Profile-Guided Java Program Partitioning for Power Aware Computing*

Sriraman Tallam Rajiv Gupta
Dept. of Computer Science
The University of Arizona
Tucson, Arizona 85721
{tmsriram,gupta}@cs.arizona.edu

Abstract

In embedded devices, like PDAs, limited battery power presents a bottleneck while running power-hungry applications. Hence, it is important for such applications to be energy aware. Consider a scenario where the embedded device is wirelessly linked to a tethered server and can exploit the computing cycles in it. This presents an opportunity to execute parts of an application remotely and reduce computing energy on the embedded device. However, such a scheme is limited by the transmission energy needed in transferring program parts to be executed remotely. In this paper, we present a fast profile-guided partitioning algorithm that can choose the program parts that can be executed remotely and save energy on the embedded device. We also discuss an optimal version of the partitioning algorithm, also based on profiles, that can maximize the energy savings on the embedded device. However the optimal algorithm has a large time and space complexity and hence needs a lot of energy to execute. We argue that the partitioning might have to be done on the client device more than once and hence the need for a fast algorithm that does not consume too much energy by itself. We have chosen Java programs in particular because of their widespread use in embedded applications. We treat Java objects as the program parts that could be considered for remote execution. Our algorithm can, in general, be applied to programs written in any Object-Oriented language. Experiments show that our approach can conserve energy in embedded devices from 29% to 43%. We also present results comparing the fast algorithm with the optimal one.

Keywords Power aware computing, Program partitioning, embedded systems, Java.

*Supported by grants from Intel, Microsoft, IBM and NSF grants CCR-0324969, CCR-0220334, CCR-0208756, CCR-0105355, and EIA-0080123 to the University of Arizona.

1. Introduction

Mobile devices like Personal Digital Assistants(PDAs), laptops, cell phones, etc. already have very powerful computing and communication capabilities but depend on battery power for their operation. Hence, it is important that applications for these devices are aware of the energy constraint and are able to perform reasonably without overrunning this critical resource.

Consider a scenario where these mobile devices can connect wirelessly to a remote server and can use its computing cycles. This presents an opportunity to save computing energy on the mobile device by executing program parts remotely on the server. A number of public places already offer wireless services to users who connect to their network. We expect that, in future, it will be increasingly possible for a connected user to also execute programs on wireless servers.

The above scheme, however, suffers from communication overhead as the mobile device needs significant energy for wireless transmissions. It would not benefit if the energy savings obtained from reduced computing do not exceed the communication energy expended. This gives rise to the problem of trying to partition the application so as to obtain an overall energy savings.

In this paper, we propose a fast program partitioning algorithm, based on program profiles, that can conserve energy on the mobile device. We partition Java programs at the object level. A Java application consists of a set of objects interacting by invoking methods on each other. A partition of an application divides the set of objects into two subsets, local and remote objects. Local objects are created in the mobile *client* device and methods invoked on these objects use the computing power of the mobile device. Remote objects are created on the wireless remote *server* and methods invoked on them use the server's computing cycles. Computing energy in the client is reduced by executing the remote objects in the server. Transmission energy is expended due to transfer of data between local and remote objects.

Our partitioning algorithm is fast but it does not guarantee optimal partitions. An optimal partition would be one that gives us maximum energy savings on running the application. We discuss another algorithm based on MIN-CUT that can give optimal partitions. This algorithm is similar to the one proposed in [12]. However, this algorithm suffers from a time complexity that is nearly cubic in the number of objects and a space complexity that is linear in the number of objects. It should be noted that partitioning is dependent on variables like communication energy per byte and battery power. If one of these variables change, we must recompute the partition before the application can be run. Hence, though the partitioning process is always done offline, it has to be performed only in the client. For instance, the communication energy needed per byte could change if the client connects to a different wireless network. The partitioning process itself consumes some of the client's energy and hence it is important that it be time and space efficient.

Some of the key applications for embedded systems are image processing and text processing applications. We have evaluated the effectiveness of our partitioning scheme on two applications. The first application finds a sub-matrix within a huge matrix. This is a key operation in image processing because an image is a two-dimensional matrix of pixels and looking for a sub-image within it is equivalent to finding a pixel sub-matrix. The second application is a text-processing application that searches for a piece of text in a huge document.

The applications described above exhibit the following properties. They manipulate large data like image matrices and documents. Also, a considerable amount of processing is involved in finding sub-matrices and searching texts. So, trivial partitioning solutions to such applications like executing them entirely on either the client or server will not work. The two extremes, one, executing them entirely on the client, will need too much computing power and second, taking all processing to the server, will need a lot of transmission energy. Hence, a partition that produces an intermediate solution is sought.

For experiments, we have used energy parameters of the Itsy [5] and the Compaq WL100 wireless card [1]. We have also varied these energy parameters for generality of results. Our experiments on the two applications show that we can conserve from 29% to 43 % of energy on the client.

The rest of the paper is organized as follows. The partitioning process is described in the next section. Section 3 discusses the fast partitioning algorithm. The experimental results are presented in Section 4. Related work is in Section 5 and Section 6 concludes.

2. Overview of Our Approach

In this section we describe how partitioned applications are executed and then provide the key steps in partitioning an application. Given a partition, the application is executed as follows. A server process is created on the server side and is responsible for running the remote portion of a Java program. The server accepts requests from the client to create objects and invoke methods on them. When the client asks the server to create an object of a class by giving the class name, the server creates the object and passes back a remote reference to the client. This object resides on the server. The client invokes a method of this object by giving its reference, the method name and the method parameters. This method is then executed on the server. This way any object can be created remotely and methods can be invoked on them.

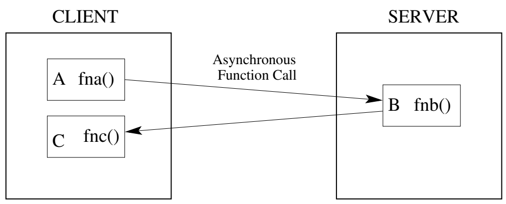

Figure 1. Illustration of Partitioning

For instance, consider Figure 1, where objects A and C reside on the client side and object B resides on the server side. Now, if object A wants to invoke a method of object B, say fnb(), it uses the remote reference to figure out that B is a remote object. It then makes an asynchronous function call to object B, pushes a state on a call stack and idles. The asynchronous call is achieved by sending a message that contains the object's function to be invoked and the function parameters. We use asynchronous function calls to avoid the client from being blocked while the call is completed. This is because object B could invoke a method of object C, to complete the call to fnb(), and hence the client must be able to receive this request from the server and respond. Both the client and server have a table that stores where each object is placed. Each time a call is completed, the corresponding state in the stack is popped off. The communication between the client and server is achieved using sockets. The main advantage of the above approach is that it does not impose any arbitrary restrictions on partitioning. The only case where an object could be forced to reside in the client is when the application enforces it. For instance, a GUI object will have to be on the client and cannot be moved. We refer to such objects as anchored objects. Thus given an execution instance with N objects, there are 2^N possible legal ways of partitioning the application. However, not all partitions will reduce the energy in the client. So, our goal is to find one of the partitions that can give us good energy savings.

It is assumed that the basic Java class files are available on the server already. This can be expected to be true in a real life scenario. However, any Java class file that is specific to the application has to be sent to the server once before any objects of this class can be created. This transfer of the class file further adds to the transmission energy and is therefore taken into account during energy computations. Also, this remote execution scenario does not change the sequential nature of the program. Although there is an opportunity to execute a few methods in parallel in the client and server we do not do it and this is beyond the scope of this paper.

It should also be noted that partitioning will not affect the application execution delay significantly. This is because the remote server, in general, is expected to be several times faster than the client device and hence remote computing would only be performed faster. This increased execution speed of the server can be typically expected to offset the extra time spent on transmitting class files or data to the server. Thus, it is expected that the impact of partitioned execution on execution delay will not be significant. In fact depending upon the relative speed of the client and server, in some situations the application may execute faster.

In the remainder of this section we discuss the key steps of our approach. The detailed partitioning algorithms will be considered in subsequent sections.

2.1. Key steps

Both the partitioning schemes are profile-guided and the first step is to profile the application for data on number of objects created, communication among objects in bytes and the computing time spent in the context of each object. The key steps are profiling, graph representation and partitioning and are discussed in the following subsections. α, β and γ are the energy parameters where α is the the transmission energy per byte, β is the Peak battery power and γ is Idle battery Power .

Data Profiling

The applications are profiled for the following data:

1. The number of objects of each java class created (N_C).

2. The amount of communication, in bytes, between each object pairs (C_{pair}).

3. The processor time spent in executing in the context of every object (T_{obj}).

The above data must be profiled initially and it needs to be done only once. The data can also be profiled on a small problem size if it has to be done on the client device. Here, we assume that, in general the profiled data will scale well with the problem size and we show in our experiments that this assumption is reasonable.

From the profiled data, the estimates of transmission energies TE_{pair} and computing energies CE_{obj} can be obtained from the following equations.

$$TE_{pair} = \alpha \times C_{pair} \tag{1}$$

$$CE_{obj} = \beta \times T_{obj} \tag{2}$$

The Idle Energy IE_{obj} is given by

$$IE_{obj} = \gamma \times T_{obj} \tag{3}$$

Equation 3 gives a conservative estimate of the idle energy. In general, the remote server is expected to only be faster than the client device and hence, the idle energy can be lower.

Graph representation

In this step, we model the energy estimates obtained in the previous step into a graph. The optimal partitioning solution builds a graph in which there is a node for every object created. The edges represent the computing and transmission energies. The space complexity for such a graph is high as the number of objects created in a typical run of the application is huge. The fast partitioning algorithm however constructs a graph in which there are two nodes per Java class. This is a big reduction in the space needed as the number of classes in a program is generally much lower. These representations are discussed in detail in the later sections.

Partitioning

In this step, the actual partitioning is done using the graphs created in the previous step. The output of the partitioner is, for each Java class, the ratio of the number of objects that have to be placed in the client and server side. A ratio is used because it can be applied to partition another instance of execution that has a different problem size and possibly different number of objects of each class.

The optimal partitioning strategy does a MIN-CUT of the graph to obtain a partition. The fast partitioning strategy however, uses a greedy approach. It takes into account the savings and expense in energy for each object before deciding to place it locally or remotely.

As mentioned earlier, the partitioning has to be recomputed if any of the energy parameters change. However, as an optimization, a history can be maintained that contains the partitions for different energy parameters. A previously computed partition can be reused if parameters match.

Using the ratio at run-time

We want to classify every object that would be created in the program run as either a local or a remote object. Using the ratios obtained from the previous step, we place objects on the client or the server. For instance, a ratio of 2 : 3 for a particular class would mean placing 3 objects remotely for every 2 objects placed locally. This is how a partitioning solution generated offline is used to partition an application at runtime.

2.2. Drawbacks of Optimal Partitioning

In this section we discuss the disadvantages of using the optimal partitioning strategy. Our optimal partitioning strategy based on MIN-CUT is similar to the task partitioning approach in [12]. This approach suffers from both time and space complexity. First, the graph would be very huge as its size is proportional to the number of objects created and that is generally a large number. For the applications we used the number of objects created was in the order of thousands. Next, MIN-CUT has a complexity that is cubic in the number of graph nodes. As mentioned earlier, the partitioning scheme is dependent on the parameters α, β and γ. Any change in these values will mean a repartitioning of the application on the client. Hence, a time-space inefficient algorithm will not be of much value as it would need a lot of energy by itself to execute and can overwhelm the savings obtained. In the next section we discuss our fast partitioning strategy that overcomes these drawbacks and also produces good partitions.

3. Fast Partitioning Strategy

In this section we discuss our fast partitioning approach that has a good time-space bound. We first discuss our modified graph representation and then describe the partitioning algorithm. Our algorithm has a worst-case time that is linear in the number of objects created.

3.1. Graph representation

The graph is constructed as follows. We create two nodes called *class nodes* in the graph for every Java class that is present in the program. One is called the *client class node* and the other is called the *server class node*. This dramatically reduces the space needed for the graph. For instance, in the applications we experimented with, the number of classes were less than 10. Of the two nodes created per class, the client class node stands for the objects created locally and the server class node stands for those created remotely. We also create the two special nodes called client node and server node. Each class node is given a node weight that is equal to the number of objects of that class created. For instance, consider a Java program P that has two classes A and B. Let N_A and N_B be the number of objects of each class. Then, for class A, we will create two class nodes, say C_A and C'_A, that are the client and server class nodes respectively. The node weight on C_A (n_A) represents the number of local objects and the node weight on C'_A (n'_A) the number of remote objects, subject to the equality.

$$n_A + n'_A = N_A \tag{4}$$

The equations for class B are derived similarly. Figure 2 shows the nodes and their weights. In the next subsections, we discuss the different kinds of edges that are generated.

C-Edges

C-Edges are the edges between the client class nodes and the server class nodes. For instance, a C-Edge between node C_A and node C'_B represents the transmission energy, C_{AB}, spent in transferring data between local objects of class A and remote objects of class B. The portion of the graph showing the C-Edges between the class nodes, for the Java program P is illustrated in Figure 2. The C-Edges are shown by bold lines. C_{AA} represents the transmission energy spent in transferring data between local and remote objects of the same class. The values C_{AB} and C_{AA} can be calculated using the equations

$$C_{AB} = \alpha * O_{AB} * n_A * n'_B \tag{5}$$

$$C_{AA} = \alpha * O_{AA} * n_A * n'_A \tag{6}$$

where, O_{AB} is the average data in bytes transmitted between an object of Class A and an object of Class B. O_{AA} is the average data transmitted between 2 objects of Class A. α is the transmission cost per byte. Again, O_{AB} and O_{AA} can be found using the equations

$$O_{AB} = T_{AB}/(N_A * N_B) \tag{7}$$

$$O_{AA} = T_{AA}/(N_A * (N_A - 1)/2) \tag{8}$$

where T_{AB} is the total data in bytes transmitted between all objects of class A and all objects of Class B and T_{AA} is the total data transmitted between objects in Class A.

The reasoning for these equations is as follows. We want to compute the average transmission between an object of class A and an object of class B, O_{AB}. From the profiled data, we know the total transmission between all objects of classes A and B. Now, we assume that all objects of Class A communicate with all objects of Class B. So, the number of possible communication edges is $N_A * N_B$ and hence, Equation 7. For the case where we want the average communication between 2 objects of Class A, the number of communication edges is $(N_A * (N_A - 1)/2)$. This gives Equation 8. The computation of C_{BA} and C_{BB} are similar.

P-Edges

P-Edges are edges between the client node and the server class nodes and also edges between the server node and the client class nodes. The former represent Idle Energies consumed and the latter represent Peak energies consumed. Consider the P-Edge between class node C_A and the server node. This represents the Peak energy, P_A, consumed by local objects of class A. Figure 2 shows the P-Edges for the Java program P. They are shown by dotted lines. P_A can be calculated using the following equation

$$P_A = \beta * t_A. \tag{9}$$

where t_A is the average CPU time consumed by an object of class A. β is the Peak CPU power in Watts. Again, t_A can be calculated using the equation

$$t_A = T_A / N_A \tag{10}$$

where T_A is the total CPU time consumed by all objects of class A. T_A is obtained from the profiled data. The computations of P_B are similar.

The P-Edges representing the Idle energies is calculated as follows. Consider the P-Edge between the client node and the server class node Ci'_A. I_A represents the Idle CPU energy consumed by the remote objects of Class A. The Idle energies can be computed using the equation

$$I_A = \gamma * t_A \tag{11}$$

where γ is the idle CPU power.

3.2. Partitioning Algorithm

In this section, we describe the fast partitioning algorithm that is applied to the graph constructed above. Figure 2 shows the complete graph with all the nodes and edges. We discuss the partitioning with reference to this example graph. A valid partition of the program is finding the values of n_A, n'_A, n_B and n'_B. These values will give us the ratio of number of objects in each class that have to be placed locally and remotely. We now show how to determine these numbers.

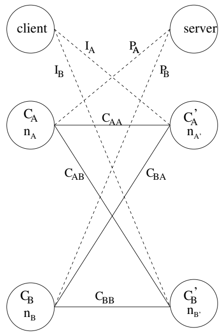

Figure 2. The Partition Graph

Before starting the partitioning we do the following initializations. $n_A = N_A$, $n'_A = 0$, $n_B = N_B$ and $n'_B = 0$. That is, we start off by treating all objects as local. We then consider moving each object based on a cost and benefit analysis. For instance, the benefit in moving an object of class A to the server is

$$benefit = P_A - I_A \tag{12}$$

That is, by moving the object to the server we save on Peak energy but we spend some Idle energy. However, the cost in moving object A to the server is the transmission energy we need to expend in communicating with this object remotely. This cost at any instant of time t can be calculated using the equation,

$$cost = ((n_A - 1) * O_{AA} + n_B * O_{AB}) * \alpha \tag{13}$$

where O_{AB} and O_{AA} are defined in Equations 7 and 8. If at time t, there are n_A objects of Class A in the client, moving one object to the server gives rise to the following. Now, this object has to communicate with the $n_A - 1$ local objects of Class A and n_B local objects of Class B. Multiplying these by the average communication between objects gives the total communication. Also, note that the cost of moving an object of a given class cannot increase with time. This is because n_A and n_B either remain the same or decrease with time. They are set to the maximum value to begin with.

This leads to the following observation. For any object, if the benefit is more than the cost, it is only profitable to move it to the server side. So, we can identify all objects whose benefit values are more than the cost and move it to the server side. Moving n objects of Class A from the client to the server can be achieved by decreasing the node weight of the client class node by n and incrementing the node weight of the server class node by n.

However, it becomes interesting if there is no such object whose benefit is greater than its cost to begin with. Recall that the cost function for an object is decreasing. So, it could happen that if we move some objects to the server side, thereby increasing the energy needed, the cost functions of some other objects could decrease and fall below its benefit. Then, by moving these objects we could again start saving. The few objects that are chosen to be moved initially are those whose $benefit/cost$ ratio is the maximum. This way, the energy that we lose initially is kept to the minimum possible. Also, before doing this step, the current partition state is saved. It could happen that we are not able to find any object whose cost reduces below its benefit. In that case we roll back to the saved state and stop.

If N is the number of objects created and C is the number of classes, our algorithm has a running time of $\bigcirc$ ($C \times N$). Also, the graph needs space that is linear in the number of classes created.

4. Experiments

In this section we discuss the experimental setup. We also describe the applications we used to test our partitioning algorithm and the results we obtained.

4.1. Experimental setup and implementation

We describe the experimental setup next. We have used the energy parameters of the Itsy [5] and the Compaq WL100 wireless card [1]. [15] have used the same constants when they measured energy benefits of remote compilation. Table 1 lists the constants and the values that we have used. The idle ratio is normally 0.5. For generality of results, we have varied the Peak Power from 2 - 5 Watts and also the idle ratio from 0.2 - 0.5. We have performed these experiments on a Pentium 4 workstation with 512 MB RAM. However, a PDA is much slower than this. So, we have computed and used a "Slowdown" ratio based on timing two busy loops on the workstation and the handheld.

Table 1. Constants used in the Experiments

Constant	Value used
Transmission cost per byte (α)	444 nanoJoules/byte
PDA Peak Power (β)	2 Watts
PDA Idle Power (γ)	1 Watt
Slowdown ratio	10
Download rate	11Mbps

4.2. Applications

We have considered two Java applications. The first is an application that finds a sub-matrix within a huge matrix. Image processing applications do perform similar computations. For example, an image application that looks for a small image within a larger image is essentially trying to find a small pixel sub-matrix within a large pixel matrix. This application has six Java classes specific to it. Typical runs of this application create objects in the order of a few thousands. Each matrix is an object and is divided into many smaller matrix objects for the purposes of searching for the sub-matrix. The problem size is the number of initial matrices and the size of the initial matrices.

The second program is an application which finds strings within large text documents. Again, this is an important operation in word-processing applications which are commonly used in handhelds. This application also has six Java classes. Typical runs of this application produce a few thousands of objects. The problem size was the size of the text document and the number of such text documents.

4.3. Results

The first experiments we performed were to measure the relative performance of the FAST and OPTIMAL algorithms. Tables 2 and 3 show the time taken by each algorithm and the % of energy they save for the applications.

Table 2. Matrix Application

Number of Objects	Time Taken in secs.		Energy Savings %	
	OPTIMAL	FAST	OPTIMAL	FAST
3111	16.7	0.188	43.0	29.29
3111	16.7	0.188	44.08	29.29
3111	18.3	0.172	44.6	29.29
4443	95.1	0.133	43.6	32.22
6663	324	0.344	44.5	33.88

Table 3. Text Application

Number of Objects	Time Taken in secs.		Energy Savings %	
	OPTIMAL	FAST	OPTIMAL	FAST
523	1.62	0.003	45.29	37.84
1043	2.03	0.003	44.25	40.8
2043	8.94	0.06	42.1	38.06
3063	36.58	0.128	46.02	38.53
4563	41	0.172	47.5	43.06

Various problem sizes were considered. The numbers were generated using an idle ratio of 0.5. The idle ratio is the ratio of Idle to Peak Power. The following observations can be made. The OPTIMAL algorithm is much slower in comparison to the FAST algorithm, which lives up to its name. Considering energy savings, in the Text processing application, the FAST algorithm nearly performs as good as the OPTIMAL algorithm though this is not the case with the Matrix application. We have found for the Matrix application that the communication involved was the dominant factor. The FAST partitioner chooses different objects for migration when compared to the OPTIMAL one. The FAST algorithm does best when it can quickly find objects whose benefit is more than the cost, which it could not do for the MATRIX application. However, the Text processing application was an ideal input as the FAST algorithm has come very close to the OPTIMAL one.

Another interesting observation is that the % Energy savings is almost independent of problem size. This shows how well these applications scale and this validates the assumption of partitioning for smaller problem sizes and using the result for larger sizes.

We then measured Energy savings by varying the Peak Power and the Idle ratio separately. The graph in Figure 3 shows the % Energy savings as the Peak power of the client is varied from 2 to 5 Watts.

The graph in Figure 4 shows the % Energy savings as the idle ratio of the client is varied from 0.2 to 0.5. Notice that the plots of both the algorithms are decreasing. The reason is the following. As the idle ratio decreases, the clients spend less energy idling and hence the benefits of remote objects are larger. Hence, the % energy savings also increase. From these results we can conclude that partitioning is an effective technique for saving battery power of a mobile device across a wide range of parameters.

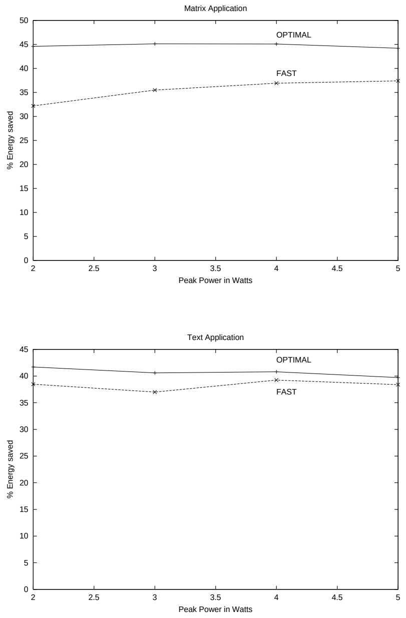

Figure 3. Power vs. Energy Saved, idle ratio=0.5

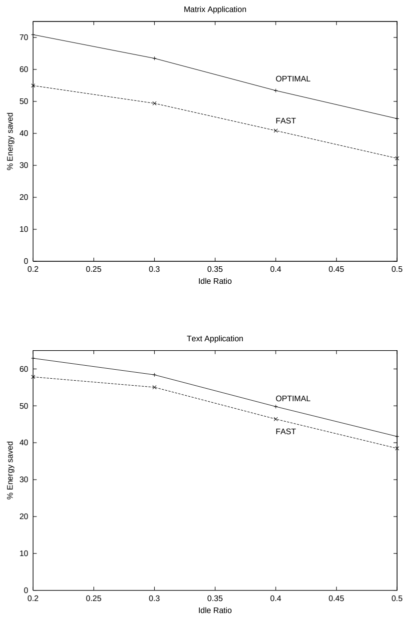

Figure 4. Idle ratio v/s Energy Saved, Peak Power = 2 Watts

5. Related Work

In [12] a task partitioning scheme is presented that distributes tasks between wireless handheld and server for distributed multimedia processing. An energy cost model is developed and a MIN-CUT algorithm is used to distribute the tasks. Though a MIN-CUT algorithm can be used with our approach, there is the problem of time and space complexity associated with it.

A compiler based partitioning scheme to statically divide a program's tasks into client and server tasks in order to save energy is discussed in [13]. A task unit considered is a procedure call and each call site to a function is a task. Then, depending on whether a call-site is assigned to the client or server, function call instances originating from this call-site are executed locally or remotely. This scheme restricts partitioning possibilities because all instances of a procedure invoked from the same call-site are forced to execute in the same side, client or server. We work at the level of an object and not a procedure and our scheme does not place any restrictions like the above on any of the objects. This makes our scheme general and allows us to explore all partitioning possibilities.

Another compiler managed remote task execution scheme for power management is discussed in [10]. Program tasks that can be executed remotely are identified statically and two versions of the application are created for the client and the server. Their scheme is also able to handle connection failures between the client and the server by using checkpointing techniques. Their work differs from ours in the following way. The process by which [10] identifies program tasks that can be executed remotely is not automatic and is application dependent and is also done statically. We automatically identify objects to be executed remotely and the allocation of objects to client and server takes place during run-time.

Other partitioning based systems are Coign [7] and J-Orchestra [18]. Coign is a partitioning system that partitions Microsoft COM applications across a network to minimize execution delay due to network communication. Coign applies a graph-cutting algorithm to partition the application. Their scheme does not target energy conservation. J-Orchestra is an automatic Java application partitioning system. Given, the object placements, the partitioner automatically rewrites the byte code placing Java RMI calls that will achieve the desired partitioning. It does not address the applications of partitioning such as conserving energy.

A system level task partitioning scheme has been proposed in [16]. A remote processing framework that can execute tasks remotely and save power has been developed by

Rudenko et al. [16]. The remote tasks are identified at the system level and process migration is done to execute the task remotely. This technique is complementary to ours in that once we have done the partitioning, during the program run, their system-level partitioning could be used by the OS to migrate some local tasks to the remote server.

A compilation service to reduce energy consumption of Java programs is proposed in [15]. A detailed study on the relative merits and demerits of moving compilation to a server, with regards to energy, is presented. They do not deal with partitioning of a Java application.

6. Conclusions

As embedded devices are battery powered, it is important for applications that run on them to be energy conscious. In the presence of a remote server, there is an opportunity to run program parts remotely. However, such a technique is limited by the transmission energy needed to transfer program parts. In this paper, we have presented a fast partitioning algorithm for Java programs that partitions at the object level. Experiments have shown that energy can be reduced from 29% to 43%. Moreover energy savings are observed across a wide range of parameters. Comparing with the algorithm that gives optimal energy savings shows that the fast algorithm performs very well.

References

[1] Compaq Computer Corporation. WL110 product specifications.

[2] T.H. Cormen, C.E. Leiserson and R.L. Rivest, "Introduction to Algorithms," Prentice-Hall.

[3] K. Farkas, J. Flinn, G. Back, D. Grunwald, and J.M. Anderson, "Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine," *ACM International Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS), 2000.

[4] R.E. Gomory and T.C. Hu, "Multi-Terminal Network Flows," *SIAM Journal of Applied Math*, 1962.

[5] W.R. Hamburgen, D.A. Wallach, M.A. Viredaz, L.S. Brakmo, C.A. Waldspurger, J.F. Bartlett, T.Mann, and K.I. Farkas, "Itsy: Stretching the bounds of Mobile Computing," *IEEE Computer*, 2001.

[6] K. Hogstedt, D. Kimelman, V.T. Rajan, T.Roth, and M. Wegman, "Graph cutting Algorithms for Distributed Applications Partitioning," *Workshop on Mathematical Modeling and Analysis*, 2000.

[7] G.C. Hunt and M.L. Scott, "The Coign Automatic Distributed Partitioning System," *Symposium on Operating System Design and Implementation* (OSDI), 1999.

[8] M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and W. Ye, "Influence of Compiler Optimizations on System Power," *Design Automation Conference* (DAC), 2000.

[9] H.S. Kim, M.J. Irwin, N. Vijaykrishnan, and M. Kandemir, "Effect of Compiler Optimizations on Memory Energy," *IEEE Workshop on Signal Processing Systems* (SiPS), 2000.

[10] U. Kremer, J. Hicks, and J.M. Rehg, "Compiler-Directed Remote Task Execution for Power Management," *Workshop on Compilers and Operating Systems for Low Power* (COLP), 2000.

[11] T. Kunz and S. Omar, "A Mobile code Toolkit for Adaptive Mobile Applications," *IEEE Workshop on Mobile Computing Systems and Applications*, 2000.

[12] Z. Li, C. Wang. and R. Xu, "Task Allocation for Distributed Multimedia processing on Wirelessly Networked Handheld Devices," *International Parallel and Distributed Processing Symposium* (IPDPS), Fort Lauderdale, Florida, April 2002.

[13] Z. Li, C. Wang. and R. Xu, "Computation Offloading to Save Energy on Handheld Devices: A Partition Scheme," *ACM Conference on Compilers, Architecture and Synthesis of Embedded Systems* (CASES), Atlanta, Georgia, 2001.

[14] P. Niemeyer and Joshua Peck, "Exploring Java," Publishers - O'REILLY.

[15] J. Palm, J. Eliot, and B.Moss, "When to Use a Compilation Service ?," *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES), 2002.

[16] A. Rudenko, P. Reiher, G.J. Popek, and G.H. Kuenning, "The Remote Processing Framework for Portable Computer Power Saving," *ACM Symposium on Applied Computing* (SAC), 1999.

[17] M. Satyanarayanan, "Pervasive Computing: Vision and Challenges," *IEEE Personal Communications*, 2001.

[18] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning," *European Conference on Object-Oriented Programming* (ECOOP), 2002.

[19] V. Tiwari, S. Malik, and A. Wolfe, "Compilation Techniques for low energy," *IEEE Symposium on Low Power Electronics* (ISLPED), 1994.

[20] M. Valluri and L. John, "Is Compiling for Performance == Compiling for Power," *Workshop on Interaction between Compilers and Computer Architectures* (INTERACT-5), 2001.

[21] M.A. Viredaz, "The Itsy Pocket Computer version 1.5: User's Manual," Technical Report, Western Research Lab, Compaq Computer Corporation, 1998.