Computation Offloading to Save Energy on Handheld Devices: A Partition Scheme *

Zhiyuan Li†
Department of Computer Sciences
Purdue University
West Lafayette, Indiana
li@cs.purdue.edu

Cheng Wang
Department of Computer Sciences
Purdue University
West Lafayette, Indiana
wangc@cs.purdue.edu

Rong Xu
Department of Computer Sciences
Purdue University
West Lafayette, Indiana
xur@cs.purdue.edu

ABSTRACT

We consider handheld computing devices which are connected to a server (or a powerful desktop machine) via a wireless LAN. On such devices, it is often possible to save the energy on the handheld by offloading its computation to the server. In this work, based on profiling information on computation time and data sharing at the level of procedure calls, we construct a *cost graph* for a given application program. We then apply a partition scheme to statically divide the program into server tasks and client tasks such that the energy consumed by the program is minimized. Experiments are performed on a suite of multimedia benchmarks. Results show considerable energy saving for several programs through offloading.

1. INTRODUCTION

The limited battery life has made power management one of the critical issues for handheld computing devices. Several known power-conservation techniques include turning off the handheld computing device screen when it is not needed [1], optimizing I/O [3], slowing down the CPU [4], among others. In this paper, we consider handheld computing devices which are connected to a server (or a powerful desktop machine) via a wireless LAN. Such wireless connectivity makes the immense information and abundant services on the network accessible to the handheld devices. Moreover, it becomes possible to save the energy on such devices by offloading the computation to the server [5].

Although the idea of offloading computation on mobile computers has been explored previously [8, 7, 5], most of the existing works target laptop computers instead of handhelds. According to Farkas et al. [2], the energy behavior is quite different on these two kinds of systems. Moreover, most existing works either use desk-top applications which are not quite suitable for current handhelds [8] or use synthetic workloads [7] instead of real programs. Existing works also primarily consider migrating an entire process to the server. Recently, one sees some interests in exploring opportunities to offload tasks inside a single process from a handheld to a server [5]. However, such task-level offloading has been conducted in a rather ad-hoc way. For example, the highest level tasks may be tried as the offloading candidates one at time, to see whether energy may be saved [5]. Experience with programs suitable for handhelds is still quite limited. More thorough schemes are needed in order to fully explore computation offloading, partially or completely, within a single process. More experiences with realistic programs for handhelds are also needed.

In this work, we develop a program partition scheme which determines whether and how to offload the computation of a given program. While the paper focuses on uniprocess programs, we expect the main idea to be applicable to the case of multiple processes. In our current scheme, offloading units are defined at the level of procedure calls. Each unique call site to a function (or a procedure) is identified as a task. We first collect profiling information on computation time and data sharing at the level of procedure calls. Based on that, we construct a *cost graph* for the given program and apply a task mapping algorithm to statically divide the program into server tasks and client tasks such that the energy consumed by the program is minimized. We perform experiments by running a well-known suite of multimedia benchmarks, namely Mediabench, on a Compaq iPAQ handheld device connected to a desktop machine via WaveLan.

The remaining sections of the paper are in the following order. Section 2 briefly surveys related work. Section 3 and section 4 presents our overall scheme and the task mapping algorithm. Section 5 reports experimental results which are collected from a Compaq iPAQ handheld device. We conclude in Section 6.

*This work is sponsored in part by National Science Foundation through grants ACI/ITR-0082834, CCR-9975309, and MIP-9610379, by Indiana 21st Century Fund, by Purdue Research Foundation, and by a donation from Sun Microsystems, Inc.

†The author names are listed in alphabetical order.

CASES'01, November 16-17, 2001, Atlanta, Georgia, USA.

2. RELATED WORK

Kremer, Hicks and Rehg [5] recently propose a compilation framework for power management on handheld computing devices. They discuss the potential benefits of application specific power management through remote task execution and present a compilation strategy to perform the analysis and transformation. Their work considers the procedure calls in the *main* routine as the candidates for remote execution. Their method evaluates the profitability of remote execution for each individual task separately. This may not be optimal, because several tasks may need to be offloaded together in order to benefit. Otherwise, the communication cost may be too heavy. Their model uses computation time to approximate energy consumption, which may not be accurate. Their experience with multimedia programs is limited to a single application.

Rudenko et al. [8] describe experiments using laptops with an early version of WaveLan radio devices. They run three desktop programs, namely the LaTex text formatting software, the GCC compiler, and a Gaussian Elimination program. They conclude that, on a noisy wireless environment, most often offloading does not save energy. On a noiseless environment, however, they show significant energy saving by offloading in some cases. Our work is based on handheld devices, running multimedia codes and a game, and we observe quite positive results. In their work, they consider offloading the whole program. Othman and Hailes [7] use simulation to show that battery life can be extended through migration at the process level. They use synthetic workload in their study.

Networked portable terminals, such as the InfoPad [6], have also been used in wireless environments. Such systems have only a network I/O device with no computational power, relying on network servers to run major processes. The lack of local computing power makes such a system run slower and consumes more energy for certain important multimedia applications, e.g. JPEG, due to the large amount of communication required.

3. A PARTITION SCHEME

In this section, we first present our execution model and cost model. We then describe how we use the cost graph to model the program behavior and cost. Lastly, we present our task mapping algorithm.

3.1 Execution Model

To explore opportunities for task-level offloading, one needs to identify, within a given program, server tasks and client tasks. Statically, under our execution model, a task corresponds to a procedure (or a function) call site. Dynamically, a task is a single invocation of the corresponding procedure. The set of server tasks may be empty, in which case the entire program runs on the handheld device. In contrast, the set of client tasks will never be empty, because realistic programs always have certain amount of I/O that must be performed on the handheld device. Nonetheless, if simple I/O tasks are the only client tasks, then one could actually run the entire program on the server after configuring the handheld device as a graphics terminal, equipped with a few other (e.g. the sound) I/O subdevices. This practice, however, does not change the fact that data need to be transmitted between the server and the handheld device and the analysis scheme must account for such costs.

It is important to note that *we do not seek to execute a client task and a server task simultaneously.* This is due to the significant computation speed gap between the two. One sees no realistic gain by parallel execution in this manner. On the other hand, potential advantage may be gained by overlapping client computation with communication. However, we presently do not take such an advantage.

In the absence of simultaneous tasks, program execution simply follows the original sequential control flow. Function invocation serves as a natural task scheduling mechanism at the run time. If a server task invokes a client task, or vice versa, a remote procedure call is performed through message passing.

Four kinds of messages are used in our model. The `task_start` message is used to send the identity of the scheduled task to the opposite side (either the server or the handheld device). Upon receiving this message, the receiver starts the execution of the identified task. Meanwhile, the sender blocks its current task. The `task_end` message indicates the termination of a previously scheduled remote task. Upon its reception, the previously blocked task resumes the execution. The `data_send` message carries a piece of data to its recipient, and the `data_request` message requests data from its recipient. Figure 1 shows an example program. A restructuring compiler, after its decision on the task mapping, creates two copies of the code, the server code and the client code, which may take the forms shown in Figure 3 and Figure 2, respectively. Statically, the four call sites (labeled by S1 through S4) are identified as four tasks. In addition, we always label main() as task S0. In the transformed codes, S0, S1 and S3 become client tasks, whereas S2 and S4 become server tasks.

Next, we discuss the data sharing issue in our execution model.

```
int x, y, z;

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

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 1: Example Program

```
int x, y, z;
main() {
  y = 1;
  scanf("%d", &x);
  if(x < 10)
    func1();
  printf("%d", z);
  if(x > 0)
    send task_start(func2());
  while(true) {
    receive message
    if(data_send received )
      /* data z */
      update z
    else if(data_request received )
      /* data y */
      send y
    else
      /* task_end */
      break;
  }
  printf("%d", z);
  send task_end
}

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

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

Figure 2: Client Program

```
int x, y, z;
main() {
  while(true) {
    receive message
    if(task_start received) {
      /* task start for func2 */
      func2();
      send task_end
    }
    else
      /* task_end */
      break;
  }
}

func2() {
  int i, a;

  request y;
  a = y;
  for(i = 0; i < 100; i++)
    a = func3(a);
  z = a;
  send z;
}

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

Figure 3: Server Program

3.2 Data Sharing

After the partitioning, all the tasks mapped to the same host can share the program state as they do in the original program. The issue of coherence arises, however, when one deals with data shared between two tasks which end up in different hosts. It seems to us, at this point, that a paging-based, software distributed shared memory (DSM) system, may introduce unacceptable overhead in terms of computation time and energy on the handheld device. At present, we maintain the program state coherence in a conservative way. A piece of data can be shared by both hosts only if the program transformations and message passing can guarantee the correct dependence relations incurred on that data.

Two methods of data communication can be used, namely the *push* method and the *pull* method. The push method sends a piece of modified data to the opposite cluster after its modification. The pull method lets the intended receiver make the request for the modified data before the data use. As shown in the example of Figure 1, neither method is guaranteed to be superior. Function *main* modifies data *y*. Functions *func1*, *func2* use data *y*. The push method always transfers *y* once, no matter how the program will be partitioned. The pull method may transfer *y* once, twice, or never, depending on the task mapping and branch condition. In our partition scheme, we consider both methods when estimating the average data communication cost and then select the better communication method.

Next, we present a cost model for our analysis and describe the cost graph used for mapping the tasks to the server or the handheld.

3.3 The Cost Graph

In previous work on task mapping for distributed computing, data sharing among tasks are always via distinct messages. No two tasks may share the same received message even if they are mapped to the same cluster. This is in contrast to our execution model which deals with ordinary programs with no tasks distributed initially. If a data item is shared by several tasks in the original program and those tasks are mapped to the same execution host, then only a single message is needed to send that data item to those tasks.

We use a part of the *PGP* program from the Mediabench suite as an example to show the extra communication cost that may be charged if messages are sent redundantly. Figure 4 shows the data communication with message sharing, where a circle represents a task and a rectangle represents a data item. Task *readkeypacket* is executed on the handheld device while the other shown tasks run on the server. Task *readkeypacket* writes the data *p*, *q* and *d* with data communication cost *e1*, *e2* and e3 respectively. The other tasks use these data. The total data communication cost is $e1 + e2 + e3$. Without sharing, the total data communication cost will be $2 * e1 + 2 * e2 + 3 * e3$, as shown in Figure 5.

Based on the above discussions, we devise a cost graph which makes the message-sharing patterns explicit. To characterize the costs associated with individual nodes and edges, we make use of the following parameters.

- b_s: the send bandwidth of the wireless network.
- b_r: the receive bandwidth of the wireless network.
- p_i: the mean power consumption rate when the hand-

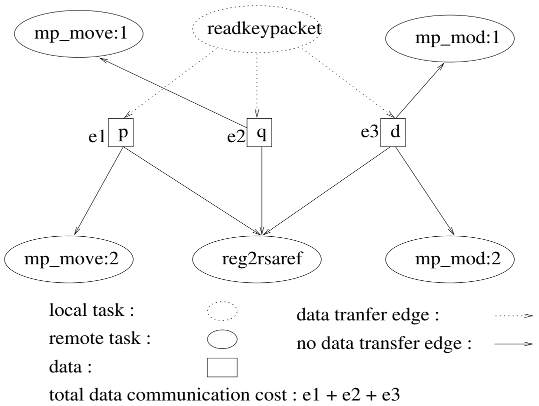

Figure 4: data communication for PGP with message sharing

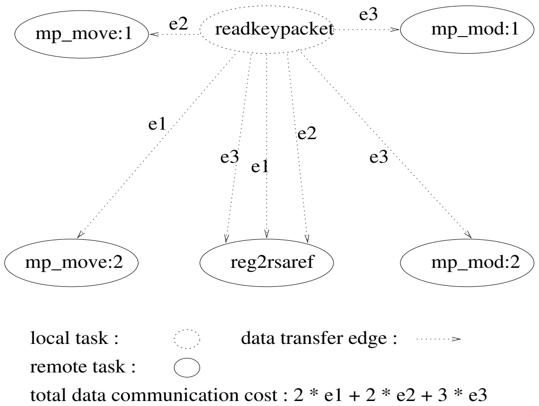

Figure 5: data communication for PGP without message sharing

held device is idle.

- p_c: the mean power consumption rate when the hand-held device performs computation.
- p_s: the mean power consumption rate when the hand-held device sends data.
- p_r: the mean power consumption rate when the hand-held device receives data

We integrate both the computation cost and the data communication cost into the cost graph $G = (V, E)$, where the node set $V = T \cup D$ is defined as follows.

- Each node $t \in T$ represents a task in the program. We use e_c to denote the energy consumed by executing t as a client task (i.e. on the hand-held device), not counting the energy consumed by tasks called by t. Similarly, we use e_s to denote the energy consumed by the idling hand-held device if t is executed as a server task. We attach the value pair (e_c, e_s) to t. To get their values, we use profiling to first measure $t_c(t)$ and $t_s(t)$, the execution time for t as a client task and as a server task respectively, not counting the tasks called by t. We then have

$$e_c(t) = t_c(t)p_c$$
$$e_s(t) = t_s(t)p_i$$

- Each node $d \in D$ represents a modified data item which is transferred between different tasks, either by the push method or the pull method. We use e_l and e_r to represent the mean energy consumption for a single instance of transferring d from client to server, and vice versa, respectively. Each of e_l and e_r should include both the energy consumed by the wireless network device and that by the idling cpu blocked by the transfer. Let $s(d)$ be the size of d, we have

$$e_l(d) = \frac{s(d)}{b_s} p_s$$
$$e_r(d) = \frac{s(d)}{b_r} p_r$$

The set $E = M \cup U$ of directed edges is defined as follows.

- Each edge $m = (t, d) \in M$ indicates that task t modifies d. We attach a value f to each edge m, which represents the mean number of times d is modified within an instance of t, considering various branching conditions. This is the estimated number of data transfers for d, using the push method (*c.f.* Section 3.2).
- Each edge $u = (d, t) \in U$ indicates that task t uses the modified value of d. We attach a value g to each edge u, which represents the mean number of times d is used in an instance of t. This is the estimated number of data transfers for d, using the pull method.

For the program in Figure 1, let us assume (e_c, e_s) to be (2, 1) for each of *S0*, *S1*, *S3* and *S4* and to be (200, 100) for *S2*. Further assume (e_l, e_r) to be (15, 10) for each of the data items. We get the cost graph shown in Figure 6.

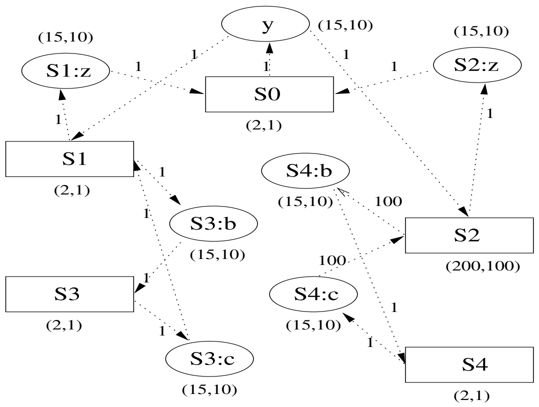

Figure 6: Cost Graph for Example Program

We assume $n(t), t \in T$ as the total number of instances for task t which is invariant to the mapping and define variable $X(t), t \in T$ as the number of client task instances for task t. Subject to

$$\begin{cases} X(t) = n(t) & t \textit{ must be client task} \\ X(t) = 0 & t \textit{ must be server task} \\ 0 \le X(t) \le n(t) & \textit{others} \end{cases} \quad (1)$$

the total energy consumption for computation should be

$$\sum_{t \in T} \Big(e_c(t)X(t) + e_s(t)\big(n(t) - X(t)\big)\Big). \quad (2)$$

We define variables $Z_l(d), Z_r(d), d \in D$ as the number of data transfers for d from client to server, and vice versa, respectively. Each of Z_l and Z_r can be computed by two different formulas, depending on push or pull:

$$\Big(Z_l(d) = X(t')f(m), m = (t', d) \in M\Big) \ or$$
$$\Big(Z_l(d) = \sum_{u=(d,t'')\in U} (n(t'') - X(t''))g(u)\Big) \quad (3)$$
$$\Big(Z_r(d) = \big(n(t') - X(t')\big)f(m), m = (t', d) \in M\Big) \ or$$
$$\Big(Z_r(d) = \sum_{u=(d,t'')\in U} X(t'')g(u)\Big) \quad (4)$$

Thus the data communication cost is

$$\sum_{d \in D} \big(Z_l(d)e_l(d) + Z_r(d)e_r(d)\big) \quad (5)$$

Based on the above, the overall optimization problem is to find X, Z that satisfy equations (1) (3) and (4) such that the sum of (2) and (5) is minimized.

4. TASK MAPPING ALGORITHM

In this section, we first present an exact branch-and-bound algorithm for the problem defined in the previous section. We then describe a heuristic algorithm which significantly prunes the search space to get an approximate solution.

4.1 Branch and Bound

The problem defined by equations (1) - (5) has the properties which allow us to use the branch-and-bound algorithm: No matter what the data communication method of Z is, its cost can be substituted by a linear expression of X in the objective function. So there exists an optimal solution in which X is maximized or minimized. We can make use of this property by branching X with the value of either 0 or n, meaning that all the instances for the task corresponding to D are mapped to the remote server, or to the handheld device, respectively. Since all the X values in the cost model are bounded, during the node traversal of the solution tree, it is easy to get the lower bound cost for all the subtree before the branching, which will cut the unnecessary branches.

For the example in Figure 1, if all the *if* conditions in the program are *true*, the branch-and-bound algorithm can find that the optimal solution will map S0, S1 and S3 to the handheld device and map S2 and S4 to the server. This partition is illustrated in Figure 2 and Figure 3

4.2 A Pruning Heuristic

The worst case complexity of Branch and Bound grows exponentially in the size of the cost graph, which can be prohibitively costly if we work on the full graph for real applications. Instead, taking advantage of the fact that one needs to focus only on the heavy tasks and heavy data items, we use a pruning heuristic to shorten the solution search time.

The overall heuristic method consists of two phases. In the first phase, we sort all the tasks t by the weight of

$$\begin{aligned} &(e_s(t) + e_c(t)) * n(t) + \\ &\sum_{m=(t,d)\in M} \Big(n(t) * \big(e_r(d) + e_l(d)\big) * f(m)\Big) + \\ &\sum_{u=(d,t)\in U} \Big(n(t) * \big(e_r(d) + e_l(d)\big) * g(u)\Big) \end{aligned}$$

We then select the subset of tasks with the heaviest weights to build a subgraph. The branch-and-bound algorithm is applied to that subgraph. In the second phase, the remaining tasks are put back to the partitioned graph one by one, in the descending order of weight. No further branching is done and the mapping decision for each task depends on the lowest cost according to the current best mapping.

The overall algorithm is presented as procedure *Task_Map* in Figure 7, which calls the recursive procedure *Map* to greedily map each task. Because all the tasks are initially sorted in the ascending order of weights, the task with a heavier weight will be handled at a deeper recursive level. The procedure *prune_level* is used to control the pruning. The *lowbound_cost* is the estimated low bound based on the mapping at upper levels and it is used for branch cutting. The *current_cost* is calculated based on the mapping of upper levels and the known best mapping for lower levels.

5. EXPERIMENTAL RESULTS

We are currently implementing our algorithm in *gcc*. To obtain a preliminary evaluation of the effectiveness of our method, we manually apply our algorithm to a suite of *Mediabench* programs and *gnugo*, which is a *go* game. In this section, we describe our experiments and present the results.

```
Procedure Task_Map
Input: (1) costgraph (2) prune_level
Output: mapping for each task
Procedure:
   sort task by weight in ascending order
   min_cost = ∞
   Map(0)      /* map task 0 */
   return mapping
end
Procedure Map(t)
   compute lowbound_cost
   if lowbound_cost < min_cost
      if t is last task
         min_cost = lowbound_cost
         mapping = current_mapping
      else
         X(t) = C(t)      /* map to client */
         Map(t + 1)      /* map next task */
         if t can run on server
            X(t) = 0      /* map to server */
            if t < prune_level      /* prune */
               compute current_cost
               if current_cost < min_cost
                  min_cost = current_cost
                  mapping = current_mapping
               endif
            else
               Map(t + 1)      /* map next task */
            endif
         endif
      endif
   endif
end
```

Figure 7: Pruned Branch and Bound Algorithm

5.1 Experiments setup

The handheld computer that we use is Compaq's iPAQ 3650 which has a 206MHZ Intel StrongArm SA1110 processor and 32MB RAM. The back-end server is a PIII 866MHZ Dell Dimension XPS. We run Linux on both machines. The wireless connection is through a Lucent Orinoco(WaveLan) Golden 11Mbps PCMCIA card inserted into a PCMCIA expansion pack for the iPAQ. During the experiments, we maintain the ordinary wireless LAN environment in which many wireless devices use the same access points for communication at the same time. Throughout the experiment, we set the screen saver and the screen back-light off. We measure the total elapsed time of each program and the electrical current drawn on the handheld device.

5.1.1 Test Programs

Table 1 lists the programs used in our experiments, their descriptions and their input parameters, including the input files we use, which can be found on the *Mediabench* web-site. A program may have rather different execution paths to provide different functionalities, we perform tests on those different paths, using different profile information to generate the cost graph. As a result, it may have different versions of transformed code.

We were unable to locate the full source code of *NIST SPHERE + RASTA*, which therefore is omitted in the experiments. We have not yet performed tests on *Mesa*. For the *gnugo* program, we use the input parameters "`-b 9 -r 2`", where "`-b 9`" means playing 9 steps in benchmark mode and "`-r 2`" means setting the random seeds as 2 which makes it easier for us to verify result.

5.1.2 Physical measurement

We connect an HP 3458A high precision digital multi-meter to measure the actual current drawn by the hand-held computer during the program execution. In order to get a reliable and accurate reading, we disconnect the batteries from both the iPAQ and the extension pack and we use an external 5V DC power supply instead. Further, we make use of the build-in trigger mechanism in the multi-meter. After the test program starts running, the iPAQ triggers the meter to do the reading in a high frequency. The trigger stops as the test program finishes. According to out real measurement, the overhead associated with the triggering interrupts is less than 0.5% and the readings are consistent over repeated runs.

We measure the electrical current drawn in different running modes. The readings are shown in Table 2. In this table, the first column "iPAQ" lists the functioning mode of iPAQ: *idle* means iPAQ waiting for interrupts; *busy* means iPAQ performing intensive computation. In the experiment, we observe that though iPAQ is busy, different operations may have different electrical current reading. We choose the values based on the average of a variety of programs. The *Send/Recv* column indicates whether there is intensive sending or receiving. During sending or receiving, the computer is neither idle nor computation-intensive. We just put '-' in the iPAQ column. The last column lists the electrical current drawn from the external DC power. Note that all the numbers we get here are with the screen off (When using the medium back-light, the current will increase about 140mA).

Table 2: Power parameters

iPAQ	WaveLan	Send/Recv	Current(A)
idle	no	no	0.110
busy	no	no	0.230
idle	yes	no	0.330
busy	yes	no	0.480
-	yes	send	0.440
-	yes	recv	0.420

In the experiments, we assume that the WaveLan is to be always connected, mimicking the realistic situations in which the users wish to maintain the continuous readiness for services such as email and instant messaging. Therefore, we use the electrical current reading in the second and the third row of the table. For the 5V voltage, we get $p_i = 1.5w$, $p_c = 2.15w$, $p_s = 2.05w$, $p_r = 1.95w$, approximately. These parameters are used when we apply our partition algorithm.

5.2 Energy Data

By applying our algorithm to the programs listed in Table 1, we find that 13 of these 18 programs can get better energy usage by offloading. For the remaining 5 programs, namely, *cjpeg*, *djpeg*, *pegwit/decrypt*, *rawcaudio* and *rawdaudio*, our algorithm finds no energy benefit to offload: the most energy-efficient running mode for these programs is to run completely locally. The reason is either due to insufficient computation which can be offloaded or due to the high communication volume required to offload the computation.

Table 3 is the experimental result of the benchmark programs, where the "*w/o expansion pack*" label means processing on the handheld device without connecting the wireless card, and the "*w/ expansion pack*" label means processing on the handheld device while WaveLan is connected. "*algorithm*" means the code generated by our algorithm. The number immediately following a program name is the num-

Table 1: Test programs

Program Name	Description	Input Parameters
toast	GSM in Mediabench, GSM voice transcoding	toast -fpl clinton.pcm
untoast	GSM in Mediabench, GSM voice transcoding	untoast -fpl clinton.pcm
encode	G.721 in Mediabench, CCITT voice compression	encode -4 -l <clinton.pcm >out.g721
decode	G.721 in Mediabench, CCITT voice decompression	decode -4 -l <clinton.pcm.g721 >out.pcm
epic	EPIC in Mediabench, image compression	epic test_image -b 25
unepic	EPIC in Mediabench, image decompression	unepic test_image.E test_image.out
pgp/encrypt	PGP in Mediabench, encryption	pgp -fes Bill -zbillms -u Bill <pgptest.plain >pgptext.pgp
pgp/decrypt	PGP in Mediabench, decryption	pgp -fdb -zbillms <pgptext.pgp >pgptest.dec
rawcaudio	ADPCM in Mediabench, speech compression	rawcaduio <clintonpcm >out.adpcm
rawdaudio	ADPCM in Mediabench, speech decompression	rawdaduio <clinton.adpcm >out.pcm
ghostscript	Postscript interpreter	gs -sDEVICE=ppm -sOutputFile=test.ppm -dNOPAUSE -q – tiger.ps
cjpeg	IJG JPEG in Mediabench,image compression	cjpeg -dct int -progressive -opt -outfile out.jpeg nasa.ppm
djpeg	IJG JPEG in Mediabench,image decompression	djpeg -ppm -outfile out.ppm nasa.jpeg
pegwit/encrypt	PEGWIT in Mediabench, encryption	pegwit -e my.pub pgptest.plain pegwit.enc <encryption_junk
pegwit/decrypt	PEGWIT in Mediabench, decryption	pegwit -d pegwit.enc pegwit.dec < my.sec
mpeg2encode	MPEG in Mediabench, mpeg encoding	mpeg2encode test.par out.m2v
mpeg2decode	MPEG in Mediabench, mpeg decoding	mpeg2decode -b mei16v2.m2v -r -f -o0 rec%d
gnugo	gnugo-2.7.242, a Go program	gnugo -b 9 -r 2

ber of repeated runs to get stable results. Different versions of the same program always repeat the same number of runs. The time column is the total execution time for the repeated runs and the current column is the average current drawn during the execution of the programs.

Figure 8 compares the running time of different programs. Since the running time of different programs varies significantly, we normalize the running time using the handheld with WaveLan on as the base.

Figure 9 shows the average electrical current drawn during the execution of the program. For all programs that can benefit from offloading, the offloading has lower current drawn than local computation with WaveLan connection.

Figure 10 plots the energy consumed by each program. The energy of processing on handheld is again used as the base for normalization. We compute the energy consumption by the whole system during the execution by the simple equation:

$$energy = voltage * current\_drawn * elapsed\_time$$

The data shows that to save the energy, for 13 of these 18 test programs, our algorithm beats the default.

We should point out that the reason for *EPIC*, *MPEG2* and *gnugo* to save energy so significantly is because StrongArm does not have a floating point unit. It uses software floating point library which is extremely slow. We also find that after applying our algorithm to the programs listed in Table 1, nearly all computation gets offloaded to the server for those programs that can benefit from offloading.

The *"w/o expansion "* data show that, if the user is willing to isolate the handheld device from the network world during a program's execution, then the energy may be saved considerably. Note that this data column does not count the energy consumed to make the system calls necessary to switch on and off the wireless connection, before and after the program execution. The offloading technique, on the other hand, may also benefit from temporarily switching off the wireless connection in a similar way, although it needs to predict when to switch the wireless back on.

6. CONCLUSION

In this paper, we have presented a partition scheme to offload computational tasks on handheld devices. Our experimental results show that, even under an ordinary, uncontrolled, wireless LAN environment, the scheme can result in significant energy-saving for half of the multimedia benchmark programs we tested. We plan to conduct more extensive experiments after the complete scheme is implemented.

7. ACKNOWLEDGMENT

The authors thank Peifeng Ni for collecting part of the experimental data , Dr. David Yau for his helpful discussions and the anonymous referees for comments on the earlier draft of this paper.

8. REFERENCES

[1] J. W. Davis. Power benchmark strategy for systems employing power management. *IEEE International Symposium on Electronics and Environment*, 1993.

[2] K. I. 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*, 2000.

[3] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. *Proceedings of the First International Conference on Mobile Computing and Networking, MobiCom'95*, pages 13–25, November 1995.

[4] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. *Proceedings of the second annual ACM International Conference on Mobile Computing and Networking*, November 1996.

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

[6] S. Narayaswamy and et al. Application and network support for infopad. *IEEE personal Communications*, 3(2):4–17, April 1996.

[7] M. Othman and S. Hailes. Power conservation strategy for mobile computers using load sharing. *Mobile*

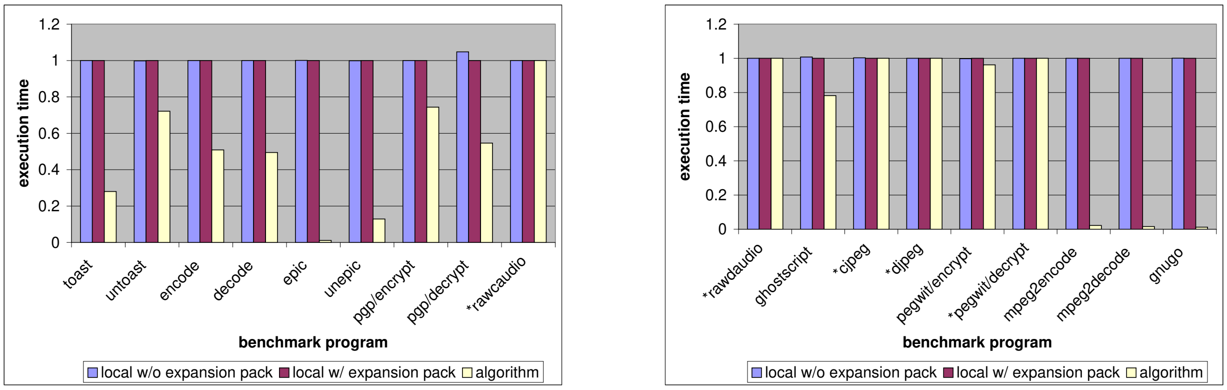

Figure 8: normalized running time on handheld device
(*program: algorithm decides to run locallly)

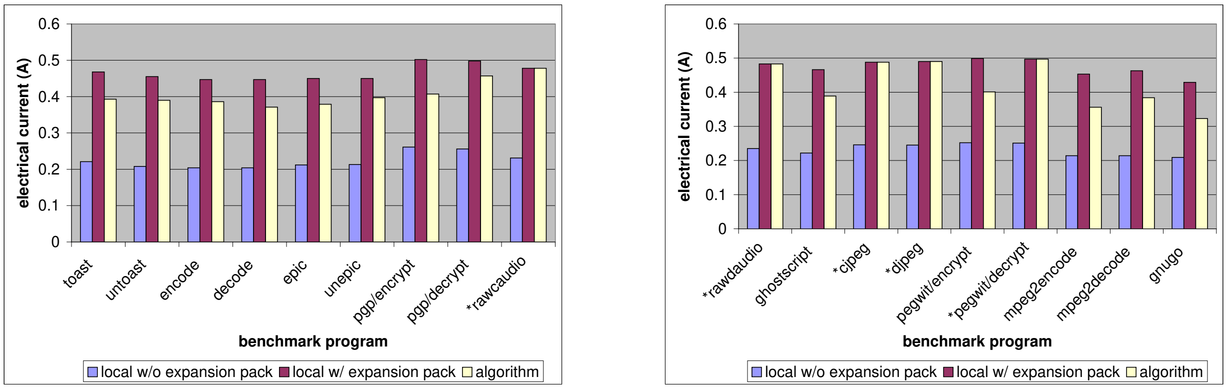

Figure 9: current drawn on handheld device
(*program: algorithm decides to run locallly)

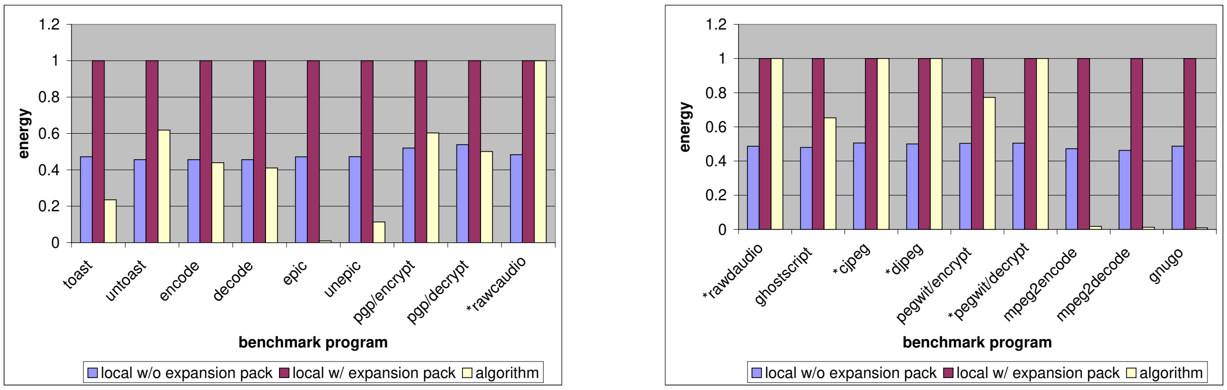

Figure 10: normalized energy consumption on the handheld device
(*program: algorithm decides to run locallly)

Table 3: Energy measurement

Program Name	w/o expansion pack		w/ expansion pack		algorithm	
	time (s)	current (A)	time (s)	current (A)	time (s)	current (A)
toast(50)	136	0.221	136	0.468	38.1	0.393
untoast(50)	52.0	0.208	52.1	0.455	37.6	0.390
encode(50)	102	0.204	102	0.447	51.9	0.386
decode(50)	189	0.204	189	0.447	93.4	0.371
epic(50)	1446	0.212	1445	0.450	15.9	0.379
unepic(50)	86.1	0.213	86.2	0.450	11.1	0.397
pgp/encrypt(100)	66.8	0.261	66.8	0.502	49.7	0.407
pgp/decrypt(100)	52.4	0.256	50.0	0.498	27.3	0.457
rawcaudio(500)	39.0	0.231	39.0	0.478	39.0	0.478
rawdaudio(500)	37.8	0.235	37.8	0.483	37.8	0.483
ghostscript(5)	67.4	0.222	66.9	0.466	52.3	0.389
cjpeg(200)	29.8	0.246	29.7	0.488	29.7	0.488
djpeg(200)	13.4	0.245	13.4	0.490	13.4	0.490
pegwit/encrypt(100)	44.2	0.252	44.3	0.499	42.6	0.401
pegwit/decrypt(100)	24.9	0.251	24.9	0.497	24.9	0.497
mpeg2encode(5)	593	0.214	593	0.453	13.1	0.356
mpeg2decode(5)	416	0.214	416	0.463	6.55	0.384
gnugo(3)	3685	0.209	3682	0.429	45.8	0.323

Computing and Communications Review, 2(1):19–26, January 1998.

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