprocessor with a 128-bit CPU, Ten Floating-Point MACs, Four Floating-Point Dividers, and an MPEG-2 Decoder," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 11, pp. 1608--1618, Nov. 1999.

[79] D.Sylvester and K.Keutzer, "A Global Wiring Paradigm for Deep Submicron Design," *IEEE Transactions on CAD/ICAS*, vol.19, No. 2, pp. 242-252, February 2000.

[80] M. Takahashi et al., "A 60-MHz 240-mW MPEG-4 Videophone LSI with 16-Mb embedded DRAM," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1713-1721, Nov. 2000.

[81] V. Tiwari, S. Malik, A. Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," *IEEE Transactions on VLSI Systems*, vol. 2, no.4, pp.437–445, Dec. 1994.

[82] V. Tiwari, S. Malik, A. Wolfe, M. Lee, "Instruction Level Power Analysis and Optimization of Software," *Journal of VLSI Signal Processing*, vol. 13, no.1-2, pp.223--233, 1996.

[83] T. Theis, "The future of Interconnection Technology," *IBM Journal of Research and Development*, vol. 44, No. 3, May 2000, pp. 379-390.

[84] H. V. Tran et al., "A 2.5-V, 256-level nonvolatile analog storage device using EEPROM technology," *IEEE International Solid-State Circuits Conference*, pp. 270-271, Feb. 1996.

[85] J. Walrand, P. Varaiya, *High-Performance Communication Networks*. Morgan Kaufman, 2000.

[86] S. Winegarden, "A bus architecture centric configurable processor system," *IEEE Custom Integrated Circuits Conference*, pp. 627--630, 1999.

[87] A. Wolfe, "Issues for Low-Power CAD Tools: A System-Level Design Study," *Design Automation for Embedded System*, vol. 1, no. 4, pp. 315-332, 1996.

[88] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.

[89] Y. Yoshida, B. Song, H. Okuhata, T. Onoye, I. Shirakawa, "An Object Code Compression Approach to Embedded Processors," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 265-268, August 1997.

[90] R. Yoshimura, T. Koat, S. Hatanaka, T. Matsuoka, K. Taniguchi, "DS-CDMA wired bus with simple interconnection topology for parallel processing system LSIs," *IEEE Solid-State Circuits Conference*, pp. 371-371, Jan. 2000.

[91] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, J. Rabaey, "A 1-V Heterogeneous Reconfigurable DSP IC for Wireless Baseband Digital Signal Processing," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1697--1704, Nov. 2000.

[92] H. Zhang, V. George, J. Rabaey, "Low-swing on-chip signaling techniques: effectiveness and robustness," *IEEE Transactions on VLSI Systems*, vol. 8, no. 3, pp. 264-272, June 2000.

[93] H. Zhang, M. Wan, V. George, J. Rabaey, "Interconnect architecture exploration for low-energy configurable single-chip DSPs," *IEEE Computer Society Workshop on VLSI*, pp. 2-8, 1999.

[94] V. Zyuban, P. Kogge, "The Energy Complexity of Register Files," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 305-310, Aug.t 1998.

[95] International Technology Roadmap for Semiconductors *http://public.itrs.net/*