Addressing the Energy Crisis in Mobile Computing with Developing Power Aware Software

Selim Gurun
UCSB Computer Science Department
gurun@cs.ucsb.edu

Chandra Krintz
UCSB Computer Science Department
ckrintz@cs.ucsb.edu

ABSTRACT

Reducing program power consumption by resource restricted devices has recently become a very active research area.The driving force behind this interest is the wide-spread popularity of portable computers, handheld devices, and cell phones.Consequently, there is an accelerating demand for increased battery life in mobile devices. One way in which we can increase battery life is to improve battery technology and to produce devices that consume less power. Alternately, we can take a software-based approach. For example, many devices and hardware components are designed with multiple levels of operating power. Application management software (compilers, runtime, and operating systems) can adjust these levels using static and dynamic techniques to reduce program power consumption. Alternately, such systems can select to off-load computation from mobile devices to more capable, wall-powered computers.

Our focus is on software-based techniques for reducing program power consumption. In this paper, we first articulate the problem and describe the state-of-the-art in energy-aware hardware features that can be exploited by software systems. We then present existing techniques that consider static and dynamic information to guide selection of device power levels. We then examine methods for partitioning computation between a battery-powered device and a wall-powered server at the method, task, and program level. Finally, we describe our current research that incorporates and extends these prior works.

1. INTRODUCTION

Reducing program power consumption has become a very active research area in recent years. The most important reason for this is the growing use of portable computers, handheld devices and cell phones. As a result, there is an ever increasing demand for a longer battery lifetime. One way to achieve this is simply to use components that consume less power. Another way is to use hardware components that provide special software optimizations that enable software-selectable power/performance levels or idle/sleep states. A third way to reduce power consumption is to push computation to other computers that do not have power limitations.

Our study is focused on the last two items: software selectable power performance adjustments and computation off-loading. We investigate various applications in terms of energy efficiency, discuss algorithms to better exploit hardware features and study computation off-loading techniques.

Table 1: Computational Resources in Typical Mobile Devices

Device	Wristwatch	Ipaq H3800	Apple PowerBook
CPU	Arm 7	StrongArm	PowerPC
Speed	19Mhz	206Mhz	1000 Mhz
Memory	8MB	64MB	512MB
Storage	8MB CF	32 MB CF	60 GB Hard drive
Display	2.7x2.0 cm	5.7x7.7 cm	17inch (23x36 cm)
Network	bluetooth	802.11b	100/1000BaseT 802.11a(54Mbp/s)

Various software techniques are used to improve hardware utilization. Commonly, these techniques require a trade-off between execution latency and energy consumption. Examples include reducing CPU performance, spinning down the hard drive, hibernating the network interface, and shutting down unused memory banks. There are other techniques that do not increase execution latency but trade off functionality for power consumption. Examples of such techniques include turning off the display or dimming the backlight. In this paper, we limit ourselves to studies that aim to improve energy efficiency of CPU and wireless network interface.

Computation offloading techniques migrate the process either partially or totally to a remote server. One important advantage of such schemes is that it has the potential to improve both performance and energy consumption; especially if the computational resources of the server are much larger than that of the client. However, it is critical to predict the cost of local and remote execution precisely and to migrate only when it is profitable to do so. There are many other challenges to overcome such as server failure, network disconnections, scalability, and security.

We begin with a short discussion of general issues concerning batteries and mobile computing. We describe strategies to decrease power consumption of CPU and wireless interface in section 3. We then discuss various computation offloading techniques to save energy in sections 4-6. Finally, we describe our work in sections 7-11 and then conclude in section 12.

2. BACKGROUND

The importance of energy management is primarily due to limited battery capacity. Unfortunately, the battery technology is improving only at a modest speed and other alternatives, such as ambient energy sources and fuel cells are far from satisfactory [43, 37]. Currently, the highest capacity batteries are based on Lithium-Ion technology; in 1991 a Li-Ion battery was able to achieve as much as 180W-h/L, this increased to 330 W-h/L in 1995 and to 380 W-h/L in 1998[27]. This means that energy density has doubled only once in the past 7 years. In comparison, Moore's law states that the CPU performance doubles each 18 months. Arguably, energy has become one of the most valuable resources in mobile computing[45].

However, it is not easy to define a typical mobile computer. This is because mobile computing includes a wide range of devices with very different characteristics. **Table 1** describes some typical mobile computers: The Apple G4[5], the H3800[6] and the IBM Wristwatch[29]. The Apple G4 is almost as capable as a desktop computer; it has a 1GHz CPU, 512MB RAM and a full set of peripheral devices including a DVD writer/reader, a gigabit ethernet network interface and embedded 54Mb/s wireless interface. However, the other mobile devices have a smaller feature set; the H3800, which is a typical handheld, consists of a 206MHz CPU, 64MB RAM and 32MB CF card as permanent storage. It supports a wireless network interface only through a sleeve unit. The Wristwatch is even smaller; it has a 19MHz CPU, 8M of memory and 8MB of permanent storage.

In particular, **Table 1** emphasizes two important points:

- Architectural differences: Mobile devices do not only have different computational resources, their architectures are also different. For example, the H3800 and the Wristwatch do not have a hard drive; thus, a policy that targets hard drive energy consumption will not be effective on these devices.
- Battery power: Energy is an important resource for mobile devices regardless of their size and configuration. For example as the table shows, all three devices have a batttery lifetime of 2-6 hours.

Architectural differences make it difficult to find a general solution to improving battery life that will target all mobile devices. In fact, it might be difficult to develop a solution even across similar devices. For example, Lorch and Smith[27] compared the power consumption of hardware components in four different laptops. (see **Table 2**). Their results show that CPU is the most important energy consumer in the Duo230 and Duo280c. In contrast, in Duo270c CPU is only the third largest energy consumer. Arguably, an energy saving policy that targets the CPU will be less effective for Duo270c when compared to Duo280c.

3. IMPROVING HARDWARE UTILIZATION

The primary power consumer in a mobile computer is the CPU. Fortunately, there are unique opportunities to improve CPU energy efficiency.

Table 2: Percentage of total power used by each component (taken from [27])

Component	386	Duo230	Duo270c	Duo280c
CPU	4%	17%	9%	25%
Hard Disk	12%	9%	4%	8%
Display	4%	4%	17&	10%
Modem	N/A	1%	0%	5%
FPU	1%	N/A	3%	N/A
Video	26%	8%	10%	6%
Memory	3%	1%	1%	1%
Other	33%	35%	28%	28%
Total	6W	5W	4W	8W

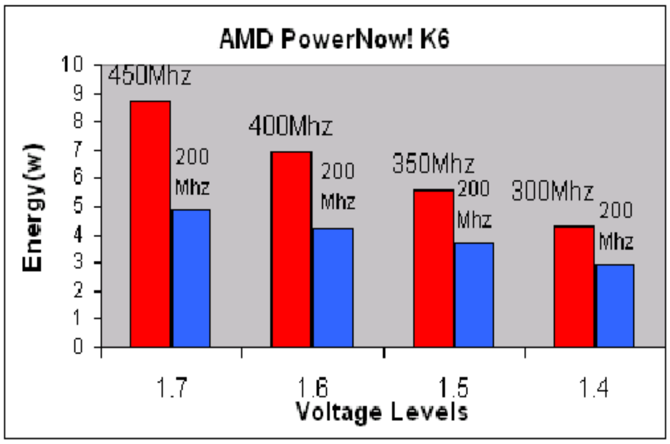

Figure 1: Power/Voltage/frequency relation for AMD PowerNow K6. The y-axis shows power for a given voltage/frequency pair. The red and blue bars shows the maximum and minimum frequency allowed for a given voltage. The minimum frequency level is always 200Mhz, the maximum frequency decrease from 450Mhz to 300Mhz. (taken from [4])

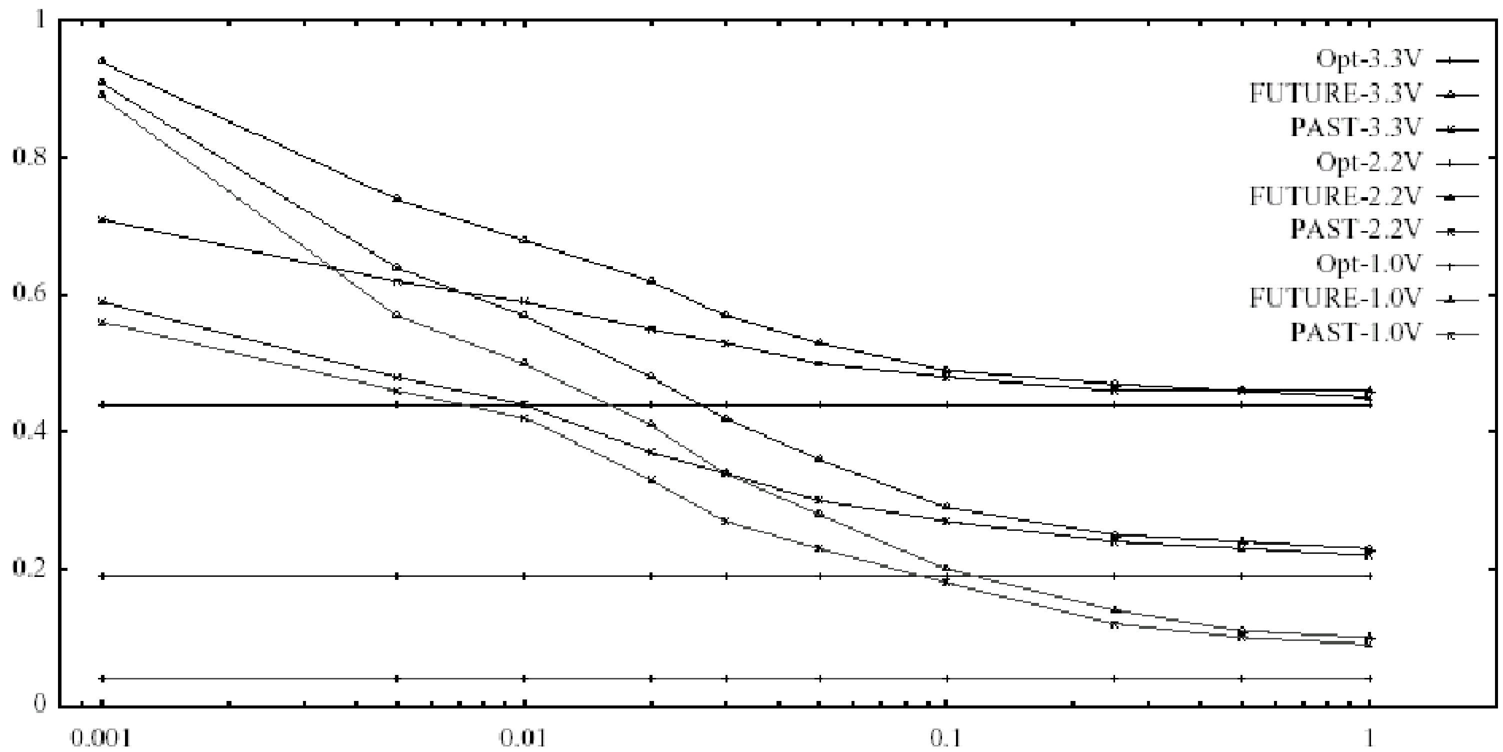

Figure 2: Weiser's findings on energy consumption and interval length. The x-axis is interval length in seconds, note that the log-log scale. The y-axis is the normalized power consumption (assumes power consumption is 1 when there is no clock scaling) (taken from [42])

The CPU is a CMOS circuit. In general, the power consumption of CMOS circuits can be briefly described by the equation below:

$$P = KCV^2 \tag{1}$$

where: K is the clock-switching rate, C is the capacitance and V is the applied voltage level. Thus, it is possible to decrease power consumption by reducing the CPU clock frequency or voltage level[42]. Furthermore, many modern CPU's support a *sleep* mode [19]; they can turn off most on-chip components, bringing down energy consumption to almost zero. A sleeping CPU generally wakes up on any interrupt and resumes. Most CPU's can switch from/to sleep state almost instantly.

Using the sleep state is one of the easiest and most effective power management mechanisms. When there is no task in the ready state, the operating system scheduler goes into an **idle loop**. In mobile devices, instead of going into an idle loop, the OS can put the CPU into sleep mode and sets a timer to wake up after a short period of time.

A second technique is reducing CPU speed. As given by the equation above, CPU energy consumption and clock frequency are linearly proportional, thus scaling down the frequency reduces power consumption of CPU. Unfortunately, the program latency and CPU frequency are inversely proportional; the program execution time will increase when clock frequency decreases. Since the energy consumption of a program is equal to product of execution latency and power consumption, scaling down frequency does not bring any energy savings. In fact, because of the increased latency, it might cause the rest of the system (such as peripheral devices, display) to consume more energy.

However, scaling down clock frequency introduces another opportunity. In CMOS circuits the settling time of a gate is -roughly- inversely proportional to the voltage level. If the settling time deadlines are relaxed, then it is possible to apply a lower voltage. Consequently, scaling down clock frequency provides quadratic savings if voltage level can also be decreased in the same proportion. In practice, this technique is less efficient because the real CPU's support only a few discrete voltage and frequency levels. Many power conscious CPU's support frequency scaling at run time, for example the StrongArm SA-1100 processor[19], which is used in Compaq iPAQ's, has 11 discrete frequency levels from 59Mhz to 206 Mhz. Also, there are a few CPU's which implement dynamic voltage scaling; examples are Transmeta Crusoe[41] and Intel Xscale[20]. **Figure 1** shows the frequency/voltage and power relationship of AMD PowerNow! processor[4]. Notice that the relationship between power and frequency/voltage relation approximately resembles the equation above.

3.1 Dynamic Voltage Scaling

Weiser et. al,[42] is among the first researchers to draw attention to the importance of dynamic voltage scaling. With this research, he attempts to answer these questions: If an operating system implements a voltage scaling mechanism, what scheduling policy should it use and with what parameters?

Table 3: Event List for simulation of Weiser's policies. (taken from [42])

SCHED	context switch away from a process
IDLE_ON	enter idle loop
IDLE_OFF	exit idle loop
FORK	start a new process
EXEC	overlay a process within another process
EXIT	terminate a process
SLEEP	wait on an event
WAKE_UP	notify a sleeping process

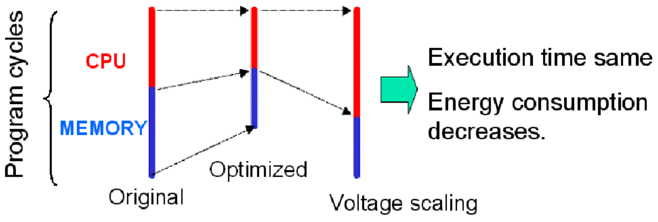

Figure 3: Trading-off execution speed optimization with battery power (taken from [18])

Weiser compares the effectiveness of different policies by simulating them on a process history trace collected from a set of UNIX workstations. The events included in the traces are shown in **Table 3**. All events are time stamped using a microsecond time scale. Weiser claims that the effect of trace collection on total load is less than 7%.

There are three different policies; OPT, FUTURE and PAST. All policies adjust the CPU clock and voltage and assumes energy savings as given in **Equation 1**. Weiser assumes a hypothetical CPU in which voltage level can be adjusted to any value between a minimum and a maximum, and that a perfect linear relationship exists between voltage level and clock frequency.

The OPT policy assumes perfect future knowledge and *unbounded delay*; it tries to stretch the run times to fill all the idle time slots and ignores any task deadlines or interactivity requirements. OPT always adjusts to the minimum voltage level that enables task completion.

The FUTURE policy is similar to OPT, except that it assumes only a small window of future knowledge. The Future is not an unbounded-delay scheduling algorithm; it never delays the task beyond the window.

The PAST policy is a realistic version of FUTURE. Instead of looking at a fixed future window, it uses a window of past behavior and assumes that the next window will be similar. PAST is also a bounded delay policy; if there are some excess cycles left in the past window, (because the CPU speed was set to a too low value), then it rolls over the cycles to the next slot.

Weiser studied window sizes between 1 millisecond and 1 second. The smaller windows provide a better response time when interactivity is a concern. Weiser argues windows of size 10 milliseconds to 50 milliseconds provide adequate response time.

Figure 2 compares the effect of interval length on total energy consumption. The graph also contains the comparison of three different voltage scales; this is the range that CPU voltage can vary.

The length of interval does not effect the savings of OPT as it assumes a full future visibility, the other two policies come very close to the level of OPT for larger intervals. Interestingly, PAST does better than FUTURE in general and this

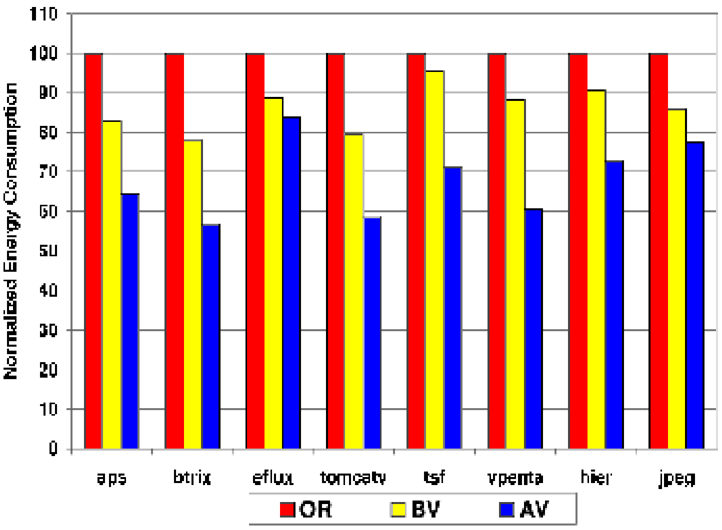

Figure 4: Results of Saputra: The bars show the normalized energy consumption of various applications when static voltage scaling is applied. The red bar is before any optimizations, yellow bar is after loop optimizations and blue is after voltage scaling (taken from [18])

difference is very apparent for intervals smaller than 100 milliseconds. This is because if PAST chooses a very slow clock speed for previous interval and cannot complete the job, then it can process the excess cycles in the next interval, effectively lengthening the interval time. However, FUTURE cannot defer the computations this way as it only considers the CPU demand for next interval.

The voltage range of CPU is also important for total energy savings. Weiser kept the maximum voltage at fixed level (i.e. 5 Volts) and simulated minimum voltage levels of 3.3, 2.2 and 1.0 volts. The CPU with smaller voltage level always achieved a better result, however, the difference between 2.2 Volts and 1.0 volt was insignificant.

3.2 Dynamic Voltage Scaling

Saputra et. al,[18] studies voltage scaling from a different perspective; instead of targeting the idle cycles at the OS level, he adds a new dimension to the compiler optimization strategy: energy conscious code generation. Saputra attempts to answer two primary questions:

- Most compiler optimizations target best performance (i.e. the shortest execution time). How do such optimizations impact energy consumption?
- Are there any compiler optimizations that target only energy consumption without changing execution time?

To address the first question, Saputra discusses four optimizations: Loop tiling, loop distribution, loop fusion and loop permutation. Loop distribution distributes the larger, complex loops into smaller and simple loops. Loop tiling also divides the loops by distributing the accesses to a large consecutive data structure into smaller many loops. Loop fusion merges individual, small loops into a larger loop. Loop permutation changes the order of loops in a deep nest. Each of these optimizations target reducing cache misses by improving data locality. We describe the energy aspect of these optimizations later in this section.

Saputra introduces a hybrid mechanism as a solution to second question. He converts the gain from performance optimizations into energy optimizations such that the execution performance of the program the is same as pre-optimized code but energy consumption is reduced. (See **Figure 3**). The first step towards this goal is to identify the effects of optimizations on application. It is possible to divide the application latency into two main parts: Execution latency due to CPU computations, and execution latency due to CPU wait cycles. Commonly, wait cycles occur from memory accesses. We refer to the former one as on-CPU time and to the latter as off-CPU time. Clock scaling only effects on-CPU latency. It does not change off-CPU latency as the memory and bus speeds are unchanged.

The goal of Saputra's hybrid mechanism is to enable optimizations that decrease both on-CPU and off-CPU execution time. After that, he reduces the clock speed such that the increase in on-CPU execution time is equal to the decrease -i.e. which is generated by optimizations- in total execution time.

To this end, Saputra investigates two different policies to set the voltage level: static and dynamic. The static policy uses a fixed voltage/clock value for the duration of the program. The dynamic policy uses multiple voltage levels for different parts of the program.

The static policy tries to maximize energy-delay product of a program. That is, it finds the best voltage/execution time pair. The energy-delay product can be described by:

$$Energy * Delay = [KCV_0^2] * [V_0/(V_0 - V_t)^2] \tag{2}$$

where V_t, V_0, K, C are threshold voltage for CMOS circuits, applied voltage, clock frequency and capacitance respectively.

To minimize **Equation 2**, Saputra computes the derivative and sets it to 0. This generates the solution $V_0 = 3V_t$. Thus, static policy simply sets voltage level to a fixed value.

The dynamic policy targets loop nests; it tries to find the best voltage level for each individual nest. During compilation, the compiler generates two tables; one for execution time and the other for energy consumption. To be more specific, if there are N voltage levels and M nests then each table has NM entries (i.e. NM voltage levels and NM execution times). This can be converted into a linear integer-programming problem, which the constraints include execution time, voltage switching overhead and energy consumption.

We show results for Saputra's static optimizations in **Figure 4**. The results indicate that their mechanism improved energy consumption 31.8% on average. The dynamic mechanism improved energy consumption 15.3% over static one.

3.3 Network Interfaces

The wireless network interface consumes a considerable amount of power; many researchers have investigated them and proposed various mechanisms to reduce energy consumption in wireless networks.[25, 12, 16]. Stemm et. al[38] investigate three aspects of the problem: Hardware power consumption behavior, energy efficiency of communication protocols, and energy-conscious design of current applications.

Stemm et. al studied a diverse group of hardware devices; one 915Mhz long range-low bit rate interface (Metricom), two Wavelan cards at 915Mhz and 2.4 GHz, one infrared interface, two PDA's and one laptop. Their findings were:

- Transmitting data is significantly more expensive than receiving
- Receiving data and idling exhibit very similar power consumption characteristics.

The communication protocols that are currently used in the wireless networks were originally developed for wired networks. Because of the fundamental differences between wired and wireless communication (such as error rate, etc), these protocols have inefficiencies. Stemm's study includes the protocols at the transport layer:

- TCP Reno: TCP Reno generates one acknowledgement for each packet
- TCP Reno with delayed acknowledgements: This is same as Reno except that it generates one acknowledgement for every other packet
- Reliable UDP with limited window: This is a variant of UDP protocol, it uses an error recovery window of size w. Each window is acknowledged once, and a single selective acknowledgement is used; if one packet is dropped, sender retransmits it. In average a little more than one acknowledgement is transmitter per window.
- Reliable UDP with unlimited window: This is similar to limited window protocol except that the window size is as large as the number of packets sent.

Stemm et. al, simulated these protocols using ns[1]. First they measured energy consumption only when the actual

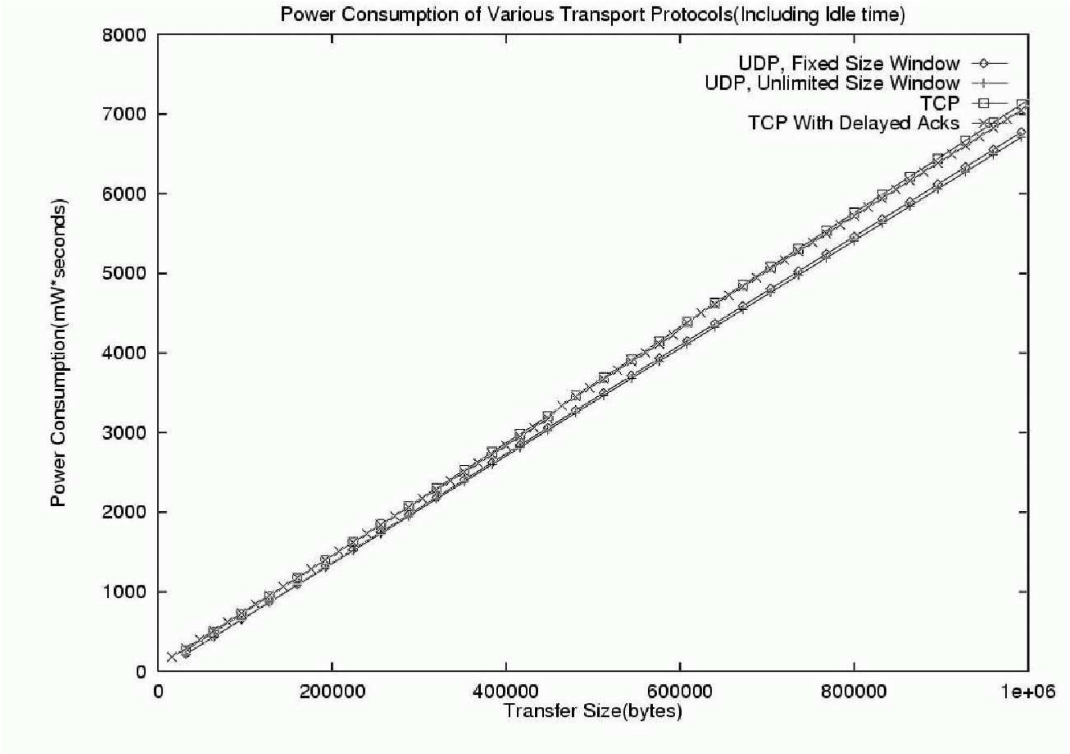

Figure 5: Energy for different protocols including idle time and transmitting power. Even though TCP uses much more transmission energy, because of the high cost of idle, the differences between protocols become invisible.(taken from [38])

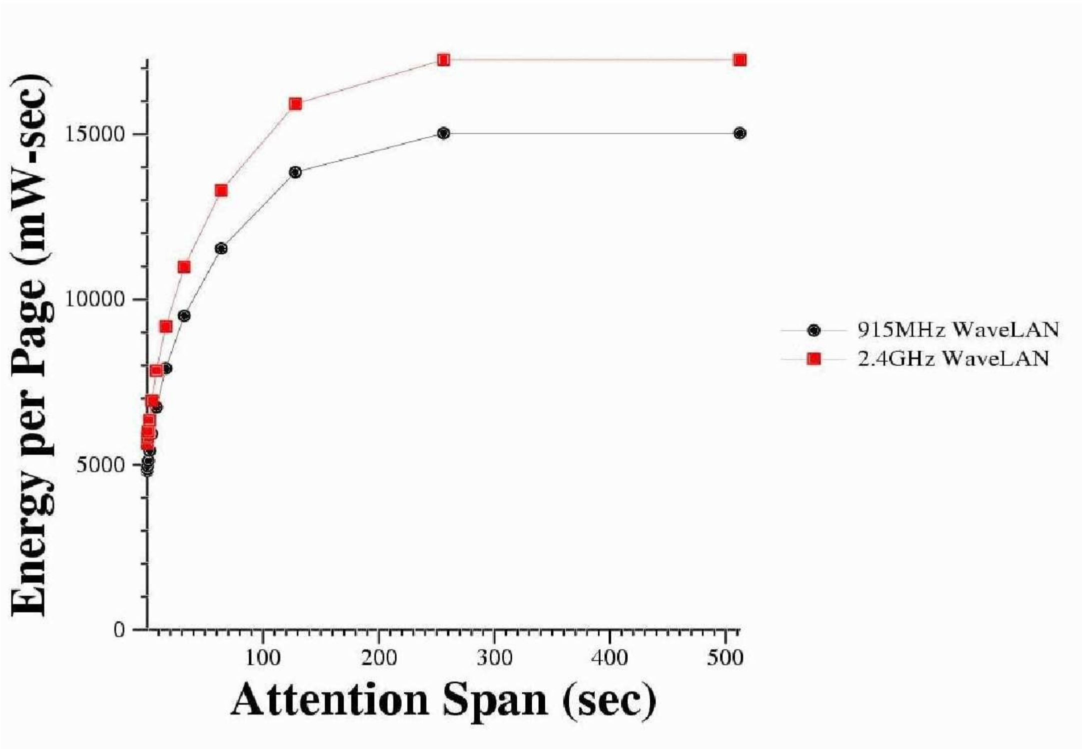

Figure 6: The effect of attention span on energy consumption. With the decreasing attention span power savings increase exponentially.(taken from [38])

transmission (send and receive) is happening. TCP pe formed worst because of the many acknowledgements. How ever, when they included idle time into the scheme, all proto cols used approximately same amount of energy. Their result are shown in **Figure 5**.

As a final step, Stemm et. al, proposed enhancements for tw applications; e-mail and web browsing. They tested bot strategies using simulation of real data that they collecte from their computer science access logs.

The email enhancements are based on a client-initiated pro tocol; the client wakes up at certain periods and checks if an email is waiting for it. The network interface is hibernate at all other times. The results are shown in **Figure 6**. The findings show that with the decreasing attention span energ savings increase exponentially.

The web enhancements are also based turning off networ interface when it is not in use. The web application turns o the network interface if no other request comes for a perio time i.e. attention span. The results are in **Figure 7**.

4. COMPUTATION OFFLOADING

Another way in which we can reduce power consumption is to use distributed computing: The mobile device can simply defer computation to a server that is connected to wall-power, i.e. computation offloading. Computation offloading and hardware utilization techniques are orthogonal. In fact, as we present in the next sections, systems that use computation offloading almost always put the mobile node into a low-power state to minimize the energy consumption.

An important aspect of computation offloading is estimating the benefit of migration. There are many volatile variables in the equation: the network bandwidth, load on local and remote CPU, available battery capacity, and the possibility of disconnection are just a few. However, it is possible to divide all these into two main groups: Cost of remote execution and cost of local execution. The remote execution cost includes network transfer and idle power consumption. The local cost execution is simply the local energy consumed. A problem that further complicates the cost estimation is difficulty of measuring power consumption at run time. Recently, smart battery monitors such as DS2760[8] became available but their precision are still very low.

Computation offloading can be done at different granularities; it is possible to remotely execute part or all the entire program, or only delegate the compilation. In the next sections, we discuss these approaches.

4.1 Partial program offloading

Kremer et. al, [26] propose an automated scheme. The compiler identifies possible computation offloading points and based on this, it generates the client and the server code. The computation offloading points are called *tasks*; each task is defined by its entry and exit point.

Kremer et. al, limits the tasks to procedure calls in the main function. Furthermore, he assumes no side effects during the execution of procedure; the only way to communicate with a

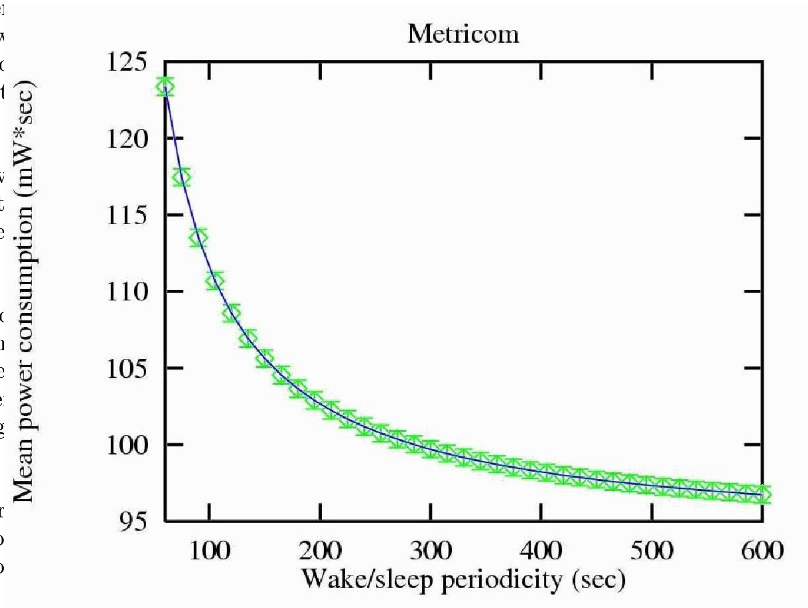

Figure 7: Energy vs. time period between each email check. The energy consumption decreases exponentially with increasing span.(taken from [38])

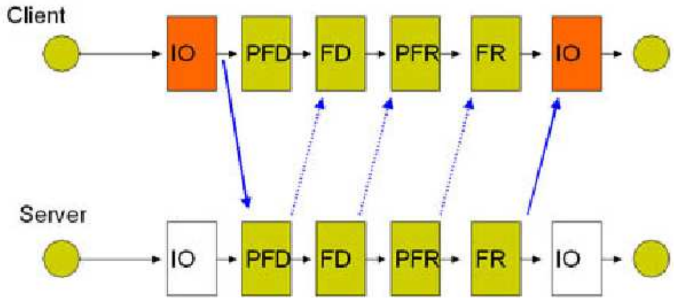

Figure 8: Illustration of checkpoints: The server task has three possible checkpoints; after PFD, FD, and PFR.(taken from [26])

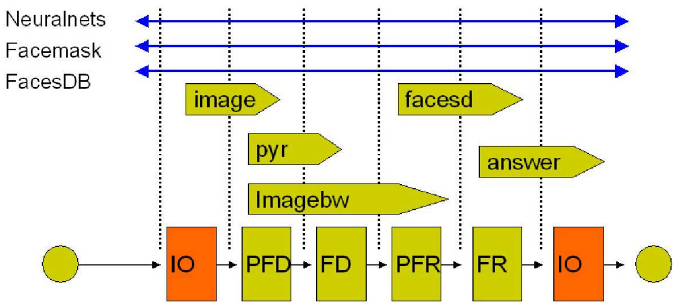

Figure 9: Shared and Replicated objects: Neuronets, Facemask and FacesDB are replicated and exist on both the client and the server. The objects inside yellow arrow are shared; any modifications may have to be reflected to other side.(taken from [26])

```
int x,y,z;
main() {
  y=1;
  scanf("%d",&x);
  if (x<10)
    S1: func1();
  if (x>0)
    S2: func2();
}

func1()  {
   S3: z=func3(y);
}

func2()  {
   int i,a;
   a=y;
   for(i=0;i<100;i++)
      S4: a=func3(a);
   z=a;
}

func3(int  b)  {
   int c;
   c=b+1;
   return c;
}
```

Figure 10: Example program: S1, S2, S3 and S4 are task entry points.(taken from [47])

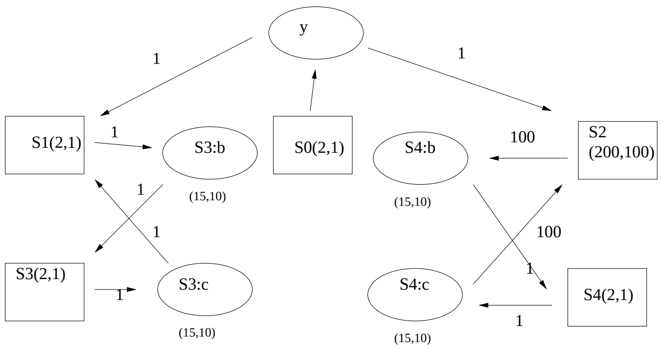

Figure 11: Cost graph: This is the partial cost graph for example program in 10. The rectangles represent the tasks and the ovals represent the shared data. The numbers next to arrows are predicted count of data accesses (see text). The number pair inside the tasks is local and remote execution cost estimation. The number pair next to data is cost of transferring data item using push and pull method.(taken from [47])

procedure is through its argument list and results. A *candidate thread* starts with the first client I/O and ends with the final client I/O. Once the candidate thread starts; it performs the rest of the computation on server until the beginning of final client I/O -i.e. including all the remaining tasks.

The compiler inserts calls to policy functions at the entry point of each task. The policy functions predict the cost of offloading at run time. The cost computation is based on a parameterized, static model. The policy function checks the availability of a communication link. Then based on the input parameter size and expected execution time, the policy function makes the remote execution decision.

The client and the server use a checkpointing mechanism to handle disconnection events (see **Figure 8**). In this example, the server has three possible checkpoints; after *PFD, FD, PFR*. The checkpointing mechanism provides the status of computation to the client and it also lets the client to do less computation in case of a disconnection.

The compiler identifies two types of data: replicated and client. The replicated data exists on both the client and the server; it is never transferred between them. The client data might be transferred during task entry and exit points. (See **Figure 9**). The thin arrows (Neuralnets, Facemask, and FacesDB) represent the three replicated data in their running example; these objects are never communicated between the client and the server. The thick arrows represent the client data; they are transferred based on the execution location of the task. For example, if **FD** is offloaded to server, then the client transfers pyr and imagebw after *PFD*.

The client application hibernates the mobile device during the remote execution and causes it to wake up periodically to check the status of the computation. The client also may request checkpoint state from server before hibernating again.

4.2 Li's Approach to Partial Program Offloading

Li et. al[47] propose another technique for computation offloading. Their approach is similar to Kremer's approach, it is based on automated partitioning of a program into client and the tasks. The partitioning granularity is again at the level of functions, i.e. each function call site is a task entry point.

However, Li does not limit server tasks to self-contained procedures, i.e., procedures without any side effects. He introduces a message passing system to provide coherency between the client and the server. The message passing mechanism can be implemented in two different ways; push or pull. In the former, the task that modifies the variable is responsible for updating the memories of other tasks. In the latter, the task that reads a shared variable is responsible of synchronizing its memory by reading the value of variable from other tasks memory. Li implements both methods.

Both methods have advantages; the push method always transfers data after modification, the pull method may require multiple transfers or no transfer at all depending on program flow. For each data element, compiler determines cost of push and pull, and chooses the best way during partitioning. **Figure 10** shows an example where the cost of push and pull is different. The main function updates the value of y. The push method requires propagating the value of y right after modification, so one transmit is required. The pull method requires synchronizing the value of y before using it, i.e. inside *func1()* and *func2()*. Thus if pull is used, either zero, one or two transmissions are required based on the value of x in *main()*.

Li's design does not allow concurrent execution of the client and the server tasks; furthermore the data sharing is allowed only if message passing and data dependencies guarantee its coherency. The compiler generates a dependency/cost graph.

The nodes of the graph represent the tasks and data. There can be edges only from a task to a data or vice versa: an edge to a message means that the message is modified by the task the edge starts; an edge to a node means node reads that (possibly) modified data. The compiler predicts the number of times the data use will happen and associates each edge with it.

The next step is to determine the costs of local and remote execution. For each task compiler estimates cost of local and remote execution, and updates the cost graph. **Figure 11** shows a part of cost graph generated for program given in **Figure 10**. The rectangles and ovals in the graph represents tasks and data, respectively. The number pair next to tasks shows the predicted cost of local and remote execution of the task. The number pair next to data represent transferring data using push or pull method. The arrows are weighted by the estimated number of times data will be accessed or updated.

Partitioning the graph is final step: the optimization constraints are minimizing the total cost and assigning each task to either a server or a client. Unfortunately, a branch and bound algorithm is not appropriate because it grows exponentially with the size of graph. Li et. al, introduce an algorithm that was uses heuristics; it applys branch and bound to a sub-graph that contains only to the nodes with highes costs.

One limitation of Li's technique is that the cost computation is based only on static parameters. Kremer's approach uses policy functions to determine cost at runtime. The size of the cost graph, heuristics based partitioning algorithm and lack of checkpointing are other drawbacks of his approach.

5. PARTIAL PROGRAM OFFLOADING - DYNAMIC COMPILATION

The previous computation offloading techniques have a common mechanism: The compiler generates two versions of code; a client and a server, and only data is transferred between nodes. Furthermore, the cost computation is based on static, parameterized models, and partitioning is done in a rather ad-hoc manner.

Static compilation also raises questions about scalability:

- Static compilation requires source code: The application might be proprietary, or it might depend on prorieatry libraries.
- Static compilation requires a separate client and server for each platform: Deploying clients and servers on different architectures and later maintaining them can be a challenge.

Our view is that a hardware-independent language, such as Java, can be solution. In fact, one of the primary design goals of Java is to be hardware-neutral and portable[17]. However, because of performance problems on limited-resource devices, Java has not received a wide acceptance yet[36].

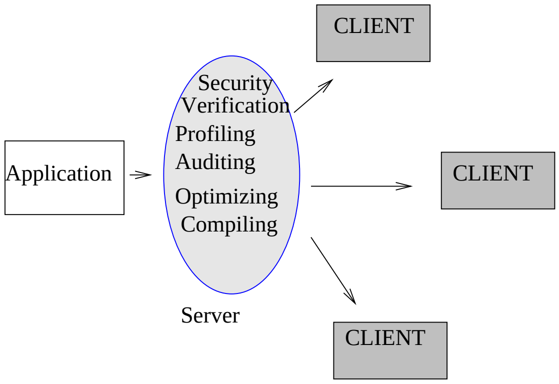

Figure 12: Distributed Virtual Machine: Server is responsible from security verification, optimization and JIT compilation (from [36])

In this section, we present an additional view of computation offloading: A hybrid mechanism that uses distributed computing as a glue to combine the advantages of Java and native compilation.

5.1 Distributed Java Virtual Machine

Sirer et. al,[36] propose a distributed Java Virtual Machine to improve performance of Java programs. Their study does not target energy efficiency, but we believe it is an important step toward achieving this goal.

Sirer argues that there are many JVM tasks that can be offloaded from client to a centralized server (see **Figure 12**). He divides these tasks to two main groups: Static and dynamic service components. Static service components examine the instruction segment of the Java programs. These include services like verification, security, compilation and optimization. For example a verifier may check the code for type-safety, or a compiler might compile the code for a client and *ship the executable*. The dynamic service components provide functionality during execution; such as library manager, security server and network management server.

The static and dynamic service components are coupled with binary rewriting. The server inserts calls to dynamic service components if it encounters operations that should be accomplished at run time. For example, the verifier might insert additional checks to make sure the conformance of code to Java VM standards[28]. This architecture allows the static service components to be implemented in a separate, unloaded machine with an abundance of available services.

Another interesting part of the study is centralization of the compilation service. The JIT compilers translate bytecode into native code, however this operation requires considerable resources. Moving the compilation to a server makes it possible to implement JIT compilation for resource-restricted systems. However, the on-demand nature of compilation introduces delay. Caching and ahead of time compilation might reduce the impact of such overhead.

One of the most important issues for this system is scalability due to centralization. Sirer evaluated the system by using

Figure 13: Distributed JIT compilation: The Java code is compiled ahead of time for minimum latency, dynamic class loading requests are resolved by proxy server.

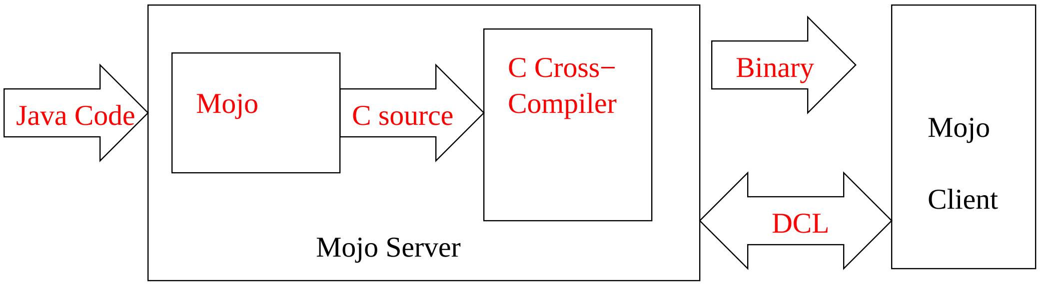

Figure 14: Distributed Virtual Machine: Server is responsible from security verification, optimization and JIT compilation

same hardware for the client and server and without using caching. He found that the system is able to handle up to 150 clients.

5.2 Proxy Compilation

Newsome improves upon the idea of distributed Java compilation[30]: he introduces a client-server mechanism to compile Java source into native code. His mechanism is illustrated in **Figure 13**. Java source can either be compiled with an ahead-of-time compiler, or a proxy server can compile it on demand. In either case, if the client needs dynamic class loading, the proxy server resolves the request.

At the center of Newsome's design is the Mojo compiler (See **Figure 14**). Mojo compiles Java classfiles into ISO C source code, and then a C cross-compiler generates native code for the client. Mojo currently omits an important portion of Java 1.1 specification including verification, interfaces, multi-dimensional arrays, exceptions, and multi-threaded execution.

Mojo reads Java classfiles and computes the transitive closure of them; this is the complete set of classes referenced by the initial class excluding the ones referenced via reflection (i.e. java.lang.Class.forname). Then Mojo loads and parses each file that are in the set of transitive closure. The next step is code generation: Mojo computes the method and field tables for each class, and generates the code for class structures and methods.

The generated C code includes a global table which stores class meta information. This is used to map the symbolic names, linked addresses of fields and methods, the class name, the superclass name, the object size, and the vtable. During execution, a call to Class.forname causes the system class loader to look for a matching meta information entry in the table. If the information cannot be found, the class loader uses the proxy compilation client library to contact the proxy server. The class or the location of the class is passed to server, which loads the class, compiles, and returns the class as a shared library. The client links this library using **dlopen()** or a similar call depending on the OS.

One optimization of this scheme is to use a manifest file listings. The client can instruct the server to generate the list of all compiled classes. This enables the server to remember what is on the client. Server stores this file in a database and during dynamic linking excludes them from shared library. This decreases the library and communication size.

The results show almost a 94% speed up over interpretation-based systems, and a 20% speed up over JIT based systems. However, Newsome did not investigate Mojo in terms of energy usage.

5.3 Evaluation of Distributed Compilation

Palm et. al[32] performed a study that is complementary to those above; he investigates the energy aspect of distributed Java compilation. He considers four primary cases:

- download and compilation (without optimization) of bytecode by the client
- download and compilation (with optimization) of bytecode by the client
- compilation (without optimization) by server, download by client
- compilation (with optimization) by server, download by client

Palm et. al,uses a static, parameterized method to evaluate these cases. They assume that the client is 7 times slower than the server machine. They substitute execution time as an indicator of energy consumption. The three other major parameters are: network transmission cost, local execution idle cost and local execution peak cost, i.e. energy consumption during compilation and execution. They compute these numbers based on technical manuals and offline experiments.

Their test environment uses Jikes RVM[3] on a 1.4 GHz Pentium 4 desktop. For each case, they collect method compilation time, method run time, the size of the method bytecode, and the size of the compiled bytecode. The benchmarks are from SPECjvm98 Suite, with the large input set.

The first part of their results show the effect of optimization at the application level, i.e., optimization is applied to all methods at once.

- If it is possible to compile and optimize the application before it is demanded, i.e. ahead of time compilation, then it is beneficial to all test cases. Alternatively, if compilation is done on demand, it might save energy in some cases.

- Using optimization at the client always increases energy consumption.

Palm et. al. also discuss optimization at a method granularity. They compare the cost of each policy on a per method level; accounting for the compilation, transmission and run time cost. Their results do not specify if the run time cost of method is dependent on invocation counts of methods. They argue that even though there is no single best method, compiling at the server ahead of time is generally better than not doing so.

6. REMOTE EXECUTION

In this section, we present techniques that do not partition the application. These techniques migrate the tasks as a whole. The techniques described in the previous sections are generally implemented at the application level; either the compiler does the partitioning or a run time system performs migration. The offloading techniques do not modify the application; instead a separate process monitors various resources and decides when to migrate the task.

6.1 Remote Processing Framework -Rudenko

One remote processing framework (RPF), designed by Rudenko[34] is based on a client-server architecture. The client side of the RPF is responsible for deciding between local and remote execution, implementing synchronization of file systems, and handling application database maintenance. The server side performs computations requested by the clients.

The RPF transfers data between the client and server; it does not allow migration of the executable. Once the RPF decides on remote execution, it checks with the server to verify whether it maintains a compatible software package and has enough computational resources. When the server accepts the request, the client sends the data files. The client can either hibernate or continue waiting idle. After completion, the server contacts the client and returns the result. In case the server cannot establish communication with the client, it keeps results until the client requests them.

The client keeps track of applications that use RPF. It records working directory of the application, the command line options, and various statistics into a database. The statistics contain measurements of power consumption during local and remote execution. The database is also used to optimize file synchronization; the client keeps track of files that were transferred to the server, and does not transfer them again if they are not modified since the last remote execution.

There are four principles that shape the local/remote execution decision process:

- Data Gathering: Initially, the RPF does not know about the characteristics of application, so it runs the application both locally and remotely.

- Primary Decision Making: The RPF determines the execution location after data gathering phase is completed.

- Validity Checking: RPF deliberately gives the wrong decision to update its knowledge database, since running always at the same location could prevent it from discovering excess capacity at the other side.

- Filtering: Because the decision process is based on execution history, filtering is needed to remove extreme cases. The RPF looks at the recent N sample to estimate mean power consumption history:

  $e_p = (1 - \alpha) * e_m + \alpha * e_l$ where:

  e_p:predicted energy cost
  e_m: mean of last n energy measurements
  e_l: last energy measurement
  α: coefficient of conservatism

The RPF uses standard interfaces and off the shelf packages. It monitors power consumption using the APM interface in Linux, and uses ftp, gzip and tar to transfer files between client and server.

Rudenko evaluated the RPF using a 486 laptop as mobile device and a Pentium based computer as a server. Their results show that applications with higher CPU and I/O requirements are more appropriate for remote execution. In fact, they argue that on average the energy cost of remote execution is linearly proportional to the size of data that is transferred. As such any application which has a complexity higher than $O(n^k)$ with $k > 1$ will benefit from remote execution for sufficiently large n.

6.2 Self-Tuned Remote Execution

Another remote execution system is Spectra[14], which was developed by Flinn et. al, at Carnegie Mellon University. Spectra is not a stand-alone system; it is the remote execution component of Auro (see **Figure 15**). Auro is a pervasive computing platform with the goal of providing persistent computational resources to its users[24, 31].

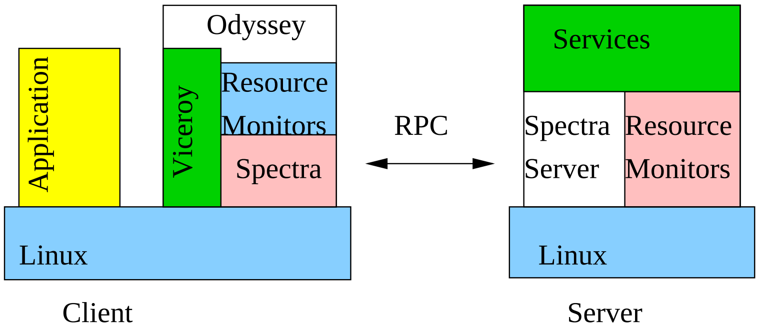

Figure 15: Spectra and its relation with the other components of Auro: Spectra provides mechanism for invoking remote services and it also continuously monitors resources of client and server. The application-Spectra interface is through Odyssey (taken from [13]).

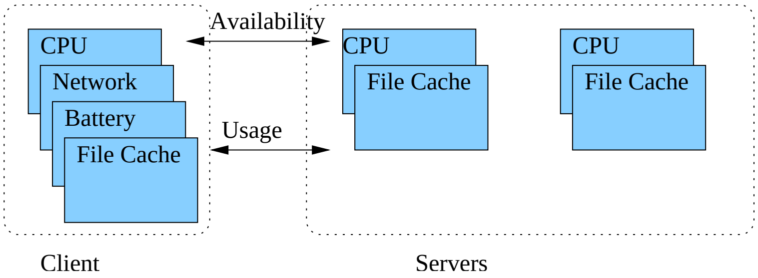

Figure 16: Resource Monitors: Spectra monitors, CPU load, network bandwidth, file cache status and battery power (taken from [14]).

Spectra decides where and how to execute the applications. To make a decision, Spectra collects information from a number of resource monitors both at the client and server side. In particular, (see **Figure 16**), Spectra monitors CPU load of the client and server, network bandwidth, battery power and file cache status.

Both the CPU and the network load predictors are not proprietary to Spectra. The CPU load monitor uses a technique explained in [10]; it measures the availability by looking at a smoothed recent CPU load history and weights the load by the CPU speed. The network monitor[31] predicts bandwidth and round trip time, it monitors bytes sent and received and the number of remote procedure calls.

The battery monitor uses the smart monitor technology[9] to measure the current flow. However, Flinn et. al, argue that battery monitor has two drawbacks: the high cost of accessing smart monitor chip using a serial interface and the limited precision.

Spectra can monitor the file cache status if underlying file system supports caching. The current implementation of Spectra supports monitoring Coda[23] file system cache. During both local and remote execution, Spectra interacts with Coda to keep track of files that are accessed by application. Spectra uses this information while predicting cost of remote execution. It assumes that the likelihood of a file being accessed is simply similar to the last time the application was run with same parameters. It adjusts the cost of network transfer by excluding the transfer cost of files that are already in cache.

Some of the applications can run at different performance and quality levels, i.e., fidelity levels. For example, a speech recognition application can provide two different vocabularies: A brief vocabulary that provides fast translation with less precision, and a detailed vocabulary that provides slow translation with high precision. Spectra has special support for those applications that have multiple fidelity levels. Spectra evaluates energy costs of different fidelity levels before giving a local or remote execution decision.

Spectra's decisions are based on three metrics. These are execution latency, battery lifetime and application fidelity. Spectra first computes a *context-free* value for each metric, then weights it by an *importance function*, and finally it calculates the product of the weighted metrics to evaluate the alternative. The context-free value is application specific; for example execution latency is the sum of network transfer time and processing latency on local and remote machines. The importance function is based on user's demand. For example, the importance function for battery lifetime is $(1/E)^c$ where E is the expected energy consumption of application, and c is the user specified importance criteria.

Flinn et. al, evaluated their mechanism on a handheld computer that is connected to a laptop through a serial link. They tested the effectiveness of Spectra using different scenarios such as maximum battery lifetime. Spectra was able to choose the best option in each scenarios.

7. OUR WORK

In the previous sections, we have presented various computation offloading techniques. However, these techniques have not experienced widespread use due to the complex and challenging nature of process migration and program resource estimation. Furthermore, the rapid advancements in hardware technology creates new opportunies for further improvements in previous research. During our survey, we realized that most computation offloading platforms are targeting laptop computers ignoring the resource constraints of handheld devices. Moreover, improvements in battery technology enable improved precision in the power monitoring mechanism. As a result, we developed a new computation offloading platform called the Remote Processing Platform (RPP).

Even though RPP can be used on laptop computers, our target platform is handheld devices as their use introduces significant challenges. First, they are much less capable than laptops in terms of resources, i.e., CPU speed, memory size, battery capacity. Second, the cost of network transfer is more expensive relative to laptop computers. This is because both handheld devices and laptop computers use same hardware as wireless network interface, however, the overall power consumption of a laptop is much greater than a handheld computer. In fact, the previous studies showed that in laptops the most important power consumers are hard drive and display. The network interface has a much less significant effect on overall consumption. The network interface can consume as much 50% of all energy consumption in a handheld computer [16, 25, 15, 27].

We developed two models one for local and one for remote execution to predict program cost. Our models use a hybrid technique: they combine profiling and high level program characterization with a cost model.

8. REMOTE PROCESSING PLATFORM

Our Remote Processing Platform, (RPP), is based on a three level client-server architecture; the client, the middleware, and the server. Both the client and the middleware are on the mobile device side; the former analyzes the submitted job and forwards it to middleware if remote execution is favorable. The latter handles the communication with the server and power management of the hardware, and monitors various parameters. The server is a wall-powered computer; it performs the computation and returns the results to middleware.

A main concern in the design of RPP is the heterogeneity in mobile computing world. Most handheld devices use low-power MIPS or StrongArm based CPU. In contrary, laptops and desktops are mainly based on Intel X86 and PowerPC technology. Even though, migrating the executable between different architectures is possible, it introduces many challenges [40, 46]. As a result, we designed RPP only to migrate data. The server is required to maintain a cross-compiled version of client's software package for remote execution.

Another important design factor is the target application type. Interactive programs have little or no gain from remote execution because of extra latency introduced by many small messages [35, 34]. Instead, RPP targets computationally intense programs. In fact, the multi-tier architecture of RPP provides an efficient mechanism to migrate multiple tasks concurrently from mobile device to the server. With RPP, we do not address are security, reliability/failure of server, and hand-offs from one server to another.

There are four main components of RPP; these are Pshell, Remmgr, Remsrv, and Program Profile Database. RPP and its components are shown in **Figure 17**. We describe the components in the following section.

8.1 pShell

The *pshell* receives the job from user and analyzes it. It estimates the local and remote execution energy cost by interacting with the *cost database*. If remote execution will be profitable, then *pShell* assigns the job to remote execution manager and waits until the job completed signal is delivered by remMgr. Otherwise it executes the job locally.

There can be multiple *pshells* running concurrently in each mobile device. Nevertheless, all of them use one common message queue to communicate with the *remmgr*.

8.2 remMgr

The *Remmgr* is the middleware layer. There is only one *Remmgr* per mobile device. It acts as a manager which is responsible of transferring the job to the server and receiving the results. The communication is done using the TCP protocol; *Remmgr* sends a control message to the *Remote server* which includes program name, command line, context (for identifying client), and input files. The *Remote server* uses a similar protocol to deliver the results back.

The *Remmgr* can be easily modified to use different policies for checking the job status at the server. The current policy targets computationally intensive applications and aims at minimizing the network interface idle time by setting it to sleep mode as much as possible. During execution, the *Remmgr* sends a "is job ready" request to Remsrv in 1 second intervals. Currently, this interval is not adaptive.

The *Remmgr* is not limited to only job migration, it also manages the power states of hardware components. These components currently include the network interface, sleeve unit, and CPU frequency. Being a research platform, *remmgr* provides a clean mechanism to easily implement different algorithms that can manage the hardware components in different ways. Our future improvements will include screen management (which currently is kept in the off state).

8.3 remSrv

The *Remote server* is a user-space program. It listens for incoming connections and when it receives a job, it forks a child process to execute it. After completion, the jobs wait in the *Outgoing* queue until they are returned. The *Remsrv* identifies each *Remmgr* with the IP address and TCP port to which it is bound.

8.4 Profile Database

The database stores program logs which include both general and program-specific attributes. The general attributes are

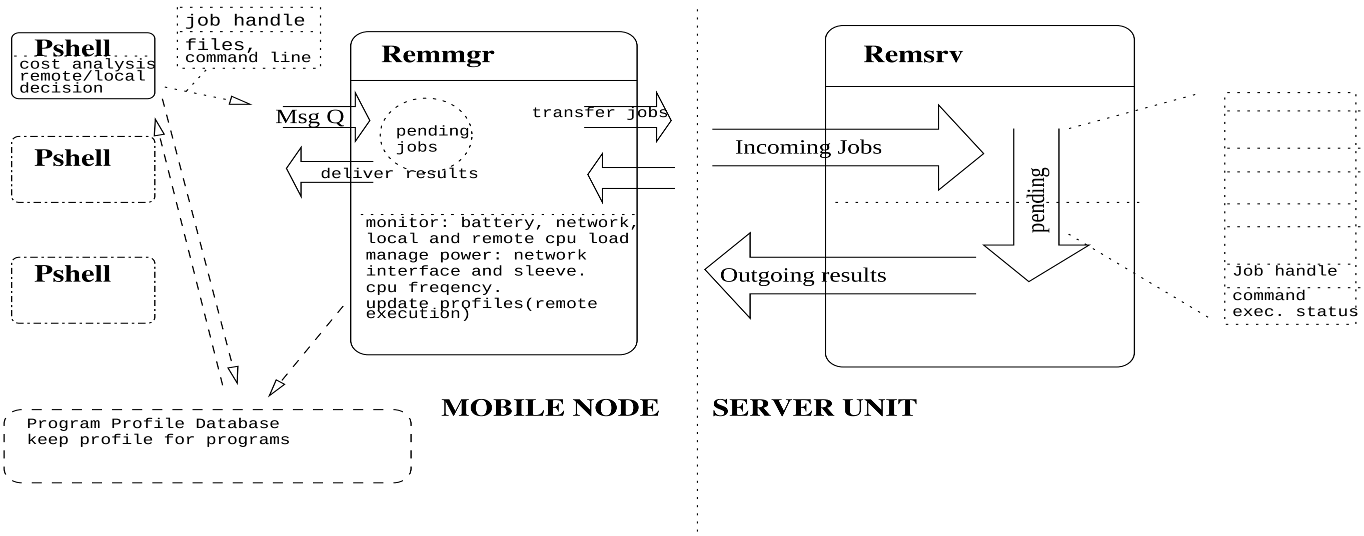

Figure 17: RPP Components

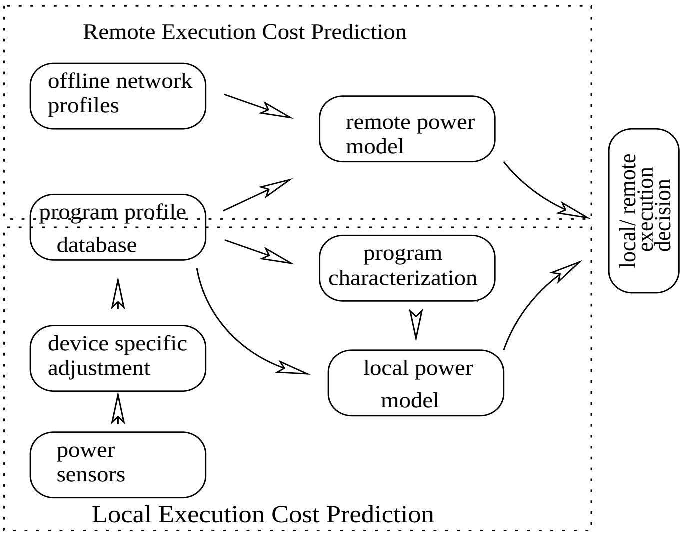

Figure 18: Decision Process

execution time, energy spent, network transfer size, execution location and error in energy prediction estimation. The application specific attributes are program command line parameters and input characteristics, e.g., file size, number of items to be sorted, etc.

9. ENERGY CONSUMPTION PREDICTION

Figure 18 summarizes the decision process for migration. The two main blocks, which are shown with dotted borders, are responsible for predicting the remote and local execution cost. The blocks share the *Profile Database*. The cost prediction can be divided into two main groups: Remote execution cost prediction and local execution cost prediction.

9.1 Remote Execution Cost Prediction

We currently use a static, parameterized model to estimate remote execution cost. We model it with the formula below:

$$E_{rp} = E_{idle} + E_{rx} + E_{tx} + E_o \tag{3}$$

where:
E_{tx}: *cost of transmitting data to server.*
E_{rx}: *cost of retrieving results from server.*
E_{idle}: *cost of waiting idle for server to complete*
E_o: *other costs(i.e. protocol)*

The studies of Stemm and Katz[38] show that for a fixed data packet size and acknowledgement size, the energy used for TCP transfer is directly proportional to the transfer size. This leads us to **Equation 4**.

$$E = p * n \tag{4}$$

where p indicates energy consumed per transferred byte and n is the transfer size.

Nonetheless, the cost function given above is not directly applicable to RPP. There are other costs imposed by the handshake and acknowledgements. We study the remote execution protocol of RPP by separating it to two parts: control messages and data messages. The control messages are intended for handshake between client and server, and checking the status of computation. The data messages are used to transfer files between client and servers.

The cost of data transfer includes an initial handshake message, transferring the data and receiving an acknowledgement. We model this cost by using a function $p(x)$ which estimates the cost of transferring one data byte, x gives the application layer packet size.

$$E_{tx} = p_{tx}(x) * n_{tx} \tag{5}$$

$$E_{rx} = p_{rx}(x) * n_{rx} \quad (6)$$

In the above equations; E_{tx} and E_{rx} are energy cost of transferring and receiving data of size n.

The current implementation of the remote execution protocol forces the client to check the status of the computation in 1 second intervals. Thus the number of control messages generated during the remote execution is proportional to remote execution latency. We model this cost by:

$$E_o = p_o * t$$

where t is expected remote execution time, p_o is mean power required during remote execution and E_o is energy consumption by client during remote execution.

9.2 Local Execution Cost Prediction

The bottom half of the **Figure 18** describes the details of local cost estimation process. The process is divided into two main tasks: collecting program profile information and characterization of program energy consumption behavior. The resulting power cost is estimated using these two tasks.

9.2.1 Profiling

Some of the advanced handheld devices, such as Compaq H3800[6], contain power sensors that can monitor current flow with a high precision. We use these sensors to profile programs offline *(and whenever possible, online)* for predicting local execution cost.

Our mechanism consists of three steps. The first step is collecting data from available power sensors. The second step is applying a model to this data and estimating the power consumption of the unit. Finally, the results are stored in a profile database.

We use the H3800 series handheld devices to evaluate this mechanism because it contains a high precision ASIC device (Dallas Semiconductors DS2760[8]) to monitor current flow through the battery. The DS2760 can measure accumulated current with a precision of 0.25mAh.

The H3800 can be extended using a PCMCIA extension unit, i.e., sleeve. The sleeve has its own battery and it is designed such that it never consumes power from main unit. If the main unit battery voltage drops below sleeve's battery voltage, then sleeve starts charging main unit's battery. This is called *trickle charging*[7]. Measuring power consumption of iPAQ is complicated further because the power sensor circuit is not yet available on the sleeve unit. In case the sleeve battery runs out of power completely, the network card stops functioning, but the main unit is not affected.

The non-existence of the ASIC circuit in the sleeve unit introduces two problems in the design and evaluation of our mechanism. First, its omission makes it impossible to detect variations in power consumption pattern of sleeve unit. Second, when the trickle charge starts, the readings from main unit power sensors will be an underestimation of real values because both main battery and sleeve battery will support the device in parallel.

The first problem is relatively easy to solve. During the local execution the power consumption of sleeve unit is constant, because the attached radio card stays in suspended state. So we compute the mean energy consumption of sleeve and use it to adjust the measured cost:

$$E_l = E_m + P_s * t \quad (7)$$

where:
E_l : *cost of local execution*
E_m : *energy used by main unit*
P_s : *power consumption of slave/second*
t : *estimated program run time*

The second problem, estimation of trickle charge effect, is more challenging and no good solution exists yet. At present, we profile the programs in two different configurations: with (main battery is half full and sleeve battery is full) and without the trickle charge (no sleeve unit).

9.2.2 Program Characterization

Algorithmic complexity is one way of characterizing the efficiency of programs. In addition it provides a way to compare the relative performance of alternative algorithms[39]. In this section, we propose a mechanism that uses algorithm asymptotic growth rate to guide energy consumption estimation of a program.[1]

The growth rate is a high level characterization of an algorithm and is not directly applicable to a real program to estimate its energy cost. First, power consumption is not directly proportional to the number of instructions executed; many factors such as caching, I/O, and memory access patterns complicate the behavior. Second, the one time costs such as initialization and allocating OS resources can be a major contributor to overall program cost with small input[2] sizes but will be negligible with the larger input sizes. Lastly, algorithmic complexity describes program behavior only for a typical (average or worst-case) data set.

In spite of these difficulties, we have found that algorithmic complexity combined with profiling is a valuable tool for predicting energy cost of some programs. This is especially true for programs whose run time show a close resemblance with the characteristics of their input. A few good candidates are:

- sorting programs
- image processing applications
- compression
- string manipulation programs

Assume there exists a *perfect* program whose run-time cost is *solely* dependent on the scale of the input such that algorithmic growth rate is $f(n) = O(n^2)$, and cost of processing

[1] The $O-notation$ provides an upper bound to the asymptotic behavior with respect to the input size, but we use it to mean average growth rate.
[2] Our use of the term small input means an input that causes short execution time, similarly for large input we mean an input that causes long execution time

Table 4: Benchmarks

Benchmark	Description	Command Line
bsort	bubble sort	bsort [INFILE] [OUTFILE]
qsort	quick sort	qsort [INFILE] [OUTFILE]
fft	fast fourier transform	fft [INFILE]
lame	mpeg audio encoder	lame [INFILE.wav] [OUTFILE.mp3]
djpeg	jpeg image decompression	djpeg -dct int -gif -outfile [OUTFILE] [INFILE]
cjpeg	jpeg image compression	cjpeg -dct int -progressive -opt -outfile [OUTFILE] [INFILE]

Algorithm CREATE_REGIONS$(f, cs_{1..n}, \epsilon)$

assumptions and definitions:

- Let cs_i is a pair of values such that $cs_i = (\theta_i, c_i)$ and assume $cs_{1..n}$ is ordered in increasing order with respect to θ.
- Let $\Delta_{i,j}$ is Manhattan distance between cs_i and cs_j if $0 < i, j \leq n$ and ∞ otherwise.
- Let $\beta_{i,j}$ is estimated cost of cs_j using cs_i and f. $\beta_{i,j} = f(\theta_j/\theta_i) * c_i$
- Let $\gamma_{i,j}$ is error rate between estimated and measured cost of j. $\gamma_{i,j} = [abs(c_j - \beta_{i,j}) * 100]/c_j$
- Let S be a set

algorithm:

1. S={}
2. For $i = 1$ to n
   (a) $r = 0; e = 0; j = i + 1; k = i - 1;$
   (b) While $j \leq n$ and $\gamma_{i,j} < \epsilon$
       i. j=j+1
   (c) While $k > 0$ and $\gamma_{i,k} < \epsilon$
       i. k=k-1
   (d) $t = max(i - k, j - i)$
       $r = min(\Delta_{i,i-t+1}, \Delta_{i,i+t-1})$
   (e) $e = max(\gamma_{i,k+1}, \gamma_{i,k+2}, ..., \gamma_{i,j-1})$
   (f) insert quadruple (θ_i, c_i, r, e) into S if $r > 0$.

Figure 19: create regions in a one dimensional input space.

a typical input of size n is c. If the input size doubles, the new cost c' would be:

$$c' = f(2n)/f(n) * c = 4n^2/n^2 * c = 4c$$

As a generalization of the above, we can write:

$$c' = f(n')/f(n) * c \tag{8}$$

The first step of our method is collecting *cost samples* that describe program run-time behavior. In the simplest case, in which only one dimension of input is dominant factor on growth rate, each cost sample is a set of 2-tuple:

$$cs := \{(\theta_1, c_1), (\theta_2, c_2), ...(\theta_i, c_i), ...\} \text{ where:}$$

θ_i : *magnitude of input i.*
c_i : *energy cost of processing input i.*

The next step is dividing the input space into regions such that each region R is in the proximity r of a cost sample and is defined by the attributes $\{\theta, c, r, e\}$. In particular, a point x belongs to R if:

$$Manhattan\_Distance(\theta, x) \leq r$$

The regions do not have to cover the entire space, nor do they have to be mutually exclusive. The algorithm *Create_Regions*, given in **Figure 19**, generate regions for a one dimensional input space. It has three parameters; a function describing *program growth rate*, *cost samples*, and *maximum desired error rate*. *Create_Regions* fits a region R_i to each cost sample cs_i using the other samples cs' as arbitrary measurements. Specifically, *Create_Regions* adds any $cs' = \{\theta', c'\}$ to R_i if:

1. The error rate between c' and its predicted value, which is calculated using **Equation 8**,cs_i and θ', is less than *maximum desired error rate*

   AND

2. Any other cost sample that is closer to cs_i is already included in R_i.

The output of the program is a set of quadruples $\{\theta_i, c_i, e, r\}$ describing regions. The first two variables indicate the value of cs_i which is used to define region. e reports the maximum error rate and r is the distance between cs_i and most distant cs'.

The algorithm predicts the energy cost of a new input set using these steps:

1. Analyze the input and extract attributes (i.e. θ).
2. if θ is in at least one region choose the one that has the smallest e. and compute the cost using **Equation 8**.

3. Otherwise use the two closest regions to compute cost as above and average the results.

If it is possible to extract more than one quantifiable property of input set, then the algorithm can easily be extended to work in a multi-dimensional space. In this case, the axis of the space will be the normalized quantification of the input set and each region will become a hypercube. The only other required changes are substituting θ and cost function $f : \theta \rightarrow \{cost\}$ with θ^k and $f : \{\theta^1, \theta^2, ...\theta^n\} \rightarrow \{cost\}$ respectively.

10. VALIDATION

To validate the design of RPP, we have used a set of 6 example programs that are representative of multimedia programs and general purpose algorithms (see **Table 4**). Cjpeg and Djpeg process jpeg images and convert them into different image formats, both of them are widely used both in the research community and in commercial software. FFT performs Fourier transform. Lame compresses a wave file into mpeg3 format. Bsort and Qsort represent a general set of algorithms that are interesting to us because of the well known algorithmic complexities.

We modified the FFT benchmark. Originally, it generated an array of random coefficients and then performed an FFT transformation over this data. We divided FFT into two programs; the first one generates the coefficients and the second one performs the actual transformation. We benchmarked only the second program.

10.1 Experimental Setup

We have a pool of 5 handhelds; all of them are Compaq H3800 iPAQs. We allocated the handheld randomly for test cases. This provided our results to be more general rather than being specific to one device. The server is a high performance Pentium III laptop. In all of our tests, the display of the handheld was turned off using the screen saver, and the server was dedicated to the experiment. The characteristics of the devices are given in **Table 5.**

Table 5: Comparison of Client vs. Server

	Server	Client
Device	IBM Thinkpad T23	Compaq H3800
CPU	Pentium III	Intel StrongArm
FPU	Yes	No
CPU Clock Speed	1100 Mhz	206 Mhz
Memory	640 Mbyte	64 Mbyte
I. Cache	16 Kbyte	16 Kbyte
D. Cache	16 Kbyte	8 Kbyte
L2 Cache	512 Kbyte	N/A
Wireless	Lucent Orinoco Gold	
Battery	online	1400 mAh
Sleeve	N/A	Dual
OS	Linux	
Kernel	2.4.17	2.4.18-rmk1

The H3800 has a 206 Mhz Strongarm CPU and 64 MB SDRAM. The StrongArm CPU is designed for low power devices and provides advanced features such as dynamically scaling down CPU clock frequency and switching to different power savings modes[19].

The wireless card is a major power consumer. **Table 6** shows the battery drain time of a fully charged H3800 under different configurations. Comparing the last two lines indicates that when the system is idle, suspending the wireless card doubles the total lifetime of battery. Moreover, removing wireless card from the iPAQ adds another 20% to total lifetime suggesting it still consumes a considerable amount of power in suspended state.

Table 6: Ipaq Battery Lifetime -CPU idle

Sleeve status	NIC status	Run time(s)	Battery(mAh)[3]
none	none	48476	1400
suspended	none	113271	3240
active	none	109389	3240
active	suspended	89657	3240
active	active	42527	3240

The H3800 has a Li-Ion based battery. The Li-Ion type of batteries are very stable against wear and do not suffer from memory effect, i.e. decrease in available capacity due to recharge cycles. In fact, during the evaluation of our mechanism, we measured the battery capacity at random times. Our 13 different measurements show that the average capacity was 1379mAh, with a minimum of 1325mAh and a maximum of 1428mAh. These numbers are very close to the rated capacity (1400mAh). We observed no perceivable change in battery capacity over time.

Based on the high-stability of Li-Ion batteries, we assume that the fluctuations in battery capacity are negligible and define a new metric, *Consumed Battery Rate* or *CBR*, to describe our results. CBR indicates the ratio of battery consumed assuming total capacity of battery is one million units. In other words, 1 CBR equals to $1 * 10^{-6}$ of total energy stored. CBR is only an approximate metric, and it is not portable between different devices unless battery types and capacities are same. However, we believe, it is an adequate metric for the scope of our evaluation.

10.2 Remote execution results

The results of remote and local execution are given in **Table 7**. For each test case, we first fully charged the iPAQ and then

[3]Rated battery values are taken from Compaq's web site. The web site gives the battery capacity of a single slot sleeve as 920mAh and claims double slot has twice the capacity of single slot sleeve.

Table 7: Remote Execution Results

Name	Input	Number of iterations: Local	Number of iterations: Remote	Increase Ratio
bsort	100K	241	2545	10.56x
qsort	40K	40351	6794	0.17x
fft	128K	3056	4175	1.36x
djpeg	monalisa.jpg	8966	3361	0.37x
cjpeg	inputsmall.ppm	88634	7980	0.09x
lame	small.wav	319	6617	20.74x

forced the RPP to locally or remotely execute the benchmarks repeatedly until the battery runs out of power. There was no other device using the wireless network, but we did not have control over noise. To study whether the network bandwidth is effected by noise, we logged the rate of data transfer in 20 seconds intervals, and found that no major fluctuations occurred in data transfer rate.

We report the number of iterations in the third and fourth columns. The fifth column shows the ratio of remote executions to local executions.

The benchmark that benefited most from remote execution was Lame. This is not surprising because it is a very resource-demanding program. For bsort, the remote execution may not be attractive for small files, however with its $O(n^2)$ complexity, the cost of computation grows quickly with a large input size. Bsort is also a memory bound program; it continuously accesses memory for retrieving data so larger caches and faster CPU of the server make remote execution a very attractive solution. FFT depends on the floating-point operations heavily, so it also benefits from remote execution.

Qsort, Djpeg and Cjpeg did not benefit from remote execution. All three of the programs are short running and do not need a lot of computational resources (refer to Tables 12 and 14 to compare run times). The current policy of RPP is not suitable for these programs, because:

- The RPP client contacts Server in 1 second intervals for retrieving completed jobs
- Suspending and reactivating the network interface card is generally instant; however, occasionally the device drivers may not initialize correctly and start a full re-initialization cycle. We noticed that it could take as long as a few seconds to re-initialize.

A better policy would be to estimate execution time, and for short programs, using a server-initiated policy without suspending the client network interface. We are planning to study this policy as part of a future work.

10.3 Remote Execution Cost Prediction

In this section, we evaluate the model described in **Section 3.1**. First, we briefly discuss the factors effecting radio transfer cost. Then we describe several experiments that we conducted for creating a network profile (left top box in **Figure 18**) and compute p_{tx} and p_{rx} based on it. Finally, we predict the remote execution cost using the CBR metric and analyze the results.

10.3.1 Profiling Network Transmission Cost

Our first test evaluates the communication protocol of RPP to find p_{tx} and p_{rx}. The protocol has two parts:

i. Transfer: The client sends a packet to the server and the server acknowledges with a 2 byte reply.

ii. Receive: The client sends a 2 byte request to the server and the server returns a packet.

Table 8: Data Transmit (Rate=11Mb/s)

Packet Size(Bytes)	Avg. Bw.(Kbytes/s)$\pm\sigma$	Time(s)
128	50.5 $\pm$0.4	22300
256	88.6 $\pm$1.1	22480
512	146.8 $\pm$1.3	22040
1024	221.0 $\pm$2.2	21920
2048	281.0 $\pm$2.3	19700
4K	374.2 $\pm$6.1	19540
8K	400.3 $\pm$4.8	18540
32K	447.2 $\pm$30.5	17781
64K	486.1 $\pm$6.7	18340

Table 9: Data Receive (Rate=11Mb/s)

Packet Size(Bytes)	Avg. Bw.(Kbytes/s)$\pm\sigma$	Time(s)
128	54.2 $\pm$0.4	21540
256	89.0 $\pm$1.2	22600
512	156.13$\pm$1.9	21760
1024	211.3 $\pm$1.5	20920
2048	262.7 $\pm$1.4	20461
4K	*211.1 $\pm$5.7*	22600
8K	373.7 $\pm$9.9	21060
32K	400.7 $\pm$9.9	21760
64K	432.2 $\pm$9.6	20400

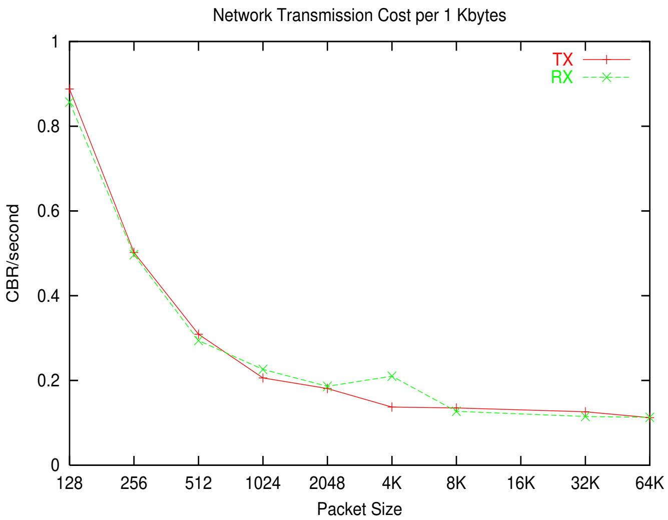

Figure 20: Network Cost

Table 10: Effect of Distance 10Meters

Dir.	Size(Bytes)	Avg. Bw.(Kbytes/s)$\pm\sigma$	Time(s)
TX	8K	401.3$\pm$7.1	18020
TX	32K	456.9$\pm$23.2	18760
TX	64K	438.4$\pm$24.16	18040
RX	8K	369.5$\pm$11.4	21010
RX	32K	402.2$\pm$14.3	21760
RX	64K	432.6$\pm$13.7	20480

Table 11: Remote Execution Protocol Costs

Name	Drain time (seconds)	Power (CBR/s)
idle	89657	11.15
pcost+idle	52740	18.96

We created a setup that includes one H3800 as the mobile node and the server unit. Then we executed the protocols above for various packet sizes. We monitored the effective network bandwidth, total bytes transferred and duration of the battery life of H3800.

Tables 8 and **9** report the results for upload and download, respectively. The first observation is that link utilization increases with packet size. The reason for lower bandwidth for smaller packets is caused by the per packet overhead in the TCP, IP, and MAC layers. We also observed a slight but stable decrease in total battery lifetime with the increasing packet size during data transmits. This is because of the acknowledgement policy in 802.11b protocol[21]. For a given data transfer period, the overall energy cost is equal to the sum of $E_{ack} + E_{nictx} + E_{nicrx}$ where E_{ack} represents energy used waiting idle, i.e. for acknowledgement, and E_{nictx} and E_{nicrx} represent energy used for transmit and receive. In most of the radio cards including the Lucent Orinoco[2] the transmit is more expensive, i.e. $Enictx > Enicrx \cong E_{idle}$. For transmitting larger packets, E_{nictx} becomes the most dominant cost factor. With smaller packets, more acknowledgements are required in a given period so E_{ack} is dominant, decreasing the energy cost slightly. We do not observe the same behavior in receive, because E_{idle} and E_{nicrx} are approximately same.

Note that in **Table 9** the effective bandwidth decreases when packet size increases from 2048 bytes to 4K. We tried the test repeatedly and found the results to be similar. We did not investigate the root cause of the problem as it is beyond the scope of this paper.

Figure 20 shows p_{tx} and p_{rx} in units of *CBR* based on **Tables 8** and **9** for different packet sizes. Our results indicate that, even though the NIC spends more energy for transmit, the application level cost for transmit and receive is approximately same. This is because of the asymmetric network bandwidth of receive and transmit which is caused by the difference in CPU power of the server and mobile node.

We also studied the effect of distance between nodes. We rearranged the test setup such that the server and mobile device is separated with 2 walls of medium thickness and 10 meters. We studied packet sizes 8K, 32K and 64K which we thought would be more vulnerable to transient noise that can occur at longer distances. The results are shown in **Table 10**. The first column gives the direction of data transfer; TX for transfer and RX for receive. The results are not significantly different from the previous results, which leads us to believe that our cost parameters are valid for a single-user environment.

10.3.2 Testing The Remote Power Model

To emprically evaluate our remote power model, we first calculate the cost of remote execution protocol, i.e., the mechanism to offload program from client to server and to check status of program execution which also includes the cost of network card hibernation. For this, we use a dummy program, called pcost, that spins in an idle loop. After downloading pcost to server, the client waits for its completion by regularly (1 sec intervals) checking the status of execution.We show the results in **Table 11**. The first row of the table shows the cost when we disable the control protocol and keep the network interface in hibernated state. The second row shows the cost when the control protocol is enabled.

Table 12 shows the results of remote execution cost estimation for the same input set as **Table 7**. The third column gives the size of the data migrated between client and server, which is the sum of the input and output files. The next column reports the execution time of a job that we calculated by remotely executing the program 10 times and averaging the results. The fifth and sixth column show the estimated cost of network transfer ($E_{tx} + E_{rx}$) and execution cost ($E_{idle} + E_o$).

The last three columns compare our prediction results with actual results. First, we use the total predicted cost (given in column 7) to estimate how many times a fully charged H3800 can remotely execute the program before it runs out of battery. The next column shows the same data measured in a real setup.

Excluding **djpeg**, the average error rate was 10% with a maximum of 18.8%. This is well in acceptable range because of these reasons: First, each test cites a time span of 10-15 hours, and even though no other wireless device uses the medium, noise and other effects can cause deterioration in signal level changing cost per packet. Second, there might be fluctuations in the amount of energy stored in battery. Third, the estimation error, i.e. the difference between the profiled execution time and real execution time accumulates during repeated iterations.

However, the error rate was unusually high in **djpeg**, so we re-studied it with three different input sets. We also included **lame** in this study. The results are in **Table 13**.

The error rate for both **lame** and **djpeg** are similar to the first study. The average estimation error for **lame** was 12.9% which was higher but still within the acceptable levels. However **djpeg** produced an error rate of 47%.

We profiled the RPP and **djpeg** at the method level and found out that almost all our benchmarks suffer from a phenomenon created by file system write operations. The iPAQ's use Compact Flash as the permanent storage device, but unlike the hard drives, a CF card can be erased and written

Table 12: Remote Execution Cost Estimation Results

Name	Input Name	Migrated Data Size (Bytes)	Execution Time (s)	Network (CBR)	Execution (CBR)	Total (CBR)	Estimated Iterations	Measured Iterations	Error %
bsort	100K	200008	21.6	23.5	409.5	433	2309	2545	9.2
qsort	40 K	80008	6.3	9.3	119.5	128.8	7776	6794	14.4
fft	132K	264008	9.0	30.8	170.7	201.5	4963	4175	18.8
lame	small.wav	209880	6.2	23.5	117.6	141.1	7087	6617	7.1
cjpeg	input_small.ppm	206476	5.3	25.5	100.5	126.0	7936	7980	0.5
djpeg	monalisa.jpg	188596	9.2	21.2	174.6	195.77	5108	3361	51.9

Table 13: Djpeg and Lame Re-test Results

		Migrated Data	Execution	Network	Execution	Total	Estimated	Measured	Error
djpeg	T99Fig12.jpg	430092	10.4	56.3	197.2	253.5	3944	2887	36.6
djpeg	maptogrc.jpg	241197	6.7	28.1	127.0	155.1	6447	3727	72.9
djpeg	sb_image.jpg	82151	5.1	9.6	97.5	107.1	9337	7038	32.6
lame	write.wav	37732	4.8	4.6	91.1	95.6	10455	9516	9.8
lame	exitWindows.wav	148849	6.2	18.3	117.6	134.9	7410	6721	10.2
lame	notepad.wav	48550	4.6	5.9	87.2	93.1	10741	9038	18.8

only a limited number of times (typically 100000); to ease the pressure on CF, our version of Linux uses a special file system, JFFS2[44]. Our observation is that the combination of CF and JFFS2 introduces long and unpredictable write operations. This unpredictability impacts **djpeg** most, since unlike the other benchmarks, it reads small input files but writes large output files.

We conducted four different experiments to observe the effects of writes on total execution time. We remotely executed the **djpeg** for each combination of configurations:

1. different file systems (tmpfs and jffs): tmpfs is a memory based file-system (i.e. ram-drive). Hence, it is very simple and efficient.

2. different write blocks sizes (4 KB, 256 KB)

For each case, we ran the **djpeg** 10 times remotely and recorded execution time. **Figure 21** gives the execution time and standard deviations in each case. As the graph shows, the fluctuations in total response greatly increases with *Jffs*, this is also the reason of higher error rate in Djpeg.

10.4 Local Execution Cost Estimation

In this section, we first present the effectiveness of offline profiling and **Equation 7**. Then we evaluate the *Program Characterization* algorithm developed in **Section 9.2.2**.

We used the output of the DS2760 ASIC circuit to monitor program energy cost. Our version of Linux reports this through the proc interface. We converted the units to CBR using the equation:

$$CBR = (1000000 * ASIC)/12960^4$$

10.4.1 Offline profiling

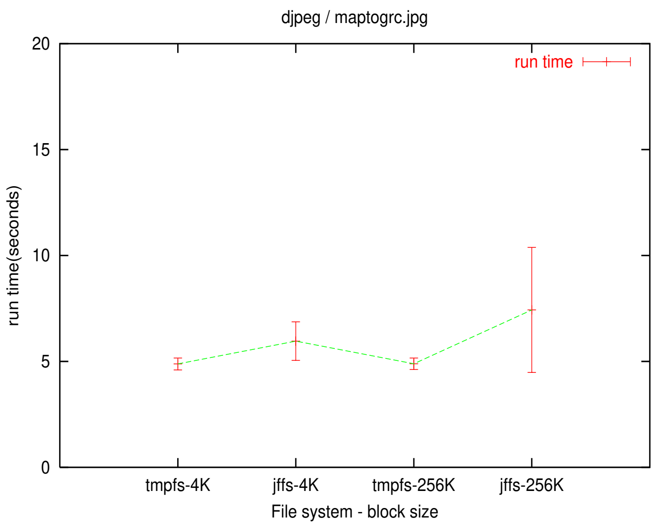

Figure 21: Flash file system comparison

[4]=total rated battery/ASIC precision=3240mAh/0.25mAh

Table 14: Local Execution Cost Estimation Results-based on offline profiling

Name	Input Name	Execution Time (s)	E_m CBR	E_s CBR	E_l CBR	Estimated Iterations	Measured Iterations	Error %
bsort	100K	145.0	3112	347.2	3536	282	241	17.0
qsort	40K	0.9	17.7	2.3	20	49840	40351	23.5
fft	128K	12.3	257.7	29.3	287	3483	3056	13.9
lame	small.wav	115.6	2340	277	2617	382	319	19.7
cjpeg	inputsmall.ppm	0.4	8.48	1.54	10	99692	88634	12.4
djpeg	monalisa.jpg	3.8	79.5	9.2	88.7	11269	8966	25.6

Table 15: Test Cases for Growth Rate

No	Benchmark Name	Device Cfg.	Training Set Input Size (energy cost in CBR units)
1	bsort	Case 1	10K, 30K, 300K (36.3, 306.3,29706.8)
2	bsort	Case 2	10K, 30K, 300K (23.1,200.6, 15663.6)
3	qsort	Case 1	10K, 30K, 100K, 200K (3.9, 17.0, 56.3, 131.2)
4	qsort	Case 2	10K, 30K, 100K, 200K (6.9,20.8,66.4,151.2)
5	lame	Case 1	refer to **Table 16**

We collected the profiling data by executing the programs on an H3800 which runs on battery power. We ran each program 3 times and collected the energy cost and run time. For *qsort,cjpeg and djpeg*, the precision of the power sensor was not enough, so we ran the programs 30 times in a row and averaged the cost.

For estimating the energy cost, we also need an estimate of P_s, i.e., mean power consumption of sleeve (**Equation 7**). We used the second and fourth row of **Table 6** to find P_s in terms of CBR units and then converted these into ASIC. Finally, we used **Equation 7** to predict cost of each program.

We verified our results using the same mechanism that we used for remote execution. First, based on the program energy cost, we estimated how many times the program can run on a fully charged H3800. Then, we measured the number of iterations on a real setup and calculated the error rate.

Table 14 reports both the profiling data and results. The average error rate between the measured and estimated iterations is 18.6 with a standard deviation of 4.7. The most important reason for the error rate is the slight variations in battery capacity from one charge to another. The test results also suggest that our estimation for P_s is much lower than the real value because for all tests, the number of iterations are *always* overestimated. Furthermore, we did not see any anomaly in *djpeg* or any other particular test case. This is because, in local execution we have direct access to power sensors. In contrast, our evaluation mechanism for remote execution is weaker, since we use the estimated total battery capacity twice in our calculations; first to estimate cost of network transfer and second to estimate number of iterations. Thus the error rate accumulates.

10.4.2 Predicting program cost using growth rate

Table 16: Lame-Cost Prediction Test Cases

Name	Input (bytes)	Output (bytes)	Measured (CBR units)
write.wav	34388	6554	745
Critical Stop.wav	48402	9079	976.8
Character Map.wav	71732	13486	1157
exit.wav	136310	25095	2006

In this section we evaluate the effectiveness of our proposed mechanism to predict energy cost using the algorithmic growth rate. We test our algorithm on three benchmarks *bsort, qsort and lame*, which have growth rates $O(n^2)$,$O(nlog(n))$ and $O(n)$) with respect to the magnitude of the input size. We did not evaluate the other benchmarks because of the complexity required to model their behavior on different inputs.

We classify the test cases into two groups to observe the effects of trickle charge.

1. **Case 1:** The main unit's battery is fully charged, sleeve unit is not connected (no trickle charge).
2. **Case 2:** The main unit's battery is approximately half full, sleeve unit is connected and sleeve battery is full (trickle charge).

Table 15 lists the details of our test configurations. The third column gives the hardware configuration. The fourth column is the cs_i, i.e., the energy cost for a given input, pairs; for each case, the upper row indicates the input size and the lower row indicates measured energy cost for the corresponding input. *Lame* is different from the other programs since the input and output sizes are different for each input. As such, we preferred to report on it in a separate table (**Table 16**).

For each of the five cases, we used the algorithm *CREATE_REGIONS* to divide the input space into regions based on the *input size*. Next, we used random data files and estimated the energy cost based on the closeness of a region (see **Section 9.2.2**).

Tables 17 and 18 show the results. The average error rate is 9.3% with only two cases being higher than 20%. We further investigated these two cases. The main reason behind the high error rate for **bsort** (row 7) is the insufficient number of samples, i.e., only 3, and their unbalanced distribution.

Table 18: Lame-Cost Prediction Results

Benchmark Name	Input Size (bytes)	Output Size (bytes)	Measured Cost (CBR units)	Predicted Cost (CBR units)	Error %
Backup.wav	38294	7469	745.4	754.6	1.2
notepad.wav	44370	8487	822.5	874.2	6.2
Hyperterminal.wav	52758	9975	874.2	1040.1	18.9
CD Player.wav	53598	10120	951.4	951.4	0.0
Calculator.wav	55284	10278	925.9	892.0	3.6
CloseProgram.wav	66590	12354	1080.2	1074.1	0.6
EmptyRecycleBin.wav	67538	12598	1054.0	1089.5	3.4

Table 17: Cost Prediction Results

test case	Input Size (bytes)	Measured (CBR units)	Predicted (CBR units)	Error %
bsort (1)	20K	136.6	135.8	0.56
	40K	545.5	544	0.28
	100K	3240.7	3402.8	5.00
	200K	12345.7	13611	10.25
bsort (2)	20K	92.6	92.6	0.00
	40K	339.5	363.4	7.04
	100K	1774.7	2271.6	28.00
	200K	7021.6	7938.3	13.05
qsort (3)	20K	8.49	10.03	18.18
	40K	22.4	24.7	10.34
	300K	200.6	211.4	5.38
	400K	290.1	294.7	1.59
qsort (4)	20K	10.0	14.7	46.15
	40K	30.9	37.0	20.00
	300K	246.9	244.6	0.93
	400K	297.8	343.4	15.2

The algorithm cannot find any region that covers $100K$ so it uses $10K$ and $30K$ to estimate it. Had we used more samples, the algorithm could adjust itself by choosing the region boundaries more effectively.

The reason for high error rate in **qsort** (row 13) is quite different. The energy cost for $10K$ seems to be quite high, which could be caused by some spike in the operating system activity during our test. Hence, the algorithm correctly rules out $10K$ but because there is no other sample next to $30K$ it cannot establish a region in this part of input space. This leads to a high error rate for $20K$ prediction.

11. COMPARISON WITH RELATED WORK

Recently, numerous proposals have been made to improve energy savings using software assisted techniques. Some of the most promising techniques are dynamic CPU voltage scaling [42][33], efficient peripheral devices management [11][38],[25] and energy aware code generation[22]. These approaches have a common point; they try to optimize energy consumption by adjusting hardware performance such as slowing down hard drive or choosing slower but more efficient instructions. The above techniques are not very effective for high-performance applications, for example slowing down the CPU would increase the total execution time which means more energy use by the display or other external devices.

In the other hand, remote execution takes a different approach by shifting the computation to another machine that does not depend on power. This approach is complementary to the techniques since it provides an alternative mechanism when adjusting hardware performance is not desirable.

A few groups have proposed different remote execution mechanisms. We describe the work of Kremer et.al.[26] and Li et.al.[47] in **Section 4.1**. In summary, their strategy is analyzing the program during compilation and then dividing it into tasks which can be offloaded to a remote server. They use execution time as a energy cost metric.

However, because these works are dependent on compile-time analysis, their method may not always choose the optimal path. The cost of remote execution might change with the increase in server CPU load or decreasing wireless network capacity due to noise. In addition, it may not be desirable to let each application manage the hardware (i.e. hibernate) if multiple tasks are running.

We describe the study of Rudenko et.al. [35][34] in **Section 6.1**. In summary, they present a client-server based remote execution framework. The client estimates the cost of local and remote execution and if remote execution is beneficial then it transfers the files to server for remote execution.

Our study has many improvements over their proposal. First of all, our platform uses current sensors to dynamically measure and adjust cost of programs. In contrast, their method is based on APM data, which is dependent on battery voltage level. The second difference is the platforms. Rudenko's tests are done on laptops but we use handheld devices. Third they measure the energy cost of the entire unit. However, this is not possible for platforms such as iPAQs because of the unavailability of sensors in sleeve units. We combine the output of energy sensors with the offline profiling to estimate total cost. Finally, we present an algorithm that can increase the precision of cost estimation by using various attributes of input file.

12. CONCLUSION

Recently, many researchers realized that energy is becoming the most important resource in mobile computing due to the limitations of batteries. As a result, the studies in the energy aware software field have gained momentum.

There are many dimensions to energy aware software, some of these are explained in [27, 45]. In this paper, we describe two of them: Improving the utilization of hardware using software and offloading computation. The former techniques target decreasing the energy consumption of hardware components that have software selectable power/performance levels. The latter techniques push the computation to another computer that is continously connected to wall-power.

We present a new computation offloading platform in this paper. Some of our improvements over existing platforms are:

- A detailed study and modeling of iPAQ H3800
- An energy cost prediction system that combines smart battery monitors and profiling
- An algorithm that uses program algorithmic complexity and attributes of data to better estimate energy consumption

In the short term, we are planning to improve RPP. In particular, our research will focus on:

- Improving the energy cost prediction capability
- Improving the efficiency of network transfer, especially by more aggressively scheduling network interface.
- Developing a cost database to automate local/remote execution decision

In the long term, we are planning to address two important problems of energy aware software.

- Offloading computation in a *heterogeneous* environment to save energy: A static compilation system requires a separate client and server for each architecture. This might not be the best choice from administrative view, i.e. deploying and maintaining software on different architectures.
- Integration of hardware utilization algorithms: Generally, each algorithm targets improving the efficiency of one hardware component, however, the components are not independent of rest of the system. A decrease in performance level of one component might effect the efficiency of other components.

13. REFERENCES

[1] *ns*. http://www-nrg.ee.lbl.gov/ns/.

[2] Agere Systems. *Orinoco World PC card Datasheet*, 2001. http://www.orinocowireless.com/products/.

[3] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeno - a compiler supported java virtual machine for servers. In *Workshop on Compiler Support for Software System (WCSSS 99)*, May 1999.

[4] AMD Corporation. *AMD PowerNow Technology - White Paper*, November 2000. Publication no=24404.

[5] Apple Corporation. *Apple Powerbook*. www.apple.com/powerbook/.

[6] Compaq Computer Corporation. *iPAQ Pocket PC*. http://www.compaq.com/products/handhelds/pocketpc/.

[7] Compaq Computer Corporation. *iPAQ H3800 Series Expansion Pack Developer's Guide*, third edition, 2002.

[8] Dallas Semiconductors. *DS2760 Data Sheet*. http://pdfserv.maxim-ic.com/arpdf/DS2760.pdf.

[9] Dallas Semiconductors. *DS2437 battery monitor*, 1999.

[10] D.Narayanan, J. Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *Third Workshop on Mobile Computing Systems and Applications*, December 2000.

[11] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*, 1995.

[12] D. Eckhardt and P. Steenkiste. Measurement and analysis of the error characteristics of an in-building wireless network. In *SIGCOMM*, pages 243–254, 1996.

[13] J. Flinn. *Extending Mobile Computer Battery Life through Energy-Aware Adaptation*. PhD thesis, Carnegie Mellon University, December 2001.

[14] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *Hot Topics in Operating Systems(HotOS-VIII)*, pages 61–66, Germany, 2001.

[15] G. H. Forman and J. Zahorjan. The challenges of mobile computing. *IEEE Computer*, 27(4), april 1994.

[16] P. Gauthier, D. Harada, and M. Stemm. Reducing power consumption for the next generation of pdas: It's in the network interface. In *Proceedings of the International Workshop on Mobile Multimedia Communications (MoMuC)*, September 1996.

[17] J. Gosling and H. McGilton. *The Java Language Environment- A White Paper*. Sun Microsystems. http://java.sun.com/docs/white/.

[18] H.Saputra, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. Hu, H.Hsu, and U.Kremer. Energy conscious compilation based on voltage scaling. In *LCTES02-SCOPES02*, June 2002.

[19] Intel. *StrongARM SA-1110 Microprocessor Developer's Manual*, October 2001. Order Number:278240-004.

[20] Intel Corporation. *Xscale*. www.intel.com/design/intelxscale/.

[21] ISO/IEC. *IEEE 802.11 Standard*. IEEE Standard for Information Technology, ISO/IEC 8802/11;1999(E),1999.

[22] M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Design Automation Conference*, pages 304–307, 2000.

[23] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[24] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.

[25] R. Kravets and P. Krishnan. Application-driven power management for mobile communication. *Wireless Networks*, 6(4):263–277, 2000.

[26] U. Kremer, J.Hicks, and J.M.Rehg. A compilation framework for power and energy management on mobile computers. In 14^{th} *International Workshop on Parallel Computing (LCPC'01)*, August 2001.

[27] J. R. Lorch and A. J. Smith. Software strategies for portable computer energy management. *IEEE Personal Communications*, 5(3):60–73, june 1998.

[28] J. Meyer and T. Downing. *Java Virtual Machine*. O'REILLY, 1997.

[29] C. Narayanaswami, N. Kamijoh, M. Raghunath, T. Inoue, T. Cipolla, and J. Sanford. Ibm's linux watch: The challenge of miniaturization. *IEEE Computer*, pages 33–41, january 2002.

[30] M. Newsome and D. Watson. Proxy compilation of dynamically loaded java classes with mojo. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 204–212. ACM Press, 2002.

[31] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 276–287. ACM Press, 1997.

[32] J. Palm, H. Lee, A. Diwan, and J. E. B. Moss. When to use a compilation service? In *LCTES02-SCOPES02*, June 2002.

[33] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proc. International Symposium on Low Power Electronics and Design*, pages 76–81, August 1998.

[34] A. Rudenko, P. Reiher, G.Popek, and G.Kuenning. The remote processing framework for portable computer power saving. In *ACM Symp. Appl. Comp.*, San Antonio,TX, February 1999.

[35] A. Rudenko, P. Reiher, G. Popek, and G. Kuenning. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review*, 2(1):19–26, january 1998.

[36] E. G. Sirer, R. Grimm, A. Gregory, and B. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *ACM Symposium on Operating System Principles (SOSP'99)*, volume 34, pages 202–216, December 1999.

[37] T. Starner. The challenges of wearable computing-part 1. *IEEE Micro*, pages 44–52, July 2001.

[38] M. Stemm and R. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Communications, vol.E80-B, no.8, p. 1125-31*, E80-B(8):1125–31, 1997.

[39] T.Cormen, C.Leiserson, and R.Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1998.

[40] M. Theimer and B. Hayes. Heterogeneous process migration by recompilation. In *11th International Conference on Distributed Computing Systems*, May 1991.

[41] Transmeta Corporation. *Crusoe*. http://www.transmeta.com/technology/index.html.

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

[43] M. Wilson, C. Zawodzinski, and M. Daugherty. Small battery - fuel cell alternative technology development. In *DOE Hydrogen Program Review*, 2001.

[44] D. Woodhouse. Jffs : The journaling flash file system. In *Ottawa Linux Symposium 2001*, July 2001.

[45] H. Zeng, C. Ellis, A. Leveck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[46] C. Zheng and C. Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *IEEE Computer*, 33(3):47–52, Mar. 2000.

[47] Z.Li, C.Wang, and R.Xu. Computation offloading to save energy on handheld devices:a partition scheme. In *Proc. of International Conference on Compilers,Architectures and Synthesis for Embedded Systems (CASES)*, pages 238–246, 2001.