# A Pervasive Internet Approach to Fine-Grain Power-Aware Computing

Ahmed Abukmail and Abdelsalam (Sumi) Helal
*University of Florida*
*Pervasive Computing Laboratory*
*Computer & Information Science & Engineering Dept.*
*Gainesville, Florida 32611, USA*
*www.harris.cise.ufl.edu*
*Email: ahmed@cise.ufl.edu, helal@cise.ufl.edu*

## Abstract

*We present a novel approach to conserve power in networked mobile devices. Our approach exploits communication within a pervasive smart space as an opportunity to save power as opposed to the classic view of communication as a drain on resources. We outsource intensive computations to the network whenever a pervasive connection to the Internet exists and when it pays off to do so. At compile-time our approach generates two versions of the program being compiled, a client version and a server version, each containing the necessary code to handle the run-time decision of executing code locally on the mobile device or remotely to the server based on power efficiency. We utilize a technique from Real-Time systems to help the compiler generate highly accurate code by calculating the number of loop iterations for each candidate section of code. This approach has the advantage of analyzing applications at a finer granularity than other similar methodologies. This is because the candidate code sections are CPU blocks represented mostly by loops. Our experimental results performed on Intel's XScale architecture and the Wi-Fi wireless technology show significant savings in power consumption by the mobile device.*

***Keywords:*** *Power Management, Computation Outsourcing, Pervasive Computing, Smart Spaces.*

## 1. Introduction

Mobile devices such as cellular phones, PDA's, laptop computers and MP3 players are becoming more popular and widespread. While these devices are becoming more powerful to be used in everyday life, they still suffer from a common problem that needs to be addressed, and that is limited battery life.

Prolonging battery life, or as often referred to as power management has been the focus of research for quite some time. This problem can be tackled on many facets by targeting the various components of a mobile computer system. In [1], an overview of techniques for power management and awareness was presented. These techniques included reducing power at the system architecture level by targeting the various components of the power equations *($P=CV^2f$)*, where $P$ is the power, $C$ is the capacitance load, $V$ is the supply voltage, and $f$ is the switching frequency. Other techniques included operating systems modifications, specific power-aware applications, as well as compiler-based techniques.

Higher-level power management techniques are considered very attractive solutions to the power preservation problem. In [2], a good argument for handling power management at the high-level was made by proposing application-level power-based APIs. One of the most attractive solutions to power management is compiler optimization as it attempts to minimize the need for programmer power-awareness. However, by studying the effects of traditional compiler optimization techniques on an application or system [3, 4], most optimizations increased the power consumed by the processor. Exception for this are loop unrolling and function inlining.

In this paper we present a compiler-based optimization technique to conserve power. Our approach views communication as an opportunity to conserve power as opposed to being a drain on the battery. Our approach is a fine-grain one as it examines, at compile-time, sections of CPU blocks that are most likely to be loops, and inserts code that will, at run-time, make a decision as to whether it would be more power-beneficial to execute the code locally on the mobile device, or remotely on a network server. The tradeoff here is between computation and communication. In order to make the determination to outsource the code or not, we utilize a methodology from real-time systems to calculate the number of loop iterations [5]. This methodology is capable of calculating loop iterations when the loops are nested, their bounds are determined at run-time, and they have multiple exits. Practically, these types of loops amount to a large majority of all loops that can be found in an application program. We targeted a subset of the C programming language, and implemented three benchmarks to verify our results. Our experimental results showed significant promise in terms of power savings.

The rest of this paper is organized as follows. In section 2, we present the concept of computation outsourcing within a smart space. In section 3, we illustrate our compile-time strategy to facilitate the run-time computation outsourcing. In section 4, we present our benchmarking and experimental results. Section 5, presents the related work, and section 6 gives our conclusion and future work.

## 2. Computation Outsourcing in a Pervasive Smart Space

Outsourcing computation is not a new idea. However, the objectives and motivation behind outsourcing the computation to a remote server, and the approach under which we're outsourcing the computation is the novel contribution here. An intelligent run-time system that we introduce can decide if it is better to execute a section of code locally

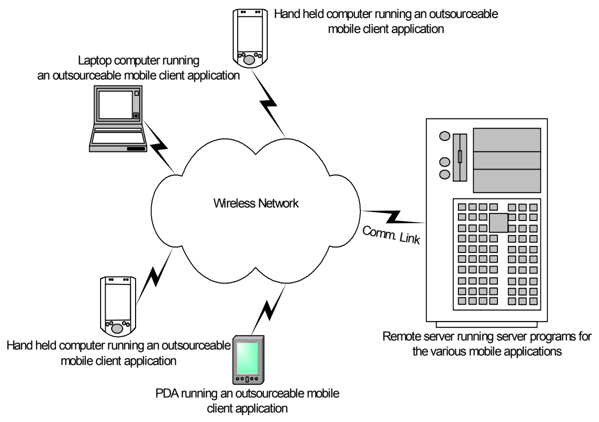
Hand held computer running an outsourceable mobile client application
Laptop computer running an outsourceable mobile client application
Wireless Network
Comm. Link
Hand held computer running an outsourceable mobile client application
PDA running an outsourceable mobile client application
Remote server running server programs for the various mobile applications

Figure 1. The framework for computation outsourcing at run-time within a smart space

on the mobile device, or if it would be more power-efficient to use a remote image of the code section (that uses free power) after sending its data through a smart space, and get the results back.

The code that is in charge of making this decision is completely transparent to the programmer. All the programmer is required to do is to compile the code to optimize for power. This will result in two versions of the program being generated which the programmer will eventually have to compile and install.
The overall framework for outsourcing is described in figure 1. The idea is that a server machine located in a pervasive smart space and accessible via a wireless network can serve as a server for a host of mobile devices such as handhelds, PDAs and laptop computers. This server at run-time will receive requests from client programs running on any of these devices for outsourcing code to the server.

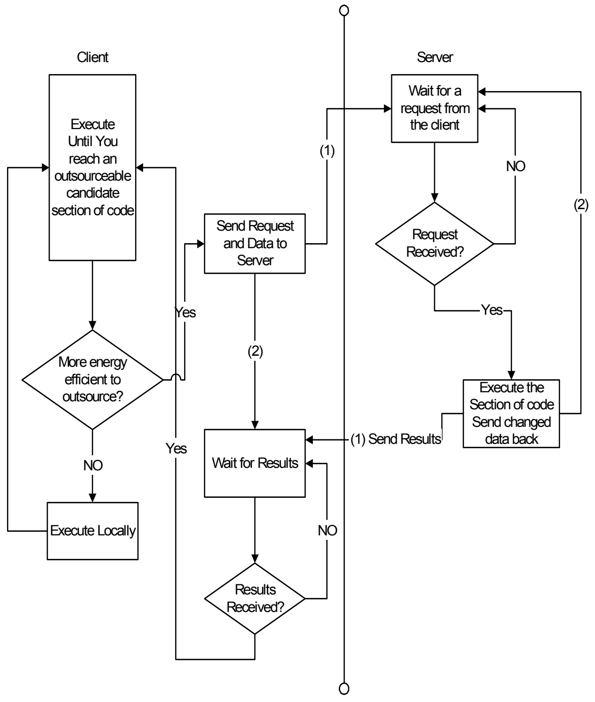
Client
Server
Execute Until You reach an outsourceable candidate section of code
More energy efficient to outsource?
NO
Yes
Yes
Execute Locally
Send Request and Data to Server
(1)
(2)
Wait for Results
NO
Results Received?
Wait for a request from the client
NO
Request Received?
Yes
(2)
Execute the Section of code Send changed data back
(1) Send Results

Figure 2. The steps for executing the client program under the outsourcing framework within a pervasive smart space

Once the client and the server are installed on their respective machines, the client application executes normally until it reaches a candidate section of code that has been designated as outsourceable. Once this section is reached, the intelligent code that was inserted at compile-time is executed to make the outsourcing decision within a pervasive smart space. In fact, the candidate code will not be executed (locally or remotely) until the decision making code is executed. This process is illustrated in detail in figure 2.

## 3. The Compilation Strategy

Our compiler optimization technique for low power analyzes a source program at the three different levels of representation (high, intermediate, and low). At the high-level, we collect information about the data involved in each loop. At the intermediate level we utilize the methodology described in [5] to calculate the number of loop iterations. The reason this is an intermediate level analysis is because in [5], they analyze the register transfer list (RTL) [6] representation of the source code. At the low level, we determine the machine instructions generated by an assembler to determine which instructions are getting executed within each loop. This new compilation technique utilizes per-existing utilities such as the gcc compiler and Metrowerks®' CodeWarrior. The source code is passed first to the gcc compiler for syntax checking. Then the code is passed along with its assembly representation generated by the assembler is passed to the optimization process that in turn will generate two version of the original source code, a client and a server, by inserting the necessary communication code needed for each version. Then each version of the code is compiled and then installed on its target machine. Figure 3 illustrates this process in details.

### 3.1. Loop Data and Iteration Acquisition

The first stage of our technique is to recognize the maximal CPU blocks (most likely these blocks will be loops). These loops constitute the opportunity for optimization (candidate code for outsourceability). Once these loops are recognized at the high-level, then we collect all the data elements associated with them, as well as determine the beginning and end file positions of these loops.

The first part of this stage is to implement a parser-like module (we call it the pseudo-parser) to recognize CPU blocks, collect the data used within each loop, and identify what variables are R-valued (do not change), and what variables are L-valued (change). We didn't need to implement a full parser here as the syntax has already been checked before entering this stage of the algorithm.

The second part of this stage of the implementation of our methodology is to figure out the number of loop iterations and associate each number with each loop identified by the pseudo-parser. In order to determine the number of loop iterations we utilize a very useful utility developed by Healy *et al* as a result of their research for predicting the worst case execution time (WCET) for a program as part of implementing a static timing analyzer in real-time systems [5]. Their approach automatically bounds the number of loop iterations. Their methodology is implemented by analyzing the register transfer list (RTL) [6]. They based their implementation on the Very Portable C

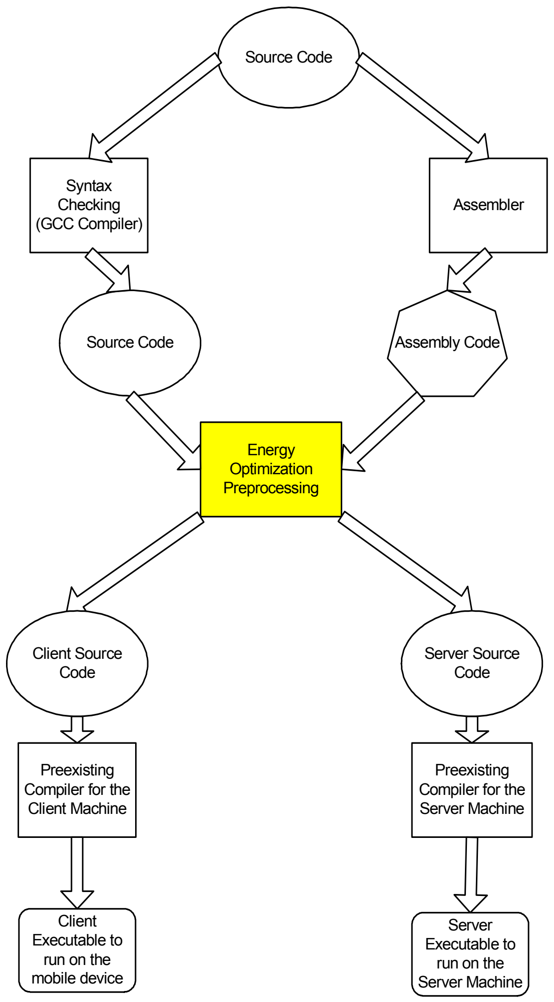
Source Code
Syntax Checking (GCC Compiler)
Assembler
Source Code
Assembly Code
Energy Optimization Preprocessing
Client Source Code
Server Source Code
Preexisting Compiler for the Client Machine
Preexisting Compiler for the Server Machine
Client Executable to run on the mobile device
Server Executable to run on the Server Machine

**Figure 3. An overall view of the compilation and optimization processes**

Compiler (vpcc) [6], where they input a C program and produce a set of files one of which is of interest to us and that is the ".inf" or INF file. The INF file contains the maximum number of iterations for each loop within the program if it's possible to determine that number. We're only interested in this number because we do not want to underestimate the execution cost of each loop. We parse the INF file generated from the modified vpcc compiler to obtain the number or formula representing the maximum number of iterations for each loop.

## 3.2. Calculating the Size of Loop Data

The next stage is to calculate the data size for each loop. In this stage, we examine each variable involved in the loop, and based on the size of the variable in bytes (including arrays), we add the value to our sum to calculate the size in bytes. Additionally, we categorize variables as either L-valued (change), or R-valued (do not change) in order to minimize the communication cost by not requiring R-valued variable to be sent back to the mobile device. Also variables within nested loops are also considered as part of outer loops, and therefore are factored into the cost of the outer loops.

## 3. 3 Identifying Loop Instructions and Total Loop Execution Cost

Using the assembly code representation of the source program, we can recognize loops within the assembly code. The target architecture (Xscale) has a unique way of identifying loops using a combination of the compare *"cmp"* instruction and the branch instructions such as *"ble, blt, bge, bgt, bne, beq", and "b"*. This way we are able to identify or rather delimit where the assembly code for each loop starts and where it ends. However, when loops are nested, we need more information in order to be able to map loops at the assembly level with those at the high level. The additional information needed is available in the structure containing information about all the loops (we call it the *loopdata* data structure). The information needed here is: which loop is nested within which loop, and that information will match what was obtained via our pseudo-parser.

Once each loop was delimited, then it was just a matter of going through the instructions that constitute the loop, and summing up their pre-measured power cost. In addition to the cost of each instruction there is a cost for pipeline stalls. This cost was obtained experimentally using multiple instruction sequences once the cost per instruction was determined. Therefore, when we recognize that certain instructions precede others (e.g. ldr before an add, or an ldr before an str), we add the measured pipeline stall power cost. This calculation gave us the cost of a single execution of the loop. At this point, we have all what we need to be able to produce the resulting client and server. The total loop execution cost becomes a matter of multiplying the cost of a single execution by the formula representing the number of loop iterations calculated before.

## 3.4 Insert Outsourcing Code

The implementation of this code was very large, but it was not difficult. As our pseudo-parser generated information of where each loop begins and where it ends. The location of where we need to generate the necessary C code to create a client/server based application becomes a matter of inserting the necessary include files, variable declaration (we declared them globally in this implementation).

Before each loop, in the client version, we inserted an *if-statement* that makes the test for outsourceability, and that test is given by a nested formula depending on the number of loop nests. The inserted code runs by checking if an outer loop is outsourceable, then go ahead and outsource it to the server (send a signal to the server, send the data, and wait for the result back). Otherwise, test for each loop inside of the outer loop recursively, and insert the necessary outsourcing code. The server version is a much simpler application, as it's composed of a *switch-statement* within an infinite *while-loop*. Each case in the switch statement represents a single loop (this includes all the loops nested within it). The server at run-time only executes a case if and only if it receives a message indicating the number of the loop to be executing. In this case it waits for input to be sent, executes the necessary code, and sends the result back to the client.

## 4. Measurements and Experimental Results

Our measurements, and experiments were done in two stages with our platform setup. The first stage was to estimate as accurately as possible the cost of each supported machine instruction (assembly instruction). Secondly, the second stage is to measure the cost of each benchmark, first without our optimization, and second with our optimization.

## 4.1 Experimental Setup

Our target architecture is an Intel® Xscale [7] which is an integral part of the Intel® PCA [8]. We chose the Sharp® Zaurus SL-5600 [9], which contains an Intel® Xscale PXA-250 processor, and is running Linux, as our mobile device. Installed on the Zaurus, is a Socket® low-power wireless LAN card [10]. The outsourcing server is an Intel® x86 machine running RedHat Linux 7.2.

Developing applications on the Zaurus was achieved using Metrowerks™ Codewarrior for the Sharp® Zaurus [11]. For measuring the power consumed, we used an Agilent® 34401A multi-meter which was connected to a Windows™ 2000 desktop computer via an IEEE488.1 General Purpose Interface Bus (GPIB) cable [12]. Installed on the desktop is Agilent®'s Intuilink plugin which works with Microsoft® Word and Excel. The voltage coming from the AC adapter remained at a constant 5 volts. Therefore, the only two factors in the power equation are the current drawn and the time in seconds as power is given by the equation $E = V * I * T$, where $E$ is the power consumed, $V$ is the voltage, $I$ is the current, and $T$ is the elapsed time.

In order to measure the cost for each assembly instruction used, we used a methodology similar to that described in [13]. We created simple programs that contain each instruction within a loop, and executed the loop multiple times, and averaged the power cost for the execution over the number of loop iterations. Also, to account for communication cost we created a simple client/server application and sent data back and forth to and from the Zaurus and measured the power consumed by communication to estimate the cost of communication per byte. This estimation worked well using our benchmarks, and resulted in producing the expected results.

## 4.2 Experimental Results

In order to test the effect of our approach on power, we implemented three different benchmarks that spanned three different formations of data and execution complexity. The first was the Fibonacci loop, which contains constant data, but it executed in $O(n)$ time. We performed the testing using the numbers, 100000, 200000, and 300000, for which we got 60, 80, and 87% power saving respectively. The communication cost for this loop was quite small as it only involved sending only a few integer variables. However the computation involved was 100K, 200K, and 300K respectively each time we executed the benchmarks. The second benchmark that we used was a square matrix multiplication loop, which ran in $O(n^3)$ where $n$ is the number or rows and the number of columns of each matrix. For matrix multiplication we used a 200x200, a 300x300, and 400x400 matrices, and for this benchmark we got power savings of 73, 80, and 88%. For this benchmark, we notice that it has an $O(n^2)$ data size in addition to a few temporary variables for executing the loop, so the execution time complexity was 1.5 times the order of magnitude of the communication size. The last benchmark that we used was a rectangular nested loop implementation of the bubble-sort algorithm which executed in $O(n^2)$ and the data size is linear, so here we have an execution time complexity that is twice the order of magnitude of the communication size. We used a rectangular nested loop implementation of the bubble-sort loop instead of the more optimized triangular nested loop implementation to facilitate the detection of the number of loop iterations. We sorted 10000, 20000, 30000, and 50000 integers, and for this benchmark the power savings exceeded 95%. Figure 4, shows the real-time comparison histograms between local and remote execution of the bubble sort loop of 50,000 integers. This histogram was generated using the Intuilink software.

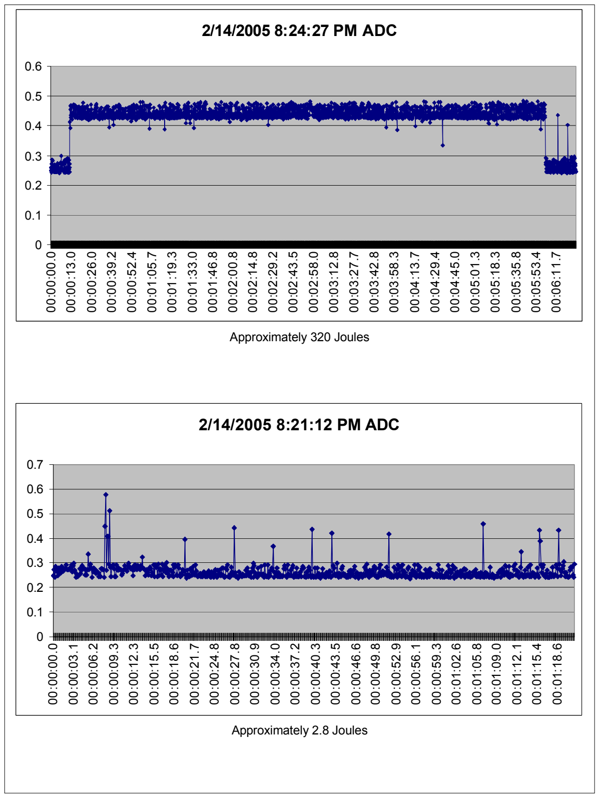

**Figure 4. A power consumption comparison between local (top) and outsourced (bottom) execution of the bubble sort loop for sorting 50,000 integers**

The benchmarks that we used are somewhat primitive. However, due to the fine granularity of our approach, they are good representatives for testing our methodology. Currently, we are working on testing this approach with larger types of benchmarks, such as voice recognition software as well as a 3-D graphics rendering application. The two applications that we are currently targeting are known to be computationally extensive to the point that it's not very practical to use them on a handheld device.

## 5. Related Work

Compiler-based energy/power management techniques have been studied for quite some time. In [14], an overview of compilation techniques to reduce power consumption was given. They suggested methodologies that mainly target the low-level code such as instruction re-ordering to reduce switching, reduction of memory operands as memory access consumes power was also presented, re-applying the code generation techniques through pattern matching by assigning a power cost as opposed to a time cost.

In [15], a proposed methodology to annotate the code at compile-time to adaptively select at run-time the optimal number of instruction to be fetched or executed in parallel to save power. However, this solution would require additional architectural changes. Other work that required a combination or hardware/compiler modification for power saving was also presented in [16], [17], and [18].

The work that is most closely related to our approach can be found in [19]. It proposes a very similar solution to ours. However, this work, while a compiler-based solution for remote execution, it targets specific tasks. Their experimental results are given for a hand-simulated compilation of their benchmark. Another research that discusses a remote processing framework to save power was also introduced in [20]. However, this work targets task execution and it is not a compile-time solution.

## 6. Conclusion and Ongoing Work

Our experimental evaluations showed that computation outsourcing within a smart space has a great potential in power management. By exporting CPU processing responsibilities into the network, the mobile device was able to deliver the expected functionality while consuming less power and lasting for a longer period of time. Therefore, communication should not be viewed as a drain on the battery, but as an opportunity to save power.

We found that, the research done in the domain of real-time systems is quite useful as its aim is always knowing as much as possible about an application before run-time (*e.g.* compile-time). The reason for that in real-time systems is to execute programs to finish within a deadline. That means that the more we know about program behavior before execution the better the program can be tuned to either meet a deadline in the case of real-time systems, or consume less power in our case of mobile and pervasive computing. However we believe that the limitations of the utilized methodology which are represented in the inability to handle non-counter-based loops, such as those loops that are pointer-based, logical-expression based, and some non-rectangular nested loops, raise the need to investigate other computer science disciplines, specifically automated software verification.

While the work done in the area of automated verification for loop invariant generation is not quite mature enough, it has a great promise to be combined with the utilized real-time systems methodology to generate accurate estimates for the number of loop iterations. We will investigate the research done in this area to determine the extent of its applicability in our work. Specifically, we will look at the work done in [21]. We believe that, when combined with the methodology we utilized for determining the loop iterations, the area of loop invariant generation will be quite beneficial to our research.

## 7. Acknowledgements

We would like to thank Dr. David Whalley of Florida State University, and Dr. Chris Healy of Furman University for providing us with the software package that we used as a basis to calculate the number of loop iterations (vpcc compiler).

## 8. References

[1] A. Abukmail, H. Helal, *"Power Awareness and Management Techniques,"* in: M. Ilyas, I. Mahgoub (Eds.), Mobile Computing Handbook, CRC Press, Boca Raton, FL, 2004.

[2] C. S. Ellis, "*The Case for Higher Level Power Management,"* in: 7th Workshop on Hot Topics in Operating Systems, Rio Rico, AZ, March 1999.

[3] M. Velluri, L. John, "Is Compiling for Performance == Compiling for Power?," in: 5th Annual Workshop on Interaction between Compilers and Computer Architecture, Monterrey, Mexico, January 2001.

[4] M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, W. Ye, *"Influence of compiler optimizations on system power,"* in: 37th Conference on Design Automation, Los Angeles, CA, June 2000.

[5] C. Healy, M Sjödin, V. Rustagi, D. Whalley, "*Bounding Loop Iterations for Timing Analysis,*" in: IEEE Real-Time Technology and Applications Symposium, Denver, CO, June 1998.

[6] M. E. Benitez, J. W. Davidson, "*A Portable Global Optimizer and Linker,*" in: ACM SIGPLAN '88, Atlanta, Georgia, July 1988.

[7] Intel Xscale Technology Overview, http://www.intel.com/design/intelxscale.

[8] Intel Personal Internet Client Architecture, http://www.intel.com/personal/wireless/handheld/pca.htm

[9] Sharp Electronics, http://www.sharpusa.com

[10] Socket Communications, http://www.socketcom.com

[11] CodeWarrior Development Studio for the Sharp Zaurus, http://www.metrowerks.com

[12] Agilent Technologies, http://www.agilent.com

[13] V. Tiwari, S. Malik, A. Wolfe, T. Lee, *"Instruction Level Power Analysis and Optimization of Software,"* VLSI Signal Processing Systems 13 (2) (1996) 1-18.

[14] V. Tiwari, S. Malik, A. Wolfe, *"Compilation Techniques for Low Energy: An Overview,"* in: International Symposium on Low Power Electronics and Design, San Diego, CA, October 1994.

[15] D. Marculescu, "*Profile-Driven Code Execution for Low Power Dissipation,"* in: International Symposium on Low Power Electronics and Design, Rapallo, Italy, July 2000.

[16] N. Bellas, I. Hajj, C. Polychronopoulos, G. Stamoulis, *"Architectural and Compiler Support for Energy Reduction in the Memory Hierarchy of High performance Microprocessors,"* in: International Symposium on Low Power Electronics and Design, Monterey, CA, February1998.

[17] C-H. Hsu, U. Kremer, M. Hsiao, *"Compiler-Directed Dynamic Frequency and Voltage Scheduling,"* in: 1st International Workshop on Power-Aware Computer Systems, Cambridge, MA, November 2000.

[18] C-L Su, C-Y Tsui, A. Despain, "*Low Power Architecture Design and Compilation Techniques for High-Performance Processors,"* Technical Report No. ACAL-TR-94-01, University of Southern California. February 1994.

[19] U. Kremer, J. Hicks, J. Rehg, "*Compiler-Directed Remote Task Execution for Power Management,"* in: Workshop on Compilers and Operating Systems for Low Power, Philadelphia, PA, October 2000.

[20] A. Rudenko, P. Reiher, G.J. Popek, G.H. Kuenning, "*The Remote Processing Framework for Portable Computer Power Saving,*" in: ACM Symposium on Applied Computing, San Antonio, TX, February 1999.

[21] C. Pasareanu, W. Visser, *"Verification of Java Programs Using Symbolic Execution and Invariant Generation,"* in: 11th International SPIN Workshop on Model Checking of Software, Barcelona, Spain, April 2004.