PERVASIVE COMPUTING APPROACH TO ENERGY MANAGEMENT

By

AHMED AHED ABUKMAIL

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2005
To my wife Nermeen; and my two beautiful daughters, Raneem and Haneen.
ACKNOWLEDGMENTS

I would like to express my greatest appreciation and gratitude to my supervisory committee chair (Dr. Abdelsalam Helal) for his guidance during my doctoral studies. I also express my gratitude to my supervisory committee members (Dr. Gerhard Ritter, Dr. Sartaj Sahni, Dr. Joseph Wilson, and Dr. Louis Block) for their help.

I thank Dr. David Whalley (of Florida State University), and Dr. Chris Healy (of Furman University) for providing the software package we used to calculate the number of loop iterations (vpcc compiler).
TABLE OF CONTENTS

ACKNOWLEDGMENTS ............................................................................................................. iv
LIST OF TABLES .................................................................................................................... vii
LIST OF FIGURES ................................................................................................................ viii
ABSTRACT .......................................................................................................................... x

CHAPTER

1 INTRODUCTION ................................................................................................................ 1

2 RELATED WORK .............................................................................................................. 4
   Energy-Management Techniques at the Architecture Level ......................................... 4
      Smart Batteries ........................................................................................................... 5
         Battery basics ......................................................................................................... 5
         Intelligent power drainage ............................................................................... 5
      Energy-Aware Processors ....................................................................................... 6
   Reducing Power through Circuitry Components ....................................................... 7
      Power and energy consumption equations ....................................................... 7
      Voltage and frequency scaling ......................................................................... 7
      Capacitance load reduction .............................................................................. 8
   Power Reduction through Architectural Design ....................................................... 9
   Operating System Techniques .................................................................................. 9
   Energy-Management Solutions .............................................................................. 9
   Memory and I/O Management .................................................................................. 10
   Communication Techniques ................................................................................... 10
   Scheduling ................................................................................................................ 11
   Software Application Techniques ........................................................................... 12
      Compilation Techniques .................................................................................... 12
         Reordering instructions ................................................................................... 12
         Reduction of memory operands .................................................................... 13
         Code generation through pattern matching ............................................... 13
         Code annotation ................................................................................................ 14
         Remote task execution .................................................................................... 14
   Application-Level Techniques .................................................................................. 14
   Tools for Low-Energy Design and Measurement ................................................... 16
      PowerScope ......................................................................................................... 17
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-1</td>
<td>Total energy saved by outsourcing the Fibonacci loop</td>
<td>56</td>
</tr>
<tr>
<td>8-2</td>
<td>Total energy saved by outsourcing the matrix multiplication loop</td>
<td>56</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Improvements in laptop technology from 1990-2001</td>
<td>24</td>
</tr>
<tr>
<td>5-1</td>
<td>Framework for computation outsourcing at runtime</td>
<td>33</td>
</tr>
<tr>
<td>5-2</td>
<td>Steps for executing a client program under the outsourcing framework</td>
<td>34</td>
</tr>
<tr>
<td>5-3</td>
<td>Resource selection rule</td>
<td>37</td>
</tr>
<tr>
<td>5-4</td>
<td>State transition rule</td>
<td>37</td>
</tr>
<tr>
<td>5-5</td>
<td>The send rules for the client and the server</td>
<td>38</td>
</tr>
<tr>
<td>5-6</td>
<td>The receive rules for the client and the server</td>
<td>38</td>
</tr>
<tr>
<td>6-1</td>
<td>Overview of compilation and optimization process</td>
<td>40</td>
</tr>
<tr>
<td>6-2</td>
<td>Example of C program passed as input to vpcc</td>
<td>43</td>
</tr>
<tr>
<td>6-3</td>
<td>Resulting INF file for the program in Figure 6-2</td>
<td>43</td>
</tr>
<tr>
<td>6-4</td>
<td>Algorithm to show implementation of the pseudo-parser</td>
<td>45</td>
</tr>
<tr>
<td>6-5</td>
<td>Algorithm to calculate loop data size once all of the variables in each loop have been collected</td>
<td>47</td>
</tr>
<tr>
<td>8-1</td>
<td>The C program used to estimate the energy cost of executing an empty for loop 100M times</td>
<td>54</td>
</tr>
<tr>
<td>8-2</td>
<td>The C program used to estimate the energy cost of the ldr instruction</td>
<td>54</td>
</tr>
<tr>
<td>8-3</td>
<td>Local versus remote execution of 300 iterations of a Fibonacci loop executing 300,000 times</td>
<td>57</td>
</tr>
<tr>
<td>8-4</td>
<td>Local versus remote execution of matrix multiplication of a 400x400 matrix</td>
<td>58</td>
</tr>
<tr>
<td>8-5</td>
<td>Local versus remote execution of a bubble sort loop which sorts 50,000 integers</td>
<td>59</td>
</tr>
<tr>
<td>8-6</td>
<td>Input file for the ray tracing application</td>
<td>60</td>
</tr>
</tbody>
</table>
8-7  The 2-D image representing the 3-D scene generated by the ray tracing application for a 200x200 image

8-8  Local versus remote execution for ray tracing three spheres in space and generating a 50x50 image

8-9  Local versus remote execution for ray tracing three spheres in space and generating a 100x100 image

8-10 Local versus remote execution for ray tracing three spheres in space and generating a 200x200 image

8-11 Energy cost for the battery and network monitors’ execution

8-12 Comparison between theoretical and experimental measurements of the total energy consumed

8-13 The amount of base energy consumed based on the server processor's speed

8-14 The effect of changing the communication metrics on the total energy consumed
Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

PERVASIVE COMPUTING APPROACH TO ENERGY MANAGEMENT

By

Ahmed Ahed Abukmail

December 2005

Chair: Abdelsalam Helal
Major Department: Computer and Information Science and Engineering.

We present a novel approach to energy management. This approach is a pervasive computing solution to the energy problem in mobile devices. Its pervasiveness arises from the fact that communication is used and viewed as an opportunity to save energy, whereby certain candidate sections of code (represented by basic program blocks, most likely loops) are outsourced from the mobile device on which they are executing, to a surrogate server machine with an infinite power source. As can be concluded, outsourcing code implies that the data involved in the computation will have to be transferred to and from the surrogate. This approach deviates from the traditional view of communication as a drain on the battery of a mobile device.

The solution presented here is a compile-time solution and optimization, augmented by the necessary run-time support. The high-level source code is augmented by additional high-level code to intelligently (at run-time) allow the application running on the mobile device to outsource basic program blocks to a server. Both client and server applications are a byproduct of our approach, as the original source code is
transformed into a client/server application where the client is installed on the mobile
device and the server is stored on a surrogate machine.

This approach uses a methodology from the domain of real-time systems to
determine the number of loop iterations, and that facilitates a compile-time computing of
executing each loop. We also gathered the necessary information about the size of the
data involved in each loop. This allows for determining the cost for outsourcing each
loop (in terms of the cost for communicating the data to the surrogate). Once both metrics
(computation energy cost, and communication energy cost) have been determined, we
can easily generate a code that allows the application to make the run-time decision of
whether outsourcing is beneficial. This approach is a fine-grain approach to energy
management, because it looks at basic program blocks (as opposed to certain predefined
tasks or functions). For this system to allow the execution of the client/server application,
the necessary run-time support was built to facilitate this behavior. Results showed
significant levels of energy saving, using our benchmarks.
CHAPTER 1
INTRODUCTION

The emergence of mobile and then pervasive computing (as new computing domains) introduced new challenges and research opportunities: one was energy management. These challenges arose from the mobility of used hardware [30]. Such hardware includes devices such as cellular phones, PDA’s, laptop computers, and even MP3 players. The mobility of these devices implies that they are powered by mobile power sources represented by the battery of each device; and that, in turn, implies that the power source is limited.

As these devices become more popular, and their use becomes more apparent and frequent, the need to manage their energy consumption becomes more vital to their operation. This is because the more often a battery needs to be charged, the more often the mobile device is rendered immobile (which reduces the pervasiveness of their applications). Managing the energy consumed by these mobile devices has been an important subject in research and industry communities of both mobile and the pervasive computing. Solutions have been presented at the various levels and layers of the computer system, and often these solutions to the energy problem involve a certain type of tradeoff.

One of the most attractive avenues to energy management are high-level energy-management techniques. Such a good argument was made for handling power management at high level that Ellis [8] proposed a power-based API to allow for synergy between the application and the system. One of the most attractive high-level solutions
to energy management is a compiler-based solution that alleviates or minimizes the need for programmer power-awareness. This is done via compiler optimization. Velluri et al. [36] studied the effect of the traditional compiler optimization techniques on system power (and therefore energy). Results showed that (except for loop unrolling and function inlining) most optimizations increased the energy consumed by the core of the processor. These results were (at least for loop unrolling) confirmed by Kandemir et al. [20].

We developed a new pervasive computing approach to energy management in mobile devices. This approach views communication (wireless communication) as an opportunity to conserve energy as opposed to being a drain on the battery. Sections of code represented by basic program blocks that are most likely to be loops have the opportunity (at run-time) to execute locally on the mobile device, or remotely on a surrogate, by communicating the data involved in their computation back and forth to the surrogate. The decision to choose local vs. remote execution is based on whether it is more energy-beneficial to execute locally, or to communicate with the surrogate and let the surrogate do the computation. Communicating with the surrogate and letting it do the computation is known as computation outsourcing. So, this approach is based on a tradeoff: the tradeoff here is between computation and communication. At runtime, based on the condition of the battery of the device (full, empty, or in between), the application can opt to go into an energy-saving mode and communicate with the surrogate for the opportunity to outsource code. This depends also on the availability of the surrogate device, and user preference, in addition to the condition of the battery. Hence, this approach will be able to discover services within a wireless network. Service discovery
must be simple as the decision-making code is inserted at compile-time. This is accomplished by two simple programs: the battery and network monitors.

While this approach allows the decision to outsource code at runtime, it is considered a compile-time solution. This is done via high-level compiler optimization, by preprocessing the original (non-energy-aware) source code, and generating two versions of the code. The first version is the client, which runs on the mobile device; and the second version is the server, which runs on the surrogate device. The client and the server are generated, by inserting the necessary high-level code, to be able (at runtime) to make the intelligent decision of either outsourcing or not.

To generate intelligent code that determines whether to outsource the code, we used a methodology from real-time systems to calculate the number of loop iterations. It is safe to assume that this methodology will not be able to calculate the number of iterations for every kind of loop. However, it is capable of calculating loop iterations when the loops are nested, and when the loop bounds need to be determined at runtime, and it also allows loops with multiple exits.

Unlike previous work, our study examined energy consumption at finer granularity. We examined certain candidate sections of programs (basic program blocks), and not entire programs, or specific tasks or functions only within a computer program. This finer granularity yields better energy saving, as our test results show. Our study targeted a subset of the C programming language to verify our algorithm and methodology. As a proof of concept for this approach, we used three different kinds benchmarks involving loops with different complexity. We also developed a 3-D graphics rendering application to further show the energy saving benefits resulting from our approach.
CHAPTER 2
RELATED WORK

Prolonging battery life (often called energy management) has long been the focus of research. This problem has many facets, which can be faced by addressing the various components of a mobile computer system. Earlier [1], we authored a book chapter on power-management techniques. These techniques include reducing energy at the system-architecture level, by targeting (reducing) various components of the power equations \( (P=CV^2f) \), where \( P \) is power, \( C \) is capacitance load, \( V \) is supply voltage, and \( f \) is switching frequency. Other techniques targeted the operating system by saving energy involved in communication, by caching, by process scheduling, and by having an energy manager. Additionally, we presented software techniques grouped into two categories: specific application techniques, and compiler-based techniques. These software techniques (like those targeting the operating system) are considered higher-level power-management techniques. Solutions that target energy management usually involve some kind of a tradeoff. This chapter presents the various power and energy management techniques, that we introduced in the book chapter [1], and other techniques and new methodologies related to our study.

Energy-Management Techniques at the Architecture Level

Hardware and architecture are among the first areas researchers investigate when looking into energy and power management. Therefore, they target the CMOS circuitry and the batteries of mobile devices. They also look for ways to develop processors that are energy and power efficient.
Smart Batteries

Battery basics

Every battery has two terminals: one positive and one negative. Electrons flow from the negative to the positive terminal, when the two are connected (when the battery is installed). Electrons are formed on the negative terminal by chemical reaction that occurs inside the battery. This reaction occurs when the electrons travel from one terminal to the other. Rechargeable batteries (the batteries that we are concerned about in mobile devices) suffer from the problem of memory effect: the battery does not fully recharge because it has not drained completely before recharging. Lithium ion batteries solved this problem recently. However, once the battery is built, and as long as it is connected, the reaction occurs draining the battery power, and eventually rendering it useless until it gets either recharged or replaced.

Intelligent power drainage

The aforementioned shortcomings of regular batteries can be summed by saying: they do not lose power/energy intelligently. Once they are connected to a device, the power drainage is going to happen. This motivated the development of the smart battery system (SBS) (http://www.sbs-forum.org/specs/index.html). This battery is different from non-smart batteries because it provides information and system status to the host machine through the System Management Bus (SMBus), and it has its own re-charger. This information can and will be used by the system, to manage power and energy in the mobile device. Operating systems standards and specifications such as advanced power management (APM) (http://support.microsoft.com/default.aspx?scid=kb;en-us;310752), and advance configuration and power interface (ACPI) (http://www.acpi.info). Currently, most laptop computers are shipped with smart batteries.
A few companies in the industry produce products for smart-battery systems. These companies include (but are not limited to) Samsung, Motorola, and Hitachi as battery vendors. Semiconductor vendors that produce chips for smart batteries include companies like Powersmart, Acer, and Adaptec.

**Energy-Aware Processors**

Another way to tackle power consumption in a computer system is to produce processors that consume less energy than regular processors do. A number of microprocessor companies in the industry have products that are energy-efficient. Intel’s Xscale architecture (http://www.intel.com/design/intelxscale) design optimizes energy consumption. Xscale represents an integral building block of Intel’s PCA architecture (http://www.intel.com/pca). In addition, Intel has the SpeedStep technology in building the Mobile Pentium III processor. SpeedStep provides the advantage of dropping to a lower frequency, and lower voltage, when the device is powered by battery; which results in conserving battery life while maintaining a high level of program performance (http://www.intel.com/support/processors/mobile/pentiumiii/ss.htm). Transmeta Corporation (Santa Clara, California) has the Crusoe family of processors (http://www.transmeta.com/crusoe/index.html). The Crusoe is an x86 compatible family of processors that is lightweight, high performance and 70% less power-consuming than other compatible mobile processors. Additionally, the ARM family of processors is widely popular, and is geared toward reducing power consumption while maintaining a high level of performance (http://www.arm.com).
Reducing Power through Circuitry Components

Power and energy consumption equations

The power equation is $P = V \times I$, where $P$ is power, $V$ is voltage, and $I$ is current. This equation can be re-written as $P = C f V^2$, where $C$ is capacitance load, and $f$ is frequency. The energy equation introduces the time factor: $E = P \times T = C f V^2 \times T$, where $T$ is elapsed time. So, the energy equation can also be re-written as $E = V \times I \times T$.

Reducing any of the variables involved in any of these equations will reduce the energy and power consumed. Capacitance load, frequency, and voltage can be managed at the hardware and architectural level.

Voltage and frequency scaling

If voltage or frequency is reduced, then energy and power consumption will be reduced. Therefore, researchers targeted voltage and frequency scaling techniques to achieve that result. Smit and Havinga [31] argued that reducing voltage means reducing performance; and thus additional hardware is needed to balance that difference to keep the same throughput.

Hsu et al. [19] attempted dynamic frequency and voltage scaling. Their results were simulated, and the claim was that their results are similar to those of Transmeta’s Crusoe TM5400. At compile-time, they assign a clock-frequency, and voltage levels for input loop executions. Their idea is to assign deadlines to tasks, and utilize these deadlines to allow for voltage and frequency scaling. Simply, if a task finishes at time $0.5t$, and its deadline is at time $t$, then it will not hurt the execution if the task takes $t$ time to complete. So, at compile-time they identify these tasks and generate code that will allow for lower voltage and lower frequency to be used in the execution. This work supports the
argument that additional hardware will be needed to achieve voltage and frequency scaling.

**Capacitance load reduction**

Reducing the capacitance load is another method to reduce the energy consumed by a computer. Two main mechanisms can be utilized to handle this goal. Smit and Havinga [31] suggested that memory capacitance is greater than that of the processor. Therefore a reduction in memory operations, to keep as much work as possible on the processor and in registers is beneficial. Gebotys [14] also has done this work by reducing the off-chip activity. The solution is basically to use a minimum cost network flow to find minimum energy solutions by simultaneous partitioning of variables into memory and possibly allocate them into registers.

Another mechanism to reduce the capacitance load is to reduce the logic state transition when handling memory addresses. In any processor the program counters tend to change multiple bits when accessing the next instruction or data from memory locations and that depends on the size of the instruction and if that instruction was a jump instruction or a sequential instruction. That transition increases the capacitance load. To minimize the transition, a Gray code solution can be utilized to facilitate memory addressing. Su et al. [33] did this solution along with a compiler. Sequential Gray codes vary only in one bit at a time. Therefore, there will only be a variation of one bit, which is the minimum that can be achieved when accessing memory addresses (excluding the jump instructions).
Power Reduction through Architectural Design

Architectural design always seems be an attractive solutions to the energy problem, and that is due to the fact that you can always leave the architecture the same but add to it certain components which will result in reducing energy and power consumption. Bellas et al. [4] introduced a compilation methodology. At the hardware level, a second level cache was added to support the new compilation strategy. This cache is a smaller cache and is much simpler to use than that of the first level cache.

Witchel et al. [37] introduced a direct addressed caches solution, which allows the software to access any cached data without the hardware doing any tag check. Therefore, the energy consumed by the tag check will be saved. A compiler solution was introduced to augment their hardware solution.

Operating System Techniques

Operating system solutions can be a very useful complement to architectural solution to energy management. While most operating systems are distributed with their own energy management solutions, additional techniques and methodologies were also researched to handle this problem.

Energy-Management Solutions

User-configurable energy management techniques are now implemented in operating system. Advanced power management (APM) is one of these techniques. In APM, users can define their desirable power scheme. Some of the variables that users can control are the screen, the hard drive, and suspend/sleep modes. Users can also instruct the screen and the hard drive to stop/sleep when the system has been idle for a while. Additionally, they can also specify if the system is idle, or, in the case of a laptop, the lid
is closed, at these points the whole computer can save its state on the hard drive, sleep, and when it wakes up it will restore itself back to the previous state.

Intel discussed some shortcomings of APM such as not considering future capabilities of PC’s that include their communication capabilities because APM drops the communication link. So, they came up with their own solution in “Instantly Available PC” which uses the ACPI specification to manage power while keeping the system online via either the modem or a LAN card. So, Intel and other industry partners contributed to the ACPI specification.

**Memory and I/O Management**

Memory access is expensive when it comes to energy consumption. Additionally, secondary storage (disk) is even more expensive. So, the more the computation is left on the processor, and the storage is done in register the better it is for energy consumption. For instance, making fewer incorrect file predictions is a good methodology to save energy [38]. This was accomplished by observing: the probability and repeated history of file access do not occur without a reason. They claim that programs access more or less the same files in the same order when they execute every time, giving themselves a good knowledge for determining what they call program-specific last-successors for each file.

**Communication Techniques**

Energy saving communication techniques are getting increasingly important. This is due to the wide use of the Internet by almost every user of a computer. So, finding ways to save power on communication will definitely be beneficial.

Communication systems have three states, sending, receiving, and idle. Switching between these three states is energy consuming [31]. Therefore, reducing the transition from one state to another will definitely reduce power loss. Therefore, instead of serving
the communication request of sending, we can buffer the data to send it in large chunks. The same thing can be done on the receiving end. We can only accept data that is of a certain size optimized for our power needs. Otherwise, our system needs to go into an idle mod. In addition, with the availability of solutions that conform the ACPI specification, there is no need any more to modify network protocols to support full connectivity while the system is in sleep mode.

Kravets and Krishnan [21] provided a method for managing the communication’s device on a host through suspend and resume operations. They claim that a communication device will continue to draw power unless it is suspended. Their solution takes into account the need for the host to know the communications patterns, so when it is suspended there is no communication going on, otherwise problems such as buffer overflow will occur on the host and other hosts. So, the functionality of the suspend and resume operations is adaptable such that it will avoid such overflow.

Additionally, Loy and Helal [24] introduced an application-based solution that can easily be utilized in developing energy management techniques in the operating system.

**Scheduling**

Energy-aware scheduling in the system requires prior knowledge of the activity inside that system [5]. Monitors can be incorporated within an operating system to handle this. Bellosa [5] introduced an event-driven energy accounting as a way to manage power in an operating system. The way this was done is by online analysis of the energy-usage patterns that are fed back into a scheduler to control the CPU’s clock speed. This work here was done with real hardware as opposed to simulation.
Software Application Techniques

High-level or software energy management techniques are getting highly attractive as solutions to the energy problem. Ellis [8] argued a case for the need for higher-level power management. The gains from higher-level energy management are great at the application level, and this stems from people’s interests. People can enhance on hardware advances on power management, but also when hardware solutions are not available software developers can handle this issue to either complement hardware solutions, or obtain better results without newer, more advanced hardware. At the software, and application techniques there are two important sections, compiler development, and application-specific techniques.

Compilation Techniques

Research and experiments have shown that, with the exception of loop unrolling, and function inlining, compiling for performance does not imply compiling for energy [36]. So, optimization techniques that target energy as opposed to performance are needed, and rightfully so, since compilation for power is extremely important. It is important because it is an attractive solution to programmers that are not energy-aware. In this section we will discuss some techniques that showed promise and some that did not when dealing with compiler optimization.

Reordering instructions

The energy and power equations showed that their values are directly proportional to the frequency of switching the signal from 0 to 1, i.e., logical state transitions. Tiwari et al. [34] mentioned that switching is a function of present input and previous state. So, the previous instruction is a factor in the function. Therefore, reordering instructions can be a factor in reducing switching activity, and therefore, reduce energy consumption. In
utilizing this technique in the 486DX2 architecture, it did not show much favorable results, it only showed little improvement.

**Reduction of memory operands**

Tiwari et al. [34] also showed, via experiments on the 486DX2 architecture, that instructions with memory operands have higher energy consumption than instructions with register operands. Pipeline stalls, misaligned accesses and cache misses add to the cost energy-wise. Compiler optimizations achieve reducing the number of memory operands. In their research, the authors claim that the most efficient way to reducing the memory operands is via register allocation for temporary and frequently used variables, and that also leads to potential reduction in pipeline stalls and cache misses. However there are some issues with register allocation optimization techniques, such as more complex compilers, longer compile time, and register allocation algorithms need to be modified to optimize for low power. Additionally, larger caches, which will result in lower miss rates, will result in this technique being a little less significant.

**Code generation through pattern matching**

Aho et al. [3] introduced code generation through pattern matching and dynamic programming, and was later utilized by Fraser et al. [13] for generating code that optimizes for performance. The idea is to find a cover for intermediate representation (DAG) for each basic block of code. Finding that cover is done using dynamic programming and tree matching so that the overall cost is minimized. The cost function introduced in Aho et al. [3] and Fraser et al. [13], take performance under consideration. Therefore, to optimize for power, the metric used is power usage as opposed to the number of clock cycles used for the instructions. It was suggested that further investigation of this technique needs to be done.
**Code annotation**

Marculescu [25] proposed a methodology to annotate the code at compile-time to adaptively select, at runtime, the optimal number of instructions to be fetched or executed in parallel to save energy. However, this solution would require additional architectural changes.

**Remote task execution**

This is believed to be one of the most attractive compiler optimization techniques because it takes into account the idea that while communication might be a drain on the battery, it can also be an opportunity to save power by shipping the execution elsewhere. The idea of compiler-based remote task execution has been proposed in other research. Kremer et al. [22] have done some work that utilizes compiler-based remote execution of certain tasks in the field of image processing applications. However, their work included only hand compilation of the code and not compiler implementation or any automation of task identification. They basically utilized what they called checkpoints in the code to determine the task’s delimiters. Additionally, Rudenko et al. [29] introduced remote task execution mechanisms. However, this approach migrates entire processes to remote servers, and then waits for their results back to the client remote machine. As far as Java applications are concerned, the work done by Palm et al. [27] introduces the possibility of migrating compilation (both optimizing and non-optimizing) step of running a Java program to a server, and that is due to the power consumption cost of compiling Java programs.

**Application-Level Techniques**

Attacking the power problem at the application level has been done quite extensively. However, the solutions provided are only useful in certain applications, and
most of the times can not be generalized to allow for utilization in other applications. Most of the work seems to have been done in the signal and image processing applications due to the amount of drainage that they inflect on the battery, since they use computation power, and screen power.

Loy and Helal have introduced an active mode power management (AM/PM) where they increase the amount of information available through the use of power aware API [24]. They saved up to 62 times the power used when compared to using no power management. They examined the amount power used by a connected wireless PC card when the connection is not needed. At that point, they take advantage of stealing back as much of the power as possible. The targeted applications in this work were an email client and a web browser.

Flinn and Satyanarayanan [10], demonstrated a collaborative relationship between operating systems and applications to meet user-specified goals for battery life. In this work, applications can dynamically modify their behavior to conserve energy. The way this is done is by the operating system monitoring of energy supply and demand. It is as simple as: if the supply of energy is plentiful applications perform best, otherwise they will be biased towards conserving energy. They used PowerScope [11] to validate the measurements of energy consumption for accurate estimation.

MP3 players are some of the most popular devices in the market at the present time. Haid et al. [16] developed an excellent application with energy awareness in mind. This work presents designing an energy-aware MP3 player. They not only tackle the hardware side of the devise, but rather the software side as well. They talked about the two technologies that influenced the design of wearable computers and these are system-
on-chip (SOC) and System-In-Package (SIP). These two technologies are the reason that both hardware and software solutions will need to be developed for energy-aware wearable devices. In this work, they analyze a single-chip multimedia system to be used in the wearable device. They take into account the entire computing environment such as hardware, packaging, and software design. They achieve low energy consumption by the use of detailed statistical analysis of the energy consumption. They use an in-house designed runtime energy estimation tool.

Additionally, Yuan et al. [39] investigated another multimedia application with respect to power-awareness. They present a middleware framework for coordinating the adaptation of multimedia application to the hardware resources. They have three goals in mind when designing their system, the first of which is a soft real-time guarantee of multimedia application deadline, the second is to have sufficient energy for multimedia applications to finish their task, and the third is to waste as little energy as possible. To meet the three goals the framework presented makes three useful contributions: the first is to make multimedia applications make energy aware processor reservation. Secondly, is to formally model adaptability of hardware, software, and user preference. Finally, deliver sufficient processor and energy resources to the application and operate the processor as slow as possible to save energy.

**Tools for Low-Energy Design and Measurement**

Power and energy estimation and measurement tools are an essential part to developing solutions to the energy problem. As developing and later measuring the power consumed must be accurate to gain benefits from any newly developed technique.
PowerScope

PowerScope allows the measuring and estimation of the energy consumed via profiling [11]. By mapping energy consumption to program structures, it can be determined which sections of a program consume more power. Also, the energy consumed by procedures/subroutines within a process can be determined. By providing this fine granularity of feedback, PowerScope allows one to focus on system components that are responsible for the largest energy consumption. Experiments by the developers of PowerScope yielded 46% reduction in total energy consumed.

The architecture of this system is composed of a data collection phase, and an offline analysis phase. The data collection phase uses an Agilent Technologies’ (http://www.agilent.com) multi-meter connected by a GPIB cable to a data collection computer running an energy monitor component. A profiling computer running an application and a system monitor component is getting power by being connected to the multi-meter for measurement.

Offline analysis utilizes a third component (the energy analyzer) which will take the results produced in the data collection phase by the system monitor, and the energy monitor and correlate them to produce the energy profile of the application.

As in CPU profilers, where sections of code wasting CPU cycles can be discovered, this profiler will allow for pinpointing the sections of code that consume lots of energy, and therefore must be dealt with.

Derivatives of SimpleScalar

SimpleScalar is used for building and modeling applications to analyze program performance, hardware and software co-verification (http://www.simplescalar.com). It is a toolset that is used to develop modeling applications to simulate real programs running
on various modern microprocessors (such as Alpha, PISA, ARM, and x86) and also various operating systems. It has simulators with different granularities (high and low level). SimpleScalar is widely used in the research community. The toolset also includes performance visualization tools, statistical analysis resources, and debug and verification infrastructure.

The licensing model for SimpleScalar allows users to extend the tool set. As a result the power management community took advantage of that to extend the tool set to develop projects for measuring and estimating power consumed by applications. These projects include the Power Analyzer Project at University of Michigan (http://www.eecs.umich.edu/~tnm/power), and the Wattch project at Princeton University (http://parapet.ee.princeton.edu).

**Other Power-Estimation Techniques**

In addition to previously mentioned tools, other research has been done to estimate energy for certain applications, systems, and devices. Cignetti et al. [7] described an energy model for the Palm™. In this work they target challenges faced by the programmers when developing energy efficient code.

**Recent Work**

Recently additional work has been done in the area of remote execution for energy management. Flinn et al. [9] present a remote execution system for mobile clients powered with a battery. The system they describe (Spectra) is a self-tuning system where it monitors resources in the environment, and based on the resources the decision is made to where the components of the application are executed. This approach, while useful and attractive, it is a coarse-grained solution to energy management where opportunities to execute code remotely are done at the application level as opposed to the basic program
block level. Additionally, it does add a certain level of overhead to the mobile device where various monitors need to be installed on the device to monitor the various resources. These monitors include, not only a battery monitor, and network monitors, but they also include, a CPU monitor, and a file-cache monitor.

**Grid Computing**

When dealing with research that targets computation outsourcing via a distributed computing system, you have to consider grid computing as it is a new model that has a great potential in benefiting this type of research. In looking into grid computing [12, 23], it looks like it would be the perfect match for this type of research as it utilizes things like Java remote method invocation (RMI), remote procedure call (RPC). However, most grid computing systems differ in an important aspect and that is the size of the problem being solved. No matter how large a program that we are trying to optimize for energy is, it will never be as large as the programs investigated within the grid computing domain, which tend to be huge in nature and usually require longer time to execute. Additionally and more importantly, the size of the middleware used within a grid computing environment is quite large, and computationally intensive to be utilized on a mobile device for energy considerations, as it will consume additional energy. That is why investigating mobile agent (MA) systems is questionable and must be pursued with care, however RPC and RMI are worth investigating, and actually they are considered for future research.

Another important difference between our work and grid computing is that in grid computing, the geographic location of the resources is of no concern. However, in our research the geographic location is quite important, as if outsourcing is to take place, the mobile device must be located within a pervasive smart space in order to be able
communicate with the surrogate via the wireless network, so proximity is important in our research.

While grid computing, in general, is not applicable as a whole computing paradigm, a simplified version of its behavior can be adopted in this research. When looking at the formal models for grid computing, we were able to recognize a formal model from grid computing that was useful in defining our own model. Nemeth and Sunderam [26] presented a formal model that is based on an abstract state machine (ASM). Their model encompasses a general description of grid and distributed computing. Their definition contains universes (sets) of resources, processes, users, task, etc. It also contains a signature consisting of function names on these universes. Their universes were: a set of processes (PROCESS), as set of applications (APPLICATION), a set of users (USER), a a set of resources (RESOURCE), a set of nodes (NODE) in the case of distributed computing but not for grid computing, a set of tasks within a process (TASK), a set attributes (ATTR) for tasks, resources, and nodes along with a subset describing architectures (ARCH), a set of locations (LOCATION), and a set of messages (MESSAGE) for interaction between processes. Along with these sets, the symbols, true, false, undef; =, and the Boolean operators are included.

The formal model presented, also included the functions that compose the signature on the super-universe containing all the aforementioned universes [26]. The signature is composed of the functions: app: \( \text{PROCESS} \to \text{APPLICATION} \), user: \( \text{PROCESS} \to \text{USER} \), request: \( \text{PROCESS} \times \text{RESOURCE} \to \{\text{true}, \text{false}\} \), uses: \( \text{PROCESS} \times \text{RESOURCE} \to \{\text{true}, \text{false}\} \), mapped: \( \text{PROCESS} \to \text{NODE} \), BelongTo: \( \text{RESOURCE} \times \text{NODE} \to \{\text{true}, \text{false}\} \), installed: \( \text{TASK} \times \text{NODE} \to \{\text{true}, \text{false}\} \), attr: \( \{\text{RESOURCE, NODE, TASK}\} \to \)}
\textit{ATTR}, arch: RESOURCE → ARCH, location: RESOURCE → LOCATION, compatible:

\textit{ATTR} × \textit{ATTR} → \{true, false\}, CanLogin: USER NODE → \{true, false\}, CanUse: USER × RESOURCE → \{true, false\}, state: PROCESS → \{running, waiting, receive\_waiting\}.

\textit{from}: MESSAGE → PROCESS, \textit{to}: MESSAGE → PROCESS, expecting: PROCESS → PROCESS + \{any\}, and \textit{event}: TAST → \{req\_res, spawn, send, receive, terminate\}.

Given the mentioned functions defined within the signature, they proceeded to define the rules for conventional distributed systems. They also, showed the problems of mapping conventional distributed systems rules into grid systems. Finally, they provided the necessary modifications to formulate the mode for grid systems.
CHAPTER 3
STATEMENT OF THE PROBLEM

The problem of energy management has gained a lot of attention in the mobile and pervasive computing research community. This is due to the increased reliance on mobile devices by a wide spectrum of users. This increased reliance on mobile devices stemmed from the increased capability of these devices. This argument was supported by Helal [18] who gave a close look at the market for Java-enabled phones and PDAs from a commercial standpoint to show that the capabilities of these devices are increasing, and will continue to increase over time. J-Phone Communications (Japan) offered handsets that are based on the J2ME technology, which include a built-in camera, a color display, and a 3-D graphics engine that displays images from different perspectives [18]. They also presented plans to offer mobile phones capable of providing geographic information, multimedia service (JPEG, and PNG graphics synchronized with sound). Helal [18] also presented information about Nokia’s plans to double its shipment of Java-enabled phones between the years 2002, and 2003 which gives a direction of where the mobile technology is headed. Helal [18] also hinted to the increased capabilities in PDAs as well by indicating that the PDA Profile (PDAP) might be too limited for where PDAs are headed and might not be able to accurately estimate the capability of high-end PDAs such as the Compaq iPAQ. Enabling mobile phones and PDAs with Java is an indication that the trend is to make these mobile devices capable of performing tasks usually performed on desktop computers.
As the capabilities of mobile devices increases, their energy usage will increase, and therefore will drain the battery quicker. A cellular phone or a PDA drains more power when it is playing an MP3 song, taking a picture, decoding and image, or recognizing a voice to voice-dial a phone number. The problem has been that the battery technology has not kept up with the other technologies comprising a mobile computer system such as PDAs and cellular phones.

Starner [32] gave a discussion of how much slower advances in battery technology have been than those for the other mobile computer components (the discussion was given for laptop computer, but information for wearable computers, PDAs and cellular phones was deduced to be similar). Figure 3-1 provides a graph representing the improvement in laptop technology from 1990-2001 [32]. As the graph indicates, CPU speed has kept up with Moor’s Law, but battery capacity has not. In fact the figure suggest that the battery capacity is lagging behind. It was also stated that mobile phone companies sell more batteries than they do phones as the lifetime of the phone is much longer than that of the battery, and consumers usually must buy more than one battery to be able make use of their mobile devices effectively. So, battery technology seems to be the slowest technology for the components of mobile devices.

The continued increase in reliance and capability of mobile devices indicate that the energy-saving problem is ongoing, and it needs to continue to be addressed on the long run. Due to their capability, mobile device users are ranging from teenagers to the elderly and they span a wide range of backgrounds and mobile device utilization. One of the most widely used feature set of mobile devices is that dealing with multimedia applications, especially among the youth. The young generation uses these devices to
play video games; take and edit pictures and videos and record and play sounds and music on them. All of these applications consume significant energy. Not only does the young generation use multimedia applications; think of the medical and dental professions. They deal with X-rays, three-dimensional (3-D) magnetic resonance images (MRI) and 3-D representation of the mouth (dental application) for determining the location of a tooth implant. Other users could include a graphics designer working to generate a 3-D model of a home, or any other scene for that matter. Voice and speech recognition applications are known to have been extremely difficult to perform on a mobile device due to the intensive computation involved.

Figure 3-1. Improvements in laptop technology from 1990-2001.
Consider a fraction of the computation involved in one of the previously mentioned application such as displaying an image representing a 3-D scene. The data involved in describing a 3-D scene is small. The output of this computation is a 2-D image display of the 3-D scene. However in order to get from one to the other, the amount of computation is quite large. In our 3-D benchmark (experimental validation chapter), we displayed 3 spheres of different colors in space. In order to generate a 200x200 image representing the scene, the PDA used ran for over 16 minutes to generate the scene, consuming approximately 0.0.47 Amps (Including the operating system and the terminal application, 0.19 without. Given that the battery that we used is a 1700 mAh (milli-Amp Hour) or 1.7 Ah, it will take less than 4 executions of the application to drain the battery of the device. Therefore, multimedia applications are very energy consuming, and must be handled differently.

The emergence of pervasive computing as a byproduct of mobile and distributed computing enabled researchers to envision and later develop new methodologies to solving the energy problem. We observed that mobile users are usually located within communication and energy rich surroundings. This is because of the presence of the mobile user within a wireless network, which usually has other machines that are not mobile (connected via a wire to the network), and thus do not suffer from the energy dissipation problem. Additionally, we observed that there exists no link between the mobile devices located within the wireless network and those non-mobile ones. We concluded that, if a link can be established between the mobile and the non-mobile devices, the mobile devices will be able to save energy by outsourcing some of their computation to the non-mobile devices. The mobile devices will however pay a
communication cost to facilitate the outsourcing. This observation led to the development of our methodology that utilizes pervasive smart spaces to outsource computation to non-mobile devices that do not suffer from the energy dissipation problem.

This methodology is considered pervasive due to the fact that, at runtime, the application intelligently makes the decision between local and remote execution by comparing the energy cost of computation vs. the energy cost of communication. The ability of the application to make such a decision is quite a challenge. Hence, the problem that is being solved in this research is to make ordinary applications, developed by non-energy-aware developers, able to make such an intelligent decision at runtime. Therefore, this approach takes as input a program in its high-level representation (source code), and transform it via a compilation process into two versions, a client version and a server version, where the client is able to send requests to the server to be able to do some computation for the client by receiving data and sending back results (only the modified data). To be able to generate these two versions of the program, a priori knowledge of the runtime behavior of the program needs to be analyzed. Therefore, the area of real-time systems is a very suitable area for utilizing in making such an analysis and determination, especially the ability to calculate the number of loop iterations. In addition, the ability to detect basic program blocks (most likely loops) at both the high-level and the low-level must be accomplished to determine both the communication cost involved in outsourcing a basic program block, and the energy cost of executing a single loop iteration.

To facilitate the runtime decision-making ability for the client, additional functionality has to be added to the mobile device to allow for the outsourcing to take
place when applicable in a pervasive environment. This functionality includes a network
monitor and a battery monitor similar to those introduced by Flinn et al. [9], however
their implementation is much simpler due to the compile-time code augmentation. The
network monitor will be able to find and locate available surrogate machines that are
running an instance of the server. Additionally, a battery monitor needs to be
implemented to monitor the status of the battery. The simplicity of these two monitors
results in a negligible additional power consumption by the system, and will be overcome
by the energy saving from the modified application.
CHAPTER 4
OVERVIEW OF THE APPROACH

To solve the problem stated in the previous section, the solution has to be composed of two parts. The first part of the solution is done at compile-time as an optimization technique at the high-level source code. The second part of the solution must provide the necessary support to the outcome of the first phase. This is due to the fact that the outcome resulting from the compile-time phase is a different formation than that initially developed.

Overview of the Compile-Time Solution

We have introduced this work as part of our fine-grain approach to power-aware computing [2]. First at compile-time, an assumption has been made that the source code has been tested and verified in its original form. Although that is done, this solution still validates the source code syntactically to make sure that no inadvertent errors were introduced along the way. In addition to syntax checking, the source code is also disassembled and the outcome of this process is an assembly representation using the mnemonic representation of each instruction of the target architecture. At this point, information about the high-level source code and the low-level instructions will become available for the optimization technique part of this contributed research.

The next step is to recognize basic program blocks (mainly loops) in both the source code and the assembly code, and simply match them. Recognizing loops at the high-level representation of the source code will result in the ability to collect all the data involved in the computation of the loop, and that will yield the energy cost of
communication for sending all the data involved in the calculation out, and receiving only the data that changes (L-Values). Also, using the technique mentioned by Healy et al. [17], the number of iterations for each loop is calculated. As for the assembly code, the loops are recognized to determine the instructions involved in each loop, which will yield the entire energy cost of executing a single iteration of the each loop. In addition to instructions, at compile-time, we recognized whole library functions such as those belonging to the math library, and we added the value of their energy cost to the cost of the loop in which they are executed. This, along with the metric calculated before to find out the number of each loop’s iteration construct a good estimate of the total cost of the local execution of each loop.

Before calculating the total cost of communication and the total cost of each loop’s computation, experiments were done to find out the cost of communicating a single unit of data (a byte), and the cost for executing each machine instruction for the target architecture. As for calculating the cost of communicating a single byte, a client/server application was tested with multiple sizes of data to communicate between two machines, and the measurement for this was recorded and averaged. As far as each instruction’s energy cost, a similar approach to that presented by Tiwari et al. [35] was utilized where each supported instruction is isolated via high-level code implementation, and executed multiple times within a loop and the final result is averaged based on the number of instructions used (we used 100 instructions within a loop executing 100 million times). In addition to testing machine instructions and verifying their cost, we tested pre-existing library function and verified their energy cost in a similar manner to the individual machine instructions.
Overview of the Runtime Support

To support the ability to outsource code, the application must be able to run in one of two modes: normal mode, or energy-saving mode. So, when an application starts, it will have to get some information based on the resources that are available. If the battery is susceptible to be drained quickly, then the application needs to run in energy-saving mode, the user also has control over this. However, if the user decides to run in normal mode, then the application should not worry about computation outsourcing.

In order for the application to be able to make the right decision, it has to contact the battery monitor at startup. The battery monitor would have already determined if energy saving is available via outsourcing (this decision is based on user preference also). Additionally, the battery monitor will contact the network monitor to check if the devices is actually connected to a network and that network contains surrogate servers. If so, then it will run in energy-saving mode listing the appropriate surrogate available for the application to utilize. This monitor is also similar to, but much simpler than, those discussed by Flinn and Satyanarayan [9] and by Gu et al. [15].

The work done by Flinn et al. [9] suggests that the cost of these monitors is “non-negligible”. This is true in their case, as a lot of the intelligence to execute code remotely is done at runtime as opposed to compile-time, and that is why their approach is a coarse-grained approach to energy management. However in our approach, while may utilize an idea presented by Flinn and Satyanarayan [9] and by Gu et al. [15], the solution is much simpler and that is because the battery monitor is a straightforward inquiry to operating system’s advance power management (APM). As far as the network monitor is concerned, it will only be invoked if an energy-saving mode of operation is decided (mainly as an outcome of the battery monitor). Therefore, the cost is negligible for these
two monitors. Implementation of the battery monitor was as easy as looking at a single file containing information about the battery at certain increments of time. As for the network monitor, several approaches can be investigated, the simplest of which was proposed by Gu et al. [15] and it is based on wireless broadcast for discovering surrogates.

**Implementation Setup**

The implementation was done on an Intel® Xscale which is an integral part of the Intel® PCA. We utilized the Sharp® Zaurus SL-5600 (http://www.sharpusa.com), which contains an Intel® Xscale PXA-250 processor, and is running Linux, as the mobile device. Installed on the Zaurus, was a low-power Socket® Communications’ (Newark, California) (http://www.socketcom.com) low-power wireless LAN card. The outsourcing server is an Intel® x86 machine running RedHat Linux 7.2.

Developing applications on the Zaurus was achieved using Metrowerks® (Austin Texas) Codewarrior for the Sharp® Zaurus (http://www.metrowerks.com). This software comes with a packaged executable to run on the mobile device only during development to be able to debug and/or execute the application on the Zaurus from a Microsoft® Windows™ where Codewarrior is installed. The software is called MetroTRK (Target Resident Kernel). MetroTRK is only used to transfer executables to the Zaurus via our wireless network at the Harris Mobile Computing Laboratory at the University of Florida.

The implementation supported only a subset of the C programming language, and handled the most popular assembly instructions, which mainly access memory, and handle the arithmetic and logic operations within the architecture. Although limited, supporting the particular subset of C we chose was adequate to develop meaningful and useful application to prove our concept.
Outsourcing computation is not a new terminology here. However, the motivation behind outsourcing the computation to a remote server, and the approach under which we are outsourcing the computation is the contribution here. Our goal from this research is to show that an intelligent runtime decision can be made to decide if it is better to execute a section of code locally on the mobile device, or would it be more energy-beneficial to send its data to a remote server, and get the results back.

**Overview**

The overall framework for outsourcing is described in Figure 5-1. The idea is that a server machine accessible via a wireless network can serve as a surrogate server for a host of mobile devices such as handhelds, PDAs and laptop computers. This server at runtime will receive requests from client programs running on any of these devices for outsourcing code to the server.

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 energy. This will result in two version of the program being generated which the programmer will eventually have to compile and install. We believe that this is not a burden on the programmer in any way, and it is not a requirement for the programmer to have any knowledge of energy requirements/constraints.

Once an application is compiled, and two versions have been generated (a server version and a client version), and they are installed on their respective machines, the user
can then execute a client application on the mobile device. This client application executes normally until it reaches a section of code that has been designated as outsource-able (having the potential for outsourcing), this is what we call the outsourcing candidate. Once this section is reached, then the intelligent code that was inserted at compile-time is executed to make the outsourcing decision. As a matter of fact, the candidate code will not be executed (locally or remotely) until the decision making code is executed.

Figure 5-1. Framework for computation outsourcing at runtime

In figure 5-2, an illustration of the outsourcing mechanism at runtime is given. The client runs on the mobile device, and once it reaches an outsource-able section of code, it determines if it is more energy beneficial to outsource or is it more beneficial to execute locally. If the determination is made to outsource, then it will send the data to the server and wait for the results back, otherwise it will continue to execute locally until it reaches the next available outsource-able section of code. At all times, the server running on the
surrogate machine is waiting for requests from client programs. Once it services the client’s request it goes back to waiting for client requests again, which occur once a candidate section of code decides to outsource its computation.

Figure 5-2. Steps for executing a client program under the outsourcing framework

The decision to outsource a section of code is not an arbitrary decision. The mobile machine user must configure it to determine if outsourcing is desirable in the first place. Therefore, there is a battery and a network monitor running on the client machine that
will help in making this determination. The battery monitor will run and ask the user to determine the outsourcing policy that the user chooses. Once that is determined, then the network monitor gets involved to determine the feasibility of outsourcing (if there is no network connection to a surrogate, then outsourcing will not occur. Once the feasibility is determined, then applications can be run in energy-saving mode, otherwise, they will run in normal mode.

Using our model, we envision the development and creation of an entity called a computation service provider (CSP). Different mobile users would subscribe to the CSP in order to service their energy-needs. The subscription will be by registering a copy of the server of the energy-aware application with the CSP. Whenever the user is within the proximity of (in the pervasive smart space containing) the surrogate machine containing the server code, it would be possible for the client to outsource code to the server located on the surrogate. The outsourcing takes place by the client communicating its data to the server, let the server process the data, and then the client will get the results back.

**Formal Model**

Nemeth and Sunderam [26] presented a formal approach for defining the functionality of a grid system. Their approach started by defining distributed systems and showed how a grid system differs from the classic distributed system environment. Our model is a much more simplistic model than that they presented. Our model has a limited number of resources, and a limited number of processes. The resources in our model are the wireless network (WiFi) and the surrogate device. Our two processes are represented by the client and by the server versions of the original code. The model presented is based on an abstract state machine (ASM).
In looking at their model, we realized that their model encompasses a general
description of grid and distributed systems. Our model is a simplified representation of
theirs. In our model, we define the process universe as \( \text{PROCESS} = \{ \text{client, server} \} \), the
resource universe as \( \text{RESOURCE} = \{ \text{wireless\_net, surrogate} \} \), and the location universe
as \( \text{LOCATION} = \{ \text{within-range, out-of-range} \} \). We use the same functions used in the
grid and distributed computing domain, and add two of our own functions which are:
\( \text{execCost}: \text{TASK} \rightarrow \text{VALUE} \), and \( \text{comCost}: \text{TASK} \rightarrow \text{VALUE} \), where \( \text{execCost} \)
is a function that produces the value of the energy consumed by a specific task of a process.
Similarly, the \( \text{comCost} \) produces the value of the energy consumed by communicating the
data for a specific task of a process.

As far as the functions that we use from grid and distributed computing are
concerned, we use the same exact definition presented by Nemeth and Sunderam [26].

The following functions are defined:

- \( \text{user}: \text{PROCESS} \rightarrow \text{USER} \)
- \( \text{request}: \text{PROCESS} \times \text{RESOURCE} \rightarrow \{ \text{true, false} \} \)
- \( \text{uses}: \text{PROCESS} \times \text{RESOURCE} \rightarrow \{ \text{true, false} \} \)
- \( \text{loc}: \text{RESOURCE} \rightarrow \text{LOCATION} \)
- \( \text{CanUse}: \text{USER} \times \text{RESOURCE} \rightarrow \{ \text{true, false} \} \)
- \( \text{state}: \text{PROCESS} \rightarrow \{ \text{running-normal, running-energy-saving, receive-waiting} \} \), we
  modified this function to fit our execution framework.
- \( \text{from}: \text{MESSAGE} \rightarrow \text{PROCESS} \)
- \( \text{to}: \text{MESSAGE} \rightarrow \text{PROCESS} \)
- \( \text{event}: \text{TASK} \rightarrow \{ \text{req-res, send, receive, terminate} \} \)

Upon defining the above functions, and universe sets, the rules for defining our
system as a simplified grid computing system can clearly be defined. We redefined the
resource selection rule (figure 5-3), the state transition rule (figure 5-4), the send rule for
both the client and the server (figure 5-5), and the receive rule for both the client and the
server (figure 5-6).
The resource selection rule:
state(client) := running-normal
state(server) := receive-waiting
if  loc(wireless_net) = within-range &
    request(client, wireless_net) = true) &
    CanUse(user(client), wireless_net) then
    request(client, wireless_net) := false
    uses(client, wireless_net) := true
    if  request(client, surrogate) &
        CanUse(client, surrogate) then
        request(client, surrogate) := false
        uses(client, surrogate) := true
    else
        uses(client, surrogate) := false
        request(client, surrogate) := true
    endif
else
    uses(client, surrogate) := false
    uses(client, wireless_net) := false
endif

Figure 5-3. Resource selection rule

The state transition rule:
if uses(client, surrogate) &
    uses(client, wireless_net) then
    state(client) := running-energy-saving
else
    state(client) := running-normal

Figure 5-4 State transition rule
Client send:
if state(client) = running-energy-saving &
commCost(task(client)) <
execCost(task(client)) then
if event(task(client)) = send(server) then
extend MESSAGE by msg with
  to(msg) := server
  from(msg) := client
end extend
state(client) := receive-waiting

Server send:
if event(task(server)) = send(client) then
extend MESSAGE by msg with
  to(msg) := client
  from(msg) := server
end extend
state(server) := receive-waiting

Figure 5-5. Send rules for the client and the server

Client receive:
if state(client) = receive-waiting &
event(task(client)) = receive(server) then
  if to(msg) = client & from(msg) = server then
    MESSAGE(msg) := false
    state(client) := running-energy-saving

Server receive:
if state(server) = receive-waiting &
event(task(server)) = receive(client) then
  if to(msg) = server & from(msg) = client then
    MESSAGE(msg) := false
    state(server) := running

Figure 5-6. Receive rules for the client and the server
CHAPTER 6
COMPILE-TIME STRATEGY

Our compiler optimization technique for low energy 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 an algorithm described by Healy et al. [17] to find out the number of loop
iterations. The reason this is an intermediate level analysis is because 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. All of the three levels of source code
analysis are embedded in our algorithm.

Overview

This new compilation technique utilizes pre-existing utilities such as the gcc
compiler and Metrowerks® assembler and compiler. We first pass the code that needs to
be compiled to the gcc compiler to make sure that it is syntactically correct, once that is
done, we remove the resulting machine code as it will not be needed. Then we pass the
same source code through our optimization preprocessing, along with the assembly code
generated from passing the original source code through the Metrowerks® assembler, to
generate the two versions of the code, the client and the server. Once the client and the
server codes are generated, then the server is compiled for the target server machine, and
the client is compiled for the mobile device. In our environment, the client is compiled
using Metrowerks® Codewarrior, and the server is compiled on a Linux machine using
the gcc compiler. Figure 6-1 shows the process for optimizing a C program using our technique.

Figure 6-1. Overview of compilation and optimization process
Energy-Optimization Process

Our optimization technique modifies the high-level code (the C source code). The input to this process is a file containing the source code, and a file containing the assembly representation of this source code. Using the source code program, we determine the number of loop iterations, the size of the data involved in the loop execution. Then we determine, using the assembly representation (low-level representation) of the program, the instructions involved, and we calculate the total energy cost for all of the instructions using the energy cost of each individual instruction. In addition to the machine instructions we handle also library functions (such as the math and standard libraries) called within each loop. The energy cost of each individual instruction was calculated using a methodology similar to that presented by Tiwari et al. [35], and we give an explanation of this in our experimental validation. We calculate the energy cost for library functions in a similar manner to that of the machine instructions. Also, we had already measured the cost of transferring one byte of data using our wireless card. Given all of these metrics, we were able to insert socket code within our source program and conditional statements to determine at runtime if it is more energy-beneficial to outsource a candidate section of code (basic program block/loop) or to execute it locally on the mobile device.

Calculating the Number of Loop Iterations

Healy et al. [17] developed a useful utility for predicting the worst-case execution time (WCET) of a program. They provided us with the software that will accomplish this task for C programs. Their algorithm is part of implementing a static timing analyzer for analyzing real-time systems, as predicting the number of loop iteration is essential for analyzing real-time systems. Their approach automatically bounds the number of loop
iterations. They handle nested loops, and loops with multiple exits. Their methodology is implemented by analyzing the register transfer list (RTL) [6], which is an intermediate representation of the source program.

First, they identify the branches that can affect the number of times the loop executes. Secondly, they calculate when each branch can change its direction. Third, they determine when each iteration branch can be reached. Finally, they calculate the minimum and maximum number of each loop’s iteration. If the loop invariant is a non-constant, for the purposes of the timing analyzer they are implementing, they allowed the user to input the minimum and maximum values for this variable, and that is not needed for our compiler optimization technique, as our methodology supports non-constant loop invariant as, at runtime, its value will be known, and we can use a formula involving the invariant to be multiplied by the cost of executing each loop once (the energy cost) which gives us as a formula that is easily evaluated at runtime to determine if a section of code should be outsourced.

Their implementation is integrated in the implementation of the Very Portable C Compiler (vpcc) introduced by Benitez and Davidson [6]. The input to the modified vpcc (we will refer to it as vpcc) is a source program with a “.c” extension, and the output is a set of files, only one of which is of interest to us, and that is the file with the same name as the source program, except with a “.inf” extension (the INF file). The INF file contains information about the maximum and the minimum number of loop iteration, and we are only interested in the maximum number of iteration in our research and that is because we do not want to under-estimate, we want to outsource with a high degree of certainty that a benefit will be gained from outsourcing. In figure 6-2, we give a sample C program
that can be compiled with vpcc, and by passing certain switches to it an INF file (Figure 6-3) will be generated.

```c
main()
{
    int i, j;

    for (i = 0; i < 100 - j; i = i + 3) {
    }
}
```

Figure 6-2. Example of C program passed as input to vpcc

```text
-3
main
! loop 0 0 1 1 -1 -1 1 2 3 4 -1 4 -1
! loop 1 1 -4 r[10] 0 r[9] 3 s -2 (100-1_j-2)/3 (100-1_j-2)/3 -1 -1 3 -1 3 -1
! block 1 lines 5-5 preds -1 succs 2 4 -1
makes_unknown 3 -1
doms 1 -1
   1 82 4 0 8 7 () 1024 7 (100) 8 4 (%o1)
   1 90 4 0 8 4 (%o1) 8 4 (%o3) 8 4 (%o1)
   1 90 7 1 1024 7 () 8 2 (%o1) 8 7 ()
   1 62 4 2 2048 4 () 0 0 () 0 0 ()
   1 82 4 0 8 7 () 1024 7 () 8 4 (%o2)
! block 2 lines 5-5 preds 1 -1 succs 3 -1
makes_unknown 3 -1
doms 1 2 -1
   2 32 4 0 8 4 (%o2) 1024 7 (3) 8 4 (%o2)
! block 3 lines 5-5 preds 3 2 -1 succs 4 3 -1
doms 1 2 3 -1
   3 90 4 1 8 4 (%o2) 8 4 (%o1) 8 7 ()
   3 74 4 2 2048 4 () 0 0 () 0 0 ()
   3 32 4 0 8 4 (%o2) 1024 7 (3) 8 4 (%o2)
! block 4 lines 5-5 preds 1 3 -1 succs -1
doms 1 4 -1
   4 80 4 0 128 4 () 8 7 () 0 0 ()
   4 15 4 0 0 0 () 0 0 () 0 0 ()
```

Figure 6-3. Resulting INF file for the program in Figure 6-2. The boldfaced expression represents the maximum number of loop iterations
Loop Data and Iterations Acquisition

The first stage of our technique is to recognize the maximal basic program blocks (most likely these blocks will be loops). These basic program blocks (loops) will constitute the opportunity for optimization (candidate code for outsource-ability). Once these basic program blocks are recognized at the high-level, then we collect all the data elements associated with them, and determine the beginning and end file positions of these loops. Additionally, we pass the relevant sections of the original source code to the program that calculates the number of iterations for each loop.

The first part of this stage is to implement a parser-like module (we call it the pseudo-parser) to recognize basic program 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 did not need to implement a full parser here as the syntax has already been checked before entering this stage of the algorithm. In figure 6-4 we present a high-level algorithm for our pseudo-parser. In addition to acquiring the data elements involved in the calculation of each loop, we also determine which C library functions have been called to determine the contribution of their energy cost to the execution of the loop. Additionally, we determine if certain loops are not outsource-able. All loops that involve Input/Output (I/O) routines are designated as non-outsource-able. This holds true also for those loops that include nested loops with I/O functions.

The second part of this stage is to figure out the number of loop iterations and associate each number with each loop calculated by the pseudo-parser. The algorithm to do this is a very simple one. This algorithm is implemented using a very simple parser that parses only the lines that contain the minimum and maximum iterations for each loop in the INF file. Once it extracts the expression representing the maximum number of
iterations, it cleans it up by removing any extra characters such as those in figure 6-3
where the expression is “(100 - 1_j-2)/3”. This particular expression is unique also as it
contains a ‘/’ which could be problematic as if everything used in the expression is an
integer then at runtime, integer division might happen, and therefore after we remove the
substring “.1_” from the expression and if we recognize the division operator, we insert
the typecasting “(double)” right before it. Hence, the resulting expression is “(100 – j – 2)
/ (double)3”.

Pseudo-Parser
Input : A C program
Output: An array of loop data structures containing:
   a. The beginning and ending file position of the loop
   b. The variables and their sizes
   c. The L-Valued variable and their sizes

get a token
nestLevel = 0
loopNumber = 0
initialize a stack of loops to empty
while there are tokens in the program do
   if the token is a loop
      loopNumber = loopNumber + 1
      nestLevel = nestLevel + 1
      assign loopNumber to the loop just entered
      assign nestLevel to the loop just entered
      push the current loopNumber on the stack
   else if the token is part of an expression including functions and structures
      if nestLevel > 0
         if you encounter a variable
            if the variable is an L-value
               insert the variable as an L-value in the loop on top of the stack
            else
               insert the variable as a variable in the loop on top of the stack
            end if
         end if
      if you encounter and I/O function then
         The loop becomes non-outsourcable.
      end if
   else if you reached an end of a loop
      nestLevel = nestLevel - 1
      pop the top of the stack
   end if
get the next token
Done.

Figure 6-4. Algorithm to show implementation of the pseudo-parser
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, if the variable is an L-value (changes), then we add the size of the variable to our sum for the L-valued variable. This is a very important aspect of this algorithm, as if the data does not change, we only need to communicate it to the server, and we do not need it back, but if it changes, then we will expect it to be sent back to the mobile device. This way, we can minimize the amount of communication needed. Additionally, we check if the loop contains other loops, and if so, then we collect the variables of the nested loops only if these variables have not already been collected by a parent loop. The algorithm in figure 6-5 illustrates this procedure.

At this point, we have the data size for each loop which when multiplied by the cost to send one byte of data gives us the total cost to send all the data within the loop added to the cost to receive all the L-valued data within the same loop.

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. Loops can be identified by three consecutive instructions, the first of which is the compare instruction “cmp”, followed by a conditional branch “ble, blt, bge, bgt, bne, beq”, followed by an unconditional branch “b”. The unconditional branch is quite useful in this regard as it sends the control outside of the loop, and all we have to do is to go to that branch location, and find the other unconditional branch that completes the loop and returns us back two instructions before the “cmp”. This way we are able to
Calculate Loop Data Size:
Input: An array of loop data structures
Output: The same array updated to include the data size of each loop

for each loop in the program do
    set the totalDataSize of the current loop to zero
    set the totalLvaluesSize of the current loop to zero
    for each variable within the loop do
        add the size of the variable to totalDataSize
    done
    for each L-valued variable within the loop do
        add the size of the variable to totalLvaluesSize
    done
    for each loop nested within the current loop
        for each variable in each nested loop
            if the variable does exist in the upper loop
                add its size to totalDataSize
            end if
        done
    for each L-valued variable in each nested loop
        if the variable does exist in the upper loop
            add its size to totalLvaluesSize
        end if
    done
done

done

done

done

done

done

done

done

done

done

done

done

done

done

done

done

Figure 6-5. Algorithm to calculate loop data size once all of the variables in each loop have been collected. This is only done for outsource-able loops.

identify or rather delimit where the assembly code for each loop starts and where it end. However, when loops are nested, we need more information 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 with which loop, and that information 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 adding their pre-measured energy 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., \texttt{str} before an \texttt{ldr}), we add the measured pipeline stall energy 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.

**Insert Outsourcing Code**

The implementation of this code was very large, but it was not difficult. As our pseudo-parser generated for us 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, and variable declaration (we declared them globally). The outsourcing code is only inserted for those loops that are flagged outsource-able.
CHAPTER 7
RUNTIME 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 when they are doing any 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.

**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 will contact the network monitor that will complete the task of setting up the device in an energy-saving mode.

**Network Monitor and Surrogate Service Discovery Server**

Once the network monitor is called, it will send out a broadcast that will be only 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 for servicing the device and 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 checks if the energy-saving file exists, and if so, then extract the information about the server from it, establish the connection, and execute in energy-saving mode, otherwise, execute in normal mode.
CHAPTER 8
EXPERIMENTAL VALIDATION

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.

Setup

Our target architecture is an Intel® Xscale which is an integral part of the Intel® PCA. We chose the Sharp® Zaurus SL-5600, which contains an Intel® Xscale PXA-250 processor, and is running Linux, as our mobile device. Installed on the Zaurus, is a Socket’s® low-power wireless LAN card. 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. This software comes with a packaged executable to run on the mobile device only during development to be able to debug and/or execute the application on the Zaurus from a Microsoft® Windows™ where Codewarrior is installed. The software is called MetroTRK (Target Resident Kernel). We use MetroTRK to transfer executables to the Zaurus via our wireless network at the Harris Mobile Computing Laboratory at the University of Florida (http://www.harris.cise.ufl.edu). This Microsoft® Windows™ machine on which Codewarrior is installed also happened to be the same machine on which we record our measurements.
For measuring the energy consumed, we used an Agilent® 34401A multi-meter which was connected to a Microsoft® Windows™ 2000 desktop computer via an IEEE488.1 General Purpose Interface Bus (GPIB) cable. Installed on the desktop is Agilent®’s IntuiLink plugin which works with Microsoft® Word and Excel. We used Excel because its plugin allows for multiple readings as opposed to Word’s single reading. The multi-meter has two J-hooks that were placed in series between the AC adapter and the Zaurus to place the multi-meter in series to measure the total current drawn by the Zaurus. The voltage coming from the AC adapter remained at a constant 5 volts. Therefore, the only two factors in the energy equation are the current drawn and the time in seconds as energy is given by the equation:

\[ E = V \times I \times T \]

Where \( E \) is the energy consumed, \( V \) is the voltage, \( I \) is the current, and \( T \) is the elapsed time.

To measure the energy consumed by either a running process on the Zaurus, or by data communication, we calculate the difference between the current drawn when the Zaurus is idle, and when the Zaurus is either running a process or sending/receiving data. To make this as accurate as possible, the only application that we ran on the Zaurus was the Linux Terminal. We also turned off the light of the Zaurus LCD. These measures that we took, and as our experiments show, resulted in a constant current drawn by the Zaurus while idle, which gave us the ability to get good measurement with as little sampling noise as possible. The multi-meter only allowed us to take samples at one tenth of a second. Therefore, we had to execute code that consumes enough time to allow for as accurate a measurement as can be obtained.
Instruction-Level Energy Cost Estimation

Before applying our optimization to source code, and besides knowing what machine instructions were used, we had to know the energy cost for each machine instruction involved. Our work targeted only a subset of instructions from the Xscale architecture, which was sufficient to testing our benchmarks, and by no means is that a limitation of our approach.

We used a methodology similar to that described by Tiwari et al. [35] to estimate the cost of each instruction. The methodology suggested executing several instances of a single machine instruction within a loop and average the energy consumed to obtain the per instruction cost. Figure 8-1 shows a small C programs which when executed gives a good estimate of the energy consumed by an empty for loop. Figure 8-2 shows a second C program that gives a good estimate for the load instruction (LDR), which loads a memory location into a register. Notice, to minimize the affect of the instructions that execute the loop header (as the loop body is the several occurrences of the load instruction) we used a register loop control variable. The ability to declare register variable in a language like C made it very possible for us to be able to isolate single instructions which enhanced the accuracy of our estimation.

In order to minimize the effects of the cache, and to better estimate inter-instruction energy consumption, we developed additional benchmarks that include multiple instances of a series of different instructions (e.g. ldr followed by str) and we observed their behavior, and recording the amount of additional energy consumption recorded, and used these metrics in our code to update the instruction-level energy consumption values calculated before. In addition to minimizing the effect of caching the instructions, our
benchmarking of each instruction, also, estimated the effect of the pipeline data hazards involved in the execution of each instruction.

**Measuring Communication Cost**

To measure communication cost per byte, we wrote small UNIX socket programs that sent data back and forth to a server, and our results confirmed the card statistics that were mentioned on the datasheet of our wireless LAN card. The data sheet suggests that this card is active 90% of the time transmitting at 265 mA and receiving at 170 mA.

```c
main()
{
    register int i;

    for(i = 0; i < 100000000; i++)
    {
        ;
    }
}
```

Figure 8-1. The C program used to estimate the energy cost of executing an empty for loop 100M times

```c
main()
{
    register int i;
    register int x;
    int a, b, c, d, e, f, g, h, m, n;

    for(i = 0; i < 100000000; i++)
    {
        // repeat 10 time the following 10 statement
        // each line represents an ldr instruction
        x = a;       x = f;
        x = b;       x = g;
        x = c;       x = h;
        x = d;       x = m;
        x = e;       x = n;
    }
}
```

Figure 8-2. The C program used to estimate the energy cost of the ldr instruction
Simple Experimental Validation

To test the effect of our approach on energy, we implemented three different simple benchmarks that span three different formations of data and execution complexity. The first of which was the Fibonacci loop, which contains constant data, but it executes in $O(n)$ time. In other words, the size of the data remains constant, while the execution changes with $n$, where $n$ is the number to which we are trying to calculate the Fibonacci number. Due to the sampling limitation of our multi-meter, we had to test this 3 times using 3 large numbers to get more accurate results of our measurements. We performed the testing using the numbers, 100000, 200000, and 300000. Another benchmark that we used was a rectangular version of the bubble-sort loop which executes in $O(n^2)$ and the data size is linear. So, as the data size grows so will the computation complexity. We sorted 10000, 20000, and 30000 integers. The last benchmark that we used was a square matrix multiplication loop, which runs 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.

Each one of the benchmarks was executed on the Zaurus before our optimization and after our optimization. Table 8-1 gives the percentage of total energy saved by outsourcing the loop that calculates the Fibonacci number. Table 8-2, shows the total energy saved from outsourcing the square matrix multiplication loop. As can be concluded from the tables, a significant amount of energy can be saved when computationally expensive code can be outsourced.

One interesting observation in our testing was the bubble-sort loop. The smallest energy saving was 98% for sorting 10,000 integers. This result is well expected due to the fact that compared to the amount of computation involved in bubble-sort, the size of the
data is very negligible. That is not the case in the matrix multiplication and the Fibonacci sequence calculation.

Table 8-1. Total energy saved by outsourcing the Fibonacci loop

<table>
<thead>
<tr>
<th>Value of $n$</th>
<th>Energy savings %</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>60</td>
</tr>
<tr>
<td>200000</td>
<td>80</td>
</tr>
<tr>
<td>300000</td>
<td>87</td>
</tr>
</tbody>
</table>

Table 8-2. Total energy saved by outsourcing the matrix multiplication loop

<table>
<thead>
<tr>
<th>Number of rows</th>
<th>Energy savings %</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>73</td>
</tr>
<tr>
<td>300</td>
<td>80</td>
</tr>
<tr>
<td>400</td>
<td>88</td>
</tr>
</tbody>
</table>

Figures 8-3, 8-4, and 8-5 show comparison histograms between the local executions of the benchmarks vs. their remote execution (outsourcing). The results were generated while executing each version on the Zaurus while the IntuiLink software is recording the data into a spreadsheet.

**Large-Scale Benchmark Validation**

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, a huge amount of calculation has 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 huge
Figure 8-3. Local versus remote execution of 300 iterations of a Fibonacci loop executing 300,000 times. A) The histogram showing the current drawn during the time intervals listed for local execution. B) The histogram showing the current drawn during the time intervals listed for remote execution (longer spikes are for sending while the shorter ones are for receiving). Voltage remains a constant 5 volts for both histograms.
Figure 8-4. Local versus remote execution of matrix multiplication of a 400x400 matrix. A) The histogram showing the current drawn during the time intervals listed for local execution. B) The histogram showing the current drawn during the time intervals listed for remote execution. Voltage remains a constant 5 volts for both histograms. The energy saving is obvious in this benchmark.
Figure 8-5. Local versus remote execution of a bubble sort loop which sorts 50,000 integers. A) The histogram showing the current drawn during the time intervals listed for local execution, and it took approximately 6 minutes. B) The histogram showing the current drawn during the time intervals listed for remote execution, and it took only 1.2 minutes. Voltage remains a constant 5 volts for both histograms.
when compared to the amount of communicating each unit of data involved in the computation. Our experimental results show 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 8-6, shows the input data passed to the ray-tracing process and figure 8-7 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 8-6. Input file for the ray tracing application

Figure 8-7. 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. Figures 8-8, 8-9, and 8-10 show the results of comparing local execution vs. remote execution for the 3 sizes of images mentioned above for the same scene.

**Other Measurements**

To accurately, and comprehensively discuss our results, other metrics and overhead costs had to be measured once our approach is applied and utilized. One of these metrics is represented by the size of the executable. The size of the executable was impacted greatly when it involved our first three benchmarks as they are considered small programs. The increase in executable size was 7 times for sort and matrix multiplication routines, and 3.5 times for the Fibonacci calculation. However, it was more encouraging when we dealt with much bigger application (which is going to be the case in real life). The size of the executable increased by only 40%.

In addition, we measured the total execution cost of the battery and network monitors to realize that they consume very little energy when running, as the work involved in their execution is very sporadic and very minimal. Figure 8-11 illustrates the total energy consumed by the monitors (only when they are running), otherwise the cost is near zero.

Compile time was another metric that we measured. It can easily be concluded that the compile-time will at least double, as the result of our approach will generate two files instead of one. However, in the process of generating two files, our energy optimization process at compile-time consumed about 1-2 additional seconds for the simple benchmarks, and about 5-7 seconds for the more complex benchmark of ray tracing.
Figure 8-8. Local versus remote execution for ray tracing three spheres in space and generating a 50x50 image. A) The histogram showing the current drawn during the time intervals listed for local execution (approximately 66 Joules in 1.2 minutes). B) The histogram showing the current drawn during the time intervals listed for remote execution (approximately 2.5 Joules in 2 seconds). Voltage remains a constant 5 volts for both histograms.
Figure 8-9. Local versus remote execution for ray tracing three spheres in space and generating a 100x100 image. A) The histogram showing the current drawn during the time intervals listed for local execution (approximately 240 Joules in 4 minutes). B) The histogram showing the current drawn during the time intervals listed for remote execution (approximately 2.5 Joules in approximately 3 seconds). Voltage remains a constant 5 volts for both histograms.
Figure 8-10. Local versus remote execution for ray tracing three spheres in space and generating a 200x200 image. A) The histogram showing the current drawn during the time intervals listed for local execution (approximately 1020 Joules in 17 minutes). B) The histogram showing the current drawn during the time intervals listed for remote execution (approximately 2.5 Joules in approximately 6 seconds). Voltage remains a constant 5 volts for both histograms.
Figure 8-11. Energy cost for the battery and network monitors’ execution. Only the current and the time are shown as the voltage is a constant 5 volts throughout the experiments.

**Normalization of Results**

In order to show the normalization of energy cost for an outsourced application, the processor speed of the server has to be taken into consideration. This is due to the overhead energy (base energy) spent between the time it sends the data to the server and the time it receives the results back. This energy spent is taken into consideration as it lengthens the time the processor is active and therefore it can not go into sleep mode (which saves energy). The amount of energy spent during sleep mode is actually quite negligible and irrelevant to the calculation of the energy consumed. Therefore, the energy consumed by an application depends on the base energy as well as the communication cost.
In our measurements, the base energy consumed by the mobile device was calculated at 0.25 Amps * 5 Volts * (server execution time). Add to it the communication energy cost, the result will be the total energy consumed by each loop executing remotely.

First of all, a model has to be devised for the local execution of each program. Assume that we are consuming PowerBase watts without running any applications on the mobile device. Our mobile device executes MipsPower MIPS/watt. Therefore, if our application requires AppInst Million instructions to execute, and it takes TLocal seconds to execute. That means that the total energy consumed during the execution of the application is given by the following equation:

\[ E_{local} = T_{local} \times \left( \text{PowerBase} + \frac{\text{AppInst}}{\text{MipsPower}} \right) \]

Given that \( T_{local} = \frac{\text{AppInst}}{\text{mips}} \), where the processor is capable of mips MIPS (should be a number very close to the processor speed). The equation for calculating the energy consumed locally can be rewritten as follows:

\[ E_{local} = \left( \frac{\text{PowerBase} \times \text{AppInst}}{\text{mips}} + \frac{\text{AppInst}^2}{\text{mips} \times \text{MipsPower}} \right) \]

Therefore, the Energy spent in the execution of the application locally depends not only on the time, and power, but it also depends on the speed of the processor. So the larger the amount of mips is, the smaller the amount of energy consumed.

At this point we can calculate the energy consumed by executing the loop (or the entire application if it has only one loop) remotely. Prior to its execution, the mobile device is consuming PowerSleep watts, However, after the start of the execution, the
device will spend $Power_{Awake}$ watts as it will wake up. Therefore, the amount of
$Power_{Base} = Power_{Sleep} + Power_{Awake}$. As the value of $Power_{Sleep}$ is not a factor in
the energy consumed by the application (the mobile device is spending it regardless of
what it does), the value of $Power_{Base}$ will be equal to the value of $Power_{Awake}$. The
server device can execute in $T_{remote}$ seconds. The energy cost of sending and receiving
the data is a function of the time to communicate the data, as well as the power cost for
the communications. Therefore, we consider $t_s$, and $t_r$, as the time it takes to send and
receive respectively, and $p_s$, and $p_r$ are the power cost for sending and receiving the data.
Therefore, the energy cost for outsourcing an application is given by the following
equation:

$$E_{remote} = t_s \times p_s + t_r \times p_r + T_{remote} \times Power_{Base}$$

Just like in the case of calculating the time it takes the client to execute a program,
the same calculation can be applied to the server. The server is running $R_{AppInst}$ million
instructions to execute, and it is running on $r_{mips}$ MIPS (or MHz). Therefore $T_{remote}$ can
be given by: $T_{remote} = \frac{R_{AppInst}}{r_{mips}}$, and as a result, the equation to calculate the energy
consumed by executing the application remotely is as follows:

$$E_{remote} = t_s \times p_s + t_r \times p_r + \frac{R_{AppInst} \times Power_{Base}}{r_{mips}}$$

Therefore, the lower the values of $t_s$, $p_s$, $t_r$, $p_r$, and the higher the value $r_{mips}$ (the
faster the server machine is), the less the amount of base energy consumed waiting for the
results. As a result, faster communication (lower $t_s$, and $t_r$) as well as faster server
machines makes a difference when outsourcing computations.
In order to model our normalization of the measurements according to the processor’s speed, we applied the above equation for calculating $E_{remote}$, both, theoretically, and experimentally using our benchmarks. Our 3-D rendering loop was benchmarked using 3 different image sizes (50x50, 100x100, and 200x200). As a result, you can see that the execution time for the applications will quadruple from one run to the other. Also, we used the same size data when running each benchmark (200x200 image) to handle (using the same executable) all benchmarks. Therefore, in effect, we modeled slowing down the processor to $\frac{1}{4}$ of its speed with each run, while keeping the communication cost the same. The processor that we used for the server was a 500 MHz Pentium III (this is also the value of the $rmips$), which means with the 50x50 image, it will have the same speed, with the 100x100 image will effectively model a 125 MHz processor, and with the 200x200 it will be a 31.5 MHz processor. We also estimated, the number of instructions executed for the 50x50 application, and they came out to be approximately 125 million instructions. The estimation was done based on the number of assembly instructions generated as well as the possible logic of the program, so it is only an estimation as opposed to an actual count. The transmission time for sending and receiving was modeled to take 1 second each time with the sending consuming .265 Amps, and the receiving consuming .170 Amps according to the specification of the WiFi card. The value of base current lost was estimated at .25 Amps. The voltage is assumed to be a constant 5 Volts throughout the process.

Theoretically, the value of the energy would be calculated using the values given by the specifications of the network card (.265, and .170) for communication, and the processor speeds for each benchmark (500, 125, and 31.5). Which means that, given the
125 million instructions application, the total amount of base energy consumed would be multiplied by $\frac{1}{4}$, 1 and 4 seconds respectively for each run and that means it should quadruple.

When we measured the values of the aforementioned metrics, we came up with values that are very close to the ones mentioned above. The run for the 50x50 image took approximately $\frac{1}{4}$ of a second, the 100x100 took approximately 1.2 seconds, and the 200x200 took approximately 4.2 seconds. The transmission time was approximately 1 minute for each the send and receive. The current cost for the send was approximately 0.255 Amps, and for the receive it was approximately .170 Amps. Figure 8-12 shows a comparison between the theoretical and the experimental measurements of the total energy consumed for each processor speed. Figure 8-13 shows a comparison between the theoretical and experimental measurements of the amount of base energy consumed. Our results match those acquired by the normalized analysis for the remote execution of the application.

![Figure 8-12. Comparison between theoretical and experimental measurements of the total energy consumed.](image)
Figure 8-13. The amount of base energy consumed based on the server processor's speed

We have mentioned before that the value of $E_{remote}$ also depends on the power as well as the time consumed during the communication ($ts$, $ps$, $tr$, $pr$). In this work, we model the theoretical behavior of the effect of these communication metrics on the energy cost of the equation. We will assume that the processor speed is 500 MHz (500 MIPS), the base current drawn is 0.25 Amps with the same application of an approximate 125 million instructions. Figure 8-14 shows the effect of changing the metrics $ts$, $ps$, $tr$ and $pr$. 
CHAPTER 9
CONCLUSION AND FUTURE WORK

Our experimental evaluations showed that computation outsourcing within a pervasive computing smart space has a great potential in energy management. By exporting CPU processing responsibilities into the network, the mobile device was able to deliver the expected functionality while consuming less energy 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 energy.

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 runtime (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 energy 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 by Pasareanu and Visser [28]. 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.

As part of our future work, we will continue the implementation of the client/server version that we have already started. However, this implementation will contain additional support for dynamic memory allocation and subroutine-level computation outsourcing. In addition, additional work needs to be done to evaluate a more accurate communication cost. This part will be an ongoing part of the research as we utilize additional outsourcing techniques. In addition, we will continue to identify specific applications that will benefit from our approach which will be a refinement of defining categories of application areas that we believe will benefit from this approach.

To handle multiple platforms, targeting languages such as Java and C++ will be necessary. This will require porting a compiler to handle both languages while augmenting with the code that we obtained to calculate the number of loop iterations. Additionally, as we will support additional types of basic program blocks we will build the compiler support that will handle estimating total execution cost of these basic program blocks such as additional types of loops than those supported so far, recursive functions (inherently these are loops), library linked functions whose energy consumption is predefined, and functions defined within the code. Additionally, we will investigate in the case of non-Java languages, the possibility to be able to cross-compile them for the most popular mobile devices.
In this research, we assumed that if a basic program block (loop) contained an I/O operation, it is determined as non-outsource-able. However, in our future work we will be looking at opportunities where I/O operations may be performed elsewhere. For input operations that involve files, if the file exists elsewhere, then it could be energy-beneficial to read the file on a remote machine, and utilize its contents remotely. Similarly, if producing the output elsewhere and basically all we are interested in is a display of this output which maybe cheaper than displaying the output locally, then that is another opportunity that needs to be investigated.

Identification of the applications, which contain the computationally expensive basic program blocks, is an essential part of this research. We will investigate and research the different types of basic program blocks that fall into this category and test them and provide them as benchmarks for our research. In addition, we will build the support for other useful applications that will benefit from our approach.

Java remote method invocation (RMI), and remote procedure call (RPC) based systems will also be applicable in our research where we will let the system make the necessary data communication according to its policies and that will benefit our approach especially at the level of outsourcing specific functions, and subroutines. These two systems are utilized in grid computing environments.

We will weigh the benefits of every outsourcing mechanism with each type of basic program block we investigate, and investigate which mechanism allows us to save more energy with a basic program block. This will lead to a hybrid approach where a single application may contain one, two, or three outsourcing mechanisms.
Also, as far as service discovery is concerned, we will investigate a lighter version of some of the well-known service discovery systems like UPnP (http://www.upnp.org), and Jini (http://www.sun.com/software/jini). We believe that these systems can be utilized without spending a large amount of energy as the intelligence has already been built in the modified program.
LIST OF REFERENCES


BIOGRAPHICAL SKETCH

Ahmed Abukmail received a Bachelor of Science degree in computer and information science from the Ohio State University (Columbus, Ohio) in 1993. In 1995, he received a Master of Science degree in computer and information science from the University of Florida. During the same year, he was admitted to the Ph.D. program.

In 1997, Ahmed left the University of Florida for 5 years to pursue a career in industry. In 2003, he returned to the University of Florida to complete his Ph.D. degree. His major research is in the areas of mobile and pervasive computing with particular interest in energy and power-aware computing, programming languages, compilers, and distributed systems.