A HIERARCHICAL APPROACH FOR POWER MANAGEMENT ON MOBILE EMBEDDED SYSTEMS*

Arliones Stevert Hoeller Junior, Lucas Francisco Wanner
and Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration
Federal University of Santa Catarina
PO Box 476 - 88049-900 - Florianópolis, SC, Brazil
{ arliones,lucas,guto } @lisha.ufsc.br

Abstract

Mobile Embedded Systems usually are simple, battery-powered systems with resource limitations. In some situations, their batteries lifetime becomes a primordial factor for reliability. Because of this, it is very important to handle power consumption of such devices in a non-restrictive and low-overhead way. This power management cannot restrict the wide variety of different low-power modes such devices often feature, thus allowing a wider system configurability. However, once in such devices processing and memory are often scarce, the power management strategy cannot compromise large amounts of system resources. In this paper we propose a simplified interface for power management of software and hardware components. The approach is based on the hierarchical organization of such components in a component-based operating system and allows power management of system components without the need for costly techniques or strategies. A case study including real implementations of system and application is presented to evaluate the technique and shows energy saves of almost 40% by just allowing applications to express when certain components are not being used.

Keywords: Power management, energy consumption management, embedded systems, mobile computing, low-power computing, embedded operating systems.

1. Introduction

In a mobile, battery-powered embedded system, battery lifetime is a primordial factor for reliability, thus making power management a very important

*This work was partially supported by FINEP (Financiadora de Estudos e Projetos) grant no. 01.04.0903.00.

issue for those systems. Embedded systems hardware usually provides some level of support for low-power operating modes. However, current software methodologies, techniques and standards for power management often focus on general purpose systems, where processing and memory overheads are mostly insignificant. Although these techniques have shown good results [1] [2][3], they impose extra processing costs or require advanced hardware resources, thus making them unusable in restricted embedded systems where processing and memory are very scarce.

Power management standards such as APM and ACPI were created focusing personal computers. These standards require either BIOS support or enough memory and processing capabilities for running a power management virtual machine. These requirements restrict their use to powerful embedded systems, which usually feature fast processors and large amounts of memory and make use of interactive operating systems such as LINUX and WINDOWS. The Advanced Power Management (APM) design assumed that the BIOS might make decisions regarding power consumption solely on monitoring the hardware. The lack of control of the operating system over the power management features of the BIOS, e. g., when the system will change power states, and the missing information on the BIOS level about the characteristics and requirements of the applications have been identified as the main drawbacks of APM [4].

The most important and established power management interface for general purpose computing systems is Advanced Configuration and Power Interface (ACPI), released in 1996 as a replacement of the previous industry standard for power management, Advanced Power Management (APM). ACPI identifies the operating system as the entity which has comprehensive knowledge about the hardware components and their usage and about the characteristics and behavior of the applications which access these hardware components. In contrast to APM, the operating system has full control over the operating modes and power management features of the hardware. ACPI is designed to not rely on the firmware and the exact implementation of the routines to access the hardware. The key to achieve this goal is the use of the ACPI source language (ASL), which is compiled to the machine language AML, similar to JAVA bytecode. Execution of the AML code is done by an interpreter in the operating system, inside a sandbox. This approach has several advantages: The interpretation of AML code prevents erroneous or malicious code to harm the system. AML code abstracts from the operating system as well as the platform or architecture it is executed on, so the burden of supporting drivers for several different operating systems or architectures is released from the hardware manufacturers [5]. However, ACPI abstracts the operating modes of the hardware in a way which may be too restrictive for embedded systems. The four device power modes defined by ACPI (D0 – D3) may be too coarse grained for

embedded applications, once most components used in such systems usually feature several low-power operating modes. Furthermore, the use of an interpreted language to access hardware components, though having substantial advantages, poses requirements on the system which by far exceed the limited resources of most embedded devices.

In addition to these standards, several techniques were developed to allow an accurate control of power consumption for individual subsystems such as CPU, memory and I/O devices. These techniques use several strategies to define the best trade-off between performance and power consumption in each situation. For example, Dynamic Voltage and Frequency Scaling (DVFS) [6] is a strategy to slow down the CPU frequency or reduce its voltage supply and, consequently, save energy. Other strategies use event counter registers available in some architectures to identify which parts of the hardware are in use and how these parts must behave to satisfy the system needs in terms of power consumption [1]. Although good results have been achieved, heuristics used to dynamicly guide the application of such techniques also impose extra processing costs or require extra hardware resources, thus becoming mostly unusable in deeply embedded systems.

In order to enable power management in embedded systems without incurring excessive overhead, we propose a simple and uniform interface for power management of software and hardware components. The mechanism behind this interface is based on the hierarchical organization of software and hardware components, and allows consistent power state migration of individual components, subsystems or the whole system. A case study is presented to demonstrate the use of the technique on a real implementation of this strategy in our component-based embedded operating system, EPOS.

This paper is organized as follows. Section 2 introduces the system power management interface for software and hardware components. Section 3 presents an application to exemplify the use of the power management interface. Section 4 gives an overview of related work. Section 5 finalizes.

2. Power Management Interface for Software and Hardware Components

Power management policies in operating systems such as LINUX and WINDOWS dynamically analyze the behavior of applications and the system in order to determine when a hardware component should change its operating mode through an ACPI-compliant interface. However, most embedded systems cannot afford the overhead of such dynamic power management strategies. Furthermore, considering that a deeply embedded system is usually comprised by a single application, the best place to determine a power management strategy is in the application itself.

Embedded Parallel Operating System (EPOS) is a component-based, application-oriented operating system. In EPOS, high-level system abstractions, such as File, Thread, Scheduler and Communicator, are exported to applications through a component interface, and interact with the underlying hardware through hardware mediators. Through the system component hierarchy, each system abstraction and hardware mediator knows the state of its resources.

Through the definition of an uniform power management interface for system components, we allow the application programmer to change the power consumption status of each component individually. The interface is comprised by two methods: one to verify the component power state (power()) and other to change it (power(user_desired_status)). The mechanism behind this interface makes use of the hierarchical organization of software and hardware components in EPOS to allow consistent state migration among system operating modes.

Low-power hardware typically used in embedded systems often present a large set of operating modes. Enabling the use of all available operating modes is likely to enhance the system configurability, but might also increase the application complexity when managing the system power consumption. In order to solve this issue, we established a set of high-level definitions for the power consumption states, which will ease the application programmer from having to understand every hardware component in the system. As in ACPI [5], four universal modes were defined: FULL, LIGHT, STANDBY and OFF. These may be extended by system components whenever needed. When the device is fully operational, it is in the FULL state. The LIGHT state will consume less energy, but will grant the proper behavior of the device, it will probably incur in performance loss. In STANDBY, the device will have its behavior changed. This state will probably be a sleep mode. When OFF the device is switched off or switched to its smallest energy consumption state.

As embedded applications grow in complexity they make use of a large number of individual system components. As such, it may be impracticable for application programmers to take care of the power consumption of each component individually. To solve this problem we allow applications to manage individual subsystems or the system as a whole.

In order to exemplify how an entire subsystem may change its operating mode, we present a brief description of the EPOS communication subsystem. This subsystem is shown in Figure 1 and is basically comprised by four families of components: Communicator, Channel, Network and NIC. NIC is a family of hardware mediators, which abstracts the hardware device to the Network family. Network is responsible for abstracting the network (e. g., Ethernet, CAN, ATM, etc). Channel is responsible for inter-process communication and uses Network to build a logical communication channel through

which messages are exchanged. Finally, a `Communicator` is an end-point for communications.

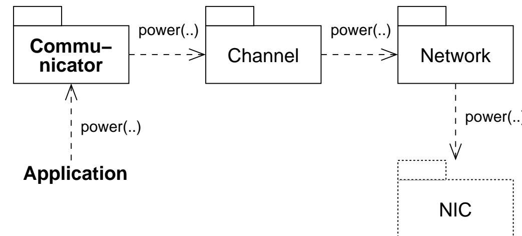

Figure 1. EPOS communication subsystem.

To grant portability of application code, the application programmer is suggested to use higher level abstractions, such as members of the `Communicator` family in the communication subsystem. In this context, our power management strategy must provide ways for the application programmer to change the power state of a communicator and this component must consistently propagate power state migrations to all software and hardware components in its hierarchy. For example, an implementation of a `Communicator` will use a `Channel` and probably an `Alarm` component to handle time-outs in the communication protocol. When the application executes a command asking the `Communicator` component to switch the operating mode to `OFF`, the `Communicator` will finish all started communications by flushing its buffers and waiting for all acknowledgment signals before shutting down other components in its hierarchy.

System-wide power management actions are handled by the `System` component in EPOS. The `System` component contains references to all subsystems used by the application. Thus, if an application wants to switch the whole system to a different operating mode, it may use the interface on the `System` component, which will propagate this request to all subsystems.

Figure 2 illustrates the system-wide power management interface may be accessed. It shows the components instantiated for a hypothetical sensing system. In this instance, the system is comprised by four components: the `CPU`, a `Communicator`, a `Sensor` and the `System` component. Each component has its own interface, which may be called by the application at anytime, and a set of power consumption levels. If the application wants to switch a specific subsystem to another power consumption level, it can access its components directly. If it wants to modify the whole system power consumption level, it may access the `System` component, which will propagate the modification through the system.

The main challenge identified on the development of power-aware components was the need for consistent operating mode propagation. This propa-

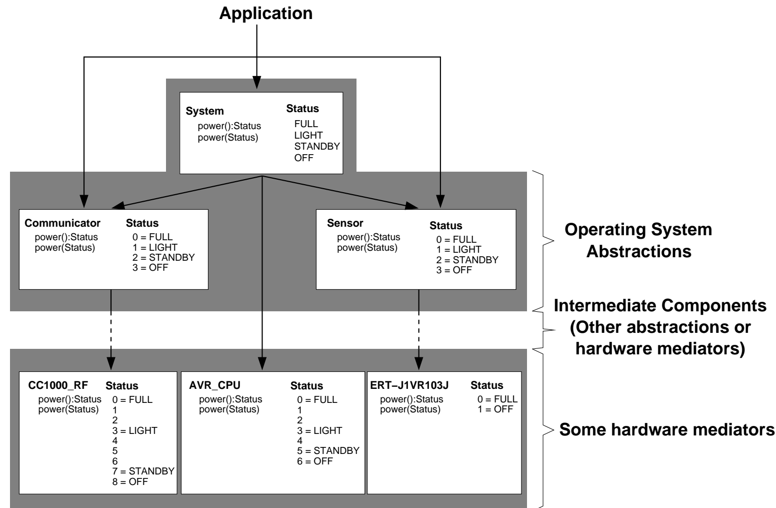

Figure 2. Accessing the power management interface.

gation must guarantee that no data will be lost and no unfinished actions will be interrupted. By letting each component handle its responsibilities (e. g., a `Communicator` flushing all its buffers and waiting for all acknowledgment signals) before propagating the power state propagation (e. g. shutting down `Alarm` and `Channel`), it is possible to guarantee consistent operating mode propagation of an entire subsystem.

In this strategy, the application programmer is expected to specify in the application when certain components aren't being used. It is done by issuing "`power`" commands to individual components, subsystems or the system. In order to free the application programmer from having to wake-up these components, such components are implemented to automatically switch on when a call is done to any of their methods. When this happens, components are switched to the their previous states or to the less energy spendable power state in which is possible to perform the required actions.

3. Case Study: Thermometer

In order to demonstrate the usability of the defined interface, a thermometer was implemented using a simple prototype with a 10 kilo ohm thermistor connected to an analog-to-digital converter channel of an Atmel ATMega16 [7] microcontroller. The embedded application is presented in Figure 3. This application uses four system components: `System`, `Alarm`, `Thermometer` (member of the `Sentient` family [8]) and `UART`. The EPOS hierarchical or-

ganization binds, for example, the Thermometer abstraction with the micro-controller's analog-to-digital converter hardware mediator.

```
System sys;
Thermometer therm;
UART uart;

void alarm_handler() {
    uart.put(therm.get());
}

int main() {
    Handler_Function handler(&alarm_handler);
    Alarm alarm(1000000, &handler);

    while(1) {
        sys.power(STANDBY);
    }
}
```

Figure 3. The Thermometer application

When the application starts, all used components are initialized by their constructors and a periodical event is registered with the Alarm component. The power state of the whole system is then switched to STANDBY through a power command issued to System. When this happens, the System component switches all system components, except for the Alarm, to *sleeping* modes. The Alarm component uses a timer to generate interrupts at a given frequency. Each time an interrupt occurs, the CPU wakes-up and the Alarm component handles all registered events currently due for execution. In this example, every two seconds the Thermometer and UART components are automatically switched on when accessed and a temperature reading is forwarded through the serial port. When all registered events are handled, the application continues normal execution on a loop which puts the System back in the STANDBY mode.

The graphics presented in Figure 4 show energy measurements for this application with and without system power management capabilities. Both graphics show the results of a mean between ten measurements. Each measurement was ten seconds long. In graphic (a) is noticed that system power consumption oscillates between 2.5 and 4 Watts. In graphic (b), the oscillation stays between 2 and 2.7 Watts. By calculating the integral of these graphics is possible to obtain energy consumption for these system instances during the time it was running. The results were 3.96 Joules for (a) and 2.45 Joules for (b), i.e., the system saved 38.1% of energy without compromising its functionality.

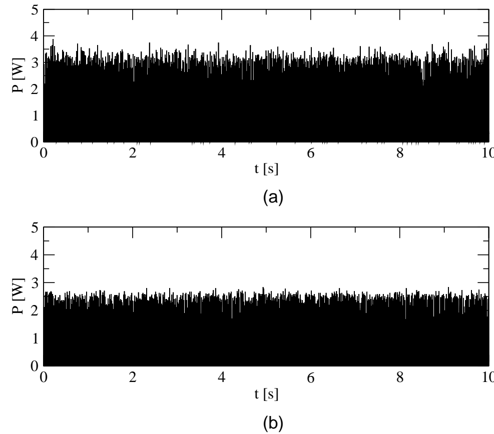

Figure 4. Power consumption for the Thermometer application *without (a) with (b)* power management.

4. Related Work

TINYOS and MANTIS are embedded operating systems focused on wireless sensor networks. In these systems energy-awareness is mostly based on low-power MACs [9, 10] and multi-hop routing power scheduling [11, 12]. This makes sense in the context of wireless sensor networks, for a significant amount of energy is spent on the communication mechanism. Although this approach shows expressive results, it often focuses on the development of low-power components instead of power-aware ones. Another drawback in these systems is the lack of configurability and standardization of a configuration interface.

SPEU (System Properties Estimation with UML) [13] is an optimization tool which takes into account performance, system footprint and energy constraints to generate either a performance-efficient, size-efficient or energy-efficient system. These informations are extracted from an UML model of the embedded application. This model must include class and sequence diagrams, so the tool can estimate performance, code-size and energy consumption of each application. The generated system is a `Java` software and is intended to run over the FEMTOJAVA [14] soft-core processor. Once SPEU only takes into account the UML diagrams, its estimations show errors as big as 85%, making it only useful to compare different design decisions. It also lacks configurability, once the optimization process is only guided by one variable, i. e., if the application programmer's design choice is performance, the system will never enter power-aware states, even if it is not using certain devices. This certainly limits its use in real-world applications.

IMPACCT (which stands for Integrated Management of Power-Aware Computing and Communication Technologies) [15] is a system-level tool for exploring power/performance tradeoffs by means of power-aware scheduling and architectural configuration. The idea behind the IMPACCT system is the embedded application analysis through a timing simulation to define the widest possible dynamic range of power/performance tradeoffs and the power mode in which each component should operate over time. This tool chain also includes a power-aware scheduler implementation for hard real-time systems. IMPACCT tools deliver a very interesting way to configure the power-aware scheduler and the power-modes of an embedded system, but is far from delivering a fast prototyping environment.

5. Conclusion

In this paper we presented an strategy to enable application-driven power management in deeply embedded systems. In order to achieve this goal we allowed application programmers to express when certain components are not being used. This is expressed through a simple power management interface which allows power mode switching of system components, subsystems or the system as a whole, making all combinations of components operating modes feasible. By using the hierarchical architecture by which system components are organized in our system, effective power management was achieved for deeply embedded systems without the need for costly techniques or strategies, thus incurring in no unnecessary processing or memory overheads.

A case study using a 8-bit microcontroller to monitor temperature in an indoor ambient showed that almost 40% of energy could be saved when using this strategy.

Acknowledgments

Authors would like to thank Augusto Born de Oliveira, Hugo Marcondes and Rafael Cancian from LISHA for very helpful discussion. We also would like to thank the Department of Computer Sciences 4 at Friedrich-Alexander Universität (Germany), its head Prof. Schröder-Preikschat and Andreas Weissel for providing equipment and some advise for this work.

References

[1] Bellosa, Frank, Weissel, Andreas, Waitz, Martin, and Kellner, Simon (2003). Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, pages 04–1 – 04–10, New Orleans, USA.

[2] Sorber, Jacob, Banerjee, Nilanjan, Corner, Mark D., and Rollins, Sami (2005). Turducken: hierarchical power management for mobile devices. In *MobiSys '05: Proceedings of the*

3rd international conference on Mobile systems, applications, and services, pages 261–274, New York, NY, USA. ACM Press.

[3] Pering, T. and Broderson, R. (1998). Dynamic voltage scaling and the design of a low-power microprocessor system. In *Proceedings of the International Symposium on Computer Architecture ISCA'98*.

[4] Intel Corp. and Microsoft Corp. (1996). *Advanced Power Management (APM) BIOS Interface Specification*, 1.2 edition.

[5] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd., and Toshiba Corp. (2004). *Advanced Configuration and Power Interface Specification*, 3.0 edition.

[6] Benini, Luca, Bogliolo, Alessandro, and Micheli, Giovanni De (1998). Dynamic power management of electronic systems. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 696–702, New York, NY, USA. ACM Press.

[7] Atmel Corp. (2004). *ATMega16L Datasheet*. San Jose, CA, 2466j edition.

[8] Wanner, Lucas Francisco, Junior, Arliones Stevert Hoeller, Polpeta, Fauze Valerio, and Frohlich, Antonio Augusto (2005). Operating system support for handling heterogeneity in wireless sensor networks. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy. IEEE.

[9] Polastre, Joseph, Szewczyk, Robert, Sharp, Cory, and Culler, David (2004). The mote revolution: Low power wireless sensor network devices. In *Proceedings of Hot Chips 16: A Symposium on High Performance Chips*.

[10] Sheth, Anmol and Han, Richard (2004). Shush: A mac protocol for transmit power controlled wireless networks. Technical Report CU-CS-986-04, Department of Computer Science, University of Colorado, Boulder.

[11] Hohlt, Barbara, Doherty, Lance, and Brewer, Eric (2004). Flexible power scheduling for sensor networks. In *Proceedings of The Third International Symposium on Information Processing in Sensor Networks*, pages 205–214, Berkley, USA. IEEE.

[12] Sheth, Anmol and Han, Richard (2003). Adaptive power control and selective radio activation for low-power infrastructure-mode 802.11 lans. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*, pages 797–802, Providence, USA. IEEE.

[13] da S. Oliveira, Marcion F., de Brisolara, Lisiane B., Carro, Luigi, and Wagner, Flávio R. (2005). An embedded sw design exploration approach based on xml estimation tools. In Rettberg, Achim, mauro C. Zanella, and Rammig, Franz J., editors, *From Specification to Embedded Systems Application*, pages 45–54, Manaus, Brazil. IFIP, Springer.

[14] Ito, S.A., Carro, L., and Jacobi, R.P. (2001). Making java work for microcontroller applications. *IEEE Design and Test of Computers*, 18(5):100–110.

[15] Chou, Pai H., Liu, Jinfeng, Li, Dexin, and Bagherzadeh, Nader (2002). Impacct: Methodology and tools for power-aware embedded systems. *DESIGN AUTOMATION FOR EMBEDDED SYSTEMS, Special Issue on Design Methodologies and Tools for Real-Time Embedded Systems*, 7(3):205–232.