Sébastien Lafond

Simulation of Embedded Systems for Energy Consumption Estimation

Turku Centre *for* Computer Science

TUCS Dissertations

No 113, January 2009

Simulation of Embedded Systems for Energy Consumption Estimation

Sébastien Lafond

To be presented, with the permission of the Faculty of Technology of Åbo Akademi University, for public criticism in the Auditorium Gamma of the Department of Information Technologies, on February 27 th, 2009, at 12 noon.

Åbo Akademi University
Department of Information Technologies
Joukahainengatan 3-5
FIN-20520 Turku, Finland

2009

Supervisor

Professor Johan Lilius
Department of Information Technologies
Åbo Akademi University
Joukahainengatan 3-5
FIN-20520 Turku
Finland

Reviewers

Professor Jarmo Takala
Department of Computer Systems
Tampere University of Technology
Korkeakoulunkatu 1
FIN-33101 Tampere
Finland

Dr. Juha-Pekka Soininen
Computing and computer architectures
VTT Electronics
Kaitoväylä 1
FIN-90571 Oulu
Finland

Opponent

Professor Jarmo Takala
Department of Computer Systems
Tampere University of Technology
Korkeakoulunkatu 1
FIN-33101 Tampere
Finland

ISBN 978-952-12-2240-5
ISSN 1239-1883

Abstract

Technology developments in semiconductor fabrication along with a rapid expansion of the market for portable devices, such as PDAs and mobile phones, make the energy consumption of embedded systems a major problem. Indeed the need to provide an increasing number of computational intensive applications and at the same time to maximize the battery life of portable devices can be seen as incompatible trends.

System simulation is a flexible and convenient method for analyzing and exploring the performance of a system or sub-system. At the same time, the increasing use of computational intensive applications strengthens the need to maximize the battery life of portable devices. As a consequence, the simulation of embedded systems for energy consumption estimation is becoming essential in order to study and explore the influence of system design choices on the system energy consumption.

The original publications presented in the second part of this thesis propose several frameworks for evaluating the effects of particular system and software architectures on the system energy consumption. From a software point of view Java and C based applications are studied, and from a hardware perspective systems using general purpose processor and heterogeneous platforms with dedicated hardware accelerators are analyzed. Papers 1 and 2 present a framework for estimating the energy consumption of an embedded Java Virtual Machine and show how an accurate energy consumption model of Java opcodes can be obtained. Paper 3 evaluates the cost-effectiveness of Forward Error Correction algorithms in terms of energy consumption and demonstrates that a substantial energy saving is achievable in a DVB-H receiver when a FEC algorithm is used for file downloading scenarios. Paper 4 and 5 present the simulation of heterogeneous platforms and point out the drawback of different mechanisms used to synchronize a hardware accelerator used as a peripheral device. Paper 6 shows that the use of a multi-bank memory architecture can lead to a 20% static energy saving without any software optimization.

Acknowledgements

All my gratitude goes to all who made this thesis come true. First of all, I would like to thank my supervisor, Professor Johan Lilius for his patience in addition to the invaluable support and trust he offered me during my doctoral studies. I wish to thank him for all the guidances he gave me for sailing in the world of academic research. I also wish to express my sincere gratitude to my co-authors, Jani Boutellier, Dr. Jerker Björkqvist, Kristian Nybom and Professor Olli Silvén. I am glad I had the opportunity to work with them.

I wish to thank Dr. Juha-Pekka Soininen from VTT and Professor Jarmo Takala from Tampere University of Technology for reviewing my thesis. They gave me constructive comments and useful suggestions for improving this thesis. I also wish to thank Professor Jarmo Takala to accept to act as an opponent during the public disputation of my thesis.

I am grateful to Professor Ralph-Johan Back, former Director of the Turku Centre for Computer Science (TUCS), for accepting my application to the TUCS graduate school. I am thankful for the remarkable working environment provided by the Turku Centre for Computer Science and the Department of Information Technologies at Åbo Akademi University. I gratefully acknowledge the financial support I received from the Turku Centre for Computer Science, the Center for Reliable Software Technology (CREST) and the Department of Information Technologies. I also would like to express my gratitude to the Nokia Foundation, the Emil Aaltonen Foundation and the Oscar Öflund Foundation for their generous grants.

I also greatly appreciated during my doctoral studies the enjoyable and pleasant atmosphere at the Embedded Systems Laboratory. I would like to thank the present and past members of the laboratory for all friendly and lively coffee meetings we had. I specially wish to thank Andreas, Dag, Dragos, Jerker, Johan E. and Kristian for all work-related and work-unrelated discussions we had along these years. The Embedded Systems Laboratory is not only made up of expertise (and energy), but contains also an important part of conviviality.

In addition I am grateful to my friends who made my stay in Finland a source

of happiness during the past years. I especially wish to thank the local French and *francophile* community for all the great moments we have shared. Particular thanks to Anna, Antoine, Christophe, Elina, Ellinor, Emmanuel, Eric, François, Henri, Julien, Katarzyna, Kathleen, Lionel, Marika, Nicolas, Pia, Séverine, Sirpa, Stefan, Stefanie, and Tuija for their friendship.

Finally, I also wish to express my warm thanks to my parents and my brothers Mathieu and Olivier for the constant support and encouragements I received from them over the years. Most of all, infinite thanks to my wife Beáta for her daily devotion and unlimited love.

College Park, November the 27th, 2008

Sébastien Lafond

List of original publications

1. Sébastien Lafond and Johan Lilius. An Energy Consumption Model for an Embedded Java Virtual Machine. In proceedings of the *19th International Conference on Architecture of Computing Systems - ARCS 2006*, Lecture Notes in Computer Science, Vol. 3894, pages 311-325. Springer-Verlag, 2006.

2. Sébastien Lafond and Johan Lilius. Energy consumption analysis for two embedded Java virtual machines. In *Journal of Systems Architecture*, volume 53, Issues 5-6, pages 328-337. Elsevier B.V., 2007.

3. Sébastien Lafond, Kristian Nybom, Jerker Björkqvist and Johan Lilius. Receiver Coding Gain in DVB-H Terminals using Application Layer FEC Codes. In proceedings of the *Third International Conference on Digital Telecommunications - ICDT 2008*, pages 110-116. IEEE, 2008

4. Sébastien Lafond, Jani Boutellier, Johan Lilius and Olli Silvén. Energy efficiency analysis of multi-stream MPEG-4 decoder systems. In *Multimedia on Mobile Devices 2008, proceedings of SPIE Vol. 6821, 68210G-1.* SPIE, 2008

5. Sébastien Lafond and Johan Lilius. Interrupt Costs in Embedded System with Short Latency Hardware Accelerators. In proceedings of the *International Conference on Engineering of Computer-Based Systems - ECBS 2008*, pages 317-325. IEEE, 2008

6. Sébastien Lafond and Johan Lilius. Static Energy Saving Through Multi-Bank Memory Architecture. In proceedings of the *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation - ICSAMOS 2006*, pages 43-49. IEEE, 2006.

Contents

Part I

Research Summary

1 Introduction

Technology developments in semiconductor fabrication [1], along with a rapid expansion of the market for portable devices, such as PDAs and mobile phones, make the energy consumption of embedded systems a major problem [2]. Indeed the need to provide an increasing number of computational intensive applications and at the same time to maximize the battery life of portable devices can be seen as incompatible trends. Although energy is solely consumed by the hardware, energy consumption can be trimmed by adequate software manipulations because the software operates the hardware activities. Therefore, for a defined hardware platform different software implementations providing the same services can have different performance in terms of energy consumption. Evaluating these performance differences is essential for determining which implementation is the optimum solution. System simulation can be an answer to the problem of performance evaluation when assessing different software implementations on several hardware platforms, as it provides a flexible and convenient solution for performance evaluation.

Energy consumption

We can often see confusion concerning the use of the terms energy and power. The electrical energy E of a system having one source of energy is expressed in joule and is defined by:

$$E = q.V$$

where q is the quantity of electric charges expressed in coulomb that has passed through the cross-section of the electrical conductors connecting the source of energy, and V is the electric potential expressed in volt between the two poles of the source of energy. Whereas, the power P is expressed in watt and is defined as the average rate at which electrical energy is consumed by an electric circuit during a time t:

$$P = \frac{E}{t}$$

Furthermore, the overall power dissipation P can be expressed as:

$$P = P_{stat} + P_{dyn}$$

where P_{stat} represents the static power dissipation and P_{dyn} represents the dynamic power dissipation. P_{stat} is caused by the leakage current I_{leak} flowing through turned-off transistors from the supply rail to ground and is expressed for one transistor as:

$$P_{stat} = I_{leak}.V_{DS}$$

where V_{DS} is the electric potential between the drain and the source of a turned-off transistor. P_{dyn} is due to the switching activity of the gates constituting the digital circuit. The dynamic power consumption per transistor is expressed as:

$$P_{dyn} = ar.f_c.C_l.V_{DD}^2$$

Most of the confusion comes from the use of the *low power* adjective to mean *having a low energy consumption* or *having a low average power dissipation*. For most of the system the important matter is certainly the energy consumed by the system, which is driven by its average power dissipation over its life time. Indeed, it is the energy consumed by a system that will affect its operating cost as well as its operating lifetime in the case where the system operates on a battery.

When considering a complete system, there are different levels [3, 4] where issues concerning energy consumption can be addressed:

- Low power electronics
- System architecture
- Code generation
- Software architecture

On each level, there are various ways [5, 3, 4, 6, 7, 8] of evaluating the energy consumption drained by a system and estimating the influence of a specific level. The original publications presented in the second part of this thesis focus, in papers 1, 2 and 3, on the simulation of different software architectures and, in papers 4, 5 and 6, on system architectures for embedded system.

Furthermore, as the circuits shrink, leakage currents and thus static power dissipation become increasingly significant [9]. To limit the energy due to the static power dissipated by RAM memories, paper 6 proposes, models and simulates a multi-bank memory architecture. All the other papers presented in the second part of this thesis propose several simulation platforms for estimating the dynamic energy consumed by the studied system.

Embedded systems and energy consumption

There are several alternatives for defining an embedded system. In [10] an embedded system definition is given based on its development environment as: *any computer system for which the primary development tools do not run on the system itself*. But in [11] an embedded system is defined based on its physical environment as: *information processing systems embedded into enclosing products such as cars, telecommunication or fabrication equipment*.

The union of these two definitions designates a complete system with dedicated software applications running on a particular hardware platform. Hence it encompasses a very large spectrum of systems ranging from small and simple systems

such as a portable watch, to large and complex stationary industrial installations such as the control system of an electrostatic accelerator. The scope of this thesis targets a restricted set of embedded systems that can be defined as *stand-alone computer based systems operating on a battery for which the primary development tools* ***can not*** *run on the system itself.*

As the definition of an embedded system is rather large we can categorize two different types of embedded systems when we look at problems resulting from their energy consumption characteristic: a) portable devices having a strong constraint on their available energy budget, thus limiting the system average power dissipation for a defined operating time and b) stationary systems having almost no limitation concerning available energy but where the level of the heat generated by the system requires cooling solutions such as heatsink, fan or liquid cooling unit.

A portable device operates on a limited energy budget defined by the capacity of its battery. As the physical dimension of a portable device is an important design characteristic, the constraint on the physical size and weight of the battery strongly limit the available energy capacity of the battery. Therefore, for such a system the average power dissipated by the device will determine the system operating time before the battery needs to be changed or recharged. Moreover, in most of portable systems the instantaneous power dissipation is not an essential characteristic as the heat conducted and/or radiated to the device surroundings stays below acceptable limits. Also several properties of a system can be affected by a limited energy budget. Energy management through voltage and frequency scaling can affect the system reliability [12, 13], whereas the battery capacity may not cope with the energy requirement of a cryptographic algorithm [14] and as a result prevents the implementation of the required security and privacy level.

On the other hand, most of stationary systems do not rely on a battery and thus do not have strong constraints concerning their energy budget. However, for such systems the maximum instantaneous dissipated power will define the required cooling solutions in order to operate within safe temperatures.

The last two decades have seen tremendous growth in the demand for such systems with the emergence of communicators, mobile phones, PDAs, palmtops, and so on. At the same time, while the increase in performance of embedded system's hardware caused a radical increase in power densities [13], an increasing number of computational intensive applications (e.g. multimedia steaming applications) are expected to run on such systems [15]. While introducing more complex applications and application execution platforms, these parallel trends make energy-aware system design, and thus system simulation for energy consumption estimation, a fundamental issue.

2 System simulation for energy consumption estimation

System simulation is a widely used method for gathering information about system performances without the need to have the system physically available. The need of system simulation arose concurrently with the fast development of computer systems in the seventieth [16, 17]. Compared to the use of a prototype board, system simulation provides a flexible and convenient solution for performance evaluation during the system design process. Historically system simulation was principally developed and used for simulating the execution time and the memory requirement of a system [18, 19, 20, 21]. However, technology developments in semiconductor fabrication [1] along with a rapid expansion of the market for portable devices make the energy consumption a major problem for embedded systems to operate on a battery [2]. Most of the simulation frameworks for energy consumption estimation can be categorized either as, an instruction-level energy simulator or, an architecture level power simulator.

2.1 Instruction-level energy simulator

The possibility of simulating the energy consumption of a processor based on an instruction-level power model is first introduced in [5] by Tiwari at al. In [5], they develop a power analysis technique for two microprocessors, the Intel 486DX2 and Fujitsu SPARClite 934, and measure the average current drained by the processor for each instruction from the processor instruction set. These measured values give the base energy cost for each instruction. In addition to the base energy cost, inter-instruction effects are also analyzed. Inter-instruction effects are due to cache misses, resource constraints and circuit state. A cache miss introduces extra latency when the data is fetched from memory because the data needs to be retrieved from a lower level memory. In the same way, a resource constraint can cause pipeline and buffer stalls which lead to an increase in the number of cycles needed to execute a sequence of instructions. These two inter-instruction effects increase the system energy consumption by introducing extra time in the software execution. On the other hand, based on the previous state and the instruction operand values, the effect of the circuit state will modify the base energy cost for each instruction execution without introducing any extra cycle penalty.

Several instruction-level energy simulators have been presented since [5] in [6, 22, 23, 24, 25, 26]. These simulators are all based on the same approach presented in [5] and illustrated in figure 1: an instruction-level power model is obtained by physical measurements of the current drained by the processor, and external memory if a memory model is also given, and the energy consumption of an application is estimated based on a simulated execution trace containing the list of executed instructions.

To the author's knowledge the possibility of establishing an opcode-level energy model for a Java Virtual Machine was not previously studied. Following a similar

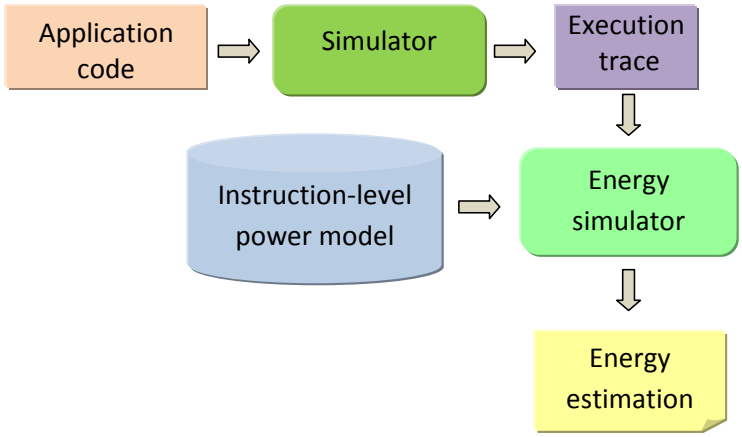


Figure 1: General approach used by instruction-level energy simulators

approach to that used in [5] to determine an instruction-level energy simulator for a processor, paper 1 and 2 describe the development of a framework used to establish an opcode-level energy model for embedded Java Virtual Machines.

Paper 1 proposes a general framework for estimating the energy consumption of an embedded Java Virtual Machine (JVM). This paper presents a number of experiments to estimate the constant overhead of the JVM energy consumption and establishes an energy consumption cost for individual Java Opcodes. The first author is responsible alone for all the results and contributions of this publication.

Paper 2 extents the results presented in paper 1. In paper 2, a comparison between the Sun Microsystems K Virtual Machine (KVM) and the simple Real-Time-Java (simpleRTJ) virtual machine is done. This publication shows that implementation differences between two embedded Java Virtual Machines can lead to great divergence regarding the JVM energy consumption. The first author is responsible alone for all the results and contributions of this publication.

Due to space requirement the result tables of papers 1 and 2 were initially only published on the internet, but they are, in this thesis, included as an appendix at the end of each paper.

Instead of using the physical measurements of the current drained by the applications we used the commercial ARMulator [27] as the processor simulator and the enprofiler tool [28, 29] as the energy simulator. This choice was motivated by the advantage given by the use of a simulated environment and also the fact that the enprofiler tool provides satisfactory precision [29] when estimating the energy consumed by the execution of an application. As for establishing the base energy cost for processor instructions in [5], the cost of each Java opcode is determined by looking at the cost differences between the execution of two particular Java classes. As the generation by hand of a Java class is an extremely tedious task, a Java class file generator producing particular pairs of Java classes was developed to extract

the energy consumption of a singular Java opcode.

Nevertheless, the inter-instruction effect due to the circuit state between two Java opcodes is negligible compared to the one at the processor instruction level. This can be explained by the corresponding code length in processor instructions of one Java opcode, ranging from 45 to few thousands processor instructions. As the inter-instruction effect, due to the circuit state, appears only between the last and first processor instructions of two consecutive Java opcodes, its value becomes insignificant in comparison to the cost of execution of the block of processor instructions corresponding to one Java opcode.

The opcode-level energy models presented in paper 1 and 2 are validated based on the execution of benchmarks [30]. For each benchmark execution the total energy costs given by the established models are compared to the energy cost given by the enprofiler energy simulator. Based on these observations the opcode-level energy models give results with an error range of -5% +10% compared to the enprofiler result. This loss in precision has to be balanced with the considerable gain in execution speed between both simulations. It takes only few seconds to compute the total energy cost of an application based on the proposed opcode-level energy models, compared to several hours needed by the enprofiler simulator.

2.2 Architecture level power simulator

A different approach to the simulation of processor energy consumption is to develop an architecture based energy model [31, 32, 33, 34, 35, 36]. With this approach each functional unit, also called a micro-architectural block, present in the processor is modeled, and the total energy consumption is estimated at every clock cycle by summing up the energy consumed by each functional unit.

Conventional Architecture

In paper 3 the simulation of a conventional architecture, containing a general purpose processor and a standard memory architecture composed by a main memory and two cache levels, is needed in order to simulate forward error correction (FEC) algorithms.

Because Sim-panalyzer [37] is a widely used [38, 39, 40, 41] and open-source cycle-accurate architecture level energy simulator, Sim-panalyzer was chosen for simulating the FEC algorithm. Sim-panalyzer was developed on top of the sim-outorder simulator, a component within the SimpleScalar-ARM simulator [42], and simulates a strongARM SA1100 processor. It computes the energy dissipation of each micro-architectural block by multiplying the switching capacitance by the number of micro-architectural accesses [43]. Sim-panalyzer simulates a configurable level one and two, data and instruction, cache memories. For each cache the associativity, the number of blocks and the block size can be configured. Moreover,

the data and instruction caches from the same level can be unified. Sim-panalyzer provides accurate results with a 9% error margin in its datapath and execution unit models and a 7% error margin in its memory power model compared to a gate level simulation [43].

FEC [44] is a technique used in telecommunication for detecting and recovering errors in the transmitted information without the need to request the sender for additional data. On the sender side, FEC algorithms add redundant data to the transmitted information in a way that retransmission of the information can often be avoided. FEC algorithms have been widely studied across all telecommunication domains [45, 46, 47, 48]. All these studies analyze the performance of FEC algorithms by analyzing the algorithm complexity, the size of the redundant data and the remaining loss probability depending on the erasure rate in the transmission channel. However, to the author's knowledge, no analysis was done in order to evaluate the cost-effectiveness of FEC algorithm in terms of energy consumption in the receiver.

Paper 3 defines the receiver coding gain, the reduction of energy while using coding compared to the energy needed without coding, for a DVB-H receiver in file downloading scenarios. Moreover, an evaluation of the cost-effectiveness of FEC algorithms in terms of energy consumption in the receiver is presented.

This publication presents a comparative study between two AL-FEC codes, the HLDPC and the Raptor code, as well as a theoretical analysis of the receiver coding gain. This paper shows that depending on the AL-FEC code and the erasure rate the receiver coding gain varies from -9 to 4 dB. It demonstrates that for one of the studied AL-FEC code, and the present technology for DVB demodulator, the use of application layer coding leads to a positive receiver coding gain when the erasure rate is above 2%. The first author contributed to a significant part of the results and contributions of this publication.

Simulation of heterogeneous platforms

Different solutions have been proposed for simulating heterogeneous platforms. In [49] Ptolemy, a well-known environment for simulation and prototyping of heterogeneous systems, is presented. Ptolemy focuses on particular type of system domains including synchronous and dynamic dataflow, and discrete-event systems.

In paper 4 and 5 the simulation of an application written in C and running on a heterogeneous platform is required. The objective of paper 4 and 5 focus only on the impact of the hardware platform on the system energy consumption. The software implementation of the applications existed "as it is", but the use of Ptolemy would require a Ptolemy specific re-implementation of them. Therefore, an extension of the general purpose processor simulator Sim-panalyzer was implemented instead of re-implementing the existent applications for Ptolemy.

Paper 4 presents an analysis of three systems simultaneously decoding multi-

ple videos on a dedicated heterogeneous platform. The compared systems consist of one fully software and two hardware accelerated solutions. One of the hardware accelerated solutions is using flag polling for synchronizing the processing units, while the other one is using a scheduling-based synchronization approach. This publication analyzes the differences between the three video decoding systems in terms of execution speed, energy consumption and cache behaviors. The first author contributed to a significant part of the results and contributions of this publication.

In paper 5 a methodology for analyzing the impact of short latency hardware accelerators on a typical embedded system is proposed. This publication analyzes the effect of the hardware accelerator granularity on the system performance with respect to the number of cache misses, the execution time and the system energy consumption. This paper demonstrates the relative important costs introduced by the mechanism for suspending a task in a real-time operating system when the hardware accelerator granularity is decreasing. The first author is responsible alone for all the results and contributions of this publication.

Memory architecture

The simulation of memories includes the simulation of CPU caches and main memories, and is relevant for most of the systems if one wants to obtain comprehensive data about the system performances. For instance papers 1 and 2 show that memory accesses consume 70% of the total energy consumption of the studied system. This reflects the importance of memory performances for stack-based machines where all operand variables and operation results are stored on the stack. Also paper 5 demonstrates the potential negative effect of caches when short latency hardware accelerators are used. This is due to an increase in the number of cache misses imputable to the relative short interval between two context switches.

Many studies concerning CPU cache configurations and simulations [50, 51, 52, 53] were done since the 1980s when the performance gap between processors and memories started to be significant [54]. Also the exploration and the simulation of different memory architectures have been extensively studied [55, 56, 57, 20, 58, 59]. As well, studies concerning multi-bank memory architectures have been presented [60, 61]. However, none of these studies propose a model for the static energy consumed by a multi-bank memory architecture.

Paper 6 establishes the static energy model for a multi-bank memory architecture and introduces the equations governing the optimization problem for decreasing its static energy consumption. The impact of different parameters on the energy consumption and a performance analysis of such memory architecture in terms of static energy consumption and execution speed is also presented. This paper shows that the use of a multi-bank memory architecture can lead to a 20% static energy saving without any software optimization, nor bank need prediction, nor dedicated allocation policy. It also highlights the predominant role of the bank occupation

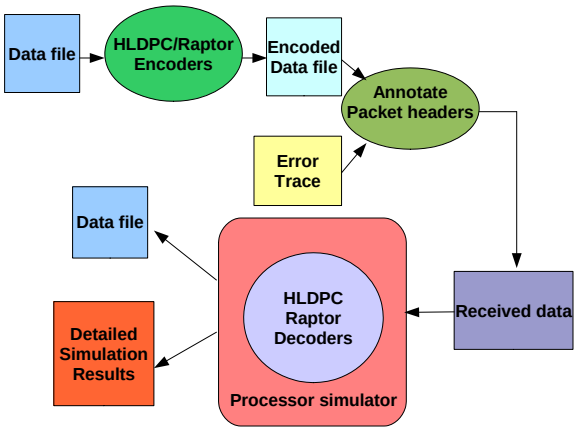


Figure 2: General view illustrating the measurement system used in paper 3

rate in the multi-bank memory architecture static energy consumption. The first author is responsible alone for all the results and contributions of this publication.

3 Sim-panalyzer extensions

Sim-panalyzer is the processor simulator used in papers 3, 4 and 5. However, in order to simulate the desired hardware platforms and/or software components several extensions were implemented into sim-panalyzer. As these extensions are not discussed in papers 3, 4 and 5 due to space requirement, the following section presents them in more detail.

3.1 File handling mechanism

In paper 3, a possible receiver coding gain in terms of energy consumption is studied when error correction at the application layer (AL-FEC) is used in file downloading scenarios over DVB-H. In order to be able to compare the energy consumption of a receiver using AL-FEC with a receiver having no error correction at the application layer, the execution cost of the AL-FEC codes needs to be evaluated. In a physical system the encoded data is received in a streaming manner from the radio receiver via a buffer. However, because AL-FEC codes for DVB-H receiver are for the moment only at a development stage and not used in real devices, the simulated algorithms were implemented with file handling mechanism for fetching the received data. Figure 2 presents the measurement system used in paper 3 where data files are represented by a rectangle box.

The received data file size being relatively important, fetching the encoded data from a file would distort the results given by the simulator. Therefore, in order to

minimize the disturbance from the file handling mechanism in the simulation of the AL-FEC codes, the Sim-panalyzer and AL-FEC codes were modified. The process of fetching the received data from a file has been moved to the simulator and only the size of the received data and a memory address are exchanged between the simulator and the AL-FEC codes via dedicated system call. Figure 3 shows the sequence diagram describing the new required operations for execution the file handling process within the simulator. Basically, in the AL-FEC codes the following code needs to be used instead of the open file command:

```
asm volatile ("swi 0x200\n\t"
              "mov %0,r0\n\t" /* output */
              :"=r"(fsize)
              );
encodedData = (unsigned char *) calloc(fsize,sizeof(unsigned char));
register int r0 asm ("r0") = (int)encodedData;
asm volatile ("swi 0x201\n\t"
              : /* no output */
              :"r"(r0) /* input */
```

In this code the dedicated system calls are implemented with software interrupt and parameter values are passed via registers. The software interrupt number 0x200 is used to implement the system call requesting the data size and the software interrupt number 0x201 is used to implement the system call providing the start address of the memory space allocated for the received data. On the Sim-panalyzer side, a switch statement implements the required functionalities for the software interrupt numbers 0x200 and 0x201. With such implementation the disturbance from the file handling mechanism in the simulation of the AL-FEC codes is minimized.

3.2 Hardware accelerators as peripheral devices

In papers 4 and 5 the simulated hardware platforms consist of a general purpose processor and a set of dedicated hardware accelerators used as peripheral devices. Figures 4 and 5 present the simulated platforms in respectively paper 4 and 5. In order to simulate the use of a hardware accelerator, a new system call and new interrupt service routine, in the case the accelerator must be synchronized by interrupt, has to be implemented.

Each hardware accelerator is triggered via a system call (implemented with a software interrupt) and if needed, signals the completion of its job by setting up the corresponding hardware interrupt.

Data transfer from the general purpose processor to an accelerator can be done via the processor registers if only few atomic variable values are passed. A predefined shared memory space is used for transferring several atomic variables or full

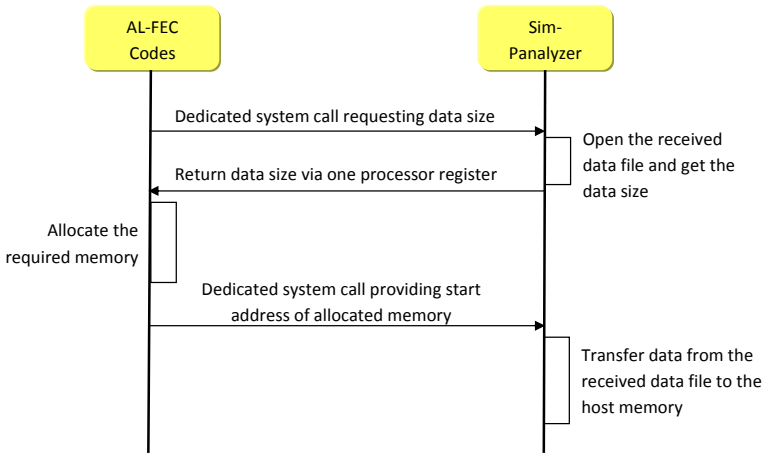


Figure 3: Sequence diagram - Dedicated communication between AL-FEC codes and Sim-Panalyzer

data structures from the general purpose processor to an accelerator. For transferring the accelerator result(s) to the general purpose processor a second predefined shared memory space is used.

Sim-panalyzer was modified in order to support the use of these predefined shared memory spaces. The implementation of dedicated system calls used to trigger the hardware accelerator executions is the same as the one presented in the previous *File handling mechanism* subsection. The use of new interrupt service routines requires few modifications in the Sim-panalyzer simulator and the RTEMS operating system. Sim-panalyzer was modified in order to assign, for each accelerator generating an interrupt, a distinctive interrupt mask to the current program status register (CPSR) when the processor is to be put in interrupt mode. In RTEMS, the interrupt handler was modified to identify, based on the interrupt mask, the interrupt source and to call the appropriate interrupt service routine (ISR).

Because the general purpose processor and the hardware accelerators are sharing the same memory space for data transfer, a cache coherency problem may arise. Figure 6 illustrates the possible problem of cache coherency between the data stored in level one and two caches and the corresponding data stored in the shared memory space. Without any mechanism for avoiding cache coherency problem, the integrity of a data is not insured for a data moved by the GPP to the shared memory space. Indeed, Sim-Panalyzer implements only the write-back mechanism for cache operations. Thus, a data value is updated in a shared memory space only when the corresponding memory location is not present or evicted from the cache level one or two. The same is true for a data moved by the accelerator to the shared memory. If the corresponding memory location is present in cache level one or two when the GPP accesses the data, the outdated data value from the cache will

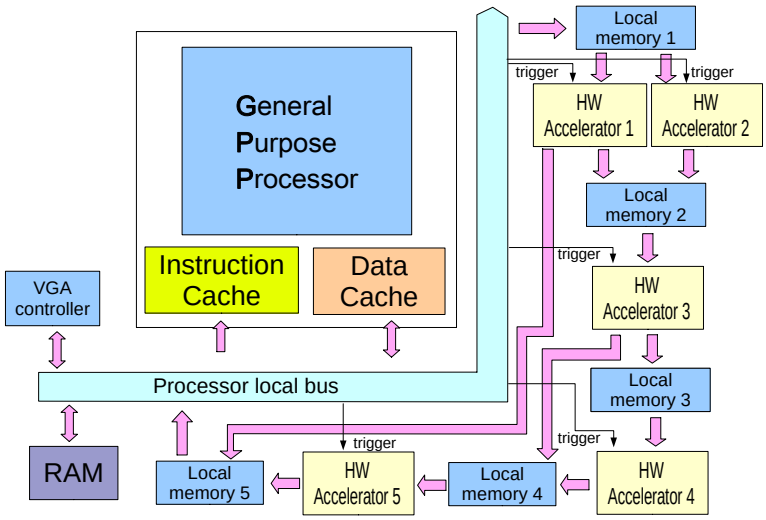


Figure 4: The heterogeneous platform simulated in paper 4

be used instead of the value computed by the accelerator.

In order to avoid cache coherency issues, Sim-Panalyzer was modified to support uncacheable memory locations. Accesses to these uncacheable memory locations are simulated as they would directly access the shared memory location without going through the caches. In order to reserve these shared memories in the simulated memory space, and also to make the simulation of different architectures possible without the need of re-compiling the simulator, the shared memory spaces are statically allocated within the RTEMS initialization phase. The start addresses of the created shared memories are then passed to the simulator via a dedicated system call.

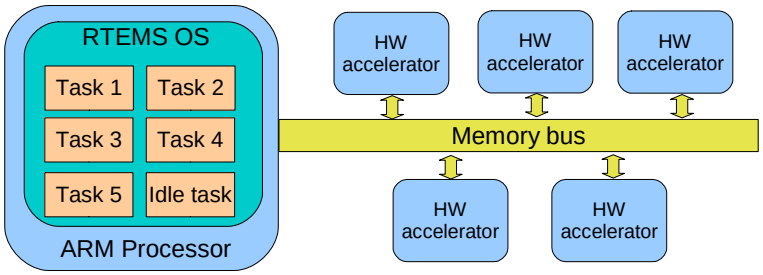


Figure 5: The heterogeneous platform simulated in paper 5

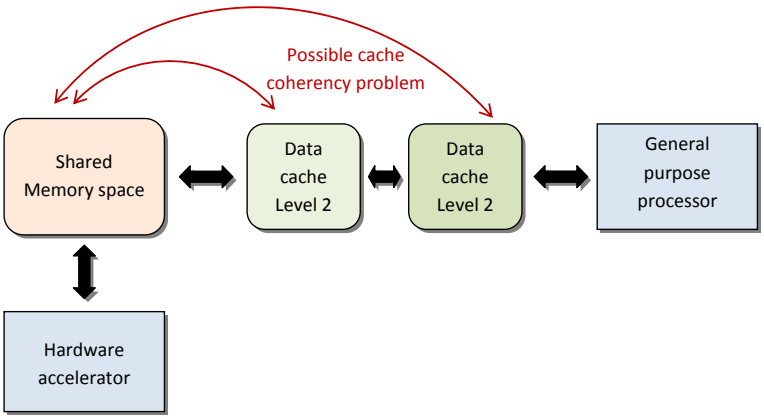


Figure 6: Cache coherency problem on a typical heterogeneous platform

4 Conclusion

The increasing number of computational intensive applications strengthens the need to maximize the battery life of portable devices. As a consequence, the simulation of embedded systems for energy consumption estimation is becoming essential if one wants to study and explore the influence of system design choices on the system energy consumption. The original publications presented in the second part of this thesis propose several frameworks for evaluating the effects of particular system and software architectures on the system energy consumption. From a software point of view Java and C based applications are studied, and from a hardware perspective systems using a general purpose processor or a heterogeneous platform with dedicated hardware accelerators are analyzed.

Papers 1 and 2 present a general framework for estimating the energy consumption of an embedded Java Virtual Machine (JVM), and establish an energy consumption cost for individual Java Opcodes. The presented opcode-level energy models are validated and provide results with an error range of -5% +10% compared to an established processor instruction-level energy simulator. However, this loss in precision is balanced by the considerable gain in execution speed between both simulations. Papers 1 and 2 can also guide developers to produce an energy-aware java application by for example limiting the use of long data type, avoiding multidimensional array and trying to use consecutive case values inside a switch statement. Moreover, the results of paper 1 and 2 can be further used for developing estimation frameworks for profiling and predicting Java application energy consumption. To the author's knowledge the results presented in papers 1 and 2 are used in the development of an energy estimation framework in [62, 63, 64].

Paper 3 defines the receiver coding gain for a DVB-H terminal in file downloading scenarios, and evaluates the cost-effectiveness of FEC algorithms in terms of

energy consumption. This publication demonstrates that a substantial energy saving is achievable in a DVB-H receiver when a forward error correction algorithm is used for file downloading scenarios. Providing an erasure model for a DVB-H network, the presented results can be used for estimating the feasible energy gain, or loss, depending on the position of the DVB-H receiver within the network.

The simulation of heterogeneous platforms is presented in paper 4 and 5. Three systems simultaneously decoding multiple videos on a dedicated heterogeneous platform are simulated in Paper 4, while paper 5 proposes a methodology for analyzing the impact of short latency hardware accelerators on a typical embedded system. Both papers point out the drawback of different mechanisms used to synchronize a hardware accelerator used as a peripheral device. Paper 4 indicates poor performance when synchronization is performed by pooling the state of a frequently accessed accelerator, and paper 5 indicates poor performance when synchronization of a short latency accelerator is achieved with interrupt. These results can help choosing the adequate synchronization mechanism for a hardware accelerator used as a peripheral device depending on its latency and usage frequency.

Because paper 3, 4 and 5 use the cycle-accurate architecture level energy simulator Sim-Panalyzer, the presented energy consumption results carry the error margin created by Sim-Panalyzer. However, with a 9% error margin in its datapath and execution unit models and a 7% error margin in its memory power model [43], Sim-panalyzer can be considered as an accurate simulator.

The static energy model for a multi-bank memory architecture and the equations governing the optimization problem for decreasing the static energy consumed by the memory architecture are presented in paper 6. This paper shows that the use of a multi-bank memory architecture can lead to a 20% static energy saving without any software optimization, nor bank need prediction, nor dedicated allocation policy. This static energy model provides the needed model if one wants to estimate the static energy consumed by a multi-bank memory architecture for a specific bank size and allocation trace.

All papers in the second part of this thesis present simulation based energy estimation approaches. The common benefit of system simulation is the ability to gather information about system performance without the need to have the system physically available. Without a simulation based approach some of the studies presented in the papers would have been complicated to conduct. This is especially true for estimating the heterogeneous platforms presented in paper 4 and 5, as well as the multi-bank memory architecture presented in paper 6. Indeed, manufacturing such platforms or memory architecture is an extremely time consuming process. During the process of system design, it is evident that the simulation of such systems is faster and more viable than manufacturing system prototypes. However, even if using a simulation based approach can be a considerably faster and cheaper approach than physically manufacturing the system, it can not always be considered as the ideal solution. First of all, there is always the need of assessing the accuracy of an estimation based approach, which means that the results

from the simulation tool need to be compared to a reference. But, in most of the cases the only reliable reference that can be used is the physical system itself. In papers 1 and 2 we also show that depending on which level the simulation approach is based, the time needed to actually run the simulation process can vary considerably. Simulating a small Java application at the processor instruction level takes several hours, whereas it takes only few second at Java opcode level. This shows that the choice of using system simulation over physical measurement must be based on the obtainable simulation speed, the number of needed simulations and the complexity of manufacturing or obtaining the studied system. However, with a growing demand for reusable hardware and software feature, mainly due to a reduction in the design to market time, we can predict that demands for system simulations performed on a high level of abstraction will most probably grow.

Bibliography

[1] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The impact of technology scaling on lifetime reliability. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 177, Washington, DC, USA, 2004. IEEE Computer Society.

[2] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[3] Gang Qu, Naoyuki Kawabe, Kimiyoshi Usami, and Miodrag Potkonjak. Function-level power estimation methodology for microprocessors. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 810–813, New York, NY, USA, 2000. ACM.

[4] Marcello Lajolo, Anand Raghunathan, Sujit Dey, Luciano Lavagno, and Alberto Sangiovanni-vincentelli. Efficient power estimation techniques for hw/sw systems. In *IEEE Alessandro Volta Memorial Workshop on Low-Power Design*, pages 191–199, 1999.

[5] 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, 1994.

[6] Tajana Šimunić, Luca Benini, and Giovanni De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *DAC '99: Proceedings of the 36th Design Automation Conference*, pages 867–872, New York, NY, USA, 1999. ACM.

[7] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low power cmos digital design. *IEEE Journal of Solid State Circuits*, 27:473–484, 1992.

[8] Ralf Kakerow. Low power design methodologies for mobile communication. In *ICCD '02: Proceedings of the 2002 IEEE International Conference on*

Computer Design: VLSI in Computers and Processors (ICCD'02), pages 8–13, Washington, DC, USA, 2002. IEEE Computer Society.

[9] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 191–201, New York, NY, USA, 2000. ACM.

[10] Pieter A. Olivier. Embedded system simulation: testdriving the toolbus. Technical report, Universiteit van Amsterdam (Netherlands), 1996.

[11] Peter Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[12] Dakai Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 35–40, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Dakai Zhu and Hakan Aydin. Energy management for real-time embedded systems with reliability requirements. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 528–534, New York, NY, USA, 2006. ACM.

[14] Nachiketh R. Potlapally, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Analyzing the energy consumption of security protocols. In *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, pages 30–35, New York, NY, USA, 2003. ACM.

[15] Daler Rakhmatov and Sarma Vrudhula. Energy management for battery-powered embedded systems. *Trans. on Embedded Computing Sys.*, 2(3):277–324, 2003.

[16] Gary J. Nutt. The simulation of computer systems in a university environment. In *ANSS '74: Proceedings of the 2nd symposium on Simulation of computer systems*, pages 25–29, Piscataway, NJ, USA, 1974. IEEE Press.

[17] M. H. MacDougall. Computer system simulation: An introduction. *ACM Computing Surveys (CSUR)*, 2(3):191–209, 1970.

[18] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The simos approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3:34–43, 1995.

[19] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 108–116, New York, NY, USA, 2002. ACM.

[20] Jie Tao, Martin Schulz, and Wolfgang Karl. A simulation tool for evaluating shared memory systems. In *ANSS '03: Proceedings of the 36th annual symposium on Simulation*, page 335, Washington, DC, USA, 2003. IEEE Computer Society.

[21] P. Magnusson and B. Werner. Efficient memory simulation in simics. In *ANSS '95: Proceedings of the 28th Annual Simulation Symposium*, page 62, Washington, DC, USA, 1995. IEEE Computer Society.

[22] Mariagiovanna Sami, Donatella Sciuto, Cristina Silvano, and Vittorio Zaccaria. Instruction-level power estimation for embedded vliw cores. In *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*, pages 34–38, New York, NY, USA, 2000. ACM.

[23] Yanbing Li and Jörg Henkel. A framework for estimation and minimizing energy dissipation of embedded hw/sw systems. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 188–193, New York, NY, USA, 1998. ACM.

[24] Marlene Wan, Yuji Ichikawa, David Lidsky, and Jan Rabaey. An energy conscious methodology for early design exploration of heterogeneous dsps. In *CICC'98: Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 111–117, May 1998.

[25] C. Brandolese, W. Fomacian, F. Salice, and D. Sciuto. An instruction-level functionality-based energy estimation model for 32-bits microprocessors. In *DAC '00: Proceedings of the 37th Design Automation Conference*, pages 346–350, 2000.

[26] Jeffry T. Russell and Margarida F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *ICCD '98: Proceedings of the International Conference on Computer Design*, pages 328–333, Oct 1998.

[27] ARM Ltd. ARM Instruction Set Simulator ARMUlator. http://www.arm.com/support/ARMulator.html.

[28] Enprofiler home page: http://ls12-www.cs.tu-dortmund.de/research/activities/encc/.

[29] Stefan Steinke, Markus Knauer, Lars Wehmeyer, and Peter Marwedel. An accurate and fine grain instruction-level energy model supporting software optimization. In *Proceedings of the Internationl Workshop: Power and Timing Modeling, Optimization and Simulation (Patmos)*, 2001.

[30] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of java grande benchmarks. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 72–80, New York, NY, USA, 1999. ACM.

[31] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, 2000.

[32] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 340–345, New York, NY, USA, 2000. ACM.

[33] Rita Yu Chen, Robert M. Owens, Mary Jane Irwin, R. S. Bajwa, and Raminder S. Bajwa. Validation of an architectural level power analysis technique. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 242–245, New York, NY, USA, 1998. ACM.

[34] Rita Yu Chen, Mary Jane Irwin, and Raminder S. Bajwa. Architecture-level power estimation and design experiments. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(1):50–66, 2001.

[35] Ping-Wen Ong and Ran-Hong Yan. Power-conscious software design-a framework for modeling software on hardware. In *Proceedings of the 1994 IEEE Symposium on Low Power Electronics*, pages 36–37, Oct 1994.

[36] Toshinori Sato, Masato Nagamatsu, and Haruyuki Tago. Power and performance simulator: Esp and its application for 100 mips/w class risc design. In *Proceedings of the IEEE Symposium on Low Power Electronics*, pages 46–47, Oct 1994.

[37] The simplescalar-arm power modeling project home page: http://www.eecs.umich.edu/˜panalyzer.

[38] Chunling Hu, Daniel A. Jimenez, and Ulrich Kremer. Toward an evaluation infrastructure for power and energy optimizations. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, page 230.2, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Allen C. Cheng and Gary S. Tyson. An energy efficient instruction set synthesis framework for low power embedded system designs. *IEEE Transactions on Computers*, 54(6):698–713, 2005.

[40] Johann Großädl, Stefan Tillich, Christian Rechberger, Michael Hofmann, and Marcel Medwed. Energy evaluation of software implementations of block ciphers under memory constraints. In *DATE '07: Proceedings of the conference*

on Design, automation and test in Europe, pages 1110–1115, San Jose, CA, USA, 2007. EDA Consortium.

[41] Radu Cornea, Alex Nicolau, and Nikil Dutt. Video stream annotations for energy trade-offs in multimedia applications. In *ISPDC '06: Proceedings of the Proceedings of The Fifth International Symposium on Parallel and Distributed Computing*, pages 17–23, Washington, DC, USA, 2006. IEEE Computer Society.

[42] Todd Austin, Eric Larson, and Dan Erns. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002.

[43] Sim-panalyzer 2.0 reference manual. Technical report, University of Michigan, University of Colorado.

[44] Vijay K. Bhargava. Forward error correction schemes for digital communications. *Communications Magazine, IEEE*, 21(1):11–19, Jan 1983.

[45] Joachim Hagenauer and Erich Lutz. Forward error correction coding for fading compensation in mobile satellite channels. *IEEE Journal on Selected Areas in Communications*, 5(2):215–225, Feb 1987.

[46] Rohit Puri, K. Ramchandran, K. W. Lee, and V. Bharghavan. Forward error correction (fec) codes based multiple description coding for internet video streaming and multicast. *Signal Processing: Image Communication*, 16(8):745 – 762, 2001.

[47] Bernd Lamparter, Otto Bohrer, Wolfgang Effelsberg, and Volker Turau. Adaptable forward error correction for multimedia data streams. Technical report, Reihe Informatik 9/93, Universitat Mannheim, 1993.

[48] Jean-Chrysostome Bolot, Sacha Fosse-Parisis, and Don Towsley. Adaptive fec-based error control for internet telephony. In *INFOCOM '99: Proceedings of the eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies.*, pages 1453–1460, 1999.

[49] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. In *Readings in hardware/software co-design*, pages 527–543. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[50] Rabin A. Sugumar and Santosh G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems (TOCS)*, 13(1):32–56, 1995.

[51] Jie Tao and Wolfgang Karl. Detailed cache simulation for detecting bottleneck, miss reason and optimization potentialities. In *Valuetools '06: Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, page 62, New York, NY, USA, 2006. ACM.

[52] Erik Berg and Erik Hagersten. Sip: Performance tuning through source code interdependence. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 177–186, Paderborn, Germany, August 2002.

[53] Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson. Tuning memory performance of sequential and parallel programs. *IEEE Computer*, 28(4):32–40, 1995.

[54] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: problems and solutions. *ACM Crossroads*, 5:2, 1999.

[55] Wei Wang, Qigang Wang, Wei Wei, and Dong Liu. Modeling and evaluating heterogeneous memory architectures by trace-driven simulation. In *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors*, pages 369–376, New York, NY, USA, 2008. ACM.

[56] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 1624–1633, New York, NY, USA, 1999. ACM.

[57] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 161–171, New York, NY, USA, 2000. ACM.

[58] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997.

[59] Michael Laurenzano, Beth Simon, Allan Snavely, and Meghan Gunn. Low cost trace-driven memory simulation using simpoint. *ACM SIGARCH Computer Architecture News*, 33(5):81–86, 2005.

[60] V. De La Luz, M. Kandemir, and I. Kolcu. Automatic data migration for reducing energy consumption in multi-bank memory systems. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 213–218, New York, NY, USA, 2002. ACM.

[61] Hanene Ben Fradj, Sébastien Icart, Cécile Belleudy, and Michel Auguin. Energy optimization of a multi-bank main memory. In *Proceedings of the 6th International Workshop, Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 196–205, 2006.

[62] Jorge Navas, Mario Mendez-Lojo, and Manuel Hermenegildo. Safe upper-bounds inference of energy consumption for java bytecode applications. In *Proceedings of the sixth NASA Langley Formal Methods Workshop*, pages 29–32. Langley Research Center, 2008.

[63] Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Alexei Kounine, Maxime Monod, and Jesper Honig Spring. The weight-watcher service and its lightweight implementation. In *IC-SAMOS'07: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 118–127, 2007.

[64] Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. Pervasive computing with frugal objects. In *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pages 13–18, Washington, DC, USA, 2007. IEEE Computer Society.

Part II

Original Publications

Paper I

An Energy Consumption Model for an Embedded Java Virtual Machine

Sébastien Lafond and Johan Lilius

Originally published in: proceedings of the *19th International Conference on Architecture of Computing Systems - ARCS 2006*, Lecture Notes in Computer Science, Vol. 3894, pages 311-325. Springer-Verlag, 2006.

An Energy Consumption Model for an Embedded Java Virtual Machine

Sébastien Lafond[1] and Johan Lilius[2]

[1] Turku Centre for Computer Science, Embedded Systems Laboratory,
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
sebastien.lafond@abo.fi
[2] Åbo Akademi University, Department of Computer Science,
Lemminkäinengatan 14A, FIN-20520 Åbo, Finland
johan.lilius@abo.fi

Abstract. In this paper we establish a general framework for estimating the energy consumption of an embedded Java virtual machine (JVM). We have designed a number of experiments to find the constant overhead of the Virtual Machine and establish an energy consumption cost for individual Java Opcodes. The results show that there is a basic constant overhead for every Java program, and that a subset of Java opcodes have an almost constant energy cost. We also show that memory access is a crucial energy consumption component.

1 Introduction

In recent years we have seen an explosion of markets for portable electronic devices such as PDAs, personal communicators and mobile phones. These battery-operated devices provide more and more functionalities and as a consequence become more and more complex. They have in common strong constraints on energy consumption, and thus maximizing battery life for such devices is crucial.

Several techniques have been developed to optimize the energy consumption of embedded systems. Those techniques can be hardware based solutions, as well as software or co-design solutions [1]. Techniques for low power optimization of software have been mostly applied on processor instructions level [2, 3] by mainly using processor specific instructions [4, 5]. Techniques on memory management have also been widely applied for optimizing energy consumption [6, 7].

At the same time, the size and complexity of applications and development constraints like getting the product to market on time, make necessary the use of high-level languages. Due to the wide diversity of hardware and OS used in the world of handheld devices, portability across systems is not easy and needs efforts. Java language eases portability by allowing application developments with an abstraction of the target platform, making the concept "write once, run it anywhere" possible.

In this paper we establish a general framework for estimating the energy consumption of an embedded Java virtual machine. We present a number of experiments to estimate the constant overhead of the JVM energy consumption and establishe an energy consumption cost for individual Java Opcodes.

W. Grass et al. (Eds.): ARCS 2006, LNCS 3894, pp. 311–325, 2006.

The major contributions of this paper are a better understanding of the energy consumption distribution of an embedded Java virtual machine (JVM) and the definition of the energy cost for the Java bytecodes.

The remainder of this paper is organized as follows. Section 2 proposes a methodology scheme used to characterize the energy consumption of an embedded Java Virtual Machine. Section 3 presents several experiments in order to define some constant overheads of the JVM and comments the repartition of the JVM energy consumption. Section 4 presents a measurement methodology used to define the energy cost of Java bytecode by cost comparison between two appropriate class files. Finally, section 5 concludes the paper and suggests future possible work. This paper is presenting the main results of [8] where more example graphs and results can be found.

2 An Energy Consumption Model of Java Applications

The Java Virtual machine is an abstract machine, making the interface between platform independent applications and the hardware, through a possible operating system. Thus the use of Java language can be seen as adding one more layer, the Java virtual machine, between the hardware and software layers. We want to study how well applying estimation techniques on the virtual machine opcodes level can be done, similarly to what has been done on processor instructions level. Figure 1 shows a simple view of the JVM life cycle. An efficient energy model should characterize each stage of the life cycle model, and thus shows in which stage(s) effort needs to be concentrated to achieve energy optimization. It seems obvious that such model needs to consider the system's hardware and software configuration and therefore is not directly portable. But the methodology used to build it can easily be applied on different configurations by changing the platform configuration parameters.

As shown in [9] the memory consumption must also be included in the model, as the memory might represent the biggest source of energy consumption on a typical embedded system. This is even more important to take into account as the JVM is a stack machine and will therefore have a relatively high memory activity.

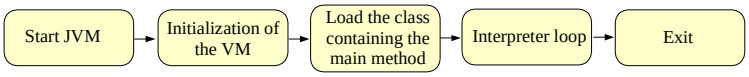


Fig. 1. Simple view of the JVM life cycle

2.1 Measurements Methodology

We chose to use the Sun Microsystems K Virtual Machine (KVM), CLDC v1.0.3, as it has been developed for a resource-constrained platform and has its source code freely available. KVM is a small virtual machine containing about 50-80 Kb of object code in its standard configuration and has a total memory footprint

in the range of 128-256 Kb. KVM can run on a 16-bit or 32-bit RISC/CISC processor clocked from 25MHz.

To build an energy model of the KVM we adapted the energy profiler *enprofiler* [10] developed by the Embedded Systems Groups at Dortmund University. The adaptation was done in order to integrate the Java environment in the results provided by the energy profiler. With the adaptation, when summing up the energy cost for each instruction execution or memory access the *enprofiler* checks in which KVM stage the event occurred and increments the corresponding costs variable. Enprofiler is a processor instructions level energy profiler for ARM7TDMI processor cores operating in Thumb mode [11] and integrating the consumption of memory accesses. It has been built from physical measurements done on an Atmel AT91EB01 evaluation board consisting of a AT91M40400 processor clocked at 33MHz and an external 512K bytes SRAM. A detailed description of the energy model used by *enprofiler* is given in [12]. According to [12] *enprofiler* shows a precision of 1.7% for the cost measurement of 12 instructions in an endless loop.

Figure 2 shows the measurements methodology scheme used to characterize each stage of the KVM life cycle. The Java class generator generates, from template classes, Java applications with the possibility to modify parameters inside the class source code. With the Java Code Compact (JCC) we compile and link together the JVM source code and the generated Java application. The executable code is run on the ARM7TDMI emulator ARMulator, which traces instructions, memory accesses and events that occur during the application execution. From this trace, we extract all information concerning the memory access addresses, size and type (read, write, sequential, non-sequential), the instructions addresses and their corresponding processor opcodes. The energy profiler *enprofiler* reads the emulator trace and accesses databases providing processor instruction costs and the cost of a memory access depending of its address, size

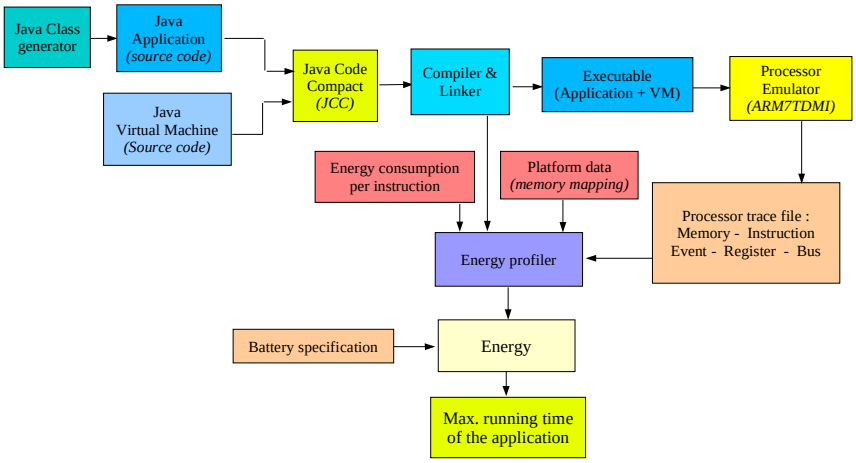


Fig. 2. Measurements methodology scheme

and type. The energy profiler estimates the energy consumed by the application and provides information on how the energy is distributed between the processor and memories for each KVM stage.

2.2 Energy Profiler

The energy profiler provides the number of instructions, memory accesses and garbage collections that occur during each KVM stage. It needs as input information on the JVM stage addresses inside the emulator trace. These addresses are provided by the linker from which eight useful address symbols are collected:

- main: this symbol represents the *main()* function of KVM, and is used by the energy profiler to detect the start of the KVM execution.
- StartJVM: represents the *StartJVM(argc, argv)* function (in StartJVM.c source file). This function only checks if the user gave a class name as argument, and then calls the *KVM_Start()* function.
- KVM_Start: represents the *KVM_Start()* function (in StartJVM.c source file). This function initializes the VM, the global variables, the profiling variables, the memory system, the hashtable, the class loading interface, the Java system classes, the class file verifier and the event handling system. It also initializes the multithreading system after loading the main application class.
- garbageCollect: represents the *garbageCollect()* function (in garbage.c source file) that performs a mark-and-sweep garbage collection.
- ExitGarbage : the ExitGarbage symbol was added into the KVM source code in order to detect the end of the garbage collector.
- Interpret : represents the *Interpret()* function (in execute.c source file) that runs the interpreter loop.
- KVM_Cleanup : KVM_Cleanup represents the *KVM_Cleanup()* function (in StartJVM.c source file). It runs several finalization functions when the VM is shut down.
- ExitVM : This symbol is used to detect the end of the KVM execution.

3 Experiments

We have run the measurement process over several representative benchmarks to characterize each stage of the KVM life cycle and determine if some stages are dominant. The benchmarks used are: a) the dhrystone benchmark, b) parts of The Java Grande Forum Benchmark Suite and the DHPC Java Grande Benchmarks. In addition to these established benchmarks we also used as reference an empty application in order to reflect the KVM basic costs. Dedicated intensive allocation applications was also used in order to study the behavior of the KVM stage costs. All benchmarks can be retrieve from [13]. For all measurements, if not explicitly expressed the KVM was compiled with an heap size of 256 Kb.

3.1 Benchmarks

Empty application: We run the empty application through the measurement process in order to find out if overhead constants in the KVM energy consumption can be determined. We can predict that one or several stage(s), like StartJVM, will have a constant energy consumption, as they have an application independent behavior. Its source code is the following:

```
public class HelloWord {
 public static void main(String arg[])
  {
  //nothing to do
  }
}
```

Intensive allocation applications: Two intensive allocation applications were used in order to study a possible application related evolutions in the KVM costs. The first application, called alloc1, instantiates inside a loop one object of class MyClass. This class doesn't contain any field and has just one *main* method. Each new class MyClass created by main is stored in the heap, and will contain only a reference to the class definitions area. Each instantiation will create a new stack frame and call the MyClass constructor which by default will only call java/lang/Object constructor method. The stack frame created by the main method contains two operand stacks and three local variables to store the object reference, the length and the loop index. This application is used to observe the evolution of different KVM stage costs with the length of the loop. The source code for alloc1 is the following:

```
public class MyClass {
  public static void main(String arg[])
  {
   int length = X;
   for(int i=0; i<=length ; i++) {
   new Myclass();
   }
  }
}
```

The second intensive allocation application, called alloc2, is similar to the precedent one with the difference that MyClass contain one field define by an integer array of size 500. Alloc2 is used to observe the weight that can take the garbage collector in comparison to the other KVM stages in extreme allocation rate. As each new instance takes approximatively 2Kb, with an heap size of 128Kb the garbage collector needs to be triggered every 64th objects created in the loop to reclaim the heap space occupied by the unreferenced objects. The source code for alloc2 is the following:

```
public class MyClass {
int[] tab = new int[500];
  public static void main(String arg[])
  {
   int length = X ;
   for(int i=0; i<=length ; i++) {
   new Myclass();
   }
  }
}
```

Dhrystone: Dhrystone tests the system's integer performance. It is a well established benchmark for performance measurement of general purpose system. We conducted the measurement process with two test executions of 50 and 250 benchmark runs.

Table 1. Benchmarks used from Java Grande Forum Benchmark suite

Low level operation benchmarks	
Name	Short description
Arith	Execution of arithmetic operations
Assign	Variable assignment
Create	Creating objects and arrays
Exception	Exception handling
Loop	Loop overheads
Math	Execution of maths library operations
Method	Method invocation
Generic	Local and Static variable handling

Java Grande Benchmarks: We used the sequential benchmarks which are the one suitable for single processor execution. Several low level operation benchmarks was used from the Java Grande Forum Benchmark Suite and the DHPC Java Grande Benchmarks. Table 1 summarize all benchmarks used for our study.

3.2 Results

This section presents the results obtained by the introduced applications and benchmarks through the measurement process.

Empty application: The empty application has been used in order to find out if overhead constants in the KVM energy consumption can be determined.

Table 2 shows the obtained results and figure 3 presents the energy consumption distribution among all KVM stages and also the distribution between the energy consumed by memory accesses and processor instruction execution.

We can make some remarks from figure 3. Even if this application does absolutely nothing, it has to be noticed that the interpreter stage represents about

Table 2. Empty application - Energy consumption of KVM's stages in μJ

StartJVM Inst.	StartJVM Mem.	KVMStart Inst.	KVMStart Mem.	Interpr. Inst.	Interpr. Mem.
9,42	20,08,	748,81	1639,18	3552,28	8273,34
		KVM Clean Inst.	KVM Clean Mem.		
		144,92	326,38		

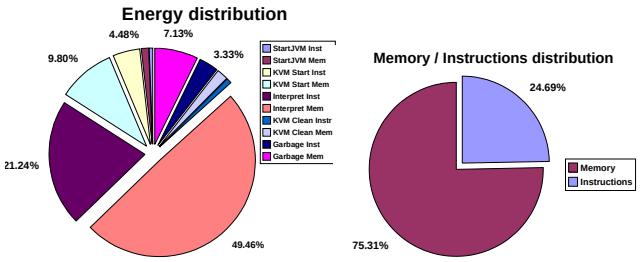


Fig. 3. Empty Application - Energy consumption distributions

70 % of the consumed energy from all stages, and memory accesses represent 75% of the total consumed energy. As the application was *'empty'* the values in table 2 represent the KVM basic costs or the minimal overhead energy cost that any application will have to dissipate.

Intensive allocation applications: From the alloc1 results in figure 4 we note that only the energy consumed by the interpreter is dependent on the loop length value. All other stages of the KVM consume a constant energy including the garbage collector, as the maximum number of created object was not enough to fill up the Java heap and trigger off a garbage collection. It is also important to notice that the energy consumed by the interpreter stage is linear and proportional to the loop length. This can be explain by the fact that the interpreter is looping over a number of constant Java opcodes. These opcodes are:

```
4 goto 18
7 new\#2  -> create a new 'MyClass' object in the heap
10 dup    -> duplicate new object reference in the operand stack
11 invokespecial \#3 -> call the constructor
14 pop -> remove the top of the operand stack
17 iinc 2 1 -> increment the second local variable by 1
18 iload\_2 -> load 2nd local variable in operand stack (i)
19 iload\_1 -> load 1st local variable in operand stack (length)
20 if\_icmple 65543
```

As the energy profiler evaluates the cost of a memory access according to the memory technology, i.e. have for each memory type (RAM, ROM, Flash, etc.) an average cost for each access type regardless of its address, and as the *new* opcode allocates the same amount of memory for all created (and already resolved) objects, it will have an identical cost for each execution.

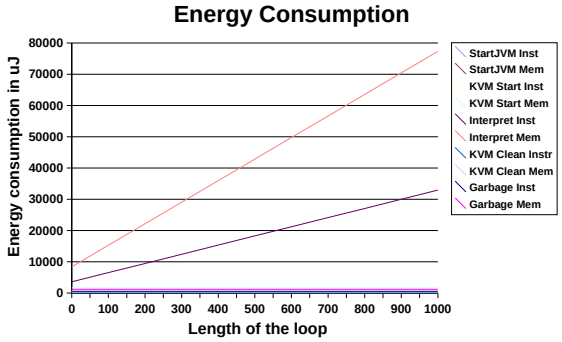


Fig. 4. Alloc1 - KVM's stages energy consumption depending of the loop length

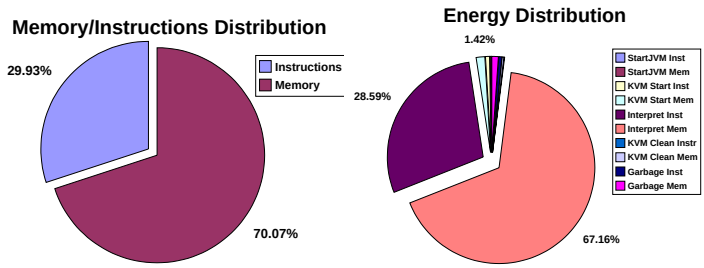


Fig. 5. Alloc1 - Energy distribution for loop length equal to 1000

The energy distribution for a loop length of 1000 presented in figure 5, is similar to the first experiment with an interpreter stage even more dominant, representing over 95% of the total energy consumed.

Alloc2 application was used to observe the garbage collector weight in comparison to other KVM stages. Several factors can influence the garbage collection behavior and thus its energy consumption: the size of the heap, the sizes and numbers of live or dead objects, and heap fragmentation. However, as shown on figure 6, the garbage collection stage will hardly exceed more than 15% of the total energy consumed even for application with intensive allocation rate. Table 3 shows the energy values consumed by the interpreter and garbage collector for alloc2 application with a loop length of 1000 where the garbage collection represent 13,65% of the interpreter stage energy consumption.

Benchmarks: Table 4 and 5 gather the results for all benchmarks. Table 4 shows for the used benchmarks the energy values in μJ for StartJVM, KVMStart and KVMClean stages. We can notice that the obtained values for each stage are very similar for all benchmarks, and there values and variations extremely small compare to the interpreter stage values show in table 5 (in mJ). We can say that with an average of 98% of the total energy consumption the interpreter stage is fare ahead the stage where the energy consumption is dissipated inside the KVM, and that StartJVM, KVMStart and KVMClean have an almost

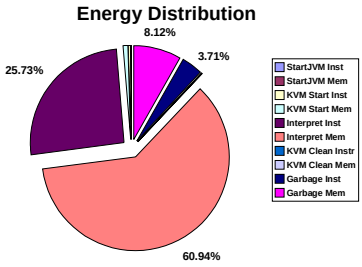


Fig. 6. Garbage collection weight

Table 3. Energy consumption values for a loop length of 1000 in μJ

Interpreter Inst.	Interpreter Mem.	Garb. Collect. Inst.	Garb. Collect. Mem.
54 035	127 949	7 789	17 057

Table 4. Stable energy costs for StartJVM ,KVMStart and KVMClean stages in μJ

	StartJvm		KVMStart		KVMClean	
Benchmark	Instuction	Memory	Instuction	Memory	Instruction	Memory
Dhrystone250	9,42	20,08	857,74	1868,40	155,41	350,31
Dhrystone50	9,42	20,08	849,82	1851,51	154,74	348,82
Arith	9,42	20,08	815,78	1776,04	145,67	328,40
Assign	9,42	20,08	823,32	1791,93	145,94	329
Create	9,42	20,08	807,81	1833,57	147,48	335,21
Exception	9,42	20,08	814,08	1772,99	145,94	329
Loop	9,42	20,08	810,01	1764,06	145,67	328,40
Method	9,42	20,08	823,89	1793,72	146,75	330,93
Generic	9,42	20,08	838,76	1828,55	152,78	344,39
Math	9,42	20,08	823,89	1793,72	146,75	330,93

Table 5. Interpreter stage energy cost and weight in mJ

	Dhrystone250	Dhrystone50	Arith	Assign	Create	Exception
Inst.	97-29.65%	88-29.30%	877-29.21%	2380-29,87%	1053-26,38%	2250-29,82%
Mem.	850-69.90%	207-68.92%	2121-70.62%	5584-70,05%	2779-69,61%	5475-70,04%

	Loop	Math	Method	Generic
Inst.	533-29,32%	2718-29,77%	533-29,86%	611-29,65%
Mem.	228-69,03%	6408-70,17%	1246-69,84%	1445-70,09%

constant and insignificant energy consumption. All measurements were done on an opteron 244 1.8GHz machine with 4Gb of RAM, and for the slowest benchmark JGFMathBench the measurement process took about 36 hours.

From all experiments done it is clear that the interpreter stage is far ahead the main source of energy consumption and a better comprehension of it is needed if someone wants to achieve energy optimization on the KVM. As the interpreter reads and executes the Java bytecode, having a closer view on the interpreter implies increasing the granularity of its energy consumption model by looking at the cost of each Java opcode interpreted.

4 Java Opcode Energy Cost

In order to get a better understanding of the interpreter energy consumption, an evaluation of each Java opcode energy cost is needed. As a strict class file structure needs to be respected, it is not possible to only execute one Java opcode. Thus a cost comparison between two class files is needed to estimate the cost difference between them. The general measurements methodology scheme used to characterize each KVM stage life cycle can be re-used with different inputs. Instead of using Java source code files we will use as input appropriate byte-code executable class files.

4.1 Measurements Methodology

Figure 7 shows an abstract view of the class files generator used to create two class files, named ClassFile and ClassFile_Ref. The opcode behavior towards the Java operand stack and the local variables array has to be defined for each studied Java opcode, i.e. provide the operand stack state needed before and resulting after the studied opcode execution as well as the number of local variables needed. Figure 8 shows an example of generated bytecode classes for the Java opcode *NOP (0x00)*. In this example ClassFile method 1, the *main* method, executes 256 *NOP* opcodes when the ClassFile_Ref method 1 executes only the

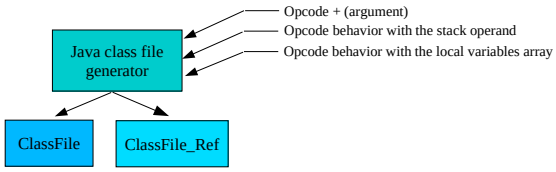


Fig. 7. Bytecode executable class file generator

ClassFile

```
Method 1:
0000d8 0009      access flags = 9
0000da 0008      name = #8<main>
0000dc 0009      descriptor = #9<([Ljava/lang/String;)V>
0000de 0001      1 field/method attributes:
                  field/method attribute 0
0000e0 0006        name = #6<Code>
0000e2 00000119  length = 281
0000e6 0000        max stack: 0
0000e8 0001        max locals: 1
0000ea 00000101  code length: 257
0000ee 00          0 nop
0000ef 00          1 nop
0000f0 00          2 nop
0000f1 00          3 nop
.............
0001ed 00          255 nop
0001ee b1          256 return
0001ef 0000        0 exception table entries:
```

ClassFile_Ref

```
Method 1:
0000d8 0009      access flags = 9
0000da 0008      name = #8<main>
0000dc 0009      descriptor = #9<([Ljava/lang/String;)V>
0000de 0001      1 field/method attributes:
                  field/method attribute 0
0000e0 0006        name = #6<Code>
0000e2 00000019    length = 25
0000e6 0000        max stack: 0
0000e8 0001        max locals: 1
0000ea 00000001    code length: 1
0000ee b1          0 return
0000ef 0000        0 exception table entries:
```

Fig. 8. Example of generated byte-code class files

compulsory *return* opcode in order to return *void* from the main method. By comparing the interpreter energy consumption for both class files we can get the energy consumption estimation for 256 *NOP* executions and thus the energy cost of one *NOP* opcode.

To ensure the estimation quality for each opcode we generate several pairs of class files executing the studied opcode and also monitor the possible energy consumption differences between all other KVM stages. All measurements were done on a Linux 700Mhz Pentium III machine with 256MB of RAM, and on average the estimation of a Java opcode cost took 3 minutes.

4.2 Results

From all Java opcodes we will not study the 51 opcodes which handle floating point values as floating point is not supported by the CLDC specification. The opcode *athrow* was also omitted from this study, it is not possible to directly estimate its energy cost using this comparison method as its cost can not be extracted from the context cost. All the same, in table 5 in [13] we can see from the opcode *checkcast* the cost of throwing an *ClassCastExeption* exception and exiting the KVM.

Due to space requirement all obtained values for each studied opcode are published in [13], where the opcodes are divided in six functional groups:

Stack and local variable operations opcodes: Tables 2 and 3 in [13] show the results concerning opcodes that operate on the operand stack and local variable. We can notice that loading a value from the local variables array to the operand stack is lightly more expensive than storing the same value back to the local variable. It is also interesting to note that the opcode *bipush* consumes about 9% less energy than *iload* and 5% less than *ilaod_x*. Thus it is more energy efficient to load an constant integer lower than 256 into the operand stack using *bipush* than initializing the local variable array with the constant and use *iload* or *ilaod_x*. The same is true if a constant integer lower than 65536 has to be loaded into the operand stack, it will be more efficient to use the opcode *bipush* instead of *iload*. But in case the integer constant can be stored in the first 4 local variables then *iload_x* becomes the most efficient opcode.

Type conversion opcodes: Table 1 in [13] shows the results for opcodes that convert value from one primitive type to another. The costs are in the same range as the stack and local variable operations opcodes as the conversion opcodes pop a value from the stack, perform a right shift or truncate the popped value and push back the result.

Arithmetic opcodes: Table 4 in [13] shows the costs for arithmetic opcodes. As it was easy to predict, the cost of an arithmetic operation is dependent on the type of the operands and the operation. Operations on long types are about 50% more expensive than on integers, except for the division of types long which is about two times more expensive than to divide integers.

Logic opcodes: As for the arithmetic opcodes, the cost of logic opcodes is also depending of the type of the operand and operations on longs are from 23% to 37% more expensive than operation on integers. Table 9 in [13] shows the costs for logic opcodes.

Control flow opcodes: The control flow opcodes are the opcodes that implement the following Java language statements: *do-while*, *while*, *if*, *if-else*, *for* and *switch*. Table 8 in [13] shows the cost for the 25 control flow opcodes. For all conditional *if* opcodes (i.e. opcodes from 0x99 to 0xa6 and *ifnull*, *ifnonnull*) the energy cost depends on a two values comparison success. If the comparison success the VM jumps to a target defined by the opcode operands, in the other case the VM continues by executing the following opcodes. The KVM *lookupswitch* implementation uses the binary search algorithm to retrieve the branch offsets associated with the case values of the switch statement. In consequence, the *lookupswitch* cost depends on the number of needed iterations through the binary tree which is determined by the position of the researched case value in the tree. As on average for a binary tree of size n it takes $(\log_2 n - 1)$ iterations to found the researched value, it is possible to determine an *lookupswitch* average cost depending on the number of case values included in the switch statement. The *tableswitch* opcode performs the same task as *lookupswitch*, with the difference that it requires a consecutive list of case values contained between one low and high endpoint. Thus the VM knows in advance the position of all case values so that the retrieving cost is the same for all cases. Compared with *lookupswitch*, *tableswitch* has a lower energy cost but generates all the more bigger class file size as the gape between the case values is great.

Objects and arrays opcodes: Tables 5 and 6 in [13] show the cost of opcodes that create and manipulate arrays and objects. The creation cost, with *newarray*, of a single dimension array of primitive type integer, long, short, byte, char or boolean is not directly dependent on array type and size, but more on the memory size that needs to be allocated for its creation. That means that the creation cost is identical for an integers array of size 8, a shorts array of size 16, or a longs array of size 4. The creation cost, with *multiarray*, of multidimensional arrays is dependent on the array dimensions and dimensions indexes values. Each dimension adds a basic cost to the array creation cost, thus creating a 2*2*2 integers array will be 70% more expensive than creating a 2*4 integers array, and especially 18 times more expensive than creating a single dimension integers array of size 8. Moreover, in order to access to one multidimensional array value the JVM has to retrieve from the first dimension the second dimension address and so one until it reaches the last dimension.

The objects creation cost depends on the objects themselves, i.e on the type and size of their constant pool, interfaces, fields,methods and their super-classes, and also on their resolution flags inside each class constant pool. A new object is resolved only once within a same class, and its address is stored in the constant pool structure of the class. Table 5 in [13] shows as an example the creation cost of an object of type *java.lang.Object* and *java.lang.String*. In addition, table 5 in [13] refers to two objects called *Class* and *subClass* which is a empty (none

interface,field nor method) sub class of *nonResolvedClass* itself empty sub class of *java.lang.Object*.

Method invocation and return opcodes: Because invoking a method implies returning from it at some point, table 7 in [13] shows the costs of different invoke/return pairs. They all invoke an empty 'already resolved' method within the same class or instance. We can notice from this table that calling a static, public or private method costs almost the same, and that the type of the returned value has not a great influence on the overall cost.

It is also important to compare all obtained values with the *NOP* energy consumption. As the opcode *NOP* is the first case statement in the interpreter switch and doesn't execute any instruction, its energy consumption represents the minimum overhead cost due to the interpreter mechanism. For the most of the stack and local variable operation opcodes the interpreter mechanism overhead represents about 70% of their energy consumption.

The obtained values allow us to get an estimation of how long the KVM will run for a given battery. If we suppose that on average the execution of one Java opcode consumes a total of 3.372μJ and is executed in 200 cycles, the average power dissipated by the processor (clocked at 33MHz) to execute Java opcodes is 0.556 Watt. Thus for the processor supply voltage sets at 3.3 Volts, an ideal 3.3 Volts 500 mAh battery will allow the KVM to run for 200 minutes.

4.3 Opcode Costs Verification

In order to verify the obtained opcode costs we calculated for each benchmark execution the value $\sum(Opcodecost * OpcodeOccurrence)$. The computed value was then compared with the cost given by the energy profiler for the interpreter stage. The occurrence for each opcode was calculated thanks to the KVM tracing ability. For control flow opcodes we checked if the branch was taken or not to attribute the correct opcode cost, but to keep the verification simple we didn't looked at the type of variable handled by $putfield$, $getfield$, *putstatic* and *getstatic*. There respective cost for handling integer was used for all occurrences. In addition for all other none static opcode costs only the respective basic cost was used. The benchmark *Exception* from the Java Grande Forum Benchmark Suite was not used as we didn't studied the cost for the opcode *athrow*.

Table 6 presents the normalized verification results where the value 100 represent for each benchmark the energy cost given by the energy profiler for the interpreter stage. For each benchmark the accuracy obtained by calculating the value $\sum(Opcodecost * OpcodeOccurrence)$ is staying between -5 and +10% of the cost given by the energy profiler. But this lost in precision has to be balance with the time needed to compute it. It takes only few seconds to calculate the occurrence

Table 6. Verification results

Dhrystone50	Arith	Assign	Loop	Create	Method	Math	Generic
103,99	105,31	95,55	100,30	97,95	102,51	96,74	109,43

for each opcode and compute the value $\sum(Opcodecost * OpcodeOccurrence)$, compare to several hours needed by the energy profiler.

5 Conclusion

Several observations have been done in this paper concerning the energy consumption of the KVM. For the hardware configuration fixed by the energy profiler, the distribution between the processor and memories is constant over the KVM execution with 70% of the energy consumed by memory accesses. This shows the major importance of the memories for embedded system runtime performance.

This paper can also guide developers to produce energy-aware java application by limiting the use of long data type, avoiding multidimentional array and trying to use consecutive case values inside a switch statement. Furthermore, the opcodes energy cost can be helpful for developing a energy-aware Java compiler as well as optimizing the JVM by pointing out the expensive opcodes. This paper shows the first steps toward an energy aware performance analysis tool for Java application, as a such tool would ask a more detailed model for a subset of opcodes.

Also as the interpreter mechanism overhead cost is a predominant factor in opcode execution cost, it will be interesting in the future to look at the cost differences between the two possible Java execution modes: interpreted or JIT compilation. JIT compilation increases significantly the execution speed, but in the same time increases memory footprint. A trade-off between execution time and memory footprint size will certainly have to be found to reach the optimum optimization point for energy consumption.

References

1. F. Parain, M. Banatre, G.C.T.H.V.I.J.P.L.: Techniques de reduction de la consommation dans les systemes embarques temps-reel. Technical report, INRIA Rennes (2000)
2. Vivek Tiwari and Sharad Malik and Andrew Wolfe: Power Analysis of Embedded Software. In: International Conference on Computer-Aided Design, San Jose CA. (1994)
3. Anil Seth, Ravindra B Keskar, R.: Algorithms for energy optimization using processor instructions. In: International conference on Compilers, architecture, and synthesis for embedded systems- Atlanta, Georgia, USA. (2001)
4. Wen-Tsong Shiue: Retargetable Compilation for Low Power. Technical report, (Silicon Metrics Corporation)
5. Mike Tien-Chien Lee, Vivek Tiwari, S.: Power analysis and low-power scheduling. In: International Symposium on System Synthesis. (1995)
6. Catherine H. Gebotys: Low Energy Memory and Register Allocation Using Network Flow. In: Design Automation Conference. (1997) 435–440
7. Fan, X., Ellis, C., Lebeck, A.: Memory controller policies for DRAM power management. International Symposium on Low Power Electronics and Design (ISLPED) (2001)

8. Lafond, S., Lilius, J.: An energy consumption model for java vitual machine. Technical Report 597, Turku Centre for Computer Science (2004)
9. Kaushik Roy, M.C.J.: Software design for low power. In: Low Power Design in Deep Submicron Electronics. (1997) 433–460
10. Enprofiler: (http://ls12-www.cs.uni-dortmund.de/research/encc/)
11. An introduction to thumb. Technical report, Advenced RISC Machines Ltd (1995)
12. Stefan Steinke, Markus Knauer, L.W.P.M.: An accurate fine grain instruction-level energy model supporting software optimization. In: PATMOS 01. (2001)
13. (http://www.abo.fi/~slafond/javacosts)

Appendix to paper 1

Table 1: Opcodes costs, conversion opcodes

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
i2l 0x85	0.928660	2.200260	198	50
l2i 0x88	0.857440	2.037840	184	47
i2b 0x91	0.928540	2.200260	198.0	50
i2c 0x92	0.928520	2.200260	198	50
i2s 0x93	0.928540	2.200260	198	50

Table 2: Opcodes costs, stack and local variable operations-part 1/2

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
nop 0x0	0.831440	1.989840	178	45
aconst_null 0x1	0.890020	2.126940	190	49
iconst_m1 0x2	0.899160	2.150940	192	50
iconst_0 0x3	0.890020	2.126940	190	49
iconst_1 0x4	0.890020	2.126940	190	49
iconst_2 0x5	0.890020	2.126940	190	49
iconst_3 0x6	0.890020	2.126940	190	49
iconst_4 0x7	0.889760	2.126940	190	49
iconst_5 0x8	0.890020	2.126940	190	49
lconst_0 0x9	0.922300	2.192040	196	50
lconst_1 0xa	0.930960	2.216040	198	51
bipush 0x10	0.926900	2.214420	198	52
sipush 0x11	0.990360	2.373900	212	58
iload 0x15	1.013700	2.434380	216	55
lload 0x16	1.167820	2.815440	248	63
aload 0x19	1.013700	2.434380	216	55
iload_0 0x1a	0.968120	2.322900	206	51
iload_1 0x1b	0.968120	2.322900	206	51
iload_2 0x1c	0.968120	2.322900	206	51
iload_3 0x1d	0.968120	2.322900	206	51
lload_0 0x1e	1.104800	2.655960	234	57
lload_1 0x1f	1.104800	2.655960	234	57
lload_2 0x20	1.104800	2.655960	234	57
lload_3 0x21	1.104800	2.655960	234	57
aload_0 0x2a	0.968120	2.322900	206	51
aload_1 0x2b	0.968120	2.322900	206	51
aload_2 0x2c	0.968120	2.322900	206	51
aload_3 0x2d	0.968120	2.322900	206	51
istore 0x36	1.004140	2.410380	214	54
lstore 0x37	1.148940	2.767440	244	61
astore 0x3a	1.004140	2.410380	214	54

Table 3: Opcodes costs, stack and local variable operations-part 2/2

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
istore_0 0x3b	0.958800	2.298900	204	50
istore_1 0x3c	0.958800	2.298900	204	50
istore_2 0x3d	0.958800	2.298900	204	50
istore_3 0x3e	0.958800	2.298900	204	50
lstore_0 0x3f	1.086160	2.607960	230	55
lstore_1 0x40	1.086160	2.607960	230	55
lstore_2 0x41	1.086160	2.607960	230	55
lstore_3 0x42	1.086160	2.607960	230	55
astore_0 0x4b	0.958800	2.298900	204	50
astore_1 0x4c	0.958800	2.298900	204	50
astore_2 0x4d	0.958800	2.298900	204	50
astore_3 0x4e	0.958800	2.298900	204	50
pop 0x57	0.857440	2.037840	184	47
pop2 0x58	0.857440	2.037840	184	47
dup 0x59	0.928740	2.200260	198	50
dup_x1 0x5a	1.040200	2.451780	220	55
dup_x2 0x5b	1.119080	2.638200	236	59
dup2 0x5c	1.026160	2.434680	218	56
dup2_x1 0x5d	1.169000	2.751300	246	62
dup2_x2 0x5e	1.321140	3.100140	276	69
swap 0x5f	0.990280	2.338680	210	52
ldc 0x12	1.022440	2.458380	218	56
ldc_w 0x13	1.085880	2.617860	232	62
ldc2_w 0x14	1.203000	2.870700	254	65

Table 4: Opcodes costs, arithmetic opcodes

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
iadd 0x60	0.957860	2.273580	204	51
isub 0x64	0.957360	2.273580	204	51
imul 0x68	0.959500	2.273580	204	51
idiv 0x6c	1.613020	3.851460	348	84
ladd 0x61	1.575480	3.631800	328	76
lsub 0x62	1.575480	3.631800	328	76
lmul 0x69	1.638000	3.865920	348	74
ldiv 0x6d	3.685660	9.344040	820	181
iinc 0x84	1.188360	2.830920	252	63
ineg 0x74	0.920080	2.176260	196	49
lneg 0x75	1.366460	3.136320	286	67
irem 0x70	1.613020	3.851460	348	84
lrem 0x71	3.685660	9.344040	820	181

Table 5: Opcodes costs, object and arrays-Part1/2

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
new 0xbb (java.lang.Object)	5.456560	12.437760	1146	240
new (java.lang.String)	5.508280	12.561060	1158	240
putfield 0xb5	4.201320	9.604260	872	185
putfield (long)0xb5	4.432900	10.139520	918	196
getfield 0xb4	4.156160	9.515160	864	183
getfield 0xb4(long)	4.324340	9.912000	898	192
putstatic 0xb3	4.100420	9.381000	856	185
putstatic (long) 0xb3	4.334720	9.949800	904	195
getstatic 0xb2	4.083360	9.357000	852	184
getstatic (long)0xb2	4.303640	9.901800	898	193
checkcast 0xc0(is String 'castable' to Object,yes)	3.726300	8.497920	774	156
checkcast 0xc0 (is Class 'castable' to Object,yes)	3.726300	8.497920	774	156
checkcast 0xc0 (is SubClass 'castable' to Class,yes)	4.243800	9.677880	888	191
checkcast(is Class 'castable' to subClass,no throws ClassCastException)	133.521178	302.378599	28250	7058
checkcast(is Object 'castable' to String,0 no throws ClassCastException)	133.292438	301.846638	28200	7044
instanceof (is String instanceof Object,yes) 0xc1	3.815760	8.733660	792	160
instanceof (is Object instanceof String,no) 0xc1	4.213440	9.600299	880	186
instanceof (is This instance of Object,yes) 0xc1	3.815760	8.733660	792	160
instanceof (is Object instance of This,no) 0xc1	4.213440	9.600299	880	186
instanceof (is This instance of String,no) 0xc1	4.442180	10.132260	930	200
instanceof (is Class instance of subClass,no)	4.442180	10.132260	930	200
instanceof (is SubClass instance of Class,yes)	4.333260	9.913620	906	195
newarray 0xbc (of size 0)	4.417460	10.073760	924	192
newarray 0xbc (of 4 bytes)	4.434700	10.114860	928	192
newarray 0xbc (of 8 bytes)	4.451940	10.155960	932	192
newarray 0xbc (of 16 bytes)	4.486420	10.238160	940	192
newarray 0xbc (of 32 bytes)	4.555380	10.402560	956	192
newarray 0xbc (of 64 bytes)	4.770880	10.928640	1006	196

Table 6: Opcodes costs, object and arrays-Part2/2

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
anearray (size=0 non resolved empty class) 0xbd	34.638980	79.455499	7388	1908
anearray (size=1 non resolved empty class) 0xbd	34.656220	79.496599	7392	1908
anearray (size=5 non resolved empty class) 0xbd	34.725180	79.660999	7408	1908
anearray (size=0 resolved empty class)0xbd	32.7967	75.5584	6996	1803
anearray (size=1 resolved empty class)0xbd	32.8139	75.5995	7000	1803
anearray (size=5 resolved empty class)0xbd	32.8829	75.7639	7016	1803
anearray (size=0 resolved java.lang.Object class)	27.806520	64.268619	5936	1502
anearray (size=1 resolved java.lang.Object class)	27.823760	64.309719	5940	1502
anearray (size=5 resolved java.lang.Object class)	27.892720	64.474119	5956	1502
anearray (size=0 resolved java.lang.String class)	27.754860	64.073619	5928	1502
anearray (size=1 resolved java.lang.String class)	27.772100	64.114719	5932	1502
anearray (size=5 resolved java.lang.String class)	27.841060	64.279119	5948	1502
multiarray (int 1 dimension,size=0/dim.)0xc5	36.323520	83.483859	7700	1872
multiarray (int 2 dimensions,size=4/dim.)0xc5	49.866399	113.914719	10542	2450
multiarray (int 4 dimension,size=2)	84.130479	190.890879	17702	3934
multiarray (nonResolved 1 dimension,size=0)	36.323520	83.483859	7700	1872
multiarray (nonResolved 1 dimension,size=5)	36.409720	83.689359	7720	1872
multiarray (nonResolved 2 dimension,size=5)	53.042079	121.065519	11212	2580
multiarray (Object 1 dimensions,size=8)	36.461440	83.812659	7732	1872
multiarray (Object 2 dimensions,size=4)	49.866399	113.914719	10542	2450
multiarray (Object 4 dimensions,size=2)	84.130479	190.890879	17702	3934
arraylength 0xbe	0.983920	2.297580	210	52
baload 0x33	1.066680	2.554380	228	60
caload 0x34	1.084380	2.610900	232	62
saload 0x35	1.075780	2.586900	230	61
iaload 0x2e	1.077460	2.588220	230	60
laload 0x2f	1.183560	2.846640	252	67
aaload 0x32	1.077460	2.588220	230	60
bastore 0x54	1.155620	2.748520	246	64
castore 0x55	1.164480	2.787420	248	65
sastore 0x56	1.164480	2.787420	248	65
iastore 0x4f	1.158240	2.774640	246	64
lastore 0x50	1.310000	3.123480	276	71
aastore 0x53	2.003000	4.674360	418	100

Table 7: Opcodes costs, method invocation and return

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
invokevirtual+return(empty method)0xb6	12.032760	27.984120	2504	520
invokevirtual+ireturn(empty method)	12.194540	28.380960	2536	529
invokevirtual+lreturn(empty method)	12.363720	28.753800	2570	537
invokevirtual+areturn(empty method,return this)0xb6	12.159580	28.284960	2528	525
invokestatic+return(empty method)0xb8	10.549340	24.648600	2198	455
invokestatic+ireturn(empty method)	10.711120	25.045440	2230	464
invokestatic+lreturn(empty method)	10.880300	25.418280	2264	472
invokespecial+return(empty method)0xb7	10.512680	24.545700	2188	450
invokespecial+ireturn(empty method)	10.674460	24.942540	2220	459
invokespecial+lreturn(empty method)	10.843640	25.315380	2254	467
invokespecial+areturn(empty method,return this)	10.639500	24.846540	2212	455

Table 8: Opcodes costs, control flow

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
ifeq 0x99 (branch not taken)	0.957140	2.255160	206	54
ifeq 0x99 (branch taken)	1.142280	2.617860	246	62
ifne 0x9a (branch not taken)	0.957140	2.255160	206	54
ifne 0x9a (branch taken)	1.142280	2.617860	246	62
iflt 0x9b (branch not taken)	0.957140	2.255160	206	54
iflt 0x9b (branch taken)	1.142280	2.617860	246	62
ifle 0x9e (branch not taken)	0.957140	2.255160	206	54
ifle 0x9e (branch taken)	1.142280	2.617860	246	62
ifgt 0x9d (branch not taken)	0.957140	2.255160	206	54
ifgt 0x9d (branch taken)	1.142280	2.617860	246	62
ifge 0x9c (branch not taken)	0.957140	2.255160	206	54
ifge 0x9c (branch taken)	1.142280	2.617860	246	62
if_icmpeq 0x9f (branch not taken)	0.995660	2.352480	214	56
if_icmpeq 0x9f (branch taken)	1.180800	2.715180	254	64
if_icmpne 0xa0 (branch not taken)	0.995660	2.352480	214	56
if_icmpne 0xa0 (branch taken)	1.180800	2.715180	254	64
if_icmplt 0xa1 (branch not taken)	0.995660	2.352480	214	56
if_icmplt 0xa1 (branch taken)	1.180800	2.715180	254	64
if_icmple 0xa4 (branch not taken)	0.995660	2.352480	214	56
if_icmple 0xa4 (branch taken)	1.180800	2.715180	254	64
if_icmpgt 0xa3 (branch not taken)	0.995660	2.352480	214	56
if_icmpgt 0xa3 (branch taken)	1.180800	2.715180	254	64
if_icmpge 0xa2 (branch not taken)	0.995660	2.352480	214	56
if_icmpge 0xa2 (branch taken)	1.180800	2.715180	254	64
lcmp 0x94 (value1<value2)	1.483220	3.404280	310	72
lcmp 0x94 (value1==value2)	1.693000	3.845520	356	79
lcmp 0x94 (value1>value2)	1.693000	3.845520	356	79
ifnull 0xc6(branch not taken)	0.957140	2.255160	206	54
ifnull 0xc6 (branch taken)	1.124720	2.593860	242	61
ifnonnull 0xc7 (branch not taken)	0.957140	2.255160	206	54
ifnonnull 0xc7 (branch taken)	1.124720	2.593860	242	61
if_acmpeq 0xa5 (branch not taken)	0.995660	2.352480	214	56
if_acmpeq 0xa5 (branch taken)	1.180800	2.715180	254	64
if_acmpne 0xa6 (branch not taken)	0.995660	2.352480	214	56
if_acmpne 0xa6 (branch taken)	1.180800	2.715180	254	64
goto 0xa7	1.034380	2.400540	222	55
goto_w 0xc8	1.108140	2.551500	238	60
lookupswitch 0xab (1 iteration)	2.077300	4.759140	438	116
lookupswitch 0xab (2 iterations)	2.721600	6.228240	572	151
lookupswitch 0xab (3 iterations)	3.365900	7.697340	706	186
lookupswitch 0xab (4 iterations)	4.010200	9.166440	840	221
tableswitch 0xaa	1.793420	4.205460	384	108

Table 9: Opcodes costs, logic opcodes

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
ishl 0x78	0.976480	2.321580	208	53
ishr 0x7a	0.976360	2.321580	208	53
iushr 0x7c	0.976420	2.321580	208	53
lshl 0x79	1.419040	3.316500	300	72
lshr 0x7b	1.419040	3.316500	300	72
lushr 0x7d	1.545040	3.593340	324	76
iand 0x7e	0.957800	2.273580	204	51
ior 0x80	0.958100	2.273580	204	51
ixor 0x82	0.958120	2.273580	204	51
land 0x7f	1.127820	2.701320	240	63
lor 0x81	1.128420	2.701320	240	63
lxor 0x83	1.128460	2.701320	240	63

Paper II

Energy consumption analysis for two embedded Java virtual machines

Sébastien Lafond and Johan Lilius

Originally published in: *Journal of Systems Architecture*, volume 53, Issues 5-6, pages 328-337. Elsevier B.V., 2007.

Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 53 (2007) 328–337
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

Energy consumption analysis for two embedded Java virtual machines

Sébastien Lafond [a,*], Johan Lilius [b]

[a] Turku Centre for Computer Science, Embedded Systems Laboratory Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
[b] Åbo Akademi University, Department of Computer Science Lemminkäinengatan 14A, FIN-20520 Åbo, Finland

Received 9 October 2006; accepted 9 October 2006
Available online 20 November 2006

Abstract

In this paper we present a general framework for estimating the energy consumption of an embedded Java virtual machine (JVM). We have designed a number of experiments to find the constant overhead and establish an energy consumption cost for individual Java opcodes for two JVMs. The results show that there is a basic constant overhead for every Java program, and that a subset of Java opcodes have an almost constant energy cost. We also show that memory access is a crucial energy consumption component.

Keywords: Energy consumption; Embedded java virtual machine; Java opcode

1. Introduction

Several techniques have been developed to optimize the energy consumption of embedded systems. Those techniques can be hardware based solutions, as well as software or co-design solutions [1]. Techniques for low power optimization of software have been mostly applied on processor instructions level [2,3] by mainly using processor specific instructions [4,5]. Techniques on memory management have also been widely applied for optimizing energy consumption [6,7].

At the same time, the size and complexity of applications and development constraints like getting the product to market on time, make necessary the use of high-level languages. Due to the wide diversity of hardware and OS used in the world of handheld devices, portability across systems is not easy and needs efforts. Java language eases portability by allowing application developments with an abstraction of the target platform, making the concept "write once, run it anywhere" possible.

In this paper we present a general framework for estimating the energy consumption of an embedded Java virtual machine. We present a number of experiments to estimate the constant overhead of the JVMs energy consumption and established an energy consumption cost for individual Java opcodes.

The major contributions of this paper are a better understanding of the energy consumption distribution of an embedded Java virtual machine

* Corresponding author. Tel.: +358 2153328.
E-mail addresses: sebastien.lafond@abo.fi (S. Lafond), johan.lilius@abo.fi (J. Lilius).

1383-7621/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2006.10.003

(JVM) and the definition of the energy cost for the Java bytecodes for two different embedded JVMs.

The remainder of this paper is organized as follows. Section 2 presents the two JVMs used in this study, and proposes a methodology scheme used to characterize the energy consumption of an embedded JVM. Section 3 presents several experiments in order to define some constant overheads of the JVMs and comments the repartition of the JVMs energy consumption. Section 4 presents a measurement methodology used to define the energy cost of Java bytecode by cost comparison between two appropriate class files. Finally, Section 5 concludes the paper and suggests future possible work. This paper extends [8] with a result comparision between two embedded JVMs.

2. An energy consumption model of Java applications

The Java virtual machine is an abstract machine, making the interface between platform independent applications and the hardware, through a possible operating system. Thus the use of Java language can be seen as adding one more layer, the Java virtual machine, between the hardware and software layers. We want to study how well applying estimation techniques on the virtual machine opcodes level can be done, similarly to what has been done on processor instructions level. Fig. 1 shows a simple view of a JVM life cycle. An efficient energy model should characterize each stage of the life cycle model, and thus shows in which stage(s) effort needs to be concentrated to achieve energy optimization. It seems obvious that such model needs to consider the system's hardware and software configuration and therefore is not directly portable. But the methodology used to build it can easily be applied on different configurations by changing the platform configuration parameters. As shown in [9] the memory consumption must also be included in the model, as the memory might represent the biggest source of energy consumption on a typical embedded system. This is even more important to take into account as the JVM is a stack machine and will therefore have a relatively high memory activity.

2.1. Measurements methodology

We chose to use the Sun Microsystems K Virtual Machine (KVM), CLDC v1.0.3, and the simple Real-Time-Java (simpleRTJ) virtual machine. KVM is a small virtual machine containing about 50–80KB of object code in its standard configuration and has a total memory footprint in the range of 128–256KB. KVM can run on a 16-bit or 32-bit RISC/CISC processor clocked from 25 MHz. The simpleRTJ is a tiny JVM targeting 8/16/32 bit embedded systems and requiring on average about 18–24KB of code memory to run.

To build an energy model of the JVMs we adapted the energy profiler *enprofiler* [10] developed by the Embedded Systems Groups at Dortmund University. The adaptation was done in order to integrate the Java environment in the results provided by the energy profiler. With the adaptation, when summing up the energy cost for each instruction execution or memory access the *enprofiler* checks in which JVM stage the event occurred and increments the corresponding costs variable. Enprofiler is a processor instructions level energy profiler for ARM7TDMI processor cores operating in Thumb mode [11] and integrating the consumption of memory accesses. It has been built from physical measurements done on an Atmel AT91EB01 evaluation board consisting of a AT91M40400 processor clocked at 33 MHz and an external 512KB SRAM. A detailed description of the energy model used by *enprofiler* is given in [12]. According to [12] *enprofiler* shows a precision of 1.7% for the cost measurement of 12 instructions in an endless loop.

Fig. 2 shows the measurements methodology scheme used to characterize each stage of the JVM life cycle. The Java class generator generates, from template classes, Java applications with the possibility to modify parameters inside the class source code. With the Java code compact (JCC) we compile and link together the KVM source code and the generated Java application. For simpleRTJ the java application is pre-linked with all needed classes into a single binary image. The executable code is run on the ARM7TDMI emulator ARMulator, which traces instructions, memory accesses and

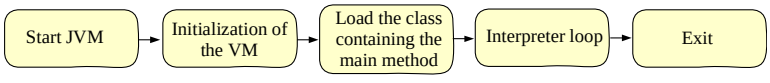


Fig. 1. Simple view of the JVM life cycle.

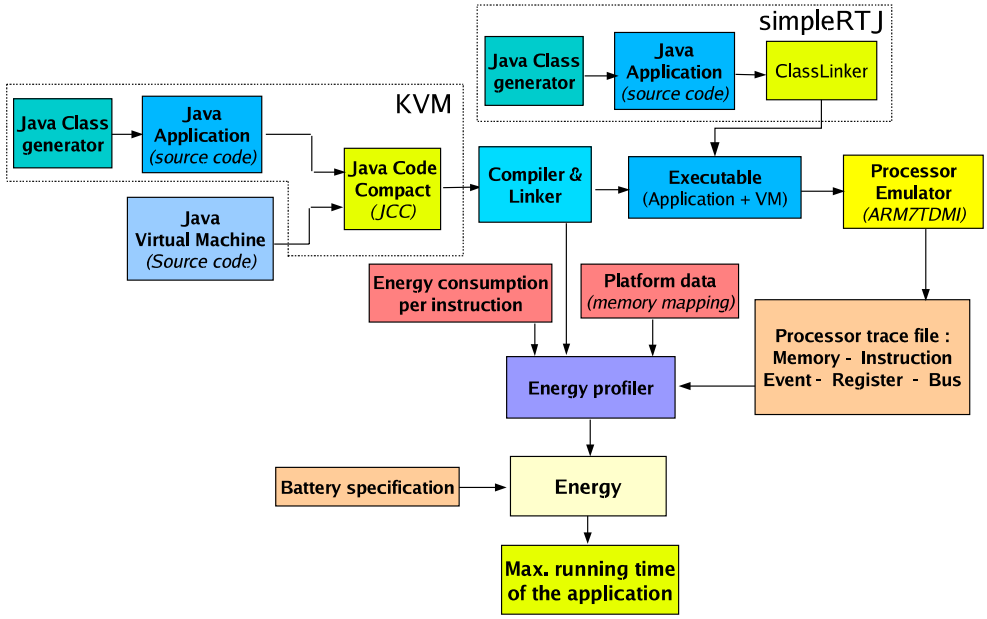


Fig. 2. Measurements methodology scheme.

events that occur during the application execution. From this trace, we extract all information concerning the memory access addresses, size and type (read, write, sequential, non-sequential), the instructions addresses and their corresponding processor opcodes. The energy profiler *enprofiler* reads the emulator trace and accesses databases providing processor instruction costs and the cost of a memory access depending of its address, size and type. The energy profiler estimates the energy consumed by the application and provides information on how the energy is distributed between the processor and memories for each JVM stage.

3. Experiments

We have run the measurement process over several representative benchmarks to characterize each stage of the JVMs life cycle and determine if some stages are dominant. We used as reference an empty application in order to reflect the JVMs basic costs. Dedicated intensive allocation applications was also used in order to study the behavior of the JVMs stage costs.

3.1. Benchmarks

Empty application: We run the empty application through the measurement process in order to find out if overhead constants in the JVMs energy consumption can be determined. Its source code is the following:

```
public class HelloWord {
 public  static  void  main(String
arg[])
  {
  //nothing to do
  }
}
```

Intensive allocation applications: Two intensive allocation applications were used in order to study a possible application related evolutions in the JVMs costs. The first application, called alloc1, instantiates inside a loop one object of class MyClass. This class does not contain any field and has just one *main* method. Each new class MyClass created by main is stored in the heap, and will contain only a reference to the class definitions area. Each instantiation will create a new stack frame and call the MyClass constructor which by default will only call java/lang/Object constructor method. The stack frame created by the main method contains two operand stacks and three local variables to store the object reference, the length and the loop index. This application is used to observe the evolution of different KVM stage costs with the length of the loop. The source code for alloc1 is the following:

```
public class MyClass {
  public static void main(String
arg[])
  {
   int length = X;
   for(int i=0; i<=length ; i++) {
   new Myclass();
   }
  }
}
```

The second intensive allocation application, called alloc2, is similar to the precedent one with the difference that MyClass contain one field define by an integer array of size 500. Alloc2 is used to observe the weight that can take the garbage collector in comparison to the other JVMs stages in extreme allocation rate. As each new instance takes approximately 2KB, with an heap size of 128KB the garbage collector needs to be triggered every 64th objects created in the loop to reclaim the heap space occupied by the unreferenced objects. The source code for alloc2 is the following:

```
public class MyClass {
int[] tab = new int[500];
  public static void main(String
arg[])
  {
   int length = X ;
   for(int i=0; i<=length ; i++) {
   new Myclass();
   }
  }
}
```

3.2. Results

This section presents the results obtained by the introduced applications through the measurement process.

Table 2
Empty application–energy consumption of simpleRTJ stages in μJ

StartJVM Inst.	StartJVM Mem.	Interpr. Inst.	Interpr. Mem.
10601.52	23905.32	1599.04	3866.66

Empty application: The empty application has been used in order to find out if overhead constants in the JVMs energy consumption can be determined.

Tables 1 and 2 show the obtained results for respectively KVM and simpleRTJ. Figs. 3 and 4 present the energy consumption distribution among respectively all KVM and simpleRTJ stages. The distribution between the energy consumed by memory accesses and processor instruction execution is also presented on these figures.

For the simpleRTJ virtual machine we defined only one initialization stage and any cleanup or post interpreter stage. This is because (a) from its code implementation it is logical to keep its initialization stage StartJVM as a single block, and (b) the simpleRTJ exit almost immediately after last opcode is executed.

From Figs. 3 and 4 we can already notice that the energy distribution between memory access and instruction execution is similar between the two JVM. The second observation is the difference of weight the interpreter takes for executing the empty application. As we will see later, this difference can be explained by the fact that the simpleRTJ implementation implies more expensive heap allocations than the KVM implementation. This reduce the simpleRTJ interpreter weight compare to the StartJVM stage weight.

As the application was '*empty*' the values in Table 1 represent the virtual machines basic costs or the minimal overhead energy cost that any application will have to dissipate.

Intensive allocation applications: From the alloc1 results in Figs. 5 and 6 we note that only the energy consumed by the interpreter is dependent on the

Table 1
Empty application–energy consumption of KVM's stages in μJ

StartJVM Inst.	StartJVM Mem.	KVMStart Inst.	KVMStart Mem.	Interpr. Inst.	Interpr. Mem.
9.42	20.08	748.81	1639.18	3552.28	8273.34
		KVM Clean Inst.	KVM Clean Mem.		
		144.92	326.38		

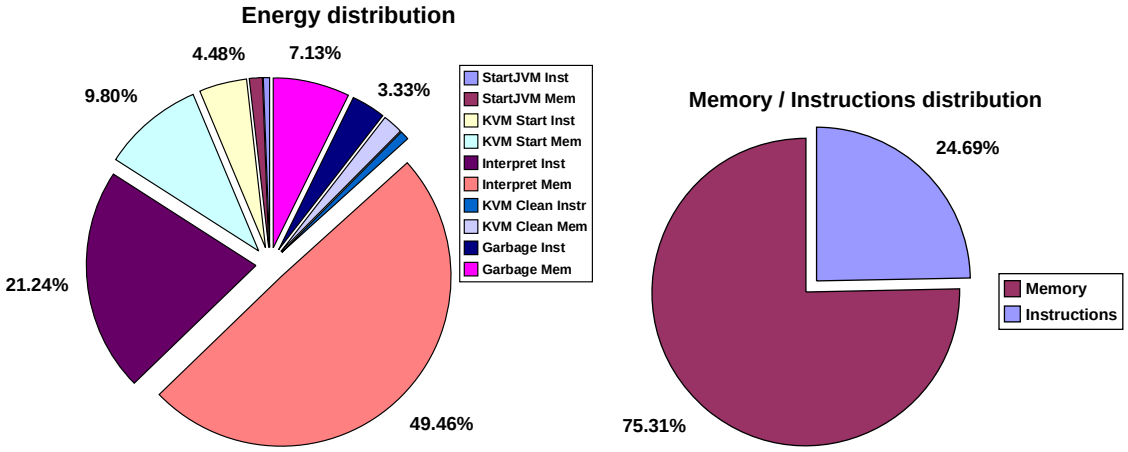


Fig. 3. Empty application–KVM energy consumption distributions.

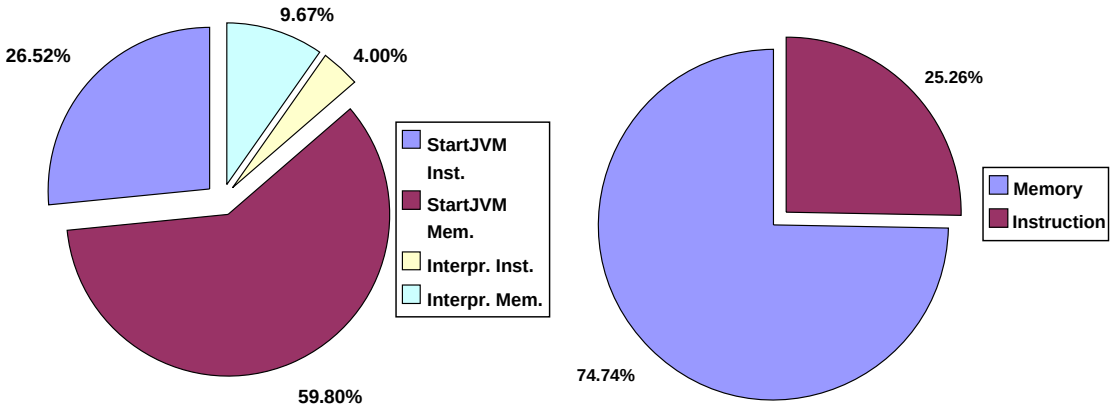


Fig. 4. Empty application–simpleRTJ energy consumption distributions.

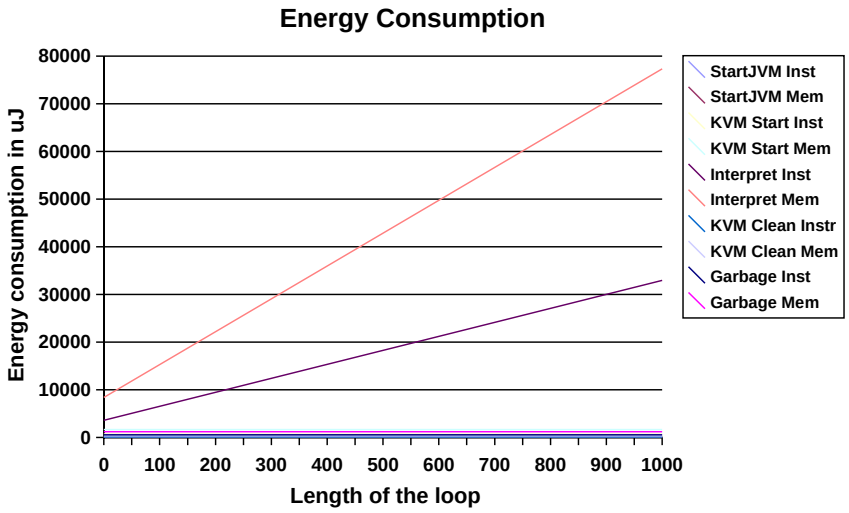


Fig. 5. Alloc1–KVM's stages energy consumption depending of the loop length.

loop length value. All other stages of the JVMs consume a constant energy including the garbage collector, as the maximum number of created object was not enough to fill up the Java heap and trigger off a garbage collection. It is important to notice that the evolution of the interpreter stage energy

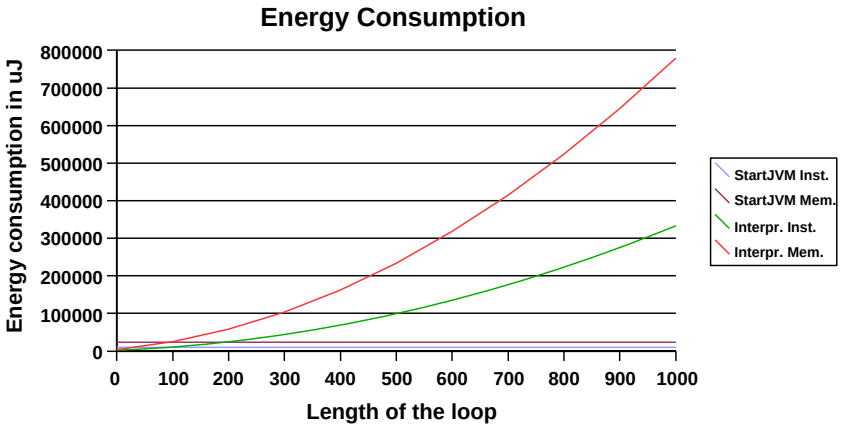


Fig. 6. Alloc1–simpleRTJ's stages energy consumption depending of the loop length.

consumption with the loop length is different between KVM and the simpleRTJ. For KVM the interpreter stage cost is linear and proportional to the loop length, whereas simpleRTJ interpreter stage cost is exponential to the length of the loop. This difference can be explain by looking at each JVM implementation for allocating new object into the heap. KVM uses a list of free memory chunk available in the heap. For each new allocation it traverses the list until it finds a free chunk enough big to hold the new object. In our case it will always find the first chunk available to store the new *MyClass* object, thus executing each time the same successive instructions. As a consequence the KVM interpreter stage cost will be linear and proportional to the loop length. SimpleRTJ uses only a list of object allocated in the heap. Each object contains a flag telling if the object is actually free space or not. For each new allocation simpleRTJ is traversing the list to find possible object having his flag set and enough big to hold the new object. If none is found simpleRTJ will allocate a new memory bloc. In our case for each new *MyClass* object allocation simpleRTJ will first probe all already allocated *MyClass* objects before allocating a new memory bloc, thus executing each time a exponential number of instructions. As a consequence simpleRTJ interpreter stage cost will be exponential to the length of the loop.

The energy distributions for a loop length of 1000 presented in Figs. 7 and 8, are similar to the first experiment with an interpreter stage even more dominant, representing over 95% of the total energy consumed.

Alloc2 application was used to observe the garbage collector weight in comparison to other JVMs stages. Several factors can influence the garbage collection behavior and thus its energy consumption: the size of the heap, the sizes and numbers of live or dead objects,heap fragmentation and naturally the technique used to implement it. Both JVMs use a mark and sweep garbage collection algorithm, with the difference that KVM implementation tries

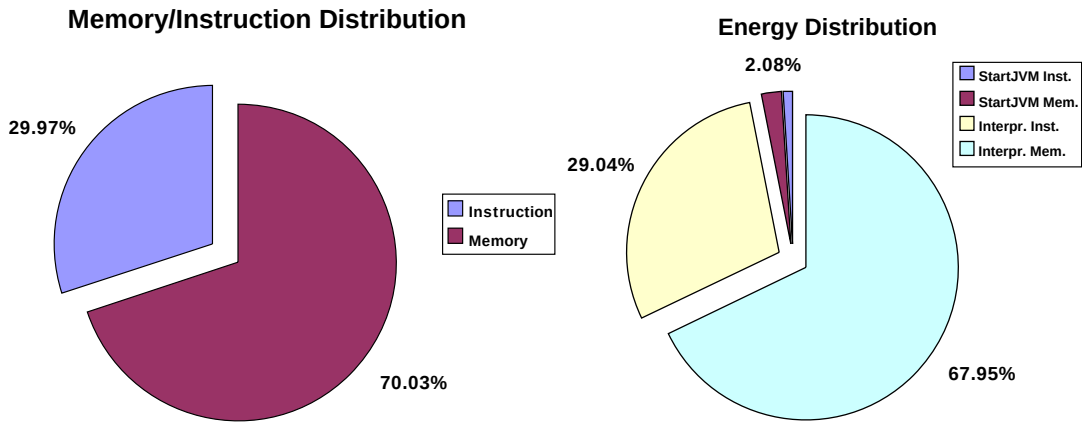


Fig. 7. Alloc1–RTJ energy distribution for loop length equal to 1000.

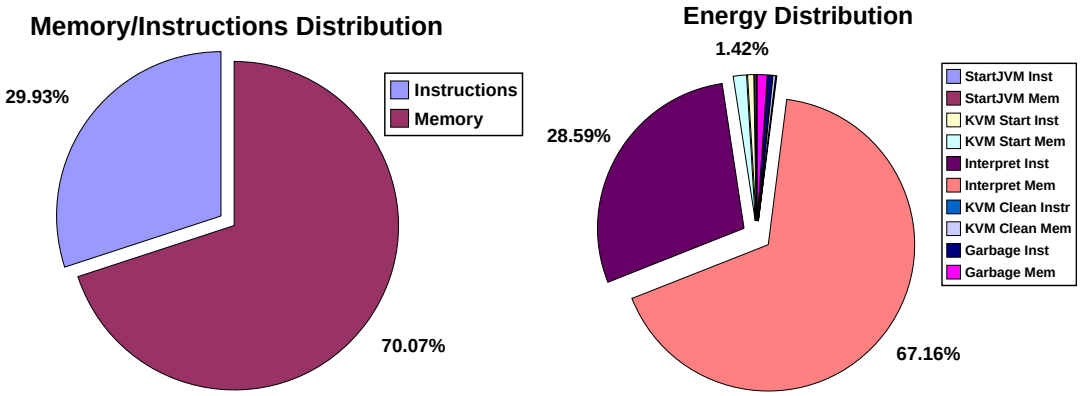


Fig. 8. Alloc1–KVM energy distribution for loop length equal to 1000.

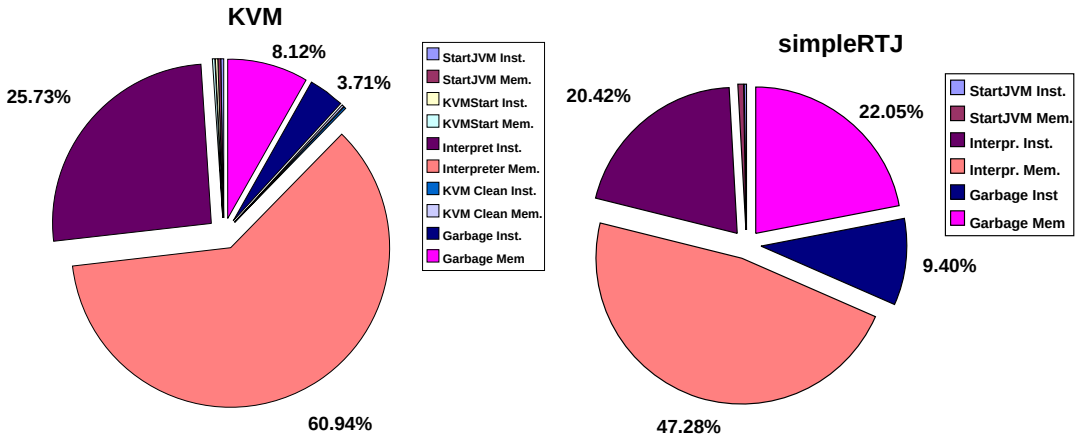


Fig. 9. Alloc2–Garbage collection weight for KVM and simpleRTJ.

to do all its work in a single pass without any recursive calls.

Fig. 9 presents the garbage collector weight for a loop length of 1000. We can observe that for an very intensive allocation rate of *dead objects* the KVM GC energy consumption represents only 10% of the total KVM energy consumption. On the other hand with the same application and parameter the simpleRTJ GC will represent almost one third of the total simpleRTJ energy consumption. This major difference is coming from the implementation variance between the JVM garbage collections.

We also run the measurement process with simpleRTJ over the all representative benchmarks presented in [8], and have the same following observation than in [8]: from all experiments done it is clear that the interpreter stage is far ahead the main source of energy consumption. Thus a better comprehension of it is needed if someone wants to achieve energy optimization on the JVMs.

As the interpreter reads and executes the Java bytecode, having a closer view on the interpreter implies increasing the granularity of its energy consumption model by looking at the cost of each Java opcode interpreted.

4. Java opcode energy cost

In order to get a better understanding of the interpreter energy consumption, an evaluation of each Java opcode energy cost is needed. As a strict class file structure needs to be respected, it is not possible to only execute one Java opcode. Thus a cost comparison between two class files is needed to estimate the cost difference between them. The general measurements methodology scheme used to characterize each JVMs stage life cycle can be re-used with different inputs. Instead of using Java source code files we will use as input appropriate byte-code executable class files.

4.1. Measurements methodology

Fig. 10 shows an abstract view of the class files generator used to create two class files, named ClassFile and ClassFile_Ref. The opcode behavior towards the Java operand stack and the local variables array has to be defined for each studied Java opcode, i.e. provide the operand stack state needed before and resulting after the studied opcode execution as well as the number of local variables needed.

To ensure the estimation quality for each opcode we generate several pairs of class files executing the studied opcode and also monitor the possible energy consumption differences between all other JVMs stages.

4.2. Results

From all Java opcodes we will not study the 51 opcodes which handle floating point values as floating point is not supported by the CLDC specification. In addition as the simpleRTJ VM does not support long type, all opcodes manipulating longs are only analysed for the KVM. The opcode *athrow* was also omitted from this study, it is not possible to directly estimate its energy cost using this comparison method as its cost can not be extracted from the context cost. All the same, in Table 5 in [13] we can see from the opcode *checkcast* the cost of throwing an *ClassCastExeption* exception and exiting the KVM.

As a general observation we can say that for most opcodes simpleRTJ gives a more expensive implementation in terms of energy and number of cycles than KVM. However the cost differences between opcode functional groups within each virtual machine are similar. Due to space requirement all obtained values for each studied opcode and each JVM are published in [13], where the opcodes are divided in six functional groups:

Stack and local variable operations opcodes: Tables 2 and 3 in [13] show the results concerning opcodes that operate on the operand stack and local variable. We can notice that loading a value from the local variables array to the operand stack is lightly more expensive than storing the same value back to the local variable. It is also interesting to note that for KVM the opcode *bipush* consumes about 9% less energy than *iload* and 5% less than *ilaod_x*. Thus it is more energy efficient to load an constant integer lower than 256 into the operand stack using *bipush* than initializing the local variable array with the constant and use *iload* or *ilaod_x*. The same is true if a constant integer lower than 65,536 has to be loaded into the operand stack, it will be more efficient to use the opcode *bipush* instead of *iload*. But in case the integer constant can be stored in the first 4 local variables then *iload_x* becomes the most efficient opcode.

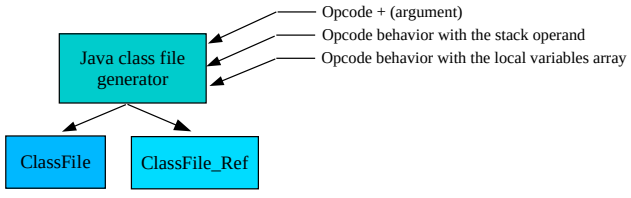


Fig. 10. Bytecode executable class file generator.

Type conversion opcodes: Table 1 in [13] shows the results for opcodes that convert value from one primitive type to another. The costs are in the same range as the stack and local variable operations opcodes as the conversion opcodes pop a value from the stack, perform a right shift or truncate the popped value and push back the result.

Arithmetic opcodes: Table 4 in [13] shows the costs for arithmetic opcodes. As it was easy to predict, the cost of an arithmetic operation is dependent on the type of the operands and the operation. For the KVM operations on long types are about 50% more expensive than on integers, except for the division of types long which is about two times more expensive than to divide integers.

Logic opcodes: As for the arithmetic opcodes, the cost of logic opcodes is also depending of the type of the operand and for the KVM operations on longs are from 23% to 37% more expensive than operation on integers. Table 9 in [13] shows the costs for logic opcodes.

Control flow opcodes: The control flow opcodes are the opcodes that implement the following Java language statements: *do-while*, *while*, *if*, *if-else*, *for* and *switch*. Table 8 in [13] shows the cost for the 25 control flow opcodes. For all conditional *if* opcodes (i.e. opcodes from 0x99 to 0xa6 and *ifnull*, *ifnonnull*) the energy cost depends on a two values comparison success. If the comparison success the VM jumps to a target defined by the opcode operands, in the other case the VM continues by executing the following opcodes.

The *tableswitch* opcode performs the same task as *lookupswitch*, with the difference that it requires a consecutive list of case values contained between

one low and high endpoint. Thus the VM knows in advance the position of all case values so that the retrieving cost is the same for all cases. Compared with *lookupswitch*, *tableswitch* has a lower energy cost but generates all the more bigger class file size as the gape between the case values is great.

Objects and arrays opcodes: Tables 5 and 6 in [13] show the cost of opcodes that create and manipulate arrays and objects. The creation cost, with *newarray*, of a single dimension array of primitive type integer, long, short, byte, char or boolean is not directly dependent on array type and size, but more on the memory size that needs to be allocated for its creation. That means that the creation cost is identical for an integers array of size 8, a shorts array of size 16, or for the KVM a longs array of size 4. The creation cost, with *multiarray*, of multidimensional arrays is dependent on the array dimensions and dimensions indexes values. Each dimension adds a basic cost to the array creation cost, thus creating a $2 * 2 * 2$ integers array will be 70% more expensive than creating a $2 * 4$ integers array, and especially 18 times more expensive than creating a single dimension integers array of size 8.

The objects creation cost depends on the objects themselves, i.e. on the type and size of their constant pool, interfaces, fields, methods and their superclasses, and also on their resolution flags inside each class constant pool. A new object is resolved only once within a same class, and its address is stored in the constant pool structure of the class. Table 5 in [13] shows as an example the creation cost of an object of type *java.lang.Object* and *java.lang.String*. In addition, Table 5 in [13] refers to two objects called *Class* and *subClass* which is a empty (none interface, field nor method) sub class of *nonResolvedClass* itself empty sub class of *java.lang.Object*.

Method invocation and return opcodes: Because invoking a method implies returning from it at some point, Table 7 in [13] shows the costs of different invoke/return pairs. They all invoke an empty 'already resolved' method within the same class or instance. We can notice from this table that calling a static, public or private method costs almost the same, and that the type of the returned value has not a great influence on the overall cost.

It is also important to compare all obtained values with the NOP energy consumption. As the opcode NOP is the first case statement in the interpreter switch and does not execute any instruction, its energy consumption represents the minimum overhead cost due to the interpreter mechanism. For the most of the stack and local variable operation opcodes the interpreter mechanism overhead represents about 70% of their energy consumption.

4.3. Opcode costs verification

In order to verify the obtained opcode costs we calculated for each benchmark execution used for the first experiments the value $\sum(Opcode\ cost * Opcode\ Occurrence)$. The computed value was then compared with the cost given by the energy profiler for the interpreter stage. KVM has a build-in implementation to trace all executed opcodes. We also added such feature to the simpleRTJ VM in order to calculate the occurrence of each opcode. For control flow opcodes we checked if the branch was taken or not to attribute the correct opcode cost, but to keep the verification simple we didn't looked at the type of variable handled by *putfield*, *getfield*, *putstatic* and *getstatic*. There respective cost for handling integer was used for all occurrences. In addition for all other none static opcode costs only the respective basic cost was used. The benchmark *Exception* from the Java Grande Forum Benchmark Suite was not used as we did not studied the cost for the opcode *athrow*.

Table 3 presents the normalized verification results where the value 100 represent for each benchmark the energy cost given by the energy profiler for the interpreter stage. For each benchmark the accuracy obtained by calculating the value $\sum(Opcode\ cost * Opcode\ Occurrence)$ is staying between -5% and $+10\%$ of the cost given by the energy profiler. But this loss in precision has to be balanced with the time needed to compute it. It takes only few seconds to calculate the occurrence for each opcode and compute the value $\sum(Opcode$

Table 3
Verification results

	Dhrystone50	Arith	Assign	Loop	Create	Method	Math	Generic
KVM	103.99	105.31	95.55	100.30	97.95	102.51	96.74	109.43
simpleRTJ	102.35	101.56	98.75	102.28	100.15	103.12	98.95	103.34

*cost * Opcode Occurrence*), compared to several hours needed by the energy profiler.

5. Conclusion

Several observations have been done in this paper concerning the energy consumption of the JVMs. For the hardware configuration fixed by the energy profiler, the distribution between the processor and memories is constant over the JVMs execution with 70% of the energy consumed by memory accesses. This shows the major importance of the memories for embedded system runtime performance. We also showed that implementation differences between two embedded JVMs can imply great divergence concerning the JVM energy consumption.

This paper can also guide developers to produce energy-aware java application by limiting the use of long data type, avoiding multidimensional array and trying to use consecutive case values inside a switch statement. Furthermore, the opcodes energy cost can be helpful for developing a energy-aware Java compiler as well as optimizing the JVMs by pointing out the expensive opcodes. This paper shows the first steps toward an energy aware performance analysis tool for Java application, as a such tool would ask for a more detailed model for a subset of opcodes.

Also as the interpreter mechanism overhead cost is a predominant factor in opcode execution cost, it will be interesting in the future to look at the cost differences between the two possible Java execution modes: interpreted or JIT compilation. JIT compilation increases significantly the execution speed, but in the same time increases memory footprint. A trade-off between execution time and memory footprint size will certainly have to be found to reach the optimum optimization point for energy consumption.

References

[1] F. Parain, M. Banatre, G. Cabillic, T. Higuera, V. Issarny, J.-P. Lesot, Techniques de reduction de la consommation dans les systemes embarques temps-reel, Technical Report, INRIA Rennes, 2000.
[2] Vivek Tiwari, Sharad Malik, Andrew Wolfe, Power analysis of embedded software, in: International Conference on Computer-Aided Design, San Jose, CA, November, 1994.
[3] Anil Seth, Ravindra B. Keskar, R. Venugopal, Algorithms for energy optimization using processor instructions, in: International conference on Compilers, Architecture, and Synthesis for Embedded Systems, Atlanta, GA, USA, 2001.
[4] Wen-Tsong Shiue, Retargetable compilation for low power. Technical Report, Silicon Metrics Corporation, 2001.
[5] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, Power analysis and low-power scheduling, in: International Symposium on System Synthesis, September, 1995.
[6] Catherine H. Gebotys, Low energy memory and register allocation using network flow, in: Design Automation Conference, June, 1997, pp. 435–440.
[7] X. Fan, C. Ellis, A. Lebeck, Memory controller policies for DRAM power management, in: International Symposium on Low Power Electronics and Design (ISLPED), August, 2001.
[8] Sébastien Lafond, Johan Lilius, An energy consumption model for an embedded java virtual machine, in: ARCS, 2006, pp. 311–325.
[9] Kaushik Roy, Mark C. Johnson, Software design for low power, in: Low Power Design in Deep Submicron Electronics, 1997, pp. 433–460.
[10] Enprofiler. <http://ls12-www.cs.uni-dortmund.de/research/encc/>.
[11] An introduction to thumb. Technical Report, Advanced RISC Machines Ltd, 1995.
[12] Stefan Steinke, Markus Knauer, Lars Wehmeyer, Peter Marwedel, An accurate fine grain instruction-level energy model supporting software optimization, in: PATMOS 01, 2001.
[13] <http://www.abo.fi/slafond/javacosts>.

Appendix to paper 2

Table 1: Opcodes costs, conversion opcodes

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
i2l 0x85	0.928660	2.200260	198	50
l2i 0x88	0.857440	2.037840	184	47
i2b 0x91	0.928540	2.200260	198	50
i2b 0x91	1.271270	3.209640	268	65
i2c 0x92	0.928520	2.200260	198	50
i2c 0x92	1.113730	2.763480	234	56
i2s 0x93	0.928540	2.200260	198	50
i2s 0x93	1.271270	3.209640	268	65

Table 2: Opcodes costs, stack and local variable operations-part 1/2

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
nop 0x0	0.831440	1.989840	178	45
nop 0x0	0.715870	1.788300	152	39
aconst_null 0x1	0.890020	2.126940	190	49
aconst_null 0x1	1.015260	2.543520	214	53
iconst_m1 0x2	0.899160	2.150940	192	50
iconst_m1 0x2	1.024400	2.567520	216	54
iconst_0 0x3	0.890020	2.126940	190	49
iconst_0 0x3	1.015260	2.543520	214	53
iconst_1 0x4	0.890020	2.126940	190	49
iconst_1 0x4	1.015260	2.543520	214	53
iconst_2 0x5	0.890020	2.126940	190	49
iconst_2 0x5	1.015260	2.543520	214	53
iconst_3 0x6	0.890020	2.126940	190	49
const_3 0x6	1.015260	2.543520	214	53
iconst_4 0x7	0.889760	2.126940	190	49
iconst_4 0x7	1.015260	2.543520	214	53
iconst_5 0x8	0.890020	2.126940	190	49
iconst_5 0x8	1.015260	2.543520	214	53
lconst_0 0x9	0.922300	2.192040	196	50
lconst_1 0xa	0.930960	2.216040	198	51
bipush 0x10	0.926900	2.214420	198	52
bipush 0x10	1.231310	3.102480	260	64
sipush 0x11	0.990360	2.373900	212	58
sipush 0x11	1.382090	3.483240	292	71
iload 0x15	1.013700	2.434380	216	55
iload 0x15	1.266740	3.160020	266	64
lload 0x16	1.167820	2.815440	248	63
aload 0x19	1.013700	2.434380	216	55
aload 0x19	1.266740	3.160020	266	64
iload_0 0x1a	0.968120	2.322900	206	51
iload_0 0x1a	1.149280	2.884800	242	59
iload_1 0x1b	0.968120	2.322900	206	51
iload_1 0x1b	1.149280	2.884800	242	59
iload_2 0x1c	0.968120	2.322900	206	51
iload_2 0x1c	1.149280	2.884800	242	59
iload_3 0x1d	0.968120	2.322900	206	51
iload_3 0x1d	1.149280	2.884800	242	59
lload_0 0x1e	1.104800	2.655960	234	57
lload_1 0x1f	1.104800	2.655960	234	57
lload_2 0x20	1.104800	2.655960	234	57
lload_3 0x21	1.104800	2.655960	234	57
aload_0 0x2a	0.968120	2.322900	206	51
aload_0 0x2a	1.149280	2.884800	242	59
aload_1 0x2b	0.968120	2.322900	206	51
aload_1 0x2b	1.149280	2.884800	242	59
aload_2 0x2c	0.968120	2.322900	206	51
aload_2 0x2c	1.149280	2.884800	242	59
aload_3 0x2d	0.968120	2.322900	206	51
aload_3 0x2d	1.149280	2.884800	242	59
istore 0x36	1.004140	2.410380	214	54
istore 0x36	1.248000	3.110700	262	64
lstore 0x37	1.148940	2.767440	244	61
astore 0x3a	1.004140	2.410380	214	54
astore 0x3a	1.248000	3.110700	262	64

Table 3: Opcodes costs, stack and local variable operations-part 2/2

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
istore_0 0x3b	0.958800	2.298900	204	50
istore_0 0x3b	1.130540	2.835480	238	59
istore_1 0x3c	0.958800	2.298900	204	50
istore_1 0x3c	1.130540	2.835480	238	59
istore_2 0x3d	0.958800	2.298900	204	50
istore_2 0x3d	1.130540	2.835480	238	59
istore_3 0x3e	0.958800	2.298900	204	50
istore_3 0x3e	1.130540	2.835480	238	59
lstore_0 0x3f	1.086160	2.607960	230	55
lstore_1 0x40	1.086160	2.607960	230	55
lstore_2 0x41	1.086160	2.607960	230	55
lstore_3 0x42	1.086160	2.607960	230	55
astore_0 0x4b	0.958800	2.298900	204	50
astore_0 0x4b	1.130540	2.835480	238	59
astore_1 0x4c	0.958800	2.298900	204	50
astore_1 0x4c	1.130540	2.835480	238	59
astore_2 0x4d	0.958800	2.298900	204	50
astore_2 0x4d	1.130540	2.835480	238	59
astore_3 0x4e	0.958800	2.298900	204	50
astore_3 0x4e	1.130540	2.835480	238	59
pop 0x57	0.857440	2.037840	184	47
pop 0x57	0.840530	2.097360	178	44
pop2 0x58	0.857440	2.037840	184	47
pop2 0x58	0.840530	2.097360	178	44
dup 0x59	0.928740	2.200260	198	50
dup 0x59	1.139510	2.803260	238	58
dup_x1 0x5a	1.040200	2.451780	220	55
dup_x1 0x5a	1.447910	3.578880	302	66
dup_x2 0x5b	1.119080	2.638200	236	59
dup_x2 0x5b	1.623770	4.010580	338	72
dup2 0x5c	1.026160	2.434680	218	56
dup2 0x5c	1.272050	3.147180	266	60
dup2_x1 0x5d	1.169000	2.751300	246	62
dup2_x1 0x5d	1.799630	4.442280	374	78
dup2_x2 0x5e	1.321140	3.100140	276	69
dup2_x2 0x5e	1.975490	4.873980	410	84
swap 0x5f	0.990280	2.338680	210	52
swap 0x5f	1.253810	3.097860	262	60
ldc 0x12	1.022440	2.458380	218	56
ldc 0x12	1.794480	4.742460	378	95
ldc_w 0x13	1.085880	2.617860	232	62
ldc_w 0x13	1.878520	4.527310	401	107
ldc2_w 0x14	1.203000	2.870700	254	65

Table 4: Opcodes costs, arithmetic opcodes

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
iadd 0x60	0.957860	2.273580	204	51
iadd 0x60	1.191570	2.959440	250	58
isub 0x64	0.957360	2.273580	204	51
isub 0x64	1.191070	2.959440	250	58
imul 0x68	0.959500	2.273580	204	51
imul 0x68	1.193210	2.959440	250	58
idiv 0x6c	1.613020	3.851460	348	84
idiv 0x6c	1.779930	4.563960	382	88
ladd 0x61	1.575480	3.631800	328	76
lsub 0x62	1.575480	3.631800	328	76
lmul 0x69	1.638000	3.865920	348	74
ldiv 0x6d	3.685660	9.344040	820	181
iinc 0x84	1.188360	2.830920	252	63
iinc 0x84	1.348330	3.459240	284	69
ineg 0x74	0.920080	2.176260	196	49
ineg 0x74	1.105290	2.739480	232	55
lneg 0x75	1.366460	3.136320	286	67
irem 0x70	1.613020	3.851460	348	84
irem 0x70	1.779930	4.563960	382	88
lrem 0x71	3.685660	9.344040	820	181

Table 5: Opcodes costs, object and arrays-Part1/2

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
new 0xbb (java.lang.Object)	5.456560	12.437760	1146	240
new 0xbb (java.lang.Object)	24.695630	58.787820	5366	1492
new (java.lang.String)	5.508280	12.561060	1158	240
new (java.lang.String)	24.695630	58.787820	5366	1492
putfield 0xb5	4.201320	9.604260	872	185
putfield 0xb5	2.249470	5.935980	474	116
putfield (long)0xb5	4.432900	10.139520	918	196
getfield 0xb4	4.156160	9.515160	864	183
getfield 0xb4	2.249970	5.935980	474	116
getfield 0xb4(long)	4.324340	9.912000	898	192
putstatic 0xb3	4.100420	9.381000	856	185
putstatic 0xb3	2.271460	5.267030	475	117
putstatic (long) 0xb3	4.334720	9.949800	904	195
getstatic 0xb2	4.083360	9.357000	852	184
getstatic 0xb2	2.243350	5.143060	468	116
getstatic (long)0xb2	4.303640	9.901800	898	193
checkcast 0xc0(is String 'castable' to Object,yes)	3.726300	8.497920	774	156
checkcast 0xc0(is String 'castable' to Object,yes)	2.614710	7.184040	554	141
instanceof (is String instanceof Object,yes) 0xc1	3.815760	8.733660	792	160
instanceof (is String instanceof Object,yes) 0xc1	2.669290	7.323780	564	141
instanceof (is Object instanceof String,no) 0xc1	4.213440	9.600299	880	186
instanceof (is Object instanceof String,no) 0xc1	2.527090	6.958500	534	134
instanceof (is This instance of Object,yes) 0xc1	3.815760	8.733660	792	160
nstanceof (is This instance of Object,yes) 0xc1	2.669290	7.323780	564	141
instanceof (is Object instance of This,no) 0xc1	4.213440	9.600299	880	186
instanceof (is Object instance of This,no) 0xc1	2.527090	6.958500	534	134
instanceof (is This instance of String,no) 0xc1	4.442180	10.132260	930	200
nstanceof (is This instance of String,no) 0xc1	2.776750	7.541100	588	148
newarray 0xbc (of size 0)	4.417460	10.073760	924	192
newarray 0xbc (of size 0)	5.512310	12.582180	1155	240
newarray 0xbc (of 4 bytes)	4.434700	10.114860	928	192
newarray 0xbc (of 4 bytes)	5.540130	12.642530	1160	241
newarray 0xbc (of 8 bytes)	4.451940	10.155960	932	192
newarray 0xbc (of 8 bytes)	5.562220	13.200130	1165	241
newarray 0xbc (of 16 bytes)	4.486420	10.238160	940	192
newarray 0xbc (of 16 bytes)	5.608090	12.797520	1175	241
newarray 0xbc (of 32 bytes)	4.555380	10.402560	956	192
newarray 0xbc (of 32 bytes)	5.683420	13.003740	1195	241
newarray 0xbc (of 64 bytes)	4.770880	10.928640	1006	196
newarray 0xbc (of 64 bytes)	5.962520	13.660750	1258	245

Table 6: Opcodes costs, object and arrays-Part2/2

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
anearray (size=0 non resolved empty class) 0xbd	34.638980	79.455499	7388	1908
anearray (size=0 non resolved empty class) 0xbd	43.282320	99.318390	9235	2385
anearray (size=1 non resolved empty class) 0xbd	34.656220	79.496599	7392	1908
anearray (size=1 non resolved empty class) 0xbd	43.320250	99.370710	9240	2385
anearray (size=5 non resolved empty class) 0xbd	34.725180	79.660999	7408	1908
anearray (size=5 non resolved empty class) 0xbd	43.412170	99.573280	9260	2385
anearray (size=0 resolved empty class)0xbd	32.7967	75.5584	6996	1803
anearray (size=0 resolved empty class)0xbd	40.997809	94.447520	8745	2253
anearray (size=1 resolved empty class)0xbd	32.8139	75.5995	7000	1803
anearray (size=1 resolved empty class)0xbd	40.012300	94.498750	8750	2253
anearray (size=5 resolved empty class)0xbd	32.8829	75.7639	7016	1803
anearray (size=5 resolved empty class)0xbd	40.343290	94.663400	8770	2253
anearray (size=0 resolved java.lang.Object class)	27.806520	64.268619	5936	1502
anearray (size=0 resolved java.lang.Object class)	34.757540	80.325210	5936	1877
anearray (size=1 resolved java.lang.Object class)	27.823760	64.309719	7420	1502
anearray (size=1 resolved java.lang.Object class)	34.779700	80.387109	9275	1877
anearray (size=5 resolved java.lang.Object class)	27.892720	64.474119	5956	1502
anearray (size=5 resolved java.lang.Object class)	34.865940	80.592640	7445	1877
anearray (size=0 resolved java.lang.String class)	27.754860	64.073619	5928	1502
anearray (size=0 resolved java.lang.String class)	34.693560	80.092020	7410	1877
anearray (size=1 resolved java.lang.String class)	27.772100	64.114719	5932	1502
anearray (size=1 resolved java.lang.String class)	34.715120	80.143330	7415	1877
anearray (size=5 resolved java.lang.String class)	27.841060	64.279119	5948	1502
anearray (size=5 resolved java.lang.String class)	34.801250	80.348759	7435	1877
multiarray (int 1 dimension,size=0/dim.)0xc5	36.323520	83.483859	7700	1872
multiarray (int 1 dimension,size=0/dim.)0xc5	40.404370	104.354750	9625	2340
multiarray (int 2 dimensions,size=4/dim.)0xc5	49.866399	113.914719	10542	2450
multiarray (int 2 dimensions,size=4/dim.)0xc5	62.332990	142.393239	13177	3062
multiarray (int 4 dimension,size=2)	84.130479	190.890879	17702	3934
multiarray (int 4 dimension,size=2)	104.320275	238.613520	22127	4917
multiarray (nonResolved 1 dimension,size=0)	36.323520	83.483859	7700	1872
multiarray (nonResolved 1 dimension,size=0)	45.404340	104.354752	9625	2340
multiarray (nonResolved 1 dimension,size=5)	36.409720	83.689359	7720	1872
multiarray (nonResolved 1 dimension,size=5)	45.511250	104.622629	9650	2340
multiarray (nonResolved 2 dimension,size=5)	53.042079	121.065519	11212	2580
multiarray (nonResolved 2 dimension,size=5)	66.329640	151.331815	14015	3225
multiarray (Object 1 dimensions,size=8)	36.461440	83.812659	7732	1872
multiarray (Object 1 dimensions,size=8)	45.575820	104.7653259	9665	2340
multiarray (Object 2 dimensions,size=4)	49.866399	113.914719	10542	2450
multiarray (Object 2 dimensions,size=4)	62.332533	142.393380	13177	3062
multiarray (Object 4 dimensions,size=2)	84.130479	190.890879	17702	3934
multiarray (Object 4 dimensions,size=2)	105.163220	238.613557	22127	4917
arraylength 0xbe	0.983920	2.297580	210	52
arraylength 0xbe	1.229930	2.871875	262	65
baload 0x33	1.066680	2.554380	228	60
baload 0x33	1.837710	4.748040	388	99
caload 0x34	1.084380	2.610900	232	62
caload 0x34	1.710840	4.441620	360	90
saload 0x35	1.075780	2.586900	230	61
saload 0x35	1.846810	4.780560	390	100
iaload 0x2e	1.077460	2.588220	230	60
iaload 0x2e	1.730280	4.499460	364	91
laload 0x2f	1.183560	2.846640	252	67
aaload 0x32	1.077460	2.588220	230	60
aaload 0x32	1.730280	4.499460	364	91
bastore 0x54	1.155620	2.748520	246	64
bastore 0x54	1.701710	4.490440	358	90
castore 0x55	1.164480	2.787420	248	65
castore 0x55	1.701710	4.490440	358	90
sastore 0x56	1.164480	2.787420	248	65
sastore 0x56	1.710570	4.529340	360	91
iastore 0x4f	1.158240	2.774640	246	64
iastore 0x4f	1.721890	4.540560	362	91
lastore 0x50	1.310000	3.123480	276	71
aastore 0x53	2.003000	4.674360	418	100
aastore 0x53	1.721890	4.540560	362	91

Table 7: Opcodes costs, method invocation and return

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
invokevirtual+return(empty method)0xb6	12.032760	27.984120	2504	520
invokevirtual+return(empty method)0xb6	13.964910	34.207920	3005	887
invokevirtual+ireturn(empty method)	12.194540	28.380960	2536	529
invokevirtual+ireturn(empty method)	14.119470	34.631400	3036	893
invokevirtual+lreturn(empty method)	12.363720	28.753800	2570	537
invokevirtual+areturn(empty method,return this)0xb6	12.159580	28.284960	2528	525
invokevirtual+areturn(empty method,return this)0xb6	4.119470	34.631400	3036	893
invokestatic+return(empty method)0xb8	10.549340	24.648600	2198	455
invokestatic+return(empty method)0xb8	13.037960	31.513560	2811	840
invokestatic+ireturn(empty method)	10.711120	25.045440	2230	464
invokestatic+ireturn(empty method)	13.192520	31.937040	2842	846
invokestatic+lreturn(empty method)	10.880300	25.418280	2264	472
invokespecial+return(empty method)0xb7	10.512680	24.545700	2188	450
invokespecial+return(empty method)0xb7	13.502640	32.652420	2909	865
invokespecial+ireturn(empty method)	10.674460	24.942540	2220	459
invokespecial+ireturn(empty method)	13.657200	33.075900	2940	871
invokespecial+lreturn(empty method)	10.843640	25.315380	2254	467
invokespecial+areturn(empty method,return this)	10.639500	24.846540	2212	455
invokespecial+areturn(empty method,return this)	13.657200	33.075900	2940	871

Table 8: Opcodes costs, control flow

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
ifeq 0x99 (branch not taken)	0.957140	2.255160	206	54
ifeq 0x99 (branch not taken)	1.098400	2.763480	232	56
ifeq 0x99 (branch taken)	1.142280	2.617860	246	62
ifeq 0x99 (branch taken)	1.314400	3.303720	278	69
ifne 0x9a (branch not taken)	0.957140	2.255160	206	54
ifne 0x9a (branch not taken)	1.098400	2.763480	232	56
ifne 0x9a (branch taken)	1.142280	2.617860	246	62
fne 0x9a (branch taken)	1.314400	3.303720	278	69
iflt 0x9b (branch not taken)	0.957140	2.255160	206	54
iflt 0x9b (branch not taken)	1.098400	2.763480	232	56
iflt 0x9b (branch taken)	1.142280	2.617860	246	62
iflt 0x9b (branch taken)	1.314400	3.303720	278	69
ifle 0x9e (branch not taken)	0.957140	2.255160	206	54
ifle 0x9e (branch not taken)	1.098400	2.763480	232	56
ifle 0x9e (branch taken)	1.142280	2.617860	246	62
ifle 0x9e (branch taken)	1.314400	3.303720	278	69
ifgt 0x9d (branch not taken)	0.957140	2.255160	206	54
ifgt 0x9d (branch not taken)	1.098400	2.763480	232	56
ifgt 0x9d (branch taken)	1.142280	2.617860	246	62
ifgt 0x9d (branch taken)	1.314400	3.303720	278	69
ifge 0x9c (branch not taken)	0.957140	2.255160	206	54
ifge 0x9c (branch not taken)	1.098400	2.763480	232	56
ifge 0x9c (branch taken)	1.142280	2.617860	246	62
ifge 0x9c (branch taken)	1.314400	3.303720	278	69
if_icmpeq 0x9f (branch not taken)	0.995660	2.352480	214	56
if_icmpeq 0x9f (branch not taken)	1.158300	2.910120	244	58
if_icmpeq 0x9f (branch taken)	1.180800	2.715180	254	64
if_icmpeq 0x9f (branch taken)	1.374300	3.450360	290	71
if_icmpne 0xa0 (branch not taken)	0.995660	2.352480	214	56
if_icmpne 0xa0 (branch not taken)	1.158300	2.910120	244	58
if_icmpne 0xa0 (branch taken)	1.180800	2.715180	254	64
if_icmpne 0xa0 (branch taken)	1.374300	3.450360	290	71
if_icmplt 0xa1 (branch not taken)	0.995660	2.352480	214	56
if_icmplt 0xa1 (branch not taken)	1.158300	2.910120	244	58
if_icmplt 0xa1 (branch taken)	1.180800	2.715180	254	64
if_icmplt 0xa1 (branch taken)	1.374300	3.450360	290	71
if_icmple 0xa4 (branch not taken)	0.995660	2.352480	214	56
if_icmple 0xa4 (branch not taken)	1.158300	2.910120	244	58
if_icmple 0xa4 (branch taken)	1.180800	2.715180	254	64
if_icmple 0xa4 (branch taken)	1.374300	3.450360	290	71
if_icmpgt 0xa3 (branch not taken)	0.995660	2.352480	214	56
if_icmpgt 0xa3 (branch not taken)	1.158300	2.910120	244	58
if_icmpgt 0xa3 (branch taken)	1.180800	2.715180	254	64
if_icmpgt 0xa3 (branch taken)	1.374300	3.450360	290	71
if_icmpge 0xa2 (branch not taken)	0.995660	2.352480	214	56
if_icmpge 0xa2 (branch not taken)	1.158300	2.910120	244	58
if_icmpge 0xa2 (branch taken)	1.180800	2.715180	254	64
if_icmpge 0xa2 (branch taken)	1.374300	3.450360	290	71
lcmp 0x94 (value1<value2)	1.483220	3.404280	310	72
lcmp 0x94 (value1==value2)	1.693000	3.845520	356	79
lcmp 0x94 (value1>value2)	1.693000	3.845520	356	79
ifnull 0xc6(branch not taken)	0.957140	2.255160	206	54
ifnull 0xc6(branch not taken)	1.314400	3.303720	278	69
ifnull 0xc6 (branch taken)	1.124720	2.593860	242	61
ifnull 0xc6 (branch taken)	1.314400	3.303720	278	69
ifnonnull 0xc7 (branch not taken)	0.957140	2.255160	206	54
ifnonnull 0xc7 (branch not taken)	1.098400	2.763480	232	56
ifnonnull 0xc7 (branch taken)	1.124720	2.593860	242	61
ifnonnull 0xc7 (branch taken)	1.098400	2.763480	232	56
if_acmpeq 0xa5 (branch not taken)	0.995660	2.352480	214	56
if_acmpeq 0xa5 (branch not taken)	1.374300	3.450360	290	71
if_acmpeq 0xa5 (branch taken)	1.180800	2.715180	254	64
if_acmpeq 0xa5 (branch taken)	1.374300	3.450360	290	71
if_acmpne 0xa6 (branch not taken)	0.995660	2.352480	214	56
if_acmpne 0xa6 (branch not taken)	1.158300	2.910120	244	58
if_acmpne 0xa6 (branch taken)	1.180800	2.715180	254	64
if_acmpne 0xa6 (branch taken)	1.158300	2.910120	244	58
goto 0xa7	1.034380	2.400540	222	55
goto 0xa7	1.036670	2.589600	220	55
goto_w 0xc8	1.108140	2.551500	238	60
goto_w 0xc8	1.206750	3.009840	256	63
lookupswitch 0xab (1 iteration)	2.077300	4.759140	438	116
lookupswitch 0xab (1 iteration)	3.403660	8.655840	736	220
lookupswitch 0xab (2 iterations)	2.721600	6.228240	572	151
lookupswitch 0xab (2 iterations)	4.461740	11.268960	968	296
lookupswitch 0xab (3 iterations)	3.365900	7.697340	706	186
lookupswitch 0xab (3 iterations)	2.874620	7.349280	620	182
lookupswitch 0xab (4 iterations)	4.010200	9.166440	840	221
lookupswitch 0xab (4 iterations)	3.668180	9.309120	794	239
tableswitch 0xaa	1.793420	4.205460	384	108
tableswitch 0xaa	1.682750	4.288380	356	87

Table 9: Opcodes costs, logic opcodes

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Nb Cycles	Nb Proc. Inst.
ishl 0x78	0.976480	2.321580	208	53
ishl 0x78	1.210190	3.007440	254	60
ishr 0x7a	0.976360	2.321580	208	53
ishr 0x7a	1.139170	2.841060	242	63
iushr 0x7c	0.976420	2.321580	208	53
iushr 0x7c	1.210130	3.007440	254	60
lshl 0x79	1.419040	3.316500	300	72
lshr 0x7b	1.419040	3.316500	300	72
lushr 0x7d	1.545040	3.593340	324	76
iand 0x7e	0.957800	2.273580	204	51
iand 0x7e	1.191510	2.959440	250	58
ior 0x80	0.958100	2.273580	204	51
ior 0x80	1.191810	2.959440	250	58
ixor 0x82	0.958120	2.273580	204	51
ixor 0x82	1.191830	2.959440	250	58
land 0x7f	1.127820	2.701320	240	63
lor 0x81	1.128420	2.701320	240	63
lxor 0x83	1.128460	2.701320	240	63

Paper III

Receiver Coding Gain in DVB-H Terminals using Application Layer FEC Codes

Sébastien Lafond, Kristian Nybom, Jerker Björkqvist and Johan Lilius

Originally published in: proceedings of the *Third International Conference on Digital Telecommunications - ICDT 2008*, pages 110-116. IEEE, 2008

The Third International Conference on Digital Telecommunications

Receiver Coding Gain in DVB-H Terminals Using Application Layer FEC Codes

Sébastien Lafond, Kristian Nybom, Jerker Björkqvist, Johan Lilius
Abo Akademi University
Department of Information Technologies
sebastien.lafond@abo. , kristian.nybom@abo. , jerker.bjorkqvist@abo. , johan.lilius@abo.

Abstract

DVB-H is targeted for broadcasting digital content to handheld devices. The content is generally divided into streaming media or file downloading. In file downloading scenarios there is typically a requirement on a zero error ratio which can be met using either a data carousel or additional forward error correction codes. Both methods however induce a higher energy consumption in the receiver. This paper analyses the energy consumption needed for two different forward error correction codes based on emulator results for typical hardware in a handheld device. The energy used for error correction is compared to the energy used when receiving more carousel rounds in order to meet the zero error ratio requirement. This difference is denoted as receiver coding gain. Additionally, error correction also leads to a reduction in the reception time.

1. Introduction

DVB-H is a relatively new standard in the set of standards developed by the DVB Project. DVB-H is mainly targeted for handheld devices, but is also intended for mobile usage in for instance cars or buses. The main use case for DVB-H is watching television broadcasts, but le downloading enables receiving digital content (MP3's, videos, etc.) for storing and later use.

DVB-H is based on the physical layer of DVB-T. DVB-T is a Coded Orthogonal Frequency Division Multiplexing (COFDM) system, where the basic data item is Transport Stream (TS) packets of size 188 bytes. In broadcast systems, such as DVB-T, errors turn up in the stream, even while using good physical layer Forward Error Correction (FEC) codes. DVB-T was not designed for mobile usage, and therefore DVB-H includes an optional FEC code at the link layer, embedded in the Multi Protocol Encapsulator (MPE or MPE-FEC) to compensate for the performance degradations due to fast fading effects in mobile channels. These additions make the delivery of standard IP packets over the DVB-T network possible, which increases the performance in mobile usage environments. Furthermore, the MPE-FEC layer adds time interleaving to the system, making it more resistant to slow fading effects (e.g. temporal obstacles). While the MPE-FEC provides an adequate performance improvement for video streaming services, where errors lead to frame losses or pixelation, it does not add a suf cient performance improvement for le downloading scenarios.

In le downloading services, an additional layer of error correction at the application layer (AL-FEC) is used, in order to deal with lost IP packets. The DVB-H standard [3] speci es a Raptor code [7] for le delivery scenarios. Although Raptor codes have several attractive properties, they may not be the best choice for application layer coding. One of the most notable reasons, which will be shown in this paper, is that their energy consumption may be too large in mobile devices for providing cost effective downloading, compared to that of other codes. In this paper we present simulation results which indicate that Hyper Low-Density Parity-Check (HLDPC) codes provide similar error correction performance as the Raptor code, but at a lower energy consumption in the host processor.

The main contributions of this paper are: (a) a comparative study between two AL-FEC codes (the HLDPC and Raptor code), (b) an analysis of the receiver coding gain in terms of energy consumption when application layer coding is used, and (c) a theoretical analysis for receiver coding gain.

2. Receiver coding gain

Coding gain for a transmitter is traditionally de ned for a speci ed error probability as the reduction in required energy per transmitted information bit $E_{b,c}^{T}$ when using error correction coding compared to the energy per information bit $E_{b,0}^{T}$ when error correction coding is not used. The coding gain from the transmitter point of view is given by

978-0-7695-3188-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ICDT.2008.28

$$G^T = \frac{E^T_{b,0}}{E^T_{b,c}} \quad (1)$$

and is often given in the logarithmic scale dB as

$$G^T_{dB} = 10 \log_{10} \frac{E^T_{b,0}}{E^T_{b,c}} \quad (2)$$

In satellite communications the coding gain is essential, where the energy budget in transmitting satellites is limited. In terrestrial systems, the availability of energy at transmitter stations is not that big an issue, rather a network planning or a regulatory matter limiting the power available at the transmitter. In handheld devices, on the other hand, the total energy budget is limited. Using the same logic as de ning the transmitter side coding gain, we can de ne the receiver coding gain. Receiver coding gain is de ned as the reduction of energy per information bit $E_{b,c}$ needed while using coding compared to the energy per information bit $E_{b,0}$ needed without coding, where the same error rate is achieved:

$$G_{dB} = 10 \log_{10} \frac{E_{b,0}}{E_{b,c}} \quad (3)$$

The use of AL-FEC in downloading should provide receiver coding gain. Furthermore, additional error correction coding should provide other bene ts, which typically could be reduced downloading time for the required objects. The rest of this paper describes the codes used for additional coding, the emulations performed, and theoretical analysis for obtaining gures on typically achievable advantages of using coding.

3. Application Layer Codes Used for Simulations

In this paper, we compare the energy consumption of the Raptor code [7], the HLDPC code [5], and a system without AL-FEC. In [5], the erasure correction and overhead performances of these codes have been compared. We therefore limit ourselves to investigating the energy consumed in the receiving device by using the AL-FEC codes. The Raptor code is standardized for IP-datacasting (IPDC) services in DVB-H, but due to its license fees, other codes that can achieve similar performances in IPDC services are of great interest.

Raptor codes are rateless codes that belong to the class of concatenated LDPC/rateless LDGM codes, which achieve a good performance in terms of erasure correction and reception overhead performances. However, the decoding algorithm given in the DVB-H standard [3] for the Raptor code has a high computational complexity. As will be shown in this paper the HLDPC code shows a signi cantly better performance than the Raptor code, even when using less complex decoding algorithms for both codes. The HLDPC code is a xed-rate code, which has a similar erasure correcting performance as the Raptor code [5], but as will be shown in section 6, the energy consumption of the HLDPC code is signi cantly lower than that of the Raptor code.

The Raptor decoding algorithm relies on Gaussian elimination of the parity-check matrix and is capable of yielding reception overhead performances in the order of 1–2 %. Because of the algorithm's high complexity we therefore compare the Raptor code with the HLDPC code, using a computationally simpler algorithm for both codes, namely the greedy iterative Belief Propagation algorithm. Using this algorithm, the erasure correcion and reception overhead performance of the Raptor code is degraded at the gain of decreasing the computational complexity in the decoder. The greedy iterative Belief Propagation algorithm works as follows:

Algorithm 3.1 *Given the value of a parity symbol and all but one of the information symbols on which it depends, set the missing information symbol to be the XOR of the parity symbol and its known information symbols.*

Clearly, this algorithm only works on erasure channels where the decoder knows which symbols are correct and which are not. Since the IP layer in a network protocol can be viewed as a packet erasure channel, where IP packets are either received without errors or corrupted and therefore discarded, the greedy iterative Belief Propagation algorithm is usable at the application layer.

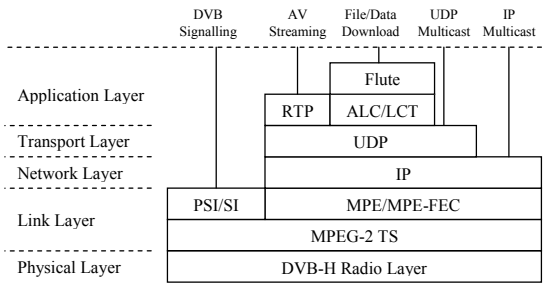


Figure 1. A overview of the system layers in a DVB-H receiver.

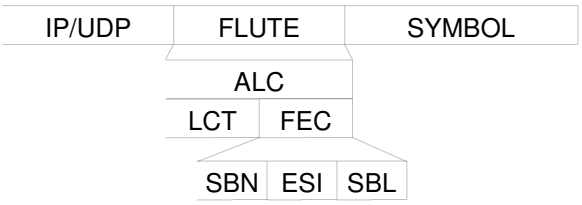


Figure 2. Packet structure for the encoded data

4. Protocols used in file downloading

The system layers in a DVB-H receiver are illustrated in figure 1, [2]. For the scope of this paper, only the application layer is of interest. In DVB-H IPDC services, the FLUTE protocol [4] is used for delivering objects to the receiving terminals. The FLUTE protocol is built on top of the Asynchronous Layered Coding (ALC) protocol, which combines the Layered Coding Transport (LCT) building block, a congestion control building block, and the FEC building block. However, the congestion control building block is not used in DVB-H IPDC services. The ALC and LCT building blocks contain relevant information for the file delivery, while the FEC building block is used by the FEC decoder. The FEC building block is comprised of three fields: the Source Block Number (SBN), the Encoding Symbol ID (ESI), and the Source Block Length (SBL). This gives the IP packet structure that is shown in figure 2.

The information obtained from the FEC building block is used in the following manner. The SBN signifies to which FEC block the received symbol belongs. The ESI is the encoding symbol index of the received symbol with the rule that if ESI $\geq$ SBL the received symbol is a parity symbol, otherwise it is an information symbol. Additionally, the Raptor code uses the ESI value as a seed to its random number generator, to create the degree and edge distributions of the symbol (see [3] for details). The SBL is the number of information symbols in the FEC block to which the received symbol belongs to.

5. Measurement Framework

A measurement framework was created in order to evaluate the extra costs created by the HLDPC and Raptor decoders. Figure 3 presents the full measurement framework used for evaluating the HLDPC and Raptor codes. The file containing the data to be transmitted over the DVB-H network was first encoded by the HLDPC or the Raptor encoder. For the measurements the IP/UDP and LCT headers were not included in the packets because the decoders only require the FEC headers for reconstructing the received object. The obtained encoded file was thus composed of packets and each packet contained one FEC header and one symbol.

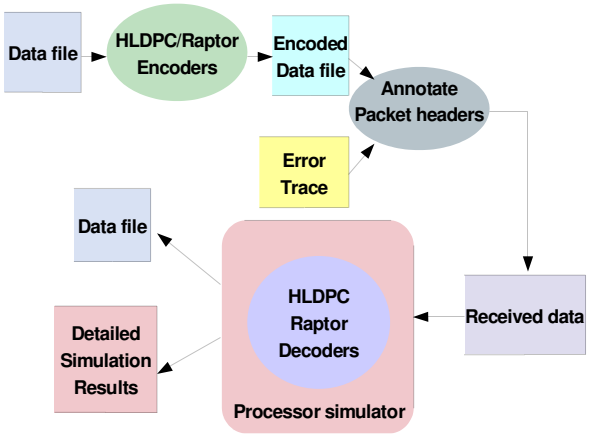


Figure 3. Measurement system - General view

The error trace was a binary map specifying whether each transmitted packet was correctly received or an erasure. In this work a Binary Erasure Channel (BEC) was used, i.e. erasures were distributed uniformly at random. Using the error trace each packet was tagged with erasure information by setting a flag in the packet header. Figure 4 shows the packet structure for the received data containing the error flag. The received data was then read by a software implementation of the HLDPC or Raptor decoder which tried to reconstruct the received object.

The Sim-Panalyzer [8] processor simulator was used for evaluating the costs generated by the execution of the decoders. Sim-Panalyzer is based on the SimpleScalar [1] processor simulator and performs cycle accurate simulations of a strongARM SA-110 processor. It computes at every simulated cycle the energy consumption of each module constituting the ARM core (clock, alu, cache, etc.). RTEMS was chosen as the operating system for this study because RTEMS 4.6.2 is to the best of our knowledge the only OS ported onto SimpleScalar (ported by Jack Whitham [9]).

Table 1. Source and code lengths in number of symbols

Code length	Source length	Code rate
4000	3000	0.75

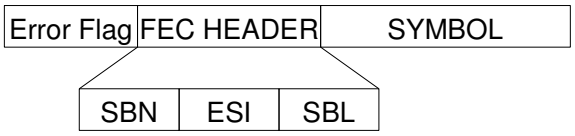


Figure 4. Packet structure for the received data

5.1. Simulation parameters

Tables 1 and 2 present the parameters used by the HLDPC and Raptor codecs. For all measurements, the number of information symbols were set to 3000 symbols and the codeword lengths were set to 4000, hence giving code rate $R = 3/4$. The Raptor code was a non-systematic code, i.e. 4000 rateless symbols were transmitted. The FEC building block used 12 bytes for the SBN, ESI, and SBL header fields and the symbol sizes were set to 1432 bytes, thereby giving IP packet payloads of 1444 bytes.

The processor parameters and the configuration of the caches must be defined in Sim-Panalyzer. For this study the processor speed was set to 233 MHz. The configuration for the level 1 instruction cache, level 1 data cache and the unified level 2 cache is presented in table 3. Table 4 shows the different latencies for each memory level. This configuration targets the average performance of the host processor in a multimedia handheld device. All other parameters used by Sim-Panalyzer were set to their default values.

6. Results

The measurement framework for the HLDPC and Raptor codes was run on the BEC with probabilities of erasures ranging from 0% to 20%. All data was transmitted in a carousel-like manner. The Raptor code was able to decode within one carousel round the received data containing up to 16% of erasures, while the HLDPC code was able to decode data containing up to 14% of erasures.

Figure 5 presents for each code the required time in clock cycles for decoding the received data depending on the erasure rate. We observe that increasing the erasure rate does not affect the execution time of the Raptor decoder while the execution time for the HLDPC decoder slightly increases

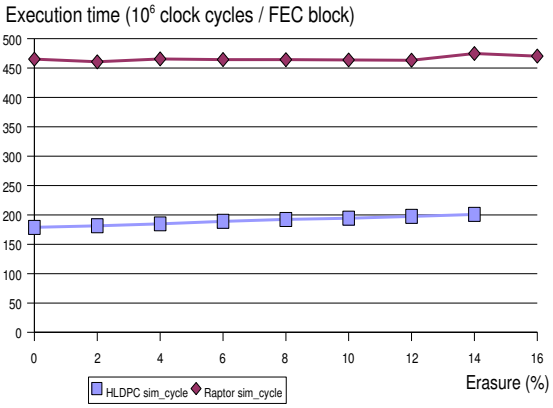


Figure 5. Clock cycles needed for complete reconstruction of received data

with the erasure rate. This is due to the Raptor code being unaffected by the erasure rate in the channel in terms of reception overhead performance, i.e. the Raptor code has an approximately constant reception overhead. In other words, almost every symbol that the Raptor code receives can be used for decoding, while the HLDPC code may receive symbols that have already been received or reconstructed and therefore are useless for the decoding procedure. For the HLDPC code, this fact reflects itself as increased execution time. The results indicate that the raptor decoder requires about $465 \cdot 10^6$ processor cycles regardless of the erasure rate. On the other hand, depending of the erasure rate the HLDPC code requires about $175 \cdot 10^6$ to $200 \cdot 10^6$ processor cycles in order to reconstruct the object.

6.1. Energy Budget Analysis

In this section we evaluate the cost of using AL-FEC in terms of energy consumption. For all the results presented in this section, we assume a data transmission rate T_b of 5 Mbits/s.

Based on Monte Carlo simulations (a similar approach as in [6]), table 5 presents for a receiver not using AL-FEC the minimum, maximum and average number of required carousel rounds on the BEC for downloading the uncoded object without errors. The given values are obtained based

Table 2. Symbol, Packet and Data file sizes in bytes

Symbol	Packet	Data file	Encoded data file
1432	1444	4 296 000	5 776 000

Table 3. Caches configuration

Caches	Associativity	Size	# blocks
il1	direct mapped	4 Kb	128
dl1	direct mapped	4 Kb	128
ul2	4-way	8 Kb	256

on 1000 experiments.

Figure 6 and Figure 7 present the energy consumption comparison for decoding the received data between a receiver without AL-FEC and a receiver using the HLDPC and the Raptor codes respectively for several average power dissipations while the data is received. The average power dissipated while receiving the data includes the power dissipated by the radio receiver and by other system units, like the screen, processor, possible speaker, etc. Figures 6 and 7 show that the energy consumption for a receiver without AL-FEC is increasing with the erasure rate, proportionally to the average power dissipated while receiving data, but at a faster pace than for a receiver using AL-FEC.

The energy consumption for a receiver without AL-FEC, illustrated in gures 6 and 7 with dashed lines, are calculated with the following equation as

$$E_{tot,0} = \epsilon_0 t_0 \bar{P} \tag{4}$$

where ϵ_0 denotes the transmission overhead for an uncoded transmission, t_0 is the time consumed for transmitting all information symbols during one carousel round, and $\bar{P}$ is the average power consumed by the receiver. If n' is the number of transmitted symbols at the time when the receiver has obtained the entire object and k is the number of information symbols in the object, then the transmission overhead is de ned as $\epsilon = \frac{n'}{k}$, hence $\epsilon \geq 1$. The energy consumption for a receiver using AL-FEC is calculated in a similar manner as

$$E_{tot,c} = \epsilon_c t_0 \bar{P} + E_c \tag{5}$$

where $E_{tot,c}$ is the total energy used for receiving an object including decoding, ϵ_c is the transmission overhead for the encoded transmission, and E_c is the total energy used by the decoder on host processor, in our case the simulated strongARM SA-110 processor. Table 6 gives as example the energy consumed by the processor functional units for decoding an object with 6% of erasures. The values of $E_{tot,c}$ are illustrated in gures 6 and 7 with continuous lines.

When dividing the total energy required by the number of source bits L_0 we get the energy per bit as

$$E_{b,c} = \frac{E_{tot,c}}{L_0} \tag{6}$$

Table 4. Memory Latencies

	il1	dl1	ul2	RAM *first chunk access*	RAM *inter chunk access*
Latency in cycles	2	2	6	30	4

Table 5. Number of required carousel rounds on the BEC when no AL-FEC is used

Erasure rate (%)	Avg	Min	Max
0	1	1	1
2	2.71	2	5
4	3.17	2	5
6	3.54	3	6
8	3.91	3	8
10	4.26	3	7
12	4.55	3	9
14	4.86	4	8
16	5.16	4	9

$$E_{b,0} = \frac{E_{tot,0}}{L_0} \tag{7}$$

Substituting the above expressions into equation 3 gives

$$G_{dB} = 10 \log_{10} \frac{\epsilon_0 t_0 \bar{P} L_0^{-1}}{\epsilon_c t_o \bar{P} L_0^{-1} + E_c L_0^{-1}} \tag{8}$$

Simplifying equation 8 and taking into consideration that $t_0 = \frac{L_0}{T_b}$, where T_b is the transmission rate in the network, the nal expression for the receiver coding gain is obtained as

$$G_{dB} = 10 \log_{10} \frac{\epsilon_0}{\epsilon_c + \frac{TbE_c}{L_0 \bar{P}}} \tag{9}$$

Note that by using probability theory, the expected value on the transmission overhead for the uncoded transmission $E\ \ \epsilon_0\ \ $ (abusing notation) over a BEC can be calculated as

$$E\ \ \epsilon_0\ \ = \sum_{\epsilon_0=1}^{\infty} \epsilon_0 \left[(1 - p_e^{\epsilon_0})^k - \left(1 - p_e^{\epsilon_0 - 1}\right)^k \right] \tag{10}$$

Table 6. Energy consumption in Joule for the processor functional units for an erasure rate of 6%

	HLDPC	Raptor
instruction cache level 1	0.362	0.860
data cache level 1	0.204	0.483
uni ed cache level 2	1.18	3.41
clock	0.207	0.509
μ architecture	0.749	1.80
ALU	0.000526	0.00132
Total (E_c)	2.70	7.07

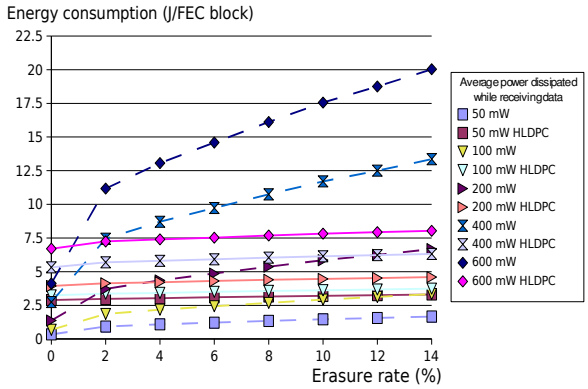


Figure 6. Energy consumption comparison between uncoded and HLDPC coded transmission

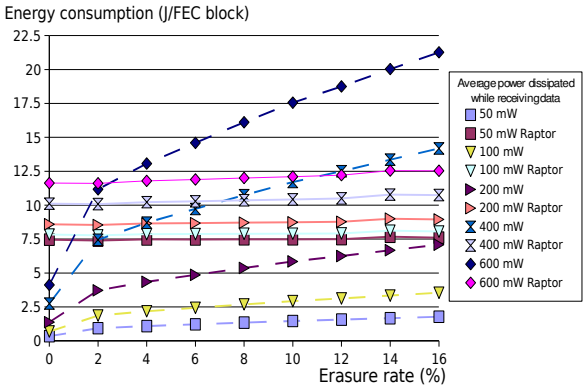


Figure 7. Energy consumption comparison between uncoded and Raptor coded transmission

where p_e is the probability of an erasure, and k is the number of information symbols in the transmitted object. To clarify the equation, the probability of all symbols being correct at a transmission overhead of ϵ_0 is $(1 - p_e^{\epsilon_0})^k$. Therefore, the expression inside the brackets signi es the probability of all symbols being correct at a transmission overhead of *exactly* ϵ_0.

Using equation 9 we can now calculate the receiver gain when AL-FEC is used. Figures 8 and 9 present the obtained receiver coding gain when the HLDPC and Raptor codes are used. As the use of AL-FEC is bene cial only when the receiver coding gain is positive, the comparison of gures 8 and 9 clearly shows better performance for the HLDPC code than the Raptor code. As an example we can see that for an average receiver power consumption $\bar{P}$ of 200mW, the HLDPC code is more ef cient than a system without AL-FEC for erasure rates of 4% and upwards. On the other hand, for the same average receiver power consumption the Raptor code is unef cient compared to a system without AL-FEC for all the erasure rates.

It is also important to note that the simulated strongARM SA-110 processor is becoming an outdated processor. As technology evolution since the late 90's concentrated efforts in developing more energy ef cient processors, we can expect that with modern processors the receiver coding gain when using AL-FEC could reach positive values for even smaller erasure rates than the one presented on gures 8 and 9.

7. Conclusions

In this paper, we evaluated the use of AL-FEC techniques for achieving error free delivery of data objects in a DVB-H system. The alternative to using AL-FEC is to retransmit the data in a data carousel thereby waiting, in the worst case, for several carousel rounds before all the data is received without errors. AL-FEC decoding is performed in the receiver general purpose host processor. In order to be ef cient from an energy point of view, the energy consumed by the host processor for handling the application layer coding should be smaller than the energy consumed by the receiver device for receiving the extra carousel rounds.

Two AL-FEC codes, the HLDPC code and Raptor code, were run in an emulator system, from which detailed information on energy dissipation could be obtained. The energy used for the AL-FEC codes was compared to the energy needed for receiving additional carousel rounds. As the exact energy performance gures for the receiver equipment (frontend) was not known, a set of different average power dissipations were used for the simulations. We believe that this set of average power dissipations covers the range of most receiver equipment characteristics.

Depending on the AL-FEC code and the erasure rate, the receiver coding gain was in the region of -9 to 4 dB. This shows that the energy used by the AL-FEC codes is of the same magnitude as the energy needed for receiving additional carousel rounds. On the other hand, by using AL-FEC codes the transmission overhead is reduced, leading to faster downloading for the end-users. Moreover, the transmission bandwidth is reduced, because the number of required carousel rounds containing the same data is reduced. Thus, using AL-FEC codes is an appealing approach.

The HLDPC and Raptor codecs used in this work, were originally implemented in a PC environment in ANSI C++, using rather na¨ve software engineering. For example, dynamic memory allocations have been frequently used, spe-

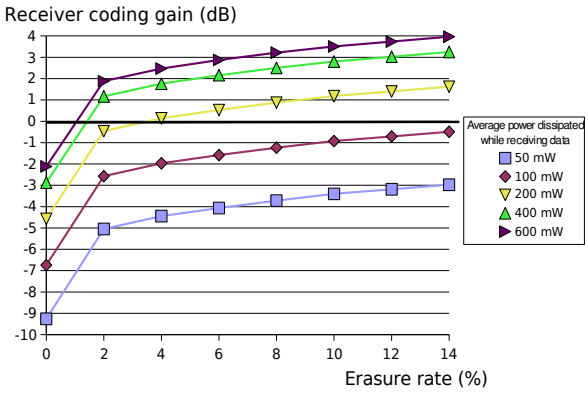


Figure 8. Receiver gain when HLDPC is used

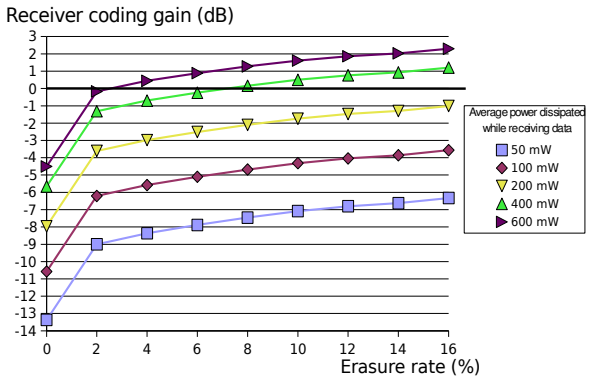


Figure 9. Receiver gain when Raptor code is used

cial processor instructions for optimizing performance have not been used. This code was then recompiled for the emulator framework. Optimizing the code in general, and speci c optimizing for the target processor architecture would certainly give some additional gain.

Future host processor architectures in mobile handsets will furthermore be more energy ef cient, hence increasing the receiver gain. Receiver chipsets for DVB-H will of course also be more energy ef cient, but assuming that the host processor development will be faster the experiments presented in this paper shows that AL-FEC codes are already fully applicable technologies, providing time and transmission bandwidth savings at practically no extra cost in the receiver.

References

[1] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, 1997.

[2] ETSI TR 102 469 V1.1.1. *Digital Video Broadcasting (DVB): IP Datacast over DVB-H: Architecture*, 2006.

[3] ETSI TS 102 472 V1.2.1. *Digital Video Broadcasting (DVB): IP Datacast over DVB-H: Content Delivery Protocols*, 2006.

[4] IETF RFC 3926. *FLUTE - File Delivery over Unidirectional Transport*, 2004.

[5] K. Nybom and J. Björkqvist. Hldpc codes - low density, low complexity, ef cient erasure correcting codes. In *Proceeding of the 13th European Wireless Conference*, Apr 2007.

[6] J. Peltotalo, S. Peltotalo, J. Harju, and R. Walsh. Performance analysis of a le delivery system based on the ute protocol. *Int. J. Commun. Syst.*, 20:633–659, 2007.

[7] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, June 2006.

[8] Sim-Panalyzer. http://www.eecs.umich.edu/˜panalyzer.

[9] J. Whitham. http://www.jwhitham.org.uk/simplescalar/.

Paper IV

Energy efficiency analysis of multi-stream MPEG-4 decoder systems

Sébastien Lafond, Jani Boutellier, Johan Lilius and Olli Silvén

Originally published in: proceedings of the *Multimedia on Mobile Devices 2008*, SPIE Vol. 6821, 68210G-1. SPIE, 2008

Energy efficiency analysis of multi-stream MPEG-4 decoder systems

Sébastien Lafond[a] and Jani Boutellier[b] and Johan Lilius[a] and Olli Silvén[b]

[a]Åbo Akademi University, Joukahainengatan 3-5, Turku, Finland
[b]University of Oulu, PO BOX 4500, Oulu, Finland

ABSTRACT

This paper presents a comparison of two systems that can simultaneously decode multiple videos on a simple CPU and dedicated function-level hardware accelerators. The first system is implemented in a traditional way, such that the decoder instances access the accelerators concurrently without external coordination. The second system implementation coordinates the tasks' accelerator accesses by scheduling. The solutions are compared by execution cycles, energy consumption and cache hit ratios. In the traditional solution each decoder task continuously requests access to the needed hardware accelerators. However, since the other tasks are competing on the same resources, the tasks must often yield and wait for their turn, which reduces the energy-efficiency. The scheduling-based approach assumes that the accelerator latencies are deterministic and assigns time slots for accelerator accesses required by each task. The accelerator access schedule is re-designed for each macroblock at run-time, thus avoiding the over-allocation of resources and improving energy-efficiency. Deterministic accelerator latencies ensue that the CPU is not interrupted when an accelerator finishes. The contribution of this study is the comparison of the accelerator timing solution against the traditional approach.

Keywords: Video codecs, Parallel processing, Power demand

1. INTRODUCTION

Multimedia applications running on modern mobile devices require huge amounts of computational resources. Performing all the required computations in software is not a feasible alternative, since general-purpose processors (GPP) offer low energy-efficiency and low data throughput. Thus, the computationally intensive application parts are often offloaded to dedicated processing elements (PEs) that can perform the computations faster and with lower power consumption.[1] However, running the application on multiple PEs raises the problem of synchronization: a processor must know before interacting with another PE, if the device is ready for the interaction. This synchronization can be performed either by polling the status of the other PE, or by letting the working PE announce when it is finished.[2] Recently, also an alternative way has been proposed for doing the synchronization of the accelerators: by using deterministic scheduling[3,4] it is possible to avoid repeated polling of other PEs, as well as interrupts.

In this paper we compare the synchronization overhead of a hardware-accelerated polling-based system against a more sophisticated synchronization-by-scheduling system. As a reference we also compare the aforementioned solutions against a full-software implementation. The comparison is based on the measurement of execution cycles, energy consumption and cache hit ratios. The measurement results have been acquired by running a multi-stream MPEG-4 decoding application on a cycle-accurate strongARM SA-1100 processor simulator.

Further author information:
Sébastien Lafond: sebastien.lafond@abo.fi
Jani Boutellier: bow@ee.oulu.fi

Multimedia on Mobile Devices 2008, edited by Reiner Creutzburg, Jarmo H. Takala
Proc. of SPIE-IS&T Electronic Imaging, SPIE Vol. 6821, 68210G, © 2008 SPIE-IS&T · 0277-786X/08/$18

2. MEASUREMENT FRAMEWORK

The application framework for our measurements was multi-stream MPEG-4 video decoding performed by the open-source XViD[5] codec. MPEG-4 video decoding was chosen as the application framework, because it is computationally very demanding and dynamic. Decoding one second of a 320x240-pixel movie (with 25 fps) involves processing 7500 macroblocks, of which each can have a different decoding procedure. The decoding procedure of a macroblock is discovered as the video bitstream is read and can not be predicted in advance. Therefore, if the decoding is accelerated by hardware, the accelerator elements must be so fine-grained that they can be adapted to the decoding needs of each macroblock.[3] Naturally, the decoding hardware designer can also just assume the worst-case scenario and allocate the maximum amount of hardware resources for each macroblock, but this will lead to an inefficient solution. Although MPEG-4 is a bit dated as a video compression scheme, the results of this paper can also be applied to upcoming standards such as RMC.[6]

The measurements were conducted on three different system configurations: one of the systems ran several independent unmodified XViD software decoders on a single GPP by using multitasking, whereas the other two systems contained the GPP and several dedicated hardware accelerators to do the same task. Since the XViD codec is originally described in monolithic c-code, some effort was required to make the decoder suitable for hardware acceleration. These modifications introduced some processing-time and memory overhead.

The codec itself is only capable of decoding one video stream at a time, therefore an OS-like wrapper application was created for the two hardware-accelerated systems. Upon start, the wrapper application initializes 1...4 video decoders and starts decoding the video streams. The wrapper application does not run the decoding of separate streams in parallel, instead it uses time-division multiplexing of the GPP processing time. When the code execution of decoder instance n reaches a point that it requires code execution on hardware accelerators, it returns control to the wrapper application along with the accelerator access requests. The wrapper application stores these accelerator access requests (but does not activate the accelerators yet) and starts decoder instance *n+1*. When decoder *n+1* code execution reaches the stage that it can not proceed without use of accelerators, decoder *n+1* returns control to the wrapper application and so on. Once all the decoder instances have finished executing the control code and are waiting for computation results from the accelerators, the wrapper application starts to execute the accelerator access requests. How this is done, depends on the synchronization scheme.

The traditional, polling-based system starts processing the stored accelerator access requests in a first-requested first-served manner. However, if an accelerator is reserved, the new access request must yield and wait until the accelerator is freed – meanwhile the system continues polling the other accelerators to see if some other access request could be executed. The second system implementation uses run-time scheduling to plan the accelerator access pattern in advance. The scheduling algorithm itself causes some overhead, but on the other hand avoids completely the polling overheads later, because the schedule tells in advance when the resources will be freed. The completely software based stream decoding does not need further explanation: all computations are performed on the GPP in traditional time-division multitasking fashion.

Figure 1 shows approximately the behaviour of the hardware accelerated solutions in a Gantt chart. W represents processing time spent within the wrapper application and D1, D2 and D3 refer to activity in the control code of the respective decoder instances. The blocks on the accelerator rows below show an arbitrary schedule of task executions on the accelerators.

2.1 Scheduler

The scheduler used by the second accelerated system implementation is based on the idea that is described in.[4] Theoretically it is a permutation flow-shop (PFS)[7] scheduler, that has been applied to the problem of PE scheduling. In PFS terminology the processing units are *machines* and the tasks perfomed by the processing units are *operations*. Dependencies between tasks are described by grouping tasks into *jobs*. A job is defined to contain an operation for each machine, and each job must access the machines in the same order. In our application, we have also used *machine skipping*, which means that for some jobs, the execution time of certain operations may be zero: i.e. nothing is performed on that machine. By the PFS definitions, the execution times of operations are deterministic. This is not a severe limitation, since the accelerated functions are very predictable. Also, the assumption about deterministic execution times has been made previously in similar contexts.[8,9] From this

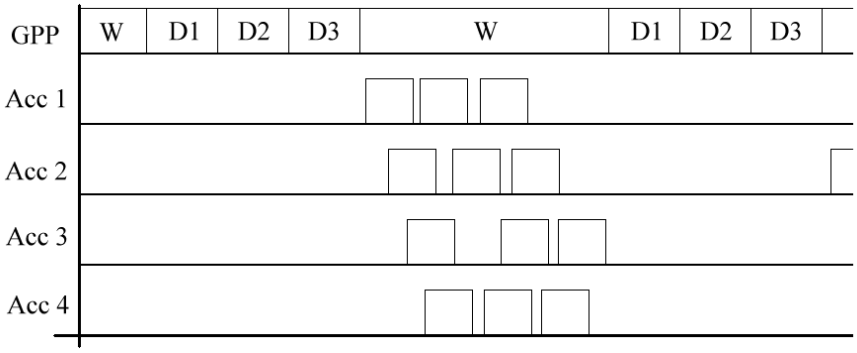


Figure 1. Sequential control code processing and parallel accelerator operation.

description, it is evident that PFS can be applied only to some scheduling problems, as the one presented in this paper.

The benefit of such a restricted scheduling problem is that the scheduling algorithm can be very straightforward and since the scheduler is called with a high frequency, a low overhead will be the most important characteristic of the scheduler. Thus, of the different scheduler implementations described in,[4] the "no job ordering, no-wait timetabling" was selected for computing schedules in this environment. No-wait timetabling (Figure 2) means that within the same job, the next operation is started immediately after the previous one finishes. In our scheduling problem this is essential, because it ensures that buffers between processing units are not overwritten too early.

2.2 Hardware Accelerators

The parts of the XViD code to be hardware accelerated, were selected manually. Evidently, it is most beneficial to use acceleration for compact parts of the code that are invoked often, e.g. nested loops. In MPEG-4 video decoding this part is found from macroblock decoding and especially in block decoding (in our case each macroblock consists of six blocks).

Besides being often invoked, it is desirable that the accelerated code parts should have a minimal amount of inputs and outputs. Based on these reasons, the hardware accelerators were created from the block decoding functions, that are depicted in Figure 3. The figure consists of boxes (accelerators), circles (buffers) and arrows that indicate the dataflow between the entities. It can be seen that there are several different dataflows, of which only some are used for each block, depending on the coding scheme.

When choosing the accelerated functions, some compromises had to be made with the modularity (amount of inputs and outputs) of accelerators. This does not affect the results between the two hardware accelerated systems, since both of them use the same accelerator units. However, it causes some extra overhead to the hardware accelerated systems when they are compared against the full-software based approach.

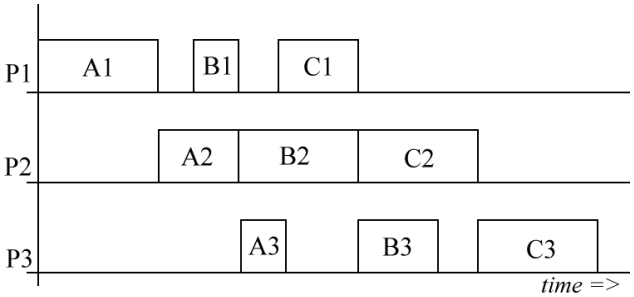


Figure 2. No-wait timetabling of three jobs (A,B,C) on three processors.

Table 1. Hardware accelerator latencies in clock cycles

Accelerator 1	Accelerator 2	Accelerator 3	Accelerator 4	Accelerator 5
16	25	13	8	200

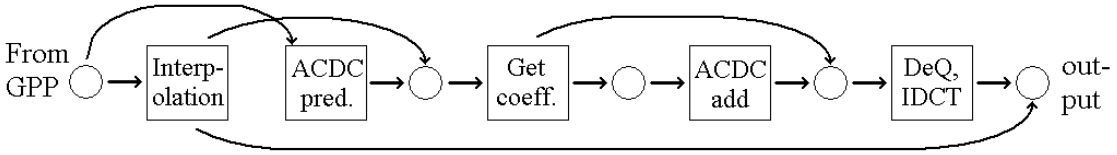


Figure 3. Data flow between accelerators.

2.3 Hardware Platform

The two hardware-accelerated decoding systems are based on the hardware platform presented in Figure 4. It consists of six PEs: a general purpose processor and five dedicated hardware accelerators. The PEs are triggered according to the polling-based approach or by the scheduling solution, that has been described previously. Both solutions trigger the accelerators in a "waterflow" manner, where each accelerator passes its computed results to the following accelerator via a shared local memory. The full-software decoding system is running on the GPP of the same platform and does not have any use for the dedicated hardware accelerators. The used hardware accelerator latencies can be seen in Table 1.

3. SIMULATION FRAMEWORK

The simulation framework presented in this section models a typical handheld device featuring basic multimedia application. It includes a hardware platform, an operating system and a set of applications and hardware accelerators.

3.1 Processor simulator

The Sim-Panalyzer[10] processor simulator was used for this study. Sim-Panalyzer is based on the SimpleScalar[11] processor simulator, and performs cycle accurate simulation of a strongARM SA-1100 processor. At every simulated cycle it computes the energy consumption of each module within the ARM core (clock, ALU, cache, etc.). The processor simulator allows running an ARM-based operating system on top of it.

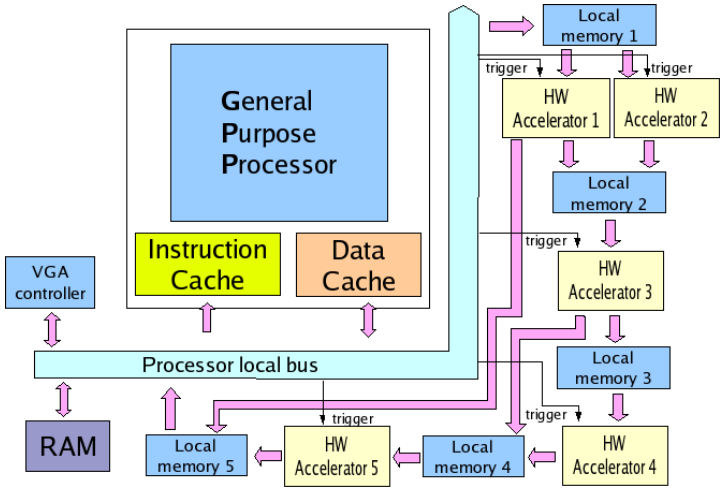


Figure 4. Hardware platform.

3.2 Operating system

The SimpleScalar port of the real-time operating system RTEMS (v. 4.6.2) was used in this study.[12] This port includes a SimpleScalar extension for supporting an interrupt based programmable timer which is needed by RTEMS. RTEMS is a free open source real-time operating system designed for embedded systems and supporting a variety of application programming interfaces (APIs) and interface standards. This real-time operating system allows the execution of a set of applications as independent tasks in a pre-emptive multitasking environment, which enabled us to conduct the measurements with multiple independent software decoders.

3.3 Accelerators

The accelerators are implemented within the Sim-Panalyzer framework and the applications can trigger the accelerator via dedicated system calls. It is the responsibility of the application to move the input data into the local memory 1 feeding the hardware accelerators 1 and 2 on Figure 4. In the same way, it is the responsibility of the application to read the results from local memory 5 when needed. For the rest of the accelerators reading input parameters is automatically done when the accelerator is triggered.

The following pseudo code illustrates how the communication with the hardware accelerators is handled for the polling-based and the scheduled systems:

```
Polling-based accelerator access
> repeat:
  > if(accelerator is free)
    > read input data to local memory
    > trigger/call hardware accelerator
  > endif
> goto repeat

Implementation using the scheduler
> compute schedule
> repeat:
  > sleep until designated time
  > read input data to local memory
  > trigger/call hardware accelerator
> goto repeat
```

In order to maintain the cache coherency the Sim-Panalyzer handles the memory accesses to the accelerator local memories as uncached memory regions. It is important to note that the execution of the hardware accelerators is performed outside the simulated platform. The energy consumption of the hardware accelerator is therefore not directly measured by the system. Instead, based of the difference in power dissipation between the full-software decoder system and hardware-accelerated systems presented in the results-section, it is possible to calculate an energy budget that tells us how much energy the accelerators can use to still provide a better energy efficiency than the full-software decoder.

Table 2. Memory latencies in cycles

	IL1	DL1	UL2	Local acc. memory	Main memory *first chunk access*	Main memory *inter chunk access*
Latency in cycles	1	1	4	4	30	4

Table 3. Configuration of the caches

Caches	Associativity	Size	# of blocks	Block size
IL1	Direct mapped	4 Kb	128	32 bytes
DL1	Direct mapped	4 Kb	128	32 bytes
UL2	4-way	8 Kb	256	32 bytes

3.4 Simulation Parameters

This subsection defines the constant and variable parameters and the corresponding values used in the simulation framework. Sim-Panalyzer defines the processor parameters and the configuration of the caches. For this study the processor speed was set at 233 MHz. The configuration for the level 1 instruction and data cache and the unified secondary cache is presented in Table 3. Table 2 shows the different latencies for the caches and the local accelerator memories. All other parameters used by the Sim-Panalyzer were set to their default values. The used input data for all systems consisted of four compressed 320x240 pixel video streams, that had 45 frames each and a nominal speed of 15 fps. For the full-software implementation RTEMS uses time slices of 50 ms, which implies 20 task switches per second. This configuration tries to model an average embedded system that could be used in a multimedia handheld device.

4. RESULTS

For clarity all graphs presenting measurement results are given at the end of the paper in Appendix A. In the abbreviations FS stands for "full-software-based", HA for "hardware accelerated", PH for "polling, hardware accelerated" and SH for "scheduler, hardware-accelerated". Figures 5 to 9 present the cost differences for initializing the MPEG-4 video decoder with the full-software and hardware accelerated decoding systems. As the two hardware-accelerated decoding systems share the same piece of code for initializing the decoder, the initialization costs are common for both systems.

Figure 5(a) presents the execution time in clock cycles and Figure 5(b) presents the power dissipated by the microarchitecture for initializing the full-software and the hardware accelerated decoding systems. The better results for the hardware accelerated decoding systems presented in these two graphs can be explained by the task switching overhead in the multi-tasking environment for the full-software system. The numbers in Figure 7 can also be explained by the task switching overhead.However, Figure 8(b) presents an increase of misses in the level one data cache for the hardware accelerated decoding systems, which according to figures 9(a) and 9(b) led to an increase of hits in the unified level 2 cache. As shown on Figure 6, this affect the power dissipated by the unified level 2 cache. These variations in cache activity are explained by the modifications done to the XViD codec, to make it suitable for hardware acceleration. As stated before, the full-software implementation uses the original XViD.

Figures 10 to 14 present the average costs for decoding one frame with the full-software and the two hardware accelerated decoding systems. For each measurement the average costs for decoding one frame is obtained by applying the following formula:

$$\text{Average cost per frame} = \frac{\text{Total cost - Initialization cost}}{\text{Total number of decoded frames}} \quad (1)$$

Figure 10(a) shows the average speed of execution for decoding one frame on each system. The system using the hardware accelerator and the scheduler is more than twice faster than the polling based system and about 40% faster than the full-software system. Thus, if we take an average power dissipation of 550 mW for a strongARM SA-1100 processor,[13] the maximum average energy budget for all accelerators must stay below 220 mW, if we want to get the scheduler based system to have a better energy efficiency than the full-software system. On the other hand, figures 10 and 11 clearly show the inefficiency of the polling based system compared to the two others. As the graphs in figures 13 and 14 show, this inefficiency in execution time and power dissipation is mainly due to a huge data access increase in the polling based system. However, it must be pointed out that the software-based polling solution used here is clearly very inefficient and could be implemented in a much more efficient way by using some kind of hardware support. Finally, with the used simulation parameters, we can

see that only the scheduler-based system is able to decode the four video streams in real time at 15 frames per second.

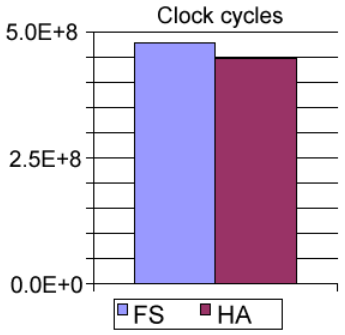
5. CONCLUSION

We have presented a comparison between three different multi-stream MPEG-4 video decoding systems. The comparison was made based on measurements of execution time, power dissipation and cache behaviours. The compared systems consisted of one fully software-based and two hardware accelerated solutions. One of the hardware-accelerated solutions used polling to do synchronization between processing elements, whereas the other one used a new scheduling-based synchronization approach. The measurement results showed that the hardware accelerated, scheduling-based solution provided the best energy efficiency of these three, if the hardware accelerators do not consume too much power.

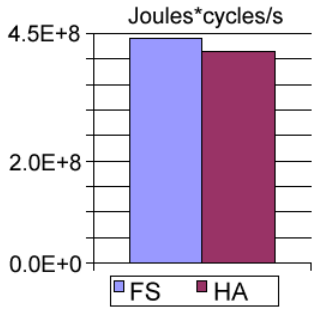
REFERENCES

1. W. Wolf, *High-Performance Embedded Computing*, Morgan Kaufmann, 2006.
2. O. P. Gangwal, A. Nieuwland, and P. Lippens, "A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems," in *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pp. 1–6, ACM, (New York, NY, USA), 2001.
3. T. Rintaluoma, O. Silven, and J. Raekallio, "Interface overheads in embedded multimedia software," *Embedded Computer Systems: Architectures, Modeling, and Simulation* , pp. 5–14, 2006.
4. J. Boutellier, S. S. Bhattacharyya, and O. Silven, "Low-overhead run-time scheduling for fine-grained acceleration of signal processing systems," *Signal Processing Systems, 2007 IEEE Workshop on* , pp. 457–462, 17-19 Oct. 2007.
5. XViD-codec, "http://www.xvid.org."
6. C. Lucarz, M. Mattavelli, J. Thomas-Kerr, and J. Janneck, "Reconfigurable media coding: a new specification model for multemedia coders," in *Proceeding of the IEEE 2007 Workshop on Signal Processing Systems (SiPS)*, 2007.
7. S. French, *Sequencing and Scheduling*, Mathematics and its applications, Ellis Horwood Limited, 1982.
8. Y.-S. Chen, C.-S. Shih, and T.-W. Kuo, "Dynamic task scheduling and processing element allocation for multi-function socs," in *Proc. 2007 Real Time and Embedded Technology and Applications Symposium*, pp. 81–90, (Bellevue, WA), April 2007.
9. Y.-J. Kim and T. Kim, "A hw/sw partitioner for multi-mode multi-task embedded applications," *The Journal of VLSI Signal Processing* **44**, pp. 269–283, 2006.
10. Sim-Panalyzer, "http://www.eecs.umich.edu/~panalyzer."
11. D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Tech. Rep. CS-TR-1997-1342, 1997.
12. RTEMS, "http://www.jwhitham.org.uk/simplescalar/."
13. Intel-Corporation, "Intel strongarm sa-1100 microprocessor for embeddedapplications, brief datasheet, 1999."

APPENDIX A.



(a) Execution time



(b) Microarchitecture power dissipation

Figure 5. Execution time and microarchitecture power dissipation during initialization stage.

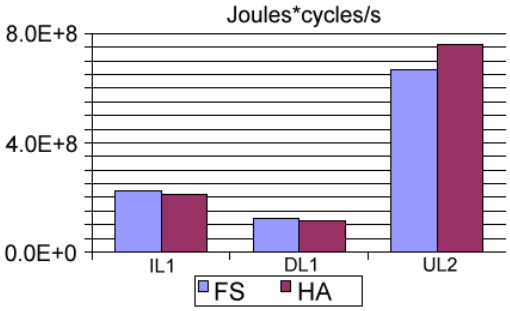
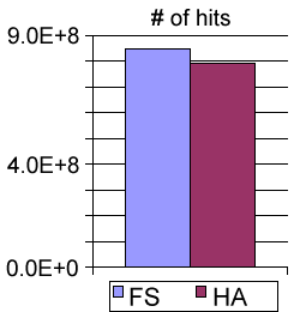
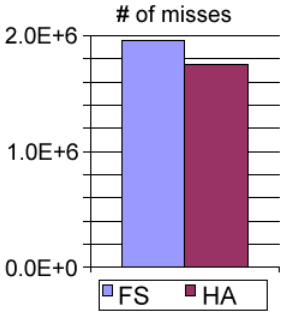


Figure 6. Power dissipation in level 1 and 2 caches during initialization stage.

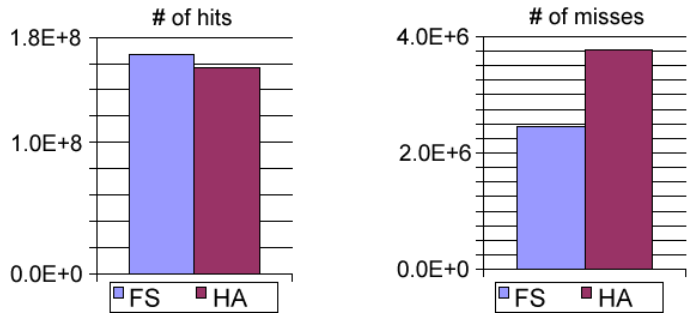


(a) IL1 hits



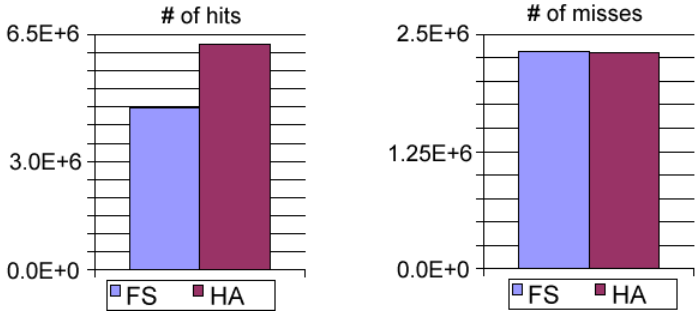
(b) IL1 misses

Figure 7. Instruction level 1 cache hits and misses during initialization stage.



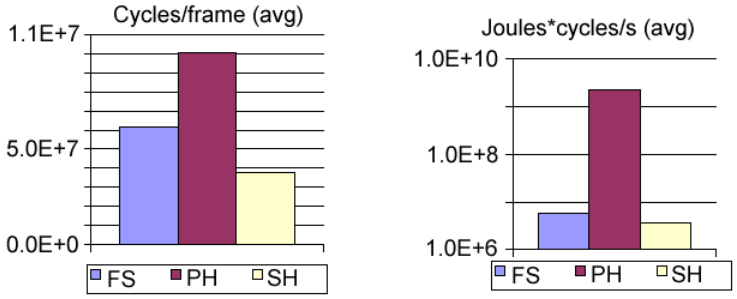
(a) DL1 hits (b) DL1 misses

Figure 8. Date level 1 cache hits and misses during initialization stage.



(a) UL2 hits (b) UL2 misses

Figure 9. Unified level 2 cache hits and misses during initialization stage.



(a) Execution time (b) Microarchitecture power dissipation

Figure 10. Execution time and microarchitecture power dissipation on average for decoding one frame.

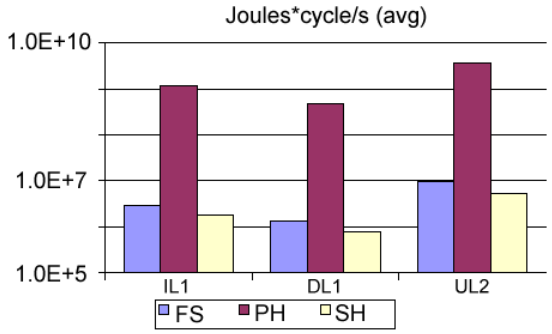
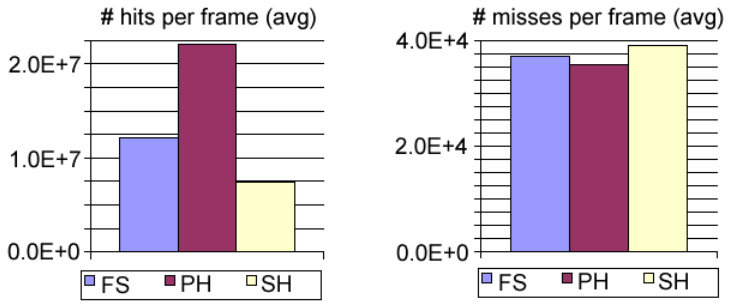
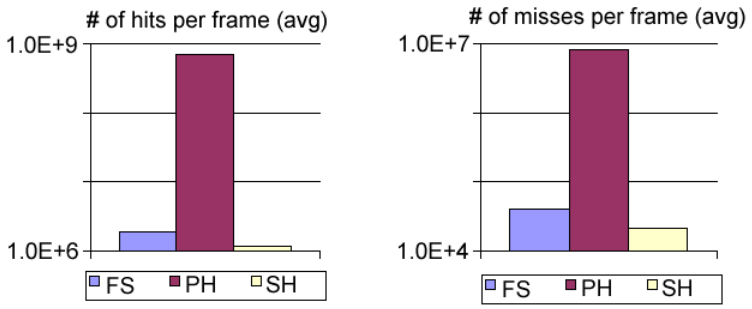


Figure 11. Power dissipation in level 1 and 2 caches on average for decoding one frame.



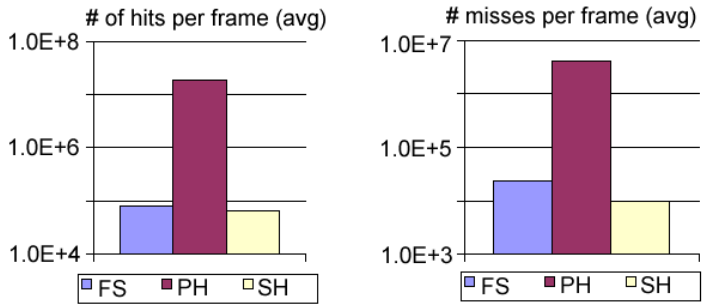
(a) IL1 hits (b) IL1 misses

Figure 12. Instruction level 1 cache hits and misses on average for decoding one frame.



(a) DL1 hits (b) DL1 misses

Figure 13. Date level 1 cache hits and misses on average for decoding one frame.



(a) UL2 hits (b) UL2 misses

Figure 14. Unified level 2 cache hits and misses on average for decoding one frame.

Paper V

Interrupt Costs in Embedded System with Short Latency Hardware Accelerators

Sébastien Lafond and Johan Lilius

Originally published in: proceedings of the *International Conference on Engineering of Computer-Based Systems - ECBS 2008*, pages 317-325. IEEE, 2008

Interrupt Costs in Embedded System with Short Latency Hardware Accelerators

Sébastien Lafond
Turku Centre for Computer Science, Embedded Systems Laboratory
sebastien.lafond@abo.fi

Johan Lilius
Åbo Akademi University, Centre for Reliable Software Technology
johan.lilius@abo.fi

Abstract

The current trend in handheld devices is to provide users with various embedded multimedia applications. Architecture developers have to use dedicated hardware accelerators to meet the timing requirements of these new applications. For physical and economical reasons the use of dedicated monolithic hardware accelerators is impractical. Instead, because the multimedia applications share common functionalities, monolithic hardware accelerators can be split into smaller accelerators to remove redundancy and save on silicon area. Unfortunately, lowering the granularity of accelerators increases synchronization calls between the main processor and the accelerators.

This paper presents a methodology for analyzing the impact of short latency hardware accelerators on a typical embedded system. We show that hardware accelerator granularity has a direct effect on system performance in terms of cache misses, execution time and thus energy consumption.

1. Introduction

Handheld devices integrate more and more functionality, and providing more multimedia applications is becoming a de facto requirement. Solutions are therefore needed for accelerating these computationally intensive applications in order to fulfill the requirements. The main acceleration approaches can be classified into two categories [8]:

1. A short portion of code is accelerated by extending the processor instruction set with a corresponding instruction. In this case the new instruction has a typical execution latency from 1 to 4 cycles, thus limiting the size of the accelerated software. Developing longer instruction would make the pipeline execution flow inefficient.
2. A full application functionality is accelerated with a monolithic hardware accelerator used as peripheral device. The hardware accelerator is then synchronized with the application by the means of interrupts. In this case the hardware accelerator has a typical execution latency from several thousand cycles up to several hundreds of thousands of cycles. However dedicated monolithic hardware accelerators are onerous to achieve due to physical and economical constraints.

The use of fine grained hardware accelerators has the advantage of saving silicon area by allowing collaborative use of common accelerated functionalities among several applications, thus cutting down implementation redundancy over several accelerators. For example, applications using reconfigurable media coding (RMC) [3] [2], where arbitrary combinations of algorithms may be assembled without predefined standardization, could easily take advantage of collaborative use of common accelerated functionalities. Also one could accelerate the time consuming DCT function in a MPEG4 video decoder and share the created accelerator with a JPEG decoder application. In such a case an access management system or dedicated scheduler is needed in order to avoid blocking state when two tasks would request the use of an accelerator at the same time. The study of such access management system or dedicated scheduler is however beyond the scope of this paper. This would restrict our study to a specific set of applications while we are here exclusively interested in analyzing the impact of short latency hardware accelerators on a typical embedded system.

Splitting a monolithic hardware accelerator into several fine grained hardware accelerators can also in some cases permit a pipelined execution of the accelerators. Nevertheless, it transfers control complexity to the software running on the processor and as a consequence increases the synchronization frequency between the accelerators and the processor. Synchronization between an accelerator and the processor is needed to inform the processor about execution

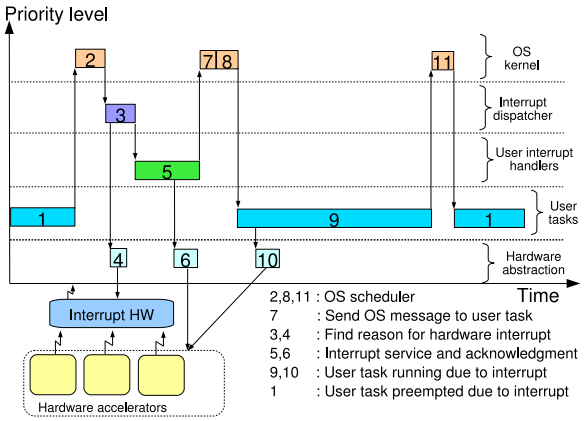


Figure 1. Execution sequence for accelerator synchronization [9]

termination of the accelerator. As a result, the use of short latency hardware accelerators tends to amplify the interface cost used for synchronization between the processor and the hardware accelerators.

A hardware accelerator is typically used as a peripheral device and synchronized with the OS running on the processor with an interrupt [9]. Figure 1 shows the sequence of operations needed in a multitasking environment when an interrupt instructs the OS about the termination of an accelerator task. Task 9 previously called a hardware accelerator. After termination of the accelerator execution an interrupt is triggered by the accelerator which will wake up the calling task 9 in order to fetch the computed results.

In a single tasking environment this extra synchronization cost is limited to the execution of the interrupt mechanism and handler as no scheduler is needed. But in a multitasking environment the extrinsic (intra-task) cache behavior will be affected by the synchronization mechanism. Each time an accelerator is called the content of the cache is changed by the new scheduled task running during the accelerator execution. This will result in a performance lost called the *cache refill penalty* [7] [6] each time an accelerator is called and an interrupt is triggered. The cache refill penalty is due to an increase of cache misses each time a context switch is performed. It introduces an increase in execution cycles and energy consumption since a cache miss leads to more bus and main memory activity. The cache refill penalty could be reduced by using various cache partitioning approaches where the cache is logically divided into multiple partitions and each partition is exclusively accessed by a single task [12] [5]. However such partitioning techniques are relevant only in the case the number of tasks is fixed and completely defined for the whole system life time. In the case of reconfigurable media coding applications, where arbitrary combinations of algorithms may be used, cache partitioning approaches would require one cache partition for each algorithm combination. This would request an unreasonable total cache size.

Moreover, since the pace of instruction execution speeds up much faster than main memory access time, the cache refill penalty has increased and will in the future continue to increase along with the difference between processor and memory speed.

The major contribution of this paper is the establishment of a simulation framework showing the overall cost due to interrupts used for synchronization between the hardware accelerators and the processor on a typical embedded system. This overall cost is composed by (a) a direct cost due to the use of hardware accelerators as peripheral devices and (b) indirect cost due to the cache refill penalty.

The methodology presented in this paper can be re-used with other platform configuration for evaluating the granularity range of new hardware accelerators which will provide a good trade off between implementation redundancy and synchronization cost.

The rest of this paper is organized as follow: In Section 2 we present the simulation framework established for this study. Section 3 gives the simulation parameters used to run the simulation framework, section 4 evaluates our results and section 5 concludes the paper.

2. Simulation framework

The simulation framework presented in this section models a typical handheld device featuring basic multimedia applications. It includes a hardware platform, an operating system and a set of applications and hardware accelerators.

The Sim-Panalyzer [10] processor simulator is used for this study. Sim-Panalyzer is based on the SimpleScalar [1] processor simulator and performs cycle accurate simulation of a strongARM SA-1100 processor. It computes at every simulated cycle the energy consumption of each module constituting the ARM core (clock, alu, cache, etc.). Such processor simulator permits the execution of an operating system ported on ARM architecture.

As RTEMS 4.6.2 has been ported onto SimpleScalar by Jack Whitham [11], RTEMS was chosen as the real-time operating system for this study. This port includes a SimpleScalar extension for supporting an interrupt based programmable timer which is needed by RTEMS. RTEMS is a free open source real-time operating system designed for embedded systems and supporting a variety of application programming interfaces (APIs) and interface standards. This real time operating system allows us to execute a set of applications as independent tasks in a pre-emptive multitasking environment, a prerequisite for our simulation.

Table 1. Selected functions to be accelerated

Application	Chosen functions	Nb of calls
GSM coder	APCM_quantization()	532
GSM decoder	GSM_RPE_Decoding()	532
JPEG comp.	forward_DCT()	128
JPEG decomp.	h2v2_fancy_upsample()	512

A set of 4 applications are chosen from the MiBench benchmark suite [4]. These applications are present on typical handheld devices: a GSM audio coder, a GSM audio decoder, a JPEG compressor and a JPEG decompressor. For each application an execution time profiling was carried out in order to identify the most time consuming functions. Out of this profiling some functions were selected to be executed on dedicated hardware accelerators. Table 1 shows the selected functions and the number of times the functions are called. Each application is implemented as a task running on the OS. An idle task with low priority is also implemented and is executed in the case all other tasks are waiting for their hardware accelerators to terminate.

The presented applications and hardware accelerators define our reference environment. In addition to this reference environment a fifth application was implemented. This last application will be called the *exploration application*, and will be used to explore the impact of a short latency hardware accelerator synchronized by interrupts on the overall performance of the system. The exploration application can be seen as an added task disturbing the reference environment.

Figure 2 represents the parameters influencing the execution pattern of the exploration application in a single task environment. Executed in the pre-emptive multitasking environment of our simulation framework, the OS will schedule other tasks to run during the suspended state of the exploration application. The exploration application is the task used for measuring the cost of short latency hardware accelerators. One hardware accelerator with variable latency is associated with the exploration application. Thus the simulation framework requires two parameters: (a) the length in cycles of execution performed before a call to the hardware accelerator is done (see Figure 2) and (b) the latency in cycles of its associated accelerator. When the accelerator execution terminates the exploration application will be scheduled by the OS to run depending on its priority and the priority of other tasks.

The complete simulation framework now consists of five tasks and their respective hardware accelerators. Figure 3 shows the system architecture used for this study. The five tasks running on the RTEMS operating system need to communicate with their corresponding hardware accelerators. For each accelerator a new system call is assigned and Sim-Panalyzer is modified to catch these five new system calls.

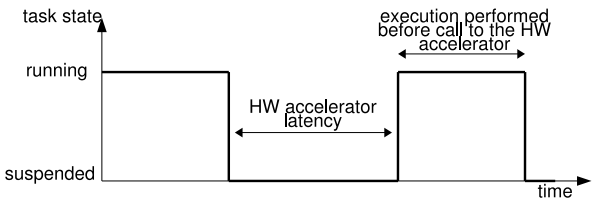


Figure 2. The exploration application in single task environment

Figure 4 represents the sequence of executed operations following a hardware accelerator call. These operations are explained as the following:

1. Sim-Panalyzer reads the possible parameters from defined registers and executes the hardware accelerator job. Then it writes the possible results on defined registers.
2. Sim-Panalyzer sets the corresponding interrupt flag valid in X cycles, X being the accelerator latency.
3. RTEMS suspends the calling task by changing its priority to a low level, making the task non-executable.
4. RTEMS schedules the remained non-suspended tasks to run.
5. The interrupt handler will acknowledge the triggered interrupt and call the OS to resume the corresponding task by restoring its previous priority.
6. Sim-Panalyzer gets the interrupt acknowledgement and clears the associated flags.
7. RTEMS schedules all non-suspended tasks to run.

It is important to note that the hardware accelerator jobs are executed within the Sim-Panalyzer simulator, which means that their executions are performed outside the simulated platform. The hardware accelerator execution costs, including possible data transfer between the processor and accelerators, are thus not taken into account in this study. This omission does not affect the measurements because the cost due to the use of interrupts and the indirect cache refill

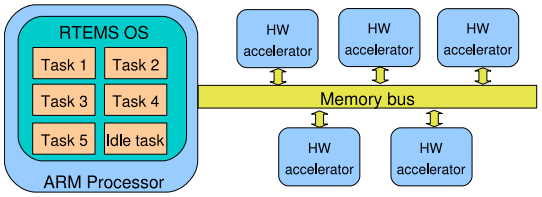


Figure 3. System architecture

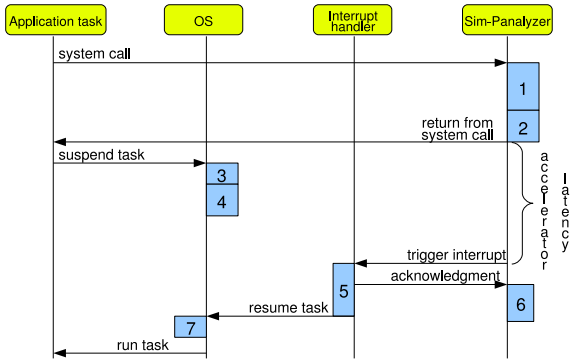


Figure 4. Sequence of operations

penalty cost are not affected by the internal accelerator activities or read/write operations initiated by the accelerator.

3. Simulation parameters

The simulation parameters bind the simulation framework within a defined execution window. This section defines the constant and variable parameters and their corresponding values used in the simulation framework.

3.1 Sim-Panalyzer

Sim-Panalyzer defines the processor parameters and the configuration of the caches. For this study the processor speed was set at 233 MHz. The configuration for the level 1 instruction cache, level 1 data cache and the unified level 2 cache is presented on table 2. Table 3 shows the different latencies for each memory level. This configuration tries to target an average embedded system performance that could be used for a multimedia handheld device. The relatively small level 1 and 2 caches compensate for the relatively small footprint of the benchmark applications (see *Applications and HW accelerators* subsection). All other parameters used by Sim-Panalyzer were set to their default values.

3.2 RTEMS

RTEMS has a few parameters influencing the timing behavior of the executed tasks. For this study all tasks, except the idle task, have their priority levels set to 10. The idle

Table 2. Caches configuration

Caches	Associativity	Size	Nb blocks	Block Size
il1	direct mapped	4 Kb	128	32 bytes
dl1	direct mapped	4 Kb	128	32 bytes
ul2	4-way	8 Kb	256	32 bytes

Table 3. Memory Latencies with a clock frequency of 233Mhz

	il1	dl1	ul2	main memory *first chunk access*	main memory *inter chunk access*
Latency in cycles	2	2	6	30	4

task has a priority of 20. Following a call to a hardware accelerator the task priority level is changed to 250, making the task un-executable by the RTEMS scheduler. The interrupt handler will restore the task priority level to 10 when the corresponding interrupt is triggered. The preemptive execution is activated and the time slice is set to 5 ticks, one tick representing one hundredth of a second. In absence of hardware accelerator interrupt, the scheduler is then set to run 20 times per second.

3.3 Applications and HW accelerators

The hardware accelerators used by the applications defining the reference environment have fixed execution latencies and table 4 shows their latency values in cycles. As input data, the jpeg compression and decompression application process a 512 by 512 pixels image, and the GSM encoder and decoder process a 2 seconds 8-bit audio signal. The executable file containing the 5 applications and the idle task consists of 370kB of instructions and 17kB of data.

Table 4. Hardware accelerator latencies

Application	Accelerator latency
GSM coder	2500 cycles
GSM decoder	2500 cycles
JPEG comp.	3000 cycles
JPEG decomp.	2500 cycles

The exploration application implements an empty loop in ARM assembly code followed by a system call to its hardware accelerator.

With such rudimentary implementation the exploration application is used to look into the effect of hardware accelerator granularity on the overall system running several other applications. This implementation has a very small instruction and data footprint in order to interfere as little as possible with the cache behavior. At first only the impact of having different hardware granularities is studied. The hardware granularity is adjusted by dividing at the same time the loop length and the accelerator latency by a multiple of 2. Splitting the hardware accelerator into 2 independent smaller accelerators is thus simulated by dividing

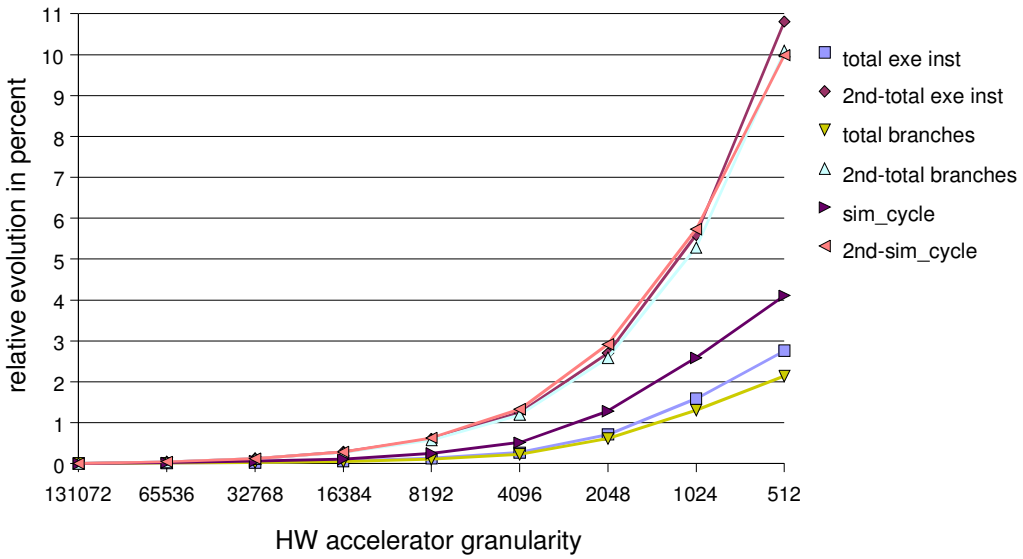


Figure 5. Influence of granularity on execution time

the exploration application loop length and its accelerator latency by 2.

A second measurement series is performed with the exploration application having data accesses after the loop. Data accesses are obtained by copying in an array of 200 integers the n^{th} array value into the *n-1* location. Adding data accesses in the exploration application is done in order to obtain a more realistic application behavior.

4. Results

We run the simulation framework with different granularities for the hardware accelerators of the exploration application. For each granularity the loop length of the exploration application is set to the accelerator latency value. For all measurements nine different granularities are used: from the coarse-grained system having 10 calls to a 131072 cycles hardware accelerator latency to the fine-grained system calling 2560 times a hardware accelerator with a latency of 512 cycles. We express the granularity in term of cycles: a granularity of 131072 represents a system having one hardware accelerator with a latency of 131072 cycles and called in our experiment 10 times. In the same way, a granularity of 65536 represents a system with one hardware accelerator having a latency of 65536 cycles and called 20 times. In our experiment a granularity of 65536 is then equivalent to a system with two hardware accelerators being called 10 times and both having a 32768 cycles latency, introducing a split coefficient of 2.

As we can see from Figure 4, if the latency is too short the accelerator will trigger an interrupt before the RTOS finished to suspend the calling task, which will result in deadlocking the calling task. In our simulation framework the fastest accelerator we are able to simulate is an accelerator with a 512 cycles latency. This indicates the mechanism for lowering a task priority in RTEMS (sequence 3 on Figure 4) takes less than 512 cycles.

All results presented in this section are relative measurements using the values obtained for the coarse-grained system as reference. In other words, all figures trace the relative evolution in percent of the measured elements compared to the values obtained for the system having a granularity of 131072. The figure legends indicate the results for the second measurement series with the term *2nd*.

Figure 5 presents for the two exploration applications the evolutions of the number of executed instructions, taken branches and total execution time in cycles for all the tasks running on the system. For the two exploration applications the three measurements show an exponential increase when the hardware granularity is reduced. For the exploration application having data accesses the total execution time increases by about 11% with the finest grained accelerator compared to the coarse-grained implementation. This is about twice of the increase obtained by the rudimentary exploration application. This difference can be explained by an increase in data cache misses, being presented afterward, due to the extra data accesses.

Figure 6 shows for the two exploration applications the granularity influence on level 1 instruction cache. On Figure 6 *il1.pdissipation* represents the total energy dissipated by the cache. The two exploration applications present similar results concerning the number of cache misses, with an increase of almost 25% when fine grained accelerators are used. This is explained by the fact that the two ex-

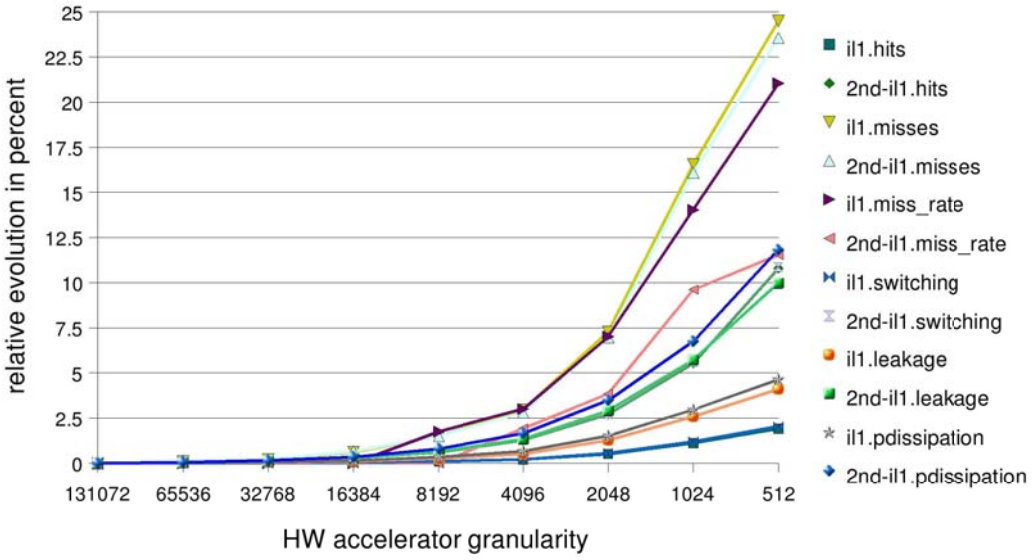


Figure 6. Influence of granularity on level 1 instruction cache

ploration applications have little difference in term of instruction footprint. Indeed only very few instructions were added in order to implement extra data accesses. However the number of cache hits is bigger with the second exploration application due to the added data accesses. Thus, as the number of cache misses is equivalent for the two exploration applications, the cache miss rate for the second exploration application is increasing at a slower rate. The leakage current is a static cost in a memory and its resulting energy consumption is only dependent on the time the memory is in use. This explains why the energy, *il1.leakage*, wasted due to leakage current inside the cache is increasing at the same rate than the increase of the total execution time. On the other hand the energy consumption due to switching activities in a memory is dependent on the number of occurring read and write accesses. As a general comment, one can say that decreasing the hardware granularity will have a relatively large impact on the level 1 instruction cache miss rate with a 25% increase for the finest grained accelerator compared to the coarse-grained implementation.

The granularity influence on the level 1 data cache for the two exploration applications is presented on Figure 7. Adding data accesses to the exploration application results in a roughly 30% cost increase compared to the rudimentary exploration application implementation. As the number of misses and hits increases in a similar proportion in the level 1 data caches, the value for the cache miss rate stays equal for the two exploration applications. Figure 7 shows that for the finest grained accelerator we obtain a 20% cache miss and 10% energy consumption increase in the level 1 data cache for the second exploration application. As a general comment we observe that decreasing the hardware granularity will primarily introduce an increase in level 1 data misses, thus driving an increase of the cache energy consumption.

Figure 8 shows for the two exploration applications the influence of granularity on the unified level 2 cache. An interesting observation is the drop of the unified level 2 cache miss rate with the reduction of hardware granularity. The miss rate decrease is due to a fast increase of the number of cache hits while the number of cache misses stays almost constant. For the two exploration applications, reducing the hardware granularity increases significantly the number of cache hits which raises the energy consumption because of the increase in switching activity.

As a general comment we can say that decreasing the hardware accelerator granularity on a typical embedded system relying on interrupt mechanism for synchronization between the accelerators and the processor will slow down the system execution time not only by increasing the number of instructions to execute but also by increasing the level one and two cache hits and misses. On the simulated system, cost increases start to be seen from the granularity level 4096 which correspond to splitting the reference hardware accelerator into 32 separate smaller accelerators.

4.1 Implication of the results

For the two exploration applications we also measured the average time needed to perform the synchronization between the accelerators and the processor. For the simple exploration applications each synchronization costs on average an extra 1375 cycles, with a standard deviation of 365 cycles, while for the second exploration application it costs

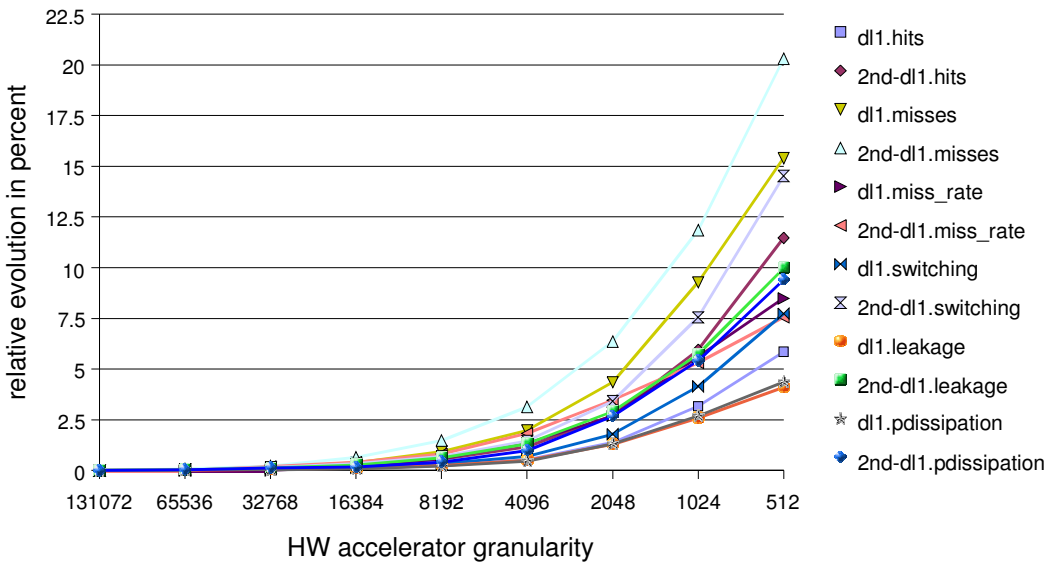


Figure 7. Influence of granularity on level 1 data cache

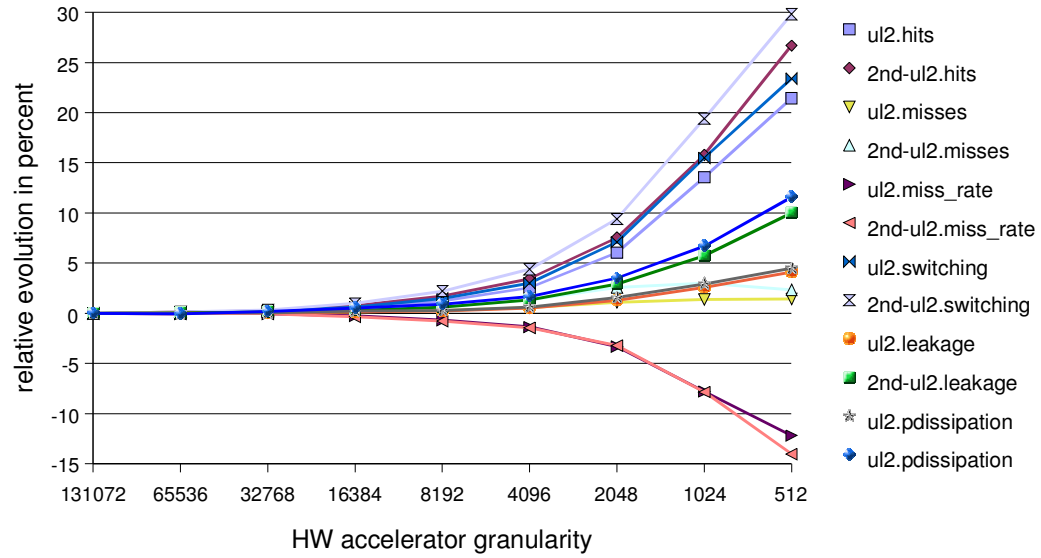


Figure 8. Influence of granularity on level 2 unified cache

on average an extra 3250 cycles with a standard deviation of 315 cycles. The cost difference between the two exploration applications is explained by the increase of the cache refill penalty in the second exploration application.

If one extracts a discrete cosine transform out of a video decoder implementation and use it as a hardware accelerator decoding a QVGA (320x240 pixels) video at 25 frames per second, the hardware accelerator will be called 45 000 times per second. Assuming on average a total cost of 3250 cycles per call, 145 Millions processing cycles per second will be wasted due to the overhead introduced by the synchronization mechanism between the processor and hardware accelerator.

5. Conclusion

In this study we presented a methodology for analyzing the impact of short latency hardware accelerators on a typical embedded system. The presented methodology can be re-used with other platform configurations for evaluating the granularity range of new hardware accelerators which will provide a good trade off between implementation redundancy and synchronization cost.

We demonstrated that when approaching the bottom line imposed by the RTOS speed in the mechanism of suspending a task, decreasing the hardware accelerator granularity will introduce relatively important extra costs in terms of cache misses, execution time and energy consumption. Therefore using a hardware accelerator can become inefficient if the accelerator latency is too short. This is due to an increase of the number of instructions to execute and number of level one and two cache hits and misses. Reducing the OS mechanism speed used in synchronization between the accelerators and the processor will push down the minimum execution latency the accelerators can have, but will also considerably increase the system execution time and energy consumption. As a direct consequence, the additional cost due to the synchronization introduced by the fine grained hardware accelerator will be in some case preponderant on the gain obtained by the accelerated software.

If one needs to implement a frequently used hardware accelerator that has a short latency, an original "interrupt and context switch free" synchronization mechanism needs to be used in order to provide an efficient solution.

6. Acknowledgement

We are thankful to professor Olli Silvén for his helpful comments concerning this study.

References

[1] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, 1997.

[2] C. L. et al. Reconfigurable media coding: a new specification model for multimedia coders. In *Proceeding of the IEEE 2007 Workshop on Signal Processing Systems (SiPS)*, 2007.

[3] J. T.-K. et al. Reconfigurable media coding: Self-describing multimedia bitstream. In *Proceeding of the IEEE 2007 Workshop on Signal Processing Systems (SiPS)*, 2007.

[4] M. R. G. et al. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[5] F. Mueller. Compiler support for software-based cache partitioning. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers & tools for real-time systems*, pages 125–133, New York, NY, USA, 1995. ACM.

[6] F. Sebek. The real cost of task pre-emptions — measuring real-time-related cache performance with a hw/sw hybrid technique. Technical report, Mälardalen Real-Time Research Centre, Department of Computer Science and Engineering, Mälardalen University, Sweden, Aug. 2002.

[7] F. Sebek and J. Gustafsson. Determining the worst case instruction cache miss-ratio. In *Proceedings of Workshop On Embedded System Codesign (ESCODES'02)*, San Jose, California, USA, 24, 2002.

[8] O. Silven and K. Jyrkkä. Observations on power-efficiency trends in mobile communication devices. *EURASIP Journal on Embedded Systems*, 2007.

[9] O. Silven, T. Rintaluoma, and K. Jyrkkä. Implementing energy efficient embedded multimedia. In *Multimedia on Mobile Devices II, Proceedings of the SPIE, Volume 6074*, 2006.

[10] Sim-Panalyzer. http://www.eecs.umich.edu/˜panalyzer.

[11] J. Whitham. http://www.jwhitham.org.uk/simplescalar/.

[12] A. Wolfe. Software-based cache partitioning for real-time applications. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems*, 1993.

Paper VI

Static Energy Saving Through Multi-Bank Memory Architecture

Sébastien Lafond and Johan Lilius

Originally published in: proceedings of the *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation - ICSAMOS 2006*, pages 43-49. IEEE, 2006.

Static Energy Saving Through Multi-Bank Memory Architecture

Sébastien Lafond
Turku Centre for Computer Science
Embedded Systems Laboratory
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
Email: sebastien.lafond@abo.fi

Johan Lilius
Åbo Akademi University
Department of Information Technologies
Lemminkäinengatan 14A, FIN-20520 Åbo, Finland
Email: johan.lilius@abo.fi

***Abstract*— Managing the energy consumption of embedded systems has become a major problem with the increasing demand for portable electronic devices. This paper propose a multi-bank memory architecture as a solution to decrease the static energy cost in memory. We set up the equations ruling the optimization problem for decreasing the memory static energy cost, analyze the impact of different parameters on the energy cost and finally present some case study results.**

I. Introduction

In recent years we have seen an explosion of the market for portable electronic devices such as PDAs, personal communicators and mobile phones. They have in common strong constraints on energy consumption, and thus maximizing battery life for such devices is crucial.

Several studies [1] show that memory is becoming a predominant energy consumption component in handheld devices. As the static energy due to leakage currents is becoming the major element of memory energy consumption [2], a reduction of the static energy cost will have a significant impact on the overall system energy consumption. In traditional systems one continuous memory region is generally used to store dynamic memory allocation and its size must to be sufficiently large to hold in any case all allocations. This required size is most of the time oversized for the average allocation behavior of the application(s), leading to a waste of static energy in the memory area used only during the worst cases. In order to cut down the cost associated to this 'most of the time' unused memory area we propose a multi-bank memory architecture (MBMA) as a solution to decrease the static energy cost of the memory. Such architecture would have the ability to follow the application(s) memory need by adjusting the number of memory bank switched on. Fig. 1 simply illustrates the general behavior of such MBMA. In Fig. 1, which doesn't take into account the possible fragmentation in a bank, the maximal energy savings would be proportional to the size of the *switched off* memory area. The remain wasted static energy would be proportional to the size of *Free Memory* area.

The major contributions of this paper are: 1) the introduction of a complete static energy costs model for a multi-bank memory architecture, 2) the establishment of the equations governing optimization problem for decreasing the static energy consumed by the memory and an analysis of the impact of the different parameters on energy consumption and 3) the performance analysis of such architecture in terms of static energy consumption and execution speed.

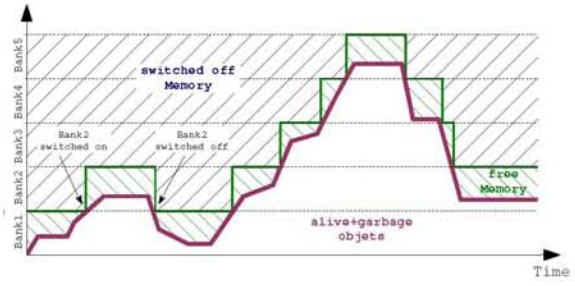


Fig. 1. MBMA behavior

The paper is organized as follow: we briefly describe software allocation behavior and the general memory model for static energy consumption. We then present the static energy consumption model for a MBMA and two reference architectures. Then we setup the optimization problem for the MBMA static energy cost and show some simulation results. We conclude with a discussion of related and future work.

II. Allocation behavior

Dynamic memory allocation is the assignment of memory block(s) to store specific data used during the runtime of an application. A dynamically allocated memory block remains in $allocated\ state$ until it is explicitly deallocated or implicitly deallocated by the means of a garbage collector (GC). Before a block is deallocated it can be in an intermediate state, called garbage, where the block content is no longer needed by the application.

The software applications(s) is driving the allocation t_a, garbage t_g and deallocation t_d events occurring in the memory. We define the variables $free$, $alive$ and $garbage$ as the total memory size of blocks in respectively free, alive and garbage state. For each new event concerning b blocks of size S Table I presents the variables updates triggered by the event.

When different size objects are dynamically allocated, a deallocated block might let an free and unuseable space for the following allocations. In that case this space contributes to memory fragmentation which denotes a waste of the memory.

1-4244-0155-0/06/$20.00 ©2006 IEEE

TABLE I

EVENTS TRIGGERING VARIABLES UPDATES

Event	Variables updates (occur simultaneously)
t_a	$free = free - b \cdot S$ $alive = alive + b \cdot S$
t_g	$alive = alive - b \cdot S$ $garbage = garbage + b \cdot S$
t_d	$garbage = garbage - b \cdot S$ $free = free + b \cdot S$

There is commonly two types of fragmentation : internal and external. Internal fragmentation refers to waste in the memory due to alignment and storage of additional information needed by the allocator such as bookkeeping. External fragmentation describes on the other hand a waste due to holes of free memory interspersed with live objects. As this study aims to analyze the external fragmentation in the context of MBMA, in the remain of this paper the term fragmentation means external fragmentation.

There is several ways to evaluate memory external fragmentation. One can measure it as a percentage of the actual memory usage or as a percentage of the amount of live objects. This late approach has been used by Johnstone and Wilson in [3]. However their four proposed ways to measure fragmentation require knowledge about the past allocation behavior which demand extra storage place if the system has to manage by itself on the pertinence to launch a memory compaction. In the context of MBMA we propose to compute the instantaneous fragmentation for each bank as:

$$fragmentation = 1 - \frac{largest\ free\ area}{total\ free\ area} \quad (1)$$

With this evaluation a bank containing all its free memory in one continuous area has a fragmentation of 0. If the largest free area tends to be relatively small compare to the total free area the bank fragmentation will approach 1. Additionaly a fully allocated bank is considered to have fragmentation of 0. This fragmentation evaluation method has several advantages: (a) it returns a value relative to the largest free block available, (b) the returned value is bounded between 0 and 1 and (c) it compute the total free memory available in the bank. These advantages can be used to precisely evaluate the state of each bank in term of fragmentation and potential memory availability for future allocations.

III. MEMORY MODEL

There are two main families of RAM technology: Static Ram (SRAM) and Dynamic RAM (DRAM). SRAM's store each bit in a memory cell that are basically flip flops build from six CMOS transistors. The static dissipation of SRAM's is due to the leakage current of each memory cell. During the idle phase the SRAM cell leakage current for a given threshold voltage and temperature will be constant. Thus the static energy consumed by an SRAM during idle time is proportional to its number of cells [4]. DRAM's stores each bit in a memory cell consisting of one capacitor and one transistor. In addition to the transistor leakage currents the DRAM static energy consumption should also include the energy dissipated in order to refresh the cells. Assuming that there is a $\frac{1}{2}$ probability for a cell to hold 1, the energy needed in order to refresh the memory cells will be proportional to half memory size.

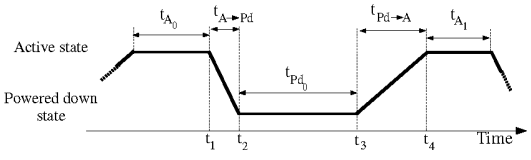


Fig. 2. Memory state transitions

A the static energy cost for a RAM or DRAM memory is proportional to its size, for a determined technology the average static power dissipated by a memory can be modeled by the following equation :

$$\overline{P}_S = k \cdot Size \quad (2)$$

where $\overline{P}$ represents the average static power dissipated by the memory, $Size$ the size of the memory and k a constant factor depending on the memory technology and hardware implementation.

We adopt a two-state memory model consisting of one active and one powered down state. In active state the memory can be accessed for read and write operation and dissipates an average static power of $\overline{P} = k \cdot Size$. In powered down state data retention is not required and the memory does not consume any energy.

Fig. 2 represents the memory state transitions and define $t_{A \to Pd}$ and $t_{Pd \to A}$ respectively the time needed to switch from state active into powered down and conversely. From Fig. 2 we assume that the initial and last state of the memory is powered down and the memory has been in active state N times. We can then define T_A, T_{Pd}, $T_{A \to Pd}$ and $T_{Pd \to A}$ as:

$$T_A = \sum_{i=0}^{N} t_{A_i} \qquad T_{Pd} = \sum_{i=0}^{N} t_{Pd_i}$$
$$T_{A \to Pd} = N \cdot t_{A \to Pd} \qquad T_{Pd \to A} = N \cdot t_{Pd \to A}$$

where T_A represents the total time during which the memory was in active state, T_{Pd} the total time during which the memory was in powered down state , $T_{A \to Pd}$ the total time during which the memory was in transition phase from active state into powered down state, and $T_{Pd \to A}$ the total time during which the memory was in transition phase from powered down state into active state.

IV. MBMA MODEL FOR STATIC ENERGY COST

The energy cost model for a MBMA consisting of B banks has the following parameters: the average static power dissipated by the i^{th} bank $\overline{P}_{S_i}$, the total time during which the i^{th} bank was powered on in active state T_{A_i}, the average static power dissipated by the i^{th} bank in transition phase

from active state into powered down state $\overline{P}_{(A\rightarrow Pd)_i}$ and from powered down state into active state $\overline{P}_{(Pd\rightarrow A)_i}$, the total time during which the i^{th} bank was respectively in transition phase from active into powered down state $T_{(A\rightarrow Pd)_i}$ and vice versa $T_{(A\rightarrow Pd)_i}$. The static energy cost model for a MBMA composed of B banks is then defined by:

$$\begin{aligned} Es_{total} &= \sum_{i=1}^{B}[(\overline{P}_{S_i}\cdot T_{A_i}) + (\overline{P}_{(A\rightarrow Pd)_i}\cdot T_{(A\rightarrow Pd)_i}) \\ &\quad +(\overline{P}_{(Pd\rightarrow A)_i}\cdot T_{(Pd\rightarrow A)_i})] \end{aligned} \quad (3)$$

During transition phases the instantaneous dissipated power will most likely not be constant. In order to simplify the expression we use the average power $\overline{P}_{(A\rightarrow Pd)_i}$ and $\overline{P}_{(Pd\rightarrow A)_i}$ instead of the respective literal expressions $\int_{t_1}^{t_2} P_{A\rightarrow Pd}(t)\cdot dt$ and $\int_{t_3}^{t_4} P_{Pd\rightarrow A}(t)\cdot dt$. This simplification doesn't change the correctness of Es_{total} definition if we consider that $t_{A\rightarrow Pd}$, $t_{Pd\rightarrow A}$, $P_{A\rightarrow Pd}(t)$ and $P_{Pd\rightarrow A}(t)$ are equal for each active to powered down and powered down to active transitions. With $BankSize_i$ standing for the i^{th}bank size we have then :

$$\begin{aligned} Es_{total} &= \sum_{i=1}^{B}[(k\cdot BankSize_i\cdot T_{A_i}) \\ &\quad +(\overline{P}_{(A\rightarrow Pd)_i}\cdot T_{(A\rightarrow Pd)_i}) \\ &\quad +(\overline{P}_{(Pd\rightarrow A)_i}\cdot T_{(Pd\rightarrow A)_i})] \end{aligned} \quad (4)$$

For each instant t, $BankSize_i$ can be further refined as :

$$BankSize_i = alive(t)_i + [(free(t)_i + garbage(t)_i)] \quad (5)$$

During power down states $alive(t)$, $free(t)$ and $garbage(t)$ are considered to be null. During the transition $t_{(Pd\rightarrow A)_i}$, $free(t)_i = BankSize_i$ and $alive(t)_i = garbage(t)_i = 0$. During transition $t_{(A\rightarrow Pd)_i}$ values for $live(t)_i$, $dead(t)_i$ and $free(t)_i$ are considered constant and equal to their respective last active state value. Thus $BankSize_i\cdot T_{A_i}$ in (4) can also be expressed by :

$$\begin{aligned} BankSize_i\cdot T_{A_i} &= \int_0^{T_{A_i}} alive(t)_i\cdot dt \\ &\quad +\int_0^{T_{A_i}} free(t)_i\cdot dt \\ &\quad +\int_0^{T_{A_i}} garbage(t)_i\cdot dt \\ &= Alive_i + Free_i + Garbage_i \end{aligned} \quad (6)$$

And Es_{total} from (4) can be re-expressed as :

$$\begin{aligned} Es_{total} &= \sum_{i=1}^{B}[(k\cdot(Alive_i + Free_i + Garbage_i)) \\ &\quad +(\overline{P}_{(A\rightarrow Pd)_i}\cdot T_{(A\rightarrow Pd)_i}) \\ &\quad +(\overline{P}_{(Pd\rightarrow A)_i}\cdot T_{(Pd\rightarrow A)_i})] \end{aligned} \quad (7)$$

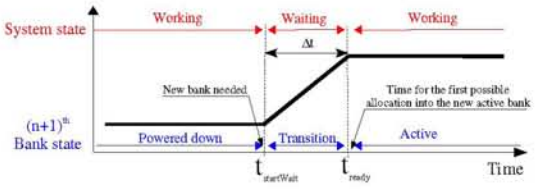


Fig. 3. Waiting state

During the system life the MBMA can be in two different states : *waiting state* and *working state*. A waiting state, as illustrated on Fig. 3, occurs when the system needs to allocate a new object on the memory but all switched on banks are full. In that case, if the system was not able to anticipate this allocation need beforehand, the system will have to wait for a time $\triangle t$ until a new bank has been switched on. The waiting cost is expressed by :

$$E_{wait} = \sum_{i=1}^{n}[BankSize_i\cdot k\cdot\triangle t] + \overline{P}_{wait}\cdot\triangle t \quad (8)$$

where E_{wait} represents the energy cost wasted during the waiting state and $\overline{P}_{wait}$ the average power dissipated during the waiting time by the $(n+1)^{th}$ bank which is in transition phase. $\triangle t$ is bounded by value 0 and $(t_{A\rightarrow Pd} + t_{Pd\rightarrow A})$ if we assume that the transition phase from active into powered down state can't be interrupted before it ends.

During the working state the MBMA has enough free available memory space for new allocation, and the need for new bank doesn't exist. With l and m representing respectively the numbers of active to powered down and powered down to active transitions occurred during T_{A_i}, $P_{Trans.}(t)$ the function describing the instantaneous power during the period from last transition request to the new transition request time , each bank is consuming during active states:

$$\begin{aligned} E_{working_i} &= k\cdot BankSize\cdot T_{A_i} \\ &\quad +l\cdot(\overline{P}_{(A\rightarrow Pd)}\cdot t_{(A\rightarrow Pd)}) \\ &\quad +m\cdot(\overline{P}_{(Pd\rightarrow A)}\cdot t_{(Pd\rightarrow A)}) \\ &\quad +\int_{t.Last}^{t_{Rpd}} P_{Trans.}(t)\cdot dt \end{aligned} \quad (9)$$

For a B banks system where W waiting states occur we can express Es_{total} as :

$$Es_{total} = \sum_{k=1}^{W} E_{wait_k} + \sum_{i=1}^{B}[E_{working_i}] \quad (10)$$

In order to simplify the model one could assume that the average power dissipated by banks during transition phases from active into powered down state, and analogously for powered-down into active state, is the same for all such transitions. In addition a conservative simplification would assign to $\overline{P}_{(A\rightarrow Pd)}$, $\overline{P}_{(Pd\rightarrow A)}$ and $\overline{P}_{wait}$ the $\overline{P}_S$ value representing the upper bound for the functions $P_{(A\rightarrow Pd)(t)}$, $P_{(Pd\rightarrow A)}(t)$ and $P_{wait}(t)$. In the same way $\triangle t$ can be simplified by its

maximal value $(t_{A\rightarrow Pd} + t_{Pd\rightarrow A})$. From (4) Es_{total} can then be re-expressed as:

$$Es_{total} = \sum_{i=1}^{B}[(k \cdot BankSize_i \cdot (T_{A_i} + T_{(A\rightarrow Pd)_i} + T_{(Pd\rightarrow A)_i})] = k \cdot \sum_{i=1}^{B}[\int_0^{(T_{A_i}+T_{(A\rightarrow Pd)_i}+T_{(Pd\rightarrow A)_i})} BankSize_i \cdot dt] \quad (11)$$

Based on these assumption we can simplify (8) and (9):

$$E_{wait} = k \cdot \sum_{i=1}^{n+1}[\int_{t_{LastTrans}}^{t_{Active}} (alive(t)_i + (free(t)_i + garbage(t)_i) \cdot dt]$$

$$E_{working_i} = k \cdot \int_0^{T_{A_i}} (alive(t)_i + free(t)_i + garbage(t)_i) \cdot dt + k \cdot l \cdot \int_0^{t_{A\rightarrow Pd}} (alive(t)_i + free(t)_i + garbage(t)_i) \cdot dt + k \cdot m \cdot \int_0^{t_{Pd\rightarrow A}} (alive(t)_i + free(t)_i + garbage(t)_i) \cdot dt$$

From now on, if not explicitly mentioned we will always refer to this simplified model.

V. Reference architectures

The ideal MBMA would be composed of an infinity of 1 bit size banks with the ability to be instantaneously switch on and off. Such ideal MBMA would permanently be able to adjust its memory size (i.e. the total size of the all banks that are powered on) to the exact system needs and thus reaches the obtainable minimum static energy consumption due to leakage current without any additional time penalty. This ideal system is reducing the functions $free(t)_i$ and $garbage(t)_i$ as well as the value of $t_{(A\rightarrow Pd)_i}$ and $t_{(Pd\rightarrow A)_i}$ to the constant zero. The ideal model can be modeled by the following equations :

$$free(t) = garbage(t) = O \quad (12)$$

$$Es_{total} = k \cdot \sum_{i=1}^{\infty}[\int_0^{T_{A_i}} alive(t)_i \cdot dt] \quad (13)$$

$$E_{wait} = 0 \quad (14)$$

$$E_{working_i} = k \cdot \int_0^{T_{A_i}} alive(t)_i \cdot dt \quad (15)$$

This theoretically best solution to reduce static energy cost can't be obtained for evident physical constraints, but we will use it as a reference.

Compare to this idealistic architecture, a 'real life' MBMA has three additional costs: the sum of E_{wait} for all waiting states, E_{wasted_i} representing the static energy consumed by $Free$ and $Garbage$ memory area during all system working states for the i^{th} bank and E_{trans_i} representing the static energy consumed by state transition during working states for the i^{th} bank. For each active state E_{wasted_i} and E_{trans_i} can be expressed using the simplified model by :

$$E_{wasted_i} = k \cdot \int_0^{T_{A_i}} (free(t)_i + garbage(t)_i) \cdot dt \quad (16)$$

$$E_{trans_i} = k \cdot l \cdot \int_0^{t_{A\rightarrow Pd}} (alive(t) + free(t) + garbage(t)) \cdot dt + k \cdot m \cdot \int_0^{t_{Pd\rightarrow A}} (alive(t) + free(t) + garbage(t)) \cdot dt \quad (17)$$

If during the system life time the MBMA will be J times in waiting state the total extra costs E_{extra} compare to the idealistic architecture reference can be expressed by :

$$E_{extra} = \sum_{m=1}^{J}[E_{wait}^m] + \sum_{i=1}^{B}[E_{wasted_i} + E_{trans_i}] \quad (18)$$

where E_{wait}^k represents the costs due to the k^{th}waiting state, E_{wasted_i} and E_{trans_i} respectively the wasted energy and transition energy consumed in i^{th} bank during all active states. Thus :

$$Es_{total} = E_{extra} + k \cdot \sum_{i=1}^{B}[\int_0^{T_{A_i}} alive(t)_i \cdot dt] \quad (19)$$

The second interesting architecture reference to compare with is the architecture consisting of only one region memory to hold all dynamic allocations. This traditional architecture, which can also be seen has an one bank architecture, has the advantage to completely eliminate the waiting cost E_{wait} and E_{trans} but to the detriment of E_{wasted}. Indeed in that case the memory size needs also to match with the worst case allocation scenario and thus most likely drives a much greater $free(t)$ function compare to the one achievable with a MBMA.

VI. Optimization problem

In order to minimize Es_{total} we need to determine the MBMA configuration parameters influencing it. By MBMA configuration parameters we mean the size of the banks, a possible implementation of allocations prediction or bank need prediction feature(s), and allocation policies. The optimization goal is to reduce to the maximum the total static energy Es_{total} consumed by a MBMA for a specific application or a specific set of applications. From (19) we can derive E_{Active} which represents the *active energy* dissipated by the MBMA, in other words the static energy that is spent only on memory blocks holding alive objects during active states. Thus E_{Active} corresponds to the minimum energy that any memory architecture will have to dissipate.

$$E_{Active} = k \cdot \sum_{i=1}^{B}[\int_0^{T_{A_i}} alive(t)_i \cdot dt] \quad (20)$$

Therefore optimizing Es_{total} comes to the problem of minimizing E_{extra} value. E_{extra} can be decomposed into the sum of three terms : E_{extra_A}, E_{extra_B} and E_{extra_C}. E_{extra_A} expresses the energy wasted during waiting states, E_{extra_B} represents the energy wasted in holding garbage objects and free memory space switched on and E_{extra_C} denotes the energy wasted during transition phases (from bank state powered-on into powered-off and vice versa) while the

system was in a working state. Fig. 4 illustrates the origin of E_{extra_C} cost components.

$$E_{extra_A} = \sum_{m=1}^{J}[E_{wait}^{m}] = \sum_{m=1}^{J}[k \cdot \sum_{i=1}^{(n+1)_m}[\int_{t_{LastTrans_m}}^{t_{Active_m}} BankSize_i \cdot dt]] \quad (21)$$

$$E_{extra_B} = \sum_{i=1}^{B}[E_{wasted_i}] = \sum_{i=1}^{B}[k \cdot \int_{0}^{T_{A_i}}(free(t)_i + garbage(t)_i) \cdot dt]$$

$$E_{extra_C} = \sum_{i=1}^{B}[E_{trans_i}] = \sum_{i=1}^{B}[k \cdot l \cdot \int_{0}^{t_{A\rightarrow Pd}}(alive(t)+free(t)+garbage(t)) \cdot dt + k \cdot m \cdot \int_{0}^{t_{Pd\rightarrow A}}(alive(t)+free(t)+garbage(t)) \cdot dt]$$

In (21) all variables labeled m refer to their respective value in m^{th} waiting state. E_{extra_A} is thus dependent on the number of waiting states that occurred during that execution. Determining the number of waiting states is not a trivial problem as it will depend on the configuration of the MBMA and the distribution of the allocation, garbage and deallocation events. The weight of E_{extra_A} inside Es_{total} is also dependent on the time needed for a bank to be switched from powered off to powered on state.

As for E_{extra_A}, E_{extra_C} weight inside Es_{total} is driven by the memory technology and more particularly by the time needed for a bank to be switched between powered off and powered on states. The more time is needed for the bank to be switched, the more predominant E_{extra_C} will be inside Es_{total}.

VII. Optimization parameters

In this section we go through parameters influencing the optimization problem introduced in above section.

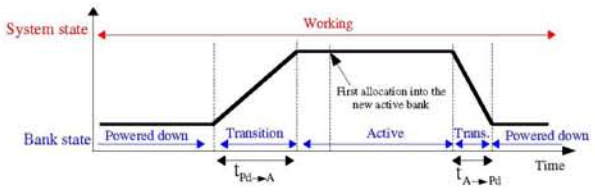


Fig. 4. E_{extra_C} components

a) Bank size: The bank size used for a MBMA has an impact on all three optimization subproblems. Increasing the banks size will increase the function $free(t)$ and thus obviously will increase E_{extra_B}. For E_{extra_A} and E_{extra_C} the increase of $free(t)$ has to be balanced by the fact that bigger banks will most likely reduce the number of waiting and transition states thus will reduce n in E_{extra_A}, l and m in E_{extra_C}. The ratio between the energy increase due to $free(t)$ and the energy decrease due to n, l and m will depend on the distribution of the allocation, garbage and deallocation events.

b) Deallocation scheme or garbage collector: Frequent garbage collections (GC) or explicit deallocation will decrease the function $garbage(t)$. Moreover if an optimum deallocation scheme is able to deallocate objects right after their last use, it would be theoretically possible to reduce function $garbage(t)$ to the constant null. But a possible decrease in $garbage(t)$ generates an identical increase of $free(t)$. In this way a better deallocation scheme reduces functions E_{extra_A} and E_{extra_C} as it reduces the number of waiting and transition states. As the possible $garbage(t)$ decrease will be identical to the $free(t)$ increase it will not affect E_{extra_B} value. In addition we also have to remember that frequent GC increases the application running time and thus tends to increase E_{extra_B} with the increase of T_{A_i}.

c) Allocation or bank need prediction: Allocation prediction or bank need prediction feature(s) intents to switch on banks beforehand in order to avoid waiting states. As a result it decreases the number of waiting states and intents to decrease E_{extra_A}. But at the same time it also drives an increase of $free(t)$ and thus an increase of E_{extra_B} and also possibly E_{extra_C}. The ratio between the energy increase due to $free(t)$ evolution and the energy decrease due to the number of waiting states decrease depends on how long beforehand banks are switched on. The extreme case would be to switch on all banks beforehand, eliminating thus E_{extra_A}, but then maximizing $free(t)$.

d) Allocation policies: Our strong feeling concerning the allocation policies is that if the policies group in a same bank objects with a similar life time, it decreases the number of waiting and transition states. Regrouping similar life time objects into particular banks is expected to increase the overall number of banks switched on and thus limiting the need for new banks. As a consequence such policy will reduce the number of waiting and transition states while $free(t)$ will increase. E_{extra_B} will then surely increase as E_{extra_A} and E_{extra_C} evolution will depend on the distribution of the allocation, garbage and deallocation events.

e) Memory fragmentation: represents free memory areas that might be unuseable for future dynamic allocations due to their relative small sizes. If these free memory areas distributed over each bank turn out to be unuseable for future allocations, they will waste during all the system runtime a static energy proportionally to their sizes. Hence fragmentation plays a role in the MBMA static energy cost as higher fragmentation lead to a potential increase of function $free(t)_i$ in E_{extra_b} and J in E_{extra_a}.

VIII. MBMA BEHAVIOR SIMULATION

The MBMA behavior was simulate on 2 different applications: a) Tobi-Tris a tetris like game written in Java, b) $Cfrac$ [5] written in C. $Cfrac$ is a allocation intensive application factoring large integers using the continued fraction method. The application input was 6 successive integers, from 21 to 37 digits, fed to the application with a 10 to 35 seconds interval.

For the Tobi-Tris Java application we used the SUN J2ME Wireless Toolkit [6] and its MIDP device emulator to capture the $allocation$, $deallocations$ and $garbage$ events. The Java application is run twice, a first run is done on the emulator compiled with default options and used to retrieve the $allocation$ and $deallocation$ events. A second run is done with the emulator compiled with options tuned to launch a garbadge collection (GC) at each bytecode execution. From this second run we are able to retrieve the $garbage$ events. The captured events $deallocation$ reflect the GC actions of the emulator's Java Virtual Machine (JVM) in the context of one memory region. We acknowledge that appropriate policies ruling the GC launch might be different for a MBMA than for a one memory region. However we want to constrain the optimization problem and mainly look first at bank size influence.

To analyze the energy behavior of an MBMA for Cfrac application we implemented a customized memory allocator. It features a first fit algorithm with one address-ordered free list per bank. When an object is allocated it scans all free lists until it finds the first free space that can hold the new object. If no free space is available, a new bank is switched on. When a object is deallocated its corresponding memory block is inserted into the respective bank free list. If there is adjacent free blocks they are coalesced in one free block. If after a $deallocation$ event one bank is left over empty the bank is switched off. The allocator is able to trace all allocations, deallocations and bank state transitions in order to compute afterward the energy cost of the MBMA.

For all MBMA behavior simulations, no allocation nor bank need prediction is used. The biggest allocated object fixes the smallest possible bank size, and the maximum number of powered on banks is used as reference for computing $E_{oneBank}$, the static energy cost if the MBMA would had only

TABLE II

CFRAC - BANK SIZE AND ENERGY COSTS IN JOULE

Bank Size (Kb)	E_{Active}	E_{extra_A}	E_{extra_B}	E_{extra_C}
2	36.03	1.61e-4	32.08	1.35e-6 %
4	36.03	7.37e-5	38.35	1.03e-6 %
8	36.03	3.40e-5	42.38	6.87e-7 %

TABLE III

CFRAC - BANK SIZE AND ENERGY COSTS COMPARISON IN JOULE

Bank Size (Kb)	**Total**	$E_{oneBank}$	**Saving**
2	68,11	85,88	20 %
4	74.38	85,88	13 %
8	78.41	85,88	8.5 %

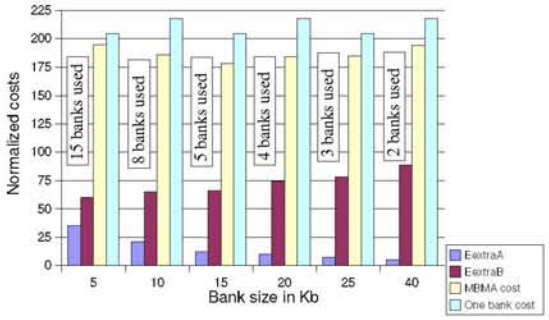


Fig. 5. Tobi-Tris - Bank size and and energy costs

one bank. Numerical value for constant k is deducted from the characteristics of the low power $\mu PD431000A$ SRAM [7]. $t_{Pd \to A}$ and $t_{A \to Pd}$ are over evaluated at 10 ms, twice the $\mu PD431000A$ operation recovery time from standby mode.

Fig. 5 shows the results for the Tobi-Tris game where all values are normalized to E_{Active}, 100 representing E_{Active} value. On Fig. 5 clearly appears the tradeoff between E_{extra_A} and E_{extra_B} to get the optimum bank size. For this application small bank size is cost unefficient due to E_{extra_A} and big bank size is cost unefficient due to E_{extra_B}. A tradeoff has to be found in between and Fig. 5 shows that the most cost efficient bank size is 15Kb. For Tobi-Tris application, with a 15Kb bank size the MBMA is using 5 banks and consumes about 175% of E_{Active}, while a 75Kb single bank architecture will consume about twice the E_{Active} value.

Table IV shows statistics on Cfrac execution for 2Kb banks. Tables II and III present the saving on static energy consumption for 3 bank sizes. With this application we note that E_{extra_A} and E_{extra_C} are relatively small compare to E_{extra_B}. This is due to the relatively small period spend in switching on and off time compare to the application run time. Each time a bank was switched on the fragmentation was compute for all already powered on banks. This gave us an average fragmentation of 0,27 with a standard deviation of 0,31. But it is important to also say that each time a new banks is switched on, on average the already switched on banks occupation rate is 99%. This clearly indicates that for this example fragmentation is causing insignificant degradation on the MBMA costs. The average occupation rate denotes the ratio over the time between the bank size and the allocated objects size in the bank. Figures from Table V indicates that banks occupation rate have a bigger impact than fragmentation. On average with a 2kb bank size only 52,91% of available memory is allocated. This is mainly due to left alone objects spread over several banks, preventing banks to be switched off.

Those results show that a substantial saving can be achieve on static memory energy consumption through MBMA with-

out any application optimization nor customized allocation policies. However it also indicates that further savings could be obtained, mainly on E_{extra_B}.

IX. Related work

Several researches have been done on data transformation or migration and memory access scheduling to exploit MBMA [8][9][10]. In contrast to these, this paper doesn't explore the possibilities to adapted the running application(s) on the system but on the contrary how to set up an optimum MBMA for a specific application. Nevertheless we believe that after the optimal memory architecture has been set further cost reductions can be achieve through application optimization. L. Benini et al. propose an algorithm for automatic scratch-pad RAMs partitioning from several application execution profiles in [11]. In this work the scratch-pad RAMs doesn't have the possibility to be switched off and on. K. Flautner et al. present in [12] a method using dynamic voltage scaling (DVS) for putting cache lines in a low-power mode, called drowsy, where data are preserved. In [12] only cache memories are addressed, while our work addreses only static energy saving on the main memory.

In [13] G. Chen et al. describe the impact of GC, compaction and bank size on an embedded Java environments with MBMA. Our work differs from [13] in that we express the optimization problem and explicitly describe the optimization parameters influencing the system, providing in this manner a total understanding on the relations between energy cost evolution and optimization parameters.

X. Conclusion and Future work

In this paper we proposed a MBMA as solution to decrease the static energy cost in memory and set up the equations ruling the optimization problem. We showed that implementing a MBMA and choosing appropriate bank size can lead to a 20% static energy saving without any software optimization, nor bank need prediction, nor dedicated allocation policies. We also observed that the banks occupation rate plays a predominant role in the MBMA static energy cost.

Future work for this study includes more simulations with deeper optimization parameter analyzes, especially on possible bank need prediction algorithms and allocation policies. We could also imagine to implement the MBMA management process within the memory as it is done for intelligent memory manager [14], leading to a probable performance improvement for cache memory architecture. Moreover, we could also investigate the possibility to compact the allocated object within all banks in order to increase the banks average occupation rate. This might be impossible to implement for conventional programming languages, such as C, Pascal, Ada, etc., but would probably better fit with object oriented language like Java. An other approach would be the use of region allocation mechanism based on objects life time. Furthermore, in the case of a multi applications platform, it is also worth exploring the solution of having several sets of different bank sizes.

TABLE IV
Cfrac - Statistics on Cfrac execution for 2Kb bank size

Total Number of Allocations: 59165
Total Number of Deallocations: 58596
Biggest allocated object in bytes: 1244
Maximum live objects size: 190041 bytes
162 times a bank has been switched on
71 times a bank has been switched off
Maximum numbers of powered on bank: 94
$\sum t_{A \rightarrow Pd}$ in mSec: 1620

TABLE V
Cfrac - Statistics on the banks occupation rate for 2Kb bank size

Average occupation rate:	52,91 %
Maximal average occupation rate:	81,69 % ($third\ bank$)
Minimal average occupation rate:	0,15 % ($94^{th}\ bank$)
Standard deviation:	23,73

References

[1] F. Catthoor, E. de Greef, and S. Suytack, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[2] J. A. Butts and G. S. Sohi, "A static power model for architects," in *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000.

[3] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: solved?" *ACM SIGPLAN Notices*, vol. 34, no. 3, pp. 26–36, 1999. [Online]. Available: citeseer.ist.psu.edu/johnstone97memory.html

[4] M. et al., "Leakage power estimation in srams," UC Irvine, CECS Technical report TR 03-32, Oct. 2003.

[5] D. Detlefs, A. Dosser, and B. Zorn, "Memory allocation costs in large c and c++ programs," *Software-Pratcice and Experience*, vol. 24(6), pp. 527–542, 1994.

[6] T. S. J. W. Toolkit, "http://java.sun.com/products/sjwtoolkit."

[7] P. S. data sheet, "http://www.necel.com/memory/."

[8] M. Kandemir, "Impact of data transformation on memory bank locality," in *DATE'04*.

[9] C.-G. Lyuh and T. Kim, "Memory access scheduling and binding considering energy minimization in multi-bank memory systems," in *DAC 2004*.

[10] V. D. L. Luz, M. Kandemir, and I. Kolcu, "Automatic data migration for reducing energy consumption in multi-bank memory systems," in *DAC*, 2002.

[11] L. Benini, A. Macii, and M. Poncino, "A recursive algorithm for low-power memory partitioning," in *ISLPED*, 2000.

[12] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. N. Mudge, "Drowsy caches: Simple techniques for reducing leakage power." in *ISCA*, 2002, pp. 148–157.

[13] G. Chen, R. Shetty, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko, "Tuning garbage collection for reducing memory system energy in an embedded java environment." *ACM Trans. Embedded Comput. Syst.*, vol. 1, no. 1, pp. 27–55, 2002.

[14] M. Rezaei and K. M. Kavi, "Intelligent memory manager: Reducing cache pollution due to memory management functions," *Journal of Systems Architecture*, vol. Volume 52, January 2006, 41-55.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2240-5
ISSN 1239-1883