Helsinki University of Technology
Department of Computer Science and Engineering
Software Technology Laboratory
Espoo 2007 TKO-C94/07

Proceedings of Seminar on Energy-Aware Software

Vesa Hirvisalo (ed.)

Helsinki University of Technology
Department of Computer Science and Engineering
Software Technology Laboratory
P.O. Box 5400
FIN-02015 HUT
Espoo
Finland

Keywords: embedded system, energy

ISBN: 978-951-22-8871-7
ISSN: 1239-6907

Preface

The Software Technology Laboratory at Helsinki University of Technology organized a seminar on energy-aware software during spring 2007. The goal of the seminar was to take a look at the current status of the field. These proceedings include some of the work presented in the seminar.

Espoo, June 2007
Seminar chair

Vesa Hirvisalo

Contents

Hardware Accelerators in Embedded Systems

Lari Ahti
Helsinki University of Technology
Software Technology Laboratory
lahti@cc.hut.fi

Abstract

Hardware acceleration is use of additional hardware to perform some function more effectively than with more general hardware implementation. Espescially in wireless embedded systems, complicated communication algortihms require high performance from system on chip. In case of a battery powered device, algorithms are often implemented with hardware accelerators in order to meet both energy consumption and performance requirements.

One implementation, that takes advantage of hardware accelerators, is software defined radio, which is a solution targetted to solve problems caused by fixed hardware in mobile communication devices. Software defined radio means layer that is capable to perform highly demanding signal processing and yet be reprogrammable after manufacturing. Reprogrammability has several advantages: Support for multiple protocols, faster time-to-market, higher chip volumes and support for late implementation changes.

1. Introduction

This paper is divided into two sections where first section will give introduction to different hardware acceleration techniques and second section will concetrate in one spesific area of hardware acceleration: Software defined radio (SDR). This paper will concetrate in embedded systems, because hardware accelerators play crusial role in many embedded applications.

2. Hardware acceleration

This section gives motivation for hardware acceleration and brief introduction to hardware acceleration techniques. Both hardware and software technologies are discussed in separate subsections.

2.1. Motivation

Motivation is for hardware acceleration is usually performance increase of the system, but also energy consumption may be motivator, especially in embedded systems. Hardware acceleration changes behaviour of the system at least in the hardware level, but may also affect software implementation. [7]

Usually hardware acceleration implementation is a tradeoff between three main factors: Performance, energy consumption and flexibility. Other factors that may have effect on selecting specific technique are for example manufacturing price, design costs and time to market. Especially in battery powered devices energy consumption will limit other factors. Another example of hardware acceleration is graphics accelerators, that are designed to render realtime graphics. In this case performance of the accelerator is maximized and other factors are limited.

2.2. Hardware techniques

In this section several hardware techiniques for hardware acceleration are discussed. Most hardware accelerators decrease flexibility of the system in order to gain performance or reduce energy consumption. Usually hardware accelerators co-operate with general purpose processor, that controls the accelerator. Acceleration hardware can be divided into several groups that vary in performance, energy consumption and flexibility. Here are listed some most common hardware accelerator implementations for embedded systems:

- **Application specific integrated circuits** (ASIC) is integrated circuit specially designed for acceleration of some function and they provide highly tuned performance for certain task, but desing costs are high and time to market is longer than with other hardware solutions. Also lact of ability reprogram ASICs after manufacturing is a disadvantage.

- **Field programmable gate arrays** (FPGA) are generally slower than ASICs and tend to use more power, but they can be made reprogrammable and design process is usually easier.

- **Digital signal processors** (DSP) provide efficient mechanism for handling real-time digital signal processing. DSPs are designed to handle streamed data and have spesialized instruction set architecture for signal processing related functions.

[7]

There are also other types of accelerators used and in many cases combination of several accelerators are used. For example graphics accelerator is usually a combination of several hardware accelerators that are located in single chip.

2.3. Software techniques

In many cases, hardware acceleration affects also the software of the device as new hardware configuration is introduced. Software techniques provide methods to take advantage of hardware configuration of device. Generally software implementations can be divided into threee methods: Instruction level acceleration, function level acceleration and architectural enhancements.

Instruction level acceleration means that most common instruction sequencies are implemented with spesialized instructions. In some cases instruction level parallelism can be used to increase performance of the system. This method limits flexibility of the system and increases performance. Spesialized instruction require extensions to instruction set architecture and therefore also software compilers may require extensions.

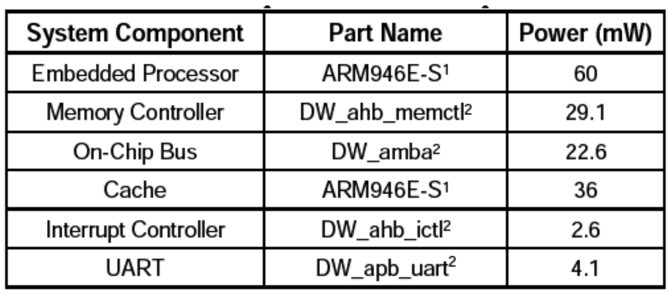
Function level acceleration means that entire algorithm is replaced with spesific hardware. Functional level acceleration typically results in more efficient solution than with instruction level acceleration, but fixed hardware limits flexibility of the system. Execution of the function level accelerator can often be executed parallel with other hardware, which increases performance even more, but can introduce new problems. Functional level accelerators algorithm must be known before manufacturing, because of fixed hardware. Also reprogramming of accelerator may be difficult or even impossible after manufacturing.

Architectural enhancements are targetted to increase performance of the existing hardware by adding more execution units or implementing parallel execution of instructions. In this method no instruction set architecture extensions are required, but performance increase in usually smaller than with other acceleration methods.

[7]

2.4. On-Chip Communication

So far separate processing units have been covered, but also communication between elements must taken into account, when discussing hardware acceleration. Figure 1 presents power consumption distribution in embedded processor system. On-chip bus power consumption presents about 15% of the sum of all elements power consumption.



System Component	Part Name	Power (mW)
Embedded Processor	ARM946E-S^1	60
Memory Controller	DW_ahb_memctl^2	29.1
On-Chip Bus	DW_amba^2	22.6
Cache	ARM946E-S^1	36
Interrupt Controller	DW_ahb_ictl^2	2.6
UART	DW_apb_uart^2	4.1

Figure 1. Power consumption of system on chip components at 200 Mhz frequency [3]

Several methods for increasing power efficiency of on-chip communication have been presented. For example bus encoding, segmented bus design, interface power management, and traffic sequencing are techniques that are designed reduce overall power consumption of the communication bus. A completely different approach to on-chip communication is "network on-chip" technology that replaces bus wiring with network between processing elements.

2.4.1 Bus encoding

In bus encoding, data transferred through bus is encoded to gain power savings. Bus encoding can be also used to address other problems like delay and crosstalk reduction. [5] Several encoding schemes have been developed. Encoding can take advantage of bus usage characteristics. For example address bus encoding can reduce switching activities on bus by exploiting the statial locality and temporal locality of the bus. [9] Bus encoding can also increase complexity of the bus, which may descrease overall performance or add additional power consumption to system.[3]

2.4.2 Segmented busses

In normal buses, each signal transmitted into bus is broadcasted to all possible bus slaves. This increases power consumption of the bus as unneccesary work is done to deliver signal. Segmented bus architecture addresses this problem by including additional logic to bus. This logic controls transmission of signals and reduces power consumption by selecting target of signal. This technique can introduce additional delay and power consumption due to control logic in bus.[3]

2.4.3 Power Management

Power management reduces power consumption by limiting performance of managed elements when possible. For example in on-chip busses idle slave interfaces can be shutdown and restarted on demand. Slave interface waking may introduce additional delay to bus and thus effective power management logic is essential. For example idle times may predicted when deciding wheter to shutdown slave interface or leave it idle. Power savings in this technology depend on system on chip architecture and power management logic.[3]

2.4.4 Traffic Sequencing

In case of multiple masters attached to bus, traffic sequencing can reduce the switching activity in bus and thus reduce overall power consumption of the bus. This method does not nessecerily require new hardware, because it can be implemented in application run on system. Another option is to create more sophisticated bus protocol that allows bus to give feedback about current state of the bus.[3]

2.4.5 Network-On-Chip (NOC)

Previous techniques described different techniques to improve power efficiency of bus based communication on chip. Network-On-Chip is a new approach to communication between processing elements and it is targetted to avoid common problems faced with busses or dedicated wiring. Idea in NOC is to replace ad-hoc global wiring structures with interconnection network. This approach benefits especially in architectures where thermal issues limit the overall performance of system on chip. So far commercial implementations are still based on bus architecture, but in future NOC implemetation may be developed. [2]

3. Software Defined Radio

Wireless communication is a field where hardware accelerators are commonly used to perform signal processing tasks. Battery powered devices require energy consumption to be as small as possible, but complicated signal processing algortihms require high computational power. To overcome problem caused by this scenario, hardware accelators are used to perform computionally demanding tasks with smaller energy consumption than with more general purpose hardware. Design and verification of fixed hardware systems is difficult, because each implementation is suited for spesific task. Reoccuring costs are also high, because reuse of previous designs is difficult. Fixed hardware may cause problems, if hardware defect is detected after manufacturing, due to lack of reprogrammability.

Software defined radio is a solution targetted to solve problems caused by fixed hardware. Software defined radio means layer that is capable to perform highly demanding signal processing and yet be reprogrammable after manufacturing. Reprogrammability has several advantages: Support for multiple protocols, faster time-to-market, higher chip volumes and support for late implementation changes. [4]

3.1. Motivation

Motivation for software defined radio is to find energy efficient way to handle multiple computationally demanding protocols in order to meet communication requirements for wireless battery powered devices. Historically application prosessing requirements have had main focus in system architecture design, but problem emphasis has shifted to network protocols and signal processing. [4] Currently several different protocols are needed to satisfy all wireless network types. Common modern wireless protocols are shown in Figure 2.

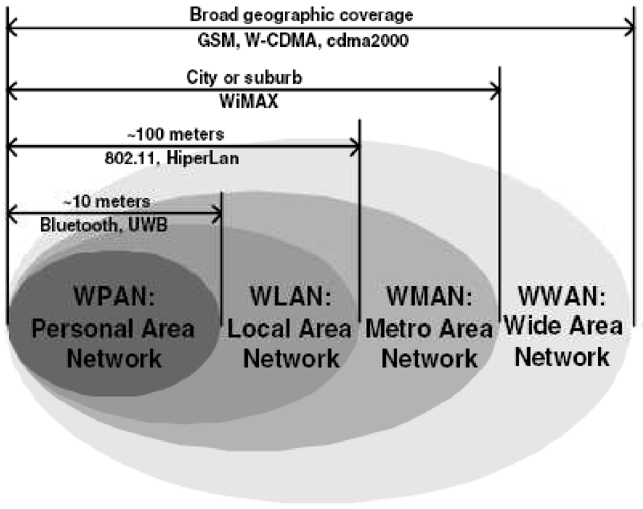


Figure 2. Modern wireless communication protocols. [4]

Problem with these protocols is that their computational requirements are much higher than with capabilities of modern DSP processors. Current DSP processors are able to perform 10 Mops/mW while modern wirless protocols require about 100 Mops/Mw. [4] To increase performance of the system, hardware accelerators are needed.

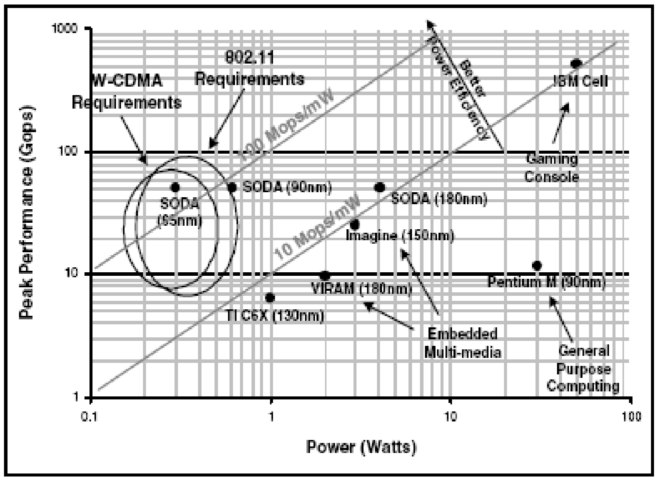


Figure 3. Computational power relative to energy consumption in some hardware implementations. [4]

3.2. Architecture suggestions

Several different software defined radio architectures have been suggested. It is interesting to note, that solutions are very different from each other. List of few interesting software defined radio architectures types:

- **Hybrid-SIMD based architecture**. This solution architecture consist of separate scalar and vector processors. Scalar processor is used to control signal processing and vector processors are used to perform actual processing. Examples of such architectures are SODA [4] and SandBlaster [6].

- **FPGA based architecture** Many solutions take advantage of FPGAs as they provide decent performance increase and are reprogrammable. Solutions usually include a processor that controls processing. Difficulties arise from requirements to meet real-time constraints. Example of architecture is picoArray [1].

- **Heterogeneous architecture** Some solutions use several heterogeneous processing elements to satisfy computational requirements. Each processing element is designed for spesific communication signal processing algorithm. This approach limits flexibility of the system, but is very efficient for spesial purpose. Also workload distribution among processing elements is difficult due to heterogeneous architecture and each processing element type must be capable to handle worst case workload.

- **VLIW DSP architecture** Very long instruction word (VLIW) DSPs can achieve high performance by executing several instructions parallel. The idea in VLIW is that several instructions can be combined into single intruction word in compile time and then executed parallel in hardware. The instruction execution energy consumption is higher than with other architectures, which limits also the overall perfomance of the architecture. Current DSP implementations don't satisfy high computational requirements of modern wireless algorithms, and thus architecture requires some type of hardware accelerator to give additional performance. One example of VLIW DSP processor is the Texas Instruments TMS320C64x DSP. [8]

4 Summary

Many embedded systems require high computational performance and extremely low power consumption. Hardware accelerators are used to provide additional performance for system-on-chip. Hardware accelerators perform some function more energy efficiently than with general purpose processors. Usually accelerators limit flexibility of the system by adding more task specific hardware to system.

Many modern wireless battery powered devices use hardware accelerators to meet with computional requirements with limited energy capacity. Historically hardware accelerator implementations were created with task specific hardware that is not very flexibile. Current mobile devices implement wide range of wireless protocols, which results in difficulties with complicated hardware implementations. More dynamic approach is required to create more generic processing elements that can be reprogrammed after manufacturing.

Software defined radio is a solution to problems caused by fixed hardware implmentations in mobile devices. The idea in SDR is to create layer that is capable to handle several complicated communication algorithms with limited energy resources. Hardware accelerators are used to gain performance without signicantly increasing power consumption of the device. Many SDR architectures have been proposed, but consumer products with SDR implementations are not yet available.

References

[1] R. Baines and D. Pulley. Software defined baseband processing for 3g basestations. In *4th International Conference on 3G Mobile Communication Technologies*, pages 123–127, 2003.

[2] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In *DAC '01:*

Proceedings of the 38th conference on Design automation, pages 684–689, New York, NY, USA, 2001. ACM Press.

[3] Kanishka Lahiri and Anand Raghunathan. Power analysis of system-level on-chip communication architectures. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 236–241, New York, NY, USA, 2004. ACM Press.

[4] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. Soda: A low-power architecture for software radio. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 89–101, Washington, DC, USA, 2006. IEEE Computer Society.

[5] Chittarsu Raghunandan, K. S. Sainarayanan, and M. B. Srinivas. Bus-encoding technique to reduce delay, power and simultaneous switching noise (ssn) in rlc interconnects. In *GLSVLSI '07: Proceedings of the 17th great lakes symposium on Great lakes symposium on VLSI*, pages 371–376, New York, NY, USA, 2007. ACM Press.

[6] Michael Schulte, John Glossner, Sanjay Jinturkar, Mayan Moudgill, Suman Mamidi, and Stamatis Vassiliadis. A low-power multithreaded processor for software defined radio. *J. VLSI Signal Process. Syst.*, 43(2-3):143–159, 2006.

[7] Eric Tell. *Design of Programmable Baseband Processors*. PhD thesis, Linkping Studies in Science and Technology, 2005.

[8] Texas Instruments. *TMS320C64x DSP Generation*, 2003. http://www.softier.com/pdf/sprt236a.pdf.

[9] Feng Wang, Yuan Xie, N. Vijaykrishnan, and M. J. Irwin. On-chip bus thermal analysis and optimization. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 850–855, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

Harnessing the Power of Software – A Survey on Energy-Aware Interfaces

Gerard Bosch i Creus
Nokia Research Center
P.O. Box 407
FIN-00045 Nokia Group (Finland)
gerard.bosch@nokia.com

***Abstract*—**
Mobile devices are becoming increasingly complex and software-intensive. The platforms supporting these devices need to cope with the growing complexity while maximizing the use of the underlying hardware. At the same time, manufacturers need to ensure that the battery life is not an encumbrance to user experience. Since battery technology does not have the capability to provide the necessary improvements in energy density, the next solution is to make devices more energy-efficient. Software can be regarded as the ultimate power consumer and therefore making more energy-efficient software can have a significant impact on the battery life of mobile devices. This paper focuses on software adaptations for energy efficiency, and specifically the interfaces to the operating system (OS). I present a survey of the previous research on the area as well as a review of the power management functionality offered by leading mobile OSs. There is a clear gap between the state of the art and the state of the practice. Bridging this gap has the potential to provide considerable energy savings and give some relief to the energy consumption problem.

I. Introduction

The exploding number of features is rapidly adding to the amount of processing power and related hardware needed for their implementation. Consumers desire more performance, more impressive multimedia, faster data connections, and better usability. As a result, devices are getting more power-hungry to the point where power consumption and thermal issues become seriously limiting factors.

Power management is required because mobile phones are battery-operated devices and run on a limited power supply. Additionally phones are becoming smaller in physical size which can make excessive power consumption heat them up more easily. Battery technology improves at a steady rate, but is not able to keep pace with the continuous upscaling of processing performance and resource usage. Current battery technology cannot offer the energy densities required to make the power consumption problem disappear. Given that battery technology does not seem to provide the necessary improvements regarding energy and power management, the next solution is to try improving the phone platforms so that the desired features can be implemented at a much lower cost in energy consumption.

Possible solutions can be addressed using both hardware and software approaches. The hardware approach is often emphasized since hardware is the part physically draining energy from the battery. From this viewpoint, it makes sense to focus on hardware optimizations. However, since the only mission of hardware is actually to fulfill software needs, one can argue that software is the ultimate consumer of energy and therefore the focus should be on software optimizations [1].

Extensive research has been produced on both hardware and software sides. The approaches should not be considered mutually exclusive, but rather synergistic in nature. Hardware should ideally provide an optimal trade-off between energy and other non-functional attributes such as performance. On the other side, software should strive to use those hardware pathways offering the optimal trade-offs for the application at hand. Most of the software is constructed with the help of supporting software development tools like compilers that may prioritize attributes such as speed or memory footprint at the expense of energy efficiency [2]. However there seems to be a growing understanding that the applications and their interaction with the underlying platform play a crucial role in power management [3], [4], [5], [6].

This paper presents a review of the available literature on software adaptations for energy efficiency. I focus on the layers ranging from the application software to middleware and its interactions with the underlying operating system (OS). In addition, I present a review of the available mechanisms on several mainstream OSs for mobile devices.

II. Tackling the right abstraction level

Software may be addressed at different levels of abstraction. Typically, the higher the abstraction level, the more impact any modifications will have on non-functional quality attributes. Figure 1 shows the impact on energy consumption for modifications at different levels of abstraction. This section reviews the different possibilities in this regard.

A. Architectural level

Software architecture is the highest level of abstraction in software design. Architects decide different high-level features and characteristics of software components at this level. Software is typically architected before proceeding onto more detailed design, although iterative software development processes imply a more cyclical nature for architecting.

The wide view that software architecture provides may offer unique opportunities for energy optimizations. Tan et al. [3]

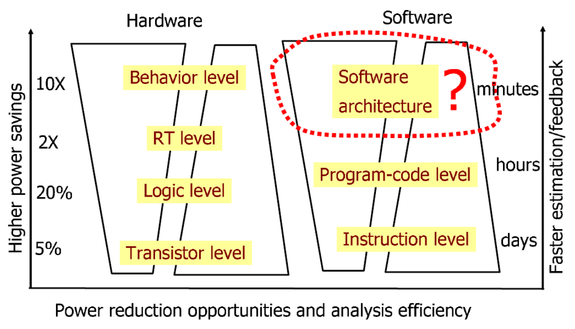


Fig. 1. Abstraction levels and their impact on energy consumption [3]

propose a tool for energy optimizations at the architectural level. Their approach requires accurate modeling in the form of *software architecture graphs* (SAGs). Using SAGs as an input, the tool exploits several aspects of software like temporal and sequential cohesion or IPC merging and replacement, among others. The authors claim the approach provides up to 66.1% power reductions.

However, such an approach requires a fairly accurate architectural description of the actual software that in some cases falls rather close to the implementation. Such accurate modeling may not be available at the time when architecture is being prepared. Additionally, the more architecture and actual implementation diverge, the more the results from this approach will suffer. Unfortunately, practice shows that software rarely preserves the designed architecture in its entirety.

B. *Instruction level*

Compilers handle most of the translation from program structure to actual platform-specific instructions except for a few cases where manual optimization is required. Most developers are used to working instead at the logical level since software may be targeted for several different platforms. Nevertheless, critical code may require manual optimizations and therefore developers undertaking such an activity should be aware of the different trade-offs at this level between performance and energy consumption.

Previous studies show that the right choice of instruction can have a strong impact on energy consumption. Simunic et al. [2] claim savings of up to 90% by modifying specific constructs on an ARM processor. For example, unsigned integers are more efficient than their signed counterpart, and 32-bit integers should be preferred over shorter types as they are much more energy-efficient. Recursive algorithms can be more energy-efficient that their equivalent iterative version in certain cases [2].

C. *Logical level*

Developers and platform providers interface at the logical level. This is the level at which most of the implementation takes place. Platform providers typically offer software development kits with a set of application programming interfaces that developers use to implement their software on a given platform. Software comes from a variety of providers: smartphone manufacturers, open-source contributors, tool vendors, and other third-parties. Together, these individuals represent a vast community with high impact on the energy expenditure of the software they create. It is in the interest of device manufacturers to assist developers on energy efficiency issues, since bad user experience related to poor battery life can have a strong impact on brand perception.

Manufacturers can provide assistance in the form of development tools targeting energy optimizations or by providing interfaces that developers can use to increase energy efficiency. Previous studies have recognized these two areas to be of vital importance in software development [6]. In the following, I focus on the interfaces that developers can use to produce more energy-efficient software.

III. Proposed Solutions

Several authors have previously addressed the topic of energy-aware interfaces between the OS and software running on top of it. This section reviews some proposed collaboration mechanisms that applications and middleware could exploit for greater energy efficiency.

A. *Adaptive applications*

The concept of adaptive applications is an integral part of the Odyssey framework [7], [4]. Odyssey is a resource management framework and in this sense it is not specifically focusing on energy. Odyssey tackles energy along with other resources such as network bandwidth, CPU, or disk cache space. Application adaptation is tightly bound to the concept of fidelity, in which applications adapt their quality of service (QoS) to their allowed resource usage levels.

Odyssey emphasizes the necessary balance between application diversity and concurrency. In this context, diversity refers to the vast differences in application software and their resource usage, and concurrency refers to the capability of satisfying the resource usage of simultaneously executing applications. The level of OS involvement is directly correlated to this balance. Application diversity is best represented by giving applications direct control over resources. On the other hand, application concurrency is best achieved by keeping resource control at the OS level, making resource management totally transparent to applications. Figure 2 presents the Odyssey architecture.

Odyssey takes a balanced approach by letting applications negotiate their resource usage with the system. Negotiation happens through a set of functions to request a certain resource level. The framework is responsible for enforcing the agreed resource levels. Odyssey monitors resource usage and notifies applications when a certain level can not be guaranteed anymore. In that case, applications must modify their resource levels. Diversity is supported by letting applications decide the mappings between resource levels and fidelity. In other words, Odyssey implements an admission control mechanism typical of real-time operating systems. However, Odyssey tackles resources at a much higher level of abstraction.

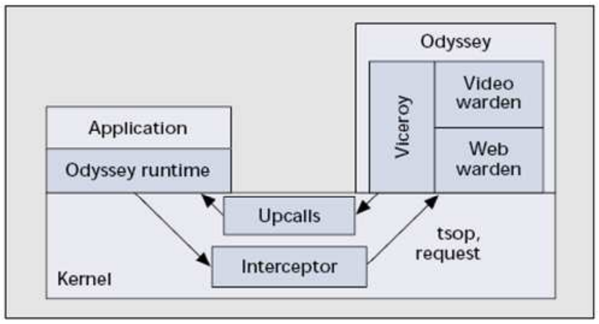


Fig. 2. Odyssey framework architecture [7]

The Odyssey framework has been validated with different applications [5], [8], [6]. Application adaptation can have a significant impact on power consumption. Flinn and Satyanarayanan achieved up to 72% savings in power consumption using application adaptation [5]. However, they acknowledge that the savings vary according to the degree to which applications lend themselves to fidelity management.

Server-side collaboration may be required for maximum energy savings in some applications. However, fidelity management is handled entirely on the client side. For example, a streaming video player may request the server side to change to a lower quality stream. Neglecting the server side will typically result in suboptimal savings.

B. Ghost hints

Anand et al. [9] propose the notion of *ghost hints* to increase energy efficiency. Their approach is based around the concept of optimizing data access paths for applications. The authors assume that applications may fetch required data through several devices such as the hard disk or the network interface. These devices may in turn implement several states with different performance to power ratios. The operating system is responsible for steering the different device states in the most energy-efficient manner. However, optimally selecting the most efficient device states requires knowledge about the future use of devices, which is not available. For example, a single access may not warrant a spin-up for a hard disk. However, the transition would pay off in the case of multiple accesses.

Applications query device power states from the OS. When accessing data, applications compute the best path to access data with the available device situation as well as the optimal path should the power state levels have been different. When applications detect that an access could have been improved by having a device in a different state, they issue a *ghost hint* to that device power manager. If the power manager receives several ghost hints in succession, it may consider changing the power state for that device. Without ghost hints, the device would not be used at all and thus the power manager would be totally oblivious to the suboptimal device state allocation. In other words, ghost hints provide a more proactive approach to device power management by allowing

TABLE I
APPLICATION CHARACTERIZATION FOR NETWORKING [10]

Parameter	Tiny	Small	Medium	Large	Huge
Received packet size	SSH		Browser Stream NFS	Stream	Download
Sent packet size				NFS	
Received/sent ratio	SSH		Stream		Download
Inactive/active ratio	Download	NFS	SSH	Browser	
Inactive period jitter	Stream			Browser	

for application input. Applications are in the best situation to offer such hints because they know the acceptable power to performance trade-offs for the user. Applications need to balance energy efficiency with access performance, which is typically dictated by the user expectations on battery life.

C. Application characterization

Weissel et al. [10] propose the concept of *application characterization* to adapt the system to the application needs. Modifying application code to provide input to the system for power management may not always be feasible. The source code or the resources to make the necessary modifications may be unavailable. Instead, the authors propose that application type may be inferred from run-time access characteristics. Weisser et al. show that network traffic characteristics can be used to identify different applications and steer the WLAN interface power modes accordingly. Their approach requires determining the traffic parameters most indicative of application type. Table I shows some of the correlations between traffic characteristics and applications [10]. The authors claim that in the context of traffic characteristics, most applications can be classified in three groups: interactive, non-interactive with strong performance requirements, and streaming applications.

Different application types have different latency and bandwidth requirements and expectations. For example, web browsing can support much higher latency than NFS directory listing without a negative impact on user experience. These requirements and expectations can then be mapped to the available power states with different power to performance ratios.

D. Idle period notifications

Idle periods provide opportunities for more aggressive power management since devices can then be transitioned to sleep modes and more energy-conservative states. Heath et al. [11], [12] propose increasing the active run-lengths in order to maximize idle periods. In addition, notifying the OS about the length of upcoming idle periods allows for improved power state steering. Since power state transitions carry an inherent cost with them (in terms of time and energy), the system needs to decide whether such transitions are worth based on a prediction of the idle period duration. Idle period notifications allow for more accurate idle time calculations than other typically used predictive methods.

The authors argue that application modifications are necessary to obtain maximum energy savings. However, being able to accurately predict idle period duration may require extensive knowledge about the platform characteristics as well as compiler support.

IV. Operating system interfaces

Operating systems and their interfaces are a critical area for power management research [6], [8]. Traditionally, operating systems have focused on other non-functional quality attributes at the expense of energy efficiency, such as performance or reliability. Especially in the case of mobile OSs, power is becoming an increasingly important resource and therefore mobile manufacturers are starting to demand better power management capabilities.

Typically, OS interfaces either leave applications out of the power management equation entirely, or tend to give applications an excessive amount of control over resources. Section III discusses part of this problem related to supporting application diversity and concurrency, and the level of OS involvement. Yung-Hsiang et al. [1], [13] identify two main categories for typical OS power management strategies, namely *autonomous* and requester-controlled. Autonomous strategies do not distinguish among requesters, and manage power in response to observed requests at the device level. On the other hand, requester-controlled approaches give requesters direct control over the power states for different devices. In practice, requester-controlled approaches are heavily based on ACPI [14] and the notion of power states. The authors argue that autonomous power management cannot reflect the diverse power consumption patterns and performance requirements for different applications. Requester-controlled alternatives do not offer clear distinctions for power states, making power state decisions impractical at the requester level. In addition, a lack of agreement on the power state for a certain device may damage hardware when multiple requesters are involved.

Yung-Hsiang et al. propose using *requester-aware* power management, a mixed approach better supporting both application diversity and concurrency. The main idea is to allow requesters to affect but not control the device power states. Power management should be as transparent as possible for developers, in the same manner that OS interfaces abstract away relatively complex domains such as memory or file management. In this spirit, the proposed system offers a very simple interface. Applications specify their performance requirements for specific devices to a central scheduler, which also monitors device requests. With this information, the scheduler is able to predict future resource usage and upcoming idle periods, which the power manager uses to set device power states in the most energy-efficient manner. In addition, this scheme allows the scheduler to cluster idle periods from different requesters

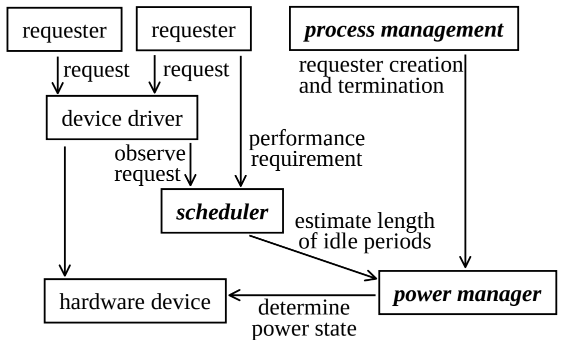


Fig. 3. Requester-aware power management

thus maximizing power saving opportunities. Figure 3 presents the requester-aware power management approach.

In the following I review the power management interfaces offered by several mainstream mobile OSs.

A. Symbian OS

Symbian OS is the leading smartphone OS in terms of market share. Nokia uses Symbian OS as the foundation of S60, its own smartphone platform. Symbian OS includes a power management framework, which mostly focuses on the interfaces toward hardware devices [15].

The Symbian power management framework is based on the concepts of *power model*, *power handlers*, and *power resources*. The framework provides a clear distinction between policy and mechanism. Power handlers represent devices requiring power management, i.e. with significant power drain on the system. The power model provides the policy for device power management. Figure 4 shows the relationships between these components.

Typically, device drivers provide an associated power handler if the device exhibits a considerable power consumption. The power model may require power handlers to take action in response to a system power event. Power handlers may also request system power state changes on behalf of their associated device. The power model powers up and down the shared power inputs for the different power handlers on demand, and informs power handlers when devices are required to power up and down, or perform an emergency shutdown. Power handlers report their power consumption to the power model, which the kernel then adds together with the CPU drain to obtain an approximate picture of the overall power consumption.

The power framework defines a set of power states that devices should implement. Table II presents the different device power states in Symbian.

In practice, the *Restart* state is not used since devices are expected to maintain their internal state. Transitions between *Idle* and *On* states are assumed to be fast and implying no data loss.

There is one power model in the system, responsible for controlling the power behavior of the hardware components. The power model is aware of each power handler in the system

TABLE II
DEVICE POWER STATES IN SYMBIAN OS

Device Power State	Description
On	Full power
Idle	Low power mode and inactive. The device can still respond to interrupts or external events
Standby	Inactive with internal state maintained
Restart	Inactive with lost internal state

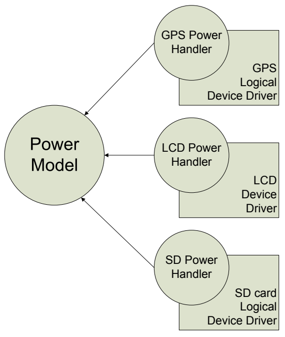


Fig. 4. Symbian OS power management architecture

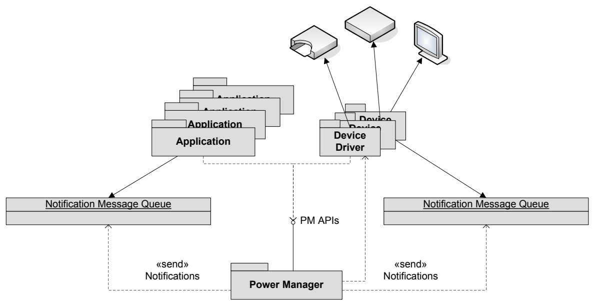


Fig. 5. Windows Mobile power management architecture

TABLE III
DEVICE POWER STATES IN WINDOWS MOBILE V5

Power State Type	Device Power State	Description
Full On	D0	Full power
Low On	D1	Fully functional at lower power or performance
Standby	D2	Partial power, standing by or wakeup request
Sleep	D3	Sleeping, minimal power needed to initiate wakeup
Off	D4	Totally off

and their requirements, and adjusts the system power states in response to power events. Power events may be caused by powering on and off the system and by changes to the state of shared power resources. The system needs to behave gracefully also in the event of a critical power failure.

Power resources are powered up and down as needed, when requested by their power handlers. Power resources use their power handlers to specify power requirements to the power model. The power model changes the system power state in response to the cumulative power requirements for all power handlers.

B. Windows Mobile

The power management infrastructure in Windows Mobile v5 is built around the Power Manager (PM) component [16]. The PM provides interfaces at the application, system and driver level for developers, with the goal of extending battery life. The PM is based on the notion of power states, making a clear separation between system and device (driver) power states. Both application and driver developers are actively encouraged to make use of the PM interfaces to control devices. The PM uses a publish-subscribe pattern to update software of impending changes in power states. Figure 5 presents the Windows Mobile power management architecture.

The PM is designed around the concept of power states and a clear division between system and device states. Devices are expected to implement a set of well-defined power states, independent of states defined by the ACPI standard [14]. Manufacturers define system power states that provide an upper limit state, *ceiling*, for all devices. Applications can dictate the bottom limit state, *floor*, for a particular device through the PM interface. Devices are allowed to manage their own power states between the set ceiling and floor levels.

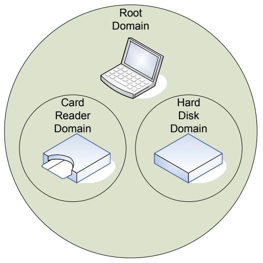
System power states are named collections of device power states, such as *Battery*, *Docked* or *UserIdle*. There is no limit for the number of system power states, and state transition does not need to be linear. Device power states follow a clearly specified hierarchy. Table III presents the device power state hierarchy. States *D0* and *D1* should be fully functional from a user perspective, and higher numbered states typically consume less power. Power management can increase device driver complexity considerably. For example, drivers may need to behave differently upon receiving a power down event while on a state other than *D4*. Dependencies to other devices (and their power states) will influence the course of action for device drivers.

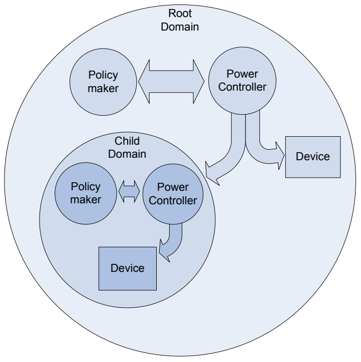
C. Darwin

Darwin is the open-source UNIX-based foundation of Apple's Mac OS X. The basic concepts underlying Darwin power management are hierarchies of devices (referred to as power domains in Darwin) and their supervising entities, policy makers and power controllers [17], [18], [19].

The fundamental entity in Darwin power management is the device, a hardware component the power of which can be adjusted independently of system power. A device may have different power states associated with it, at least two - on and off. Darwin associates several attributes to the power state of each device:

- Power used by the device in that state





(a) Power domain hierarchy (b) Darwin power management architecture

Fig. 6. Darwin power management

- Capabilities of the device in that state
- Power required to move the device to the next higher state (currently unused)
- Time required to move the device to that state (currently unused)

Another entity in Darwin power management is a power domain, a switchable source of power in the system providing power to one or more devices that are considered part of the power domain. Power domains dictate the highest power state for the devices they contain, and like devices, they have a range of power states, with at least on and off states supported. Power domains are hierarchical, with the top-level domain being the root power domain, which represents the main power of the system. Figure 6(a) presents an example of the power domain hierarchy.

Darwin defines two supervising entities for devices and power domains: policy makers and power controllers. Policy makers are the objects responsible for deciding when to change the power state of devices and power domains, based on different factors. The major factor in this decision is device idleness. When the system detects that a device is idle, it will try to reduce its power state, and policy makers are the entities responsible for deciding when to change the power state of a device or domain. However, the entities responsible for implementing these changes are power controllers. Other factors that the policy makers take into account include the aggressiveness, which is defined by the user in different contexts (such as AC-plugged or running from batteries) or by the system itself (such as when the battery reaches a critically low state).

A power controller knows about the power states of a device and can steer the device between them. Additionally, it reports power-related information of a device to the policy maker to assist in decision-making. Figure 6 presents an overview of the interoperation between the different Darwin power management entities.

There are a few fundamental differences between policy makers for devices and power domains. First, policy makers for power domains do not alter the power consumption of devices, but the character of the power supplied to members of the domain. This may imply adjusting clock rates or voltages, for example. Secondly, policy makers for power domains do not base their decisions on idleness like device policy makers. Instead, they base their decisions based on requests from their members (which include policy makers and power controllers). Therefore, policy makers are responsible for the power supplied to the domain they supervise, and they request the power state for the domain they belong to.

The power management architecture also provides support for notification of power state changes to interested entities. Power controllers are automatically notified of power state changes. Other interested objects may include driver objects, that will need to subscribe to the notifications by implementing a couple of function callbacks. In addition, user processes may also request notifications of system and device power events.

V. Discussion

There seems to be a clear gap between the state of the art and the state of the practice. While several studies show that application input is essential for effective power management, operating systems are slow adopting the results of power management research.

Symbian OS offers a comprehensive framework for power management, yet fails to reflect application requirements, assuming that a hardware view offers a complete enough picture of the power requirements. This effectively turns Symbian power management into a reactive system rather than proactive, therefore missing power saving opportunities. It is a clear example of autonomous power management. In addition, while the framework is designed to provide an accurate picture of the device power consumption, this is rarely achieved in practice. Hardware may exhibit complex power consumption patterns and manufacturers tend to neglect power consumption notifications. Symbian OS power management is typically reduced to a device management framework for system startup and shutdown events.

Windows Mobile power management takes instead a requester-controlled approach, with the implied drawbacks. The framework relies on deice drivers to take the most intelligent decisions regarding device power states, which may not always be possible due to their narrower scope with respect to the system. In addition, the abstraction level offered to applications is not adequate [1]. The system should be ultimately responsible for resource management, albeit through collaboration with applications.

Darwin provides a very simple approach to power management. Idleness and user-defined aggressiveness are the major factors governing policy making. However, I argue that waiting for an idleness timer to expire is a potential energy waste and should be avoided. Application inputs are not considered at all, with the implied loss of power-saving opportunities.

It seems that leading mobile operating systems do not exploit the advantages offered by application adaptation. This could provide significant power savings as exposed in III. Additional techniques such as idle period notification and ghost hints could provide additional savings. Given that power efficiency is taking a front seat in mobile OS design, the techniques presented provide promising methods to reduce the burden on the battery.

VI. Conclusions and further work

This paper has presented a survey of the published research on software adaptations for energy efficiency. In addition, I reviewed three mainstream mobile OSs and the power management functionality they offer. There seems to exist a big gap between the state of the art and the state of the practice, as evidenced by the lack of application input to power management interfaces. Operating systems could greatly benefit from the inclusion of the reviewed techniques in their design, since power management seems to play an increasingly important role in driving OS design. Battery technology cannot provide the necessary improvements to increase or even maintain the battery life of mobile devices, that face an increasing amount of power-hungry features and hardware components. Software needs to become more energy-efficient, and collaboration with the OS is a promising area for achieving this goal.

References

[1] Y.-H. Lu, L. Benini, and G. D. Micheli, "Requester-aware power reduction," in *International Symposium on System Synthesis*. Stanford University, September 2000, pp. 18–23. [Online]. Available: http://doi.acm.org/10.1145/501790.501796

[2] T. Simunic, L. Benini, and G. De Micheli, "Energy-efficient design of battery-powered embedded systems," in *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'98)*, June 1998. [Online]. Available: http://www.acm.org/pubs/articles/proceedings/dac/313817/p212-simunic/p212-simunic.pdf

[3] T. K. Tan, A. Raghunathan, and N. Jha, "Software architectural transformations: A new approach to low energy embedded software," in *Proceedings of the Conference on Design Automation and Test in Europe (DATE'03)*, 2003. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1253742

[4] B. Noble, "System support for mobile, adaptive applications," *IEEE Personal Communications*, vol. 7, no. 1, pp. 44–49, February 2000. [Online]. Available: http://www.cs.cmu.edu/~coda/docdir/ieeepcs00.pdf

[5] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," in *Proceedings of the Seventeenth Symposium on Operating System Principles (SOSP'99)*, December 1999. [Online]. Available: http://portal.acm.org/citation.cfm?doid=319151.319155

[6] C. Ellis, "The case for higher level power management," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS'99)*, March 1999. [Online]. Available: http://www.cs.duke.edu/~carla/ellis.pdf

[7] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile application-aware adaptation for mobility," in *Proceedings of the Sixteenth Symposium on Operating System Principles (SOSP'97)*, Saint Malo, France, 1997, pp. 276–287. [Online]. Available: http://portal.acm.org/citation.cfm?id=269005.266708

[8] C.Ellis, A. Lebeck, and A. Vahdat, "System support for energy management in mobile and embedded workloads: A white paper," Duke University, Department of Computer Science, Tech. Rep., October 1999. [Online]. Available: http://www.cs.duke.edu/~carla/research/whitepaper.pdf

[9] M. Anand, E. B. Nightingale, and J. Flinn, "Ghosts in the machine: Interfaces for better power management," in *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (MOBISYS'04)*, June 2004. [Online]. Available: http://www.eecs.umich.edu/~anandm/mobisys.pdf

[10] A. Weissel, M. Faerber, and F. Bellosa, "Application characterization for wireless network power management," in *Proceedings of the International Conference on Architecture of Computing Systems (ARCS'04)*, January 2004. [Online]. Available: http://citeseer.ist.psu.edu/649995.html

[11] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini, "Application transformations for energy and performance-aware device management," in *Proceedings of the Eleventh Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, September 2002. [Online]. Available: http://dx.doi.org/10.1109/PACT.2002.1106011

[12] ——, "Code transformations for energy-efficient device management," *IEEE Transactions on Computers*, vol. 53, no. 8, August 2004. [Online]. Available: http://www.cs.rutgers.edu/~ricardob/papers/tc04.pdf

[13] Y.-H. Lu, L. Benini, and G. D. Micheli, "Power-aware operating systems for interactive systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 2, April 2002. [Online]. Available: http://dx.doi.org/10.1109/92.994989

[14] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba, *Advanced Configuration and Power Interface Specification 3.0a*, Dec. 2005. [Online]. Available: http://www.acpi.info/DOWNLOADS/ACPIspec30a.pdf

[15] "Power management framework," Symbian Developer Library. [Online]. Available: http://www.symbian.com/developer/techlib/v70docs/sdl_v7.0/doc_source/baseporting/KernelProgramming/PowerManagementFramework/index.html

[16] J. Looney, "New Power Manager States in Windows Mobile V5 and How to Use Them," CLI327. Mobile & Embedded DevCon 2005. Presentation.

[17] "Power management," Apple Developer Connection. [Online]. Available: http://developer.apple.com/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/PowerMgmt/chapter_10_section_3.html

[18] "Power management for Macintosh; getting started," Apple Developer Connection. [Online]. Available: http://developer.apple.com/technotes/tn2002/tn2075.html

[19] "Technical Note TN2075. Power Saving Features for the PowerBook G4 computer," Apple Developer Connection. [Online]. Available: http://developer.apple.com/documentation/Hardware/Developer_Notes/Macintosh_CPUs-G4/PowerBook_G4Apr02/1Introduction/Power_Saving_Features.html#TPXREF115

Compiler memory energy optimizations

Peter Majorin
Software Technology Laboratory/TKK

Abstract

The energy consumption of memories is the most energy-consuming part in a processor architecture of embedded systems. The most promising technique so far to get large memory energy savings is to replace processor caches that are close to the processor core with software allocated on-chip memories or use them together. Numerous research has shown that scratchpads which are software controlled fast on-chip memories, can save significantly energy compared to systems using only caches, even with simple allocation algorithms. In this paper, we give an overview of the state-of-the-art scratchpad allocation algorithms for single- and multiple scratchpad configurations, and treat briefly scratchpads used in multitasking environments, and some complementary techniques to scratchpad allocation. We also give an overview of frameworks and compiler analyses designed for such memory optimizations.

1. Introduction

The memory subsystem is one of the most energy-consuming part in the processor architecture of several embedded systems. Furthermore, instruction fetching alone takes a large part of the total energy consumed by a processor (typically 27-50% depending on the processor). Data accessed by the processor from memory may consume up to 43% of the energy [15, 2]. Therefore, large energy savings can be obtained by designing an appropriate memory hierarchy that consumes as little energy as possible. As a rule of thumb, smaller memories consume less energy than larger, so we need to find ways of utilizing this small space as well as possible. The energy consumption in general is problematic in battery-operated embedded systems (especially portable), which usually have limited energy resources.

However, traditional compiler optimizations focus on speed and performance, and do not usually take into account the memory hardware. Optimizing for speed and code size is not the same as optimizing for energy. Therefore, different compiler optimization strategies targeted for energy savings are needed. In this paper we give a broad survey of the state-of-the art memory energy optimizations developed so far.

The paper is organized as follows: we first go through on-chip memories and their relevance for memory optimizations, and then present some program analyses commonly needed in memory optimizations. Then we present a short survey of state-of-the-art scratchpad allocation algorithms, and techniques behind the allocations. At last we present some research done on frameworks for compiler memory optimizations, and finally we conclude.

2. Low-power on-chip memories used for memory energy optimizations

We give here an overview of a few hardware memory components that can be used to save energy. Scratchpads and loop caches are the most common low-power memories found in the literature regarding memory optimizations, e.g. [17, 15, 6]. Although a usual processor cache is also a low-power on-chip memory, it is somewhat uninteresting from a compiler's point of view, because it is not usually controlled in software. However, caches cannot be ignored if present along with a scratchpad [20] for optimal energy savings.

2.1. Scratchpad

A scratchpad is a relatively small and fast on-chip memory (or part of it) that is accessed explicitly. It is also referred to as *tightly-coupled memory (TCM)*, as it lies in the vicinity of the CPU. Scratchpad memory (SPM) can be used to store both data and code. Some modern architectures have separate scratchpad memories for instructions and data. The size of the scratchpad memory is of the same magnitude as a level 1 cache (in practice some kB or smaller).

Scratchpads consume significantly less energy than caches do, because they lack the control logic of caches, and because of this missing logic they are totally predictable. While caches also reduce energy consumption, they also introduce predictability problems, which is an issue for real-time systems. The downside with scratchpads is that they must be allocated by the programmer or by a compiler or optimizer tool in a compiler-controlled way. However, once allocation has been done properly, scratchpads have demonstrated their superiority over caches in energy consumption, predictability and performance even with simple allocation algorithms.

Data transfer to and from a scratchpad can be performed by direct copying by usual CPU instructions (some overhead) or by hardware support (DMA), which is more efficient [9]. Operating system calls can be used to allocate scratchpads in a multitasking system, or just access the hardware directly in simpler systems.

2.2. Loop cache

A loop cache [6] is another low power on-chip memory, which is more limited than a cache, and therefore consumes less energy. In contrast to scratchpads, loop caches can be hardware controlled, but they can only contain code. As the name implies, it is used to store loop code; the loop code must fit entirely into the loop cache to be effective. A loop cache can also be software controlled, and it is then equivalent to a scratchpad that can only store instructions.

3. Compiler analyses for memory optimizations

3.1. Static and dynamic analyses

For memory optimizations, static and dynamic analyses can be used; both have their advantages and disadvantages, and at best they can be used to complement each other[5, 16].

Dynamic analyses cannot usually capture all possible program executions, but are easy to perform; the program to be profiled is just run with various inputs a number of times to obtain the program run-time behavior. In a particular run, everything about the program can be found out, including the memory accesses done and where program execution time is spent, but this information is obviously only valid for that particular run of the program.

Static analyses on the other hand attempt to find out program run-time behavior by considering all program executions at the same time. This makes static analyses sound (can capture all program executions), but in practice program semantics must be approximated to make the analyses feasible, also resulting in inaccuracies of the results. Moreover, over a decade of research in automatic flow analysis shows that statically analyzing program behavior to obtain accurate information is a very difficult problem for larger programs.

But for memory optimizations, an accurate knowledge about where program execution time is spent is essential, so program profiling should be used instead of inaccurate static analyses. To obtain results that generalize reasonably well over all program executions, many program runs have to be performed, which can be costly in terms of profiling time. Accurate profiling is needed because it is obvious that most program execution time is spent in loops, but this information is not enough: we need to know more precisely *which* loops are important and where *exactly* in the loops most time is spent (there is not much room on on-chip memories).

3.2. Loop analysis

Loop analysis [1] is a static analysis which compilers use to locate important code to be optimized. In the context of energy optimizations loop headers can be used as a basis for growing *traces* (Section 3.3). If the on-chip memory is large enough, or the loop is small enough, entire loops can be placed on the on-chip memory at once.

3.3. Trace analysis

Traces [14] (frequently executed straight-line sequences of basic blocks) can be used in the context of memory optimizations. Traces can be generated from the profiling data and static loop analysis; the loop header provides a starting point to grow the trace from. A trace is terminated when its tail execution frequency decreases below a certain fixed threshold value as compared to the header execution frequency. An advantage with a trace is that it can cross procedure boundaries so that opportunities for saving energy at the interprocedural level are not missed. Furthermore, the trace building must be tailored to a certain memory hierarchy; the size of the trace must not exceed a SPM and caches must be taken into account if they are present [20].

3.4. Statistical measures

Instead of performing a structural analysis on code to identify loops and to build traces out of these, the authors in [9] suggest a novel heuristic they call *concomitance*. This is a statistical measure of the temporal correlation between blocks of instructions. The advantage with this method is that it can capture hot spots in the program without needing to identify the structure of the program. Traces are still needed as profiling data as with the other methods.

4. Scratchpad allocation algorithms

The allocation algorithms presented here optimize energy consumption with respect to average case energy consumption (ACEC); also other optimization criteria exist such as worst case energy consumption (WCEC) [10]. Optimization with respect to energy consumption may result in performance improvements as well; this is usually the case when doing memory optimizations because low-power memories are usually faster as well.

The inputs for all the allocation algorithms are a power model for the instructions and the hot spots of the program to be optimized in the form of basic blocks, procedures, loops, traces and global variables. The energy savings obtained by scratchpad allocation is often compared against a cache of a similar size, or against a static allocation (Section 4.1), if appropriate.

Dynamic data structures such as a stack and heap memory remain problematic, because their sizes are not usually known at compile-time. However, these issues have received some research so far [3].

4.1. Static Allocation

Static allocation has been studied extensively, e.g. [17]: the contents (what variables and code) of the scratchpad is loaded in the start of the program and this allocation remains unchanged during program execution. The problem to solve is the integer knapsack problem: ILP (integer linear programming) or dynamic programming can be used to optimally solve this problem: the selected procedures, traces, loops and variables based on profiling data and energy model are placed on scratchpad so as to optimize its filling.

Static allocation still finds use in studies of more complex environments such as multi-banked and cache-aware scratchpad allocations. It also serves as a benchmark to compare the effectiveness of dynamic allocations against.

4.2. Dynamic Allocation

Dynamic allocation is harder than static allocation, but it has been demonstrated to save more energy than static approaches. In dynamic allocation, the allocation of scratchpad can change during runtime. In contrast to static allocation, program points have to be identified in a program to load and evict the scratchpad. Several different approaches have been proposed for dynamic allocation [19, 18, 9], most of them being heuristic methods.

The motivation for dynamic allocation can be seen in Table 1. Smaller SPM sizes save more energy, because each fetch costs less energy, but on the other hand a smaller SPM can hold less code or data. Dynamic allocation can therefore save more energy than with static allocation, because it is able to utilize better the limited storage space. This is the case for larger programs that have several hot spots and alter between these. The conclusion is that very small SPM sizes may save most energy, but this is also program-dependent. We next present some state-of-the-art dynamic allocation methods.

Size (bytes)	fetch (SPM) (nJ)	fetch (I-cache) (nJ)
64	0.1803	0.2961
128	0.1888	0.3059
256	0.1980	0.4732
512	0.2188	0.4966
1024	0.2404	0.5233
2048	0.2748	0.5655
4096	0.3277	0.6351

Table 1. Energy consumed by a SPM vs. an I-cache (0.18 μm) of equal size from the CACTI model [15].

The approach by Verma *et al.* [19] is based on ILP, and solved in the following phases:

1. Determine candidate SP (scratchpad) objects as in static allocation (code and data)
2. Perform liveness analysis on the SP objects
3. Assignment of SP memory objects and their spill locations in code
4. Computation of memory addresses of the SP objects

Some of the above steps were approximated with heuristic methods, because they take a very long time to compute for larger programs if done with ILP. The authors report 26% average energy savings as compared to a static allocation method.

Here we see the typical structure of a dynamic scratchpad allocation algorithm: since the objects can be evicted from the scratchpad at any point in the program, we have to determine the live ranges of the objects to be able to reason about how long we must keep the objects on the scratchpad before we can evict them. We also have to determine the points in the program where to evict and load the scratchpad. Finally we have to decide where in the scratchpad we put the loaded scratchpad object.

The approach by Janapsatya *et al.*[9] is based on a statistical method and considers only instructions. Rather than using a structural approach to identify loops and traces, they identify temporally correlated blocks of code directly from traces. The authors report 41.9% savings in energy when compared to a similar sized cache. Part of this large saving comes from their SPM controller and DMA support for scratchpad transfers.

The approach by Udayakumaran *et al.* [18] annotates the program CFG (control flow graph) with timestamps to reason about the eviction/placement strategy. Both code and data objects are considered, as well the stack. Because data is considered, the authors use a run-time disambiguator to correct memory references, which causes some overhead. Energy savings of 31.3% on the average are reported as compared to a static method.

The approach by Ravindran *et al.* [15] considers only instructions, and uses an iterative liveness analysis of traces to hoist the allocation of traces upwards in the program CFG. This is because loading naively in all basicblocks of a trace at its entry will result in a sub-optimal allocation strategy that can be improved upon. The advantage with this method is that it is heuristic method and requires much less computation than solving an ILP problem.

The drawback with dynamic allocation is that it causes aliasing problems; memory references become invalid to SP, when its allocation is changed during program execution. Furthermore, dynamic allocation may also be a problem in strict real-time applications, because of the extra copy code inserted to handle the scratchpad at various program locations. In addition, if aliasing problems are addressed a run-time disambiguator costs additional processor time. To avoid aliasing, but still get some benefits from dynamic allocation, data could be allocated statically and code dynamically, largely avoiding the aliasing problems (this is still a problem with code called via function pointers).

4.3. Multi-banked scratchpad allocation

Dividing the memory hierarchy in several smaller memory banks, instead of using a single monolithic memory bank has many advantages. Smaller banks consume less energy per access and unused banks can be turned off to save static leakage energy. Furthermore, memory accesses can be made in parallel, giving additional performance [8].

Multi-banked scratchpad allocation has been studied in e.g. [22, 12].

The results of Wehmeyer *et al*. [22] show that using many smaller scratchpads instead of a large one becomes beneficial when a program is large enough to be able to utilize a bigger scratchpad size. Energy savings of up to 22% were reported as compared to a single scratchpad system for a total of 32 kB SPM size for both cases.

Kandemir *et al.* [12] on the other hand focus on optimizing array accesses in loop nests in a multi-banked scratchpad system in order to minimize leakage current loss. The motivation for this is that the speed and density of the CMOS transistors is expected to rise in future, so that static leakage management is expected to become very important. Their method is based on optimizing bank locality, which means that successive SPM accesses should come to the same bank as much as possible, which makes it possible to put the other banks in a low-power idle state for as long time as possible. Turning on and off the memory banks results in small performance penalties, but an average leakage energy of over 40% is saved over all the benchmarked programs.

4.4. Scratchpad allocation in a multitasking environment

Scratchpad allocation in multitasking systems has also been considered in [13, 4].

The problem considered in the first paper is how to choose an appropriate static allocation for code and data for a set of statically scheduled processes on a single scratchpad. Furthermore, the execution time of the processes and their energy consumption is assumed be known *a priori*. The goal is minimize the energy consumption over the *entire set* of processes.

The allocation strategies considered are *saving*, *non-saving* and *hybrid* (which is a mixture of the two previous). The saving approach allocates the single scratchpad completely to the active process, while the non-saving approach divides the scratchpad evenly among all processes. The hybrid approach uses a common memory area for all processes, but also areas that remain dedicated to certain processes. This reduces the overhead of context switches by having a common region which can be used for shared data that all processes use frequently. Therefore, context switches matter only for the saving and hybrid approaches, which cause some overhead, but allows the scratchpad to be better utilized, if it is small.

As a result, it was found that the non-saving approach worked best for large SPM sizes (1-4 kB), while the saving approach worked best for small SPM sizes (up to 512 B). The hybrid approach, on the other hand, worked well for all scratchpad sizes, but required most computational time. Energy savings of 9-20% were reported as compared to a non-saving allocation that does not attempt to minimize the energy consumption over all processes.

The second paper [4] presents a novel way of using a scratchpad in a virtual memory system with an MMU (memory management unit) to store swapped-in pages into an SPM. Their page allocation algorithm considers only a single process system and code allocation, but they state that their method is easily extended to a multi-process environment and data allocation. A 33% reduction in energy consumption is reported, as compared to fully-cached configuration.

4.5. Complementary scratchpad optimizations

In this section we cover some complementary memory optimizations that can be used together with a scratchpad to save even more energy than with a scratchpad alone.

A hardware-controlled loop cache has been studied together with an Instruction Register File [7]. It was found that allocating frequently used instructions into a register file along with a hardware-controlled loop cache can save more energy than using these in isolation. Here, we see that instead of a hardware-controlled loop cache, a scratchpad could be used, saving even more energy.

Processor caches have also been studied together with an SPM in [20], where the authors studied a memory hierarchy consisting of a scratchpad together with an I-cache and a static allocation algorithm of code objects was used. It was found that using a scratchpad along with a cache can result in poor energy savings if the cache behavior is not taken into account, resulting in needless cache thrashing. Therefore, cache misses and hits need to be taken into account when deciding what code objects to place on scratchpad. Their formulation of the problem is a nonlinear optimization problem, which consists of cache model represented as a conflict graph. This problem is then linearized and solved optimally and near-optimally as an ILP problem.

5. Compiler frameworks for memory optimizations

Recently, some proposals for a compilation and simulation framework for energy-aware (memory) optimizations have been made in [23, 11, 21]. A central question that arises in such a framework is at what program level are the optimizations performed and what intermediate representations are used. In addition, an energy profiler is needed to evaluate the optimizations, and finally a code transformer to transform the program to use the energy optimizations. The energy profiler can further be split into a CPU simulator, an energy model, and a hardware model. Here, further consideration must be given to at what hardware level is the energy modeled (e.g. instruction-level). The memory hierarchy including caches and also the CPU pipeline need to be modeled in a cycle-accurate way to obtain an accurate energy estimate for a program.

The authors in [21] present a memory aware C compilation framework (MACC). This framework contains a simulation and a compilation part, where both have an access to an instruction-level energy database. The simulator models both the memory hierarchy (DRAM, cache and scratchpad) and the CPU in a cycle-accurate manner. The framework supports both static and dynamic scratchpad optimizations at the assembly/linker level. The authors found source-level optimizations (C code) to be useful and complementary to memory optimizations performed at the instruction level. The source-level optimizations that were considered included *array partitioning* and *array tiling*; these optimizations split larger arrays into smaller arrays (if needed) such that the smaller part can be allocated to a scratchpad memory. Such optimizations would be difficult to perform at the binary level, and they would be hardware dependent.

On the other hand, the work of [11] presents an energy-aware compilation framework (EAC) and focus only on high-level energy optimizations, including memory optimizations. Array-dominated programs are common in DSP and multimedia applications, and large energy savings can be obtained by source-level optimizations. The authors take the view that simulations and profiling takes a too long time, so the code is analyzed statically instead, taking as input technology parameters (memory model, buses etc.). Validation of the methods is still performed with a simulator.

The previously mentioned frameworks have a drawback in that they are meant for research use, and are not yet mature for common use. In the author's opinion, the problem with these frameworks is also that they rely on previously developed components which do not necessarily fit well together, creating an unnecessarily complex tool chain with many intermediate formats. Also, integrating energy-awareness in a compiler architecture may not be a good idea in the long run, because it would tie the users to a specific compiler. Otherwise, energy-awareness would need to be integrated separately in each compiler.

6. Conclusions

Considerable research effort has recently (2001-2006) been invested in scratchpad allocation, and as a result this field begins to be mature for practical applications. Large energy savings using scratchpad-aware compilation were reported in many research papers. Energy savings of around 20-40% compared to a system with an equal-sized cache for a single scratchpad system have been demonstrated, depending on what allocation method was used.

Future research effort in this field should address the implementation of practical tools and frameworks that can be used to study energy savings of various scratchpad allocation techniques in combination with other complementary (memory) energy optimizations. In particular, attention should be paid to what intermediate representations are needed in such tools and what their interfaces are. Furthermore, retargetability is in practice a very important issue, given the amount of different development tools and hardware (DSPs and GPPs) that are used in embedded systems. As we have seen, energy aware-compilation frameworks have quite differing requirements than a traditional compiler framework. Therefore, it could be a good idea to separate the energy awareness into different tools instead of trying to integrate these parts into a compiler framework. This is possible at least for memory optimizations at the binary/linker level, including some scratchpad allocation algorithms. Source-level memory optimizations are more difficult to separate from a compiler architecture, however.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques & Tools*. Pearson Addison-Wesley, 2nd edition, 2007.

[2] Bruno Bouyssounouse and Joseph Sifakis. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, volume 3436 of *Lecture Notes in Computer Science*. Springer, 2005.

[3] Angel Dominguez. *Heap Data Allocation to Scratch-Pad Memory in Embedded Systems*. PhD dissertation, University of Maryland, College Park, Department of Electrical and Computer Engineering, 2007.

[4] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. In *Proceedings of the 7th International Conference on Embedded Software (EMSOFT'06)*, pages 321–330, Seoul, Korea, October 2006.

[5] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, Oregon, USA, May 2003.

[6] Ann Gordon-Ross, Susan Cotterell, and Frank Vahid. Tiny Instruction Caches for Low Power Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 2(4):449–481, November 2003.

[7] Stephen Hines, Gary Tyson, and David Whalley. Reducing Instruction Fetch Cost by Packing Instructions into RegisterWindows. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 19 – 29, Barcelona, Spain, November 2005.

[8] Jason D. Hiser. *Effective Algorithms for Partitioned Memory Hierarchies in Embedded Systems*. PhD dissertation, University of Virginia, School of Engineering and Applied Science, May 2005.

[9] Andhi Janapsatya, Aleksandar Ignjatovic, and Sri Parameswaran. Exploiting Statistical Information for Implementation of Instruction Scratchpad Memory in Embedded System. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):816–829, August 2006.

[10] Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. Estimating the Worst-Case Energy Consumption of Embedded Software. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 81–90, San Jose, California, USA, April 2006.

[11] I. Kadayif, M. Kandemir, G. Chen, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. Compiler-Directed High-Level Energy Estimation and Optimization. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):819–850, November 2005.

[12] Mahmut Kandemir, Mary Jane Irwin, Guilin Chen, and Ibrahim Kolcu. Compiler-Guided Leakage Optimization for Banked Scratch-Pad Memories. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(10):1136–1146, October 2005.

[13] Lars Wehmeyer and Urs Helmig and Peter Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: a first approach. In *3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTI-Media'05)*, pages 115–120, September 2005.

[14] Lian Li and Jingling Xue. Trace-based leakage energy optimisations at link time. *Journal of Systems Architecture*, 53(1):1–20, January 2007.

[15] Rajiv A. Ravindran, Pracheeti D. Nagarkar, Ganesh S. Dasika, Eric D. Marsman, Robert M. Senger, and Scott A. Mahlke. Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*, pages 179–190, San Jose, California, USA, March 2005.

[16] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto On-chip Memory. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, Kyoto, Japan, October 2002.

[17] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, Paris, France, March 2002.

[18] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic Allocation for Scratch-Pad Memory Using Compile-Time Decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):472–511, May 2006.

[19] Manish Verma and Peter Marwedel. Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors. *IEEE Transactions on Very Large*

Scale Integration (VLSI) Systems, 14(8):802–815, August 2006.

[20] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-Aware Scratchpad-Allocation Algorithms for Energy-Constrained Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):2035–2051, October 2006.

[21] Manish Verma, Lars Wehmeyer, Robert Pyka, Peter Marwedel, and Luca Benini. Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations. In *Proceedings of the 6th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'06)*, pages 279–288, Samos, Greece, July 2006.

[22] Lars Wehmeyer, Urs Helmig, and Peter Marwedel. Compiler-optimized Usage of Partitioned Memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI'04)*, pages 114–120, Munich, Germany, June 2004.

[23] Joseph Zambreno, Mahmut T. Kandemir, and Alok N. Choudhary. Enhancing Compiler Techniques for Memory Energy Optimizations. In *Proceedings of the Second International Conference on Embedded Software (EMSOFT'02)*, pages 364–381, Grenoble, France, October 2002.

Measuring the CPU energy consumption of a modern mobile device

Antti P Miettinen
antti.p.miettinen@nokia.com

Abstract

Modern mobile devices employ complex system-on-chip (SoC) processors as their main computing engine. Inside a SoC several functional blocks can share the same external supply voltage line. This makes measurement based study of a single block, e.g. the main CPU, challenging as the current drawn by an individual block cannot be measured directly.

The goal of this work was to characterize the feasibility of using simple board level current measurement instrumentation for studying the energy consumed by program code run on the ARM926 core inside the OMAP1710 SoC. The results indicate that by using carefully planned experimental setups energy consumption related to the following factors can be studied at least qualitatively:

- instruction path bit switching
- data cache reads and writes
- multiplier usage and data dependency
- register bank bit switching

However, planning the test setups and performing the measurements is quite time consuming and significant uncertainty remains in the error margins. SoC level instrumentation would certainly provide a much more reliable and powerful research tool.

1 Introduction

The environment for developing software for an embedded hardware target often consists of

- a macro board hosting
    - the hardware components present in the actual end user device

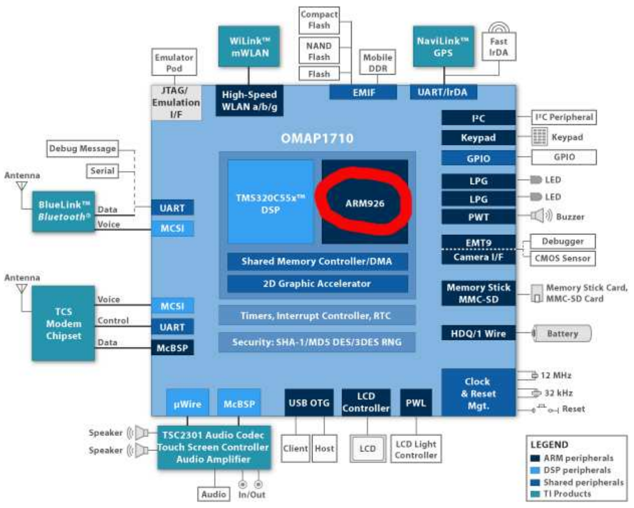


Figure 1: ARM926 core inside OMAP1710, picture from [1]

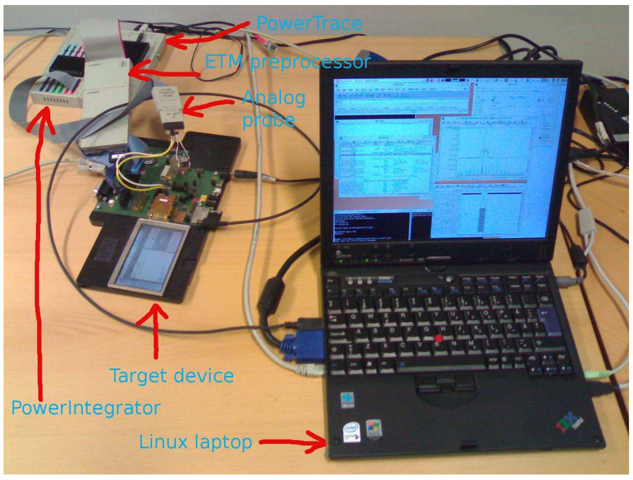


Figure 2: Embedded software development environment

  - additional trace and debug instrumentation
- in circuit debug and trace hardware
- development host with
  - cross compilation tools for the target device
  - software tools interfacing with the debug and trace hardware

Traditionally the instrumentation to facilitate development time activities is focused on satisfying functional requirements, i.e. the normal software development task of making the software work. However, the instrumentation can also be useful for addressing nonfunctional requirements. The goal of this work is to explore the extent to which energy efficiency can be addressed with this kind of simple board level instrumentation.

2 Tools and methods

2.1 Test environment

The hardware used in this work consist of a development board hosting an OMAP1710 [1] processor and various peripherals, including

- LCD display
- USB device port
- Serial console port
- MMC slot
- JTAG port [2]
- ETM port [3]
- jumpers in various voltage lines for connecting a shunt resistor for current measurement

The JTAG and ETM ports and the current measurement points are connected to a commercial debug, trace and measurement system developed by Lauterbach [4] Datentechnik GmbH. The system includes a PowerTrace module, a JTAG adapter, an ETM preprocessor module, a PowerIntegrator module and an Analog probe (containing an A/D converter). The system allows tracing and controlling the instruction execution of the ARM926 [5] core and simultaneus measurement of the supply line current with maximum sampling frequency of 625kHz.

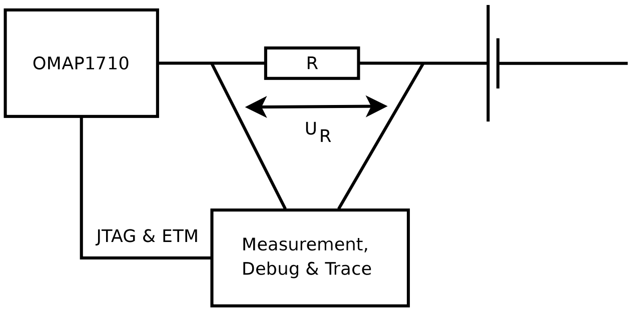


Figure 3: Measurement setup

A Linux laptop (Ubuntu/feisty [6]) was used to host the Trace32 software that interfaces with the debug and trace hardware. The toolchain for compiling software for the the ARM Linux based target environment was constructed with *crosstool* [7].

2.2 Workflow and methodology

For testing energy consumption of different primitives it is necessary to be able to run test programs on the target hardware and trace their execution and power consumption. For this purpose simple C programs with inline assembly sections were constructed.

The test programs were run under the control of the Trace32 debugger. The operating system awareness feature of the Trace32 debugger allows breaking execution when a named process is executed by the Linux kernel. This provides a convenient method for enabling tracing and measurement specifically for the code sections of interest: when the test program is loaded the trace buffer can be cleared and a breakpoint can be set at the end of the test code.

During the test run the ETM tracing was enabled and the Analog probe recorded the voltage drop over a shunt resistor placed to the supply line of the OMAP1710 processor. The supply voltage is constant so the voltage drop gives a direct measure of the current drawn by the processor:

$$I = \frac{U_R}{R} \tag{1}$$

Power can be obtained simply by multiplying (1) by the constant supply voltage:

$$P = UI \tag{2}$$

and energy by further multiplying (2) by time:

$$E_{insn} = Pt_{insn} \tag{3}$$

As the ETM trace provides a cycle accurate instruction execution trace the energy per instruction can be calculated from cycles per instruction (CPI) and CPU clock speed:

$$t_{insn} = \frac{CPI}{f}, \Rightarrow E_{insn} = P\frac{CPI}{f} \tag{4}$$

The ETM trace provides also a convenient method for choosing the current samples corresponding to execution of only the test code as the Trace32 is able to display the A/D samples in sync with the ETM trace.

Although the methodology seems trivial, the challenge lies in obtaining the power level that is representative for the power level of the factor of interest. As the OMAP1710 contains much more than just the ARM926, absolute power levels are not of interest. Another important issue is to asses the repeatability of the measurements because of unknown uncontrollable factors in the setup. A rough measure for the contribution of secondary factors can be obtained by measuring the power level when the ARM926 is in the wait-for-interrupt mode. This still includes power contribution from the ARM core but for the purpose of this work, the ARM idle power is also a secondary factor.

As the sampling frequency of the A/D converter is orders of magnitude smaller than the clock speed of the circuit being measured, the only feasible way to study instruction level effects seems to be contructing test programs that excercise the phenomenon of interest for sufficiently long time and use averaging to obtain a representative power value.

For estimating the error in the value obtained by averaging the straight forward approach is to use e.g. sample standard deviation. However, both the sample mean and standard deviation can give biased estimates. The current drawn by a digital circuit is by nature composed of current peaks. Sampling might get synchronized and therefore the average value could be biased.

For energy measurements it would make sense to perform the A/D conversion by integrating the current over the sampling period. This could be achieved by analog integrating circuitry but implementing this was not feasible because of time constraints of this work.

2.3 Test programs

2.3.1 General

As the goal of this work was simply to establish whether a given effect is measurable, the approach to the problem was more or less *ad hoc.* However, a brief look into ARM9 microarchitecture [8] is useful for constructing feasible tests.

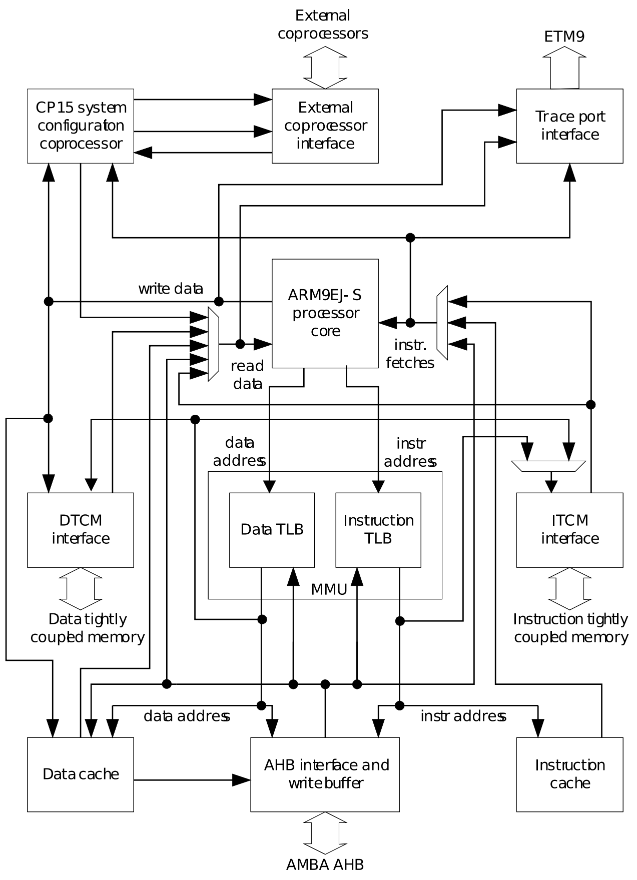


Figure 4: ARM926 overview from [9]

The ARM926EJ-S inside the OMAP1710 is a member of the ARM9 general-purpose microprocessors and is targeted for multitasking operating systems with full memory protection and virtual memory support. The caches and MMU can be significant power consumers and can be studied as primary factors but must also be addressed as secondary factors when studying e.g. core internal factors. The JTAG and ETM instrumentation also consume power so the test setup should strive to keep the contribution of those blocks as constant as possible.

The actual ARM9EJ-S [10] processor core implements the ARM v5TE architecture. This includes support for ARM, Thumb and Jazelle instruction sets and DSP instructions. The pipeline consists of five stages:

- Instruction fetch
- Decode

Fetch	Decode	Execute	Memory	Writeback
Instruction Fetch	ARM Decode: Reg. Address Decode, Register Read / Thumb Decode: Reg. Address Decode, Register Read	Shifter ALU	Memory Data access	ALU Result and / or Load data Writeback

Figure 5: ARM9 pipeline from [8]

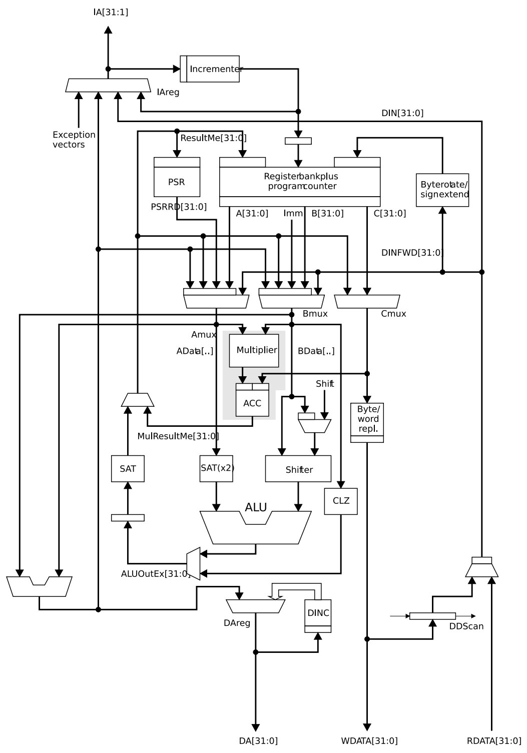


Figure 6: ARM9 data path from [11]

- Execute
- Memory access
- Register writeback

The datapath consists of a register bank (containing program counter) with three read ports and two write ports. Execution units include e.g. multiplier, accumulator, shifter and ALU blocks.

The dynamic power consumption of a digital circuit is relative to the switching capacitance:

$$P = \alpha \times C \times V^2 \times f \tag{5}$$

where α is the switching activity. We can use the overall architectural description of the ARM926 core to guide our guesses as to what affects the switching activity related to executing different code sequences. From the above description we can assume for example that the following factors could affect the energy consumed:

- Instruction fetch
- Data read
- Data write
- Functional unit activity
- Register access

To limit the scope of this work, the test programs were constructed so that instruction and data fetches occur always from cache, i.e. code sequences of test loops were kept smaller than instruction cache (32k) and data accesses were localized to areas smaller than data cache (16k). To maximize the proportion of instructions that excercise the factor under study test loops were contructed as instruction sequences of 4096 instructions repeating the effect being measured. The loop overhead was below ten instructions so the loop overhead effect on power should be quite small.

2.3.2 Instruction fetch

One could assume that for instruction fetch the amount of alternating bits in consecutive instruction words would affect the amount of switching in the instruction pipeline. Therefore, a feasible test series would be a set of test cases where the number of alternating bits between consecutive instructions is varied and the power level for each case is measured. However, the test should also be constructed so that the amount of bit switching in the instruction words does not affect the switching activity in the data path.

The nop instruction is probably the first thing that comes to mind. The ARM ISA does not actually define any dedicated nop instruction. Instead no-operation instruction execution can be achieved by e.g. moving the value from register zero to register zero. This allows constructing different instruction streams with no effect. However, it turns out that the *functionally no-operation* `mov rx,rx` instruction is not short circuited in any way by the instruction decode/execution logic and does actually cause data path activity. This can be observed by measuring the power level of `mov r15,r15`, i.e. using the program counter as the operand register. Accessing the program counter causes clearly higher power level for the instruction stream than using any other register.

Fortunately most of ARM instructions can also be conditionally executed. The data path activity can be prevented by using a condition code which never gets true during the test.

Two test cases were constructed for measuring the effect of bit toggling in instruction words. One test varies the operand register of a `mov rx,rx` instruction allowing varying the number of alternating bits with values 0, 2, 4 and 8 bits. Another test varied the immediate value of a `mov r0,#imm` instruction allowing bit variation between zero and eight bits. In both tests a condition code forced to false was used to eliminate the data path contribution.

2.3.3 Data access

Simple tests were constructed for exercising `ldr` and `str` instructions accessing repeatedly the same address. For isolating the data access power contribution from instruction execution, the test sequences were run with conditional execution with always true condition and never true condition.

2.3.4 Functional unit activity

For exercising the data path, the instruction path switching is the secondary factor to be eliminated. Instruction fetch bit switching variation is minimized by using long test code sequences of exactly the same instruction. Simple tests excercising the `add` and `mul` instructions were constructed as well as `mov` with shift for exercising the shifter.

2.3.5 Register access

For testing the effect of bit switching in the register bank with minimal functional unit contribution, a test program moving values to `r0` alternatively from `r1` and `r2` was constructed. The test was repeaded with different values stored to registers `r1` and `r2`. As the actual code sequence does not change, only the values in the registers, the instruction path contribution should remain constant and the variation should be due to the datapath.

3 Results

3.1 General

As the measured absolute power levels are not of interest for this work and even absolute differences between different test cases can be misleading, choosing the metric to describe the energy consumption of different factors is not very straight forward. In the following presentation the energy and power values are presented as *number of standard deviations above idle* level, where the standard deviation is taken from the idle power level measurement. The variation of the standard deviation was very small between different tests so this unit closely matches the error margin of the different tests.

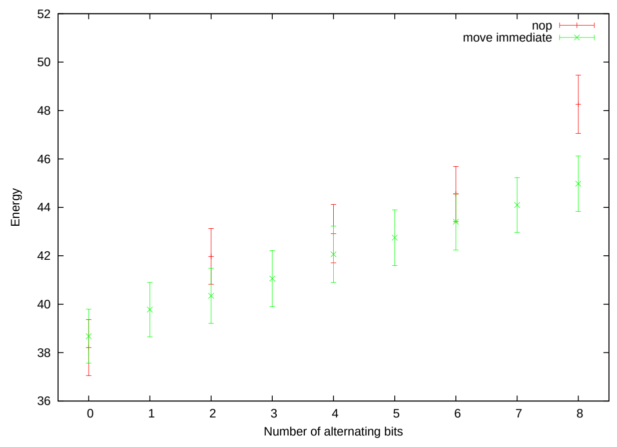


Figure 7: Instruction path bit switching effect

3.2 Instruction fetch

Figure 7 shows the relative energy of `mov rx,rx` and `mov r0,#immediate` instructions as a function of the number of alternating bits. As can be seen the trend is very clear and the two test cases agree reasonably well. Using register `r15` (program counter) seems to deviate slightly from the other data points, which could suggest that PC might be handled specially already in the instruction decode stage.

3.3 Data access

The relative energies for load and store instructions are shown below:

instruction	data	taken	not taken
`ldr`	all zeros	70 ± 1	39 ± 1
`ldr`	all ones	72 ± 1	39 ± 1
`str`	all zeros	63 ± 1	38 ± 1
`str`	all ones	63 ± 1	38 ± 1

As can be seen, the data cache access is clearly measurable and the difference between reads and writes is also clear. A surprising finding is that apparently *all ones* data takes slightly more energy than *all zeros* data (at least for reads). Similar phenomenon was observed in other tests too but usually the effect was well below error margin. In principle, the value of constant data should not affect the switching activity.

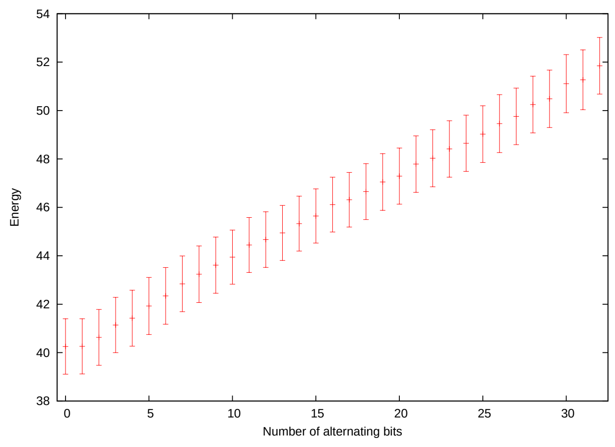


Figure 8: Register bank bit switching effect

3.4 Functional unit activity

The relative energies for the tested data processing instructions are shown below:

instruction	minimum	maximum
`mul`	2 × (48 ± 1)	2 × (62 ± 1)
`add`	38 ± 1	39 ± 1
`mov` with shift	39 ± 1	39 ± 1

The instructions were run with different data values as operands and as can be seen multiplication shows significant data dependecy. High cost of multiplication is clearly measurable already before taking CPI into account. On the other hand it seems that adder and shifter consume neglible energy (the power level difference to e.g. nop is below error margin).

3.5 Register access

Figure 7 shows the relative energy of `mov r0,r1`, `mov r0,r2` code sequence as a function of the number of alternating bits in the values in registers `r1` and `r2`. As for instruction path, the trend is very clear.

4 Conclusions and future work

Before this work I had doubts about the feasibility of using the SoC supply line current for measuring core level effects. However, the measurements indicate that at least rudimentary analysis can be performed with carefully planned and performed tests.

Even though one needs to be careful about drawing conclusions about even the relative differences it seems that the only significant functional unit energy-wise is the multiplier. The effects of e.g. bit switching in the instruction path and in the register bank as well as the data cache reads and writes are clearly measurable but with the performed tests it was not possible to measure the effects of e.g. shift and addition.

The measurements performed in this work were quite limited. For example comparing ARM and Thumb execution was completely omitted as was all testing related to the branch instructions. Also only addition was tested as an arithmetic-logic operation. Possible future work could include more complete tests and comparisons to e.g. ARM11 and Cortex architectures.

References

[1] Texas instruments OMAP1710 overview. http://focus.ti.com/...

[2] Joint test action group, standard test access port and boundary-scan architecture, ieee 1149.1. http://en.wikipedia.org/wiki/JTAG.

[3] Embedded trace macrocell architecture specification. http://www.arm.com/pdfs/...

[4] Lauterbach datentechnik GmbH debug and trace products. http://www.lauterbach.com/.

[5] ARM926EJ-S technical reference manual. http://arm.com/pdfs/DDI0198D_926_TRM.pdf.

[6] Ubuntu, community developed linux-based operating system. http://www.ubuntu.com/.

[7] Dan Kegel. Building and testing gcc/glibc cross toolchains. http://kegel.com/crosstool/.

[8] The ARM9 family - high performance microprocessors for embedded applications. In *ICCD '98: Proceedings of the International Conference on Computer Design*, page 230, Washington, DC, USA, 1998. IEEE Computer Society.

[9] ARM926EJ-S product overview. http://www.arm.com/pdfs/DVI0035B_926_PO.pdf.

[10] ARM9EJ-S technical reference manual. http://www.arm.com/pdfs/DDI0222B_9EJS_r1p2.pdf.

[11] ARM9E-S (rev 2) technical reference manual. http://www.arm.com/pdfs/DDI0222B_9EJS_r1p2.pdf.

Energy-Aware Scheduling

Juhani Peltonen
Software Technology Laboratory/TKK
jopelto2@cc.hut.fi

Abstract

Because of increase in processing power and slow development in battery technology in contrast to circuit technology, battery life has become the most limiting factor in mobile devices. One of the most energy consuming part is the processor. A common mechanism to lower processor's energy consumption is dynamic voltage scaling (DVS) which scales both voltage and frequency. There exist numerous different scheduling algorithms which try to minimize the processor's energy consumption. This paper considers several algorithms for energy-aware scheduling. The algorithms are EDF and RM for hard real-time systems, one interval and stochastic scheduling algorithm for soft real-time systems and two compiler-assisted and procrastination scheduling algorithms also for soft real-time systems. The considered algorithms use dynamic voltage scaling to adjust processor's performance.

1. Introduction

Due to increase in processing power battery life has become the most limiting factor in mobile devices. The part, that accounts significantly for total energy consumption, is the microprocessor. It is typical, however, that the peak performance of the processor is rarely needed, which means that most of the time a lower performance processor would suffice. A common technique to lower processor's energy consumption is dynamic voltage scaling (DVS).

DVS is a technique to lower processor's performance. The idea is to run the processor at lower speed by reducing both voltage and frequency. Because of non-linear relationship between voltage and consumed energy, running at low speeds saves energy. Performance scaling, however, interferes with real-time requirements of applications and can cause deadline misses. That is why care should be taken when the performance is scaled.

This paper concentrates on DVS scheduling algorithms. Focus is on real-time systems. The rest of the paper is organized as follows: section 2 gives some background to real-time systems and dynamic voltage scaling. Section 3 discusses earliest deadline first and rate monotonic scheduling algorithms for hard real-time systems both to which DVS is integrated. Section 4 considers interval scheduling in which system load is monitored at fixed intervals. Section 5 discusses stochastic soft real-time scheduling for applications with statistical performance requirements. Section 6 discusses some compiler-assisted scheduling algorithms. Section 7 considers procrastination scheduling which tries to take leakage current into consideration. Finally, section 8 discusses about feasibility of different scheduling algorithms.

2. Background

2.1 Real-Time Systems

Real-time systems [9] are systems which run applications which have deadline requirements. The system should ensure that deadlines are not missed.

Real-time systems can be divided into two categories: hard and soft. Hard real-time systems differ from soft ones in that in hard real-time systems all deadlines should be met. In soft real-time systems, missing a deadline doesn't cause damage but decreases effectiveness. Soft real-time systems typically provide statistical performance guarantees (i.e. meeting deadlines at certain probability). Multimedia applications are typical examples of applications which have soft real-time requirements. For example, for an application, which displays a video, it is not crucial to display every frame on time, although it is preferable.

The third group of applications, besides the ones which have either hard or soft real-time requirements, are best-effort ones. Best-effort applications are applications which have no performance requirements. They are not considered in this paper.

2.2 Dynamic Voltage Scaling

Dynamic voltage scaling is a common method to adjust processor's energy consumption. It exploits the following characteristics of CMOS-based circuits: the consumed energy is proportional to the square of the voltage and the maximum frequency is determined by the voltage [2]. In other words, by using lower frequencies it is possible to use lower voltages which leads to savings in energy. Although the processor's performance could be scaled by scaling just the frequency, energy savings are likely to be negligible because of linear relationship between energy consumption and frequency.

Current processors (e.g. Intel Pentium M [5]) have only some discrete speed settings. This means that it is not always possible to find the optimal speed. Usually, the speed to be used is acquired by rounding the calculated speed upwards to the closest matching one. Some algorithms, that are discussed later, benefit from small amount of speed settings in a way that they consume less energy compared to a situation in which there is more speed settings available. Some other algorithms behave just the opposite.

The usage of DVS implies that using a lower performance saves energy. However, as is shown in [11], this doesn't always hold but, in some cases, it is more energy efficient not to reduce voltage below a certain point. In other words, more energy can sometimes be saved by running at a higher speed and then idle rather than running at a lower speed and idle less. Also, as technology scales, leakage current becomes more and more significant decreasing the effectiveness of DVS [3]. DVS scheduling algorithms can take the aforementioned facts into consideration to increase energy savings.

3. EDF and RM

This sections discusses the approach to DVS scheduling described in [13], in which DVS was integrated to earliest deadline first (EDF) and rate monotonic (RM) scheduling algorithms.

Earliest deadline first (EDF) [8] is a dynamic, preemptive scheduling algorithm for hard real-time systems. EDF assigns priorities to tasks according to their deadlines such that the task, which has the earliest deadline, gets the highest priority and the task, which has the latest deadline, gets the lowest priority. The task with the highest priority will be executed. For a set of tasks to be schedulable, EDF scheduler requires that the processor utilization factor doesn't exceed one i.e. $U = C_1/P_1 + \cdots + C_n/P_n \leq 1$ [8], where C_i denotes the execution time, P_i the period and C_i/P_i the fraction of processor time spent in the execution of task i.

Rate monotonic (RM) [8] is a fixed priority, preemptive scheduling algorithm which assigns priorities according to task periods. The task with the shortest period gets the highest priority. For a set of tasks to be schedulable, it must be that $U = C_1/P_1 + \cdots + C_n/P_n \leq n\left(2^{1/n} - 1\right)$.

The first approach taken in [13] was static voltage scaling which simply selects the lowest possible execution speed with which a given task set remains schedulable. All tasks in the set are executed with the selected speed. The worst-case execution times (WCETs) [15] are scaled according to the new utilization factor U by amount of $1/U$. The selected processor utilization is changed only when the task set is changed. The approach is simple, but doesn't take into consideration the fact that tasks typically use less cycles than their worst-cases require. Because it is typical that tasks finish earlier than their WCET based deadlines, static scaling loses energy by executing tasks with a speed that is too high.

The second approach, cycle-conserving RT-DVS (real-time DVS), exploits the fact that a typical execution time for a task is generally much less than its WCET. The name, cycle-conserving, implies that instead of wasting cycles by idling, cycles are conserved by reducing performance. The idea of the algorithm for EDF is as follows: if a task T_i completes earlier than its WCET, the processor utilization is recomputed by using the actual execution time cc_i in place of the WCET C_i. Because the new utilization factor is less than the previous one, there is a possibility to use lower performance. The performance scaling doesn't affect schedulability of the task set, until the task T_i gets re-scheduled, because the schedulability test continues to hold. When the task T_i gets re-scheduled, the actual execution time cc_i is replaced by the WCET C_i. At this point, the utilization can increase. Because several tasks can finish early, the cycle-conserving algorithm can gain significant energy savings.

Cycle-conserving RT-DVS for RM is a bit different because of $O(n^2)$ (where n is the number of tasks to scheduled) requirement for the schedulability test. The same approach, that calculated a new utilization for cycle-conserving EDF, could be used, though. The algorithm starts with a utilization obtained from the static scaling algorithm which uses WCETs to calculate the utilization. After that, the work that should be accomplished before the next deadline is spread out between the current time and the next deadline. After a task has been executed the remaining work is again spread out between the current time and the next deadline. This procedure is repeated.

The third approach, look-ahead RT-DVS EDF, tries to defer as much work as possible so that it is now possible to use as low utilization values as possible. In contrast to cycle-conserving RT-DVS, look-ahead RT-DVS starts from low speeds instead of high ones. The idea of the algorithm is as follows: the algorithm pushes as much work over the next deadline as possible and computes the minimum number of cycles needed before the next deadline such that no future deadlines are missed. The execution speed is set just high enough to execute the selected number of cycles before the next deadline. Because the algorithm starts from a low utilization it is possible that high utilizations are required later. If tasks tend to finish early then it is likely that no highest utilizations are needed.

Based on the simulation results in [13], energy savings were significant. The available voltage and frequency settings of the processor had large impact on the effectiveness of algorithms. Because look-ahead EDF tries to defer work, large amount of settings allowed the algorithm to match closely to the desired performance level which required high performance settings later increasing energy consumption. With fewer number of settings, less work was deferred requiring less high performance processing later. Static and cycle-conserving EDF benefit from large number of settings because the more setting there were, the more closely they could match the calculated performance. When applications' used less computation time when their WCET, cycle-conserving RM didn't adapt that well to the situation. Instead, cycle-conserving and look-ahead EDF showed significant reductions in energy consumption. Simulation results showed that look-ahead and cycle-conserving algorithms significantly reduce total energy consumption. Measurements on actual platform reduced energy consumption from 20% to 40%.

4. Interval Scheduling

This sections discusses two intervals schedulers, PAST and AVG$_N$, originally proposed by [14], and is based on the work in [4].

Interval schedulers work by performing predictions of future system loads and scalings at fixed intervals. At each interval, the scheduler predicts the processor utilization based on one or more preceding intervals. In PAST, the predicted load for the current interval is set to be as high as the last interval. In AVG$_N$, where N is the number of intervals to be averaged, a "weighted utilization" at time t, W_t, is calculated as a function of the utilization of the previous interval U_{t-1} and the previous weighted utilization, W_{t-1}, such that $W_t = (NW_{t-1} + U_{t-1}) / (N+1)$ [4]. AVG$_0$ corresponds to PAST.

Measurements in [4] were done using an actual test platform, the Itsy Pocket Computer, which ran a port of Linux operating system. The Linux kernel was modified to support the tested algorithms. The tested applications were for web browsing, text reading, chess and MPEG video and audio.

The best policy, that [4] found, was PAST (i.e. AVG$_0$), which scaled frequency such that it selected only the minimum or maximum frequency. The minimum frequency was selected when the processor's utilization dropped below 93%. The maximum frequency was selected when utilization was greater than 98%. Other policies were examined with $N > 0$ and by scaling frequency by a one step or by doubling or halving it. Reasons for PAST to be the best one were that it never missed any deadlines and that it also saved energy (although the amount of saved energy was small). Other policies didn't give good results. A problem with AVG$_N$ was that the response time of the system was increased with increased N. The greater the value of N was, the more time it took before the frequency was scaled when a change in the system load occurred.

PAST was later improved with PACE (Processor Acceleration to Conserve Energy) in [10]. PACE is not a complete DVS algorithm But a method to improve existing ones. In short, PACE changes the way how tasks are scheduled to decrease expected energy consumption without changing performance. In the modified version of PAST, energy consumption dropped significantly compared to the original one.

5. Stochastic SRT Scheduling

In this section, stochastic soft real-time (SRT) scheduling is discussed. Discussion is based on [17].

GRACE-OS, the scheduler presented in [17], is a scheduler for mobile devices that primarily run multimedia applications. Instead of requiring worst-case executions times, GRACE-OS uses a probability distribution of applications' cycle demands. The distribution is obtained at run-time. Although there are great variations in cycle demands of applications, their probability distribution is stable (or changes slowly and smoothly) indicating that it is feasible to perform stochastic scheduling based on demand distribution.

GRACE-OS scheduler consists of three components: a profiler, a scheduler and a speed adaptor. The profiler constructs the probability distribution by monitoring cycle usage of applications. The scheduler schedules tasks such that given performance guarantees are met. The speed adaptor adjusts CPU speed based on task's demands in order to save energy.

Cycles are allocated to applications by GRACE-OS as follows: for a task to meet its deadlines at probability ρ, the scheduler allocates C cycles such that the probability that a task doesn't use more than C cycles, is at least ρ i.e. $\mathcal{P}[X \leq C] \geq \rho$ [17]. The cycle count C for the task can be found from its probability distribution. After cycle counts are found, an earliest deadline first based scheduling algorithm is used to dispatch a task, which has the earliest deadline and a positive budget (the budget is set to C for every period and, as the task is executed, its budget is decreased by the amount of cycles it used). If a task uses its budget without finishing its job, the scheduler can either notify it to abort or let it to run in best-effort mode. In best-effort mode, a task can either be set to use surplus cycles from other tasks or to block until the next period.

The approach to scheduling, that GRACE-OS takes, is to start executing a task from a low speed and accelerate as execution progresses. The approach was taken because most multimedia tasks use less than their allocated cycles so it is possible to avoid the highest speeds. The scheduler uses a speed schedule, which is a list of scaling points, to scale the speed of the processor as follows: if a task has used n cycles, such that $n \geq x$, where x is a cycle count for speed y, speed is increased to y. The speed schedule is based on demand distribution and is calculated such that it minimizes the energy consumption while meeting statistical performance requirements.

Measurements in [17] were performed using a laptop which ran Linux. Applications were codecs for video, audio and speech. Linux kernel was modified to support the experiments. The new system calls incurred negligible overhead (from 0.0004% to 0.5%). Overhead of construction of the demand distribution depended on the size of the profiling window (i.e. how many jobs of a task are kept tracked) and the size of histogram groups. Overhead varied from 0.1% to 100%, which means that the demand distribution should be estimated rarely. Cost of the scheduling was at most 0.4%. Energy savings ranged from 10% to 72% with a single high-demand application and from 7% to 64% with multiple concurrent applications. GRACE-OS meets almost all deadlines in a lightly loaded system and meets deadlines within statistical requirements in a heavily loaded system.

6. Compiler-Assisted Approaches

This section describes some methods to DVS scheduling with a help of a compiler and is based on the work in [12] and [16].

The idea in compiler-assisted scheduling is that the compiler analyzes the program and provides information about it; for example, the compiler can insert information into the program which is used by the run-time scheduler.

In [12], checkpoints were inserted at loop boundaries and procedure call sites. Checkpoints were used to calculate the actual execution time of a program section such that the checkpoint at the beginning of the section recorded the current time and the checkpoint at the end of the section computed the actual execution time. The execution time was compared to the WCET and based on the result, the processor's speed could be adjusted if necessary. The following voltage adjustment schemes were used: no power management (NPM) where all tasks execute at maximum speed; static power management (SPM) where the minimum performance is precalculated based on WCETs and deadlines; dynamic power management-proportional (DPM-P) where the utilization of the processor is recalculated after every section, which allows further sections to slow down; dynamic power management-greedy (DPM-G) where all unused time is given to the next section and dynamic power management-statistical (DPM-S) where the speed of the processor is calculated based on the unused time so far and the unused time in the future which is based on the average execution time of a section. The measurement results in [12] were reported as a function of the slack (unused time). NPM performed worse, which is quite obvious. DPM-G had very stable energy consumption which available slack had very little influence. This was because when there was much slack, the first few sections were executed very slowly, which consumed most of the slack. When further sections were started to execute, there was very little slack available which increased processor utilization. Most of the sections were executed with almost no slack, causing the consumed energy to be at the same level. The energy consumption of the other three schemes, SPM, DPM-P and DPM-S, decreased as slack was increased. Energy consumption of the DPM-S scheme was always smaller than other schemes.

The approach in [16] was to divide scheduling into design and run-time phases. Reasons for this were that the scheme better optimizes the embedded software design, it gives the system more run-time flexibility and it reduces run-time computation complexity.

In [16], the atomic unit of the design-time scheduler was called a thread node and the atomic unit of the run-time scheduler was called a thread frame. Thread frame consists of thread nodes. The design-time scheduler, which works on thread node granularity, explores different scheduling combinations and gives several solutions which the run-time scheduler uses. Naturally, the more solutions the design-time scheduler provides, the better results the run-time scheduler can achieve but with increased overhead. A genetic algorithm was used in [16] to find solutions at design-time. Reasons for selecting a genetic algorithm were its speed and near-optimal solution. For larger problems, algorithms that search for optimal solution are not applicable because of long computation time they require.

The design-time scheduler provides a set of solutions from which the run-time scheduler chooses one. The chosen one is the one which optimizes the system's energy consumption when all ready-to-run thread frames are considered.

The two phase approach was tested with randomly generated examples and also with an actual ADSL-modem. At the randomly generated experiments, there was two parallel processors: the other worked three times faster and used three times the voltage than the other and consumed nine times more energy. The results were that the two-processor solution, compared to the one-processor one, consumed up to 72% less energy. In the ADSL-modem experiment, energy savings ranged from 20% to 40%.

7. Procrastination Scheduling

This section discusses procrastination scheduling and is based on the work in [6] and [7].

Procrastination scheduling tries to delay the execution of a task to maximize the duration of idle intervals. The goal is to minimize the energy consumption. Procrastination scheduling also considers leakage current which has become a major concern while technology has scaled [7]. As [7] demonstrated, there exist a certain speed (critical speed) below which static energy consumption starts to dominate. Thus, in some cases, it is more energy efficient to execute with the critical speed and shut down the system than to execute below the critical speed. Naturally, shutting down and waking up the processor isn't free but requires saving and restoring registers, caches, etc., which incurs additional energy consumption. [7] used a threshold value such that if the length of an idle period is less than the threshold value, the processor is not shutted down.

The procrastination algorithm presented in [7] is as follows: first, a maximum procrastination interval, Z_i, is pre-calculated for every task (Z_i defines the maximum time that a task i can be delayed while guaranteeing that all deadlines are met). When the processor is in the shutdown state, a controller keeps track of time and wakes up the processor after a time period, which is determined by the minimum Z_i of tasks that the controller has received. After the processor has woken up, it schedules the highest priority task with the assigned slowdown factor (the slowdown factor scales the execution speed of the task). Tasks are scheduled with EDF policy.

Tests in [7] were performed in a simulator. The following approaches were compared: no DVS (no-DVS) where all tasks are executed at maximum performance; traditional DVS (DVS) where tasks are executed with a minimum possible slowdown factor; critical speed DVS (CS-DVS) where no task gets a slowdown below the critical speed and critical speed DVS with procrastination (CS-DVS-P) which is a CS-DVS with procrastination. All algorithms performed almost identically above the critical speed. However, below the critical speed CS-DVS consumed up to 5% less energy compared to DVS and at utilization of 10% CS-DVS-P consumed 18% less energy compared to CS-DVS. All algorithms clearly outperformed no-DVS.

The work in [6] proposes a dynamic slack (unused run-time) reclamation algorithm with procrastination scheduling. The idea of the slack reclamation algorithm is as follows: an early completion of a task results in a (dynamic) slack which is stored in a free run time list (FRT-list). The list is sorted by the priority of the slacks such that the slack with the highest priority goes to the head of the list (slacks are always taken from the head). A task can reclaim a slack from the FRT-list if the slack has higher or equal priority to the task. When a task arrives in the system, it is assigned a time budget. Each task is allowed to use its run-time and equal or higher priority run-time from the FRT-list. The algorithm can perform both dynamic slowdown and dynamic procrastination.

An important thing to consider is how the available slack is used: it can either be used to slowdown or procrastination. The solution to the slack distribution in [6] is as follows: if the processor is in the shutdown state and the available slack would be consumed by executing a task at critical speed, dynamic procrastination is not performed. Otherwise, the shutdown state is continued. When the processor has woken up, it uses the available slack for dynamic slowdown the critical speed being the lower bound.

Based on the experiments in [6] dynamic slack reclamation with dynamic procrastination doesn't offer significant energy savings compared to dynamic slack reclamation with static procrastination. This is because the energy consumption due to the leakage current is already mostly avoided by the static procrastination. However, when intervals are not long enough for static procrastination for the shutdown to be energy efficient, dynamic procrastination will extend the intervals resulting in significant energy savings [6].

8. Conclusions

Dynamic voltage scaling is a good solution to save significant amounts of energy. However, its efficiency as such diminishes as static (leakage) current gets larger because of smaller and smaller scales in circuit manufacturing. Static current is also the cause of the fact that lower voltages and frequencies will not always lead to energy savings but there exists a critical speed below which static current starts to dominate energy consumption. Operating below the critical speed is not energy efficient anymore and it can actually be more energy efficient to shut down the processor than to continue executing with the critical speed.

Procrastination scheduling tries to address the aforementioned facts. The proposed procrastination scheduling algorithms, in [7, 6] discussed in section 7, are good candidates for energy-efficient scheduling because of their simplicity and small overhead and because of their ability to take leakage current into consideration. They do not yet offer large energy savings compared to traditional DVS algorithms, though. It is predicted that chip's leakage current increases about five times each generation [1] which can make procrastination scheduling an effective solution. One thing to consider in [7, 6] is that both papers used a controller which handled all the interrupts and task arrivals while the processor was in the shutdown state. Neither one gave any details about the controller.

Compiler-assisted approaches are interesting ones because programs can be analyzed off-line when there is a lot more processing time available as there is at run-time. They should also decrease run-time scheduling overhead. A major drawback of some approaches, like the one in [12] discussed in section 6, is the need to modify existing programs. Another drawback, although not as severe as the previous one, is the need to analyze programs. The best solution would naturally be the one which would be able to use existing programs as such, without a need for modifications or analyzing. The approach in [16], discussed in section 6, needs no program modifications but analysis is still required. Because of aforementioned reasons, compiler-assisted approaches are not that well suited as a general solution. However, for some dedicated devices, compiler-assisted approach can be a good choice.

From the point of what additional information is needed in advance about applications, stochastic (SRT) scheduling is a good choice. The algorithm in [17], discussed in section 5, needs no prior knowledge about applications. All required information is acquired at run-time. One possible problem in statistical approaches is the ability to respond to rapid changes in the computation requirements. The approach in [17], for example, used a run-time generated probability distribution to estimate applications' cycle demands. Even though the distribution was relatively stable in spite of varying computation requirements, there is still the time at the first runs of applications that it takes to construct the distribution and to let it stabilize. Before the distribution is stabilized, it is possible that the predicted cycle demands are not that good with respect to energy consumption. Profiling could also be performed off-line but its suitability, for instance, to multimedia applications is questionable. Constructing the probability distribution can also incur large overhead. The overhead can be avoided by updating the distribution less frequently. But, the more time there is between updates, the more time it takes to get a distribution that is stable enough at the beginning. Of course, the distribution could be updated frequently when an application is started to run but that would increase the overhead.

Interval scheduling, discussed in section 4, is a rather simple approach which makes it a tempting alternative. One problem, and probably the biggest one, with interval scheduling is the choice for the length of the interval. Too long intervals will cause reduced system responsiveness to rapid changes but incur less overhead. Too short intervals, instead, will give higher responsiveness but incur more overhead.

The energy efficient versions of EDF and RM, discussed in section 3, save energy compared to situation without energy-awareness and are still able to meet all deadlines. The algorithms proposed in [13] incurred small overhead and saved energy. One problem with EDF and RM is the need of WCETs which can be hard to determine. Nevertheless, the energy-aware versions of EDF and RM are not much more complex than the traditional ones. And, because they don't miss deadlines and are still able to provide significant energy savings, they are good options.

References

[1] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[2] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power CMOS digital design, 1992.

[3] I. M. Duarte D., Vijaykrishnan N. and Y.-F. Tsai. Impact of technology scaling and packaging on dynamic voltage scaling techniques. In *15th Annual IEEE International ASIC/SOC Conference*, September 2002.

[4] D. Grunwald, P. Levis, K. I. Farkas, C. B. M. III, and M. Neufeld. Policies for dynamic clock scheduling. In *OSDI*, pages 73–86, 2000.

[5] Intel. *Intel Pentium M Processor Datasheet*, April 2004.

[6] R. Jejurikar and R. Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 111–116, New York, NY, USA, 2005. ACM Press.

[7] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 275–280, New York, NY, USA, 2004. ACM Press.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[9] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 1st edition, 2000.

[10] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 50–61, New York, NY, USA, 2001. ACM Press.

[11] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 35–44, New York, NY, USA, 2002. ACM Press.

[12] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low-Power (COLP'00)*, October 2000.

[13] P. Pillai and K. 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] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*, pages 13–23, 1994.

[15] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, , and P. Stenstrm. The determination of worst-case execution times—overview of the methods and survey of tools. accepted for ACM Transactions on Embedded Computing Systems (TECS), 2007.

[16] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor SOCs. *IEEE Des. Test*, 18(5):46–58, 2001.

[17] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 149–163, New York, NY, USA, 2003. ACM Press.

Instruction-level Energy Consumption Models

Kristian Söderblom
Software Technology Laboratory/TKK
kps@cs.hut.fi

Abstract

A lot of effort is being spent on finding ways to decrease energy consumption and increase battery life in portable devices like mobile phones. To minimize the energy consumption in a device, designers have to optimize at all levels and in all parts of the device, starting at the system level. Energy optimizations of the source code and in an energy aware compiler can be used to reduce energy consumption in the software part of the device. Energy estimation at the machine instruction level requires an instruction-level energy consumption model. This paper focuses on the well known instruction-level energy model by Tiwari et al. *In addition, the paper briefly describes some other models that attempt in various ways to be more powerful than the Tiwari model. Finally the paper concludes.*

1. Introduction

The recent years have seen a huge increase in the amount of available types of hand-held devices like mobile phones, cameras, PDAs etc. New features and software in the devices, and decreasing device size mean that energy consumption and battery life continue to be very important design factors. To reduce the time to market and expenses there has been a move from application specific logic to programmable devices. Energy consumption in a device depends primarily on the hardware and its design, but also, if the device is programmable, on the software running on the device.

A huge amount of advances and improvements have been made in embedded system hardware, especially mobile phone hardware [16]. Batteries keep improving slowly but steadily. However, these advances are not in themselves enough to minimize energy consumption in devices. One reason for this is the digital convergence, which requires there to be more and more software in the devices, for example multimedia codecs.

To improve battery life in devices and cope with thermal issues we need low power design and power estimation methods. While hardware design is often top-down, starting at the system level, hardware designers use low level tools, at the circuit level and gate level, to simulate their designs and estimate power consumption. Software developers need higher level tools. One reason is that circuit and gate level information is not available in processor manuals. Low level tools are also slow. With high level energy estimates, for example at the instruction level, software developers can get feedback on how energy efficient their code is. If an energy-aware compiler is used, the compiler itself can make energy optimizations and provide feedback to the user about energy consumption. With a traditional compiler, which optimizes only for speed and/or size, the programmer can use an energy estimate to get feedback about source level optimizations.

The structure of the paper is as follows. In chapter 2, we present high-level energy estimation and optimization as complementary to more traditional low-level methods. Chapter 3 presents the well established instruction-level power analysis methodology introduced by Tiwari *et al*. Chapter 4 presents some other models which build on the same ideas. In chapter 5 the paper concludes.

2. High-level energy estimation and optimization

Before talking about different abstraction levels, we want to mention some basic facts. Average energy equals average power multiplied by time: $E = P \times T$. When the supply voltage in a device is V and the current is I, power equals $V \times I$. Energy can be used to perform work, that is to handle a user's work load. Power means how much energy is used per time unit. While energy and power are not the same thing, this difference is often not emphasized in power estimation literature.

To be able to optimize the energy consumption in devices, optimizations have to be done at all possible levels and in all components, both hardware and software. Huge gains can be realized by making good design choices at a high level of abstraction. For example, optimizing software for power at the source code level can in some cases lead to energy savings of 90% [9].

2.1. Top-down design flow and energy estimation

Embedded system design often begins at the highest level, namely the system level. Macii *et al*. [7] present a possible low-power design flow. The design process is top-down, that is, first the designer partitions the design into components or modules, for example analog or digital parts, or hardware and software components of the system. For hardware parts that are developed in-house, the top-down design continues for the modules down to the lowest levels. To estimate the energy consumption in devices, hardware designers can use tools with a low level of abstraction, since all the details of the design are available to them. For example, the SPICE tool is a tool for simulating a design at the circuit level.

Energy estimates are also needed at higher levels, for example designers need to make system level power budgets, determining how much energy the different parts of the system are allowed to use. To deal with the increasing complexity and features in devices, there is a need for design tools working at high level [16].

In general one can say that low level methods require a lot of information and require time consuming simulation, but can be very accurate in estimating energy consumption. At higher levels, less details are known, so the estimation can be faster, but this might not capture all the subtle effects in the design. One should note that often relatively correct power estimates are enough when making design tradeoffs. If one wants to make sure a component doesn't exceed its energy budget, absolutely correct estimates are required.

2.2. High-level abstraction of the design

High-level power estimation [4, 7] methods are not as mature as those at the lower levels of abstraction. However, a lot of research has gone into developing high-level power estimation models and methods. These high levels are, in order of increasing abstraction: the *architecture*, *algorithm* and *system* levels. The architecture level is the lowest level of these: it deals with things like voltages, capacitances and technology parameters of hardware components like memories, registers etc. The algorithm level deals with *behavior*- and *instruction*-level issues. On the behavior level the activity of resources is predicted, either statically or dynamically (using profiling). On the instruction level, hardware details are abstracted behind a well known interface, the machine instruction set. The highest level is the system level, where supported features, hardware, compilers et cetera, are selected. This paper focuses on the *software*, that is, instruction level.

2.3. Instruction level modeling, estimation and optimization

For software people that are used to programming and compilers, the instruction level is an intuitive level to make energy optimizations. Both the programmer and the compiler can have an effect on the energy consumption of the resulting software.

At the instruction level, we are dealing with machine instructions of a certain processor. The hardware details disappear behind a simple interface, the instruction set. To be able to estimate the energy consumed by software, a *model* must be made of the energy consumption. When a program is given as input the model is used to *estimate* the amount of energy used by the hardware as a result of running this particular program. The model can be used to estimate the energy consumed when running a whole program or just a certain sequence of instructions.

The instruction sequence used as input can be obtained statically or dynamically. Statically one can for example estimate energy consumption of individual basic blocks, or if more advanced static analyzes are available, of the whole program. In the dynamic case, one simulates the program for one or many different inputs and can then estimate the power of the resulting instruction sequences.

Application software on embedded devices should be compiled with an energy aware compiler. These compilers need to consider, in addition to *performance* and *code size*, *energy* or *power.* Energy aware compilers need energy estimates of instruction sequences to be able to minimize the energy consumption of compiled programs. The energy aware compiler can use an instruction-level power model to make power-performance tradeoffs. Good estimation speed allows a compiler to try more alternatives. If an energy aware compiler is not available, one can use an energy analyzer or simulator to evaluate the energy efficiency of source level optimizations.

3. Tiwari model

In 1994 Tiwari *et al.* published a methodology [14, 13] for developing and validating an instruction level energy consumption model for any processor. One motivation for this is that instruction set of a programmable device is always known, whereas all hardware details might not. The Tiwari model relies on physical measurements taken from hardware, which can be done by third parties. Tiwari calls the approach *instruction level power analysis* (ILPA). Being able to estimate the energy consumption of machine instructions, one can proceed to estimate the energy consumption of entire programs. The presented methodology makes few assumptions which makes it applicable to a wide variety of processors (e.g. CISC, RISC, and DSP).

3.1. Tiwari's expression for energy consumption[1]

In the Tiwari model, each instruction has a base cost, and in addition to this, there is an inter-instruction cost, the overhead of executing two specific instructions after one another. To this is added energy of other effects like branch misprediction, pipeline effects and caches:

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k$$

E_p = energy consumed by program
B_i = base cost of instruction i
N_i = number of executions of instruction i
$O_{i,j}$ = overhead of executing instruction pair i, j
$N_{i,j}$ = number of executions of instruction pair i, j
E_k = energy of other effects

3.2. Base costs

The base cost of an instruction is the energy necessarily used in the processor to execute the instruction. By definition, it does not include any energy consumption caused by executing multiple instructions, e.g. stalls or cache misses. One should note that even in power optimized processors, base costs for instructions with similar functionality tend to be the same. This leads to the idea of grouping instructions to reduce the amount of needed measurements and speed up the power estimation.

Having a fixed average base cost of an instruction, independent of operand values, is justified because variation due to operand values have been found to be quite small (at most <10%) [13].

Note that memory operands accessing different kinds of memory could lead to huge differences in the energy consumption. A possible solution is to have separate base cost tables for different kinds of memory.

3.3. Inter-instruction (overhead) costs

The inter-instruction cost is a cost which involves different instructions. When two different instructions are executed after one another, changes in the circuit state cause some energy consumption overhead. The inter-instruction effects can also be e.g. stalls or cache misses[13]. The inter-instruction effect is usually in the order of 10% of the base cost or so, depending very much on the processor.

One interesting thing to note about inter-instruction costs is that the more power-optimized the processor is, the more likely it is that inter-instruction effects are relatively big. On processors that have few power optimizations instructions that take an equal amout of clock cycles to execute take approximately the same amount of energy. This is because energy is used in parts of the processor to calculate results that are later discarded.

[1]Remember that $E = P \times T = (V \times I) \times (N \times \tau)$, where N is the number of clock cycles and τ the length of the clock cycle.

3.4. Energy of other effects

The other effects are for example prefetch buffer and write buffer stalls, pipeline stalls, and cache misses.

3.5. Measurements

In the Tiwari model, estimates are based on current measurements. Core voltage and clock frequency are assumed to be constant. Therefore it is enough to measure the current usage. Base costs can be determined by putting the relevant instruction in an infinite loop and measuring the current with an ammeter. Inter-instruction costs can be measured by alternating between two different instructions in the loop; the current will in this case be greater than the average of the base currents for the two relevant instructions.

In theory, for a processor with n instructions we need to measure n base costs and $O(n^2)$ inter-instruction costs. In practice, instructions can be grouped: e.g. arithmetic, bit operation, logical, move, etc. Energy consumption of instructions in a group is almost the same, because the instructions exercise the same parts of the processor. The way to measure the other effects is to write some code where these occur and subtract the base and inter-instruction costs from the total energy consumed.

3.6. Memory

If an instruction accesses memory, the energy consumption depends on the specific memory (e.g. on-chip or off-chip). Each kind of memory might require an own set of measurements. If one considers the cases where the instructions and data can both be either in on-chip or off-chip memory, there is four different combinations. This sort of measurements have been done for example for the ARM7TDMI at Dortmund [12]. For example load instructions that access only off-chip memory are found to be about ten times more energy consuming than those that use only on-chip memory.

3.7. Application

The Tiwari model has been found to work well on both CISC (e.g. i486) [14], RISC (e.g. ARM7), and on DSPs: [5],[1].

Once the model has been established it can be used to estimate the *average* energy consumed by a certain sequence of machine instructions. Note that the instruction level energy model is not as complex as that needed in a cycle-accurate energy simulator, where one needs architecture level information, for example energy consumption in different parts of the processor pipeline.

The Tiwari model can be used to check that the software parts are within the given energy budget. Also, even though the estimates are of average power, more fine grained estimates can be made. That is, energy estimates can be made for parts of programs, not only for the whole program.

The Tiwari model can be used to evaluate compiler optimizations. One possible optimization that comes to mind when familiar with the Tiwari model is rescheduling instructions for low power. It is often possible to change the order of the instructions in a program without changing the end result of the computation. The idea is to reorder the instructions so that the inter-instruction effects are minimized. In [10] a list-scheduling algorithm is presented for this purpose.

Hardware designers can use the model to determine whether some instructions in typical software use unnecessarily much power and should be optimized at the micro-architecture level.

4. Extensions and other models

The Tiwari model has some limitations, most notably: the energy estimate is only an average. It does not tell exactly how much power is needed at every instant, or the worst case energy consumption.

4.1. Dortmund model

In 2001, a parametric energy consumption model [11], made at the University of Dortmund, was published. The model is used in the research group's energy aware compiler, **encc**, and in general in the scratchpad research[15] that is going on there. In this model, parameters and auxiliary functions model bit toggling on busses, memory hierarchy, functional unit activity etc.; parameters are estimated from current measurements using linear regression.

4.2. A parametric model based on regression analysis

A technique to create an energy consumption model for a RISC processor using measurements and a regression model was presented in [6]. Experiments showed an average energy estimation error of 2.5% for random instruction sequences.

4.3. Instantaneous dynamic power consumption model

Instantaneous power consumption can be estimated by modeling the processor as an LTI (linear time invariant) system, where the program instructions are the input signal and the power consumption is the response. [8]

4.4. Worst case energy consumption

A *worst-case energy consumption* (WCEC) model is necessarily more complex than an average energy consumption model, for example it needs to know the worst case execution time (WCET) to estimate the worst case leakage energy [2]. This requires extending the instruction level model with features from the architecture level.

5. Conclusions

Instruction level power analysis can be used for example in energy aware compilers. Software developers can get feedback from the compiler about how energy efficient their code is and use this knowledge to learn about writing more power efficient programs. Instruction level power estimates can also be used as a part of even higher level models. For example, a framework for optimizing at the source code level can employ an instruction level power model to make a database of estimated energy for probable code sequences [3].

References

[1] Miguel Casas-Sanchez, Jose Rizo-Morente, and Chris J. Bleakley. Power Consumption Characterisation of the Texas Instruments TMS320VC5510 DSP. In *15th International Workshop on Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation (PATMOS'05)*, pages 561–570, September 2005.

[2] Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. Estimating the Worst-Case Energy Consumption of Embedded Software. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 81–90, San Jose, California, USA, April 2006.

[3] I. Kadayif, M. Kandemir, G. Chen, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. Compiler-Directed High-Level Energy Estimation and Optimization. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):819–850, November 2005.

[4] Paul Landman. High-Level Power Estimation. In *Proceedings of the 1996 international symposium on Low power electronics and design*, pages 29–35, Monterey, California, United States, August 1996.

[5] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita. Power Analysis and Minimization Techniques for Embedded DSP Software. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(1):123–135, March 1997.

[6] Sheayun Lee, Andreas Ermedahl, and Sang Lyul Min. An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, 2001.

[7] E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1061–1079, November 1998.

[8] Jose Rizo-Morente, Miguel Casas-Sanchez, and C.J. Bleakley. Dynamic Current Modeling at the Instruction Level. In *Proceedings of the 2006 International Symposium on Low Power Electronics and Design (ISLPED'06)*, pages 95–100, Tegernsee, Germany, October 2006.

[9] T. Simunic, L. Benini, and G. De Micheli. Energy-Efficient Design of Battery-Powered Embedded Systems. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'99)*, pages 212–217, San Diego, CA, USA, August 1999.

[10] G. Sinevriotis and T. Stouraitis. A novel list-scheduling algorithm for the low-energy program execution. In *IEEE International Symposium on Circuits and Systems (ISCAS 2002)*, pages IV94–100, Phoenix-Scottsdale, AZ, USA, May 2002.

[11] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Yverdon, Switzerland, September 2001.

[12] Michael Theokharidis. Energiemessung von ARM7TDMI Prozessor-Instruktionen. Diploma thesis, University of Dortmund, Department of Computer Science 12, November 2000.

[13] V. Tiwari, S. Malik, A. Wolfe, and T. C. Lee. Instruction Level Power Analysis and Optimization of Software. *Journal of VLSI Signal Processing*, 13(2):1–18, August 1996.

[14] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, December 1994.

[15] Lars Wehmeyer, Urs Helmig, and Peter Marwedel. Compiler-optimized Usage of Partitioned Memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI'04)*, pages 114–120, Munich, Germany, June 2004.

[16] Yrjö Neuvo. Cellular phones as embedded systems. In *2004 IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, February 2004.

Energy Accounting

Timo Töyry
Software Technology Laboratory/TKK
ttoyry@cc.hut.fi

Abstract

In this paper I will present shortly the two method most commomly used in energy accounting these are model based energy accounting and measurement based energy accounting. I will also do a sort survey to taday's computers power management in form of ACPI.

1 Introduction

Energy consumption is now one of the biggest challenges of today's mobile devices. As traditional solution almost every chip and other devices have many different energy saving modes, but they are not efficiently used in today's operating systems since devices are put to lower energy mode after some timeout period and then put back to higher energy mode when some process needs it.

There is a new different approach to this issue: What if user is given the power to choose how long system should stay running with certain set of programs. Then the user shall prioritize his/hers running programs in relation to each other to reflect their importance to he/she. And then the operating system takes care that users requirements are met if they are realistic. One method to achieve this is so called energy accounting.

The term energy accounting could be defined for example as so: A system which allows operating system to control the amount of energy consumed by some program. This requires that operating system have knowledge about that programs energy usage. The knowledge about program's power consumption can be carried out many different ways. Probably the two most common methods are model based energy accounting, which uses a model of the system to calculate estimates for used energy, and measurement based energy accounting which employs realtime measurements of power usage of the system. In following chapters I will introduce the today's standard in computers power management the ACPI and these two methods in little bit more detail and also prototype examples for both.

2 ACPI

ACPI (Advanced Configuration and Power Interface) specification was originally introduced by Intel, Toshiba and Phoenix in December 1996. Compaq (now Hewlett-Packard) and Microsoft joined to development group later. ACPI specification has been updated couple times after is initial introduction. The current version is 3.0b which was published in October 2006. ACPI specification defines common interfaces for energy management for both software and hardware. ACPI allows the operating system to control energy management of whole system as well individual devices. The core of ACPI is ACPI System Description Tables which expose power saving modes of hardware to operating system. Usually ACPI hardware support means only that hardware provides these tables to above software layer. In these tables are listed for example energy modes supported by the device. ACPI is in wide use nowadays, actually it is the most commonly used energy saving interface in all kind of computers today. But unfortunately ACPI does not provide any information about energy consumption to software (operating system). For this reason ACPI is not usable for energy accounting. [1], [2],[3].

3 Model based energy accounting

Model based energy accounting employs some model of used hardware to estimate its energy consumption. These systems do not usually have any feedback from hardware about real energy consumption, but when the model is developed energy consumption of system in certain states are measured and the model is then fitted to measurement data. There are couple different approaches to build an energy model for system [4], [6].

Event counter based models employs hardware performance counters available in system as input to the model. Performance counters are configured to measure relevant events for energy consumption such as CPU cache misses. This kind of model can be used only for CPUs since other devices do not usually have any performance counters. Only exception is the main memory whose usage can be measured although indirectly by CPU cache misses and memory write backs. The accuracy of model is relative to amount of input data from performance counters. The model itself is usually very simple and does not require much computing which makes its results suitable to be used as input for energy accounting algorithm in running system. The simple model is also the weak point of this approach because simplifications can cause inaccurancies in estimation [4], [6].

State based models represents system as a state machine which changes states between different energy states. States are changed correspondingly to actions of running program. When the model is built its energy states are calibrated to match real energy consumption of the state. It is done by measuring energy consumption of components related to the inspected energy state and adjusting the model when required to match measured values. These models give usually more accurate estimates about systems energy consumption than event counter models. The good point of this approach is that it is an all software solution, but the model can not represent any variation in energy consumption within states so the accuracy of the model is essentially defined by number of states used. In energy accounting cost of using a device is usually charged from process which causes that device to change its state to higher energy state[4], [6].

Third different approach to obtain estimates for the energy consumption of system is statistical modeling. The energy estimates are calculated statistically from measurement data which is collected while test programs are ran on system. The measurement data contains usually the process id and value of program counter of currently running process and measured total energy consumption of system. The sample rate of measurement data is quite low compared to clock rate of the CPU, but if data are collected low enough there will be statistically significant amount of samples for every instruction of inspected program. Estimates obtained by this method are more detailed and accurate than estimates from both event counter based and state bases models [4], [6].

3.1 ECOSystem

ECOSystem (Energy-Centric Operating System) is an example of usage state based model in energy accounting. ECOSystem was developed by Zeng et al [6].

3.1.1 Currentcy model

Currentcy model is the energy model used in ECOSystem. Currentcy is an coined term, which is probably invented by the authors, combining the meanings of current and currency. They state that currentcy is the key feature of their model because it forms a common unit for energy allocation and accounting over different hardware devices and software, in which different processes are competing for limited hardware resources [6].

One unit of currentcy is defined as right to consume certain amount of energy in certain period of time. In the prototype system one currentcy is specified to correspond 0.01 mJ of battery energy [6].

In currentcy model power costs consists from two parts, first part is so called base cost and the second part is cost from managed devices. Base cost is defined to include lowest power states of managed devices and default power state of unmanaged devices. The second part of cost is from use of managed devices (in the prototype CPU, hard drive and WLAN) so that they go to higher energy state. All managed devices may have own charging policies for higher energy states than the lowest energy state. Base costs are not charged from processes currentcy containers, but they are taken in account in total energy consumption when targeting to certain battery life [6].

There are two main aspects in currentcy allocation first and most important is target battery life selected by user. The target battery life specifies the amount of currentcy avalaible in each epoch. Epoch is the time interval in which the allocated currentcy should be consumed. Second thing is the currentcy allocation to competing processes in epochs this is done by user given priorities. If process consumes all its currentcy it will be halted until next epoch and new currentcy allocation even if it is otherwise ready to run. Processes can also accumulate some unused currentcy for next epochs to pay some more expensive task. However currentcy accumulation is quite strictly limited to avoid situations where many processes have lots of currentcy to spend in single epoch. If these wealthy processes use all their currentcy one there will be heavy peak in battery discharge rate which is an unwanted effect that can reduce battery life [6], [5].

The processes are required to pay for usage of managed devices when they are going to use them, for example a process pays for execution time on CPU before it is executed. The process can be executed as long as it have currentcy left to pay for execution [6].

3.1.2 The Prototype

The prototype is a modified version of RedHad Linux 2.4.0-test9. They added custom resource containers, that supports currentcy allocation and accounting and so energy accounting, to the kernel. In the prototype they had currentcy management implemented for three different devices those were CPU, hard drive and Wireless network card [6].

They used an IBM Thinkpad T20 laptop as hardware platform for their system. The currentcy energy model was tuned to match real measured energy consumption values of the platform. The CPU on that laptop was 650 MHz Intel Pentium III processor. The used power model for CPU was very simple and coarse since it assumed constant energy consumption in active CPU. More accurate results could be obtained by using an event counter model to track internal energy states of the CPU. Hard drive was a IBM Travelstar 12GN and was connected to system trough standard ATA interface. The Travelstar had all standard ATA power states and some additional internal power states managed by an undocumented power management algorithm which might reduce overall energy savings from disk management since the operating system is not aware of disk real energy state. Wireless network card was an Orinoco Silver PC card, which had three different power states; sleep, receive and transmit. Sleep state is the lowest energy state and transmit state is the highest. The card also have two modes for communication with base station, the states are active mode, in which card have continuous connection to base station and all incoming data is transferred immediately. The other mode is periodical polling mode in which card is most of time in sleep mode and periodically wakes up and requests new incoming data from base station. The base station buffers the traffic directed to the card while it is in sleep [6].

4 Measurement based energy accounting

Measurement based energy accounting requires some special hardware support, which takes care of the measurements. Unlike in model based energy accounting systems measurement based energy accounting systems have realtime (or at least almost realtime) feedback for consumed energy. Since the hardware provides information about energy consumption and if operating system make use of this information it efficiently voids the need for energy model of system. Measurement based energy accounting could provide better performance and lower energy consumption in energy accounting system because it have less computational overhead compared to model based energy accounting [4].

4.1 PLEB2

PLEB2 is a prototype system built by Snowdon et al. The prototype is a single board computer which is based on Intel XScale PXA255 processor. The PXA255 have ARMv5TE compatible core which operates at 400 MHz clock speed, and some peripheral devices like memory, DMA, interrupt and LCD display controllers. The ARMv5TE core contains the CPU core, some onchip SRAM memory and some flash memory [4].

The system have three switching power supplies which provide power from lithium-ion battery independently to CPU core, memory and IO. Each powersupply line have a current sensor for energy measurement [4].

The system have some on-board peripherals which are infra-red, USB and serial ports. They get their power from IO-power. There are also an 8-bit Atmel AVR microcontroller on-board as supervisor for the rest of the system. The system have some extension possibilities for additional peripheral devices since the unused connectors of the XScale are wired to some extension sockets on-board [4].

The PLEB2 platform with its ARM core and peripherals models a typical embedded system. They had modified couple Linux systems (Linux 2.4.19, Linux 2.6.8 and L4ka::Pistachio) to be run able on the PLEB2 system [4].

The power consumed in each area (CPU, memory and IO) is calculated from measured current output of each power supply and known well regulated output voltage, which is considered to be constant. The on-board AVR microcontroller, which have a build in analog to digital converter, is responsible for periodically gathering the measurement data from current sensors. The sampling rate of measurements are limited by the speed of AD-converter, which is able to read sensor in rate up to 15 kHz, but since there are three sensors maximum sampling rate per sensor is then 5 kHz when all sensors are sampled in equal intervals. When AD-conversion is ready the sample is transferred to XScale, which stores it with corresponding process id and program counter for later evaluation or uses it in realtime energy accounting. The transferring process of measurement results from AVR to XScale is very slow, because data is transferred trough slow I2C serial bus, and causes total of five interrupts to XScale. The first interrupt is mark for XScale that microcontroller starts reading a sensor so the XScale can save correct process id and program counter value for the measurement data. Second interrupt is mark that microcontroller are ready with AD-conversion and XScale should start transfer data from microcontroller. The rest three interrupts are caused by data transfer trough the I2C bus [4].

5 Conclusions

Next some comparison between model and measure based energy accounting systems. Firstly both systems have their strong points. For model based system it is systems all software nature, there are no need for special hardware to take care of energy consumption measurements. This is probably one of the weak points of measurement based systems. The strongest point for measurement bases system is the measured realtime information about real energy consumption. And because the energy consumption is measured there is no need for system model, which may require significant amount of computing. In contrast this is one of draw backs of model based systems since they can not be sure about the energy consumption. Another strong area of measurement based system compared to model based system, which usually uses some kind of state model of system, is that measurement based system can capture variations in energy usage within the states of the model and is there for more accurate [6], [4].

Model based energy accounting is currently more interesting technology because it can be implemented to existing system without any need for additional hardware. It only requires some measures about systems energy consumption in calibration phase of the model. Model based energy accounting can be successfully used to lengthen battery life of laptop computer. Zeng et al. proved this with their ECOSystem prototype. They were able to achieve battery life up to 25 hours which tough was possible with expense of interactivity (the response times for example for net surfing got very long) [6].

Measurement based energy accounting is more like future technology mainly due lack of hardware support in current systems. However Snowdon et al. demonstrated with their PLEB2 platform that is possible to build measurement based energy accounting system with realtime measurements from multiple sensors [4].

References

[1] S. Balakrishnan and J. Ramanan. Power-aware operating systems using acpi, cs 736 project. Technical report, University of Wisconsin, Computer Science Department, 2001.

[2] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. *Advanced Configuration and Power Interface Specification 3.0b*, October 2006.

[3] Intel. Acpi overview, 2000.

[4] D. C. Snowdon, S. M. Petters, and G. Heiser. Power measurement as the basis for power management. In *Proceedings of the 2005 Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2005.

[5] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Currentcy: Unifying policies for resource management. In *Proceedings of the USENIX 2003 Annual Technical Conference*, June 2003.

[6] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *ASPLOS02*, October 2002.