A near-Zero Run-time Energy Overhead within a Computation Outsourcing Framework for Energy Management in Mobile Devices

Ahmed Abukmail
School of Computing
University of Southern Mississippi
ahmed.abukmail@usm.edu

Abdelsalam (Sumi) Helal
Computer Science and Engineering Dept.
Southern Methodist University
helal@engr.smu.edu

Abstract

In order to support run-time computation outsourcing to save energy, a certain amount of overhead has to be incurred to facilitate appropriate communication. As computation outsourcing is a methodology for saving energy on mobile devices, the amount of overhead incurred must be kept to a minimum. In this work, we support our compile-time methodology to facilitate the outsourcing of intensive computation with a run-time monitoring system that consumes an extremely low amount of energy (near-Zero). This near-Zero overhead resulted from analyzing the code at compile-time rather than run-time execution and profiling. The compile-time strategy utilized in our work analyzes the code at multiple levels of abstraction (High, Medium, and Low). The result of the analysis, which takes advantage of a real-time systems technique that calculates the maximum number of loop iterations, hence giving us a worst-case execution time for each loop within the benchmark application, allows for a fine-grain analysis of our benchmark. Resulting from analyzing the code, a client/server version of the applications is produced. As a result of producing this client/server version, certain run-time support has to take place on both the machine executing the client (the mobile device) as the machine executing the server. Our experimental results as performed on a Sharp Zaurus, utilizing Wi-Fi as a means of communication, showed tremendous energy saving while incurring a near-Zero run-time overhead.

Key Words- Energy Management, Outsourcing, Smart Spaces, Energy Monitor, Battery Monitor.

1. Introduction

Energy/Power management in mobile devices has been and continues to be a target for research. This is due to the popularity of mobile devices such as cellular telephones, PDAs, notebook computers, as well as the all popular digital media players such as Apple's iPod/iPhone and Microsoft's Zune. These devices continue to become more powerful, efficient, and have multi-purposes and capabilities. One of the things that these devices have in common is that they're battery operated and therefore their energy consumption must be managed. The energy problem has been and continues to be tackled on many facets targeting the different layers of a computer system. We presented an overview of power awareness and management techniques that deal with this problem [1]. These techniques targeted reducing power at the hardware/architecture level, the operating system, as well as at the application level.

High-level energy management techniques are considered highly attractive. Ellis made a case for high level energy management by proposing application-level power-aware APIs [2]. Compiler-based energy management is one of the most attractive solutions as it minimizes the need for programmer's energy-awareness. The effect of traditional compiler optimization techniques for power-awareness has been studied and showed lack of promise to alleviate the energy problem [3, 4].

Our work exploits compiler-based energy management in a novel way that is distant from the traditional compiler optimization techniques, not only to aid in facilitating computation outsourcing, but also to minimize the overhead required to support the computation outsourcing mechanism. Our technique exploits communication as an opportunity to save energy as opposed to being a drain on the battery. In [5], we introduced a fine-grained approach to energy management as it examines basic program blocks that are mainly represented by loops, and inserts code that will decide at run-time whether it would be more energy-beneficial to execute code locally on the mobile device, or remotely on a server within a pervasive smart space (surrogate server). As a result, a tradeoff between computation and communication will take place at run-time.

To be able to insert the necessary code that makes the determination for outsourcing, we utilized a methodology from real-time systems to calculate the number of loop iterations [6]. This methodology supports nested loops as well as loops with multiple exits. In our implementation, we supported a subset of the C programming language.

The ability to pre-calculate the number of loop iterations and therefore predicting the behavior of an application allowed for very little work to be done to support outsourcing at run-time. Therefore, we were able to implement an extremely low energy (near-Zero) run-time support represented mainly by a battery monitor and a network monitor. As a result it is not necessary to create run-time support as extensive as that presented in [7] where five different monitors were needed. Our work showed significant energy savings while keeping the run-time overhead to a near-Zero.

The rest of this paper is organized as follows: in section 2 we present an introduction to computation outsourcing. In section 3, we present an explanation of our compile-time algorithm. Section 4 presents the work done at run-time. In section 5 we present our benchmark as well as our experimental results. Section 6 presents the related work, and section 7 gives our conclusion and future work.

2. Computation Outsourcing

The idea of outsourcing computation to remote servers is not a new idea. However, the motivation for doing so is our contribution. We presented the outsourcing concept in [5]. The intelligent run-time system introduced decides if it's better to outsource or to execute the code locally on the mobile device based on the energy consumption factor.

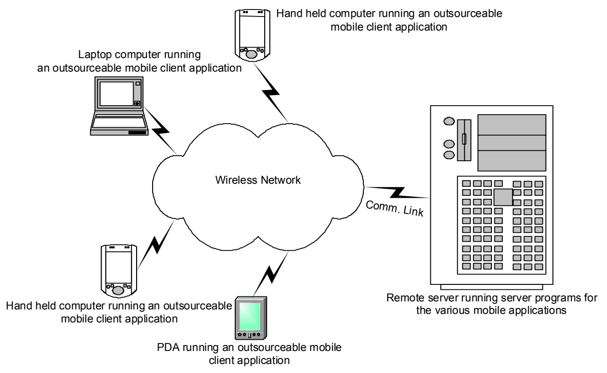
Hand held computer running an outsourceable mobile client application
Laptop computer running an outsourceable mobile client application
Wireless Network
Comm. Link
Hand held computer running an outsourceable mobile client application
PDA running an outsourceable mobile client application
Remote server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

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

The code in charge of making the outsourcing decision is completely transparent to the programmer developing the application. The only programmer responsibility is to compile the code to be optimized for energy. The result will be two versions of the code a client and a server in high-level code (C code in this case). The programmer thereafter will compile both versions of the code. The client code will run on the mobile device and the server will be placed on a surrogate server machine within the target pervasive smart space. We presented in [5] the overall framework for outsourcing in figure 1.

In order to make this work, certain run-time support has to take place in order to handle proper communication between the client and the server. As a result additional code has to be installed on both the client machine (mobile device) and the server machine.

Once both the client and the server have been installed on their respective machines along with the run-time support application, then the execution goes in parallel as it is described in figure 2.

```
Client:
    1.  Execute normally until you reach a
        candidate section of code
    2.  If energy efficient to outsource
           Send a request and data to server
           Wait for results
        Else
           Execute Locally
    3.  Goto 1.

Server:
    1.  Wait for requests from client
        machines.
    2.  If request received
           Process request
           Send results back to client.
    3.  Goto 1.
```

Figure 2. Client/Server execution under the outsourcing Framework within a pervasive smart space.

3. The Compile-Time Strategy

Our compiler optimization technique presented in [5] analyzes the source code of a program at three different levels of abstraction (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 [6] to calculate the number of loop iterations. The reason this is an intermediate level analysis is because in [6], they analyze the register transfer list (RTL) [8] 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 pre-existing utilities such as the gcc compiler and Metrowerks' (Now Freescale) 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, 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 installed on its target machine.

3.1. Loop Data and Iteration Acquisition

The first stage of our technique is to recognize the maximal basic program 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. In this stage we identify L-valued data, and R-valued data in order to minimize the communication. We called this stage of our implementation a pseudo-parser. We utilized Healy *et. al.'*s work to determine the number of loop iterations as a result of their work to predict the worst case execution time (WCET) for a program by implementing a static timing analyzer for real-time systems [6].

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 (Intel's 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 by 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. Run-time Support

Here we present the two monitors used as runtime support for the outsourcing mechanism. The two monitors work together and they get executed based on the user preference and the battery condition. The attractive property of these two monitors can be summed in the fact that most of the time they are not consuming any energy. In fact, they consume very little energy even when they are doing work. The way these two monitors work depends on the user preference in the first place, and once that has been determined, the condition of the battery of the mobile device takes control of the decision making process. These two monitors will run on the client mobile device. In addition to these two monitors, a server program will run on the surrogate device waiting for connections from the client. The battery monitor, network monitor, and the server will together establish the service detection within the wireless network.

4.1. Battery Monitor

The battery monitor gets executed either by the user or the operating system. This is also a decision to be configured by the user. The user chooses if he/she desires to run in energy-saving mode or in normal mode. If normal mode is selected, then nothing happens and the monitor exits. Otherwise, if energy-saving mode is selected, then the battery monitor will ask the user if the energy saving is to take place immediately, or it should wait until the battery gets below a certain limit. If energy saving is to take place immediately, then the battery monitor will immediately call the network monitor. Otherwise, the battery monitor will sleep (consuming a very negligible amount of energy) and periodically check the status of the battery by contacting the operating system. Contacting the operating system is a very trivial matter as it will only look at a file called "/proc/apm", and extract the remaining percentage of the battery. Once it reaches the limit specified by the user, then it will contact the network monitor that will complete the task of setting up the device in an energy-saving mode.

4.2. Network Monitor and Surrogate Discovery

Once the network monitor is called, it will send out a broadcast that will be received by a network server that is providing any service for the client. This server will be running on the surrogate machine that is to service the client. Once the server receives the broadcast it then will establish a handshake with the client device and inform the device of its name, and that it is ready to service the device and that it supports outsourcing. At that point, the network monitor will create a configuration file that is to be opened by the application to determine if it would run in energy-saving mode, or normal mode. If any type of error occurs on the way to creating this file, the file will not be created, and hence there will be no energy-saving mode.

At runtime the client application will start running and checking if the energy-saving file exists, and if so, then it extracts the information about the server from it, establish the connection, and execute in energy-saving mode, otherwise, it executes in normal mode.

5. Experimental Validation

5.1. Experimental Setup

Our target architecture was Intel's XScale which is an integral part of Intel's PCA. We chose the Sharp Zaurus SL-5600 containing Intel's XScale PXA-250 processor, and running Linux. We installed Socket Communications' low-power wireless LAN card. We outsourced to a PC running Redhat Linux 7.2. For measurements and instrumentation we used an Agilent Technologies' 34401A multimeter and the Intuilink Plugin to collect the data.

Assuming a constant voltage of 5 volts, the two factors remaining in the energy equation are the current and the time as $E = V \cdot I \cdot T$, where E is the energy, V is the voltage, I is the current drawn, and T is time in seconds.

5.2. Per-Instruction and Communication Cost

Prior to the execution of our algorithm we had to measure the cost of communication and the cost of the machine instructions to calculate the energy consumed by each instruction as well as the per-byte transfer communication cost. We used a method similar to that described in [9] in order to acquire the energy cost per instruction. We created a simple program that executes each instruction within a loop, executed the loop multiple times and averaged the cost. As for the communication cost, we created a simple client/server application and send and received data and measured the energy cost via the multimeter.

5.3. Experimental Results

Testing the effect of our approach on energy consumption required the implementation of various benchmarks. In [5], we showed a great deal of energy saving using three very simple benchmarks. First we tested the Fibonacci loop which is composed of a constant data, and a complexity of $O(n)$, and we were able to achieve energy savings between 60% and 87% depending on the size of the Fibonacci number (100K – 300K). Secondly we tested the matrix multiplication loop which has $O(n^2)$ data and $O(n^3)$ complexity. For matrices of size 200x200, 300x300, and 400x400 we achieve energy savings of 73%, 80%, and 88%. We also tested a square bubble sort algorithm which has $O(n)$ data and $O(n^2)$ complexity. We were able to achieve energy savings exceeding 95%.

To show the benefits of our approach, a more realistic benchmark had to be developed to show that this approach has more meaningful and potentially industry-utilizable benefits. Some of the most computationally intensive computations are those involved in generating an image representing a 3-D graphics scene. To generate a 3-D graphics scene, the input and output of the program are extremely non-expensive processes, as even more complex scenes can be described with a virtually small amount of data. Also, the output is always a 2-D image.

But to get from a 3-D description of the scene to a 2-D image depicting the scene, highly complex calculations have to be done. In this benchmark, the size of the data is $O(n^2)$ and the order of the computation is also $O(n^2)$. However, the amount of constant calculation within each iteration is large when compared to the amount of communicating each unit of data involved in the computation. Our experimental results showed a significant amount of energy saving for generating a scene by ray-tracing 3 spheres of different sizes and colors in space to generate 3 different images of 50x50, 100x100, and 200x200. Figure 3, shows the input data passed to the ray-tracing process and figure 4 shows the image produced.

The input to the ray tracing application is quite a simple input composed of a few floating point numbers. These numbers represent the description of the world composed of: the observer, the light source, and the parameters describing three spheres in space.

```
200 200
.1
200 200
4 4 4
.8 .8 .8
4 4 4
0 0 0
1 1000
5 5
3
0.1 0.1 0.1 0.3
.5 .4 .3
1 .8 1 10
0.5 0.5 0.8 0.2
.3 .4 .2
1 .8 1 10
0.2 0.8 0.5 0.1
.4 .3 .5
1 .8 1 10
```

Figure 3. Input file for the ray tracing application.

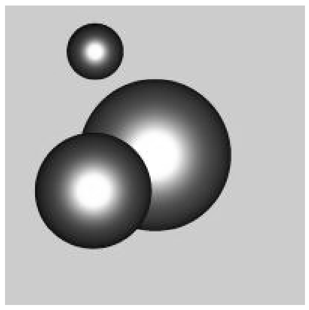

Figure 4. The 2-D image representing the 3-D scene generated by the ray tracing application for a 200x200 image.

We did our experiments on various sizes of data and we generated a 50x50 image, a 100x100 image, and a 200x200 image for the same scene. The amount of computation was so large that in all three cases, the computation was outsourced. Figure 5 shows the results of comparing local execution vs. remote execution for the generation of the 200x200 image.

Notice in the figure the base energy consumed prior to the execution of both application (the local version and the remote version), there is hardly any difference in the energy consumed and that is due to the near-Zero cost of the run-time support monitors running on the mobile device. This is an extremely attractive feature of our solution due to our compile-time analysis of the code at the various levels of abstraction.

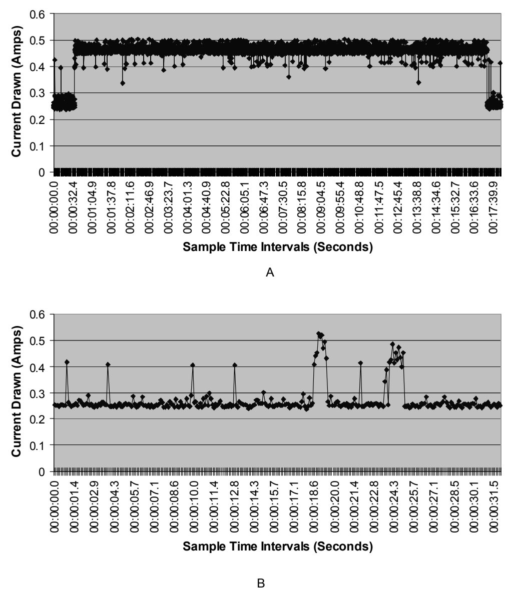

Figure 5. Local (top) vs. remote (bottom) execution for the ray tracing application of a 200x200 image.

6. Related Work

Energy management at compile-time has been the subject of a fair amount of research. An overview of compilation techniques was given in [10]. Suggested techniques included targeting low-level code such as instruction reordering, reducing memory operands, and applying pattern matching by assigning power/energy cost as opposed to performance cost.

Marculescu proposed a technique that annotates the code at compile-time for optimal parallel instruction fetch and execute [11]. This solution requires additional architectural changes. Other work that was done at the compiler level along with hardware changes was presented in [12, 13, 14].

The work that is closely related to ours was presented by Kremer *et. al.* [15]. Their work proposes to remotely execute the code, however they targeted specific tasks as opposed to the finer granularity we present. The experimental results that they present are for a hand-simulated compilation of their benchmarks. Other work that addresses remote processing to save energy was also introduced in [16], but it targets tasks execution and it's not a compile-time solution.

As for our run-time support, a more extensive work was done by Flinn *et. al.* to handle that facet of the work [7]. However, they utilized a more complex system with five different monitors executing.

7. Conclusion and Future Work

Our experimental results and validation showed that there is a significant amount to be gained by computation outsourcing when managing energy. By exporting CPU processing responsibilities into the network, the mobile device was able to deliver the expected results while consuming less energy for longer time. As a result, it is not always the case the communication should be viewed as a drain on the battery, but rather should be exploited as a mean to save energy when possible.

We believe that research that is capable of predicting the behavior of programs (real-time systems, and possibly machine learning techniques) will be greatly beneficial as it allows for knowing as much as possible about a program before executing it. As a result, the more we know about a program's behavior before we run it, and once the program is transformed into a client/server application, the runtime support will be kept to a minimum.

Additional work in this area would be greatly beneficial when it comes to energy management. Not only will it allow for saving energy on mobile devices, but it will also allow for the ability to execute larger, and more computationally intensive applications on the mobile devices, hence expanding their capabilities. Our work, in specific, has some limitations that must be dealt with in the future. Such limitations include the inability to handle non-counter-based loops, pointer-based loops, and some non-rectangular nested loops. As a result, these limitations will be investigated. In addition we will investigate the applicability of the work done in automated verification for loop invariant generation such as that work done in [17].

In addition, this work will have to eventually include certain security measures since wireless communication is considered, and we will be investigating issues relating to security while keeping in mind that security measure do consume energy.

8. References

[1] A. Abukmail, A. 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] A. Abukmail, A. Helal, *"A Pervasive Internet Approach to Fine-Grain Power-Aware Computing,"* in: The 2006 International Symposium on Applications and the Internet (SAINT 2006), Phoenix, AZ, January 2006.

[6] 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.

[7] J. Flinn, S. Park, M. Satyanarayannan, *"Balancing Performance, Energy, and Quality in Pervasive Computing,"* in: 22nd International Computing Systems, Vienna, Austria, July 2002.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.