ISSN 1688-2784

Universidad de la República
Facultad de Ingeniería

Energy-efficient memories for wireless sensor networks

Tesis presentada a la Facultad de Ingeniería de la Universidad de la República por

Leonardo Steinfeld Volpe

en cumplimiento parcial de los requerimientos para la obtención del título de Doctor en Ingeniería Eléctrica.

Director de Tesis
Dr. Luigi Carro ..... Univ. Federal do Rio Grande do Sul, Brasil

Tribunal
Dr. Eduardo Grampín ..... Universidad de la República
Dr. Augusto Fröhlich* .... Univ. Federal de Santa Catarina, Brasil
Dr. Diógenes da Silva Jr* ... Univ. Federal de Minas Gerais, Brasil

* Revisor externo

Director Académico
Dr. Fernado Silveira ..... Universidad de la República

Montevideo
28 de noviembre de 2013

Energy-efficient memories for wireless sensor networks, Leonardo Steinfeld Volpe

ISSN 1688-2784

Esta tesis fue preparada en LaTeX usando la clase iietesis (v1.0).
Contiene un total de 130 páginas.
Compilada el Monday 16th June, 2014.
http://iie.fing.edu.uy/

Acknowledgments

This thesis would not have been possible without the support of many people around me. I apologize beforehand to those who are not explicitly mentioned here, although I hope I had expressed my gratitude.

First of all, I would like to thank my advisor, Luigi, for his support along the time during the thesis and sharing his valuable time. I also would like to specially thank Fernado whose help was fundamental during these years. I am sure that this thesis would not have been possible without his support.

I also wish to thank Patricia, who was the third fundamental cornerstone, because she encouraged me during these long hard years, and for her effort specially during my stays in Porto Alegre and during the last 6 months taking care of little Salva more than usual.

I also wish to thanks my colleagues and administrative staff at IIE.

The work in this thesis is in part supported by ANII, CSIC, and CAPES .

Last but not least, I would like o thank the reviewers and members of the jury who kindly accepted to review my work.

This page is intentionally left blank.

A Salvador y Patricia, mis verdaderos amores.

This page is intentionally left blank.

Resumen

Las redes de sensores inalámbricas (RSI o WSN, por sus siglas en inglés) agregan computación y sensado al mundo físico, posibilitando un rango de aplicaciones sin precedentes en muchos campos de la vida cotidiana, como por ejemplo monitoreo ambiental, manejo de ganado, cuidado de personas adultas mayores y medicina, solo por mencionar algunas. Una RSI consta de nodos sensores, los cuales representan un nuevo tipo de computadora embebida en red, caracterizada por tener grandes restricciones de recursos. El diseño de un nodo sensor presenta muchos desafíos, ya que es necesario que sean, pequeños, confiables, de bajo costo y con muy bajo consumo de energía, ya que se alimentan de pilas o recolectan energía del medio. En un nodo sensor, la potencia instantánea del transceptor (radio) es usualmente algunos órdenes de magnitud mayor que la potencia de procesamiento. Sin embargo, la energía de comunicación es solamente dos veces mayor que la energía de procesamiento. Por otro lado, el escalado de la tecnología CMOS permite mayor *performance* a menores precios, posibilitando aplicaciones distribuidas más refinadas con más procesamiento local. El aumento de la complejidad de las aplicaciones requiere memorias de mayor tamaño, que a su vez aumenta el consumo de potencia. Este escenario empeora ya que las corrientes de fuga son cada vez más importantes en transistores de menor tamaño.

En el presente trabajo de tesis se caracteriza el consumo de energía de un nodo sensor, y se investigan diferentes arquitecturas de memoria para ser integrado en las RSI futuras, mostrando como las memorias SRAM con un estado de *sleep* pueden ser convenientes en sistemas que operan con bajos ciclos de trabajo. Si además la memoria se divide en bancos que pueden ser controlados de manera independiente, se pueden poner los bancos inactivos en estado *sleep*, incluso cuando el sistema está activo. Aunque esta es una técnica conocida, los límites de ahorro de energía no habían sido exhaustivamente determinados, ni tampoco la influencia de la política de gestión de energía usado. Se propone un nuevo modelo detallado del ahorro de energía para bancos uniformes con dos políticas de gestión: *best-oracle* y *greedy*. Nuestro modelo proporciona información valiosa de los factores fundamentales (provenientes del sistema y la carga de trabajo) que son escenciales para alcanzar el máximo ahorro alcanzable. Gracias a nuestro modelado, en tiempo de diseño se puede estimar el número óptimo de bancos para lograr grandes ahorros de energía. El problema de asignación del código a los bancos fue resuelto usando programación lineal entera. En el contexto de esta tesis, se realizaron experimentos usando dos aplicaciones reales de redes de sensores inalámbricas (basadas en TinyOS y ContikiOS). Los resultados mostraron una reducción de energía cercano a 80%

para un *overhead* de partición de 1% con una memoria de diez bancos para una aplicación con gran carga. El ahorro depende del patrón de acceso a memoria y los parámetros de la memoria (tales como cantidad de bancos, *overhead* de partición, reducción de energía del estado *sleep* y el costo energético de *wake-up*. El ahorro de energía decrece para ciclos de trabajo bajos. Sin embargo, igualmente se alcanzan ahorros de energía significativos, por ejemplo, aproximadamente 50% para ciclos de trabajo de 3% usando la memoria anterior. Finalmente, nuestros resultados sugieren que debe ser cuidadosamente evaluado el uso de políticas de gestión de energia avanzados, ya que la política *best-oracle* es solo marginalmente mejor que la política *greedy*.

Abstract

Wireless sensor networks (WSNs) embed computation and sensing in the physical world, enabling an unprecedented spectrum of applications in several fields of daily life, such as environmental monitoring, cattle management, elderly care, and medicine to name a few. A WSN comprises sensor nodes, which represents a new class of networked embedded computer characterized by severe resource constraints. The design of a sensor node presents many challenges, as they are expected to be small, reliable, low cost, and low power, since they are powered from batteries or harvest energy from the surrounding environment. In a sensor node, the instantaneous power of the transceiver is usually several orders of magnitude higher than processing power. Nevertheless, if average power is considered in actual applications, the communication energy is only about two times higher than the processing energy. The scaling of CMOS technology provides higher performance at lower prices, enabling more refined distributed applications with augmented local processing. The increased complexity of applications demands for enlarged memory size, which in turn increases the power drain. This scenario becomes even worse as leakage power is becoming more and more important in small feature transistor sizes.

In this work the energy consumption of a sensor node is characterized, and different memory architectures were investigated to be integrated in future wireless sensor networks, showing that SRAM memories with sleep state may benefit from low duty-cycle operating system. SRAM memory with power-manageable banks puts idle banks in sleep state to further reduce the leakage power, even when the system is active. Although it is a well known technique, the energy savings limits were not exhaustively stated, nor the influence of the power management strategy adopted. We proposed a novel and detailed model of the energy saving for uniform banks with two power management schemes: a best-oracle policy and a simple greedy policy. Our model gives valuable insight into key factors (coming from the system and the workload) that are critical for reaching the maximum achievable energy saving. Thanks to our modeling, at design time a near optimum number of banks can be estimated to reach more aggressive energy savings. The memory content allocation problem was solved by an integer linear program formulation. In the framework of this thesis, experiments were carried out for two real wireless sensor network application (based on TinyOS and ContikiOS). Results showed energy reduction close to 80% for a partition overhead of 1% with a memory of ten banks for an application under high workload. Energy saving depends on the

access patterns to memory and memory parameters (such as number of banks, partitioning overhead, energy reduction of the sleep state and the wake-up energy cost). The energy saving drops for low duty-cycles. However, a very significant reduction of energy can be achieved, for example, roughly 50% for a 3% duty-cycle operation using the above memory.

Finally, our findings suggest that adopting an advanced power management must be carefully evaluated, since the best-oracle is only marginally better than a greedy policy.

Contents

This page is intentionally left blank.

Chapter 1

Introduction

This chapter introduces the context of the work presented in this thesis. The first section presents a description of the wireless sensor networks and the motivation of this work, to finally highlight some open challenges that need to be addressed to enable an ubiquitous adoption of these networks. The next section discusses the specific goals of our work and continues enumerating the main contributions of the present thesis. Finally, we conclude the chapter providing a road-map for the rest of this document.

1.1 Motivation

Wireless sensor networks (WSNs) embed computation and sensing in the physical world, enabling an unprecedented spectrum of applications in several fields of daily life. A WSN comprises sensor nodes, which sense the environment, process the acquired data and communicate the information to other sensor nodes. In traditional sense-and-send applications the sensor nodes route the raw data to a sink node to make it available to users, while in distributed computing applications the data is processed in the network by neighbor nodes, exchanging messages to directly output meaningful information or take actions. The sensor node represents a new class of networked embedded computer that is characterized by severe resource constraints: computation power, available memory for program and data storage, and especially energy.

Since the nodes are usually placed in inaccessible locations and in large number, they are powered from batteries or harvest energy from the environment [38]. In both cases low-power operation is mandatory, in the former case to avoid frequent batteries or nodes replacement, and in latter case to suit the scarce available energy or to minimize the harvesting system cost and size (e.g. solar panel). A canonical sensor node consists of the following blocks: a processing component (usually a microcontroller) with wireless communication capabilities (RF transceiver), sensors/actuators and a power supply subsystem [95]. The communication and computation subsystems are the main sources of energy dissipation in a WSN node. The RF transceiver is usually the most power-consuming component of a sensor

node, and since the WSN emergence the communication energy cost has dominated the overall budget. Consequently, a significant research effort has been made since then for reducing this communication energy cost [19] [5]. The Medium Access Control (MAC) layer design is crucial, since it directly controls the transceiver determining the power profile drain. The use of advanced MAC protocols has helped in improving the energy efficiency for communication [20], becoming comparable to the processing energy [51].

Nevertheless, in more recent intensive processing applications the energy spent in computing could be much larger than the energy spent in communication. Nowadays, it is generally accepted that efforts toward energy reduction should target communication and processing [87]. At the same time, reducing the computational energy enables further optimizations concerning communication, since usually actual techniques are restricted by the low computational capabilities and low memory footprints.

Despite computation energy concerns, expectations for higher performance and lower prices continue to increase with each new CMOS technology generation. In this regard, as technology scales, novel applications can be devised, where previously their cost in terms of price were unacceptable, since the capacity of sensor nodes integrated processors has increased. More refined applications performing distributing computing, also known as collaborative information processing, will become key elements of larger pervasive infrastructures such as Internet of Things (IoT) [109] or Cyber-Physical Systems (CPS) [80]. However, this increasing complexity of applications using wireless sensor networks soon becomes a barrier to the adoption of these networks. The currently available wireless sensor network programming models do not scale well from simple data collection models to collaborative information processing ones. On a different scenario, complex distributed applications have been developed for powerful platforms (such laptops, smartphones, etc.), but they are not appropriate, since they require intensive computation that current WSN platforms can not afford. If they were equipped with a more powerful microcontroller, they would result in a too power-hungry platform, sacrificing energy [29]. New programming models are essential to develop complex distributed applications, and at the same time obtain a decent level of energy-efficiency.

Novel virtual-machines (VMs) and middlewares that are designed with WSN in mind are efforts in the direction of complex networks. There are several benefits in using virtual machines in WSN [72]. First, VMs allow applications to be developed uniformly across WSN platforms. Platform-independent applications can be written using VM abstractions whose implementations are scaled to meet different resource constraints. VMs provide a clean separation of system software and application software, which reduces the cost of reprogramming after deployment. Finally, VMs mask the variations among the WSN platforms through a common execution framework [68]. Middlewares even go further, adding a higher-level of abstraction to allow a smaller programming effort of distributed systems, such as the WSNs, and to achieve interoperability [80]. Middleware is usually defined as the software that lies between the operating system and applications running

on each node of the system providing support for the development, maintenance, deployment and execution of WSN application [24]. A middleware provides standard interfaces, abstractions and a set of services to hide the working internals that simplify the application development, while hiding the heterogeneity of the system, which then enables interoperability. Nevertheless, higher abstractions add new levels of indirection increasing the execution overhead, which in turns increases the energy consumption. Summarizing, one of the actual major issues of WSN research is providing high-level application programming abstractions and reducing the processing energy consumption to leverage creative and more complex applications of the future.

Regarding processing, a programmable processor, in contrast to hardwired logic, allows a single hardware resource to implement different applications by running different software stored in memory. The cost of this unquestionable flexibility is that a processor spends significant more energy on instruction and data supply than performing arithmetic [28]. Microcontrollers embed, in addition to a processor, programmable I/O peripherals (such as timers/counters, serial ports, and so on) and memory. They usually have two kinds of memory: a read-write memory (e.g. SRAM) and non-volatile memory (e.g. Flash). The former memory is needed for storing temporal data (variables), and the later for storing the program and constants, retaining the content even though it is not powered. The code can be executed directly from non-volatile memory, known as execute in place (XIP) or first copied from Flash to RAM (shadow memory) [61]. As executing code from SRAM consumes considerable less energy than executing from Flash (this matter is addressed later in Chapter 3), to provide enough SRAM space for placing the code should be considered in WSN applications in order to reduce the node energy consumption. Since SRAM occupies more area per bit than Flash memory, there is a trade-off between area cost and energy reduction. The best option for the memory architecture should be carefully considered, choosing from full shadow the code from Flash into SRAM to schemes borrowing ideas from scratch-pad or even cache memories themselves [9].

Moreover, the aforementioned increased complexity of applications (reflected in software complexity) demands for increasing memory size. In some applications the code size is doubling every ten months [108]. WSN applications follows this general trend, but at a lower rate hampered by hardware limitations, as can be observed in the evolution of applications memory footprints (as we shall see in the next chapter). Larger memory size requires more power, as it has been found that the memory system is responsible for a large portion of the total energy budget in SoCs [25]. A downside of the scaling of CMOS technology is the increase of leakage current to the point that it may represent up to 50% of the total power consumption of a circuit [59, 63].

This problem is getting worst as more transistors are put in large circuits as memory arrays. Therefore, the reduction of leakage power in the memory system is definitely a primary target in future embedded systems, particularly in deeply pervasive systems such as WSN.

Partitioning a SRAM memory into multiple banks that can be independently

accessed reduces the dynamic power consumption, and since only one bank is active per access, the remaining idle banks can be put into a low-leakage sleep state to also reduce the static power [41]. However, the power and area overhead due to the extra wiring and duplication of address and control logic prevents an arbitrary fine partitioning into a large number of small banks. Therefore, the final number of banks should be carefully chosen at design time, taking into account this partitioning overhead. The memory organization may be limited to equally-sized banks, or it can allow any bank size [73]. Moreover, the strategy for the bank states management may range from a greedy policy (as soon as a bank memory is not being accessed it is put into low leakage state) to the use of more sophisticated prediction algorithms [18].

1.2 Goals and contributions

Some of the most important challenges that need to be addressed for future wireless sensor networks are:

- high level programming abstractions for the ease of development and to enable more complex applications;

- energy-efficient platforms compatible with application lifetime requirements

Considering the challenges above and all motivations discussed before, the main goal of this thesis is memory energy reduction aiming to extending WSN node lifetime, and at the same time enabling higher level of programming abstraction for application development and consequently contribute to a widespread adoption of WSN technology.

This thesis makes three main contributions: Firstly, it contributes with a characterization and analysis of the run-time execution of typical current application developed based on the most popular WSN operating systems (TinyOS and ContikiOS). We show that memory energy consumption may hamper the spreading of WSN application that involves complex processing.

A second contribution of this work is, after a review of available memories for using in WSN processors to hold program code, the benefits of using a SRAM memory with a low-leakage sleep state are stated.

A third and the most important contribution of this work is the proposal of using banked memories with power management to cope with the increasing leakage current, result from the scaling of CMOS technology. This could help to manage the increasing complexity of applications and, at the same time, to extend the life-time of the WSN nodes. Analytical expressions for the energy savings are derived from a memory model to determine the impact of design decision as power management strategy, memory architecture and technology parameters, and the application run-time behavior.

1.3 Thesis Organization

This thesis is divided into seven chapters, whose contents are summarized in the following paragraphs:

- Chapter 1 has described the wireless sensor networks and the motivation for this thesis, enumerating the main challenges and briefly describing the contribution of this work.
- Chapter 2 reviews actual platforms (software and hardware), and low-power techniques and methodologies, particularly those which are applied for reducing the energy consumption in sensor nodes. Next, the energy consumption is characterized, showing its dependence on the hardware platform power parameters and software architecture.
- Chapter 3 describes current memories available for embedded system, reviews its main characteristics highlighting which are suitable for WSN embedded processors with special emphasis on energy efficiency. The energy consumption of Flash and SRAM are compared, focusing on the pros and cons of each memory technology. The SRAM memory with sleep state is introduced, and potential energy savings delimited. Finally, a memory with power-manageable banks is modeled and the main characteristics described.
- Chapter 4 investigates in depth the memory architecture introduced last in the previous chapter. Expressions for the energy savings are derived, gaining valuable insight into key factors that are fundamental for achieving huge savings. Two power management policies are modeled and discussed.
- Chapter 5 presents experiments carried out to asses energy savings. The energy savings limits predicted by our model are compared to savings results obtained by simulation for different configurations.
- Chapter 6 summarizes the work developed in the framework of this thesis, highlights the main contributions and discusses future works.

This page is intentionally left blank.

Chapter 2

Sensor node platform

A key element of wireless sensor networks is the sensor node. A basic sensor node is composed of the following building blocks: a processing element with a radio (RF transceiver), sensors/actuators and a power supply subsystem. Node power consumption results from the sum of the power contributions of its electronic components, which in turn depend on the component state and the actual operation performed. The power profile drain, i.e. the instantaneous power as a function of the time, determines the effective node energy consumption. The hardware together with software and external factors, as the environment and the interaction with the network, dictates the power profile drain of each node. The hardware defines the power level consumption. The network design and the communication protocols influence the behavior of the nodes. The message exchange within or between the nodes determine the state and actions of the node, particularly the radio operation mode, i.e. receive, transmit or sleep mode, and the microcontroller opeartion mode. Finally, the software implements the final application, usually on top of an operating system or a higher lever abstraction. The operating system provides services to ease application development, including a communication stack that implements network protocols. Also, provides abstractions across platforms to hide hardware resources differences and thus improve portability.

This chapter introduces the main design challenges of sensor node platforms. The first section briefly presents the evolution of hardware sensor nodes and future trends. It also surveys some efforts on designing ultra low-power processors, some targeting wireless sensor networks. The second section discuss the software sensor node, describing in detail the most widespread software platforms, TinyOS and ContikiOS, and emerging Java virtual machines. The third section introduces the fundamentals of low-power techniques and the main concepts for dynamic power management. The next section presents an energy characterization of a popular sensor node, showing that increasing the complexity of applications must be accompanied with a reduction of processing power. Finally, the conclusion summarizes the chapter, including a brief discussion of the main aspects.

2.1 Sensor node hardware

2.1.1 Evolution and future trends of hardware platforms

The wireless sensor networks was envisaged in the early nineties as a pervasive technology where sensor nodes are as small as fine particles of dust. This idea was materialized in the Smart Dust project by Kristofer S. J. Pister et al. at University of California, Berkeley, aiming at integrating a complete sensing/communication platform inside a cubic millimeter [62]. Later, in collaboration with other researches, including David Culler and his group from Berkeley too, they developed also hardware and software platforms following the open-source model. These hardware platforms, in the order of a few cubic centimeters, were built using commercial off-the-shelf (COTS) chips, named macro motes or simply motes. These sensor nodes together with its sequels (available as commercial products too), commonly named as Berkeley motes, are still one of the most popular in the academia, as the *mica* [27] and *telos* [90]. The software platform developed is composed by the operating system TinyOS [55] and the nesC language [44] (nesC, for network embedded systems C).

Most hardware platforms that are built using COTS components use microcontrollers of one of the two families: AVR from Atmel Corporation (8-bit processor, e.g. Atmega128) and MSP430 from Texas Instruments (16-bit processor, e.g. MSP430F1611). The selected radio varies according to the frequency range usually in one of the ISM bands[1], and if they are compliant or not with the IEEE 802.15.4 LR-WPANs standard (low-rate wireless personal area networks). They have evolved from byte-oriented to packet-oriented radio, incorporating in silicon new functionalities defined in the standard, such as data encryption or automatic preamble generator.

Some years later after the release of the first sensor nodes, more powerful platforms were developed, featuring ARM cores running at hundreds of MHz, for example Sun SPOT by SUN (now Oracle). However, their high power requirements drain the batteries in days or even hours, therefore limiting its use to a narrow set of applications. More recently, some sensor nodes were equipped with newer low power 32-bit microcontrollers, offering increased computation power with a power consumption of about one order of magnitude higher than the Berkeley motes [66].

Table 2.1 compares some of the most representatives motes from the beginning of the WSN to the present: *telos*, *micaz*, *AVR-Raven*, *Wismote*, *LOTUS* and *econotag*. The table shows the microcontroller and radio chips used in each sensor node (for an extensive list of surveyed platforms, please see [50]). For the sake of clarity are only considered the main characteristics of the chips, and the current consumption of the most important operation modes.

As can be seen from the table, there is a clear trend toward increasing the processing power, adopting 32-bits processors running at higher frequencies. The

[1]There are other alternatives for the transmission medium different from the electromagnetic waves (radio frequency), such as optical or water (in underwater sensor networks).

Table 2.1: Sensor nodes comparison.

	unit	telos	micaz	AVRaven	wismote	LOTUS	econotag
Microcontroller		MSP430F1611	Atmega128L	AtMega1284p	MSP430F5438	LPC1758 (NXP)	MC13224V
Core		MSP430	AVR	AVR	MSP430X	ARM Cortex-M3	TDMI ARM7
		16-bits	8-bits	8-bits	16-bits	32-bit	32-bit
Speed (f_{max})	MHz	8	8	20	25	100	24
Prog. mem.	KB	48	128	128	256	512	128
Data mem.	KB	10	4	16	16	64	96
Active current	mA	4	5		8.9	50	3.3
Sleep current	μA	2	15		2.1	10	5.1
Transceiver		CC2420		AT86RF230	CC2520	AT86RF231	Integrated PiP
Rx current	mA	19.7		16	18.5	13.2	22
Tx current	mA	17.4		17	25.8	14.4	29
Link budget	dBm	94		104	103	104	100
Sleep current	μA	426		1500	1600	400	-
Deep sleep current	μA	20		0.02	175	0.02	5.1

transceiver consumption power consumption remains at the same level but providing an increase of output power (about 3 dBm in average).

In order to have a complete and accurate picture of the overall power consumption of the different sensor motes, it would be necessary to precise the application characteristics and workload, e.g., how much processing and communication is needed and how long the radio and microcontroller remains in a sleep state. Unfortunately there is no neutral benchmarks available yet, to even compare ultra low-power processors[2]. From the data in Table 2.1, it is apparent that the amount of memory is increasing, and that the program memory increases at a higher rate than data memory. This fact could be interpreted as increasing demand for memory by applications.

Because of space limitations, we have not included a significant number of sensor nodes and modules built using COTS components (commercial or academic) but the tendency is confirmed: increasing processing power with lower rate of power consumption rise, as presented by Ko et al. [66].

2.1.2 Ultra low-power processors

In the last years, there has been a large amount of research dealing with platform power optimization for WSN. Most proposals follow an application-driven approach to design and implement a custom system architecture (see [88] for a brief review and comparison of different approaches). These architectures are more power efficient than general purpose commodity microcontrollers.

Raval et al. [97] conducted a workload analysis for a WSN application scenario running TinyOS-based application software on a platform equipped with a ATmega128L microcontroller (the same microcontroller present in a mica sensor node). The application included a simple filter processing, where the output data was sent to a sink node. They therefore proposed a processor platform tuned for running a suite of applications developed using TinyOS. The tuned platform included an application-specific programmable processor core and a hardware accelerator for offloading a small, but frequently used, set of instructions. The platform maintained near binary compatibility with the conventional microcontroller, consuming 48% less energy than the baseline processor when executing the same WSN application suite.

Hempestead et. al. [51] embraced also the accelerator-based computing paradigm, including acceleration for the network layer (routing) and the application layer (data filtering), but focusing on reducing the leakage current during long idle times. The proposed architecture disable the accelerators via V_{DD}-gating. According to their results the proposed architecture achieved between one to two orders of magnitude reduction in power dissipation over commonly used microcontroller, depending on the workload. More specifically, 100 times less power when idle, and from 10-600 times less energy when active.

[2]At the time of writing of the present work, the Embedded Microprocessor Benchmark Consortium (EEMBC) is working on ULPBench ™Benchmark Software, with the participation of dozens of top companies that design and manufacture microcontrollers.

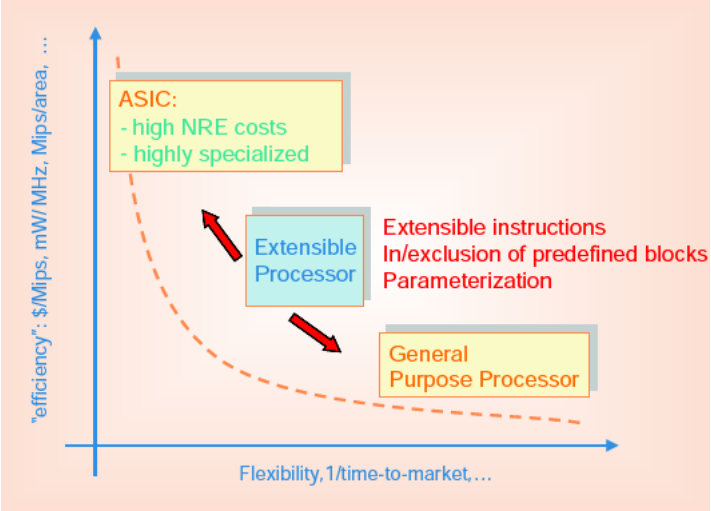

Figure 2.1: Custom hard-wired logic, extensible processor and general purpose processor [52].

Pasha et al. [88] architecture followed a hardware specialization with power gating approach, but incorporated the concept of distributed hardware microtasks in control-oriented applications. They proposed a complete system-level flow, in which a node was built out of microtasks that are activated on an event-driven basis. An application-specific architecture is synthesized for each microtask, optimized by hand. They combined this specialized architecture with power gating to reduce both dynamic and static power. As in the Hempestead work, the focus was on the bulk workload such as device drivers, medium access control (MAC) protocols and routing, since the nodes normally don't have a compute-intensive operating system kernel (as shall be detailed in Section 2.2).

Kwong et al. [69] adopted the aboves techniques, and additionally applied supply voltage scaling to below the device threshold in logic and SRAM circuits. They proposed a technique to mitigate the effects of process variation, increasingly important at low voltages in deeply scaled technologies. A 16-bit microcontroller core was designed to operate at sub-threshold voltage down to 300 mV. Energy-constrained systems that can afford some performance degradation may benefit from the subthreshold processor design.

The confrontation between general purpose microcontrollers and application-specific programmable processors can not be addressed without the economic point of view. Although we do not deal with this issue that arises when considering the scale dimension, it should be taken into account to get a low power sensor node, and, at the same time, with lower cost for a wide-spread adoption of WSNs. It is also worth mentioning that as the more flexible a processor, the more likely to be massively chosen. Fig. 2.1 reproduce the diagram presented by Henkel et al. [52] showing the trade-off of custom hard-wired logic, extensible processor (application specific processor) and general purpose processor. The aforementioned works focus on reducing the energy spent in processing, in general. Thus, it can be enlightening seeing where the energy goes in a processor.

Table 2.2: Processor configuration and energy consumption per operation (based on [6]).

RISC Processor		
Technology	TSMCCL013G (V_{DD} 1.2V)	
Clock Frequency	200 MHz	
Average Power	72 mW	
Instruction supply	70 pJ	42 %
Data supply	47 pJ	28 %
Arithmetic (add & multiply)	10 pJ	6 %
Clock and control logic	40 pJ	24 %

Dally et al. [28] proposed an efficient programmable architecture for compute-intensive embedded applications, and for comparison purposes a detailed energy breakdown of a conventional RISC (SPARC V8) processor was provided. The results are reproduced partially in Table 2.2 (see [28] and [6] for details). The table shows the relative energy for instruction supply, data supply, arithmetic, and clock and control logic, representing 42% , 28%, 6% and 24% of the total energy consumption respectively. The presented data is quite revealing since it shows that the processor spends most of its energy on instruction and data supply, 70%. This result indicates that some research effort should focus on reducing the energy consumption on supplying data to processors.

Moreover, Verma [110] illustrated that the relative SRAM memory consumption in processors increases as processors consumption decreases by applying advanced design techniques. The processor with the lowest power consumption, among the surveyed ones in [110], is a custom MSP430 processor with 16 KB SRAM cache, operating at 0.3 V [70], where the embedded SRAM consumes 69% of the total processor power. Hence, the energy consumption in ultra-low-power processors is greatly dominated by memory accesses.

Chapter 3 analyze in detail the memory energy consumption evolution with the technology scaling, showing that the reduction of memory energy consumption is a major goal for future and more complex WSNs.

In summary, reducing the power taken by naïve memory organizations enables more computationally demanding algorithms to be implemented with the extra power resources, expanding the range of WSN applications. Moreover, the communication protocols are actually restricted by the low computational capabilities and low memory footprints of current low-power processors, hence reducing the computational energy will enable to adopt more complex communications protocols leading to further optimizations to reduce the communication energy [51].

2.2 Sensor node software

A sensor node is basically a reactive system that responds to external stimulus: a successful reception of a packet, a communication timeout, a time trigger to initiate some measurement, data ready interruption that may trigger further processing, and so on. Sensor network applications are intrinsically event-driven, so that the software designer or programmer typically adopts event-based software architectures or operating systems, like TinyOS [71] and ContikiOS [36]. Both are perhaps the most popular embedded operating systems for wireless sensor networks in the academia and the industry. There are other interesting alternatives developed by the research community, but they do not have such a large amount of people involved (user and developers), nor any other clear advantages. To mention a few of them: Mantis OS [15] and SOS [49], which are no longer under active development; LiteOS [21] provides Unix-like abstractions but apparently it is not fully operational; and RETOS [23], which includes interesting features like variable system timer, but the source code is not provided. For a review of operating systems and network protocols please refer to [38]. Additionally, there are other operating systems from outside the academia, some commercially offered, e.g., FreeRTOS (open source but do not include network support) and μC/OS (open source with many licensing options). Some microcontroller, radio and SoC manufacturing companies provide operating systems (e.g., TI-RTOS from Texas Instrument and MQX^{TM}from Freescale) and stacks (e.g. simpliciTI and Z-Stack from Texas Instruments, BitCloud from Atmel).

The remainder of this section briefly present TinyOS and ContikiOS. Also, some Java virtual machines specially designed for sensor nodes are reviewed.

2.2.1 TinyOS

TinyOS[3] based applications are developed using nesC [44], the companion programming language which is an extension of the C programming language. TinyOS embraces the component-based paradigm where different functions are encapsulated within components which are connected each other using interfaces [71]. TinyOS provides components and interfaces for the very common needs, from low-level abstractions such as timing management and I/O interface, to higher levels such as communication protocol stacks (including medium access control and routing). The architecture relies on non-blocking functions that run to completion, and consequently a single stack is used avoiding high-cost context-switches.

Some of these software components abstract hardware components. TinyOS extends to all components the split-phase needed to avoid busy-wait in hardware transactions. For this purpose *commands* and *events* are used, which are essentially functions. Normally an operation is initiated through a *command* call. When a transaction lasts too long, the operation is completed later by a *event* callback. Operations initiated from a hardware interrupt (interrupt service routine, ISR), labeled as asynchronous, must be kept short to not delay other ISRs. Larger

[3] www.tinyos.net

computations must be deferred through a function post, *task*, to be scheduled later outside of the ISR context. The tasks are serviced by a simple tasker in a first-input first-output basis. As the tasks run to completion, long tasks should be split in many shorter ones to share the processor time.

TinyOS provides timing services through *timers* components based on one or several system ticks implemented using hardware timers. TinyOS also includes a library to support threads, named TOSThreads [65].

2.2.2 ContikiOS

ContikiOS[4] based applications are developed using the multithreading programming model based on protothreads [37]. The programming language is standard ANSI C. Each process has its corresponding protothread. Processes uses a special construct that have a blocking semantic for waiting for events by verifying a condition. If the condition is false the process yields the processor. The protothreads are implemented as an ordinary function that returns when it blocks (the condition is false). But before returning, the function saves the location where it got blocked, so that it can resume the execution later at that point. The protothread primitives are ingeniously defined by macros, which expands in a switch-case construction. The kernel schedules blocked processes in response to an event. The events are either internal, e.g. a message posted from other processes, or external, e.g. triggered by a hardware interrupt. For that, the kernel has a queue of pending events from which the next event to be processed is picked. Then the kernel traverse the process list calling the respective protothread to process the event.

ContikiOS has a process to offer timer services. A system tick periodically wakes up the process.

The protothread adoption by ContikiOS has some implications: the application has a single stack, since the protothreads returns when blocks; local variables values are not preserved between invocations; and since the kernel is not preemptive the processes should explicitly yield control in long computations. As a consequence ContikiOS and TinyOS share many characteristics. Despite taht they follow different paradigms, i.e. component-based model versus multithreading, at execution time they behave similarly, since both are event-driven in nature.

2.2.3 Higher level abstractions: Java virtual machines

There are many advantages in using higher level languages in WSNs [80]. Moreover, the adoption of Java as the wireless sensor network programing language is an interesting option, considering the worthwhile features of the language, such as productivity gain thanks to the associated object-oriented software engineering. Furthermore, Java virtual machines (JVM) provide memory protection through type safety, dynamic memory management with garbage collection, and a clean interface for developing platform-independent applications. Middlewares, if present, are easily implemented on top of a JVM.

[4] www.contiki-os.org

Finally, the Java language is very popular among programmers and it definitely requires a much shorter learning curve than the nesC of TinyOS and even the widespread language C. As a consequence, Java is a good candidate for becoming the wireless sensor network language within the Internet of Things.

Many Java virtual machines for wireless sensor networks have been reported, which are implemented on "bare metal" microcontrollers. Squawk [99] targeted the relative rich resource SunSPOT platform. Whereas Darjeeling [16] and Taka Tuka [4] were designed to meet the resource constraints of the so-called Berkeley motes. All of them implement a split architecture, aiming to offload embedded run-time processing. The bytecodes are post processed on host, performing bytecode verification, static linking within group of classes and optimizing bytecodes to reduce code size. The achieved code reduction was up to 3-4 times w.r.t. the original Java classes [16].

These two virtual machines, in fact, are no truly implemented on "bare metal", since they rely in an underlaying operating system, either TinyOS and/or ContikiOS. Nevertheless, it is worth noting that it is simply periodically scheduled a task or process to run the interpreter. They also use the already present hardware abstraction layer (drivers to access the hardware platform) and the communication primitives. As a result, the virtual machines as software platform, currently depend on existing modules and protocol stack to build applications. In other words, the software implemented in Java constitutes still a thin layer. However, low-level hardware devices can be successfully accessed using an object-oriented abstraction, *Hardware objects* introduced by [98], and hardware interrupts by [67].

The potential of middlewares built on top of a Java virtual machines has been assessed implementing a mobile object tracking system [2]. We modeled the application using UML (Unified Modeling Language) and implemented using mobile agents with the help of the frameworks JADE [8] and later AFME [30]. An initial evaluation were made from the extracted metrics from the UML models as well as from the generated Java code. The application developed using AFME framework was deployed on a network of SunSPOT sensor nodes running the Squawk Java virtual machine, confirming that significant consumption of processing resources and energy consumption [29].

Additionally. Mote Runner [22] is an interesting virtual machine that execute "bytecodes" generated from the compiled Java or C# programs. An event-driven programming model is adopted using delegates as primitive run-time types.

The interpretation overhead cost and the extra memory required in Java may argue against interpreted languages adoption in WSN. However, bytecodes show a denser representation than their directly executed counterpart, as a consequence interpreted code exhibits smaller power dissipation during over-the-air reprogramming. Consequently, a huge savings in code space and amount of transmitted information can be obtained when a relatively high rate of code updates are needed [101]. Finally, the case for interpreted languages expands as the processing power consumption is reduced; or stated otherwise, lower processing power enables higher abstraction levels and a wider adoption of wireless sensor networks.

2.3 Low-power techniques and methodologies

In most cases, sensor nodes are powered from batteries, hence the main goal of sensor node design is to extend the node's lifetime. The battery characteristics, including the current rate-capacity effect and the recovery effect, could prevent adopting a simple, and at the same time, precise battery discharge model. In this work we are not considering these second order effects. Decoupling capacitors are usually distributed along the supply bus, which flatten the demand of current drain from the battery. As a result, both effects are minimized, and at the same time the battery usage is maximized [89]. In this context, the metric used for low-power design is energy or average power, which are interchangeable. Moreover, power is hereinafter sometimes interchanged with current, although not explicitly mentioned, when a nominal supply voltage is considered.

Section 2.1.2 presented the energy breakdown inside a processor, describing *where* the energy goes. Next, it is briefly described *why* the energy is consumed in digital circuits and *how* to reduce the power consumption by means of design techniques and methodologies.

2.3.1 Fundamentals of low-power design

The power dissipation of digital circuits are classified in dynamic and static. Dynamic power consumption is associated with the switching of logic values. Static power refers to the power consumed when the device is powered up but it is idle.

The dynamic power has two components: the first corresponds to useful logic operation coming from charging and discharging the output capacitance on a gate; the second is due to short-circuit power that flows through the inverter transistors during a transition (when both are not fully off). The later component is useless power, and it can be minimized by a careful design. The former component can be expressed as:

$$P_{dyn} = CV_{DD}\Delta V f \alpha_{sw}, \tag{2.1}$$

where C is the total capacitance of the circuit, V_{DD} is the supply voltage, ΔV is the voltage swing, f is the clock frequency, and α_{sw} denotes the expected switching activity ratio.

The dynamic power can be minimized performing optimizations at different levels: architectural, logic and circuit design, aiming at reducing the contribution of the different components present in the equation. Some techniques focus on lowering the switching activity in the logic gates for a specific function. Because the dynamic power depends quadratically on the supply voltage (when it is consider rail-to-rail operation, i.e., $\Delta V = V_{DD}$), lowering the supply voltage is an effective way of reducing the dynamic power.

In addition, the relationship between the transistor gate delay, t_{inv}, and supply voltage is given by:

$$t_{inv} = \frac{kV_{DD}}{(V_{DD} - V_{th})^{\alpha}}; \tag{2.2}$$

where V_{th} is the threshold voltage, α has a value in the range 1 to 2, and k is a technology constant [94].

From this it is clear that, to keep the performance, V_{th} must be lowered with V_{DD} (and hence V_{GS}). However, lowering V_{th} results in an exponential increase in the sub-threshold leakage current.

These conflicting design objectives had lead to optimization approaches such as: multi-voltage, where different blocks in a circuit have different supply voltages according to the speed requirements; multi-threshold, trading leakage current for speed at different parts of a circuit; clock gating, stopping the clock to reduce the dynamic power to zero. If some performance degradation is accepted, the operating frequency is lowered, so that the voltage can be decreased, technique know as voltage-frequency scaling. In small features technologies, from 90 nm and beyond, the leakage is becoming increasingly important, and it can not dismissed [45]. A common technique to reduce the static power is to shut down the power supply of an inactive logic block, known as power gating.

2.3.2 Dynamic power management

Dynamic power management (DPM) refers to the design methodology that dynamically modifies the configuration of a system to decrease its power consumption, while meeting some performance requirement. DPM includes a set techniques to achieve energy-efficient computation by selectively reducing the necessary performance or turning off components when they are underutilized or idle.

DPM relies on two fundamental premises: the system computation load is non uniform during the operation time; and second, it is possible to predict with a certain degree of accuracy, the fluctuations of workload [10]. Another important consideration is that the observation and prediction should not involve a significant energy cost, and that this cost must balance the obtained energy saving.

DPM can be applied at different levels, i.e., to the whole system as a unit, or to their constituent components. The system components or modules are supposed to have multiple modes of operation or power states, and that it is possible to dynamically switch between these states. Needless to say, the more power consumption, the higher performance achieved or the more services offered. The transitions between operation modes may have an associated cost, in terms of delay and energy. If the component is not operational during a non instantaneous transition, there is a performance loss. Moreover, the transition energy cost has a major impact on the benefits of DPM, as we shall see later.

Each component can be modeled using a finite-state representation, called power state machine. This simple abstract model holds for many single-chip components like processors and memories [10]. The system can be modeled as the Cartesian product of the finite-state machine of the system components, however neither all the transition nor the states are valid. This approach is used in the next section to model a sensor node build from COTS components. The power model of a component having two power states, namely active and sleep, is shown in Fig. 2.2.

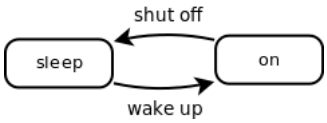

Figure 2.2: Power model of a two states component.

A power manager is the system component that control the states of the other system modules by implementing a policy based on some observation and/or assumption on the workload [10]. The power management approach for defining a policy can be divided into two different categories: non-adaptive or adaptive. The first category includes: greedy and fixed time-out, and the second: predictive wake-up or shut-down. The static techniques are ineffective when the workload is non-stationary; hence some adaptive methods can be applied, for example to dynamically adjust the time-out parameter at the expense of implementation complexity [40].

The break-even time for a sleep state, or any inactive state, is the minimum time required to compensate the cost of entering the state without performance degradation. The inactivity time in the sleep state has a duration equal to the sum of the actual time spent in the sleep state and the time spent to enter and exit it. To be worthwhile entering the sleep time, the inactivity time must be enough to enter and exit the sleep state, and to compensate the additional power consumption.

The break-even time is the sum of the total transition time, T_{tr}, i.e., the time required to enter and exit the sleep state, and the minimum time that has to be spent in the sleep state to compensate the additional transition energy, E_{tr}.

The total transition energy is the sum of the wake-up and the shut-down energies, $E_{tr} = E_{wkp} + E_{sht}$. The average power transition is $P_{tr} = E_{tr}/T_{tr}$.

The break-even time is

$$T_{be} = \frac{E_{tr} - T_{tr}P_{slp}}{P_{act} - P_{slp}}, \quad \text{if} \quad P_{tr} > P_{act}$$

$$T_{be} = T_{tr}, \qquad \text{if} \quad P_{tr} \leq P_{act} \tag{2.3}$$

where P_{act} and P_{slp} are the active and sleep power respectively.

In the case of the telos's microcontroller, the transition time is equal to the wake-up time, since the shut-down time is zero. During wake-up, the microcontroller core and peripheral circuit are powered. It must wait until the digitally controlled oscillator stabilizes to source the master clock and become operational again. The microcontroller wake-up time from sleep state (*LMP3*) is about 6 μs (for a clock frequency greater than 1 MHz). The power consumption required during this process is less than the active power. Consequently, the break-even time is equal the transition time, $T_{be} = T_{tr}$, since $P_{tr} \leq P_{act}$ (Eq. 2.3).

The power saving potential by adopting a dynamic power management depends on system and the workload. Depends on the system through the following factors: the power level of the different states, the performance constraints, and the management policy and implementation of the power manager.

We define an ideal power manager as one that has a priori knowledge of the entire workload trace, so that it is capable of controlling the component optimally. The ideal power manager wakes the component up just in time for serving upcoming requests, if no performance penalty is tolerated, and it puts the component in sleep state at the beginning of all idle periods longer than the break-even time. We name the policy used by the ideal power manager, best-oracle. *Best* because the component is optimally controlled, and *oracle* in the sense that it "knows the future" workload, so that the ideal power manager is able to take optimal decisions.

The simplest policy is based on a greedy method, which turns the component off a soon as it is idle, and the component is turned on as soon as it is required. A major advantage of greedy policy is its simplicity. But it has two important drawbacks: the strategy does not consider the energy cost of transitions, hence, it may put the component in a lower power state for a short period of time, although if it not worth it; and since it takes some time for the component to be operational again, the response time is increased. One of the most common technique to reduce the first effect is fixed-timeout policy. It uses the elapsed idle time to decide a transition to a lower power state. This simple algorithm may improve the efficiency of the greedy policy, preventing energy waste in short idle times at the expense of reducing the energy saving in long idles times.

Consider the corner case of a system or component in which the transitions between power states are instantaneous and have negligible power cost. The optimum policy is greedy, since it worth transitioning to low power states to save energy all the times. Additionally, there is no performance loss, since the component is fully operational as soon as a new service request is received. Intuitively, the lower the latency and the energy cost of the transitions, the more likely greedy policy approaches to the optimum energy savings.

2.4 Characterization of sensor platforms

We conducted some experiments in order to investigate the energy breakdown in a telos sensor mote. The most important components of macro motes are, as already mentioned, the microcontroller and the radio. The node also includes one or more sensors. The sensor in some cases may represent an significant contribution to node power consumption and processor time for acquiring the data from the sensor [100]. One example is CMOS cameras in image applications [96]. Next, for sake of simplicity a sensor with negligible contribution, in terms of power and time, is considered, such as a temperature sensor included in telos mote [90].

2.4.1 Power model

Following the guidelines of DPM, the strategy for reduced power consumption is to keep most of the time the node in states of minimum power.

These states must guarantee proper operation according to requirements, e.g. availability of certain services. The power management approach most widely adopted is greedy.

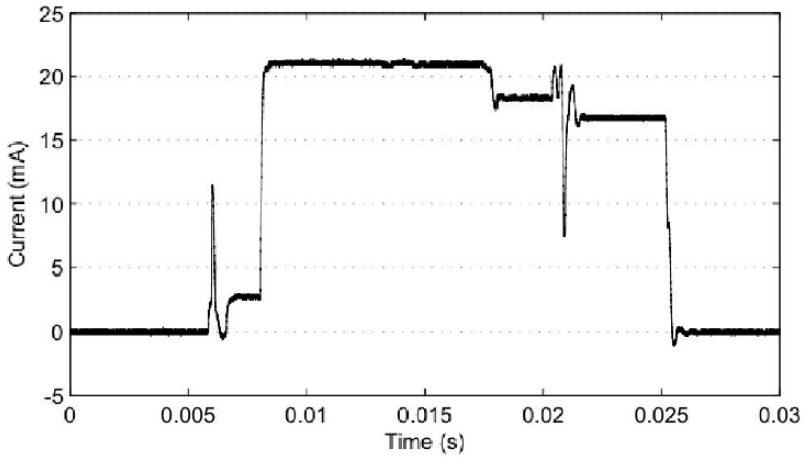

Figure 2.3: Typical current profile of sensor node (telos) [60].

Fig. 2.3 shows an example of current consumption of a telos node as a function of time, where can be clearly identified different current levels. These levels are roughly constant for a certain period of time, which correspond to the given node states. The node states result from the composition of the microcontroller and radio states. The microcontroller operates alternating between two states: sleep and active. The sleep state corresponds to the low-power mode *LPM3*, in which the core processor is power-down and a timer/counter is feed with an asynchronous external crystal oscillator of 32768 Hz. This timer is used by the operating system (e.g. TinyOS and ContikiOS) to generate a system tick, even when the microcontroller is in a low-power mode. In active mode, the processor run normally clocked from an integrated digitally controller oscillator (by default TinyOS at 4.20 MHz and ContikiOS at 3.9 MHz.). The radio has many operational states, but it can be represented accurately, functionally and in terms of power consumption, with three states: sleep, transmitting[5], and receiving. Fig. 2.4 shows the microcontroller (top left), and the radio (bottom left) state charts. However, not all node compound states are valid.

The sensor node state chart (Fig. 2.4, right side) only represents valid states and transitions. Table 2.3 lists node states and the corresponding current consumption[6]. The nominal consumption values corresponds to values extracted from datasheets, and the measured values were obtained on a single sensor node. Despite the fact that the consumption may vary between particular devices and with operating condition [56], these values facilitate a first order comparison of power levels of node states.

From the table is evident that the instantaneous power of the radio (receive or transmit mode) exceeds the processing power (microcontroller in active mode) in around one order of magnitude. Therefore, in the early days of wireless sensor

[5]An single output power level of 0 dBm is usually used, and thus, modeled.

[6]The value showed for the active mode current corresponds to f_{DCO}=1 MHz.

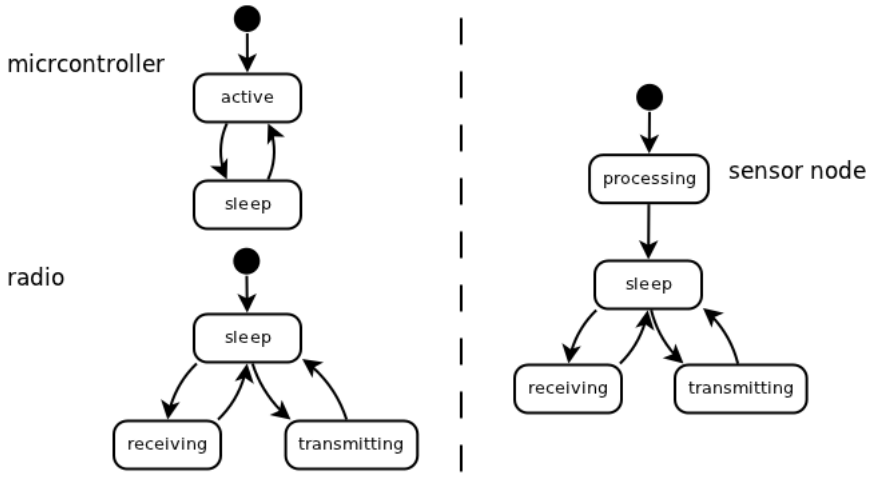

Figure 2.4: Sensor node states.

Table 2.3: Sensor node states and consumption (telos).

	State		Consumption	
Sensor node	Microcontroller	Radio	Nominal	Measured
Processing	Active	Sleep	500 μA	363 μA
Receiving	Active	Receiving	19.3 mA	21.0 mA
Transmitting	Active	Transmitting	17.9 mA	20.8 mA
Sleep	Sleep	Sleep	5.1 μA	5.2 μA

networks, the communication energy cost dominated the overall budget. Consequently a significant research effort has been made since then to reduce the communication energy contribution. The MAC layer design is crucial, since it directly controls the transceiver determining the power profile drain. The use of advanced MAC protocols had helped in improving the energy efficiency for communication [20].

Prayati et al. [91] isolated and measured each contribution to the overall node power consumption, confirming the values above and providing a model from which the total energy can be estimated as a function of the different power levels and the corresponding duty-cycles (i.e. the fraction of time a power contribution is present).

The roughly constant energy consumption associated to different states has been exploited to estimate the energy consumption by the node itself, in ContikiOS (Energest [36]), and later in TinyOS (Quanto [42]). Also, it has been used to profile the radio activity using a second sensor node that logs the sequence of time during which the radio spend in each state [75]. This method had served to assess the communication efficiency in real deployments in precise agriculture applications [76].

In Energest and Quanto the node measure and accumulate the elapsed time in each state. Periodically the values are reported to the sink node where each energy consumption contribution is computed. The energy consumption can be expressed as the sum of each state contribution, computed as the power level multiplied by the accumulated time in the corresponding state. The state transitions contribution can be neglected. The microcontroller wake-up time is about 6 μs, for clock frequency greater than 1 MHz, which correspond to six cycles at 1 MHz and less than two of cycles at 4 MHz. The radio oscillator startup time, i.e. transition from low to active power, is about 600 μs, and corresponds to the time spent to transmit about 18 bytes, roughly the MAC protocol overhead.

The sensor node energy consumption after elapsed a time t is

$$E(t) = P_{prc}t_{prc} + P_{tx}t_{tx} + P_{rx}t_{rx} + P_{slp}t_{slp} \tag{2.4}$$

where $t_{prc} + t_{tx} + t_{rx} + t_{slp} = t$, and the time and power subscripts refers to processing, transmitting, receiving and sleep state respectively, of Fig. 2.4 (right side) and Table 2.3.

Thus, the average power ($P_{avg} = E/t$) is

$$P_{avg} = P_{prc}d_{prc} + P_{tx}d_{tx} + P_{rx}d_{rx} + P_{slp}d_{slp} \tag{2.5}$$

where $d_i = t_i/t$ for $i \in \{prc, rx, tx, slp\}$, the respective duty-cycles.

The expression can be rewritten as

$$P_{avg} = P^*_{prc}d_{prc} + P^*_{tx}d_{tx} + P^*_{rx}d_{rx} + P_{slp} \tag{2.6}$$

where $P^*_i = P_i - P_{slp}$ for $i \in \{prc, rx, tx\}$ is the power increment from the sleep power baseline.

Consequently, the average power consumption can be computed adding to the sleep power baseline consumption, the power contribution of the other states, expressed as the power increment respect to this reference multiplied by the corresponding average duty-cycle.

2.4.2 Energy breakdown

We took two different approaches to estimate the energy contribution of the different component states. The first method was based on the Energest module included in ContikiOS [36]. The average power consumption was estimated using Eq. (2.6) from the information of the time spent in the different states of the microcontroller and the radio. The second one was based on simulations and offline trace analysis, and it was focused on further investigate the microcontroller time breakdown.

The criteria for selecting the case study were: public availability of source files, practical, and almost ready-to-use applications. Unfortunately, the number of cases complying with the aforementioned restrictions are scarce. We chose two data-collection application from the standard distribution of TinyOS (version 2.1.0) and ContikiOS (release 2.5). Both applications are similar, each node of

Table 2.4: Application parameters (size in bytes).

OS	Application	text	bss	data	#functions
TinyOS	MultihopOscilloscope	32058	122	3534	1081
ContikiOS	rpl-collect (udp-sender)	47552	232	9250	489

the network periodically samples a temperature sensor and the readings are transmitted to a sink node using a network collection protocol. MultihopOscilloscope (TinyOS) uses Low-Power listening (LPL) protocol [79] for medium access control, and CTP (Collection Tree Protocol) [46] for network routing. On the other hand, rpl-collect (ContikiOS) rely in ContikiMAC [34,35] for communicating with neighboring nodes, and the Contiki implementation of RPL (IPv6 Routing Protocol for Low power and Lossy Networks) [112], ContikiRPL [107], for routing packages.

The applications were compiled for a telos sensor node using the standard toolchain required by the software platforms (see the corresponding project site for details). Table 2.4 summarizes section sizes and number of functions of the selected applications.

Time measurement in field (Energest)

The ContikiOS module Energest measures the time elapsed in predefined states. The time resolution is 7.8125 ms (inherited form the *rtimer* module, 1/128 sec) Energest includes the sensor states showed in Fig. 2.4 where the state *processing* is split in two states to differentiate the processing performed in a interrupt context from the normal processing (usually named as cooperative to highlight its non-preemtiveness nature). The Energest and the application source codes were modified to remove some limitations, and thus suit our needs (for details see Appendix B.2.1).

The ContikiOS network application comprises two different node applications (executable): udp-sink, the base node connected to a PC, and udp-sender, the application for other sensor nodes. We carried out an experiment running the application using two sensor nodes (udp-sender) and a base node (udp-sink) for more than 20 hours. The sensor nodes periodically send the accumulated time (also temperature reading) in each state to the base node. The base node forwards the information to a PC via USB. The companion application Collect-View shows in real-time the collected data, while it saves the information in plain text to a file. Finally, the saved data was processed to compute the total accumulated time in each state. Table 2.5 shows the results: accumulated time in each state, resulting duty-cycle, power consumption (extracted from Table 2.3 adjusting the processing current to 1.8 mA corresponding to 3.9 MHz), average current in μA and in percentage.

The total processing time, i.e. normal time plus the time in the context of interrupt service routine, represented a duty-cycle of 3.19% for the microcontroller,

Table 2.5: Energest results for udp-sender application .

	Time (ms)	Duty-cycle	Current	Average current	
Processing (norm)	1131047	1.52%	1.80 mA	27.4 μA	16.2%
Processing(IRQ)	1241124	1.67%	1.80 mA	30.0 μA	17.8%
Receiving	391476	0.527%	20.6 mA	108 μA	64.2%
Transmitting	11911	0.016%	19.2 mA	3.07 μA	1.8%
Sleep	71917221	96.81%	5 μA	4.84 μA	2.9%
Total	74289393	100%	-	169 μA	100%

and the remaining 96.81% was in sleep state (LPM3). The radio spent 0.527% of the time in receive mode, and 0.016% in transmit mode. The receive and transmit time added together resulted in a radio duty-cycle of 0.543%. Needless to say, that the application communication requirements directly influences the radio usage, and thus, its energy consumption. However, the results obtained shows that the radio consumption was dominated by the idle listening of the MAC protocol. More precisely, the reception time was on average 30 times greater than the transmission time.

Next, the microcontroller and radio utilization and consumption are compared. The percentages indicates that the processing time was about six times higher than communication time. Now, considering the power level of each state results that the average communication power (transmit plus receive) was only about two times higher than the processing power. It is apparent from this table that current MAC protocols, as ContikiMAC [34, 35] which is an enhanced version of the low-power listening (LPL), achieve an extremely low duty-cycle.

This application example do not process the acquired data, but simply send the raw data to a sink, so that the processing workload roughly corresponds to communication protocols and OS housekeeping only. Consequently, the processing energy is expected to rise with increasing complexity of applications. This results are quite revealing, and push again for processing energy reduction.

Simulation and trace analysis

The second method consists in getting an execution trace, which is processed later. Since current sensor nodes do not support real-time execution trace generation, we simulated the network using COOJA [39]. The telos node-level simulation relies on MSPsim, an instruction-level emulator for the MSP430 microcontroller, which also simulates hardware peripherals such as sensors, radio modules or LEDs. MSPSim is designed to be used as a COOJA plug-in, allowing to access to the MSPSim command-line client from COOJA. We modified MSPSim's code to add a new command for controlling the debug mode, so that it is possible to obtain any node execution trace from COOJA. COOJA is able to simulate the execution of multiples nodes and the interaction between them. Nodes interact by means of message

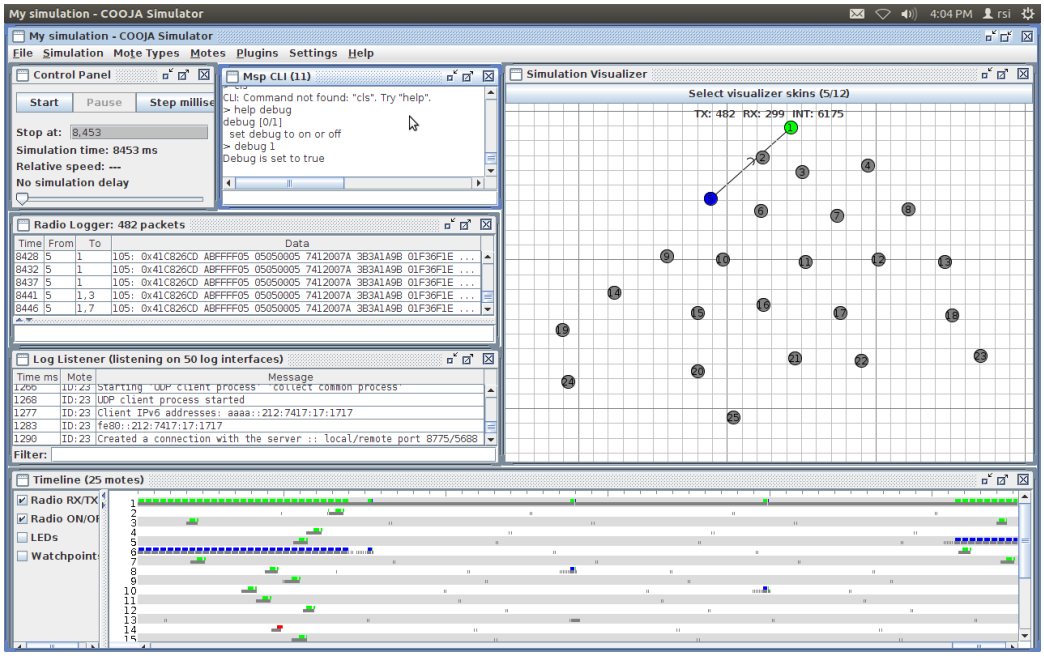

Figure 2.5: COOJA graphical user interface showing: main controls and logger windows (top left), sensor nodes locations and communication exchange, and timeline with the radio activities.

exchange, and COOJA model the radio-propagation, including interference, collision and so on. Fig. 2.5 show a screenshot of COOJA graphical user interface. In that regard, the importance of a complete simulation environment has to be highlighted: executables applications that run in real sensor nodes can be also simulated. COOJA is able to simulate any supported platform, irrespective of the operating system (if any) and the toolchain, provided that applications are compiled to elf format (executable file format).

The execution trace of a particular node can be analyzed to determine the transition between the different power states. In the context of our work, an execution trace is the time series of executed instructions addresses, i.e. a sequence of pairs: access time and address. The access time is considered from the start of the simulation.

From the executable file description (elf) and some knowledge of the embedded operating system, the instruction that forces a transition in the power model can be determined. In this case, the microcontroller and radio power states were considered individually.

For the microcontroller were distinguished three states: sleep, normal processing, and interrupt context. Fig. 2.6 shows the corresponding diagram state. The transitions were defined by the following conditions: to sleep state (*lpm*), the execution of the instruction that modifies the status register accordingly (set the LPM3 bits); to interrupt context (*isr*), the execution of an address corresponding to the start of any interrupt service routine (extracted from the interrupt vector);

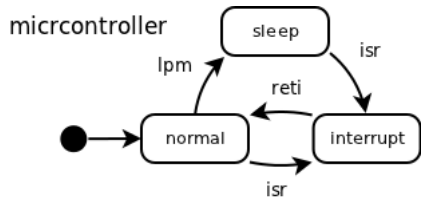

Figure 2.6: Microcontroller power states: sleep, normal processing and interrupt context; and transition conditions: lpm (low power mode), isr (isr start address), and reti (return from isr).

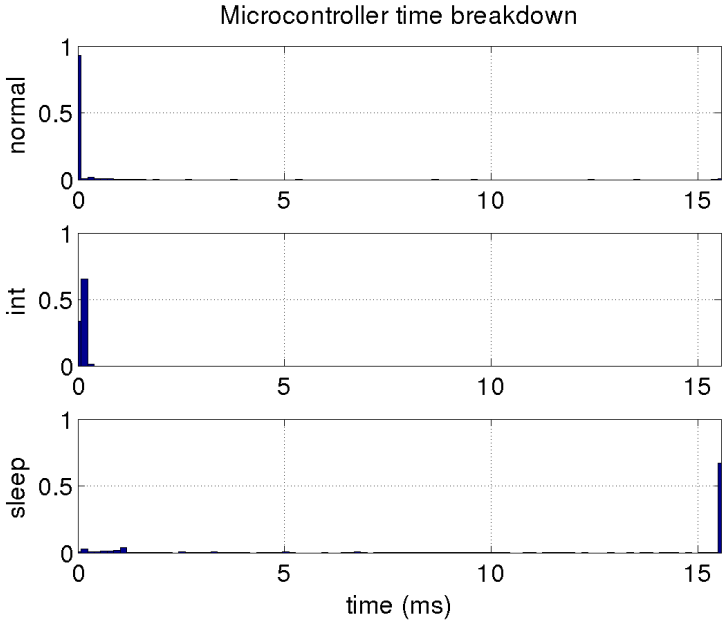

Figure 2.7: Elapsed time distribution of microcontroller states.

and to normal execution (*reti*), start returning from a interrupt service routing, i.e., after executing a *reti* instruction. Notice that the sleep state is always exited when an interruption occurs, and that nested interruptions are disable.

Fig. 2.7 shows the distribution of the elapsed time in the different microcontroller states. The maximum time spent in sleep state (and in any state) was about 15 ms, corresponding to period of the system tick. Most of the cases, more than 90%, the microcontroller stayed in normal processing less than 150 μs (corresponding to the width of the bin in the histogram). Similar results were obtained for the interrupt context, with the difference that in all instances the interrupt context lasted at most 450 μs (3 bins wide)

Fig. 2.8 show a box-and-whisker diagram for the same data, graphically depicting the data quartiles (the box in the middle encloses 50% of the data), indicating variability outside the upper and lower quartiles (using whiskers, the lines extending from the boxes), and marking the outliers (individual points using cross-marks). Inside the figure a zoom shows the detail for normal and interrupt context. From the digram we can see that the maximum time in interrupt context was actually

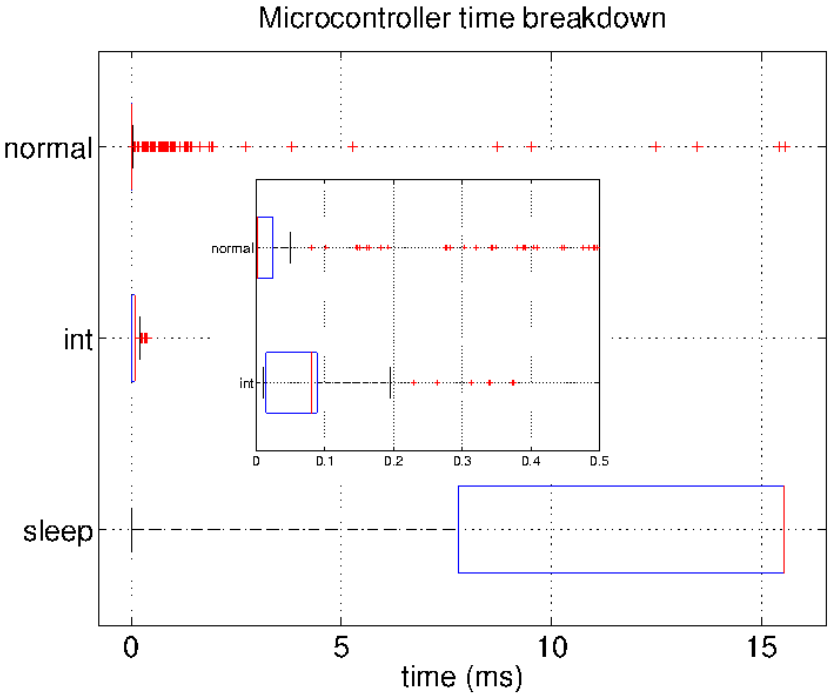

Figure 2.8: Box-and-whisker diagram of the elapsed time of microcontroller states.

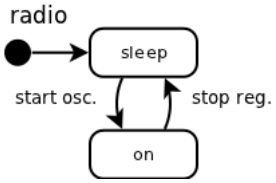

Figure 2.9: Radio power states: sleep and on; and transition conditions: stop reg. (shut off the voltage regulator) and start osc. (turn the oscillator on).

375 μs.

Two states were considered for the radio: sleep and on. The on state accounted for the states with relative high power consumption, and it comprised receive and transmit mode. The state transitions were determined by detecting when it was invoked two particular functions to issue a command to the radio. As shown in Fig. 2.9 a command to shut off the internal voltage regulator (*stop reg.*) triggers a transition to the sleep state; and a command to turn the oscillator on (*start osc.*) forces a transition to on state.

Fig. 2.10 shows the distribution of the elapsed time in sleep and on states of the radio. It can be seen that both present a bimodal distribution. The on state has two mode values, one close to 12 ms, associated to listen events (low-power listening) and the other close to 100 ms associated to transmit and receive. It seems clear that this protocol implementation or its default parameter values are such that the radio usage was not as efficient as ContikiMAC (see [20] for a review of communication protocols). Although, this method could be used to study in

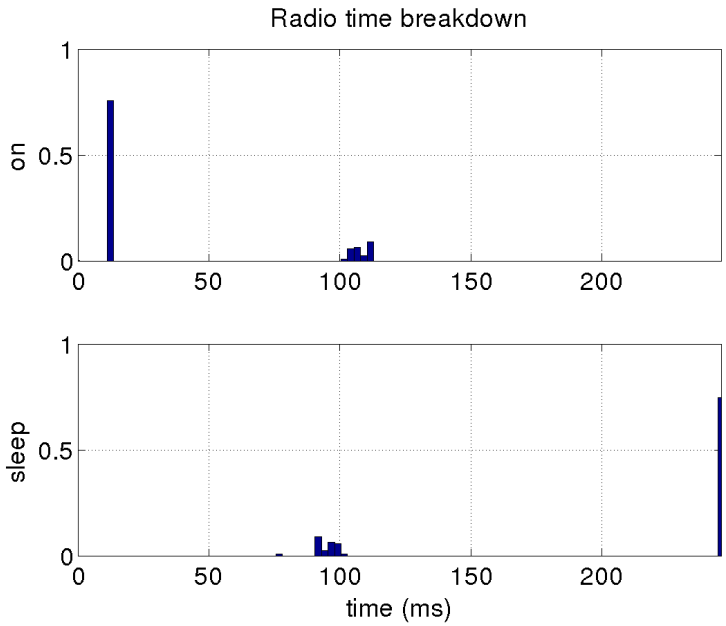

Figure 2.10: Elapsed time distribution of radio states.

detail the radio usage, providing more information than the average time values get by Energest.

Finally, one question that needs to be asked is whether greedy policy is an appropriate strategy for managing the node power states. Fist, it must be pointed out that as a reactive system the node act in response to events, and that most of the cases after finishing the respective processing is highly likely the node will became idle for a while, until the next event. The minimum time in sleep state should compensate the transition cost to be worthwhile. Fig. 2.11 shows the cumulative frequency distribution of the sleep time. From the curve distribution we can see that the sleep time is less than 80 μs at most in 0.8%. We also verified that the minimum sleep time was of 22.8 μs (value not shown in any figure).

The break-even time was estimated to be about 6 μs, less than the minimum time in sleep state. Consequently, greedy is the optimum policy to put the node in sleep state. The wake-up transition from sleep is, albeit small, not null. However, the incremented latency is the wake-up transition time, i.e., 6 μs, comparable to the interrupt latency. Obviously, it make no sense to adopt a predictive wake-up strategy.

2.5 Summary and conclusions

The development of wireless sensor networks on top of high level abstraction has many advantages, as productivity gain, portability, just to mention a few. The most popular software platform are currently TinyOS and ContikiOS, which are event-based software architectures based on different paradigms, component-based

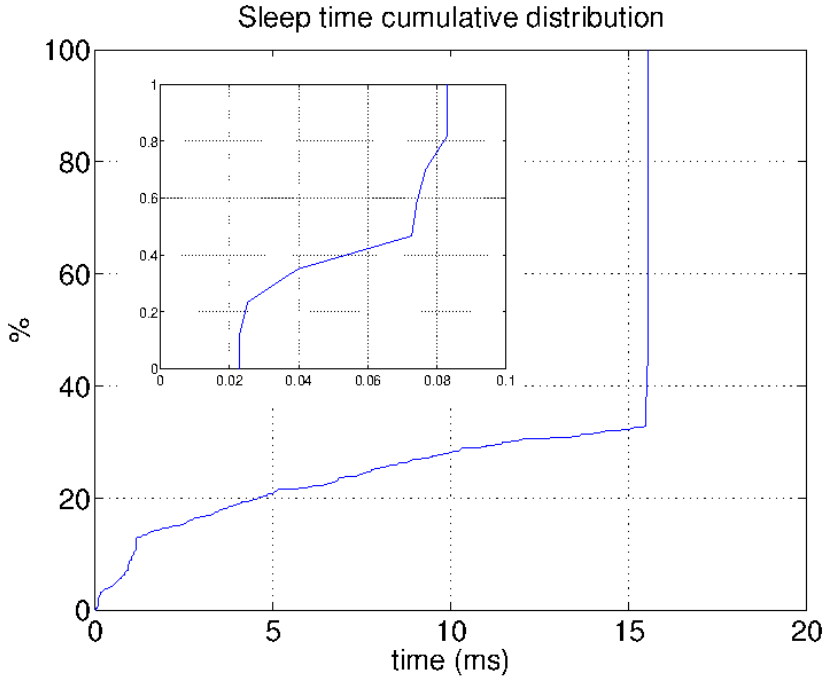

Figure 2.11: Sleep time cumulative frequency analysis.

model and multithreading respectively. However, they share many characteristics at execution time, since both are event-driven in nature. Java is a very popular language among programmers and it definitely requires a much shorter learning curve than nesC and even pure C, used in TinyOS and ContikiOS. It presents other worthwhile features that makes Java virtual machine adoption in wireless sensor networks an interesting option for application development. Emerging virtual machines still constitutes a thin layer, and currently relies on existing modules and protocol stack provided by the underlying OS. Future virtual machines and middlewares for creating progressively more complex applications are expected rely on an underlaying event-driven software architecture, to highly exploit the nature of wireless sensor networks applications. But yet, they would require more and more processing power and available memory, therefore they should be accompanied with energy efficient platforms.

In regards to hardware platforms, there is a clear trend toward enlarging processing power with lower rate of energy consumption rise, exploiting the benefits of the Moore's Law. On the other hand, it was pointed out that the processor spends most of its energy on data supply, i.e., memory access. This result shows that efficient memory design for supplying data to processors is a major issue, and that some research effort should be dedicated to focus on reducing memory energy consumption.

Low-power design requires to manage trade-offs between conflicting design objectives. Some techniques dynamically modify the configuration of a system to decrease its power consumption. For example, power and clock gating, and

voltage-frequency scaling are widely used. The system relies in a power manager to control the other system modules, which implements a policy based on the workload characteristics (the workload statistics may be represented by the probability distribution of the idle periods). The ideal power manager has a priori knowledge of the entire workload trace, and controls the component optimally. Obviously, the ideal power manager is unavailable in practice, and it only serves to obtain the inherent exploitability of the pair system-workload, representing an upper bound of the potential savings. On the contrary the greedy policy is the simplest policy to implement. Additionally, it is the optimum method when transitioning the states has negligible energy cost, and it is tolerated some performance degradation or the wake-up is instantaneous.

The results of the experiments carried out on a telos sensor node showed that despite of the instantaneous power of the radio is tens of times higher than the processing power, the processing time was about six times higher than communication time, as a consequence the communication energy was only about two times higher than the processing energy.

The applications examples in question simply send the data to a sink, so that the processing workload include only communication protocols and operating system tasks. Hence, the increasing complexity of applications is rising the demand for processing power, which in turn pressures to reduce the processing energy dominated by memory.

Chapter 3

Memories for sensor nodes

Memories in sensor nodes are required to hold the program instructions of programmable processors, constants values and temporal workspace data. In almost all cases, they are embedded in a microcontroller or system on chip to serve the processing unit also within it [53]. Additionally, the sensor node may include another memory for bulk storage.

This chapter introduces different memories technologies with the goal of reviewing more deeply those suitable for processors nodes. The first section introduces the fundamentals of semiconductor memories, needed to better understand present technologies and some under development. The internal architecture, and the main organizations are briefly described. The second section presents the Flash technology, focusing on NOR architecture which is appropriate for holding program code, and the static RAM (SRAM), which can be used for both program and data memory. In the following section we present a comparison of Flash and SRAM memories, centering attention on energy consumption across transistor types and operation frequency. Next, a SRAM memory with sleep state is considered, and the average power reduction determined as a function of memory parameters and the workload characteristics. Then, a SRAM with power-manageable is introduced that, unlike the previous memory, includes putting idle banks in sleep state while the memory is active.

We conclude the chapter with a summary, including a brief discussion and giving the main directions in memory optimization for sensor nodes.

3.1 Memory fundamentals

The memories can be categorized according to different characteristics. We intentionally avoid traditional tree-like taxonomies such as the presented by Nachtergaele et al. [81] because they are not suited for our needs, since many introduced characteristic applies to most memory types.

Depending on whether a memory is capable to retain its contents even if the power supply is switched off, they can be classified in two types: volatile and non-volatile. In the one hand, volatile memories requires power to maintain the stored

information, and consequently loses its stored data when the power supply memory is turned off. In the other hand, non-volatile memories can retain the stored information even when not powered preserving the data stored during long periods with the power turned off. Examples of non-volatile memory includes Flash memory, and older technologies such as permanent read-only-memory (ROM), programmable read-only memory (PROM), erasable programmable read only memory (EPROM) and electrically erasable programmable read-only memory (EEPROM) which evolved to the newer and popular Flash. More recently, emerging non-volatile memories are available, e.g. magnetoresistive RAM (MRAM), some of them even integrated into commercial microcontrollers , such ferroelectric RAM (FeRAM or F-RAM) [113].

Examples of volatile memory are dynamic RAM (DRAM) and static RAM (SRAM). They are usually used as primary memory and cache memory in personal computers respectively, since SRAM is faster than DRAM, while consumes more energy and offer lower memory capacity per unit area. SRAM is widely adopted in small embedded systems, such as sensor nodes.

Random access refers to the capability to access memory elements at an arbitrary location in equal time (the term random is used as having no specific pattern), thus memories implement a direct-addressing scheme for elements access. On the other hand, sequential access refers to memories in which sequential element are accessed in relative short times while a remote element takes longer time to access. In the former case data can be efficiently accessed in any random order while in the latter case just contiguous elements are accessed efficiently, so they are adopted for very specific functions.

Read-write refers to the ability to write a memory element as easily as read from. On the opposite side is read-only memory (ROM), meaning that the data stored in it can never be modified (e.g. mask ROM fabricated with the final data stored in it). In its strictest sense the term ROM refers only to read-only memory, however, sometimes is used less rigorous way to name some types of memory which can be erased and re-programmed many times in the field. Although, the process of writing takes longer than reading the memory, and memory must be erased in large blocks before it can be re-written.

Unfortunately, many adopted terms related to memory types are misleading, guiding to misconceptions. Some of them are mentioned below. The term ROM gets stuck for Flash memory when used in substitution of mask ROM to storage of program code and constants. However, Flash might be considered as a read/write memory. Similarly, firmware, term coined to describe software in non-volatile memory that rarely or never change during the product lifetime, sometimes is used to name software in embedded systems, despite it is expected to be updated relatively often. Modern types of DRAM read the data in bursts, so in strict sense are not a RAM. Finally, one of the great contradictions is when the term random-access memory is occasionally used as synonym of read/write memories, while even first ROMs were read-only memories with random access. While some of this terms and usage are generally accepted, this work will use the strict definition for each type of memory characteristic.

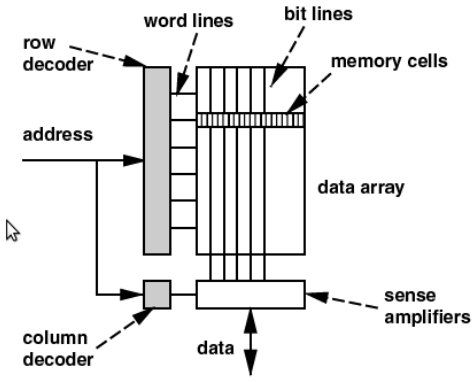

Figure 3.1: Simplified memory structure [52].

Other important parameters are: access time, read/write endurances, operation speed, cell size, capacity, and operation voltages. Based in considerations described in the next section, and with the use of memories for storing program code in mind, the SRAM and the Flash memory are described more deeply in the following sections.

3.1.1 Memory internal architecture

Most semiconductor memories are comprised of a memory cell array and peripheral circuits. A memory cell stores one bit of data (with the exception of some kind of Flash memory that store multiple bits per cell). Peripheral circuits, needed to access the memory cells, include decoders and drivers for rows and columns, amplifiers, I/O and control logic circuits. The memory cells in the array are grouped into words of fixed size, W, which can be accessed by an address, in a direct access implementation. The maximum number of addressable words M with an address of N bits is 2 raised by N ($M = 2^N$).

In a straightforward implementation, the $W \times M$ memory cells are arranged in a two-dimensional array, which results in a quite slender array. A row is selected by decoding the corresponding address. Each column of the word correspond to a bit. The row and column lines are named wordline and bitlines, respectively. Actually, physical considerations, such as the capacitance and resistance of the lines, limits the size of the array, so that the array might be divided into multiple blanks. The memory address is logically divided in three fields that select the bank, the column and the row of the word in the memory. Each field is decoded so that the correct bank is selected through the corresponding driver and the right cells in that bank are enabled. In a write operation the data is driven into the cells, and during a read operation the store data is driven form the cell to the detection circuits. Particular circuits are used for reading and writing, according to the type of the memory and theirs requirements. These peripheral circuits are subject to power optimization as any logic circuit, applying the techniques presented in the previous chapter. Fig. 3.1 shows a simplified typical memory structure.

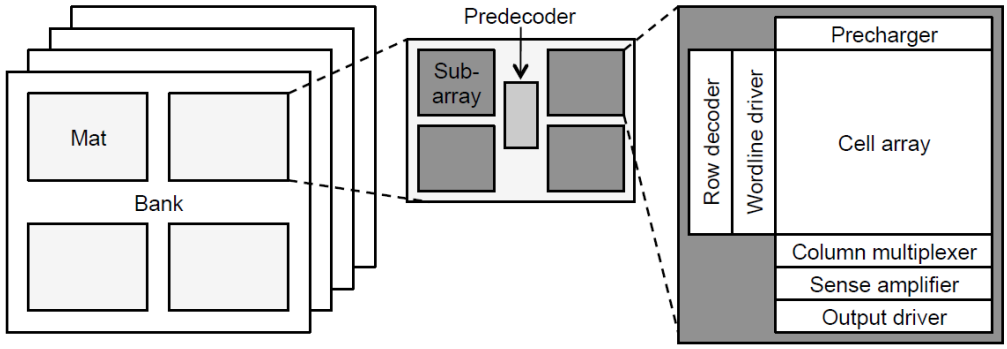

Figure 3.2: Memory organization in three hierarchy levels: bank, mat and subarray [33].

The internal architecture is usually organized in a hierarchical fashion to optimize access times, area and power consumption. Fig. 3.2 shows a memory with three hierarchy levels, namely bank (already mentioned), mat, and subarray, for the particular case of SRAM but it is generally applicable in any memory type. The bank is the top-level structure. It is a fully-functional memory unit which can be operated independently and even concurrently accessed. Depending on its size, one memory can have can have multiple banks. Each bank is comprised of multiple mats that are connected together in either H-tree or bus-like fashion. The mat is composed of subarrays. There is one predecoder block per mat. The subarray is the bottom-level structure, and contains peripheral circuitry including row decoders, column multiplexers, detection circuits and output drivers. The aforementioned limits in the hierarchical structure are arbitrary, and the place for a specific block or function is chosen accordingly, trading different properties or characteristic.

3.1.2 Memories for embedded systems and sensor nodes

A programmable processor needs at least a memory to hold the program code and another for temporal data. A microcontroller usually has two kinds of memory: a read-write memory (e.g. SRAM) and non-volatile memory (e.g. Flash). The former memory is needed for storing temporal data (variables) that are modified at run time, and the later for storing the program and constants, retaining its content even though it is not powered. The code can be executed directly from non-volatile memory, known as execute in place (XIP), in such a case the random-access feature is needed. Or the code can be first copied from Flash to SRAM, technique known as shadow memory [12]. In this case, the non-volatile memory can have sequential access and it can be relative slow, since the content is copied at boot time, prior to the application execution from SRAM, which is faster than Flash allowing higher clock frequencies.

SRAM memory is used in embedded system as the read/write memory required for variables and temporal data, since the SRAM is fast enough to feed data to the processor, avoiding power hungry power cache memories needed to speed up

slow DRAM.

As will be discussed later in Section 3.2, accesses to SRAM generally consumes much less energy than Flash. So, executing code from SRAM consumes considerable less energy than executing from Flash. However, SRAM occupies more area per bit than Flash memory. Consequently, there is a trade-off between area cost and energy reduction that should be carefully considered.

On the other hand, the recently introduced FRAM memory in the microcontroller market opens new possibilities, where applications benefits from non-volatile storage with faster and lower energy write access. However, to hold the program and constants as a direct substitution of Flash memory does not has any clear advantage. Additionally, despite of having faster write cycles compared to Flash memory, FRAM still present significant higher energy consumption than SRAM [1], preventing its adoption it as the temporal data memory.

3.1.3 Memory organizations

The most common memory organization for embedded system are summarized in Fig. 3.3, either suitable for program or data memory. The first is the simplest and commonly adopted in low-end microcontrollers, in which the whole address space is mapped into a single memory [1]. Several optimized organization have been proposed with performance and energy enhancement, such cache and scratch-pad memories. The main concepts may be applied to embedded (on-chip) and off-chip memories, and even to different memory technologies.

In the embedded area, scratch-pad memories (SPMs) are preferred over caches and are widely employed because they achieve comparable performance with higher power efficiency. The most frequently accessed addresses are allocated to the scratch-pad, while the remaining content is assigned to the main memory (Fig. 3.3, darker gray depict more relative accesses). Since, each access to the scratch-pad account less energy compared to the main memory, significant energy saving are achieved [85]. The contents of the SPM are either static and do not change at run time, or dynamic, that is, the contents change during the operation time to reflect the most frequently accessed data at any given time. The static allocation is accomplished at compilation time, while the dynamic must be performed at run time by specialized libraries. Dynamic allocation provides flexibility to adapt to workload statistics that varies in time, at the expense of implementation complexity and execution overhead. In some embedded systems simplicity is more desirable than flexibility and improved energy reduction, so that, static allocation is favored. In order to be able to exploit fully the potential of the scratch-pad memory with static allocation, the scratch-pad memory must contain almost all the "hot spots" of the accessed address space during the program execution.

Menichelli et al. [78] investigated the memory access distribution on embedded applications. They selected a set of application benchmarks from Mibench [48]. A trace of the fetched addresses were obtained by means of instruction-level sim-

[1] We have intentionally excluded the discussion between von Neumann and Harvard architectures

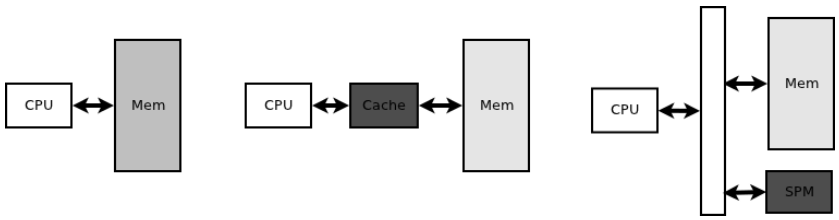

Figure 3.3: Common memory organization: single memory and address space, cache and main memory, scratch-pad and main memory.

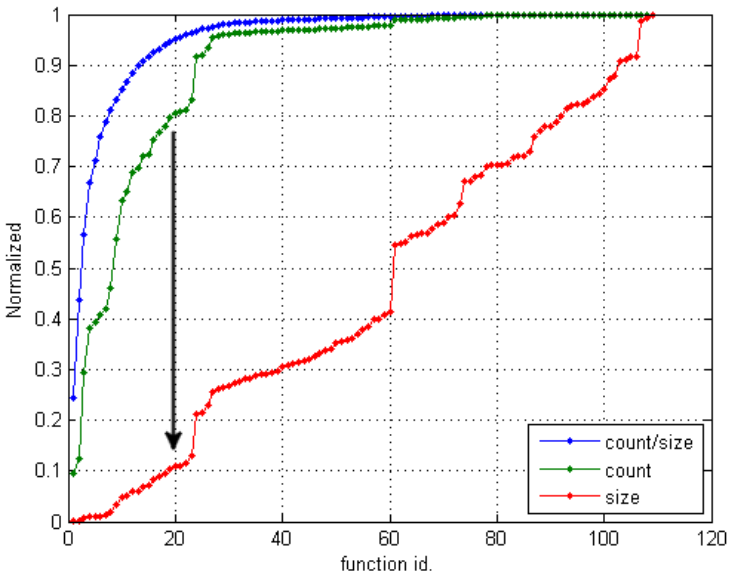

Figure 3.4: Normalized cumulative frequency analysis for the memory access.

ulations of an ARM processor platform. They verified that the most accessed addresses fit on a very small portion of the address space, showing the effectiveness of this memory organization.

Similarly, we follow a analogous procedure considering the MultihopOscilloscope application (TinyOS) and the method described in Section 2.4.2 to get memory access traces. The unit of allocation, i.e., the block to be allocated into the scratch-pad or into the main memory, were defined equal to the relocation unit of the linker-locator, that is, the object codes corresponding to each symbol of the executable program: application and library functions plus compiler generated global symbols.

A normalized cumulative frequency analysis were carried out considering the allocation units. The following steps were then taken. First, it was computed the access count to each allocation unit for the whole trace. Then, for each allocation unit the access count was divided by its size, obtaining the count-to-size ratio. Finally, the allocation units are ordered in ascending order by the count-to-size ratio. The allocation units with higher count-to-size ratio comes first, and they are

preferred to be assigned to the scratch-pad, since leads to higher profits. Fig. 3.4 shows the normalized cumulative values of the memory access count-to-size ratio. It can also be seen the cumulative size and count, that is the total size and count as more allocation units are considered for the scratch-pad.

Menichelli et al. have found that the 80% of the most accessed memory addresses fit in a SPM of size less than 1% of the total footprint. It can be observed that for our experiment, Fig. 3.4, the 80% more accessed allocation units occupied more that 10% of the total footprint, corresponding to the first 20 functions (indicated with the arrow). Thus, the power reduction of SPM in this application example would not be significant, because it did not exhibit enough "hot spots". We argue that the access distribution spread along almost all the address space is typical in event-driven application, as seen in Section 2.2.

Finally, this characteristic is highly exploited in the present work, proposing the adoption of banked memory with power management in wireless sensor networks (although, it may be used in other event-driven application context). This architecture is proposed to be used as a substitution of the single memory in the first category in Fig. 3.3. Nevertheless, the concept could be generalized and combined with other organizations. This memory architecture is introduced later in this chapter (Section 3.4).

3.2 Flash and SRAM memories

In this section, Flash and SRAM memories technologies are presented more in detail, particularly the consumption issue, central to embedded system in general and sensor node in particular.

3.2.1 Flash memory

Flash is a non-volatile memory which can be electrically erased and reprogrammed. This device was developed from EEPROM , but unlike EEPROM the Flash memory is able to erase a block of a data "in a flash", fact that give the name to this memory.

The core of a flash is a non-volatile cell build form one transistor with a floating gate where electrons are trapped onto. The write or program operation have an effect on the the threshold voltage of the cell, changing its value and as a consequence the current/voltage characteristic of the cell, i.e., I_{DS}-V_{GS} curve. Fig. 3.5 shows a cell and the current/voltage characteristic for a logic value "1" and a logic value "0".

In Flash memories the erased operation (writing of logic "1") is slow but independent of the amount of cells being erased. So that the cells are grouped in large blocks to obtain significant speed advantage over old-style EEPROM (in which cells are erased individually), when writing large amounts of data. While, on the other hand, the program operation (the writing of logic "0") can be accomplished on individual cells. As a consequence, before any writing can take place on a particular cell, all the cells of the corresponding block must be erased. The fact

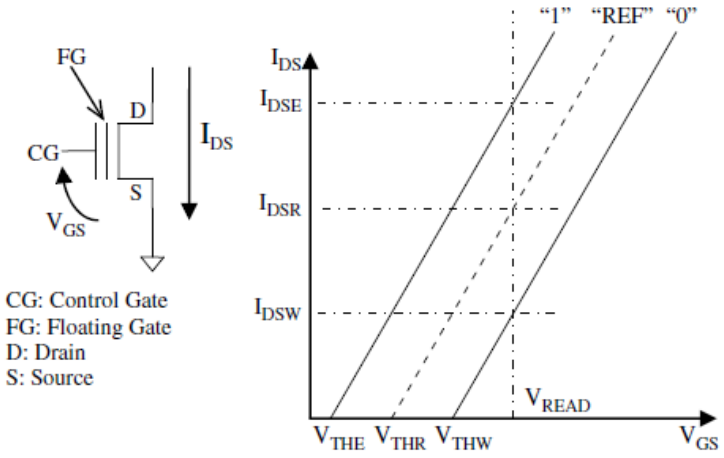

Figure 3.5: Flash cell and the current/voltage characteristic for the "1" and "0" logic values [26].

that the erase operation is restricted to blocks, allows one to design the array in a compact configuration, and therefore in a very competitive size, from an economic point of view.

The read operation of a cell is performed applying suited voltages to its terminals and measuring the current that flows through the cell, further explained in the next section for two particular architectures.

NOR and NAND architectures

Based on the described cell, different memory organizations are obtained depending on how they are interconnected. The most commonly Flash memories are NOR and NAND, which are named after the respective logic gate. The former aims at fast random access, and the later aims at high bit density. In a read operation NOR and NAND memories use the current in different ways, as a consequence they present different energy and access time performance. Fig. 3.6 shows the connection of NOR and NAND cells.

The floating gates of a NOR flash memory are connected in parallel to the bitlines, resembling the parallel connection of transistors in a CMOS NOR gate. This allows individual cell access to be read and programmed. NOR-based flash has long erase and write times, but since it provides full address and data buses, it allows random access, and as a consequence usage for code storage and direct execution. The NOR type offers fast read access times, as fast as DRAM, although not as fast as SRAM or ROM. The NOR architecture requires one contact per two cells, which consumes the most area of all the flash architecture alternatives.

The cells of a NAND flash memory are connected in series, in a string shape, resembling a NAND gate. The NAND string is composed of flash gates, two select transistors, one source line, and one common bit-line contact for all its cells, as shown in Fig. 3.6. The lack of metal contacts in between cells of NAND string,

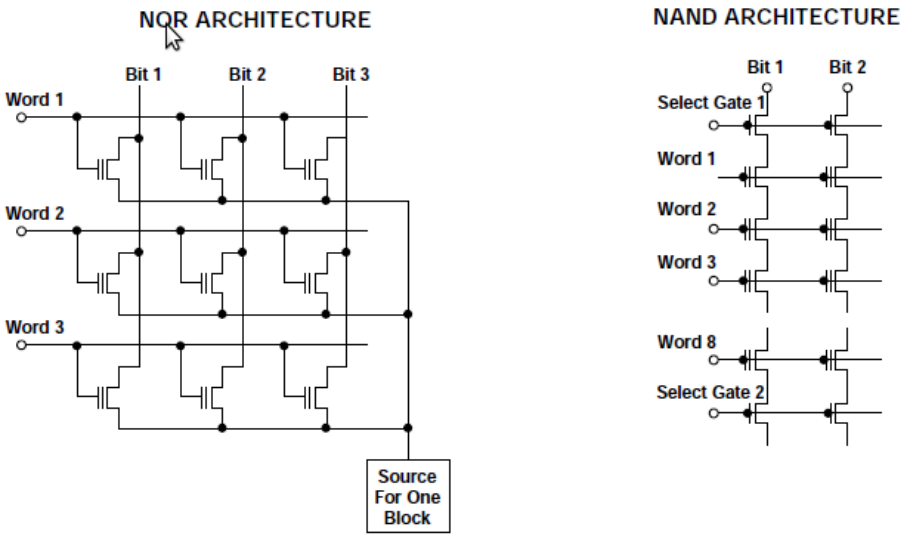

Figure 3.6: Flash architectures: NOR and NAND [74].

allows a more compact cell size than NOR. The occupied area is $4F^2$ for a NAND cell, while $10F^2$ for NOR cell [57]. Thus, the NAND flash memory has the smallest cell size, and so the bit-cost is the lowest among flash memories.

The NAND flash trades the random access feature to reduce the chip area per cell, therefore higher density and greater storage volume than NOR flash [13]. Since NAND cells are denser than NOR cells they achieves reduced erase and write times. The NAND flash memory and the NOR flash memory are required to operate at the same supply voltage as that of the controller devices for simplicity in the low-power systems.

Flash energy consumption

In the NOR Flash architecture, the read of a matrix cell is done in a differential way, i.e., making a comparison between the current of the read cell and the current of a reference cell which is physically identical to the matrix cell and biased with the same voltages V_{GS} and V_{DS}. In the case of memories storing only one bit per cell, the electrical characteristics of the I_{DS}-V_{GS} of the written cell (logic "0") and of the erased cell (logic "1") are separated as was sketched in Fig. 3.5. This is due to the fact that the two cells have different threshold voltages V_{THW} and V_{THE}. The threshold voltage of the reference cell is placed between the erased and the written cell characteristics, so that the two different logic values are correctly distinguished. The current of the cells is converted into a voltage by means of a current to voltage converter (I/V), a circuit able to supply to the output a voltage whose value depends on the value of the current at its input. The voltages are then compared through a voltage comparator which outputs the cell status, i.e. the logic value.

The involved current are relatively large, between 1 to 100 μA [26,54,83]. This

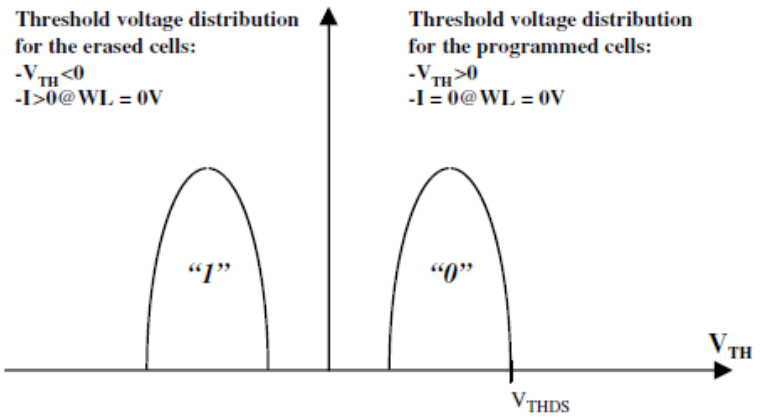

Figure 3.7: Threshold voltage distribution in a NAND cell, for erased and programmed [26].

current level allows a high speed sensing operation, achieving random-access read operation up to 25 MHz, and based on a speed-enhanced cell and architecture optimization, up to 80 MHz [54].

In NAND Flash memories, an erased cell has a negative threshold voltage, while a programmed cell has a positive voltage. The cell string are placed together with two selection transistors at both ends, as depicted in Fig. 3.7. To read a particular cell, all the other gates of the string are biased with a high voltage. This ensures that they act as pass-transistors, irrespective of their threshold voltage (i.e., they are erased or programed). The gate of the cell to be read is set to 0 V, so that the stack will sink current if the addressed cell is erased. On the contrary, if the cell is programmed no current is sunk through the string.

Because the stack is composed by several transistors in series, typically in groups of 16 or 32, the current to be sensed in the NAND structure is very low, in the order of 200-300 nA. This weak current is unfeasible to be detected with a differential structure as the used for NOR architecture. Charge integration is the reading method used in NAND architectures. The parasitic capacity of the bitline is precharged to a fixed value. Subsequently, if the cell is erased, the bitline is discharged by the sink current; on the contrary, if the cell is programmed, the bitline remains at the its initial value. Then, the voltage final value is detected and latched. The discharge time is carefully tunned, and it is about 5-10 μs [26].

In conclusion, the different reading techniques used in NOR and NAND architectures affect the time and current involved in a read cycle.

To the best of our knowledge, there is no report that compares the read energy of both architectures. Considering the current values and the time it takes a read cycle, we consider that the NAND and NOR read energy are in the same order of magnitude.

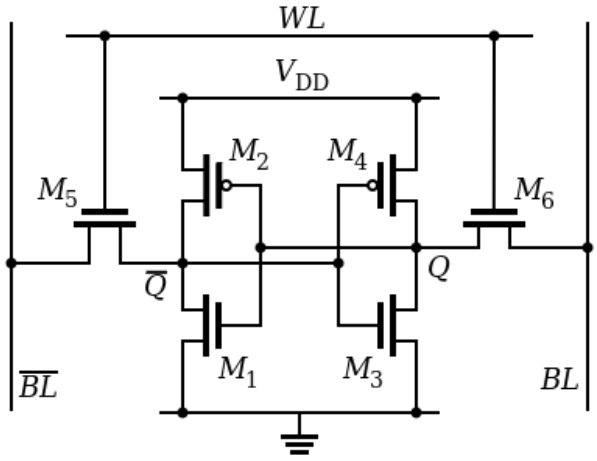

Figure 3.8: A six-transistor CMOS SRAM cell.

3.2.2 SRAM memory

SRAM memory is the first choice in small embedded systems, such as sensor nodes, for temporary data. The most adopted SRAM cell is composed by six transistors (as shown in Fig. 3.8): four transistors forming a flip-flop and two transistors to connect the flip-flop to the internal circuitry for accessing purposing. The flip-flop is a bistable circuit composed of two inverters, in which the bit is stored as a logic value. The flip-flop sides are connected to a pair of lines, B and $\bar{B}$, in columns. The transistor gate of the access transistors are connected to the so called wordline. Each wordline gives access to multiple cells in groups, forming a row.

When the cell is idle, the access transistors are off and the data is retained in a stable state. When a cell is accessed, the flip-flop sides are connected to the internal SRAM circuit by turning on the access transistors. The selected row is set to V_{CC}. In a read operation the bitlines are connected to the sense amplifier that detects the stored data in the cell. This logic signal is feed to the output buffer. During a write cycle the data coming from input are connected to the drivers that force the data onto the flip-flop. After a read or write operation completion, the wordline is set to $0V$. In the case of a read operation the flip-flop keeps the original data, while in the write operation the new data loaded during the cycle is stored on the latch.

SRAM energy consumption and optimizations

The SRAM power consumption can be classified as usual in dynamic, coming from switching capacitances, and static, due to leakage currents in transistors [83]. The switching of long and heavily loaded bit and wordlines is responsible of the major dynamic power cost. Initial works had tackled the dynamic power consumption and a wide range of solutions has been proposed [86] based in memory partition. This has an influence on the effective capacitance and the switching factor (C and α_{sw} of Eq. (2.1) in Section 2.3.1). Applying such techniques, SRAM results in a

relatively low dynamic power, because only a small portion of the whole memory switches at any given time. Additionally, static power may dominate in SRAM because a very large part of the memory is quiescent most of the time and depends directly on the number of transistors. So that, the larger the memory array, the more static power is drain. In CMOS technologies the static power is mainly due to leakage through off transistors, subthreshold leakage, and also due to gate leakage from 45 nm and smaller feature sizes, and both worsens with technology scaling [45]. In this scenario, the reduction of both dynamic and leakage power in the memory subsystem is definitely a primary target in future sensor node design.

Dynamic energy reduction: partitioning

Partitioning is one of the most successful techniques for memory energy optimization, where a large memory array is subsequently divided into smaller arrays in such a way that each of these can be independently controlled and accessed [73]. The access energy is reduced as banks becomes smaller, i.e. more banks are added for a given memory size. However, an excessively large number of small banks leads to an inefficient solution due to an increased area and also energy cost because of wiring overhead. Consequently, the aim of the partitioning is to find the best balance between energy savings, and delay and area overheads [32]. Several sub-arrays share decoders, precharge and read/write circuits, where internal control circuitry provide mutually exclusive activation of those sub-arrays. Moreover, the internal control circuitry is merged with row/column decoders to generate sub-array selection signals, so introduction of extra circuitry is efficiently limited. Nowadays, memory arrays are generally partitioned horizontally -using a divided word-line technique, and vertically using a hierarchical divided bit-line technique [32].

Static energy reduction: low-leakage sleep state

The major source of leakage current in low power SRAM is due to the retention power in the memory cells, since the peripheral circuit contributions are negligible [64]. Several techniques have been proposed to reduce the SRAM sub-threshold leakage power, at circuit and architectural level. Two options are available: using a power-off state that almost remove the leakage but the information stored is lost, and using a sleep state that retains data while consuming less leakage power. In this work we are considering only state-preserving mechanisms. It is worth emphasizing that any realistic leakage power reduction technique must ensure that the data is reliably retained during the standby period. Besides the leakage reduction goal, the energy overhead consumed during the transition to active state is essential, since it has direct impact on the overall energy reduction. The most effective techniques to date are based on voltage manipulation: lowering the supply voltage, increasing the bias voltages of the transistors inside the cell or a combination of both [92]. The most straightforward voltage scaling approach to lowering standby leakage power in memory sections is reducing its supply voltage, V_{DD}. This method reduces powers because of the voltage reduction and the leakage current reduction. The leakage current varies exponentially as the V_{DD} is increased [17, 47]. Using this

approach, it has been shown that any SRAM cell has a critical voltage (called the data retention voltage or DRV) at which a stored bit (0 or 1) is retained reliably [17].

Qin et al. developed an analytical model of DRV to investigate the dependence of DRV on process variations and design parameters [73]. However, as a result of process variations, the cell's DRV exhibits variation with an approximately normal distribution having a diminishing tail, consequently the worst-case standby supply voltage should be chosen, i.e. a voltage value that is larger than the highest DRV among all cells in an SRAM. A test chip was implemented and measured, demonstrating the feasibility of a leakage power reduction of 97% [73].

In contrast to the worst-case design, Kumar et al. proposed a more aggressive reduction of the standby supply voltage using error-control coding to ensure reliable data-storage [14]. A memory architecture based on the Hamming code was proposed with a predicted power reduction of 33% while accounting for coding overheads [14]. However, this techniques should be carefully weighed against the active and static power of the extra cells and components [92].

When a SRAM memory is in the sleep state, the memory cell needs to go back from the data retention voltage to the active voltage, before an access could be successfully achieved. The source voltage recovery causes the loading of internal capacitances, with an associated energy cost. The transition delay from active to sleep doesn't have any impact since the next clock cycle another bock will be accessed. On the contrary, when a block goes to active mode the processor must stall until the V_{DD} is reached. The maximum reactivation current and the wake-up latency are strongly correlated. Calimera et al. [7] have proposed a new reactivation solution that helps in controlling power supply fluctuations and achieving minimum reactivation times of a few clock cycles at the cost of increased area.

In addition, as already indicated in Section 2.3.1 the effective power reduction depends on the system parameters, in this case the leakage power reduction of the sleep state and the wake-up energy of the memory, and the workload statistics. Both factors are fundamental for a substantial energy savings.

3.2.3 Energy comparison: Flash vs. SRAM

As stated before, SRAM is the sole choice for read/write memory in small embedded systems, mainly due to its very low power consumption. However, the question about what it is the best option for program code and constants is still unanswered. If the required clock frequency is not relative high, a natural alternative is the NOR-flash for executing in place. However, SRAM memory has some interesting features as a substitution of flash, such as its lower read time and energy, and, in case of a program update, substantively lower write energy. Additionally, SRAM reaches higher frequency than Flash by several orders of magnitude.

In order to better compare and evaluate both technologies we used NVSim [33][2], an estimation tool for non-volatile memories. This tool is based on the popular CACTI estimation tool [105], but developed from scratch. Several technologies

[2]Project site: http://www.nvsim.org

Table 3.1: Read energy.

Read energy (J)	HP	LOP	LSTP
Flash	11.036×10^{-9}	11.925×10^{-9}	14.299×10^{-9}
SRAM	36.164×10^{-12}	20.748×10^{-12}	52.761×10^{-12}

Table 3.2: Leakage power.

Leakage power (W)	HP	LOP	LSTP
Flash	155.1×10^{-3}	3.426×10^{-3}	73.37×10^{-6}
SRAM	1.805	105.0×10^{-3}	703.4×10^{-6}

are modeled including NAND Flash and SRAM memory. They claim that the Flash memory modeled correspond to a NAND architecture. However, the access times are close to the values reported for NOR memory [54]. Taking into account the previous considerations regarding a similar read energy for both architectures, we based our findings on the estimation given by NVSim. Hereafter, we refer to this memory simply as Flash.

We also tried to verify the results with the data published by ITRS [57]. Nevertheless, the reported values are useless since the given values are in terms of current per cell, and do not consider the differences that arises by varying the memory size.

We obtained the main characteristic parameters for Flash and SRAM for several configurations, including three types of devices: high performance (HP), low operating power (LOP) and low standby power (LSTP). HP devices aims at high operating clock frequency, so that transistors have the highest performance and the highest leakage current. LOP transistors have the lowest V_{DD} for low operating power, trading lower performance and off-current. Finally, LSTP aims at the lowest possible static power dissipation resulting in the lowest speed performance and off-current (highest threshold voltage V_{th}) of the three [58]. The used memory configuration was: 512 KB size and 64 bits word width [3].

The estimation tool outputs, among other results: the power leakage, the read energy and latency. The later parameter correspond to the inverse of the maximum operational frequency. The read energy, leakage power and maximum operational frequency are shown in Table 3.1, Table 3.2 and Table 3.3, respectively.

The data from these tables are quite revealing in several ways. For both memory types the effects of choosing different transistor types are similar. The leakage is reduced by 2500x when comparing HP with LSTP. The HP frequency is slightly higher than LSTP (x5 in Flash and 1.5x in SRAM). The LOP is an intermediate

[3]For further details about the memory and cell configuration, please see the Appendix A.2 in page 90.

Table 3.3: Maximum frequency.

Maximum frequency (Hz)	HP	LOP	LSTP
Flash	23.8×10^6	9.77×10^6	4.83×10^6
SRAM	5.66×10^9	4.36×10^9	3.37×10^9

case with an improvement respect to HP of 50x and 20x in leakage power reduction for Flash and SRAM respectively, at a cost of reducing a little the maximum frequency (2.5x and 1.3x). There is no significant effect on the read energy, but a small reduction when a LOP device is used. The higher reduction is 2.5x for SRAM if compared to LSTP.

On the one hand, SRAM read energy is much lower than Flash memory, between 300x and 600x depending on the transistor type. On the other hand, the SRAM leakage power is 10x higher if compared to Flash. Flash end up in a reduced leakage power at the cost of increased read energy. The Flash cell does not leak, as opposed to the flip-flop in SRAM cell. In contrast a simple differential read circuit can not be used due to lower currents present in the read operation. Accordingly, the trade-off between leakage power and read energy must be carefully evaluated.

The total read power of a memory depends on the operating frequency, through the dynamic read power, and the leakage as static power. The average read power can be expressed as,

$$P_{read} = P_{sta} + E_{dyn} f \qquad f \leq f_{max} \tag{3.1}$$

where P_{sta} is the static power and E_{dyn} is the energy consumption per read cycle.

Fig. 3.9 shows the total power as a function of the frequency for Flash and SRAM considering the different device types. From the curves we can graphically appreciate some of the previous observations. The static power of Flash memory is lower by 10x than SRAM for the respective device type and SRAM has higher maximum frequency and lower read energy (slope of the curve) than Flash.

Considering that for different transistor types there are no significant differences in energy access, if there is no restrictions for performance, the reduced leakage obtained from LSTP is preferred for Flash and SRAM. From Fig. 3.9 one can see from the power curves that LSTP consumes less power than LOP up to 1 MHz and 3 GHz for Flash and SRAM respectively. Depending on the required operation frequency range, the least power consuming memory can be chosen. For example, a Flash memory (LSTP) consumes less power than the remainder memories but just up to 50 KHz, then SRAM memory (LSTP) is preferred up to 5 GHz, frequency from which SRAM memory (LOP) is preferred.

3.3 SRAM with sleep state

The technique presented in Section 3.2.2, in which the supply voltage is scaled down, can be used to reduce the static power when the SRAM memory is idle. In

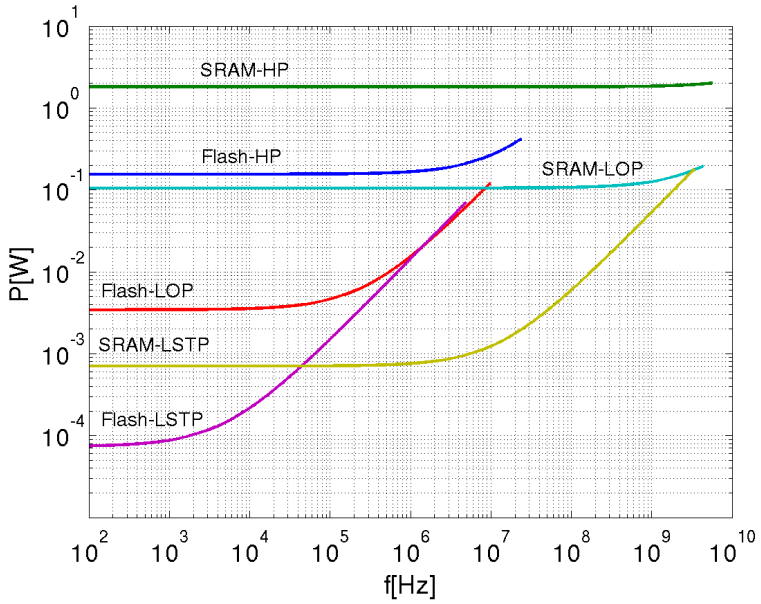

Figure 3.9: Flash and SRAM power consumption as a function of the operating frequency.

the case of the SRAM, the whole memory cell array is put in sleep state achieving important static power reduction. On the contrary, in Flash only would be reasonable applied to peripheral circuit, since the leakage of memory cells is no significant. As a result, in the case of Flash memory only marginal savings are achieved.

The power saving potential of a SRAM memory with sleep state depends on the memory parameters and the workload. The power consumed in the sleep state is a fraction of the ready power, since we suppose that a technique based on reducing the supply voltage is used to exponentially reduce the leakage. We assume a reduction factor of the leakage in sleep state of 0.1, which is a common value adopted in the literature [73, 82, 93].

Before a memory bank could be successfully accessed, the memory cells need to go back from the data retention voltage to the standby voltage, which involves the loading of internal capacitances. Since the path of the currents in this process is similar to the path of the leakage currents, the wake-up energy is in a close relation to the leakage energy. Nemeth et al. [82] computed the activation energy and the leakage energy drain in one cycle, to find the proportionality constant. In the literature it is reported that the proportionality constant respect to active energy varies from about one [17] to hundreds [73].

The average power of a SRAM memory with sleep state can be expressed as,

$$P_{read} = E_{dyn}\hat{f}d + P_{sta}d + P_{slp}(1-d) + E_{wkp}r_{wkp} \tag{3.2}$$

where P_{sta} and E_{dyn} are as defined in Eq. (3.1), and $\hat{f}$ is the new operating frequency. Additionally, P_{slp} is the leakage power in sleep state, d is the average

duty-cycle (i.e. the fraction of time the memory is read), r_{wkp} the average wake-up rate (i.e. the inverse of the average period), and E_{wkp} the wake-up energy.

The dynamic power remains unchanged, only that $\hat{f}$ is $1/d$ times faster than f, that is, $\hat{f} = f/d$ with $d < 1$. In other words, the increased frequency leads to a duty-cycle less than 100%, allowing the memory to enter in sleep state. Note that we are considering a greedy policy for manage the memory, so that the memory enters in sleep state whenever it is not active.

The last equation can be reformulated as,

$$P_{read} = P^*_{sta} + E_{dyn} f, \tag{3.3}$$

where P^*_{sta} is an equivalent static power defined as

$$P^*_{sta} = P_{sta} d + P_{slp}(1 - d) + E_{wkp} r_{wkp}. \tag{3.4}$$

This value depends on the following memory parameters: static power of the ready and sleep state, and the wake-up energy. Also depends on the workload, through duty-cycle d, and the wake-up rate r_{wkp}. The average power due to wake-up transition from sleep to ready, $P_{wkp} = E_{wkp} r_{wkp}$, depends on the memory (E_{wkp}) and its usage (r_{wkp}).

Eq. (3.4) can be rewritten as:

$$P^*_{sta} = (P_{sta} - P_{slp} + \frac{E_{wkp}}{\tau}) d + P_{slp} \tag{3.5}$$

where τ the time the memory is in ready state, during which the memory is read. We define $T_{wkp} = 1/r_{wkp}$ the average time between wake-ups, so that $r_{wkp} = d/\tau$. Then, if it holds

$$\tau >> \frac{E_{wkp}}{P_{sta}} = \frac{E_{wkp}}{E_{sta}} T_{wkp}, \tag{3.6}$$

the wake-up contribution can be neglected.

Considering that the wake-up to static energy ratio ranges from one to hundreds, we have that the wake-up contribution to the equivalent static power can be disregarded, if the sleep time is tens to thousands clock cycles (applying the 10x rule of thumb to neglect a term). It was found that in average the sensor node in average is in sleep state about ten milliseconds. The curves in Fig. 3.9 can be recomputed considering the P^*_{sta} and find the range for which SRAM consumes less. Let's consider a duty-cycle of 1%, a leakage reduction power of 10x for the sleep state respect to the normal leakage.

As shown in Fig. 3.10 a significant reduction in average power is obtained, corresponding to the equivalent static power. For high frequencies the dynamic energy is more significant and the average power reaches the levels corresponding to the memory without power management. For frequencies up to 300 MHz the more energy-efficient memory is the SRAM-LSTP with power management.

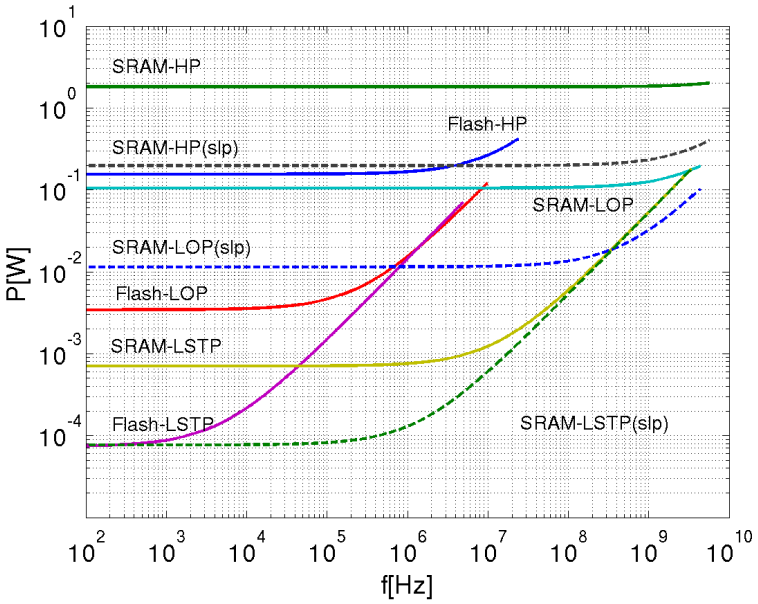

Figure 3.10: Flash, SRAM and SRAM with sleep state power consumption as a function of the operating frequency.

3.4 SRAM with power-manageable banks

The SRAM with sleep state, presented in the previous section, minimize the average power consumption taking advantage of the system idleness, by putting the whole memory in a low leakage state to reduce the drain of static power.

A SRAM memory is usually partitioned in banks to reduce the dynamic power. As stated previously, in Section 3.2.2, a memory bank is a fully-functional unit, and only one bank is active at a time. Since only one bank is active per access, the remaining idle banks could be put into the sleep state to save power. Partitioning a memory in banks, incurs in energy overhead that needs to be considered.

The SRAM memory with power-manageable banks adds to each bank a sleep circuit to put the bank in low-leakage state. These sleep circuits are controlled by the power manager.

3.4.1 Memory energy model

In this section we model the energy consumption of a SRAM memory with power-manageable banks. First, we revisit the model derived in the previous section for a monolithic SRAM memory with a sleep state, but in this case we derive an expression as a function on the respective energy per cycle. Next, the dependence of the energy on the memory size is modeled. Then, we derive an expression of the energy consumption for this memory without considering the partition overhead. Finally, the partition overhead is included and the effective energy consumption expression found.

Basic energy model

Up to this point, the power consumption has been modeled considering the power as the sum of the static power and the dynamic power, the latter computed as the dynamic energy multiplied by the operating frequency ($P_{dyn} = E_{dyn}f$). If a sleep state was available, and a duty-cycle defined, the average power was computed. This simple model served to study the power consumption dependency on frequency.

Henceforth, it assumed the system (processor and memory) operates at a fixed frequency. Therefore, memory consumption is conveniently modeled in terms of energy per cycle, as follows.

The static power consumed by a memory depends on its actual state: *ready* or *sleep*. During the ready state read cycles can be performed, but not in the sleep state. Since the memory remains in one of these states for a certain number of cycles, the static energy consumed can be expressed in terms of energy per cycle (E_{rdy} and E_{slp}) and the number of cycles in each state. Each memory access, performed during the ready state, consumes a certain amount of energy (E_{acc}). The ready period during which memory is accessed is usually called the active period, and the total energy spent corresponds to the sum of the access and the ready energy ($E_{act} = E_{acc} + E_{rdy}$), i.e., the dynamic and static energy. On the other hand, the ready cycles without access are called idle cycles, consuming only static energy ($E_{idl} = E_{rdy}$). Each state transition from sleep to active (i.e. the wake-up transition) has an associated energy cost (E_{wkp}) and a latency penalty, considered in Section 4.3.3.

Based on the parameters defined above, the total energy consumption of a memory, when n cycles have elapsed, can be defined as

$$E = E_{act}n_{act} + E_{idl}n_{idl} + E_{slp}n_{slp} + E_{wkp}n_{wkp}, \tag{3.7}$$

where n_{act}, n_{idl} and n_{slp} are the number of the cycles in which the memory is in active, idle and in sleep state respectively, and n_{wkp} is the number of times the memory switches from sleep to active state. We define the ratios $r_k = n_k/n$ for $k \in \{act, idl, slp, wkp\}$ where n is the elapsed number of cycles. So the average energy per cycle is

$$\bar{E} = E_{act}r_{act} + E_{idl}r_{idl} + E_{slp}r_{slp} + E_{wkp}r_{wkp}. \tag{3.8}$$

Fig. 3.11 depicts the power consumption profile of a memory, based on this model in which it is considered the average power (energy) per cycle. For illustrating purposes the static power is selected as reference, and the remainder terms are shown as a function of the static level in logarithmic scale. Fig. 3.11 (left side) shows in blue the static power depending on the state: sleep in light blue, and ready in dark blue. The access energy is represented in red, and the wake-up energy in dark red each time there is a transition from sleep to ready state. Fig. 3.11 (right side) shows the same profile, in which access energy is combined with the ready energy to constitute the active energy, shown in green, and the ready cycles without access are identified as idle cycles.

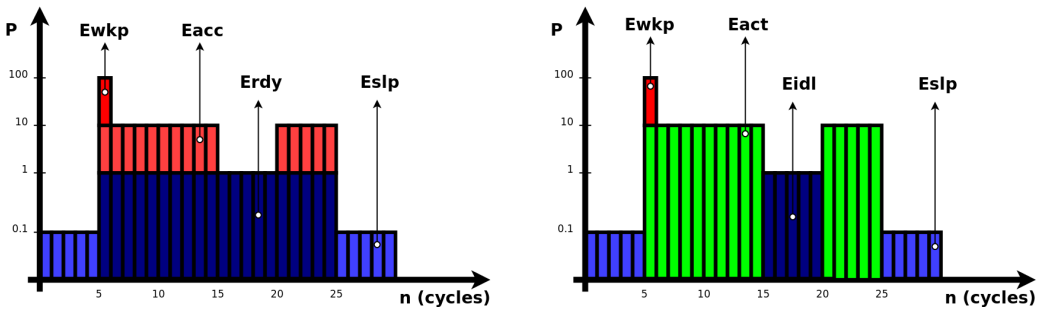

Figure 3.11: Energy consumption model.

Energy variation with memory size

The basis of memory banking is that the energy increases with the size of the memory. Consequently, if a single bank memory is partitioned in several banks, each bank is expected to reduce its energy consumption. In order to evaluate the energy saving appropriately when a banked memory is adopted, we next model the energy consumption of a memory as a function of its size. The energy values in Eq. (3.8), E_{act}, E_{idl}, E_{slp} and E_{wkp}, are generally considered simply proportional to the size of the memory [47]. We investigated the dependence of the involved parameters on the memory size, mainly using an estimation tool, CACTI [105]. However, not all parameters can be obtained from CACTI.

Table 3.4 lists the energy parameters and the used method to find the respective values. CACTI outputs the dynamic energy and the leakage power. The former value corresponds to the access energy of our model (E_{acc}). The leakage power was used to calculate the leakage energy per cycle, i.e., the idle energy of our model (E_{idl}), at the lowest maximum operating frequency among all memory sizes. The active energy (E_{act}) was directly computed (dynamic plus leakage).

The remaining energy parameters are not calculated by CACTI, so these values were estimated.

Similarly to the case of the SRAM with sleep state, we assumed that a technique based on reducing the supply voltage was used to reduce the leakage. Therefore, the energy consumed per cycle in the sleep state (E_{slp}) was a fraction of the idle energy. We assume that the leakage power in sleep state is 0.1 the idle power [73, 93].

Based on the limit values found in the literature for the wake-up energy cost (one [17] and hundreds [73]), we considered three different values to asses the impact of the wake-up cost on the energy savings, the aforementioned values, i.e., one and one hundred, and an intermediate value of ten.

We followed the described method to get a set of parameters for each memory size, ranging from 512 B to 256 KB (in a sequence of power of two), in order to model the dependence of the memory energy with its size. The CACTI memory configuration was set for a pure RAM memory, one read/write port, 65 nm technology and a high performance ITRS transistor type. Each energy parameter was considered at a time and the values varying the memory size were fitted to a

Table 3.4: Energy parameters and method to find the respective values.

Parameter	Method
E_{acc}	read access: CACTI estimation
E_{idl}	leakage: CACTI estimation
E_{act}	direct calculation: $E_{acc} + E_{idl}$
E_{slp}	estimation, $E_{idl}/10$
E_{wkp}	estimation, $K \cdot E_{act}, K = \{1, 10, 100\}$

Table 3.5: Energy curve parameters as function of memory size.

Parameter	a (nJ/B)	b (adim.)	Model
E_{idl}	3.28×10^{-7}	1.09	linear function
E_{slp}	$E_{idl}/10$	same as E_{idl}	linear function
E_{acc}	7.95×10^{-5}	0.48	square root function
E_{act}	1.78×10^{-6}	0.96	linear function
E_{wkp}	$K \cdot E_{act}, K = \{1, 10, 100\}$	same as E_{act}	linear function

power function $\mathbf{E}(S) = aS^b$ (least-square method), where $\mathbf{E}(S)$ is the energy per cycle and S the memory size. The resulting fitting coefficients (the proportionality constant a and the exponent b) and the model adopted are presented in Table 3.5. Figure 3.12 shows all the energy estimated values and the fitted curves.

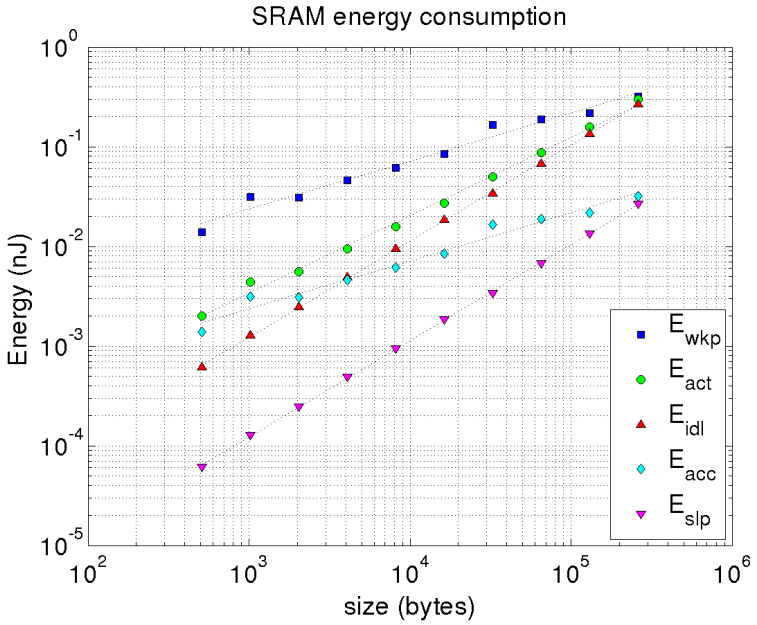

Figure 3.12: Energy consumption per cycle as a function of the memory size.

The dependence on the memory size of the energy parameters obtained directly from CACTI can be explained by examining the simulation output and analyzing

the relative contribution of each memory component [102]. The leakage energy in idle state grew nearly linearly, because the memory-cell leakage represented about 70% of the total energy and the number of memory-cells was directly proportional to the memory size. The access energy varied approximately as the square root of the size. It can be observed that between 70% and 80% of the dynamic energy come from bit-lines, sense amps, and other resource shared between memory-cells. The active energy (access plus idle, dynamic plus leakage), finally ended up varying almost linearly with size (exponent equal to one), because the leakage energy becomes more important than the dynamic energy with increasing size. However, for small footprints a exponent less than one or a polynomial model should be used.

Hereafter, for sake of simplicity, we will work based on simple models, that is the active, idle, sleep and wake-up energy are proportional to the memory size:

$$\mathbf{E}_k(S) = a_k S \tag{3.9}$$

for $k \in \{act, idl, slp, wkp\}$, where S is the memory size in bytes, and a_k is the corresponding constant of proportionality in Table 3.5.

Energy consumption expression

Using Eq. (3.9) the energy consumption of a bank of size s in a banked memory of total size S can be modeled as

$$\mathbf{E}_k(s) = E_k \frac{s}{S}, \tag{3.10}$$

where $E_k = a_k S$ is the corresponding energy consumption per cycle of the whole memory.

Now, considering a banked memory of N equally sized banks Eq. (3.10) becomes

$$\mathbf{E}_k\left(\frac{S}{N}\right) = \frac{E_k}{N}. \tag{3.11}$$

The total energy consumption per cycle of the whole banked memory can be found adding each bank energy contribution,

$$\bar{E}_N = \sum_{i=1}^{N} \left(\frac{E_{act}}{N} r_{act_i} + \frac{E_{idl}}{N} r_{idl_i} + \frac{E_{slp}}{N} r_{slp_i} + \frac{E_{wkp}}{N} r_{wkp_i} \right), \tag{3.12}$$

where the first three terms of the sum represent the active, idle and sleep energy as a function of the fraction of active, idle and sleep cycles respectively performed by each bank i. The last term of the sum represents the wake-up energy as a function of the average wake-up rate of each memory bank, that is, the average number of cycles elapsed between two consecutive bank transitions from sleep to active (for example, one transition in 1000 cycles).

The Eq. (3.12) can alternatively rewritten as

$$\bar{E}_N = \frac{E_{act}}{N} \sum_{i=1}^{N} r_{act_i} + \frac{E_{idl}}{N} \sum_{i=1}^{N} r_{idl_i} + \frac{E_{slp}}{N} \sum_{i=1}^{N} r_{slp_i} + \frac{E_{wkp}}{N} \sum_{i=1}^{N} r_{wkp_i}, \tag{3.13}$$

where each term represents the energy of each kind of contribution, i.e., active, idle, sleep and wake-up.

Let's focus on the sleep energy contribution together, particularly in the summation. The total number of cycles is $n = n_{act_i} + n_{idl_i} + n_{slp_i}$ for all banks, so $r_{act_i} + r_{idl_i} + r_{slp_i} = 1$. Then

$$\sum_{i=1}^{N} r_{slp_i} = \sum_{i=1}^{N} 1 - r_{act_i} - r_{idl_i} \tag{3.14}$$

And since there is only one bank active per cycle, the sum of the active cycles for all banks is the total number of cycles, in other words

$$\sum_{i=1}^{N} r_{act_i} = 1 \tag{3.15}$$

so Eq. (3.14) simplifies to

$$\sum_{i=1}^{N} r_{slp_i} = N - 1 - \sum_{i=1}^{N} r_{idl_i} \tag{3.16}$$

Finally, substituting Eq. (3.16) in Eq. (3.13), the total energy consumption per cycle is

$$\bar{E}_N = \frac{E_{act}}{N} + \frac{N-1}{N} E_{slp} + \frac{E_{idl} - E_{slp}}{N} \sum_{i=1}^{N} r_{idl_i} + \frac{E_{wkp}}{N} \sum_{i=1}^{N} r_{wkp_i}. \tag{3.17}$$

A preliminary analysis of the Eq. (3.17) shows that, in a memory with power-manageable banks, the active energy consumption is reduced by N relative to the monolithic memory. This is due to the fact that only one bank, which is one N^{th} the size of whole memory, is active at a time. The remaining $N - 1$ banks are in sleep state, consuming a fraction $(N - 1)/N$ of the total sleep energy. The third term corresponds to the idle energy and the last term to the wake-up energy, and both depend on the respective accumulated cycles ratio among the banks.

Effective energy consumption

The partitioning overhead must be considered to determine the effective energy consumption. A previous work had characterized the partitioning overhead as a function of the number of banks for a partitioned memory of arbitrary sizes [73]. The hardware overhead was due to an additional decoder (to translate addresses and control signals into the multiple control and address signals), and the wiring to connect the decoder to the banks [11]. As the number of memory banks increases, the complexity of the decoder was roughly constant, but the wiring overhead increases [73].

Table 3.6 reproduces the data presented in the aforementioned work, where the percentages are relating to the active energy. Fig. 3.13 shows graphically the

Table 3.6: Partition overhead as function of the number of banks (values extracted from [73]).

Number of banks	2	3	4	5
Overhead	3.5%	5.6%	7.3%	9%

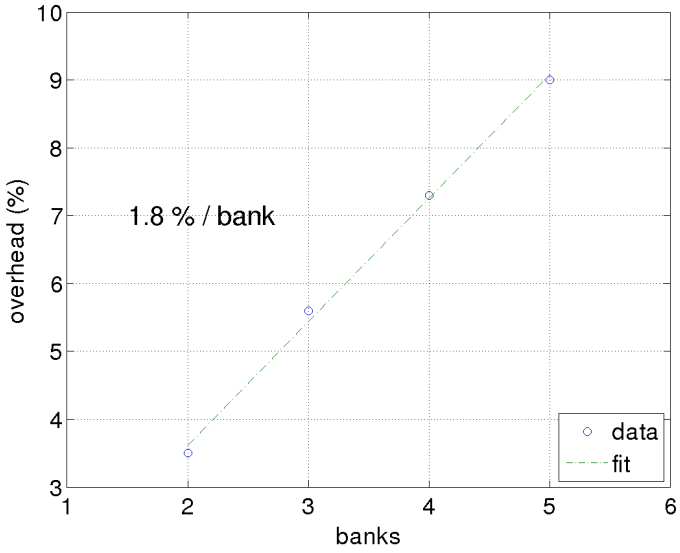

Figure 3.13: Partition overhead.

partition overhead as a function of the bank partitions. It can be clearly seen that the partition overhead is proportional to the active energy of an equivalent monolithic memory and roughly linear in the number of banks. The resulting overhead factor is approximately 1.8% per bank (also shown in the Figure).

Consequently, the relative overhead energy can be modeled as:

$$E_N^{ovhd} = k_{ovhd} N E_{act}. \tag{3.18}$$

In this work, the memory is partitioned into equally-sized banks. As a result the overhead is expected to decrease, leading to a lower value for the overhead factor. In order to asses the impact of the partition overhead, we keep the overhead factor as a design parameter.

Finally, the energy consumption of the banked memory, considering the partition overhead is

$$\bar{E}_N = \frac{E_{act}}{N} + \frac{N-1}{N} E_{slp} + \frac{E_{idl} - E_{slp}}{N} \sum_{i=1}^{N} r_{idl_i} + \frac{E_{wkp}}{N} \sum_{i=1}^{N} r_{wkp_i} + k_{ovhd} N E_{act}. \tag{3.19}$$

The energy consumption is expected to decrease as more banks are added, however the partition overhead grows with the number of banks. Presumably, there exist an optimum number of banks for which the energy consumption is minimized.

This model is used in the next chapter to determine the energy savings achieved using a banked memory with power-manageable banks, when compared to a monolithic memory. Also, the optimum number of banks is determined as a function of the memory parameters and the workload.

3.5 Summary and conclusions

Programmable processors relay in memories to hold the program instructions and temporal data. These memories are responsible of a significant part of the power budget in sensor nodes. Energy-efficient memory design represent an opportunity to further reduce the energy consumption in sensor nodes, enabling a broader adoption of the wireless sensor networks. In this chapter we reviewed existing options for memories in sensor networks.

The processor needs two kinds of memory, albeit could be the same technology, a read-write memory and non-volatile memory. Emerging non-volatile technologies, such as FRAM, are an interesting option for applications that need to write non-volatile data intensively. However, they not present any clear advantage as a direct substitution of Flash memory for code memory. In embedded system, NOR-Flash memory is typically used as non-volatile code memory, and SRAM as temporal data memory. Despite the code can be executed directly from non-volatile memory (NOR Flash), sometimes it is first copied from a non-volatile storage (NAND Flash) to SRAM at boot time, and later executed from SRAM. Usually, the rationale behind this choice is to achieve higher performance. Additionally, SRAM consumes less power than Flash for increasing frequency. On the contrary, Flash is more efficient at lower frequencies, since static power dominates over dynamic power, and Flash benefits from its low-leakage memory cell. However, a SRAM memory with a sleep state could be more energy efficient for a wide range of frequency. This memory reduces the static power by putting the whole memory in a low-leakage state when it is idle. This simple technique is very profitable in low duty-cycle system with relative long idle periods.

Additionally, a SRAM banked memory has the attribute that only one bank is active per access. SRAM with power-manageable banks can put the idle banks into the sleep state, reducing the leakage power even in always-on operation (100% duty-cycle). The bank energy is reduced by the number of banks, compared to the energy of the equivalent monolithic memory. On the other hand, before a bank could be accessed, it must be reactivated with an associated energy cost. Furthermore, partitioning the memory in independently manageable banks has an energy overhead. Consequently, the number of bank should be carefully selected.

The event-driven nature of sensor nodes is highly exploited by this last memory architecture. In the next chapter this memory is further analyzed, considering different power management strategies.

On the other hand, optimized memory hierarchies previously proposed, as those including cache and scratch-pad memories, are not well-suited for wireless sensor network applications. This is because the nodes normally don't have a compute-intensive kernel, and the access to memory are more evenly spread. The more

refined proposed architecture could be used as a single memory or adopted in any other hierarchies, in wireless sensor networks or in other event-driven application context.

Chapter 4

Banked memory for sensor nodes

The fundamental idea behind a memory with power-manageable banks is that only one bank is active per access, and the remaining idle banks can be put into the sleep state. The memory architecture relies on that banks can be individually managed to control its power state, so that they are able to enter in a low-leakage sleep state to reduce the static power. However, the transition from sleep state to the ready state has an energy cost by itself. As a consequence, the leakage saving on the sleep state should compensate this bank wake-up cost. The overall wake-up energy cost considering all banks can be minimized by properly allocating the program code into the different banks. Highly correlated memory blocks allocated to the same bank leads to a bank access pattern with a high temporal locality, thus reducing the number of wake-ups. Hence, the total wake-up fraction across the banks is very low and the energy saving maximized. On the other hand, the partitioning overhead due to sleep transistors, extra wiring and duplication of address and control logic limit the number of banks that effectively reduce the energy consumption. So that, the partition overhead must be taken into account to find the optimum number of banks.

In this chapter it is explored in depth the memory introduced last in the previous chapter, and its potential in the context of wireless sensor networks as duty-cycled systems. The first section presents the proposed architecture and methodology to greatly profit the memory structure and achieve huge energy saving. Different power management strategies options are described, and greedy and best-oracle are selected for evaluation. The second section reviews related work and highlights the main differences with the present work. The third section derive expressions for the energy saving considering both power management policies, in the case the memory is continuously accessed, that is, there is always an active bank. The next section considers a memory with power-manageable banks which it is also duty-cycled as a unit.

We conclude the chapter with a summary, including a brief discussion.

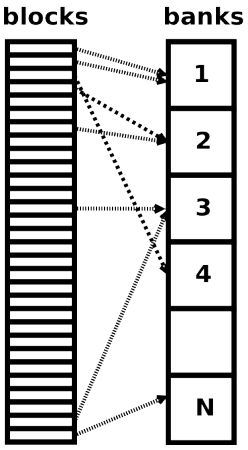

Figure 4.1: Block Allocation to banks.

4.1 Proposed architecture and methodology

A SRAM memory with power-manageable banks is composed by a memory cell array divided in banks, each of which has a sleep circuit to put the bank in low-leakage state, and a power manager that controls the bank states.

The memory consumption is given by the Eq. (3.19) derived in Section 3.4, reproduced here for ease of reference,

$$\bar{E}_N = \frac{E_{act}}{N} + \frac{N-1}{N} E_{slp} + \frac{E_{idl} - E_{slp}}{N} \sum_{i=1}^{N} r_{idl_i} + \frac{E_{wkp}}{N} \sum_{i=1}^{N} r_{wkp_i} + k_{ovhd} N E_{act}. \quad (4.1)$$

The energy consumption depends on energy memory parameters, number of banks, resulting idle cycles ratio and wake-up rate, and partition overhead. Given the characteristic parameters of the memory, including the partition overhead, the user should choose the number of banks to finally allocate the memory content into the banks. The memory content must be distributed among the banks with the objective of minimizing the energy consumption. For that, the memory content is divided in blocks, the allocation unit, which are mapped to banks using a specified method. Fig. 4.1 depict the procedure in which the code blocks are allocated to memory banks.

The proposed methodology for distributing the code to banks is shown in Fig. 4.2. The method is based on using a memory access trace to solve an optimization problem. The hypothesis behind this methodology is that execution workload has the same statistical properties than the workload used during the optimization phase. The application binary program is divided in memory blocks, e.g. basic blocks or any other arbitrarily defined. Then, the memory access trace is obtained by simulation or execution of the program to get the access trace to the defined blocks. The access pattern to blocks and a memory configuration are input to the optimization solver that outputs the block-to-bank mapping that minimizes the energy consumption of the banked memory. The optimization also outputs the activation signals that control when a bank is sleeping or ready to be accessed.

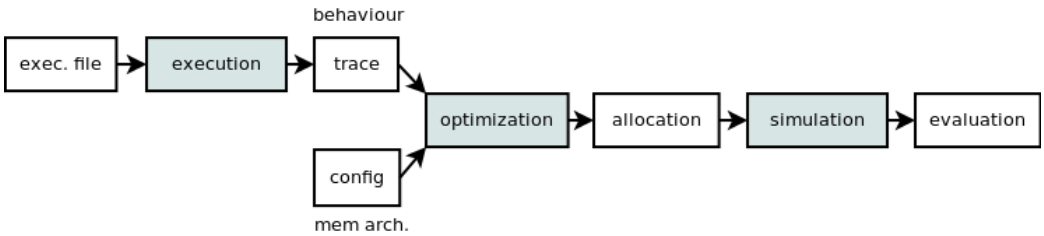

Figure 4.2: Design flow.

The memory configuration specifies the number of banks, the memory energy parameters, and the power management strategy used to control the banks states. Next, the energy saving is obtained by simulating the application execution from the banked memory, in which the original program code was reallocated among the banks.

The process may be repeated for different numbers of banks to find the optimum number by simply comparing the obtained energy saving in each step. In addition, at design time the power management strategy could be modified to asses its impact on the energy saving.

The user may configure the memory using the procedure described above; or conversely, he or she could set the number of banks based on an estimation from the memory parameters only. The estimated number of banks may result in a suboptimal solution, however it can be used to narrow the search space to its neighborhood. Nevertheless, the memory content needs to be allocated to banks using the block-to-bank map provided by the optimization solver. The binary of the application, specified as a relocatable object file (e.g. elf format file), is modified using a patching tool (as proposed by [3] and [77,111]) so that the blocks are rearranged to the banks.

Before applying the patch tool, the block-to-bank map must be completed, i.e. all banks have a corresponding bank. It can be possible that the trace used to determine map, do not cover the 100% of the executable. Thus, the remaining blocks must be allocated to available banks using a complementary algorithm based on some heuristic.

4.1.1 Memory power management

The memory bank states (sleep or ready) are defined using a given power management strategy. This strategy defines when a bank is put in sleep state and when it is woken up, and therefore includes the information if a bank remains in idle state even if it is not accessed. The basis of the chosen strategy may range from a very simple one to highly sophisticated prediction algorithms [18]. The energy savings may depend much on the adopted strategy. Taking this into account, we consider two power management strategies: a simple greedy policy and the best-oracle policy, as defined in 2.3.2. In the greedy strategy, as soon as a memory bank is not being accessed it is put into sleep state. The greedy policy is one of the simplest possible management schemes. Conversely, an best-oracle policy is based on the

best prediction algorithm in the sense that follows. The optimization takes into account the whole access trace, including information of future access, to obtain the schedule of bank states that maximize the energy savings. In this way, we are able to assess the energy saving using two algorithms in opposite ends in terms of complexity.

Subsequently, the power management module must implement the algorithm in hardware. At run time it must manage the bank states in accordance with the access patterns to the banks. The implementation of the greedy power management is straightforward. As soon as a new bank is accessed, it is woken up and the previously active bank is put in sleep state. The performance of the best-oracle policy that was obtained at the optimization stage can not be achieved at run time, since there is no practical prediction algorithm that can beat the off-line optimum oracle strategy. As a consequence, the energy saving obtained by the best-oracle policy represents the maximum achievable savings using any power management.

The power management may be implemented locally, in case of greedy or time-out policy, or centrally, if global and historical information should be considered to decide which and when is wake-up or put in sleep state. In any case a stall signal is needed to stop the processor until the corresponding bank is in ready state. The control for greedy policy is straightforward, while for time-out policy a monostable or a chain of flip-flops can be used to hold the needed clock cycles until the bank is ready for accessing.

4.2 Related work

SRAM memory banking along with power management to put memory banks into a sleep mode is a well known technique. Farrahi et al. [41] initially presented the memory partitioning problem to exploit the sleep mode operation to minimize power consumption, showing that it is a NP-hard problem, and that some special classes are solvable in polynomial time. Results were obtained for a set of synthetic data, randomly generated with controlled parameters, and the effectiveness of the algorithm was assessed by comparing to a random partitioning algorithm.

This idea of reducing energy consumption by increasing memory elements idleness have been applied to scratch-pad and cache memories, in applications with high performance requirements (see Loghi et al. [73] for a brief survey).

Focusing on SRAM memory partitioning, Benini et al. [11] applied this technique to highly data-intensive application (e.g., digital filtering, transformations, stream processing), which contain few control conditions. They proposed a recursive bi-partitioning algorithm to solve the memory partitioning problem using simulated execution traces of a set of embedded applications. Golubeva et al. [47] continue this line of investigation, considering the availability of a low-leakage sleep state for each memory block in a scratch-pad memory. The partition algorithm proposed is based on a randomized search in the solution space. Finally, Loghi et al. [73] proposed an optimal partitioning algorithm based on an implicit enumeration of the partitioning solutions. They proved a theoretical property of the search space, exploited to reduce the number of partition boundaries to be enu-

merated, making exhaustive exploration feasible. A set of applications taken from the MiBench [48] application suite were used to get execution traces. These works consider splitting the address space into multiple, contiguous memory banks. Consequently, the partition algorithm is restricted to finding the optimal boundaries between the memory banks.

Ozturk and Kandemir [84] proposed a series of techniques starting with the relocation and merge of memory blocks of the address space into memory banks, relaxing the aforementioned restriction. They formulated each of these techniques as an ILP (integer linear programming) problem, and solved them using a commercial solver. They target also data-intensive embedded applications (e.g. multimedia processing: image and video), which manipulates multidimensional arrays (with affine subscript expressions) of data using a series of nested loops (with compile time known bounds). Therefore, a static compiler analysis is used to extract data-access patterns, which in turn are the input for finding the solution. The explored techniques include nonuniform bank sizes, data migration, data compression, and data replication.

All mentioned works report energy savings in terms of a relative percentage of some baseline, i.e., the consumption of an equivalent monolithic memory. However, the presented results not only depend on the proposed technique, the memory architecture or the particular case study, but also on the selected technology. Since different technologies were chosen in each work, the comparison between them is difficult.

We follow a methodology similar to the one employed in [84], detailed in the previous section, in which a memory access trace is used to solve an optimization problem for allocating the application memory divided in blocks to memory banks. Our work differs from the previous literature in three main aspects. First, we propose using banked memories with power management for code memory, in this case in WSN, exploiting the event-driven characteristics of this class of application [102]. We derive expressions for energy savings based on a detailed model that favors the analysis of the different factors determining the effective energy saving [102]. Second, we address the power management issue obtaining results for a greedy policy [104], one of the simplest possible management scheme, and a best-oracle policy, representing the best prediction algorithm. Finally, we assess the benefit of whether or not to adopt an advanced memory bank state management, as a function of different wake-up energy costs [103].

4.3 Energy saving expressions

In this section we derive expressions for the energy savings of a memory of equally sized banks with two different management schemes: greedy and best-oracle. Initially, the partition overhead is not considered, for the sake of simplicity. Next, this overhead is included to find an expression for the effective energy saving. Also, a formula for an estimated optimum number of banks is given, based only on memory constructive parameters: energy consumption values and partition overhead as a function of the number of banks. Finally, the energy saving is determined

in the case the memory is also duty-cycled completely, that is, all banks enter in sleep state.

4.3.1 Energy saving for the greedy policy

When a greedy strategy is used each bank is either in active or sleep state (there are no idle cycles), $r_{idl_i} = 0$, and from Eq. (4.1) (disregarding the partition overhead contribution) one obtains:

$$\bar{E}_N^{greedy} = \frac{E_{act}}{N} + \frac{N-1}{N} E_{slp} + \frac{E_{wkp}}{N} \sum_{i=1}^{N} r_{wkp_i}. \tag{4.2}$$

We define the energy savings of a banked memory as the relative deviation of the energy consumption of a single bank memory which is always active (i.e. $N = 1$, so that $\bar{E}_1 = E_{act}$)

$$\delta E = \frac{E_1 - \bar{E}_N}{E_1}. \tag{4.3}$$

Thus, the energy saving of a banked memory of N uniform banks is

$$\delta E_N^{greedy} = \frac{N-1}{N} \left(1 - \frac{E_{slp}}{E_{act}}\right) - \frac{1}{N} \frac{E_{wkp}}{E_{act}} \sum_{i=1}^{N} r_{wkp_i}. \tag{4.4}$$

The first term is related to active consumption reduction, coming from having $N-1$ banks in sleep state and only one bank in active state. The last term, which is related to the cost of wake-ups, depends on the accumulated wake-up rate among the banks. It is directly proportional to the wake-up to active energy ratio and inversely proportional to the number of banks.

In order to maximize the energy saving in a memory having N uniform banks, the optimization algorithm must minimize the accumulated wake-up rate. Note that the optimum content distribution among the banks does not depend on the wake-up cost, but rather the wake-up cost determines the final energy saving. However, the allocation of blocks to banks must consider the bank size constraint. Finally, the energy saving can be improved by increasing N and at the same time keeping the accumulated wake-up rate low.

The maximum achievable saving corresponds to the sleep to active rate, which is equivalent to have the whole memory in sleep state, but one bank in active state.

Even so, the partition overhead limits the maximum number of banks and reduces the achieved savings.

4.3.2 Energy saving for the best-oracle policy

Consider a memory with a power management, different from greedy, by means of which a bank may remain in idle state, even if it will not be immediately accessed.

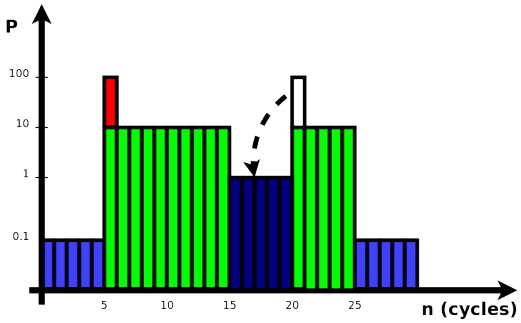

Figure 4.3: Energy comparison: stay in idle state vs. enter in sleep state (y-axis in logarithmic scale).

In a similar way to the greedy policy, the expression for the energy savings can be determined as:

$$\delta E_N^{oracle} = \frac{N-1}{N}\left(1-\frac{E_{slp}}{E_{act}}\right) - \frac{1}{N}\left(\frac{E_{idl}-E_{slp}}{E_{act}}\right)\sum_{i=1}^{N} r_{idl_i} - \\ -\frac{1}{N}\frac{E_{wkp}}{E_{act}}\sum_{i=1}^{N} r_{wkp_i}. \tag{4.5}$$

Compared to Eq. (4.4), Eq. (4.5) has an additional term, which is related to the energy increase caused by the idle cycles. This does not necessarily imply that the energy saving is reduced. On the contrary, if the banks are controlled by the best-oracle policy, the accumulated wake-up ratio decreases achieving higher energy savings. Fig. 4.3 depicts the situation in which the bank consumes less energy staying idle (dark blue), than entering in sleep state to later waking up (unfilled).

Note that the expression for the ideal power manager (best-oracle policy) and any other policy different from greedy is the same. The difference is that, in the case of best-oracle, the memory content allocation to banks was optimal at design-time, so that at run time the energy is minimized by optimally, again, controlling the bank states. Consequently, the idle cycles ratio and the wake-up rate resulting from best-oracle, leads to the lowest energy consumption.

4.3.3 Effective energy saving

As mentioned previously, the wake-up transition from sleep to active state of a bank memory has an associated latency. This latency forces the microprocessor to stall until the bank is ready. The microprocessor may remain idle for a few cycles each time a new bank is woken up, incrementing the energy drain. This extra microprocessor energy can be included in the bank wake-up energy and for simplicity we will not consider it explicitly. Moreover, if the wake-up rate is small and the active power of the microprocessor is much higher than idle power, this overhead can be neglected. Additionally, the extra time due to the wake-up transition is not an issue in low duty-cycle applications, since it simply increases the duty-cycle slightly.

The partitioning overhead must be included to assess the effective energy consumption. As was mentioned in Section 3.4.1, the energy increase is proportional to the active energy of an equivalent monolithic memory and it is approximately linear with the number of banks, resulting in an overhead factor of approximately 1.8% per bank.

Consequently, the relative overhead energy can be found dividing the energy overhead, Eq. (3.18), by the energy of an equivalent monolithic memory, obtaining

$$\delta E_N^{ovhd} = k_{ovhd} N. \tag{4.6}$$

The effective energy saving δE_N^{eff} is calculated subtracting Eq. (4.6) from Eq. (4.5). This expression applies for both policies, and the effective saving for the greedy strategy is obtained by setting r_{idl_i} equal to zero for all banks. The number of banks that maximize the effective energy saving can be determined from δE_N^{eff} finding its maximum.

δE_N^{eff} presents a maximum for

$$N_{opt} = \sqrt{\frac{1}{k_{ovhd}} \left[\left(1 - \frac{E_{slp}}{E_{act}}\right) + \left(\frac{E_{idl} - E_{slp}}{E_{act}}\right) \sum_{i=1}^{N} r_{idl_i} + \frac{E_{wkp}}{E_{act}} \sum_{i=1}^{N} r_{wkp_i} \right]}. \tag{4.7}$$

It can be seen that the optimal number of banks depends on the memory parameters and the resulting access pattern to the memory. The optimal number of banks, and also the effective energy saving achieved, grows with decreasing partition overhead factor. With regards to behavioral dependence, how effectively the memory content was allocated to banks at design time, and the performance of power management policy at run time, have influence in the last two terms multiplying the inverse of the overhead factor.

From the previous formula, an estimated value of the optimum number of banks can be found, based only on the memory constructive parameters (energy consumption values and partition overhead). Obviously, this estimation is general and independent of the application and run-time behavior. The estimated value agrees with the optimum number of banks, as long as the wake-up contribution is relatively small. Note that in the best-oracle policy, the idle energy cycles came in to reduce the overall energy consumption, so if the wake-up energy decrease, the idle energy also drops.

The maximum energy savings is defined for both cases for null wake-up contributions:

$$\delta E_N^{max} = \frac{N-1}{N} \left(1 - \frac{E_{slp}}{E_{act}}\right) - k_{ovhd} N. \tag{4.8}$$

Finally δE^{max} presents a maximum for

$$N_{opt}^{est} = \sqrt{\frac{1}{k_{ovhd}} \left(1 - \frac{E_{slp}}{E_{act}}\right)}, \tag{4.9}$$

obtaining the described estimation of the optimum number of banks (or near optimum number of banks).

4.4 Energy saving in a duty-cycled memory

Let us consider first a monolithic memory with sleep state, but unlike the case analyzed in Section 3.3, we include a power management different from greedy, so that the memory may remain idle without entering in sleep state.

The basic energy model derived in Section 3.4.1 is reproduced here for the sake of convenience,

$$\bar{E} = E_{act} r_{act} + E_{idl} r_{idl} + E_{slp} r_{slp} + E_{wkp} r_{wkp},$$

where the energy parameters and the cycles ratio refers to the whole memory.

Next, the active cycles ratio r_{act}, is replaced by the usual notation d for the duty-cycle ($d = r_{act}$), and since the memory is in one only state at a time, the following substitution is made, $r_{slp} = 1 - d - r_{idl}$. Finally we obtain the average energy consumption per cycle of a memory with sleep state.

$$\bar{E} = (E_{act} - E_{slp})d + E_{slp} + (E_{idl} - E_{slp})r_{idl} + E_{wkp} r_{wkp}. \tag{4.10}$$

Subsequently, based on the model of the memory with power-manageable banks, an expression for the energy consumption is derived.

The energy consumption per cycle is given by Eq. (3.13), and for easy reference we reproduce it here (partition overhead not included yet),

$$\bar{E}_N = \frac{E_{act}}{N} \sum_{i=1}^{N} r_{act_i} + \frac{E_{idl}}{N} \sum_{i=1}^{N} r_{idl_i} + \frac{E_{slp}}{N} \sum_{i=1}^{N} r_{slp_i} + \frac{E_{wkp}}{N} \sum_{i=1}^{N} r_{wkp_i}.$$

It holds that for any bank,

$$r_{act_i} + r_{idl_i} + r_{slp_i} = 1. \tag{4.11}$$

In other words each bank spends cycles in one of the possible states.

Besides, the duty-cycle, d, is

$$\sum_{i=1}^{N} r_{act_i} = d, \tag{4.12}$$

since one bank is active at a time, and the total active cycles corresponds to global duty-cycle.

The partition overhead increase the consumption only when the memory is active state,

$$E_N^{ovhd}(d) = k_{ovhd} N E_{act} d. \tag{4.13}$$

Using the definitions above we get

$$\bar{E}_N(d) = \frac{E_{act} - E_{slp}}{N} d + E_{slp} + \frac{E_{idl} - E_{slp}}{N} \sum_{i=1}^{N} r_{idl_i} + \frac{E_{wkp}}{N} \sum_{i=1}^{N} r_{wkp_i} + k_{ovhd} N E_{act} d. \tag{4.14}$$

Needless to say, for a memory with only one single manageable bank (i.e. $N = 1$), or for a duty-cycle of 100% (i.e. $d = 1$), we get the respective expressions previously found.

In order to easily compare a memory with power-manageable banks with a monolithic memory, we draw the following definitions. We say that the memory is in active state when any of its banks is active, and that it is in sleep state if all its banks are in sleep state. Similarly, the memory is idle if all its banks are idle. The idle cycles of a bank are discriminated in idle cycles during which another bank is active ($\hat{r}_{idl_i}$) hence the memory is active, and global idle cycles (r_{idl}) when the memory is idle. The same applies the wake-up rate, differentiating between bank wake-ups ($\hat{r}_{wkp_i}$) when the memory is active, form memory wake-ups (r_{wkp}). Then,

$$\sum_{i=1}^{N} r_{idl_i} = \sum_{i=1}^{N} r_{idl} + \hat{r}_{idl_i} = N r_{idl} + \sum_{i=1}^{N} \hat{r}_{idl_i} \tag{4.15}$$

and,

$$\sum_{i=1}^{N} r_{wkp_i} = \sum_{i=1}^{N} r_{wkp} + \hat{r}_{wkp_i} = r_{wkp} + \sum_{i=1}^{N} \hat{r}_{wkp_i}. \tag{4.16}$$

Note that, in the first equation, the global idle ratio is multiplied by N, since when there is global idle cycles, all banks are idle. In the case of wake-ups, the global wake-up ratio term is alone, since when takes place a global memory wake-up only the corresponding bank is waken up, and not the whole memory.

Substituting Eq. (4.15) and Eq. (4.16) in Eq. (4.14),

$$\begin{aligned} \bar{E}_N(d) &= \frac{E_{act} - E_{slp}}{N} d + E_{slp} + (E_{idl} - E_{slp}) r_{idl} + \frac{E_{wkp}}{N} r_{wkp} \\ &+ \frac{E_{idl} - E_{slp}}{N} \sum_{i=1}^{N} \hat{r}_{idl_i} + \frac{E_{wkp}}{N} \sum_{i=1}^{N} r_{wkp_i} + k_{ovhd} N E_{act} d. \end{aligned} \tag{4.17}$$

Comparing the energy consumption of a monolithic memory, Eq. (4.10), and a memory with power-manageable banks, Eq. (4.17), it can be appreciated the following differences:

$$\begin{aligned} E_{wkp} r_{wkp} &\rightarrow \frac{E_{wkp}}{N} r_{wkp} \\ (E_{act} - E_{slp}) d &\rightarrow (E_{act} - E_{slp}) \frac{d}{N} + \frac{E_{idl} - E_{slp}}{N} \sum_{i=1}^{N} \hat{r}_{idl_i} + \\ &+ \frac{E_{wkp}}{N} \sum_{i=1}^{N} \hat{r}_{wkp_i} + k_{ovhd} N E_{act} d \end{aligned} \tag{4.18}$$

Interestingly, the first difference states that the wake-up energy is reduced by N, since only one bank is waken up, instead of the whole memory. The second difference can be interpreted as the memory in active state benefits form individually managing banks, reducing the energy consumption. The previous findings

related to a memory continuously accessed, studied in the last section, are valid for the active state.

If the power management policy is greedy, for the whole memory and also for controlling individually the bank states, the energy consumption is

$$\bar{E}_N(d) = E_{act}\frac{d}{N} + E_{slp}\left(1 - \frac{d}{N}\right) + \frac{E_{wkp}}{N}\left(r_{wkp} + \sum_{i=1}^{N} r_{wkp_i}\right) + k_{ovhd}NE_{act}d \tag{4.19}$$

From the equation above it is even more straightforward to interpret that banking reinforce the saving of duty-cycling, having an equivalent effect of further reducing the duty-cycle by N. On the other hand, if the duty-cycle is low enough, the energy reduction might not be significant. The latter depends on the relative weight of the sleep energy relative to the active energy, and obviously, the wake-up energy contribution.

Note that, in this case, we had not drawn an expression for the energy saving relative to a single bank memory with sleep state. This is because it would only be enlighten in corner cases with no special interest or already studied.

4.5 Summary and conclusions

SRAM memory with power-manageable banks can reduce highly its energy consumption by putting idle banks in sleep state. The partition overhead increases as more banks area added, limiting the number of banks that effectively reduce the energy consumption. The optimum number of banks that minimize the energy saving can be estimated based on memory parameters only: partition overhead factor, active, sleep and wake-up energy. So that, the memory can be hard-wired configured at design time to meet the most common applications. However, this estimated number of banks may produce a suboptimal solution under certain conditions, such as relative high wake-up energy. The number of banks can be adjusted according to the particular application. The proposed methodology includes iteratively search the optimum partition to finally configure the number of banks.

Whether the number of banks is configured by the user or not, the memory content must be mapped to memory banks. For that, the memory content is divided in allocation units, named blocks, which are assigned to banks using an optimization algorithm. The optimization phase is based on the availability of a memory access trace to blocks, which is obtained by simulation. Since the simulation may not cover all the code address space, the unallocated blocks by the previous method need to be distributed along the banks. The block coverage may be incomplete simply because the program has unreachable blocks, or the test case was not appropriate (including not long enough trace).

The power manager that controls the bank states may vary from one based on the simplest policy, greedy, to an ideal power manager that rely in the best-oracle policy. Since, both policies are in opposite ends in terms of complexity and results, they are selected for consideration to asses its impact on the energy saving. All other policies fall between these limits.

The most relevant research studies that are based on the concepts behind banked memory with power management, were reviewed. However, it is virtually impossible to compare the proposals and results presented by the studied works, since they depends on the particular case study and the selected technology. The power management issue had been ignored, with no attempt to differentiate between various options. Although, an implicit power manager was evidently assumed, in each study. Ozturk and Kandemir [84] had used an ideal power manager, but we had to deduce this from the optimization problem formulation [1]. Their findings might have been more interesting if they had included a detailed model for deriving expressions for energy savings. This would have enable an analysis of the influence of different factors on the effective energy saving.

In this chapter we had presented a model and the corresponding energy consumption expressions for a greedy policy and best-oracle policy. We had found three crucial parameters: the sleep to active energy ratio, since determine the maximum energy savings; the wake-up to active energy ratio, since it might reduce the saving for poor temporal locality access patterns; and the partition overhead that limits adding extra banks for augmented savings. The best-oracle policy could keep a bank in idle state if entering in sleep state would not result beneficial. Thus, the energy saving expression differs in a term, the one related to the idle cycles ratio. Consequently, the idle cycles ratio and the wake-up rate resulting from best-oracle, leads to the lowest energy consumption.

In the case of greedy, the memory content allocation to banks was optimal during the optimization phase. Later, the banks are managed following the greedy policy. Conversely, in the best-oracle, in addition to allocate the block optimally in the blanks, the energy is minimized by optimally, again, controlling the bank states at run time.

Finally, the energy consumption for a duty-cycled operation was derived. The benefits of a banked memory that also puts the banks in sleep state while the memory is active were determined. The resulting energy consumption could be analyzed from two different perspectives. Firstly, as a memory originally managed as a unit (as seen in Section 3.3) that include putting idle banks in sleep state, even when the memory is active. Secondly, a banked memory with power-manageable banks (as seen in Section 3.4), that operates duty-cycling, so that, the previously energy savings is only profitable during the active period.

The extra saving will be significant if the energy consumption related to the active period is relative high respect to the sleep energy. In other words, a high duty-cycle operation tend to obtain higher benefits from individually managed memory banks.

[1] The difference between the formulation for an ideal policy and a greedy policy is in one expression, which changes from less than or equal to equal, as will be seen in the next chapter.

i

This page is intentionally left blank.

Chapter 5

Experimentation and results

An SRAM with power-manageable banks is expected to reduce considerably the static power of large memory cell arrays. In the previous chapter this memory was modeled to find expressions for energy consumption and energy savings, considering two opposite power management policies: best-oracle and greedy. The partition overhead was also included in analytical expressions, and an estimation of the optimal number of banks was obtained. However there are still some unanswered issues regarding technological and practical issues. One question that needs to be answered is whether applications code accesses to memory can be arranged in memory banks with enough temporal locality, so that the wake-up rate is kept low, and consequently, the saving is significant. Another open question is if it worths adopting a predictive power management, instead of rely in the straightforward greedy policy. In this chapter we present experiments comparing the predicted energy savings by our model with the energy savings obtained in practical applications. The first section is concerned with the evaluation methodology used in this chapter. It begins by examining alternatives for benchmark applications. Next, it is described the optimization problem that seeks to minimize the energy consumption by allocating the memory contents to memory banks properly. Then, it is briefly described the question of assigning the remaining unallocated blocks to the banks. Finally, the simulation tools for the evaluation are described. The second section presents the results and discusses the findings of this research, focusing on the following three key issues: power management policy, partition overhead and the accuracy of the estimated value, and energy saving in duty-cycle applications. The last section concludes the chapter with a summary and a critical discussion.

5.1 Evaluation methodology

This section describes the methodology used for the evaluation of the SRAM with power-manageable banks. First, we describe the applications selected and the criteria adopted for that. The block-to-bank map is almost completely built by solving an integer linear program. The problem formulation used for that, is described in detail afterwards. It can happen that the obtained map in the previous

Table 5.1: Application parameters (size in bytes).

Application	text	bss	data	#func.	avg. size
MultihopOscilloscope	32058	122	3534	1081	29.6
rpl-collect (udp-sender)	47552	232	9250	489	97.2

phase is incomplete, i.e., some blocks haven't been assigned to any bank. An approach to solve this problem is addressed later. Finally, the simulation tools are listed.

5.1.1 Benchmarks and applications

The benchmarks previously used for memory energy comparison, such as MiBench [48], are not adecuate for evaluating our proposal. In most cases, each benchmark is an algorithm implementation compiled as an independent application. The application, executed in batches, usually reads data inputs for the algorithm from files and outputs the processing results to the console or to a file. The sensor nodes has a limited amount of memory, and usually do not have a file system, preventing using Mibench in sensor nodes as is. But, the main limitation is that they do not capture the external event timing [7] needed to evaluate the memory regions idling.

Lacking available and general accepted benchmarks for our purposes, the idea was to select some representative applications of typical wireless sensor networks. We based our evaluation using the same applications as those employed for platform energy characterization : MultihopOscilloscope (TinyOS) and rpl-collect (ContikiOS). They are data-collection applications, in which each node of the network periodically samples a sensor and the readings are transmitted to a sink node using a network collection protocol (please refer to Section 2.4.2 for details).

The applications were compiled for a telos node. Table 5.1 summarizes the sections sizes, the number of functions and the function average size of the selected applications. It can be found that in both cases the code memory (text segment) is between five and nine times larger than the data memory (bss plus data segment). This relationship, which is typical in current wireless sensor networks applications, supports our preference of code memory for using a memory with power-manageable banks. As was argued in the motivation section (Section 1.1), code memories are increasingly large, so it is expected that this relationship is maintained or even grow. Particular applications that demand also large data memories, as signal processing, may embrace this in addition to other orthogonal techniques.

5.1.2 Optimization problem: memory contents allocation to banks

Next, we define an integer linear program that minimizes the energy consumption of a banked memory with power management by optimally distributing the application code divided in blocks to memory banks. Later, the parameter selection for the evaluation are described.

Problem formulation

The memory has N memory banks $B = \{1, \ldots, N\}$, of equal size $s_b, b \in B$. The application code is divided in M memory blocks $D = \{1, \ldots, M\}$ of size $s_d, d \in D$. We are further given an access pattern to these blocks over time by a_{dt}. A value of $a_{dt} = 1$ indicates that block d is accessed at time t. We want to determine an allocation of blocks to banks that respects the size constraints, and an activation schedule of the banks that minimizes total energy consumption, and such that banks that are accessed at time t are ready at time t. Let $l_{db} \in \{0, 1\}$ indicate that block d is allocated to bank b, and $o_{bt} \in \{0, 1\}$ that bank b is ready at time t. We define auxiliary indicator variables $a_{bt} \in \{0, 1\}$ representing the access of bank b at time t, $o^+_{bt} \in \{0, 1\}$ representing the wake-up transition of bank b at time t [1]. Let further $T = \{1, \ldots, t\}$ be set of access times. We assume that time 0 represents the initial state where all banks are in sleep state. For a given number of banks, the partition overhead is fixed, hence the problem formulation does not need to include this term.

Now, we can model the problem of finding the allocation and power management strategy by the following integer program:

$$\text{minimize} \sum_{\substack{t \in T \\ b \in B}} E_{acc} a_{bt} + E_{rdy} o_{bt} + E_{wkp} o^+_{bt} \tag{5.1}$$

subject to

$$o^+_{bt} \geq o_{bt} - o_{b,t-1} \qquad \forall b \in B, t \in T \tag{5.2}$$

$$o_{bt} \geq a_{bt} \qquad \forall b \in B, t \in T \tag{5.3}$$

$$a_{bt} = \sum_{d \in D} l_{bd} a_{dt} \qquad \forall b \in B, t \in T \tag{5.4}$$

$$\sum_{b \in B} l_{db} = 1 \qquad \forall d \in D \tag{5.5}$$

$$\sum_{d \in D} l_{db} s_d \leq s_b \qquad \forall b \in B \tag{5.6}$$

$$o_{b0} = 0 \qquad \forall b \in B \tag{5.7}$$

$$l_{db} \in \{0, 1\} \qquad d \in D, b \in B \tag{5.8}$$

$$o_{bt} \in \{0, 1\} \qquad b \in B, t \in T \cup \{0\} \tag{5.9}$$

$$o^+_{bt}, a_{bt} \in \{0, 1\} \qquad b \in B, t \in T. \tag{5.10}$$

[1] Note that the formulation models access and ready state separately, so that there is no explicit active states.

Eq. (5.2) defines wake-up transitions: if some bank is ready at time t, but has not been ready at time $t-1$, a wake-up transition occurred.[2] Eq. (5.3) and (5.4) define the access pattern for a given allocation. Restriction (5.5) guarantees that every block has been allocated to exactly one memory bank, and restriction (5.6) limits the total size of the allocated blocks to the size of the bank.

The above formulation corresponds to the best-oracle strategy, since it does not limit the activation schedules. For a greedy power management the constraint given by Eq. (5.3) can be modified, so that a bank is ready only when it is accessed.

$$o_{bt} = a_{bt} \qquad \forall b \in B, t \in T. \tag{5.11}$$

The formulation was described using AMPL (A Mathematical Programming Language) , and the corresponding models are listed in Appendix B.1.

Initially, the ILP solver used was *glpsol*, included in GLPK (GNU Linear Programming Kit) [3]. But, due to memory limitations later *CPLEX* optimizer by IBM® was adopted [4] The solver was executed in a PC and computation time for solving a single problem ranged form minutes to several hours or days, depending on the power management (oracle takes much longer time than greedy to find the solution) and the number of banks.

Parameters selection

The size of the blocks could be chosen regular (equally sized) or irregular, ranging from the minimum basic blocks to arbitrary size. For the sake of simplicity, the block set was selected as those defined by the program functions and the compiler generated global symbols (user and library functions, plus those created by the compiler). This facilitate the modification of the application binary, by means of a patching tool [3, 77]. Additionally, the conversion from address trace to block trace is straightforward.

The size of the blocks ranges from tens to hundreds of bytes (see Table 5.1), in accordance with the general guideline of writing short functions, considering the run-to-completion characteristic of TinyOS and any non-preemptive event-driven software architecture.

The energy parameters for the memory are the following. The active, idle and sleep values are those shown in Table 3.5 (Section 3.4.1), where the sleep state energy is a factor of 0.1 the idle energy. Three different values of wake-up energy cost were considered: 1, 10 and 100 times the access energy.

The problem of allocating the code to equally sized banks was solved for up six banks, for both power management strategies.

The total memory size was considered 10% larger than the application size, to ensure the feasibility of the solution. For each experiment the bank memory access

[2]Since the variables involved in the inequalities are binary, $a \geq b$ corresponds to the logical implication, $a \Rightarrow b$.

[3]http://www.gnu.org/software/glpk/

[4]Since, we had already modeled the problem using AMPL, gplsol itself was used to convert the problem formulation to the format used by cplex.

patterns a_{bt} have been determined using the trace a_{dt} and the allocation map l_{bd} (how block are allocated to banks), given by the corresponding solution. For the best-oracle power management the solution also outputs o_{bt}, the bank states for each cycle (i.e. ready or sleep). Finally, the average energy consumption is calculated using the memory energy parameters and the energy saving is determined comparing with a single bank memory with no power management.

5.1.3 Simulation and evaluation tools

We simulated the network using COOJA to get an execution trace. We followed the same methodology as before (Section 2.4.2) for getting a memory block access trace.

For the experiments we set up an unique scenario based on a configuration consisting of a network composed of 25 nodes. The simulation time varied form about 10 to 60 seconds, corresponding to roughly 300k and 2500k executed instructions respectively. Note that the node frequently enters in sleep mode, during that time no instruction are executed.

During the course of this investigation, we had used different trace lengths for the solving the optimization problem, seeking a balance between solving time and results. The results presented in this work were get considering a 5000 cycles trace. A framework were developed using MATLAB® to process the trace, invoke the ilp solver and analyze the results.

5.2 Results and Discussion

In this section the experimental results are presented and discussed.

5.2.1 Power management policy

Fig. 5.1 shows the energy savings for the intermediate value of wake-up energy (ten times the active energy) as a function of the number of banks for best-oracle and greedy policy in both applications (based on TinyOS and ContikiOS). In this analysis we have intentionally discarded the partition overhead, considered later. As the number of banks increases, the energy savings approaches the corresponding value of having all banks but one in sleep state. The figure shows that the best-oracle policy outperforms the greedy policy for both applications, as expected, and both are within 2% and 5% of the theoretical limit for the energy savings. In all cases, except for six banks, ContikiOS outperforms TinyOS by a narrow margin. The results presented hereafter are similar for both applications, and only the corresponding to TinyOS are analyzed more deeply.

Fig. 5.2 shows the fraction of cycles and the energy breakdown for a memory having five banks of equal size, where each contribution (i.e. access, ready, sleep, wake-up) is averaged among the different banks. The upper part clearly shows that the fraction of access cycles are equal in both cases and represent 20% of

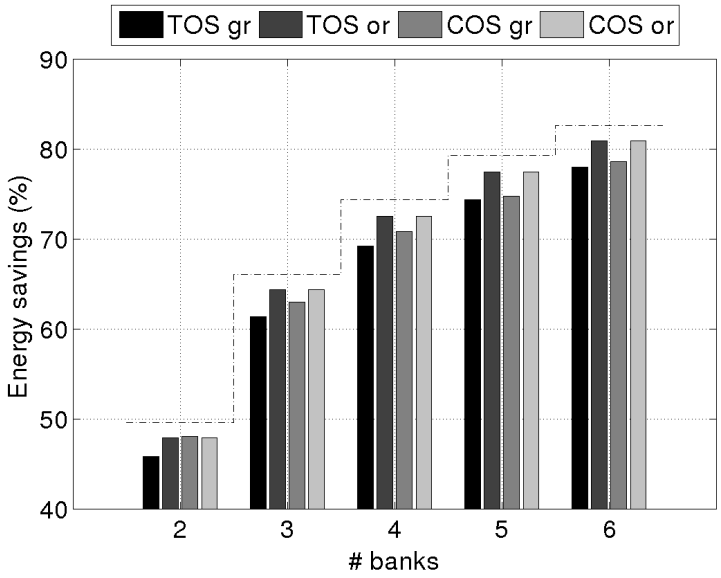

Figure 5.1: Energy savings as a function of the number of banks for best-oracle and greedy policy (denoted *gr* and *or*) in TinyOS and ContikiOS applications (denoted *TOS* and *COS*) and the theoretical limit (dashed line).

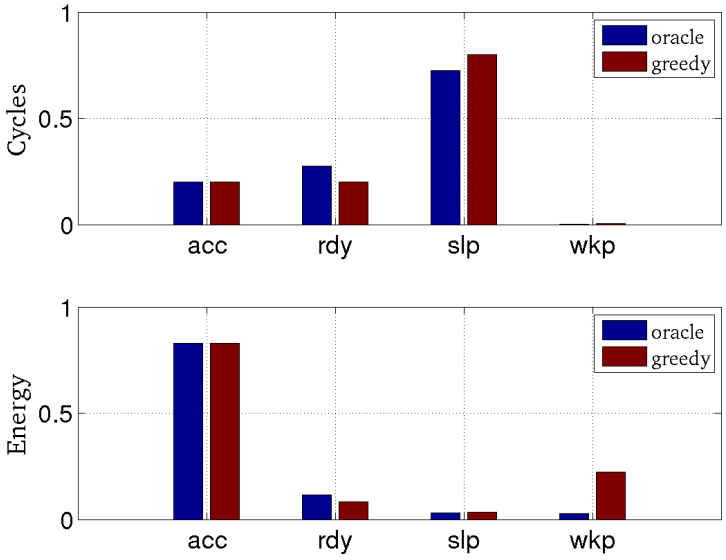

Figure 5.2: Fraction of cycles and energy breakdown where each contribution is averaged among the different banks.

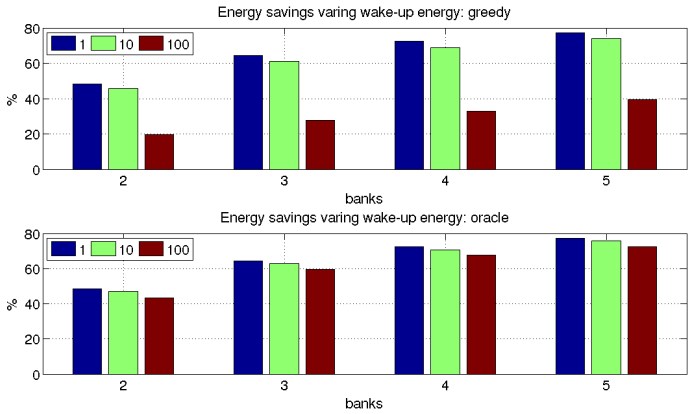

Figure 5.3: Energy savings as a function of the number of banks for best-oracle and greedy policy (denoted *gr* and *or*) for increasing wake-up cost factor (TinyOS application.

the total number of cycles, since five banks are considered (only one bank of N is active, in this case five).

For the greedy policy the number of ready cycles is equal to the access cycles, since both correspond to the active compound state. On the other hand, for the best-oracle policy part of the ready cycles correspond to active cycles, and the rest to idle cycles, in which the banks are ready but not accessed. Moreover, for the greedy policy 80% of the cycles are sleep cycles ($N-1$ banks are in sleep state) while for the best-oracle policy this percentage is slightly smaller, since idle cycles are used to reduce the wake-up rate from 0.5% to 0.12% (not visible in Fig. 5.2). The energy breakdown, Fig. 5.2 (lower part), shows that the difference between best-oracle and greedy comes from the wake-up transitions. In this case study, due to its event-driven nature, the code memory access patterns are triggered by events, leading to a chain of function calls starting with the interrupt subroutine. This chain may include the execution of subsequent functions calls starting with a queued handler function called by a basic scheduler. This applies both to TinyOS and ContikiOS, in the first case the queued function is a deferred task and in the latter case is a protothread. The allocation of highly correlated functions to the same bank leads to a bank access pattern with a high temporal locality. Hence, the total wake-up fraction across the banks is very low. This explains the modest gain of applying the best-oracle policy.

Finally, Fig. 5.3 shows the energy savings for different wake-up energy costs: one, ten (analyzed so far) and one hundred. It can be seen that for high wake-up energy costs the best-oracle policy outperforms the greedy policy by roughly 40%, reducing the saving from 70% to 30% for five banks. On the contrary, for a low wake-up cost the difference is marginal. In the middle, for moderate cost, the difference is about 2%. The energy increase due to the wake-up transitions, the last factor in Eqs. (4.4) and (4.5) is proportional to the wake-up cost. This saving lost can be reduced using the best-oracle policy by increasing the idle cycles with low relative energy cost.

Table 5.2: Optimum number of banks as a function of partition overhead.

$k_{ovhd}(\%)$	0	1	2	3	5
N_{opt}^{est}	∞	10	7	6	4
$\delta E_{N,eff}^{max}(\%)$	98.2	78.3	70.1	63.8	53.6

5.2.2 Partition overhead effect on energy savings

The optimum number of banks estimated using Eq. (4.9) (after rounding) as a function of k_{ovhd} (1%, 2%, 3% and 5%) is shown in Table 5.2. The energy savings is limited by the partition overhead, reaching a maximum of 78.3% for an overhead of 1%. The energy saving limit, as the partition overhead tends to zero and N to infinity, is 98.2% ($1 - E_{slp}/E_{act}$).

Table 5.3 compares the energy saving results as a function of the number of banks and the partition overhead. In the upper part, the table gives the maximum achievable savings calculated using Eq. (4.8). It can be observed that with a partition overhead of 3% the optimum number of banks is six, whereas with 5% is four, both highlighted in gray. Obviously, these number of banks corresponds to the values in Table 5.2. In the middle part of the table it can be observed that the maximum energy saving for greedy strategy with 3% and 5% of partition overhead is achieved for six and five banks respectively, different from the estimated value using the maximum achievable savings. This means that the saving loss due to wake-up transitions shifts the optimum number of banks. Finally, similar results are obtained for the best-oracle strategy, but with higher energy savings.

5.2.3 Duty-cycle operation

Fig. 5.4 shows the energy consumption of the memory, Eq. (4.19), normalized respect to E_{act} as a function of the number of banks for different duty-cycles. It is considered the maximum achievable savings, i.e., the wake-up energy contribution is neglected, but it is not the partition overhead. Let's briefly analyze the corner cases: null duty-cycle corresponds to an inactive memory consuming a constant power (a baseline relative energy of E_{slp}/E_{act}) and 100% duyt-cycle corresponds to an always-on memory studied so far. It can be appreciated that the energy is minimum for ten banks, as estimated.

The duty-cycle found in the characterization presented in Chapter 2 (Section 2.4.2) is about 3%. Fig. 5.5 plots the energy consumption relative to a single bank memory (with sleep state). It can be observed that for a 3% duty-cycle an energy reduction of 50% is obtained for ten banks (marked with black dot). Then, for higher duty-cycles this energy reduction is increased as suggested by the model.

In conclusion, even though the memory maximizes savings for higher duty-cycles, for relatively low duty-cycles substantials savings are achieved, 50% for a 3% duty-cycle using a memory with the optimum number of banks.

Table 5.3: Energy saving comparison: maximum, greedy and best-oracle.

maximum		number of banks				
		2	3	4	5	6
$k_{ovhd}(\%)$	1	47.08	62.44	69.62	73.53	75.80
	2	45.08	59.44	65.62	68.53	69.80
	3	43.08	56.44	61.62	63.53	63.80
	5	39.08	50.44	53.62	53.53	51.80
greedy		number of banks				
		2	3	4	5	6
$k_{ovhd}(\%)$	1	43.82	58.33	65.18	69.36	71.99
	2	41.82	55.33	61.18	64.36	65.99
	3	39.82	52.33	57.18	59.36	59.99
	5	35.82	46.33	49.18	49.36	47.99
best-oracle		number of banks				
		2	3	4	5	6
$k_{ovhd}(\%)$	1	46.40	61.88	69.12	73.07	75.41
	2	44.40	58.88	65.12	68.07	69.41
	3	42.40	55.88	61.12	63.07	63.41
	5	38.40	49.88	53.12	53.07	51.41

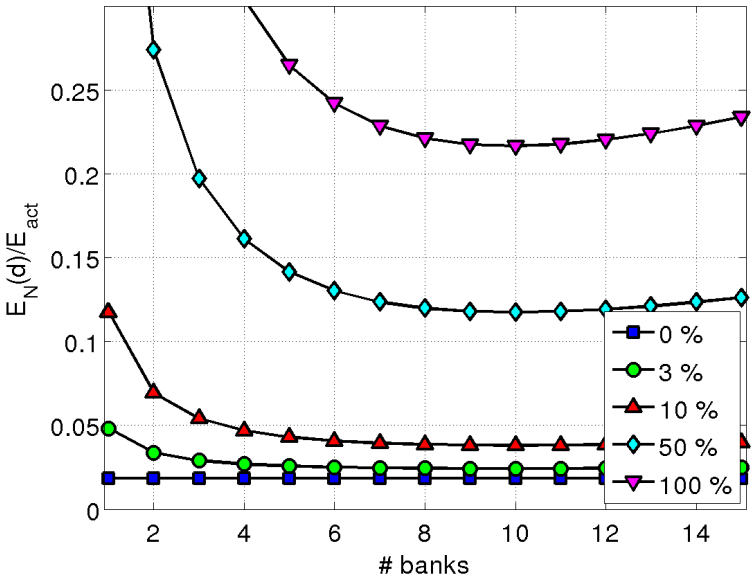

Figure 5.4: Energy normalized respect to E_{act}.

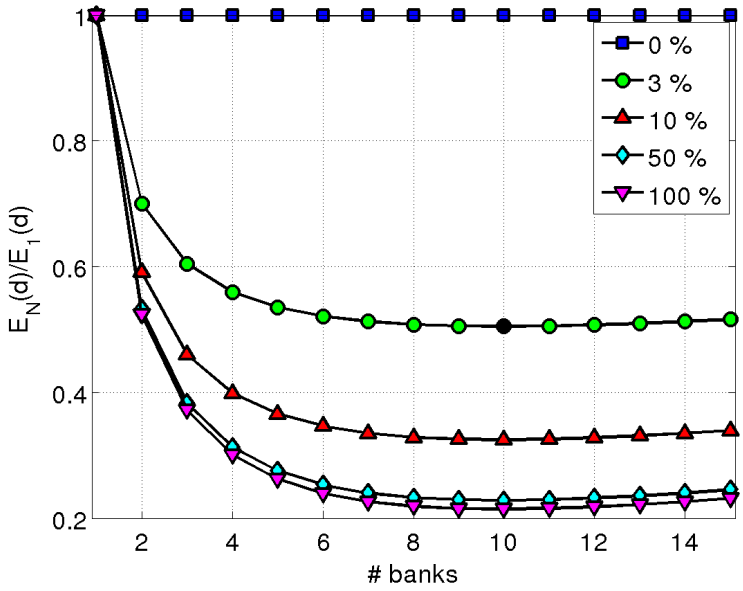

Figure 5.5: Energy normalized respect to a single-bank memory (with sleep state).

5.3 Summary and conclusions

The presented experiments were designed to determine the effect of previously identified factors on the achieved energy savings. Returning to the questions posed at the beginning of this chapter, it is now possible to state that the code memory contents of applications can be properly allocated to memory banks, so that a significant saving is obtained. The energy saving is augmented by minimizing the accumulated wake-up rate and the idle cycles. The optimization algorithm allocate blocks to banks in such a way that bank accesses present high temporal locality, and thus, low wake-up rates.

The results of the experiments indicate that the energy savings are close to the maximum achievable, corresponding to have all banks but one in sleep state. As the number of banks increases, the energy savings approaches the limit given by the memory design parameters and limited by the partition overhead. We have found energy savings up to 78.3% for a partition overhead of 1% with a memory of ten banks. Savings increases with duty-cycle. In other words for low duty-cycles lower savings are obtained. In any case, important energy reductions could be get. For example, for a 3% duty-cycle operation the energy is reduced by roughly from 78.3% to 50%. The final savings also depends on the sleep energy contribution to the total energy. The lower the sleep energy the higher the impact of individually mange the banks in the active state.

One of the more significant findings to emerge from this work is that the savings difference between best-oracle and greedy is scarce for relative low wake-up energy costs. Additionally, both are within 2% and 5% of the maximum achievable savings. On the contrary, the extra benefit of the best-oracle over the greedy policy

is significant for high wake-up energy costs. Consequently, the additional benefit of using an advanced algorithm to predict future access to banks must justify the increased complexity and compensate the extra energy and area cost.

It was also shown that the estimated number of banks in most cases agrees with the optimum number. The possible difference came from the saving loss due to wake-up transitions, being the optimum number greater than the estimation. All the same, the estimated value can reduce the search space to quickly find the optimum, starting from this value to higher number of banks.

The results of the experimental evaluation show that huge savings are obtained using a SRAM memory with power-manageable banks. The current findings enhance our understanding of the potential and limitations of these memories.

This page is intentionally left blank.

Chapter 6

Conclusions

This chapter concludes this work. First, we summarize the work of individual chapters into a single concluding section. Next, we present the main contributions. Finally, we discuss future perspectives of current research work.

6.1 Summary of the thesis

One of the main challenges in wireless sensor networks is to provide high level programming abstractions to enable more complex applications, accompanied with energy-efficient platforms that satisfies extended lifetime requirements.

High level abstractions have many advantages since ease the development of wireless sensor networks providing productivity gain, portability, among other benefits. For example, Java requires a much shorter learning curve than other specialized languages (such nesC of TinyOS) and even language C. It presents other worthwhile features that makes Java virtual machine adoption in wireless sensor networks an interesting option for application development. The scaling of CMOS technology enables refined applications built on top of emerging virtual machines that benefit from increased available computation power. However, these more and more complex applications requires also enlarged memory size to hold the programs. The energy drain by memories is paying the price for the flexibility of programmable processors. A drawback of smaller feature transistor sizes is the increased leakage power. Additionally, the energy budget of processors is dominated by the memories that supplies data to them. Energy-efficient memory design represent an opportunity to reduce the energy consumption in sensor nodes, enabling a broader adoption of the wireless sensor networks.

NOR-Flash is more efficient than SRAM at lower frequencies, since static power prevails over dynamic power, and it benefits from its low-leakage feature. SRAM is more efficient that NOR-Flash for a wide range of frequency if the static power is reduced. Some general applicable low-power techniques involves dynamically decrease the system power consumption, trading performance for power, or even entering in an inactivity state of reduced power. In the case of SRAM memory can be applied to the whole cell array, or to individual banks. On the one hand, the

SRAM with a sleep state, manageable as a unit, is very profitable in low duty-cycle system with relative long idle periods. On the other hand, SRAM with power-manageable banks can put the idle banks into the sleep state, reducing the leakage power, even when the system is active. The fundamental idea behind a memory with power-manageable banks is that only one bank is active per access, and the remaining idle banks can be put into the sleep state. The memory architecture relies on that banks can be individually managed to control its power state.

The power manager controls the memory banks states in base of a given policy. The greedy policy is the simplest policy to implement, and puts a bank in sleep state as becomes idle, and back to ready state as soon as it is required. On the contrary, the ideal power manager controls the banks optimally since has a priori knowledge of the entire workload trace. For this reason the policy is named best-oracle. Evidently, there is no real implementation of the best-oracle policy, and it only serves as an upper bound reference of the potential savings.

A detailed model of the energy saving was presented for uniform banks with the mentioned power management schemes. The model gives valuable insight into key factors (coming from the system and the workload) that are critical for reaching the maximum achievable energy saving. The energy savings increases as a function of the number of banks, and it is limited by the partition overhead.

We evaluated the benefits of using a partitioned memory in WSNs by the simulation of two real applications, one based on TinyOS and the other on ContikiOS. The energy saving is maximized by properly allocating the program memory to the banks in order to minimize the accumulated wake-up rate and the idle cycles. The memory content allocation problem were solved by an integer linear program formulation.

The energy saving obtained by simulations were compared with the limits given by the derived expressions, showing a good correspondence. The best-oracle policy outperformed the greedy policy as expected. However, the extra benefit of the best-oracle over the greedy policy was significant only for high wake-up energy costs. Conversely, for relative low wake-up energy costs, the difference between best-oracle and greedy was scarce. In this case the additional benefit of using an advanced algorithm to predict future access to banks probably won't justify the increasing complexity and compensate the extra energy and area cost.

Thanks to our modeling, a near optimum number of banks can be estimated reducing the search space to quickly find the optimum, by restricting the search to its neighborhood. Moreover, in most cases, the estimated number of banks agrees with the optimum number.

We have found that aggressive energy savings can be obtained using a banked memory, close to 80% for a partition overhead of 1% with a memory of ten banks using the most recent technology parameters. Highly correlated memory blocks allocated to the same bank led to a bank access pattern with a high temporal locality, thus reducing the number of wake-ups. Hence, the total wake-up fraction across the banks was very low. The energy saving scales down with the duty-cycle, but even for low duty-cycles significant extra benefits are obtained.

6.2 Main contributions

The main contributions of this work are organized in three groups, described next.

1. Characterization and analysis of the run-time execution of typical current applications, showing that memory energy consumption may hinder the spreading of future WSN applications.

2. Review of memory technologies with focus in code memory energy consumption, demonstrating that SRAM memory with power-manageable banks helps to cope with increasing leakage current of the scaling of CMOS technology.

3. Proposal and in-depth study of SRAM memory with power-manageable banks for extending sensor nodes life-time, and at the same time, enabling the increasing complexity of applications.

The third and most important contribution is described in detail below.

- We have proposed a novel and detailed model of the energy saving, gaining valuable insight into key factors, coming from the system and the workload, that are critical for reaching the maximum achievable energy saving. A crucial parameter is the sleep to active energy ratio, since determine the maximum energy savings. The energy savings approaches to this maximum for increasing number of banks. However, the effective energy saving is limited by the partition overhead and the wake-up energy cost of the banks. There is an optimum number of banks, which depends on the previously mentioned parameters.

- Our experimental results showed that memory contents can be arranged in memory banks to have high temporal locality accesses, and, consequently, the overall wake-up rate is minimized. For that, we followed a methodology in which a memory access trace is used to solve an optimization problem that outputs the block-to-bank map. The optimum number of banks is found iteratively.

- We have addressed the power management issue, evaluating two power management policies: best-oracle and greedy. Both policies are in opposite ends in terms of complexity and results, thus covering all the potential possibilities. Our findings suggests that adopting an advanced power management probably might not justify its implementation energy cost, since the best-oracle is only marginally better than a greedy strategy for low and moderate wake-up energy cost.

- We have provided a formula for estimating the optimum number of banks. In most cases the estimated value match the optimum. The estimated value reduce the search space to quickly find the optimum and determine the block-to-bank map.

- We have found an energy reduction close to 80%, depending on the memory parameters (partitioning overhead, energy reduction of the sleep state and the wake-up energy cost) for an application under high workload.
- We have considered the duty-cycle operation of this memory, presents in wireless sensor network nodes. The energy saving scales down lowering the duty-cycle. The additional energy saving due to individually manage the memory banks will be significant for relative high duty-cycles and low consumption in the sleep state.

Finally, a number of limitations need to be noted regarding the present work.

- The results are based on models and simulations. Our finding contributes to conduct the efforts in low-power integrated-circuit design, when they are particularly applied to banked memories. Despite the models were build from reliable sources, it would be interesting to deeply investigate memory internal architectures to design and fabricate a test chip to confirm the different memory parameters (including partition overhead) and the predicted savings.
- The evaluation performed has been done using applications developed with the two most popular state of the art software platforms: TinyOS and ContikiOS. In both operating systems the code memory was several times larger than the data memory. In this fact we founded to choose adopting a memory with power-manageable banks for instructions. In the case of virtual machines, the programs are bytecodes which can be read from data or code memory (they are actually assigned to the read-only data segment). All the same, nowadays they still only represent a thin layer that relay in an underneath operating system. Therefore the code memory still prevails, and the virtual machine would present a run-time behavior similar to those studied. However, the previous hypothesis need to be reviewed in the future for new virtual machines designed for wireless sensor network.
- The access pattern to banks in execution time must be similar to the optimization phase in order to achieve similar energy saving. This means that the optimization or "training" trace should be statistical representative of any trace. However, relative short traces were used for the optimization to obtain reasonable solving time. To overcome this limitation the segment trace was carefully selected, however other solutions could be more effective.

6.3 Future works

This research has thrown up many questions in need of further investigation.

- Further research might explore instruction and data memory sharing the same banked memory, in accordance to the limitation mentioned above regarding code and data memory sizes. As in the present work, the key point

is whether the data memory access pattern is such that the content can be distributed among the banks to get an overall low wake-up rate. Memory protection issues should be taken into account, however a first approach could be having two separate memories sharing the power manager.

- A number of possible future studies on the field of optimization are obvious. They may range from heuristic algorithms to cope with larger traces to these combined with other techniques based on static analysis. In this direction it would be very interesting to work towards design and build a tool to automatically process the linker-locator output to asign blocks to banks. However, it needs to take into account that traditional techniques applied to internal code structure might not be directly applicable, since memory access patterns are highly influenced by external interaction through hardware interruptions (e.g. sensors, radio, and so on). Additionally, the block-to-bank mapping need to be completed using some algorithm, if the optimization phase did not cover the whole binary application. The algorithm could be based on extra execution information, i.e., traces not used in the previous optimization phase, and on static or structural information, such program call graphs.

- More work is needed on power management policies for the case of high wake-up energy cost. Our findings suggest that a predictive algorithm to manage the banks might lead to higher energy savings than a greedy policy. However, the implementation costs and performance should be evaluated to determine its advantage. Besides, time-out policy is as simple as greedy, and it may improve energy savings in certain conditions (memory energy parameters and workload characteristics).

- The proposed model is easily extended for a memory with non-uniform sized banks. The idea is dividing a memory in several banks. The memory can be configured to merge banks into sets. Each set constitute a manageable unit, being equivalent to the banks of the present work. Primary analysis suggests that non-uniform banks get the better of memories with relatively high wake-up energy cost and high partition overhead. More research is required to assess if it is worth to work on a more flexible structure as the required in a memory with non-uniform banks. Note that, a memory with uniform sized banks is a particular case of this memory, however the partition overhead model need to be revisited in this case.

- Further research regarding practical implementation and use of this memories would be of great help to favor its widest adoption. Accordingly, it is of primary importance address the design of a configurable memory. Regular internal structure might help in reducing partition overhead, which accompanied with appropriate solution could allow to configure the memory in the field. Based on the provided models it is needed to further study the effects on the energy saving of different practical issues. For example, one question that need to be answered is how many banks to choose to allocate a binary

application in a given configurable memory. The memory could be much larger than the application size. It could be better to use the least amount of banks and shutdown the unused ones (with a power consumption lower than the sleep state), or use more banks, if the wake-up rate is reduced, but this would lead to an increased sleep power .

- Finally, our proposal was conceived for wireless sensor nodes, so it was assessed accordingly. However, it seems that the power-manageable banks could lead to huge energy saving in other event-driven applications

Minors future works or not central to this thesis:

- Evaluate the processing workload due to time management only. It seems that an important fraction of time the microcontroller wakes up just to increment the system tick (or decrement timer counters) and verify if any time request had time-out. If this is true, it should strongly consider adopt a variable timer, like the proposed for RTOS [23]. (see Fig. 2.7 and 2.8 and the respective discussion).

Appendix A

Memory simulations

A.1 CACTI estimations

For the memory estimation the web version of CACTI was used (version CACTI 5.3 rev. 174). It can be accessed in http://www.hpl.hp.com/research/cacti/.

Configurations

- Pure RAM Interface
- RAM Size (bytes): from 512 to 32768
- Nr. of Banks: 1
- Read/Write Ports: 1
- Read Ports: 0
- Write Ports: 0
- Single Ended Read Ports: 0
- Nr. of Bits Read Out: 16
- Technology Node (nm): 65
- Temperature (300-400 K, steps of 10): 400
- RAM cell/transistor type in data array (choose ITRS transistor for SRAM cell): ITRS-HP
- Peripheral and global circuitry transistor type in data array: ITRS-HP
- RAM cell/transistor type in tag array (choose ITRS transistor for SRAM cell): ITRS-HP
- Peripheral and global circuitry transistor type in tag array: ITRS-HP

- Interconnect projection type: Conservative
- Type of wire outside mat: Semi-global

A.2 NVSim estimations

The application can be downloaded from the project site: http://nvsim.org/ (password required that should be asked for the authors).

There are two categories of the configuration input files: .cfg and .cell[1].

Memory: file.cfg

Next, we list the different options for each parameter of the general configuration file, where some possible values are given between curly braces. Some notes are added between parenthesis.

```
DesignTarget: {cache, RAM, CAM} (Note: if RAM is selected Cache
  option ignored)
OptimizationTarget: {ReadLatency, WriteLatency, ReadDynamicEnergy,
  WriteDynamicEnergy, LeakagePower, Area} = ReadLatency
ProcessNode: {200, 120, 90, 65, 45, 32}
Capacity (KB): 512 (Note: can select MB or KB and the number.)
WordWidth (bit): {512, 64, x} = 32
DeviceRoadmap: {HP, LSTP, LOP}
LocalWireType: {LocalAggressive, LocalConservative, SemiAggressive,
  SemiConservative, GlobalAggressive, GlobalConservative}
  = LocalAggressive
LocalWireRepeaterType: {RepeatedOpt, Repeated5\%Penalty, ..., }
  = RepeatedNone
LocalWireUseLowSwing: {Yes, No} = No
GlobalWireType: {LocalAggressive, LocalConservative, SemiAggressive,
  SemiConservative, GlobalAggressive, GlobalConservative}
  = GlobalAggressive
GlobalWireRepeaterType: {RepeatedNone, RepeatedOpt,
  Repeated5\%Penalty, ...,  Repeated50\%Penalty} = RepeatedOpt
GlobalWireUseLowSwing: {Yes, No} = No
Routing: {H-tree, non-H-tree} = H-tree
InternalSensing: {true, false} = true
MemoryCellInputFile: {SRAM.cell, Memristor\_1.cell,
  PCRAM\_JSSC\_2007.cell, PCRAM\_JSSC\_2008.cell,
  MRAM\_ISSCC\_2007.cell, PCRAM\_IEDM\_2004.cell,
  MRAM\_ISSCC\_2010\_14\_2.cell, SLCNAND.cell}
  = SLCNAND.cell or SRAM.cell
```

[1]See: http://nvsim.org/wiki/index.php?title=Documentation for limited documentation.

```
Temperature (K): 380
BufferDesignOptimization: {latency, area, balance} = latency
(Note: the following parameters are optional and force a
given configuration.)
ForceBank (Total AxB, Active CxD): {8x1, 1x1;  1x4, 1x4;
  2x2, 2x2}: 2x2, 2x2
ForceMuxSenseAmp: 8
ForceMuxOutputLev1: 4
ForceMuxOutputLev2: 1
ForceBank (Total AxB, Active CxD): 1x1, 1x1
ForceMuxSenseAmp: 1
ForceMuxOutputLev1: 1
ForceMuxOutputLev2: 2
```

Cell

Next, two examples of cell configuration file are listed.

Example: SLCNAND.cell

```
-MemCellType: SLCNAND
-CellArea ($F^2$): 4
-CellAspectRatio: 1
-GateCouplingRatio: 0.7
-FlashEraseTime (ms): 1.25
-FlashProgramTime (us): 200
-FlashEraseVoltage (V): 16
-FlashProgramVoltage (V): 6
-FlashPassVoltage (V): 3.8
-ReadMode: voltage
-ReadVoltage (V): 0.5
```

Example: SRAM.cell

```
-MemCellType: SRAM
-CellArea ($F^2$): 146
-CellAspectRatio: 1.46
-ReadMode: voltage
-AccessType: CMOS
-AccessCMOSWidth (F): 1.31
-SRAMCellNMOSWidth (F): 2.08
-SRAMCellPMOSWidth (F): 1.23
```

This page is intentionally left blank.

Appendix B

Evaluation tools

B.1 Integer linear program: AMPL model

Next the AMPL model is listed for the best-oracle policy.

```
#############################################################
## model for memory blocks allocation to banks
#############################################################

#### parameters ####

## number of memory banks
param n >=0;
## memory banks (index set)
set B := 1..n;
## activation, active mode, access, and deactivation
   energy
param ep >=0;
param e  >=0;
param ea >=0;
param em >=0;

## overhead constant
param koh >= 0;

## memory bank sizes
param sb { B } >= 0;
## number of memory blocks
param m >=0;
## memory blocks (index set)
set M := 1..m;
## memory block sizes
param sm { M } >= 0;
```

```
## number of time steps
param t >=0;
## time points (index set)
set T0:= 1..t;
set T := 2..t;

## block access pattern
param am{T0} integer;
# a: block access pattern
param a{mi in M, ti in T0} binary := if (am[ti]=mi) then
    1;

#### decision variables ####

## allocation of memory blocks to banks
var l { B cross M } binary;
## access of memory banks (auxiliary)
var ab { B cross T } binary;
## bank active during t
var o { B cross T0 } binary;
## bank activated at start of t
var op { B cross T } binary;

#### model ####

minimize TotalEnergyConsumption:
        sum { t0 in T, b0 in B } (op[b0,t0]*ep*sb[b0] +
           ab[b0,t0]*ea*sb[b0] + o[b0,t0]*e*sb[b0]);

subject to Activation { b0 in B, t0 in T }:
        op[b0,t0] >= o[b0,t0]-o[b0,t0-1];

subject to Access { b0 in B, t0 in T}:
        ab[b0,t0] = sum{m0 in M} l[b0,m0]*a[m0,t0];

subject to ActiveWhenAccessed { b0 in B, t0 in T}:
        o[b0,t0] >= ab[b0,t0];

subject to AllocatedOnce { m0 in M }:
        sum { b0 in B } l[b0,m0] = 1;

subject to RespectsSize { b0 in B }:
        sum { m0 in M } l[b0,m0]*sm[m0] <= sb[b0];
```

```
subject to StartsDeactivate { b0 in B }:
        o[b0,1] = 0;

solve;

display sb;
display sm;
display a;
display am;
display a;
display l;
display o;
display ab;
display op;

end;
```

B.2 Simulation tools

Initial wireless sensor networks simulations were performed using the following tools:

- AVRora [1] developed for the AVR microcontrollers [106], which later incorporates the simulation IEEE 802.15.4 compliant radio chips, allowing emulation of sensor nodes such as MicaZ [31]. In this work it was used as stand-alone simulator (later it was incorporated to COOJA).
- WSim[2] [43] is a platform simulator that relies on cycle accurate full platform simulation. It can be used standalone for debuging purposes or with the WSNet simulator to perform the simulation of a complete sensor network.

B.2.1 Energest modifications

Energest functionality is implemented in ContikiOS as a module that could be included or not in an application through macros. The idea is to measure and accumulate the elapsed time in different power states. Periodically these values are reported to the sink node where each energy consumption contribution is computed.

In the code location where a change in the power state it is performed, a macro signaling the transition is called. Two macro are used, one for entering a state, so that it is started the time measurement, and other to existing a state, recording the elaspsed time in that state and accumulating the time measurement.

[1] http://compilers.cs.ucla.edu/avrora/
[2] http://wsim.gforge.inria.fr/

Listing B.1: Code excerpt .

```
1
2  while(1) {
3  // ...
4        do {
5          watchdog_periodic();
6          r = process_run();
7        } while(r > 0);
8  // ...
9          static unsigned long irq_energest = 0;
10         /* Re-enable interrupts and go to sleep atomically. */
11         ENERGEST_OFF(ENERGEST_TYPE_CPU);
12         ENERGEST_ON(ENERGEST_TYPE_LPM);
13         /* We only want to measure the processing done in IRQs when we are asleep
               , so we discard the processing time done when we were awake. */
14        energest_type_set(ENERGEST_TYPE_IRQ, irq_energest);
15         watchdog_stop();
16         /* check if the DCO needs to be on - if so - only LPM 1 */
17         if (msp430_dco_required) {
18           _BIS_SR(GIE | CPUOFF); /* LPM1 sleep for DMA to work!. */
19         } else {
20           _BIS_SR(GIE | SCG0 | SCG1 | CPUOFF); /* LPM3 sleep. This statement
                 will block until the CPU is woken up by an interrupt that sets the
                 wake up flag. */
21         }
22         /* We get the current processing time for interrupts that was done during
               the LPM and store it for next time around. */
23         dint();
24         irq_energest = energest_type_time(ENERGEST_TYPE_IRQ);
25         eint();
26         watchdog_start();
27         ENERGEST_OFF(ENERGEST_TYPE_LPM);
28         ENERGEST_ON(ENERGEST_TYPE_CPU);
29       }
```

The states considered in the actual implementation are: LPM, IRQ, CPU, Transmit, and Listening, corresponding to the low power mode (LPM3), microcontroller active mode in interrupt context, microcontroller in active state in normal mode, radio transmitting and radio in reception mode, respectively.

The Listing B.1 with code excerpt of the main loop of ContikiOS show the use of the Energest macros.

Modifications

The main limitation is that the application that send the report to the sink node, collect-view, allocated data variables of 16-bits wide. We could verified that this maximum time is overflow for the LPM and CPU states in normal operation. The Energest module has 32-bit variables for accumulating the elapsed time. When the application is preparing the message with these data, if an accumulated time does no fit in a 16-bits word, it scaled all the times values. This procedure keep the relations between the different measured times, but introduce an error for computing the total accumulate time though summing all the reported times.

This limitation was easily fixed using spare data variable in the message structure, to send 32-bits accumulated time for all the considered state.

This page is intentionally left blank.

Bibliography

[1] MSP430FR573x, MSP430FR572x Mixed Signal Microcontroller (Rev. H). http://www.ti.com/lit/ds/symlink/msp430fr5739.pdf, September 2013.

[2] R. Allgayer, Leonardo Steinfeld, C. E. Pereira, Luigi Carro, and F. R. Wagner. Aplicação de Agentes Móveis em Redes de Sensores sem Fio para Localização e Seguimento de Objetos Alvos Móveis. In *Congresso Brasileiro de Automática, 18. Bonito, MS, Brasil*, 2010.

[3] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '04, pages 259–267, New York, NY, USA, 2004. ACM.

[4] Faisal Aslam, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan Rührup, and Zastash Uzmi. Optimized Java Binary and Virtual Machine for Tiny Motes Distributed Computing in Sensor Systems. In Rajmohan Rajaraman, Thomas Moscibroda, Adam Dunkels, and Anna Scaglione, editors, *Distributed Computing in Sensor Systems*, volume 6131 of *Lecture Notes in Computer Science*, chapter 2, pages 15–30. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[5] A. Bachir, M. Dohler, T. Watteyne, and K. K. Leung. MAC Essentials for Wireless Sensor Networks. *Communications Surveys & Tutorials, IEEE*, 12(2):222–248, 2010.

[6] J. Balfour, W. J. Dally, D. Black-Schaffer, V. Parikh, and JongSoo Park. An Energy-Efficient Processor Architecture for Embedded Systems. *Computer Architecture Letters*, 7(1):29–32, January 2008.

[7] Tobias Becker, Peter Jamieson, Wayne Luk, Peter Y. K. Cheung, and Tero Rissa. Towards benchmarking energy efficiency of reconfigurable architectures. In *2008 International Conference on Field Programmable Logic and Applications*, pages 691–694. IEEE, 2008.

[8] Fabio L. Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. John Wiley & Sons, April 2007.

[9] Tony Benavides, Justin Treon, Jared Hulbert, and Weide Chang. The Enabling of an Execute-In-Place Architecture to Reduce the Embedded System Memory Footprint and Boot Time. *JCP*, 3(1):79–89, 2008.

[10] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, June 2000.

[11] L. Benini, L. Macchiarulo, A. Macii, and M. Poncino. Layout-driven memory synthesis for embedded systems-on-chip. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 10(2):96–105, April 2002.

[12] D. Bertozzi and L. Benini. Hardware Platforms for Third-Generation Mobile Terminals Memories in Wireless Systems. In Rino Micheloni, Giovanni Campardo, and Piero Olivo, editors, *Memories in Wireless Systems*, Signals and Communication Technology, chapter 1, pages 1–28. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[13] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, April 2003.

[14] Subhasis Bhattacharjee and Dhiraj K. Pradhan. LPRAM: a low power DRAM with testability. In *ASP-DAC '04: Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, pages 390–393, Piscataway, NJ, USA, 2004. IEEE Press.

[15] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, August 2005.

[16] Niels Brouwers, Peter Corke, and Koen Langendoen. A java compatible virtual machine for wireless sensor nodes. In *the 6th ACM conference*, SenSys '08, pages 369–370, New York, New York, USA, 2008. ACM Press.

[17] A. Calimera, L. Benini, A. Macii, E. Macii, and M. Poncino. Design of a Flexible Reactivation Cell for Safe Power-Mode Transition in Power-Gated Circuits. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(9):1979–1993, September 2009.

[18] A. Calimera, A. Macii, E. Macii, and M. Poncino. Design Techniques and Architectures for Low-Leakage SRAMs. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 59(9):1992–2007, 2012.

[19] Edgar H. Callaway. The Wireless Sensor Network MAC. In *Handbook of Sensor Networks*, pages 239–276. John Wiley & Sons, Inc., 2005.

[20] C. Cano, B. Bellalta, A. Sfairopoulou, and M. Oliver. Low energy operation in WSNs: A survey of preamble sampling MAC protocols. *Computer Networks*, 55(15):3351–3363, October 2011.

[21] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks. In *2008 7th International Conference on Information Processing in Sensor Networks (IPSN)*, IPSN '08, pages 233–244, Washington, DC, USA, April 2008. IEEE.

[22] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov. Mote Runner: A Multi-language Virtual Machine for Small Embedded Devices. In *Sensor Technologies and Applications, 2009. SENSORCOMM '09. Third International Conference on*, volume 0, pages 117–125, Los Alamitos, CA, USA, June 2009. IEEE.

[23] Hojung Cha, Sukwon Choi, Inuk Jung, Hyoseung Kim, Hyojeong Shin, Jaehyun Yoo, and Chanmin Yoon. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 148–157, New York, NY, USA, 2007. ACM.

[24] Ioannis Chatzigiannakis, Georgios Mylonas, and Sotiris Nikoletseas. 50 ways to build your application: A survey of middleware and systems for Wireless Sensor Networks. In *2007 IEEE Conference on Emerging Technologies & Factory Automation (EFTA 2007)*, pages 466–473. IEEE, September 2007.

[25] G. Chen, F. Li, M. Kandemir, O. Ozturk, and I. Demirkiran. Compiler-Directed Management of Leakage Power in Software-Managed Memories. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, volume 00, pages 450–451. IEEE, 2006.

[26] L. Crippa, R. Micheloni, I. Motta, and M. Sangalli. Nonvolatile Memories: NOR vs. NAND Architectures Memories in Wireless Systems. In Rino Micheloni, Giovanni Campardo, and Piero Olivo, editors, *Memories in Wireless Systems*, Signals and Communication Technology, chapter 2, pages 29–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[27] David Culler, Jason Hill, Mike Horton, Kris Pister, Robert Szewczyk, and A. Woo. Mica: The commercialization of microsensor motes. *Sensors Magazine*, 19(4):40–48, 2002.

[28] William J. Dally, James Chen, R. Curtis Harting, James Balfour, David Black-Shaffer, Vishal Parikh, David Sheffield, and Jongsoo Park. Efficient Embedded Computing. *Computer*, 41(7):27–32, July 2008.

[29] Edison de Freitas, Bernhard Bösch, Rodrigo Allgayer, Leonardo Steinfeld, Flávio Wagner, Luigi Carro, Carlos Pereira, and Tony Larsson. Mobile Agents Model and Performance Analysis of a Wireless Sensor Network Target Tracking Application. In *Proceedings of the 11th International Conference and 4th International Conference on Smart Spaces and Next Generation Wired/Wireless Networking*, NEW2AN'11/ruSMART'11, pages 274–286, Berlin, Heidelberg, 2011. Springer-Verlag.

[30] Edison P. de Freitas, Bernhard Bösch, Rodrigo Allgayer, Leonardo Steinfeld, Carlos Pereira, Tony Larsson, F. W. Wagner, and L. Carro. Análise de Desempenho da Utilização do Framework AFME em uma Aplicação de Seguimento de Trajetória para Rede de Sensores sem Fio utilizando Agentes Móveis. In *SBAI 2011*, São João del-Rei, September 2011.

[31] Rodolfo de Paz Alberola and Dirk Pesch. AvroraZ: extending Avrora with an IEEE 802.15.4 compliant radio chip model. In *Proceedings of the 3nd ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*, PM2HW2N '08, pages 43–50, New York, NY, USA, 2008. ACM.

[32] Minh Q. Do, Mindaugas Drazdziulis, Per L. Edefors, and Lars Bengtsson. Leakage-Conscious Architecture-Level Power Estimation for Partitioned and Power-Gated SRAM Arrays. In *ISQED '07: Proceedings of the 8th International Symposium on Quality Electronic Design*, pages 185–191, Washington, DC, USA, 2007. IEEE Computer Society.

[33] Xiangyu Dong, Cong Xu, Yuan Xie, and N. P. Jouppi. NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(7):994–1007, July 2012.

[34] Adam Dunkels. The ContikiMAC Radio Duty Cycling Protocol. Technical Report T2011:13, Swedish Institute of Computer Science, December 2011.

[35] Adam Dunkels, Luca Mottola, Nicolas Tsiftes, Fredrik Österlind, Joakim Eriksson, and Niclas Finne. The announcement layer: beacon coordination for the sensornet stack. In *Proceedings of the 8th European conference on Wireless sensor networks*, EWSN'11, pages 211–226, Berlin, Heidelberg, 2011. Springer-Verlag.

[36] Adam Dunkels, Fredrik Österlind, Nicolas Tsiftes, and Zhitao He. Software-based On-line Energy Estimation for Sensor Nodes. In *Proceedings of the Fourth Workshop on Embedded Networked Sensors (Emnets IV)*, Cork, Ireland, June 2007.

[37] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *In Proc. 2006 SenSys*, pages 29–42, 2006.

[38] Prabal Dutta and Adam Dunkels. Operating systems and network protocols for wireless sensor networks. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 370(1958):68–84, January 2012.

[39] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro J. Marrón. COOJA/MSPSim: interoperability testing for wireless sensor networks. In *Proceedings*

of the 2nd International Conference on Simulation Tools and Techniques, Simutools '09, pages 1–7, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[40] Farzan Fallah and Massoud Pedram. Circuit and System Level Power Management. In Massoud Pedram and JanM Rabaey, editors, *Power Aware Design Methodologies*, pages 373–412. Springer US, 2002.

[41] Amir H. Farrahi, Gustavo E. Téllez, and Majid Sarrafzadeh. Memory segmentation to exploit sleep mode operation. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, pages 36–41, New York, NY, USA, 1995. ACM.

[42] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 323–338, Berkeley, CA, USA, 2008. USENIX Association.

[43] Antoine Fraboulet, Guillaume Chelius, and Eric Fleury. Worldsens: development and prototyping tools for application specific wireless sensors networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 176–185, New York, NY, USA, 2007. ACM.

[44] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The ¡i¿nesC¡/i¿ language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5):1–11, May 2003.

[45] Alan Gibbons, Aitken Robert, Kaijian Shi, Michael Keating, and David Flynn. *Low Power Methodology Manual*. Springer US, Boston, MA, 2007.

[46] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 1–14, New York, NY, USA, 2009. ACM.

[47] Olga Golubeva, Mirko Loghi, Massimo Poncino, and Enrico Macii. Architectural leakage-aware management of partitioned scratchpad memories. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1665–1670, San Jose, CA, USA, 2007. EDA Consortium.

[48] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14. IEEE, 2001.

[49] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *the 3rd international conference*, MobiSys '05, pages 163–176, New York, New York, USA, 2005. ACM Press.

[50] Vlado Handziski. *Double-Anchored Software Architecture for Wireless Sensor Networks*. PhD thesis, Berlin, Technische Universtität Berlin, Diss., 2011, 2011.

[51] M. Hempstead, D. Brooks, and G. Wei. An Accelerator-Based Wireless Sensor Network Processor in 130 nm CMOS. *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, 1(2):193–202, June 2011.

[52] Jörg Henkel, Sri Parameswaran, and Newton Cheung. Application-Specific Embedded Processors. In Jörg Henkel and Sri Parameswaran, editors, *Designing Embedded Processors*, pages 3–23. Springer Netherlands, 2007.

[53] H. Hidaka. Evolution of embedded flash memory technology for MCU. In *IC Design & Technology (ICICDT), 2011 IEEE International Conference on*, pages 1–4. IEEE, May 2011.

[54] Hideto Hidaka. Embedded Flash Memory. In Kevin Zhang, editor, *Embedded Memories for Nano-Scale VLSIs*, Integrated Circuits and Systems, pages 177–240. Springer US, 2009.

[55] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, November 2000.

[56] Philipp Hurni, Benjamin Nyffenegger, Torsten Braun, and Anton Hergenroeder. On the accuracy of software-based energy estimation techniques. In *Proceedings of the 8th European conference on Wireless sensor networks*, EWSN'11, pages 49–64, Berlin, Heidelberg, 2011. Springer-Verlag.

[57] ITRS. *International Technology Roadmap for Semiconductors, 2011 Edition, Emerging Research Devices (ERD)*, 2011.

[58] ITRS. *International Technology Roadmap for Semiconductors, 2011 Edition, Process Integration, Devices and Structures (PIDS)*, 2011.

[59] ITRS. *International Technology Roadmap for Semiconductors, 2011 Edition, System Drivers*, 2011.

[60] Xiaofan Jiang, P. Dutta, D. Culler, and I. Stoica. Micro Power Meter for Energy Monitoring of Wireless Sensor Networks at Scale. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 186–195. IEEE, April 2007.

[61] Yongsoo Joo, Yongseok Choi, Chanik Park, Sung W. Chung, EuiYoung Chung, and Naehyuck Chang. Demand paging for OneNAND^{TM} Flash eXecute-in-place. In *the 4th international conference*, pages 229–234, New York, New York, USA, October 2006. ACM Press.

[62] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: mobile networking for " Smart Dust". In *the 5th annual ACM/IEEE international conference*, MobiCom '99, pages 271–278, New York, New York, USA, 1999. ACM Press.

[63] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, December 2003.

[64] B. S. Kiyoo Itoh, K. Sasaki, and Y. Nakagome. Trends in low-power RAM circuit technologies. *Proceedings of the IEEE*, 83(4):524–543, April 1995.

[65] Kevin Klues, Chieh Jan Mike Liang, Jeongyeup Paek, E. Răzvan Musăloiu, Philip Levis, Andreas Terzis, and Ramesh Govindan. TOSThreads: thread-safe and non-invasive preemption in TinyOS. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 127–140, New York, NY, USA, 2009. ACM.

[66] JeongGil Ko, Kevin Klues, Christian Richter, Wanja Hofer, Branislav Kusy, Michael Bruenig, Thomas Schmid, Qiang Wang, Prabal Dutta, and Andreas Terzis. Low power or high performance? a tradeoff whose time has come (and nearly gone). In *Proceedings of the 9th European conference on Wireless Sensor Networks*, EWSN'12, pages 98–114, Berlin, Heidelberg, 2012. Springer-Verlag.

[67] Stephan Korsholm, Martin Schoeberl, and Anders P. Ravn. Interrupt Handlers in Java. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:453–457, 2008.

[68] Joel Koshy, Ingwar Wirjawan, Raju Pandey, and Yann Ramin. Balancing computation and communication costs: The case for hybrid execution in sensor networks. *Ad Hoc Networks*, 6(8):1185–1200, November 2008.

[69] J. Kwong and A. P. Chandrakasan. An Energy-Efficient Biomedical Signal Processing Platform. *Solid-State Circuits, IEEE Journal of*, 46(7):1742–1753, July 2011.

[70] J. Kwong, Y. Ramadass, N. Verma, M. Koesler, K. Huber, H. Moormann, and A. Chandrakasan. A 65nm Sub-Vt Microcontroller with Integrated SRAM and Switched-Capacitor DC-DC Converter. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 318–616. IEEE, February 2008.

[71] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*, chapter 7, pages 115–148. Springer-Verlag, Berlin/Heidelberg, 2005.

[72] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, October 2002.

[73] Mirko Loghi, Olga Golubeva, Enrico Macii, and Massimo Poncino. Architectural Leakage Power Minimization of Scratchpad Memories by Application-Driven Subbanking. *IEEE Transactions on Computers*, 59(7):891–904, July 2010.

[74] B. Matas, C. DeSubercausau, and Integrated C. Corporation. *Memory, 1997: Complete Coverage of Dram, Sram, Eprom, and Flash Memory IC's*. Integrated Circuit Engineering Corporation, 1997.

[75] Pablo Mazzara, Leonardo Steinfeld, Fernando Silveira, and Jorge Villaverde. Herramienta para depuración de redes de sensores inalámbricos. In *Congreso Argentino de Sistemas Embebidos (CASE), Buenos Aires, Argentina. Libro de Trabajos*, 2010.

[76] Pablo Mazzara, Leonardo Steinfeld, Jorge Villaverde, Fernando Silveira, German Fierro, Alvaro Otero, Celmira Saravia, N. Barlocco, P. Vergara, and D. Gar'ın. Despliegue y Depuración de Redes de Sensores Inalámbricos para Aplicaciones al Agro. In Roberto F. Mar'ıa Eugenia Torres, editor, *Reunión de Trabajo en Procesamiento de la Información y Control, 14, RPIC 201. Paraná, Argentina*. Universidad Nacional de Entre Ríos, 2011.

[77] A. K. I. Mendonça, D. P. Volpato, J. L. Güntzel, and L. C. V. dos Santos. Mapping Data and Code into Scratchpads from Relocatable Binaries. In *VLSI, 2009. ISVLSI '09. IEEE Computer Society Annual Symposium on*, pages 157–162, Washington, DC, USA, May 2009. IEEE.

[78] F. Menichelli and M. Olivieri. Static Minimization of Total Energy Consumption in Memory Subsystem for Scratchpad-Based Systems-on-Chips. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(2):161–171, February 2009.

[79] David Moss, Jonathan Hui, and Kevin Klues. Low power listening. *TinyOS Core Working Group, TEP*, 105, 2007.

[80] Luca Mottola and Gian Picco. Middleware for wireless sensor networks: an outlook. *Journal of Internet Services and Applications*, 3:31–39, 2012.

[81] Lode Nachtergaele, Francky Catthoor, and Chidamber Kulkarni. Random-Access Data Storage Components in Customized Architectures. *IEEE Des. Test*, 18(3):40–54, May 2001.

[82] J. Nemeth, Rui Min, Wen-Ben Jone, and Yiming Hu. Location Cache Design and Performance Analysis for Chip Multiprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(1):104–117, January 2011.

[83] Yukihito Oowaki and Tohru Tanzawa. Low Power Memory Design. In Massoud Pedram and JanM Rabaey, editors, *Power Aware Design Methodologies*, pages 51–89. Springer US, 2002.

[84] Ozcan Ozturk and Mahmut Kandemir. ILP-Based energy minimization techniques for banked memories. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):1–40, July 2008.

[85] Preeti R. Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):682–704, July 2000.

[86] Preeti R. Panda, Alexandru Nicolau, and Nikil Dutt. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration.* Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[87] M. A. Pasha, S. Derrien, and O. Sentieys. A complete design-flow for the generation of ultra low-power WSN node architectures based on micro-tasking. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 693–698. IEEE, June 2010.

[88] Muhammad A. Pasha, Steven Derrien, and Olivier Sentieys. System-Level Synthesis for Wireless Sensor Node Controllers: A Complete Design Flow. *ACM Trans. Des. Autom. Electron. Syst.*, 17(1), January 2012.

[89] Massoud Pedram and Qing Wu. Design considerations for battery-powered electronics. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 861–866, New York, NY, USA, 1999. ACM.

[90] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 364–369. IEEE, April 2005.

[91] A. Prayati, Ch Antonopoulos, T. Stoyanova, C. Koulamas, and G. Papadopoulos. A modeling approach on the TelosB WSN platform power consumption. *Journal of Systems and Software*, 83(8):1355–1363, August 2010.

[92] Hulfang Qin, Yu Cao, D. Markovic, A. Vladimirescu, and J. Rabaey. SRAM leakage suppression by minimizing standby supply voltage. In *Quality Electronic Design, 2004. Proceedings. 5th International Symposium on*, volume 0, pages 55–60, Los Alamitos, CA, USA, 2004. IEEE.

[93] Jan Rabaey. Optimizing Power @ Standby – Memory. In *Low Power Design Essentials*, Integrated Circuits and Systems, pages 233–248. Springer US, 2009.

[94] Jan M. Rabaey, Anantha P. Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits*, volume 2. Prentice Hall, 2003.

[95] V. Raghunathan, C. Schurgers, Sung Park, and M. B. Srivastava. Energy-aware wireless microsensor networks. *Signal Processing Magazine, IEEE*, 19(2):40–50, March 2002.

[96] Mohammad Rahimi, Rick Baer, Obimdinachi I. Iroezi, Juan C. Garcia, Jay Warrior, Deborah Estrin, and Mani Srivastava. Cyclops: in situ image sensing and interpretation in wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 192–204, New York, NY, USA, 2005. ACM.

[97] R. K. Raval, C. H. Fernandez, and C. J. Bleakley. Low-power TinyOS tuned processor platform for wireless sensor network motes. *ACM Trans. Des. Autom. Electron. Syst.*, 15(3), June 2010.

[98] Martin Schoeberl, Christian Thalinger, Stephan Korsholm, and Anders P. Ravn. Hardware Objects for Java. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:445–452, 2008.

[99] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java™ on the bare metal of wireless sensor devices: the squawk Java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM.

[100] J. A. Stankovic and T. He. Energy management in sensor networks. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 370(1958), 2012.

[101] Leonardo Steinfeld and Luigi Carro. The Case for Interpreted Languages in Wireless Sensor Networks. In *IESS 09 - International Embedded Systems Symposium. Langenargen, Germany*, pages 279–289. Springer, September 2009.

[102] Leonardo Steinfeld, Marcus Ritt, Luigi Carro, and Fernando Silveira. Optimum design of a banked memory with power management for wireless sensor networks. In *Memory Architecture and Organization Workshop in conjunction with ESWEEK*, October 2012.

[103] Leonardo Steinfeld, Marcus Ritt, Luigi Carro, and Fernando Silveira. A new memory banking system for energy-efficient wireless sensor networks. In *The 9th IEEE International Conference on Distributed Computing in Sensor Systems 2013 (IEEE DCoSS 2013)*, pages 215–222, Cambridge, USA, May 2013.

[104] Leonardo Steinfeld, Marcus Ritt, Fernando Silveira, and Luigi Carro. Low-power processors require effective memory partitioning. In *IESS 13 - International Embedded Systems Symposium. Paderborn, Germany*, pages 73–81. Springer, June 2013.

[105] Shyamkumar Thoziyoor, Jung H. Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *2008 International Symposium on Computer Architecture*, pages 51–62, Washington, DC, USA, June 2008. IEEE.

[106] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 477–482. IEEE, April 2005.

[107] Nicolas Tsiftes, Joakim Eriksson, Niclas Finne, Fredrik Österlind, Joel Höglund, and Adam Dunkels. A framework for low-power IPv6 routing simulation, experimentation, and evaluation. *SIGCOMM Comput. Commun. Rev.*, 40(4):479–480, August 2010.

[108] Frits Vaandrager. *Introduction*, volume 1494 of *Lecture Notes in Computer Science*, chapter 1, pages 1–3. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[109] Jean P. Vasseur and Adam Dunkels. *Interconnecting Smart Objects with IP: The Next Internet*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[110] N. Verma. Analysis Towards Minimization of Total SRAM Energy Over Active and Idle Operating Modes. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(9):1695–1703, 2011.

[111] D. P. Volpato, A. K. I. Mendonça, L. C. V. dos Santos, and J. L. Güntzel. A Post-compiling Approach that Exploits Code Granularity in Scratchpads to Improve Energy Efficiency. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 127–132. IEEE, July 2010.

[112] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550 (Proposed Standard), March 2012.

[113] M. Zwerg, A. Baumann, R. Kuhn, M. Arnold, R. Nerlich, M. Herzog, R. Ledwa, C. Sichert, V. Rzehak, P. Thanigai, and B. O. Eversmann. An 82uA/MHz microcontroller with embedded FeRAM for energy-harvesting applications. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 334–336. IEEE, February 2011.

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

List of Figures

Esta es la última página.
Compilado el Monday 16th June, 2014.
http://iie.fing.edu.uy/