Application-Driven Power Management for Embedded Systems

Arliones Stevert Hoeller Junior, Lucas Francisco Wanner and Antnio Augusto Frhlich

Laboratory for Software and Hardware Integration
P.O.Box 476, 88040900
Florianpolis - Brazil
{arliones,lucas,guto}@lisha.ufsc.br

Abstract

Deeply 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.

1. Introdução

Embedded systems are used to monitor and control different objects of interrest, such as machines, motors, electronic devices, natural habitats, etc. As these systems find encompassing usage, increases the need for them to operate in a *power-aware* fashion, i.e., to control their power consumption. However, most methodologies, techniques, and software standards for power management developed so far were conceived for general purpose systems, where processing and memory overhead are usually insignificant, and are not suited for the severely resource-limited, deeply embedded systems.

ACPI and APM are two widely used power management standards that specify interfaces between software and hardware to control power consumption. Although both are widely used in general purpose computer hardware, they require processing capabilities or hardware support that may not allow their use in embedded systems. As embedded systems hardware usually present several different low power "operating modes" with specific semantics for each device, managing power consumption in terms of standard "energy levels" for very heterogenous hardware may be too restrictive.

In addition to these standards, several techniques were developed to gather system information at runtime, and use this information to guide decisions regarding power management. These techniques are classified as *Dynamic Power Management* (DPM) [3], and include, for example, *Dynamic Voltage and Frequency Scaling* (DVFS) techniques, which dynamically reduce processor frequency or voltage whenever possible, in order to reduce power consumption [2, 10, 13].

Embedded systems devices are typically very simple and resource-limited, but may provide a wide range of configurable hardware characteristics. For example, an Atmel ATMega [5] microcontroller offers eight different operating modes, and allows fine-grain configuration of most of its devices (e.g. ADC, UART), with direct impact on power consumption. Operation in variable voltages and frequencies may also be possible with external circuitry. On the other hand, software environments for embedded systems (i.e. embedded operating systems and libraries) fall short of providing adequate power management support.

Embedded operating systems typically provide a simple *Hardware Abstraction Layer* (HAL), and offer primitive support for scheduling, file systems, communication, etc. In most of these systems, applications are expected to handle power management through a platform-specific HAL. In this scenario, application portability is compromised, as the application must adapt to the particular characteristics of specific hardware.

This work explores application-driven power management, in order to allow *power aware* operation of deeply embedded systems, without compromising application portability and without incurring excessive overhead. The goal of our power management system is to allow applications to express when certain software components are not being used, permitting the system to migrate hardware resources associated with these components to lower power levels. Several issues regarding architectural diferences between different hardware devices and cuncurrent access of hardware resources by different software components emerged from this goal. In order to deal with these these issues, our system was built upon the following structures:

Generic Power Management Interface: A generic power management interface was specified to allow changes of operating modes to have the same semantics across all system components. This interface is thus simple and uniform, is present in every system component, and contributes to application portability.

Message Propagation: In order to make full use of operating system services, as well as to allow application portability, applications are expected to use high level system components, and not simply its hardware abstraction layer. However, for appli-

cations to manage power consumption though these *software* components, a mechanism must be stablished, through which messages are propagated through the system's hardware and software component hierarchy. In order to allow system-wide power management, a global component that agregates information regarding every component currently present in the system was specified.

Formalization of changes in operating modes: In order to allow consistent migrations in the power state of a component, a formal model was specified though a Petri net system. In this model, pre- and post-conditions for state migrations of every system component were defined. Generic programming resources were used to resolve the Petri net in compilation time, avoiding the overhead of a runtime interpreter.

As próximas seções discutirão como estas estruturas foram projetadas e implementadas no sistema operacional EPOS [7]. Na seção 2, ... Até o final.

2. O gerente de consumo de energia proposto

Power management policies in conventional operating system (e.g. Linux, Windows) dynamically analise the system's behavior in order to determine when a hardware component should change its operation to a lower or higher power consumption mode. The software implementation responsible for those state migrations often rely on hardware-specific interfaces, which are exported through rigid, and sometimes incomplete APIs (application programming interfaces) or device drivers. In these environments, power managing interfaces are standardized in a very low abstraction level, closer to the actual hardware than the system's abstractions.

Most general purpose computer hardware devices implement either APM (*Advanced Power Management*) or ACPI (*Advanced Configuration Power Interface*) standard interfaces to allow power management. Although these standards share little in comon, their objective is the same: to allow devices to be turned on, off or to be put in a low power consumption mode for a certain period of time. These techniques work well in environments such as servers that frequently do not use certain resources or laptop computers, that may *suspend to disk* or turn off the system when battery charge is low. These procedures, however, are hardly ever applicable in embedded systems.

Most of the resource in power managing interfaces and techniques is focused on general purpose computer hardware (e.g. personal computers, servers, laptop and handheld computers), and many research efforts focus on managing the consumption of the main microprocessor (CPU), as these devices are responsible for most of the power consumption in these systems. In embedded systems, however, processors and microcontrollers are usually very simples, and consume little power. Most of the power consumed by these systems comes from peripheral devices. Thus, power managing for these systems must focus on fine grain techniques that conserve power from peripheral devices, while allowing the system to operate properly.

Embedded systems often have to deal with severe resource restrictions, from restricted hardware capabilities (e.g. memory, processing power) or functional requirements (e.g. availability, real-time responsiveness). Thus, most embedded systems cannot afford the cost of dynamic, active power managers. Previous research [**?**, **?**, **?**, **?**] indicates that the most efficient power managing techniques are the ones that take into consideration the behavior of the target applications for a given system. Considering that most embedded system perform specific tasks, and run a single application [**?**], we may conclude that the best place to define a power managing strategy is in the application itself.

In this paper, we present a software infra-structure that allows application-driven power management for embedded systems. We provide an uniform, hardware-independent API (*Application Programming Interface*) that allows applications to change operating modes for every component in an embedded operating system. In order to ensure correct and deterministic behavior, relations and dependencies regarding power management for every component are formalized through Petri Networks. This formalization allows high-level analisys of the power state migration procedures for every component, and stablishes a message exchange mechanism in which components coordenate to ensure consistent power state changes in subsystems (e.g. communication, processing, sensing), or the system as a whole.

This section describes DSPM, an application-driven, Deterministic Static Power Manager for embeddded systems. We stablish an *Application Programming Interface* (API) implemented by every system component, that allows changes in the component's power state. *Migration Networks* formalize the changes in operating modes of components or groups of components (subsystems), and controls components instancies, allowing the system to know every component that is currently in use and to propagate systemwide changes in operation modes.

2.1 Power Managing Interface for Software and Hardware Components

In our power management strategy, the application programmer is expected to specify in his source code, whenever ceitain components will not be used. Thus, an uniform API to allow power management was defined. This interface allows interaction between the application and the system, between system components and hardware devices, and directly between application and hardware. In order to free the application programmer from having to *wakeup* components whenever they are needed, the power managing mechanism abstracted by this interface ensures that components return to their previous operational states whenever they are used.

Figure 1 presents all these interaction modes in a hipothetical system instance. The application may access a global component (`System`) that has knowledge of every other component in the system (in this case `IPC`, `Processing`, `Sensing` and their respective underlying components), triggering a system-wide power mode change. Annother way the application may use this interface is through subsystems (e.g., *Inter-Process Communication* (`IPC`), `Processing`, `Sensing`). In this way, messages are propagated only to the components used in the implementation of each subsystem. The application may also acess the hardware directly, using the API available in the device drivers, such as *Network Interface Card* (`NIC`), `CPU`, `Thermistor`. The API is also used between the system's components, as the message exchanges between `System` and the three subsystems in the figure illustrates.

In order to attain application portability, and to facilitate application development, the power managing interface was defined with a minimal set of methods and universal operating modes with unified semantics thoughout the system. Portability comes from the fact that the application doesn't need to implement specific procedures for each device in order to change its operating mode. These procedures are abstracted by the API. Easiness of use comes from the fact that the application programmer doesn't need to analyse specific hardware manuals in order to indentify available operating modes, the procedures to change those modes, and the consequences of these changes.

Two methods are defined in the API: one to change the operating mode, and another to identify the current mode. In addition to these methods, the API includes a list of modes available to each component. This list does not have a fixed size, as each component must enumerate in it every operating mode available. Low-power hard-

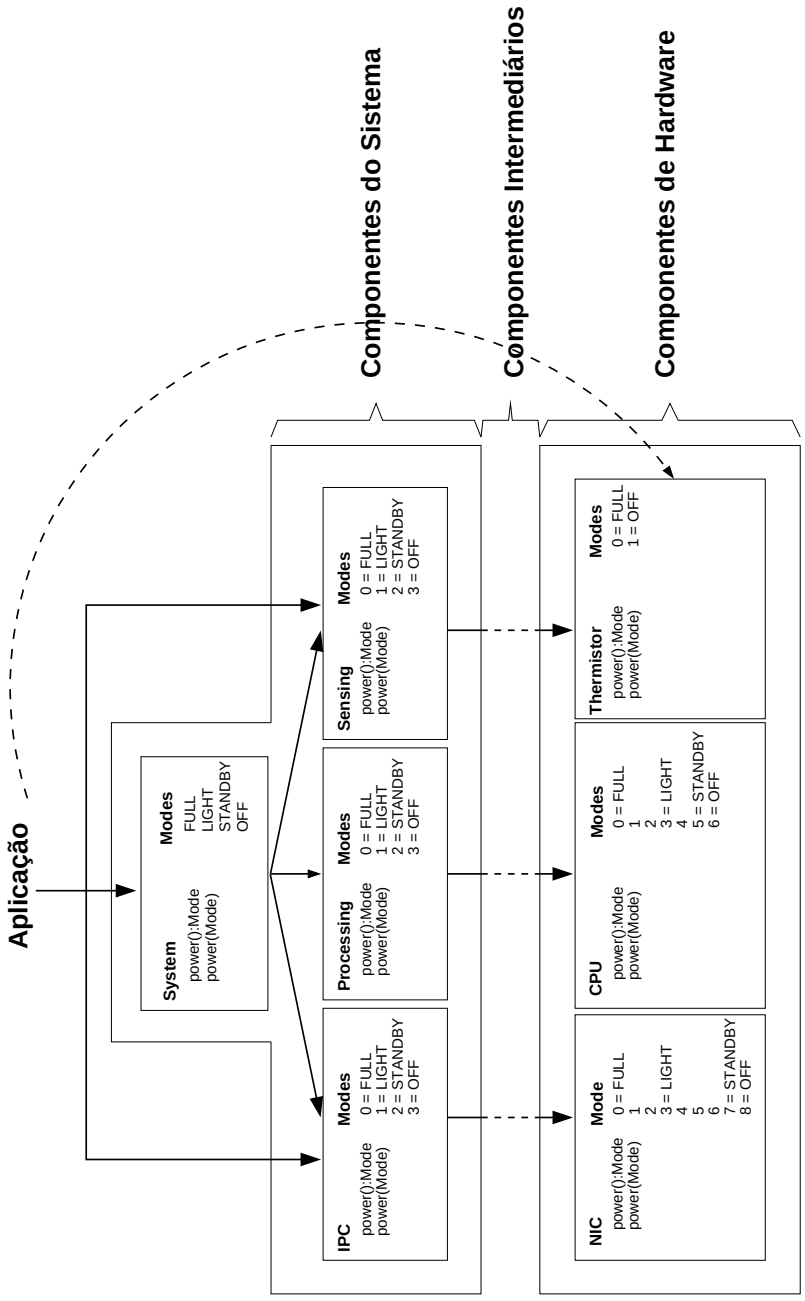

Figure 1. Power Manager API

ware components often present a wide range of operating modes. Allowing every mode to be used increses system configurability, but may increase application complexity and compromise portability. In order to deal with this issue, a set of high level universal operations was defined: `FULL`, `LIGHT`, `STANDBY` and `OFF`. These modes free the programmer from having to know details regarding the modes available hardware components in the system. However, these modes may be extended as necessary. It is up to the application programmer to associate universal modes to specific modes available to hardware devices.

When the device is operating at full capacity, it is in the `FULL` mode. In this operation mode, the system configures the device to operate providing its service in the most efficient manner possible, including all its functionalities, but at full power consumption. The `LIGHT` mode puts the device in an operating mode where it offers most of its functionalities, but consumes less power than the full mode and, very likely degrades its performance. Examples of this mode include devices that allow operation in different voltage supplies or frequencies (DVFS - *Dynamic Voltage and Frequency Scaling*). The migration from the `LIGHT` mode to the `FULL` mode is fast, and usually does not imply in considerable delay for the application.

In the `STANDBY` and `OFF` modes, the device stops operating. When in `STANDBY`, however, the device is ready to continue operating when necessary, and is able to continue its operation from the point before it was stoped. An exemple of such a mode is the

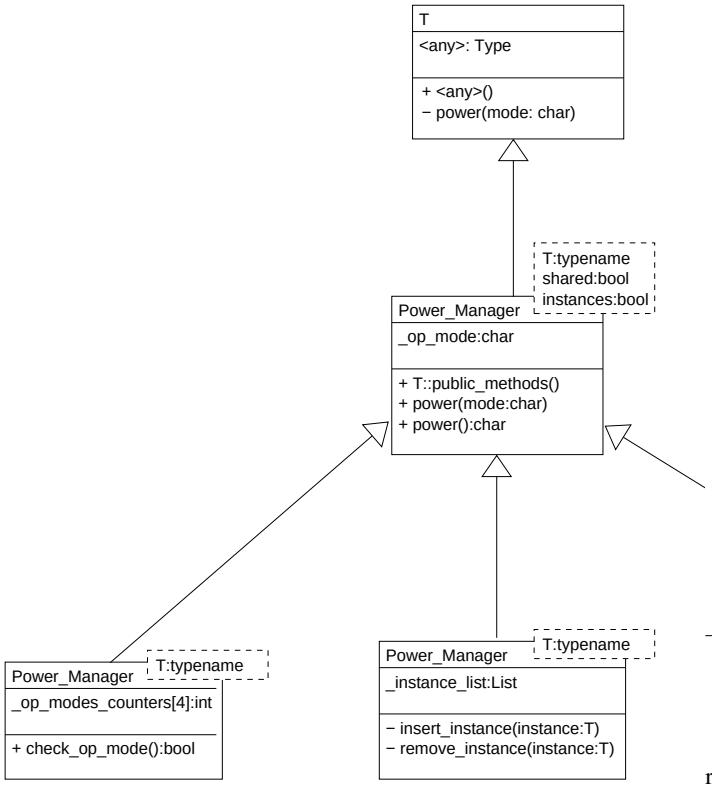

Figure 2. UML Diagram for the Power Management aspect

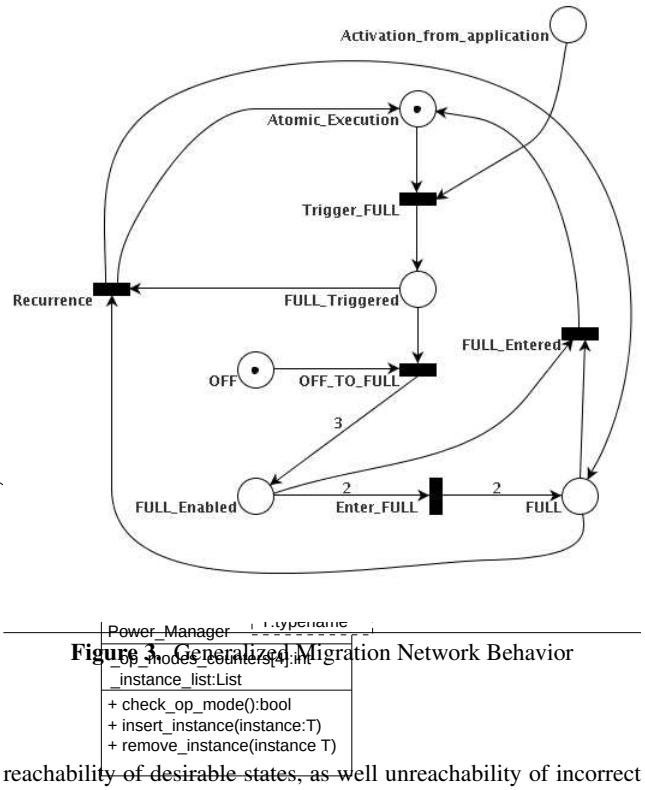

Figure 3. Generalized Migration Network Behavior

"sleep" modes of a processor. Altough in this mode the device is stopped, it still consumes a small amount of power. This power is required to keep the device's internal memory and registers alive until it is restarted. In the `OFF`, however, the device is turned off, and loses its internal configuration. When a component migrates from an `OFF` state to another state, it is reset.

In addition to the functional requirements, the API should be easily mantainable and appliable to existing systems, As power management is a non-functional requirement for operating systems [**?**], our power management API was modeled as an aspect [11], and may thus be isolated from the rest of the system. Figure 2 presents an UML diagram for this aspect, representing also its dependancies to other system components.

In order for a power management strategy to be properly abstracted as an aspect, this strategy must be scenario-independent, and appliable to any component. There is no generic method to implement the migration between different operating modes, as these procedures are dependant to the particular characteristics of different devices. Thus, each system component must implement a method to allow changes in its power state. This method must be private, and innacessible from the application and other components whenever the power management aspect has not been applied to the system.

2.2 Operation mode migration networks

In order to map coherent conectivity between different abstraction levels in the system, a formal operating mode migration network was defined. In this section, we describe this formal mechanism, which was defined through Petri networks. These networks feature clear graphical representation, and a wide range of mathematical analisys models [12]. These models allow proof of liveness and reachability of desirable states, as well unreachability of incorrect states.

Altough the procedures to migrate power states are specific to each component (both software and hardware), the control and dispatch of these migrations may be expressed in a generic form. In order to allow that, a network of mode migrations that specifies the transitions between different operating modes was formalized. Figura 3 presents a simplified overview of this network, illustrating the migration of a component from the OFF to the FULL mode. As illustrated in the figure, there are places associated with the existing operating modes (FULL and OFF). A resource in these places marks the component's current operating mode.

The `Atomic_Execution` place is responsible for ensuring that different mode change operatiions do not execute simultaniously. For that, this palce is always initialized with one resource. This resource enables the transactions that enable changes in operating mode. The moment this transaction is triggered (through a function call to the power management API), the transactions that would start different migrations are disabled, as the resource in the `Atomic_Execution` place is consumed. Additionallym a new resource inserted into the `Triggering_FULL` place enables the transactions that remove the resources that marks the component's current operating mode (OFF). As the component in the example is in the OFF state, only the `OFF_TO_FULL` transaction is enabled. When this transaction is triggered, the resource that marked the `OFF` place is conumed, and three resources are inserted into the `FULL_Enable` place. This anables the `Enter_FULL` transaction, that is responsible for executing the operations that actually change the component's power mode. After this transaction is triggered, two resources are inserted into the `FULL` place, anebling the `FULL_Entered` transaction, which finalizes the process, consuming the final resource in the `FULL_Enable` place, and inserting one resource back into the `Atomic_Execution` place. The entire process results with a resource removed from the `OFF` place and inserted into the `FULL` place. In order to avoid deadlocks when transactions that result in the component's current operating mode, a `Recurrence` transaction was inserted into the

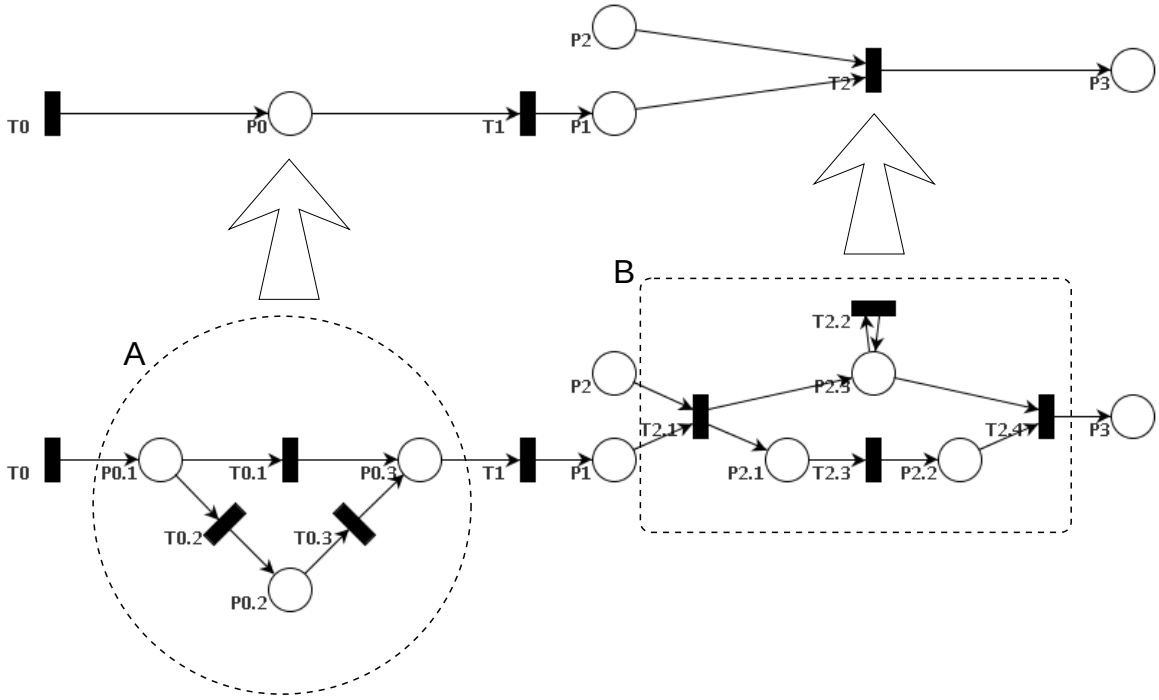

Figure 4. Hieralquical Petri Network

model. This transaction returns the resource removed from the `Atomic_Execution` place in case of recurrency.

This Petri network was analysed through tradditional Petri net tools, and was found to be deadlock free, and to have finite reachability.:

The generalized network represents the transitions of operating mode from a high level perspective, where the particular characteristics involved in the transition of each component are not specified. However, a refinement process is required in order to allow the inferrence of the migration proderes from this network model. This refinement explores the hierarquical characteristic of Petri nets, which allows an entire network to be replaced by a place or transaction in order to model a higher level abstraction and, on the other hand, allows places and transactions to be replaced with sub-networks in order to provide a refined, more detailed model. Figure 4 presents the notation for this representation. In this example, the higher abstraction network `P0` abstracts the `A` sub-network, and the `T2` abstracts the `B` sub-network.

In order to refine the migration procedures responsible for migrating the operating mode, the `Enter` transactions are replaced by sub-networks that implement the migration procedures in higher detail. Figure 5 presents the sub-network that implements the migration of the `B-MAC` component to the FULL operating mode. In order to form the migration network for this component, this subnetwork replaces the `Enter_FULL` transaction in the general migration network. This subnetwork also presents transactions that abstract the triggering of transactions that change the operating mode of other components.

2.3 Message Propagation

As the complexity of embedded application increases, more system components are used. Thus, the control of individual components' power consumption by the application may be inpracticable. For example, figure 6 presents a hipothetical *power-aware* application. The application implements a remote monitoring module, that periodically samples a pressure sensor, and sends the value read through a GPRS modem. Figure 6(a) illustrates the complexity resulting from controlling individual components. In this example, the application must stop the TCP/IP communication stack prior to turning off a modem, i.e. all pening data must be sent before the communication may be stopped. After the modem is turned off, the application turns off the serial ports (UART) used to communicate with the modem. Similar complexities are present in almost every subsystem. Abstracting these details enhances the usability of the power management API, as figures 6(b) and 6(c) present.

In order for a subsystem to be deactivated or migrated to low-power operating modes in an efficient manner, it is necessary to ensure that the software and hardware artifacts first finalize operations currently executing, or adapt to the new operating parameters. It is also necessary to ensure that these subsystems operate correctly after returning to their funcional operating modes. Thus, a mechanism and a policy for interaction between components must be established.

Given the presented API and migration networks, it is simple to infer the migration procedures for each subsystem. The interaction mechanism is thus formed by message exchanges through the API. The policy may be derived from the migration networks for each subsystem. This policy forms the correct sequence for the migration of each component. Figure 5,presents transactions that trigger migrations in the networks of other components (e.g., `Radio.Trigger_FULL`). These transactions are the points in which messages are exchanged between components.

A partir da interpretação destas redes é possível a montagem, em tempo de compilação, dos métodos que garantirão a sequência correta de execução dos procedimentos de migração. No exemplo da Figura 5 pode-se observar a conexão de três outras redes de migração à rede do `B-MAC` (`Timer`, `SPI` e `Radio`). O `B-MAC` é uma implementação em software de um MAC (Media Access Control) para um módulo de rede de sensores sem fio [**?**]. Neste dispositivo, a comunicação entre o processador e

```
int main() {
  while(1) {
    modem_send(dest, sensor_read());
    alarm(120000000, NO_BLOCK);
    tcpip_stack_standby();
    modem_standby();
    uart_standby();
    sensor_standby();
    adc_standby();
    process_standby();
    scheduler_standby();
    cpu_standby();
  }
}
```

(a) Controlando todos componentes

```
int main() {
  while(1) {
    modem_send(dest, sensor_read());
    alarm(120000000, NO_BLOCK);

    modem_standby();

    sensor_standby();

    process_standby();


  }
}
```

(b) Controlando subsistemas

```
int main() {
  while(1) {
    modem_send(dest, sensor_read());
    alarm(120000000, NO_BLOCK);
    system_standby();
  }
}
```

(c) Controlando todo o sistema

Figure 6. Aplicações hipotéticas com gerência do consumo de energia dirigido pela aplicação

o rádio é realizada através de um barramento serial (SPI). Neste exemplo, espera-se que a aplicação utilize a API do componente `B-MAC` como interface do subsistema de comunicação. Ao executar um comando "`MAC.power(OFF)`", por exemplo, o desligamento do subsistema de comunicação deve iniciar pelo desligamento do próprio `B-MAC`, que deve esvaziar seus buffers de envio e desligar o `Timer` que utiliza para recepções antes de requisitar que o rádio se desligue. Como as redes de migração foram organizadas de modo hierárquico, o resultado final da geração do procedimento de migração do subsistema de comunicação seria um procedimento algoritmico como o representado na Figura 7.

Propagação para todo o sistema

Ações de gerência do consumo de energia do sistema como um todo são tradadas por um componente global do sistema (`System`). Este componente contém referências para todos os subsistemas em uso pela aplicação. Então, se uma aplicação deseja alterar o modo de operação do sistema inteiro, isto pode ser feito acessando a API deste componente, que propagará este pedido para todos os subsistemas. Esta lista deve ser montada em tempo de execução através do aspecto de gerência de energia, que utilizará as chamadas de construção e destruição de componentes para, respectivamente, incluir e remover referências a instâncias de componentes desta lista. Quando a API de gerência do consumo de energia do sistema é acessada pela aplicação, o sistema realiza uma varredura pela lista de instâncias que possui, disparando chamadas às APIs dos componentes que registrou.

2.3.1 Compartilhamento de recursos

O compartilhamento de recursos é uma característica de sistemas computacionais que precisa ser tratada nesta proposta. Problemas podem ocorrer na migração de modos de operação quando componentes de alto nível compartilham, o mesmo componente de hardware. Por exemplo, uma aplicação que utiliza dois sensores que compartilham o mesmo conversor analógico-digital (ADC) multiplexado não pode ter o ADC desligado devido à solicitação de um dos sensores se o outro sensor ainda o está utilizando.

Para resolver este problema, adotou-se um mecanismo de *contadores de uso*. Cada componente compartilhado possui contadores que indicam quantos componentes solicitam cada modo de operação. Sempre que uma chamada é realizada à API, o contador referente ao estado atual do componente é decrementado, e o contador referente ao estado pretendido é incrementado. A migração solicitada é realizada sempre que a maioria absoluta das referências estiverem contabilizadas em um único contador.

3. Implementation

This section will present the power manager's implementation. This includes the message propagation mechanism and the definition of a descriptive language to represent the operating modes nets. This language will be used to enable automatic generation of the operating mode switching methods. Therefore, the experimental environment, which is the EPOS [**?**] operating system, is briefly described.

3.1 Static resolution of the Operating Modes Nets

The Operating Mode Nets presented at section 2.2 offer to this proposal a way to specify the operating mode switching procedures. Although there is a lot of mathematical analysis models to interpret these nets at execution time, these models demand for processing and memory capabilities well beyond of embedded devices' capacities. However, once in this proposal the nets and the application are known at system generation time, online interpretation of these nets is not needed, and can be eliminated. The elimination is done by analyzing the nets at compilation time, thus generating the operating mode switching methods accordingly to the system configuration.

An open-source graphical Petri Net editor called Pipe2 [1] was used to generate the operation mode nets. This tool was then modified to export the operation mode nets as a XML based language which feeds an analysis tool with information about the operating mode transition procedures. At system generation time, this

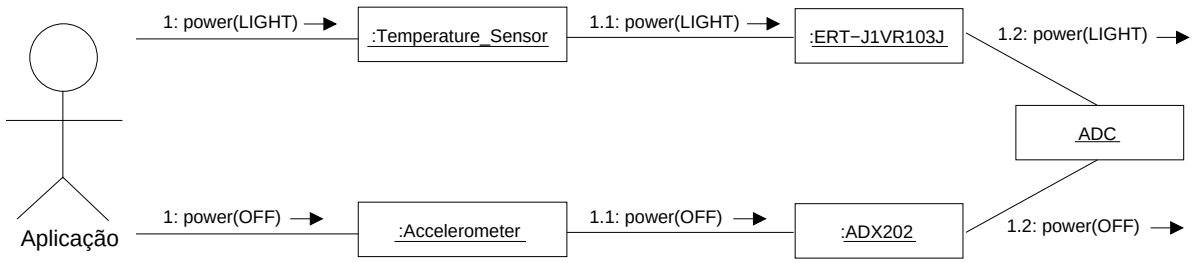

Figure 8. ADC sendo compartilhado por dois sensores.

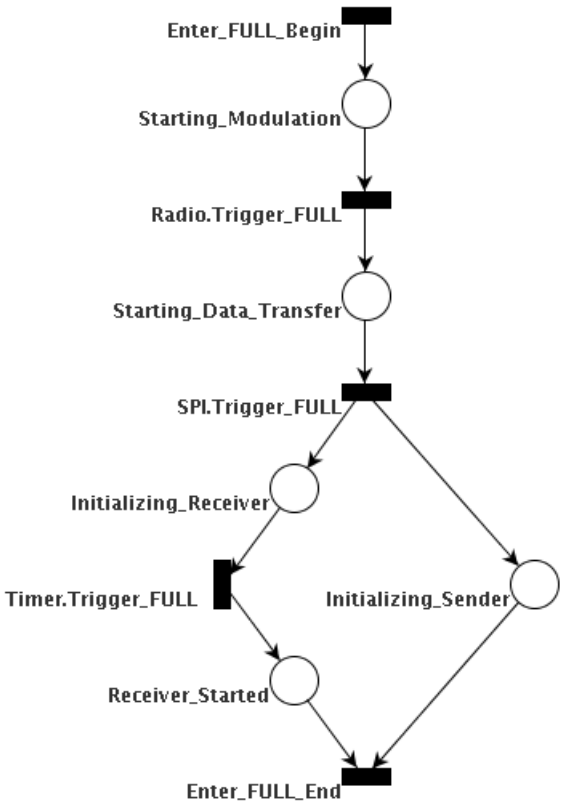

Figure 5. Sub-network implementing the migration procedures for the `B-MAC` component.

```
void MAC::power(int mode) {
  switch(mode) {
  // ...
  case STANDBY:
    this->flush_buffers();
    timer->power(STANDBY);
    this->disable();
    radio->power(STANDBY);
    break;
  // ...
  }
}

void Timer::power(int mode) {
  switch(mode) {
  // ...
  case STANDBY:
    this->disable();
    break;
  // ...
  }
}

void Radio::power(int mode) {
  switch(mode) {
  // ...
  case STANDBY:
    this->disable();
    break;
  // ...
  }
}
```

Figure 7. Métodos de migração de modo de operação derivados da redes de migração

tool uses system information about which components are in use (EPOS provides this information after an application analisys [18]) to generate the operating mode switching procedures which will be needed. These procedures are aggregated by an aspect, which is applied to the components which power consumption must be handled. This aspect aggregate some data to control the power consumption of the components it is applied to and, also, adds code to garantee that components will be in an operating mode in which the requested operation can be executed. Information about in which operating mode each system interface method works was included in EPOS' information database.

The aspect was implemented as a *scenario adapter*, i.e., a static meta-programmed construct (C++ template class) which extends the components (C++ classes) over which it will be applied, aggregating data by declaring its variables, and modifying code by using polymorfism. As template dependencies are solved at compilation time in C++, the polymorphism haven't aggregated aditional overhead to the system. This happens because, once power management is switched on for a certain component, the system configuration prevents the application from calling the original version of the component, incurring in the existence of only one used method.

Figure 9(a) shows a source code fragment of the Power_Manager aspect. There are two configurable features for this manager: *shared* and *instances*. *shared* is used to inform that there is the possibility of the managed component be shared. When this happens, the power manager tracks the component's users, preventing itself from switching off in use components. *instances* enables a funcionality which keeps track of components instances. It is used to allow message propagation from higher level components. These features are configured by the system or by the application programmer, and the aspect is specialized to operate accordingly to this configuration. Figures 9(a), 9(b), and 9(c), shows the methods which were automatically generated for a UART components, i.e.,

```
template<typename T,
    bool shared = Traits<T>::shared,
    bool instances = Traits<T>::instances>
class Power_Manager;

template<typename T>
class Power_Manager<T,false,false>
    : public T {
public:
    //T::Constructors
    Power_Manager() : T(){}
    Power_Manager(unsigned int a) : T(a){}

    // ...

    // T::public_methods();
    char get();
    void put(char c);

    // ...

public:
    void power(char mode);
    char power(){ return _op_mode; };

private:
    char _op_mode;
    char _prev_op_mode;
};

// ...
```

(a) Power_Manager aspect declaration

```
template<>
void Power_Manager<ATMega128_UART,
    Traits<ATMega128_UART>::shared,
    Traits<ATMega128_UART>::instances
>::power(char mode) {

    _prev_op_mode = _op_mode;
    _op_mode = mode;
    switch(mode) {
    case FULL:
        this->tx_enable();
        this->rx_enable();
        break;
    case LIGHT: //Send-Only
        this->tx_enable();
        this->rx_finish();
        this->rx_disable();
        break;
    case STANDBY: //Receive-Only
        this->tx_flush();
        this->tx_disable();
        this->rx_enable();
        break;
    case OFF:
        this->tx_flush();
        this->tx_disable();
        this->rx_finish();
        this->rx_disable();
        break;
    }
}
```

(b) Automatically generated `power` method

```
template<>
char Power_Manager<ATMega128_UART,
    Traits<ATMega128_UART>::shared,
    Traits<ATMega128_UART>::instances
>::get() {
    if(!((_op_mode == STANDBY)
    || (_op_mode == FULL))) {
        if((_prev_op_mode == STANDBY)
        || (_prev_op_mode == FULL))
            power(_prev_op_mode);
        else
            power(FULL);
    }

    return ATMega128_UART::get();
}
```

(c) Wrapped interface method

```
template<>
void Power_Manager<ATMega128_UART,
    Traits<ATMega128_UART>::shared,
    Traits<ATMega128_UART>::instances
>::put(char c) {
    if(!((_op_mode == LIGHT)
    || (_op_mode == FULL))) {
        if((_prev_op_mode == LIGHT)
        || (_prev_op_mode == FULL))
            power(_prev_op_mode);
        else
            power(FULL);
    }

    ATMega128_UART::put(c);
}
```

(d) Wrapped interface method

Figure 9. Fragment of the Power_Manager aspect

the operating mode switching procedure (*power*) and the wrapping methods for the UART's interface methods *get* and *put*.

3.2 Ambiente Experimental

To test this proposal, this power manager was adapted to operate over the EPOS operating system [7]. Implementations of this system for AVR microcontrolers were used. Also, support for other devices featured by some used platforms featured was developed in cooperation with a parallel work which explored operating system support for wireless sensor networks [**?**]. Among these devices are communication devices (UART and radio MACs) and sensing devices (thermistores, photo-diodes, accelerometers, anolog-digital conversors and high-level abstractions for sensing).

3.2.1 EPOS - Embedded Parallel Operating System

The EPOS operating system was proposed by Fröhlich in his PhD thesis as a prototype to prove the concepts behind his Application-Oriented System Design methodology (AOSD) [7]. This methodology uses several advanced software engineering and programming techniques that, combined, enable the generation of optimized operating systems for dedicated applications. Since its creation, EPOS have been used as platform to validate and extend AOSD's concepts and, recently, has also been successfully used as a hardware design methodology [**?**]. This methology has evoluted and EPOS is becoming a complete solution for software/hardware co-design of embedded systems [**?**].

The main goals of the EPOS system are to allow application programmers to write architecture-independent applications, and, through the application analysis, to deliver a run-time software support for such applications which complies all resources that a specific application needs, and nothing else. In order to achieve these goals, EPOS relies on AOSD's concepts of *Inflated Interfaces*, *Hardware Mediators* and *System Abstractions*.

Hardware mediators are software constructs that mediate the interaction between operating system components, called *System Abstractions*, and hardware components. This mediation is done through the definition of a rigid interface for each group (family) of hardware components. This interface is called *Inflated Interface*, and it is responsible for defining the main difference between hardware mediators and HALs. Instead of building a monolithic layer encapsulating the resources available in the hardware platform, each hardware mediator abstracts the correspondent hardware component functionalities and deliver these functionalities to the operating system through the inflated interface. As hardware mediators are intended to be mostly meta-programmed, their code is dissolved in the abstractions as the "interface contract" is met, generating virtually no overhead to the system [15].

Each system abstraction or hardware mediator is composed by a set of similar operating system components. These components are organized according to the *Family-Based Design* paradigm, and have their *commonalities* and *variabilities* explored through different class hierarchies. For instance, the Thread family of system abstractions is comprised by several different thread implementations (e.g., Exclusive_Thread, RT_Thread), and the Scheduler family is comprised by several schedulers (e.g., FCFS_Scheduler, EDF_Scheduler, RM_Scheduler). Composition rules help a graphical tool to suggest for the application programmer a running environment for each application. An example of composition rule would be the mandatory use of the RT_Thread member when a real-time scheduler (e.g., Erliest Deadline First (EDF) and Rate Monotonic (RM)) is chosen.

System Abstractions and Hardware Mediators are intended to be collected from an *Application-Oriented Domain Analysis and Decomposition* process. This analysis process is quite similar to *object-oriented decomposition*. The main difference is that the Application-Oriented System Design is a multi-paradigm design methodology, so other entities, such as *aspects* and *configurable features*, must come out from this analysis. The use of *configurable features* and *scenario aspects*, allied to advanced programming techniques, such as *Static Meta-programming* and *Aspect-Oriented Programming*, deliver to the application programmer a widely configurable and adaptive system.

4. Case study

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 [5] microcontroller. The embedded application is presented in Figure 10. This application uses four system components: System, Alarm, Thermometer (member of the Sentient family [19]) and UART. The EPOS hierarchical organization binds, for example, the Thermometer abstraction with the microcontroller's analog-to-digital converter hardware mediator.

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.

```
#include <system.h>
#include <sentient.h>
#include <uart.h>
#include <timepiece.h>

using namespace System;

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 10. A aplicação Thermometer

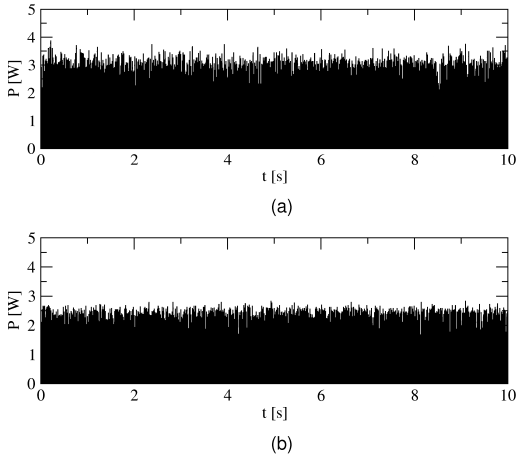

Figure 11. Consumo de energia para a aplicação Thermometer *sem (a)* e *com (b)* gerenciamento do consumo de energia.

The graphics presented in Figure 11 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.

	.data	Overhead	.code	Overhead
Base system	183	0%	9,596	0%
Managing (UART)	186	1.64%	10,302	7.36%
Managing all	189	3.28%	10,338	7.73%

Table 1. System footprints. Sizes in bytes.

Table 1 shows system footprint sizes without power management, managing only one component (UART) and managing all components. As can be seen, only 6 bytes of data and 742 B of code were added to the system. This overhead includes the code and data necessary to handle the operating mode nets.

5. 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 [14, 17] and multi-hop routing power scheduling [8, 16]. 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) [6] 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 [9] 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.

CIAO (CIAO is Aspect-Oriented) is a project which aims the development of a fine-grained product-line operating system for embedded and deeply embedded systems. It focuses the abstraction of non-functional properties as aspects, enhancing system configurability. One of these non-functional properties is energy consumption. CIAO is still in an early development stage, but already shows to be somewhat different from the EPOS approach. EPOS also uses aspects to abstract some non-functional properties but, while CIAO uses aspect-orientation as the whole system design methodology, EPOS uses it as an extra development tool instead. In the particular case of power-awareness, it is not desirable to model it as an aspect because, even being a non-functional property, it will impose functional consequences for the system, i. e., migration for lower power consumption modes often change the system behavior.

IMPACCT (which stands for Integrated Management of Power-Aware Computing and Communication Technologies) [4] 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.

6. 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, Rafael Cancian and, specially, Prof. Jlio Zseremetta from Federal University of Santa Catarina for very helpful discussion.

References

[1] Nadeem Akharware. Pipe2: Platform independent petri net editor. Technical report, Imperial College of Science, Technology and Medicine, London, UK, Sep 2005.

[2] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. 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, Sep 2003.

[3] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. 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, 1998. ACM Press.

[4] Pai H. Chou, Jinfeng Liu, Dexin Li, and Nader Bagherzadeh. 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, Oct 2002.

[5] Atmel Corporation. *ATMega16L Datasheet*. San Jose, CA, 2466j edition, Oct 2004.

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

[7] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

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

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

[10] Russ Joseph, David Brooks, and M. Martonosi. Live, runtime power measurements as a foundation for evaluating power/performance

tradeoffs. In *Workshop on Complexity Effectice Design WCED, held in conjunction with ISCA-28*, June 2001.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[12] James L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, Sep 1977.

[13] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, New York, NY, USA, 2001. ACM Press.

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

[15] Fauze Valerio Polpeta and Antônio Augusto Fröhlich. Portability in component-based embedded systems. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 271–280. Springer, Aizu, Japan, sep 2004.

[16] Anmol Sheth and Richard Han. 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, May 2003. IEEE.

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

[18] Gustavo Fortes Tondello and Antônio Augusto Fröhlich. On the automatic configuration of application-oriented operating systems. In *Proceedings of the 3rd ACS/IEEE International Conference on Computer Systems and Applications*, pages 120–123, Egypt, jan 2005.

[19] Lucas Francisco Wanner, Arliones Stevert Hoeller Junior, Fauze Valerio Polpeta, and Antonio Augusto Frohlich. 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, Sep 2005. IEEE.