Gerenciamento de Energia em Sistemas de Sensoriamento Remoto

Arliones Stevert Hoeller Junior[1], **Lucas Francisco Wanner**[1], **Augusto Born de Oliveira**[1], **Roger Kreutz Immich**[1], **e Antônio Augusto Fröhlich**[1]

[1]Laboratório de Integração Software/Hardware
Campus Universitário - UFSC
P.O.Box 476, 88040-900
Florianópolis, Brasil

{arliones,lucas,augusto,roger,guto}@lisha.ufsc.br

Abstract. *Remote sensing 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. In this paper we propose a simplified interface for application-driven power management of software and hardware components in a hierarchically organized, component-based operating system. This method allowed power management of system components without the need for costly techniques or strategies. A case study which implements a thermometer for indoor environments showed energy savings of almost 40% by just allowing applications to express when certain components are not being used.*

Resumo. *Sistemas de sensoriamento remoto são normalmente simples, com limitações de recursos e alimentados por baterias. Em algumas situações, a vida útil destas baterias torna-se um fator primordial para a disponibilidade destes sistemas. Devido a isso, é muito importante tratar o consumo de energia destes dispositivos de uma maneira não-restritiva e eficiente. Neste artigo é proposta uma interface simplificada e uniforme para permitir que a aplicação dirija o gerenciamento do consumo energia de componentes de software e hardware em um sistema operacional baseado em componentes hierarquicamente organizados. Este método permitiu a gerência do consumo de energia sem a necessidade de técnicas ou estratégias custosas em termos de processamento ou memória. Um estudo de caso implementando um termômetro para ambientes fechados mostrou economias de energia de até 40% apenas por permitir que aplicações expressem quando certos componentes não estão mais sendo usados.*

1. Introdução

Sistemas de sensoriamento são plataformas computacionais utilizadas para monitorar e/ou controlar os ambientes nos quais estão inseridos. Estes ambientes podem ser máquinas, motores, dispositivos eletrônicos, ambientes físicos (e.g, módulos de sensoriamento em uma rede de sensores sem-fio monitorando um habitat), etc. Neste contexto, é muito importante que estes sistemas sejam *power-aware*, i.e., capazes de gerenciar seu consumo

de energia, possibilitando a diminuição do consumo e o controle do aquecimento. Contudo, a maioria das metodologias, técnicas e padrões de software para gerenciamento de energia não se mostram viáveis em sistemas embarcados de sensoriamento com recursos limitados. Isto ocorre porque aquelas estratégias foram concebidas focando sistemas de propósito geral, onde sobrecargas de processamento ou memória são geralmente insignificantes. Além disso, a grande variedade de componentes de hardware existente para sistemas de sensoriamento exige um sistema de suporte de execução adequadamente projetado para liberar o programador de dependências arquiteturais.

ACPI e APM são os padrões de gerência de energia mais utilizados pela indústria hoje. Embora muito usados em sistemas de propósito geral, eles impõem requisitos de recursos adicionais de hardware ou capacidade de processamento que podem impossibilitar seu uso em sistemas embarcados. Como componentes de sistemas embarcados geralmente apresentam vários modos de operação para baixo consumo de energia, o uso de tais padrões pode ainda se tornar muito restritivo. Em conjunto a estes padrões, outras técnicas foram desenvolvidas para tratar o consumo de energia de sistemas eletrônicos. A maioria deles são classificados como dinâmicos (*Dynamic Power Management* - DPM) [Benini et al. 1998]. Estas técnicas reúnem informação através da análise do comportamento do sistema, e usam esta informação para guiar as decisões acerca do gerenciamento de energia. Exemplos de técnicas DPM são técnicas de *Dynamic Voltage and Frequency Scaling* (DVFS), que, dinamicamente, alteram a fonte de tensão e/ou frequência do processador para diminuir o consumo [Bellosa et al. 2003, Joseph et al. 2001, Pillai and Shin 2001].

Embora simples, dispositivos de sensoriamento também permitem gerenciamento de energia provendo diferentes modos de operação e uma grande gama de características configuráveis do hardware. Entretanto, a grande variedade de componentes de hardware existente para sistemas de sensoriamento faz com que estes sistemas tenham características bastante heterogêneas. Sendo assim, uma aplicação desenvolvida para uma dada plataforma de sensores dificilmente será portável para outra diferente, a menos que o sistema de suporte de execução nestas plataformas entregue mecanismos adequados para abstração e encapsulamento da plataforma de sensoriamento. Além das diferenças arquiteturais, módulos de sensores (e.g. sensor de temperatura, luz ou movimento) apresentam uma gama ainda maior de variabilidade. Módulos de sensores que apresentam a mesma funcionalidade muitas vezes tem diferentes interfaces de acesso, características, parâmetros operacionais e capacidades de operação *power-aware*. De fato, hardware para sistemas de sensoriamento suportam gerenciamento de energia, mas os ambientes de software existentes (sistemas operacionais e bibliotecas para sistemas embarcados) não provêm suporte adequado para este fim. Um sistema de suporte de execução adequadamente projetado poderia abstrair estas dependências arquiteturais, promover portabilidade entre diferentes plataformas de sensoriamento, e permitir gerência de energia. Um subsistema de sensoriamento de alto nível poderia fornecer acesso transparente a famílias de dispositivos sensores. Neste sentido, nós propomos uma interface de software/hardware capaz de abstrair famílias de dispositivos sensores de forma uniforme. Definimos classes de dispositivos sensores baseados em sua funcionalidade (e.g. medir aceleração ou temperatura), e estabelecemos um substrato comum para cada classes. Um mecanismo de auto-descrição baseado em software permite que aplicações usem características estendidas de sensores específicos, e agrega informações para gerência de energia.

Neste artigo nós exploramos gerência de energia dirigida pela aplicação para permitir o controle do consumo de energia em sistemas embarcados de sensoriamento sem implicar em sobrecarga excessiva. Nesta estratégia, a portabilidade do código da aplicação é garantida através do uso de *mediadores de hardware* [Polpeta and Fröhlich 2004] para criar uma interface de software/hardware capaz de abstrair famílias de dispositivos sensores de forma uniforme, e da organização hierárquica pela qual os componentes de software e hardware foram organizados. Gerência de energia dirigida pela aplicação foi possível permitindo que as aplicações expressem quando certos componentes de software ou hardware não estão mais sendo utilizados. Isto é feito através de uma interface simples e uniforme presente em todos os componentes do sistema. Quando a aplicação desliga um componente, este componente permanece desligado até que seja novamente acessado. Quando isto acontece, o mecanismo de gerência de energia é responsável por ligar este componente e qualquer outro componente que seja necessário para garantir que o sistema opere corretamente. Este mecanismo foi implementado permitindo a propagação de mensagens através dos componentes do sistema, o que se tornou possível devido ao modo hierárquico pelo qual estes componentes estão organizados. Um estudo de caso é apresentado para demonstrar o uso da técnica através de uma implementação real deste mecanismo utilizando um sistema operacional baseado em componentes desenvolvido especificamente para plataformas embarcadas, o EPOS [Fröhlich 2001].

Este artigo está organizado como segue. A seção 2 introduz a interface de gerência de energia para componentes de software e hardware. A seção 3 descreve o subsistema de sensoriamento que permite o gerenciamento do consumo de energia nestes dispositivos. A seção 4 apresenta uma aplicação para exemplificar o uso desta interface. A seção 5 apresenta uma revisão de trabalhos correlatos. A seção 6 finaliza o artigo.

2. Interface de Gerenciamento do Consumo de Energia para Componentes de Software e Hardware

Políticas para gerenciamento do consumo de energia em sistemas operacionais como LINUX e WINDOWS analisam dinamicamente o comportamento do sistema para determinar quando um componente de hardware deve modificar seu modo de operação através de uma interface compatível com ACPI. Contudo, a maioria dos sistemas de sensoriamento não podem arcar com os custos destas estratégias dinâmicas. Além disso, considerando que estes sistemas normalmente são compostos por uma única aplicação, o melhor lugar para determinar a estratégia de gerenciamento do consumo de energia é na própria aplicação.

Através da definição de uma interface uniforme para o gerenciamento do consumo de energia de componentes do sistema nós permitimos ao programador da aplicação mudar o estado de operação de cada componente do sistema, tanto de software quanto de hardware. A interface é composta por dois métodos: um para verificar o estado de operação atual do componente (`power()`) e outro para modificá-lo (`power(estado_desejado)`). O mecanismo por trás desta interface faz uso da organização hierárquica de componentes de software e hardware para permitir que esta migração de estados seja feita de forma consistente.

Componentes de hardware de baixa potência que são utilizados em sistemas de sensoriamento frequentemente apresentam um grande conjunto de modos de operação.

Permitir o uso de todos os modos de operação disponíveis, embora aumente a configurabilidade do sistema, pode aumentar a complexidade das aplicações quando gerenciando o consumo de energia. Para resolver este problema, foi estabelecido um conjunto de definições de alto-nível para os estados de cada componente, tornando desnecessário que o programador da aplicação conheça detalhes de cada um dos componentes de hardware que seu sistema possui. De modo semelhante ao ACPI [Corporation et al. 2004], quatro modos universais foram definidos: `FULL`, `LIGHT`, `STANDBY` and `OFF`. Contudo, estes modos podem ser extendidos pelos componentes sempre que necessário. Quando o dispositivo está operando com toda sua capacidade, ele está no estado `FULL`. O estado `LIGHT` coloca o dispositivo em modos de operação onde ele continua oferecendo as mesmas funcionalidades, porém consumindo menos energia e, muito provavelmente, implicando em alguma degradação de sua performance. No estado `STANDBY` o dispositivo para de operar, entrando em um estado do qual possa voltar rapidamente quando solicitado e continuar sua operação do ponto em que parou. Este estado provavelmente será um modo de *sleep*. Quando no estado `OFF` o dispositivo é desligado. É importante ressaltar que sempre que um dispositivo é desligado o retorno dele a um estado operacional implica em uma reinicialização, podendo assim implicar em perda de dados ou configurações previamente definidas.

Conforme as aplicações embarcadas nestes dispositivos de sensoriamento crescem em complexidade, estas passam a fazer uso de um maior número de componentes de sistemas. Com isso, pode tornar-se impraticável para programadores de aplicação controlar o consumo de energia de cada componente individualmente. Para solucionar este problema foi permitido que as aplicações alterem também o estado de operação de subsistemas (e.g., subsistema de comunicação, processamento, sensoriamento, etc) ou do sistema como um todo.

Para exemplificar como um subsistema inteiro pode alterar seu modo de operação, é apresentado uma breve descrição do ambiente experimental, o sistema operacional EPOS [Fröhlich 2001], e seu subsistema de comunicação. O *Embedded Parallel Operating System* (EPOS) é um sistema operacional orientado a aplicação baseado em componentes. No EPOS, abstrações de sistema de alto nível, como `File`, `Thread`, `Scheduler` e `Communicator`, são exportados para aplicações através de uma interface de componente. Estes componentes interagem com os dispositivos eletrônicos que utilizam através de mediadores de hardware [Polpeta and Fröhlich 2004]. Devido à hierarquia pela qual os componentes são organizados no sistema, cada abstração ou mediador sabe o estado de seus recursos. O subsistema de comunicação é apresentado na Figura 1 e compreende, basicamente, quatro famílias de componentes: `Communicator`, `Channel`, `Network` e `NIC`. `NIC` é a família de mediadores de hardware que abstrai o dispositivo de comunicação para a família `Network`. A família `Network` é responsável por abstrair o tipo de rede (e.g., Ethernet, CAN, ATM, etc). `Channel` é responsável por realizar a comunicação entre processos (IPC) e usa a `Network` para construir um canal lógico de comunicação através do qual mensagens são trocadas. Finalmente, o `Communicator` é um ponto-final para comunicações, e inclui uma estrutura configurável para adaptação de protocolos.

Para garantir portabilidade do código da aplicação, é esperado que o programador desta use abstrações de alto nível do sistema, como os membros da família

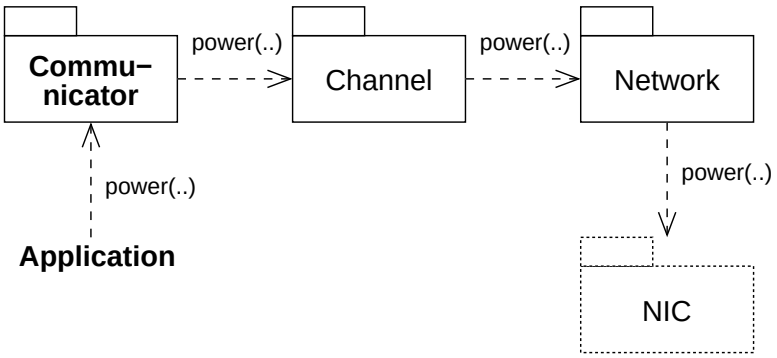

Figura 1. Subsistema de comunicação do EPOS.

`Communicator` no subsistema de comunicação. Neste contexto, a estratégia de gerenciamento do consumo de energia precisa prover meios pelos quais o programador da aplicação possa, apenas alterando o estado de um `Communicator`, modificar o modo de operação de todos os componentes envolvidos na real implementação das funcionalidades desenvolvidas por aquele componente. Isto é feito propagando mensagens através da estrutura hierárquica de componentes. Por exemplo, uma implementação de um `Communicator` vai utilizar um `Channel` e, provavelmente, um componente de temporização chamado `Alarm` para tratar *time-outs* no protocolo de comunicação. Quando a aplicação executa um comando solicitando que o `Communicator` altere seu modo de operação para `STANDBY`, o `Communicator` finalizará todas as comunicações iniciadas, esvaziando seus *buffers* e aguardando por todas as confirmações de recebimento (sinais `ACK`) antes de propagar mensagens de `STANDBY` para os outros componentes que usa.

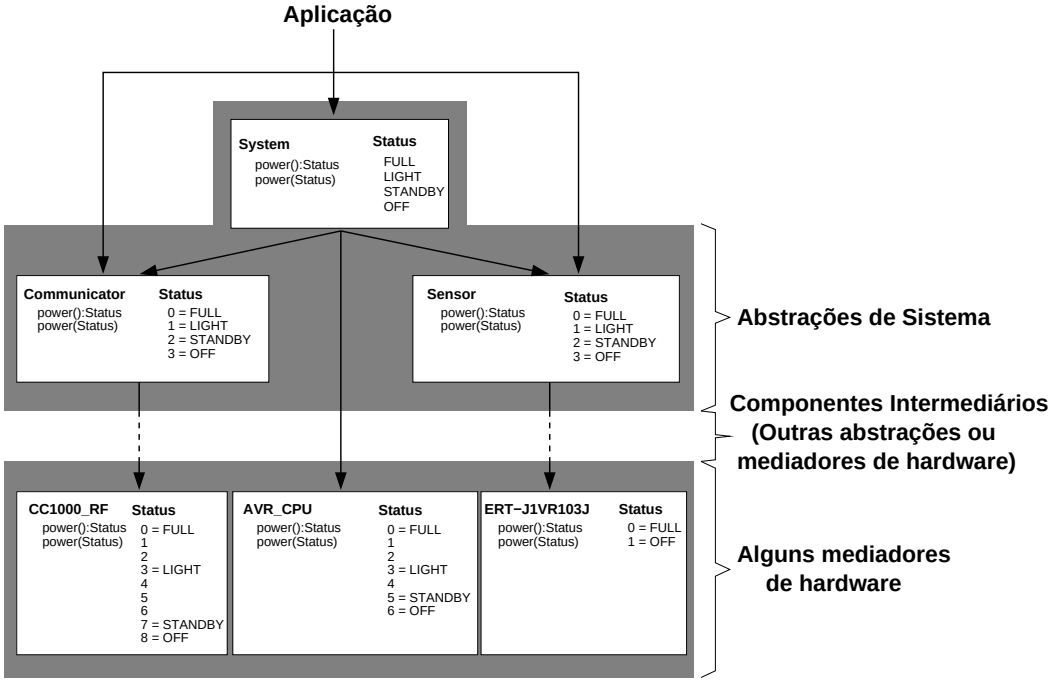

Figura 2. Acessando a interface de gerenciamento do consumo de energia.

Ações de gerência do consumo de energia do sistema como um todo no EPOS são

tradadas pelo componente `System`. `System` contém referências para todos os subsistemas em uso pela aplicação. Então, se uma aplicação deseja alterar o modo de operação de todo o sistema, isto pode ser feito acessando a interface deste componente, que propagará este pedido para todos os subsistemas.

A Figura 2 ilustra como a interface de gerenciamento do consumo de energia pode ser acessada. Ela mostra alguns dos componentes instanciados para um sistema de sensoriamento. Nesta instância, o sistema é composto por três subsistemas: processamento (`CPU`), comunicação (`Communicator`), sensoriamento (`Sensor`). A figura ainda mostra o componente `System`. Cada componente tem sua própria interface, que pode ser chamada pela aplicação a qualquer momento, e um conjunto de níveis de consumo de energia específico para cada componente. Se a aplicação deseja modificar o modo de operação de um subsistema específico, esta deve acessar diretamente seus componentes de mais alto nível hierárquico. Se a aplicação deseja modificar o modo de operação de todo o sistema, ela deve acessar o componente `System`, que propagará as mensagens através do sistema.

O principal desafio identificado no desenvolvimento de componentes *power-aware* foi a necessidade da migração consistente entre modos de operação de subsistemas inteiros em um ambiente onde as funcionalidades do subsistema estão divididos em vários componentes de software. Isto foi resolvido através do mecanismo de propagação de mensagens implementado, que garante que nenhum dado será perdido e nenhuma ação inacabada será interrompida. Permitindo que cada componente trate de suas responsabilidades (e.g., um `Communicator` esvaziando seus *buffers* e aguardando pelos sinais `ACK`) antes de propagar as mensagens de alteração do modo de operação para os componentes abaixo em sua hierarquia (e.g., para `Alarm` e `Channel`), é possível garantir troca de modo de operação consistente de um subsistema inteiro.

Nesta estratégia, é esperado que o programador da aplicação especifique, em seu código-fonte, quando certos componentes não estão sendo utilizados. Isto é feito executando os métodos "`power`" para cada componente, subsistema ou para o sistema. Para evitar que o programador tenha que, manualmente, acordar cada um destes componentes, o mecanismo de gerência garante que estes componentes retomem o seu estado anterior automaticamente quando acessados.

3. Componentes de Sensoriamento no EPOS

Para permitir gerenciamento do consumo de energia em hardware de sensoriamento heterogêneo, nós propomos uma interface de software/hardware que é capaz de abstrair dispositivos sensores uniformemente. Definimos classes de dispositivos sensores baseado em sua finalidade (e.g. medir aceleração ou temperatura), e estabelecemos um substrato comum para cada classe. Cada dispositivo individual é capaz de descrever suas propriedades, de maneira simular ao *data sheet eletrônico de transdutor* do padrão IEEE 1451. Uma camada fina de software adapta dispositivos individuais (e.g. converte leituras de ADC em valores contextualizados, calibra o dispositivo) para adequa-los ás características mínimas da sua classe de sensores. Desta forma, um termistor simples é exportado para a aplicação exatamente da mesma forma que um sensor de temperatura digital complexo. A auto-descrição baseada em software permite que as aplicações utilizem características estendidas de sensores individuais.

3.1. Subsistema de Sensoriamento do EPOS

A figura 3 apresenta uma visão geral simplificada do subsistema de sensoriamento do EPOS. Métodos comuns a todos dispositivos de sensoriamento são definidos pela interface `Sensor_Common`. O método `sample()` provê leituras para um único sensor em um único canal (i.e. habilita o dispositivo, espera os dados estarem disponíveis, lê o sensor, desabilita o dispositivo e retorna a leitura convertida em unidades físicas previamente determinadas). Os métodos `enable()`, `disable()`, `data_ready()` e `get()` permitem que o sistema operacional e as aplicações realizem controle preciso sobre leituras de sensores (e.g. realizar leituras seqüenciais, obter dados não convertidos de sensores). O método `convert(int v)` pode ser utilizado para converter valores não processados de sensores em unidades científicas ou de engenharia. O método `calibrate()` executa um método de calibragem específico para plataforma e dispositivo sensor, que pode exigir interação com o usuário, dependendo do sensor.

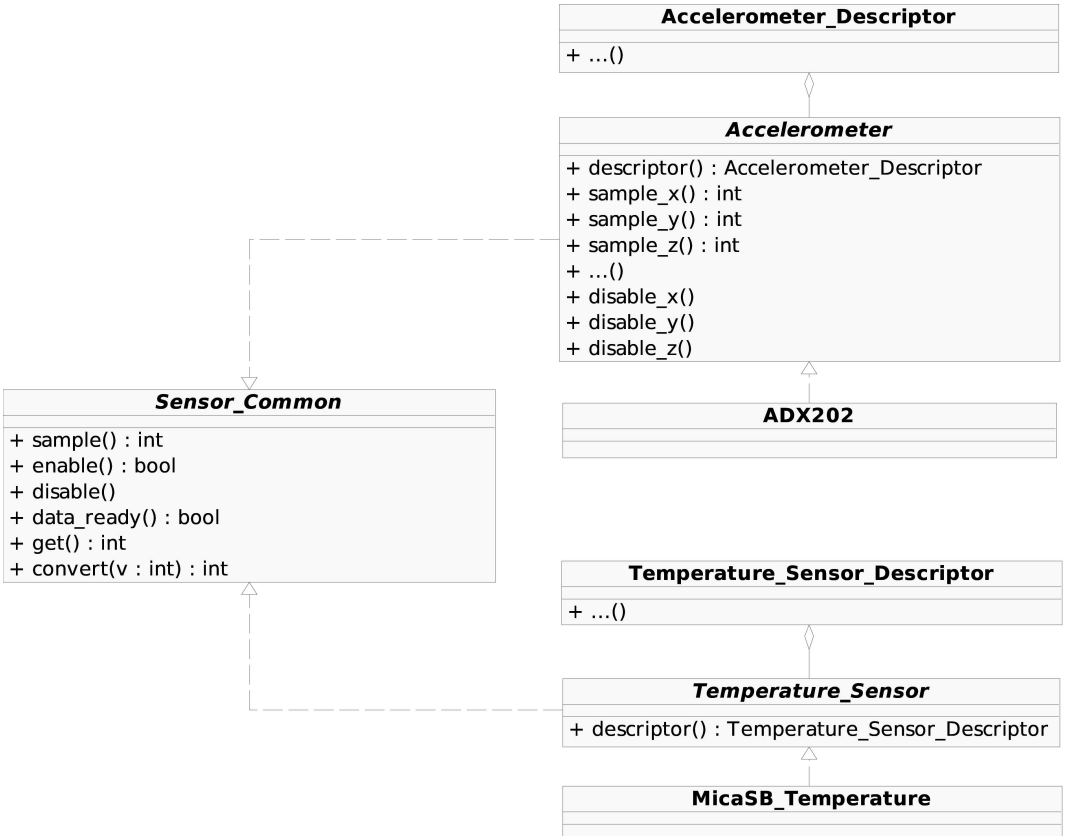

Figura 3. Visão Geral do Subsistema de Sensoriamento do EPOS.

Cada família de sensor pode especializar a interface `Sensor_Common` para abstrair adequadamente características específicas da família. A família `Magnetometer`, pode adicionar, por exemplo, metodos para realizar leituras em diferentes eixos de sensibilidade. A família `Thermistor`, por outro lado, provavelmente não precisará estender a interface comum básica. Cada família também define uma estrutura `Descriptor` específica, que define campos para precisão, dados para calibração e unidades físicas.

Cada dispositivo sensor implementa uma das interfaces definidas e pode fornecer métodos específicos para calibração, configuração e operação. Além disso, cada dispo-

```
class Accelerometer_Descriptor {

  int type();
  int precision();
  int physical_unit();
  // ...
  int sensitivity_x();
  int sensitivity_y();
  int sensitivity_z();
  int frequency();
  // ...
  struct Calibration;
  // ...

};
```

(a) Descritor da Família

```
template <> struct Traits<Mica2_Accelerometer>
{
  // Concreto   (Um dispositivo)
  typedef LIST<ADXL202> SENSORS;
};
template <> struct Traits<Mica2_Accelerometer>
{
  // Concreto (Dois dispositivos do mesmo tipo)
  typedef LIST<ADXL202,ADXL202> SENSORS;
};
template <> struct Traits<Mica2_Accelerometer>
{
  // Polimórfico (Dispositivos diferentes)
  typedef LIST<ADXL202,ADXL103> SENSORS;
};
```

(b) Lista de Dispositivos Exemplo

```
template <> struct Traits<ADX202>
{
  static const unsigned long CLOCK      = Traits<Machine>::CLOCK;
  static const unsigned char CHANNEL_X  = 3;
  static const unsigned char CHANNEL_Y  = 4;
  static const unsigned char PORT       = Traits<Machine>::IO::PORTC;
  static const unsigned char PORT_DIR   = Traits<Machine>::IO::DDRC;
  static const unsigned char ENABLE     = 0x10;
};
```

(c) Traits de Configuração Exemplo

Figura 4. Visão geral da família de Acelerômetros

sitivo preenche a estrutura `Descriptor` da família com valores específicos do sensor. Valores de padrão de configuração para cada dispositivo (e.g. freqüência, ganho, etc.) são armazenados em uma estrutura de *traits de configuração*.

Sempre que o sistema operacional ou uma aplicação precisam fazer referência a um dispositivo de sensoriamento, estes podem utilizar um *dispositivo específico* (e.g. `MicaSB_Temperature`) e realizar operações específicas do dispositivo, ou utilizar a *classe do dispositivo*, e restringir-se ás operações definidas por aquela classe. A estrutura de *traits de configuração* lista todos dos dispositivos de uma dada classe que estão presentes em uma dada configuração do sistema. Uma realização estaticamente meta-programada da classe do dispositivo agrega todos os dispositivos listados pelos *traits de configuração*. Esta realização é concreta quando todos os dispositivos de uma classe são do mesmo tipo, e polimórfica quando tipos diferentes de sensores existem em uma mesma classe.

3.2. Família exemplo de dispositivos: Acelerômetros

A família de dispositivos Acelerômetors estende a interface `Sensor` básica adicionando métodos para leitura de diferentes eixos de sensibilidade (ver figure 3). Dispositivos específicos implementam a interface `Accelerometer` completamente ou parcialmente (e.g. um acelerômetro de dois eixos não implementará métodos para o eixo `z`). A família também define uma classe `Descriptor` (parcialmente apresentada na figura 4). Dispositivos específicos podem ter estruturas próprias para calibração e `traits de configuração` definidos pelo usuário.

A família é realizada por um *wrapper* meta-programado. Uma lista de disposi-

tivos é definida por cada *machine* (e.g. `Mica2`). Se todos os dispositivos na lista são do mesmo tipo, a realização `Accelerometer` será concreta. Em outro caso, será polimórfica (Figura 4. A lista de dispositivos também define a ordem para o construtor `Accelerometer(int unit)`.

A aplicação pode utilizar tanto a realização `Accelerometer` quanto a implementação específica de um dispositivo. No primeiro caso, o programador de aplicação está restrito aos métodos definidos pela classe geral, mas obtém uma interface completamente portável. Métodos específicos de dispositivos podem ser utilizados somente atravéz da implementação destes (e.g. `ADXL202`). Entretanto, a aplicação pode usar a classe `Accelerometer` e utilizar a estrutura `Descriptor` para obter dados específicos dos dispositivos.

4. Estudo de Caso: Termômetro

Para demonstrar a usabilidade da interface definida, um termômetro foi implementado utilizando um protótipo com um termistor (resistor sensível a temperatura) de 10 KΩ conectado a um canal do conversor analógico-digital de um microcontrolador ATMega16, da Atmel [Corporation 2004]. A aplicação implementada para esta plataforma é apresentada na Figura 5. Esta aplicação utiliza quatro componentes do sistema: `System`, `Alarm`, `Temperature_Sensor` e `UART`. A organização hierárquica do EPOS amarra, por exemplo, a abstração de sistema `Temperature_Sensor` ao conversor analógico-digital (`ADC`) do microcontrolador.

```
#include <system.h>
#include <sensor.h>
#include <uart.h>
#include <alarm.h>

using namespace System;

System sys;
Temperature_Sensor sensor;
UART uart;

void alarm_handler() {
    uart.put(sensor.sample());
}

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

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

Figura 5. A aplicação Termômetro

Quando a aplicação inicia, todos os componentes são inicializados através de seus construtores, e um evento periódico é registrado no componente `Alarm`. O modo de

operação de todo o sistema é então alterado para `STANDBY` através do método `power` do componente `System`. Quando isto acontece, o componente `System` coloca todos os componentes do sistema em `STANDBY`, com excessão do `Alarm`, para garantir que o sistema não perca sincronismo. O `Alarm` utiliza um temporizador (*timer*) para gerar interrupções a uma dada frequência. A cada interrupção de tempo, a CPU acorda e o componente `Alarm` trata todos os eventos registrados, executando os que atingiram seu período. Neste exemplo, a cada segundo os componentes `Temperature_Sensor` e `UART` são acordados automaticamente quando são acessados e uma leitura de temperatura é enviada através da porta serial. Quando todos os eventos registrados são tratados, a aplicação continua a execução normal chegando ao laço principal, que põe o componente `System` de volta em `STANDBY`.

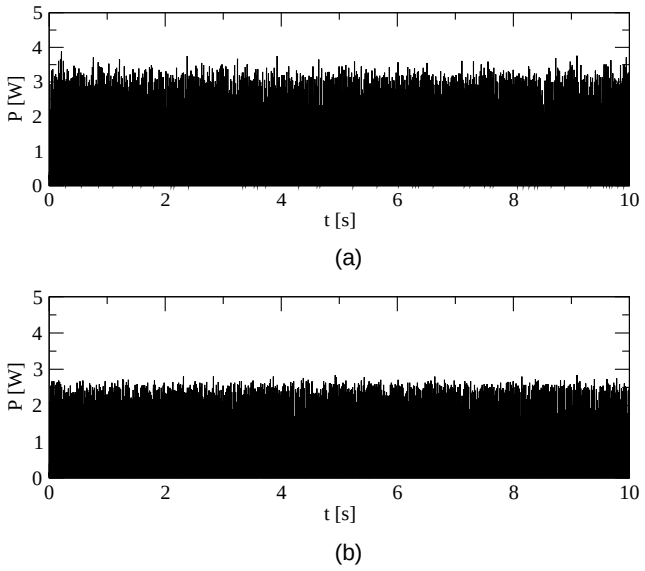

Figura 6. Consumo de energia para a aplicação Termômetro *sem (a)* e *com (b)* gerenciamento do consumo de energia.

Os gráficos apresentados na Figura 6 mostram medições do consumo de energia para esta aplicação sem e com gerência deste consumo pelo sistema. Ambos gráficos mostram o resultado de uma média de dez medições. Cada medição foi tomada por dez segundos. No gráfico (a) (sem gerência) é observado que o consumo do sistema oscila entre 2.5 e 4 Watts. No gráfico (b) (com gerência), a oscilação permanece entre 2 e 2.7 Watts. Calculando a integral destes gráficos, é possivel obter o consumo destas instâncias de sistema durante o tempo em que rodaram. Os resultados são 3.96 Joules para (a) e 2.45 Joules para (b), i.e., o sistema economizou 38.1% de energia sem comprometer a funcionalidade do sistema.

5. Trabalhos Correlatos

TINYOS e MANTIS são sistemas operacionais para plataformas embarcadas, mais especificamente, para plataformas de redes de sensores sem fio. Nestes sistemas, controle do consumo de energia foca principalmente na implementação de MACs de baixo consumo de energia [Polastre et al. 2004, Sheth and Han 2004] e escalonadores para roteamento multi-salto [Hohlt et al. 2004, Sheth and Han 2003]. Isto faz sentido no contexto de redes de sensores sem fio porque uma parte muito significante do consumo de energia do

sistema é devido ao mecanismo de comunicação. Embora estes sistemas mostrem resultados expressivos, eles constantemente focam no desenvolvimento de componentes que consumam o mínimo de energia possível, ao invés de permitir trocas entre consumo e performance, permitindo atender necessidades específicas de algumas aplicações. Outro incoveniente destes sistemas é o baixo nível de configuração oferecida pelos componentes de software e a falta de uma interface padronizada para acesso aos componentes do sistema.

SPEU (*System Properties Estimation with UML*) é uma ferramenta de otimização que leva em conta performance, tamanho de código do sistema e especifições de consumo de energia para gerar estimativas de sistemas otimizados para performance, tamanho de código ou energia. Estas informações são extraídas do modelo UML da aplicação embarcada. Este modelo precisa incluir diagramas de classe e de sequência. O protótipo usa um ambiente `Java` baseado no processador FEMTOJAVA [Ito et al. 2001]. Já que o SPEU apenas considera os diagramas UML, estas estimativas mostram erros de até 85%, tornando-a útil apenas para comparar diferentes decisões de projeto. O ambiente ainda é pouco configurável, já que o processo de otimização é guiado por apenas uma variável, i.e., se a decisão do projetista da aplicação for por um sistema que ofereça o máximo de performance, o sistema nunca entrará em modos de baixo consumo de energia, mesmo que não esteja utilizando certos dispositivos. Isto certamente limita seu uso em situações reais.

CIAO (*CiAO is Aspect-Oriented*) é um projeto que foca o desenvolvimento de um sistema operacional de baixa granularidade para linhas de produção de sistemas embarcados. Este sistema abstrai propriedades não funcionais em aspectos, aumentando a configurabilidade do sistema. Uma destas propriedades não funcionais é consumo de energia. CIAO continua em um estágio inicial de desenvolvimento, mas se mostra relativamente diferente do método utilizado pelo EPOS. O EPOS também utiliza aspectos para abstrair algumas propriedades não funcionais mas, enquanto o CIAO utiliza orientação a aspectos como metodologia de projeto para o todo o sistema, o EPOS usa aspectos como uma ferramenta extra de desenvolvimento de software. No caso particular do controle do consumo de energia, não é desejável modelar isto como um aspecto porque, embora seja uma propriedade não funcional, muitas vezes decisões tomadas para diminuir o consumo de energia afetam o funcionamento do dispositivo, modificando o comportamento do sistema [Lohmann et al. 2005].

O IMPACCT (*Integrated Management of Power-Aware Computing and Communication Technologies*) [Chou et al. 2002] é uma ferramenta de pré-escalonamento para explorar trocas entre consumo de energia e performance através de escalonamento e configuração arquitetural. A idéia por trás do IMPACCT é analisar a aplicação através de simulação para definir a maior gama de trocas entre consumo de energia e performance para cada componente durante a execução. Esta ferramenta inclui um escalonador *power-aware* para sistemas *hard real-time*. As ferramentas do IMAPCCT implementam um modo muito interessante de configurar escalonamento *hard real-time* focando consumo de energia em sistemas embarcados, porém está longe de provêr um ambiente onde a prototipação seja fácil e rápida.

6. Conclusão

Neste artigo foi apresentada uma estratégia para permitir gerenciamento de energia dirigido pela aplicação em sistemas de sensoriamento. Para atingir este objetivo foi permitido que programadores de aplicação expressem os pontos em que certos componentes não estão sendo utilizados. Isto é feito através de uma interface de software simples e uniforme que permite a troca de modos de operação de componentes individualmente, agrupados em subsistemas ou do sistema como um todo, tornando todas as combinações de modos de operação possíveis. Explorando a estrutura hierárquica pela qual componentes de software e hardware são organizados em nosso sistema, gerenciamento do consumo de energia eficiente foi possível sem lançar mão de técnicas ou estratégias custosas, gerando sistemas sem sobrecargas de processamento e memória desnecessárias.

Um estudo de caso utilizando um microcontrolador de 8 bits para monitorar temperatura em um ambiente fechado mostrou que quase 40% de energia foi economizada utilizando esta estratégia, o que implicou em apenas uma linha de código alterada na aplicação.

Agradecimentos Os autores gostariam de agradecer Hugo Marcondes e Rafael Cancian do LISHA pelas produtivas discussões. Também gostariam de agradecer ao Departamento de Ciências da Computação 4 da Universidade Friedrich-Alexander (Alemanha), seu coordenador Prof. Schröder-Preikschat e Andreas Weissel por terem providenciado equipamento e orientação inicial a este trabalho.

Referências

Bellosa, F., Weissel, A., Waitz, M., and Kellner, S. (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.

Benini, L., Bogliolo, A., and Micheli, G. D. (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.

Chou, P. H., Liu, J., Li, D., and Bagherzadeh, N. (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.

Corporation, A. (2004). *ATMega16L Datasheet*. San Jose, CA, 2466j edition.

Corporation, H.-P., Corporation, I., Corporation, M., Ltd., P. T., and Corporation, T. (2004). *Advanced Configuration and Power Interface Specification*, 3.0 edition.

Fröhlich, A. A. (2001). *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin.

Hohlt, B., Doherty, L., and Brewer, E. (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.

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

Joseph, R., Brooks, D., and Martonosi, M. (2001). 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*.

Lohmann, D., Schröder-Preikschat, W., and Spinczyk, O. (2005). Functional and non-functional properties in a family of embedded operating systems. In *Proceedings of the Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems*, Sedona, USA. IEEEPress.

Pillai, P. and Shin, K. G. (2001). 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. ACM Press.

Polastre, J., Szewczyk, R., Sharp, C., and Culler, D. (2004). The mote revolution: Low power wireless sensor network devices. In *Proceedings of Hot Chips 16: A Symposium on High Performance Chips*.

Polpeta, F. V. and Fröhlich, A. A. (2004). 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.

Sheth, A. and Han, R. (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.

Sheth, A. and Han, R. (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.