Parametric Analysis for Adaptive Computation Offloading

Cheng Wang
Department of Computer Science
Purdue University
West Lafayette, IN 47907
wangc@cs.purdue.edu

Zhiyuan Li
Department of Computer Science
Purdue University
West Lafayette, IN 47907
li@cs.purdue.edu

ABSTRACT

Many programs can be invoked under different execution options, input parameters and data files. Such different execution contexts may lead to strikingly different execution instances. The optimal code generation may be sensitive to the execution instances. In this paper, we show how to use parametric program analysis to deal with this issue for the optimization problem of computation offloading.

Computation offloading has been shown to be an effective way to improve performance and energy saving on mobile devices. Optimal program partitioning for computation offloading depends on the tradeoff between the computation workload and the communication cost. The computation workload and communication requirement may change with different execution instances. Optimal decisions on program partitioning must be made at run time when sufficient information about workload and communication requirement becomes available.

Our cost analysis obtains program computation workload and communication cost expressed as functions of run-time parameters, and our parametric partitioning algorithm finds the optimal program partitioning corresponding to different ranges of run-time parameters. At run time, the transformed program self-schedules its tasks on either the mobile device or the server, based on the optimal program partitioning that corresponds to the current values of run-time parameters. Experimental results on an HP IPAQ handheld device show that different run-time parameters can lead to quite different program partitioning decisions.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization, Code Generation*

General Terms

Algorithms, Design, Experimentation, Measurement, Performance

PLDI'04, June 9–11, 2004, Washington, DC, USA.

Keywords

Program Partitioning, Program Analysis, Program Transformation, Computation Offloading, Program Profiling, Handheld Devices, Adaptive Optimization, Distributed System

1. INTRODUCTION

Many programs can be invoked under different execution options, input parameters and data files. Such different execution contexts may lead to strikingly different execution instances. The optimal code generation may be sensitive to the execution instances. In this paper, we show how to use parametric program analysis to deal with this issue for the optimization problem of computation offloading.

Computation offloading has been shown to be an effective way to improve performance and energy saving on mobile devices [10, 9, 11]. In a client-server distributed computing environment, the efficiency of an application program can be improved by careful partitioning of the program between the server and the client. Optimal program partitioning depends on the tradeoff between the computation workload and the communication cost.

1.1 The Problem

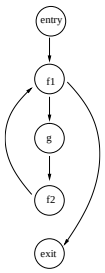
The computation workload and communication requirement may change with different execution instances. A program partitioning decision may work well under certain options and workloads, but may work poorly under other options and workloads. Correct decisions on program partitioning must be made at run time when sufficient information about workload and communication requirement becomes available.

Figure 1 (a) shows an example from an audio encoding application. Function f repeatedly gets a frame of the audio sample data from a certain audio device and puts them into the input buffer $inbuf$. It then calls certain encoder function g, which encodes each unit of data from $inbuf$ and stores it in $outbuf$. At last, function f writes the encoded data from $outbuf$ to the output device. Here g is a function pointer which, under different run-time options, can map to different encoders.

The program workload and communication requirement in the above example depend on three pieces of run-time information: the number of input frames x, the buffer size y and the amount of computation z, performed in g to encode a unit of data. Let the computation workload in each iteration of the inner most loop of f equals 1. The computation workload in function f will be $1 * y * 2 * x = 2xy$. The computation workload in function g is xyz. If we offload

```
f() {
  for(j = 0; j < x; j++) {
f1: for(i = 0; i < y; i++) {
      get audio sample p;
      inbuf[i] = p;
    }
    g();
f2: for(i = 0; i < y; i++) {
      q = outbuf[i];
      write q to output;
    }
  }
}
```

(a)



(b)

Figure 1: An Example

function g to the server, the mobile device will have only the computation workload from function f. However, we now need to send data *inbuf* from the client to the server and send data *outbuf* from the server back to the client. Suppose the data transfer startup cost is 6 and communication cost to transfer a unit of data is 1. We get the total data communication cost $(6 + 1 * y) * 2 * x = 12x + 2xy$. If we offload both functions f and g to the server, there will be no computation cost on the mobile device. However, we need to send data p from the client to the server and data q from the server back to the client, with the data communication cost $(6 + 1) * y * 2 * x = 14xy$. Table 1 lists the computation workload and the communication cost under different choices of computation offloading. From this table, We see that different values of x, y and z may change the optimal choice of computation offloading. The condition for f to be offloaded is: $14xy < xyz + 2xy$ && $14xy < 12x + 4xy$, i.e. $12 < z$ && $5y < 6$. The condition for g to be offloaded is: $12x + 4xy < xyz + 2xy \parallel 14xy < xyz + 2xy$, i.e. $12 + 2y < yz \parallel 12 < z$.

Table 1: Cost for Different Computation Offloading

offload	-	g	f, g
computation workload	xyz + 2xy	2xy	0
communication cost	0	12x + 2xy	14xy
total cost	xyz + 2xy	12x + 4xy	14xy

The problem to solve in this paper is how to find optimal program partitioning and formulate it in terms of the program execution parameters.

1.2 The Solution

In our computation offloading, we model the optimal program partitioning problem as a min-cut network flow problem with run-time parameters. Our *parametric partitioning algorithm* finds the optimal program partitioning corresponding to different ranges of run-time parameters. We then transform the program into a distributed program which, at run time self-schedules its computation tasks on either the mobile device or the server. The task scheduling is based on the optimal program partitioning that corresponds to the current values of run-time parameters. The program example in Figure 1 (a) is transformed into the form shown in Figure 2.

In order to derive the optimal partitioning, several issues must be addressed. Table 2 lists the key issues and their solutions. These are presented in the rest of the paper, which is organized as follows. We first present our program partitioning model in section 2. Based on our program partitioning model, we present a parametric cost analysis for program partitioning in section 3 and give a parametric partitioning algorithm in section 4. We discuss implementation issues in section 5 and report the experimental results in section 6. We discuss related work in section 7 and conclude in section 8.

```
In the caller to f():

...
if ((12 < z) && (5 * y < 6))
  call server_f();
else
  call client_f();
...
```

```
In f():

...
if ((12 + 2 * y < y * z)
  || (12 < z))
    call server_g();
else
    call client_g();
...
```

Figure 2: Program Transformation

Table 2: Main Contributions

Issues	Solutions
Redundant data transfer	data validity states
Inexact data dependency information	memory abstraction
Cost depending on input parameters	parametric cost analysis and parametric partitioning algorithm

2. PROGRAM PARTITIONING MODEL

To explore opportunities for computation offloading, we divide an ordinary program into tasks and assign each task either to the server or to the client (i.e. the mobile device), but not both. Server tasks and client tasks do not run simultaneously in our current model. Instead, they take turns to execute by following the original program control flow.

The task scheduling and data transfers between the server and the client are implemented by message passing. At any moment, only one host (the active host) performs the computation. Meanwhile, the other host (the passive host) blocks its task execution and acts in accordance to the incoming messages. To start a task on the passive host, the active host sends a message to the passive host. Upon receiving this message, the receiver becomes the active host and starts the execution of the corresponding task. Meanwhile, the sender blocks its current task execution and becomes the passive host. The active host can also send a message to the passive host indicating the termination of its current task, which will cause the receiver to resume its previously blocked task execution.

2.1 Task Control Flow Graph

We next show how to divide an ordinary program into tasks, and represent the program execution by a task control flow graph ($TCFG$). The $TCFG$ is the program control flow graph (V, E) whose nodes represent tasks. Each edge $e = (v_i, v_j) \in E$ denotes the fact that task v_j may be executed immediately after task v_i under certain control flow conditions.

For a program (for example, a C program) composed of functions, it is tempting to simply define the task as a func-

tion. Unfortunately, if functions are the only nodes in the $TCFG$, then the execution order among the function calls can not be specified by the $TCFG$. Therefore, the tasks must be defined at a finer grain than functions.

There exist different ways to form the tasks, as long as the following definitions are met:

Definition 1: A *task* is a consecutive statement segment in a function that starts with a statement called the *task header*, and ends with a *task branch*. There exist no task branches or task headers in the middle.

Definition 2: A *task header* is a statement of two kinds:

1. The branch target of a *task branch*
2. The statement immediately following a *task branch*

Definition 3: A *task branch* is a branch statement (including conditional and unconditional jumps, function calls and returns) between different *tasks*.

By definition, the entry statement of each function must be a task header because it starts a statement segment in a function. Function calls and returns must be task branches because they jump between statement segments in different functions. Other branch statements may be task branches depending on whether or not they jump between different tasks.

Once the task headers are designated, it is easy to find the task header for each statement. We simply trace the control flow backward from the given statement until reaching a task header. This header is the wanted. Furthermore, we can identify each task by its unique header. It is easy to see that we can make each basic block a task and each branch statement a task branch. In this way, the $TCFG$ is the conventional control flow graph. However, we want the tasks to be as large as possible in order to reduce the size of the $TCFG$. Hence, we use the following iterative algorithm to build the $TCFG$:

Algorithm 1:

```
input: a program
output: TCFG = (V, E)

make the first statement of each function
a task header;
change = TRUE;
while (change) {
  change = FALSE;
  for (each branch statement s) {
    let t be the branch target of s;
    let r be the statement following s;
    find the task header hs of s
    find the task header ht of t
    if (hs != ht) {
      if (r is not a task header) {
        make r a task header;
        change = TRUE;
      }
      if  (t is not a task header) {
        make t a task header;
        change = TRUE;
      }
    }
  }
}

V = {all task headers};
for (each branch statement s) {
  let t be the branch target of s;
  let r be the statement following s;
  find the task header hs of s
  find the task header ht of t
  if (hs != ht) {
    add edge (hs, ht) to E;
    if (s is a conditional branch statement) {
      add edge (hs, r) to E;
    }
  }
}
```

For the example program in Figure 1 (a), Algorithm 1 builds the $TCFG$ shown in Figure 1 (b).

The task assignment decision is represented by a boolean value $M(v)$ for each task v such that:

$$M(v) = \begin{cases} 1 & \text{task } v \text{ is assigned to the } \textbf{server} \\ 0 & \text{task } v \text{ is assigned to the } \textbf{client} \end{cases}$$

2.2 Data Validity States

In this section, we present a new method to represent data communication requirement in the $TCFG$. In traditional approaches (see [12] for an example), whenever a task produces a piece of data which are consumed by another task, a communication cost is charged as long as these two tasks run on different machines. Such an approach may exaggerate the communication cost when the $TCFG$ is generated from an ordinary sequential program, because it overlooks the fact that the produced data may have consumers in more than one task. Take the $TCFG$ in Figure 3 (a) as an example. Task A writes to data item d, task B and task C read data item d. If we partition the program such that task A runs on the server, task B and task C run on the client, we will have two DU-chains from the server to the client: $A \rightarrow B$ and $A \rightarrow C$. The traditional approach charges the communication cost for both DU-chains. However, if we take a closer look at this example, we see that only the data transfer from A to B is necessary, because task C can share the received data with task B.

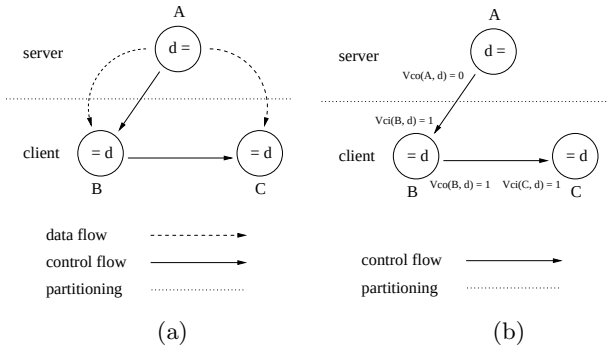


Figure 3: Example of Data Transfer

To overcome the above problem, we represent the data transfer requirement by data validity states. Each shared

data item in the original program has two copies, one on the server and the other on the client. Hence all the tasks assigned to the same host share the same data. We use four binary variables to indicate the data validity states:

$V_{si}(v,d)$: Is data item d valid on the **server** at the **entry** of task v?

$V_{so}(v,d)$: Is data item d valid on the **server** at the **exit** of task v?

$V_{ci}(v,d)$: Is data item d valid on the **client** at the **entry** of task v?

$V_{co}(v,d)$: Is data item d valid on the **client** at the **exit** of task v?

Each of these boolean variables has the value 1 if data item d is valid and value 0 if invalid.

Figure 3 (b) shows an example of data validity states. After the data item d is written on the server in task A, the data item d on the client becomes invalid ($V_{co}(A,d) = 0$). Before the data item d is read on the client in task B, the data item d on the client must be made valid ($V_{ci}(B,d) = 1$). Therefore, the state of data item d on the client changes from invalid to valid, which means that a data transfer is required for d from the server to the client on edge (A,B). From task B to task C there is no need to transfer d from the server to the client, because d on the client remains valid ($V_{co}(B,d) = 1$, $V_{ci}(C,d) = 1$). In this figure, we only show V_{ci} and V_{co}, which determine the data transfers from the server to the client. Data transfers from the client to the server are determined by V_{si} and V_{so}, which always equal 1 in this example.

2.3 Memory Abstraction

To transform an ordinary program into a distributed program for computation offloading, we must honor all the data dependences under all possible program execution contexts. Unfortunately, there often exist control and data flow information which can not be determined at compile time. We take an approach called *memory abstraction* to overcome this difficulty.

At compile time, we abstract all the memory (including code and data) accessed at run time by a finite set of *typed abstract memory locations*. Each run-time memory address is represented by a unique abstract memory location, although an abstract memory location may represent multiple run-time memory addresses. The abstraction of run-time memory is a common approach used by pointer analysis techniques [14] to obtain conservative but safe point-to relations. The type information is needed to maintain the correct data endians and address values during the data transfers between the server and the client. We statically determine all the data transfers in terms of the abstract memory locations and insert message passing primitives for them. At run time, we perform dynamic bookkeeping to correctly map abstract memory locations to their physical memory. This mapping is used by message passing primitives to determine the exact data memory locations.

Figure 4 shows an example of memory abstraction. Function f allocates a linked list with n elements and returns the list header. Although the amount of memory used in this function depends on the function parameter n, we can still abstract all the memory used in this function into six

```
 1: struct list {
 2:   int index;
 3:   struct list *next;
 4: };                              Abstract Memory Locations:
 5: struct list *f(int n)   {
 6:   int i;                          A1: f
 7:   struct list *p, *q = NULL;      A2: n
 8:   for(i = 0; i < n; i++) {        A3: i
 9:     p = malloc(...);              A4: p
10:     p->index = i;                 A5: q
11:     p->next = q;                  A6: memory allocated
12:     q = p;                            in line 9
13:   }
14:   return q;
15: }
```

Figure 4: Example of Memory Abstraction

abstract memory locations $A1$ to $A6$. Here, $A6$ represents all the dynamically allocated memory in line 9.

The dynamic bookkeeping mentioned above is performed by a *registration mechanism* based on the *registration table* and the *mapping table*. Each host has a registration table, whose entries are indexed by the abstract memory location ID for lookup. Each entry in the registration table contains a list of memory addresses for that abstract memory location. The server also maintains a mapping table. Entries in the mapping table contain the mapping of memory addresses for the same data on the server and on the client.

The dynamic *translation mechanism* on the server translates the data representation between the two hosts. Data on the server is first translated and then sent to the client, and data on the client is first sent to the server and then translated on the server. This mechanism translates not only the data endians but also the address values contained in pointer variables, using the mapping table.

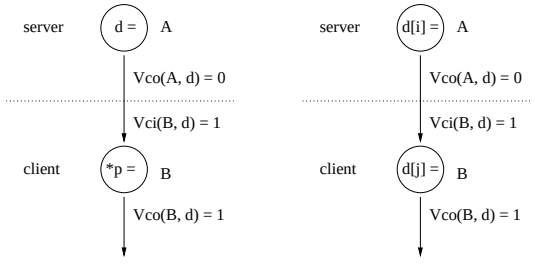
To reduce run-time overhead due to registration, our registration mechanism works only on dynamically allocated data that are *accessed by both hosts*. For each dynamically allocated data item d, we use two variables for the data access states such that:

$$N_s(d) = \begin{cases} 1 & \text{data item } d \text{ is } \textbf{accessed} \text{ on the } \textbf{server} \\ 0 & \text{data item } d \text{ is } \textbf{not accessed} \text{ on the } \textbf{server} \end{cases}$$

$$N_c(d) = \begin{cases} 1 & \text{data item } d \text{ is } \textbf{accessed} \text{ on the } \textbf{client} \\ 0 & \text{data item } d \text{ is } \textbf{not accessed} \text{ on the } \textbf{client} \end{cases}$$

2.4 Program Partitioning Constraints

The task assignment is performed under three kinds of constraints: semantic constraints, data validity state constraints and data access state constraints.



(a) possible write (b) partial write

Figure 5: Example of Conservative Constraint

The program semantics require the distributed program to

have the identical external behavior as the original program that runs on the client only. Hence, all the I/O operations run on the client, as required by the following semantic constraint:

Semantic Constraint: If task v performs I/O operations, then $M(v) = 0$.

There are four data validity state constraints: First, the local copy of data must be valid before each data read (**Read Constraint**). Second, after each data write, the copy of the data on the current host becomes valid and the copy on the opposite host becomes invalid (**Write Constraint**). Third, if the data is not written within a task, then the local copy of the data is valid at the exit of the task only if it is valid at the entry of the task (**Transitive Constraint**). Last, if the data are possibly or partially written in a task, we conservatively require the local copy of that data to be valid before the write (**Conservative Constraint**).

The conservative constraint is necessary because, in static analysis, we sometimes can only obtain the possible or partial data write information. Figure 5 (a) shows an example of possible data write, where pointer p may point to data item d. Under the write constraint, data item d on the client becomes valid after the data write through pointer p ($V_{co}(B, d) = 1$). However, this will cause a problem if p does not point to d. To overcome this problem, we need a conservative constraint to require data item d on the client to be valid before the possible data write ($V_{ci}(B, d) = 1$). In this way, there will be no problem no matter p points to d or not. As a side effect, the conservative constraint will cause a (conservative) data transfer from the server to the client in this case, because data item d on the client is invalid after the data write on the server ($V_{co}(A, d) = 0$). Figure 5 (b) shows a similar example of partial data write. Here, we treat the whole array d as a data unit. Thus each write to an array element is a partial write.

Formally, we can express the data validity state constraints as follows:

Read Constraint: If task v has a (definite, possible or partial) upward exposed read of data item d, then $M(v) \Rightarrow V_{si}(v, d)$ and $\neg M(v) \Rightarrow V_{ci}(v, d)$.

Write Constraint: If data item d is (definitely, possibly or partially) written in task v, then $M(v) = V_{so}(v, d)$ and $\neg M(v) = V_{co}(v, d)$.

Transitive Constraint: If data item d is definitely not written in task v, then $V_{so}(v, d) \Rightarrow V_{si}(v, d)$ and $V_{co}(v, d) \Rightarrow V_{ci}(v, d)$.

Conservative Constraint: If data item d is possibly or partially written in task v, then $M(v) \Rightarrow V_{si}(v, d)$ and $\neg M(v) \Rightarrow V_{ci}(v, d)$.

The data access states depend on the task assignments. Data are accessed on a host only if certain tasks that access the data are assigned to this host. Hence we have the following data access state constraint:

Data Access State Constraint: if data item d is (definitely, possibly or partially) accessed within task v, then $M(v) \Rightarrow N_s(d)$ and $\neg M(v) \Rightarrow N_c(d)$.

3. PARAMETRIC COST ANALYSIS

To find the optimal program partitioning for computation offloading, we perform cost analysis. Two issues are considered. First, different program partitioning decisions have different costs. Hence, we define each cost in such a way that it is counted only *under certain program partitioning conditions*. Second, the value of the cost may change in different program execution instances. Hence, we represent the cost in terms of the *program input parameter values*.

3.1 Cost Factors

Four kinds of costs are identified in our program partitioning:

- **Computation Cost**

  Computation cost is the cost for task execution. For each task $v \in V$, if v is assigned to the server ($M(v) = 1$), there is a server computation cost $c_s(v)$. If v is assigned to the client ($M(v) = 0$), there is a client computation cost $c_c(v)$. The total computation cost can be expressed as:

$$\sum_{v \in V} M(v)c_s(v) + \neg M(v)c_c(v) \tag{1}$$

- **Data Communication Cost**

  Data communication cost is the cost for data transfer between the server and the client. On each edge $e = (v_i, v_j) \in E$, if data item d is transfered from the client to the server on edge e ($V_{so}(v_i, d) = 0$, $V_{si}(v_j, d) = 1$), there will be a client-to-server data communication cost $c_{csd}(v_i, v_j, d)$. If data item d is transfered from the server to the client on edge e ($V_{co}(v_i, d) = 0$, $V_{ci}(v_j, d) = 1$), there will be a server-to-client data communication cost $c_{scd}(v_i, v_j, d)$. The total communication cost can be expressed as:

$$\sum_{(v_i,v_j) \in E, d} \neg V_{so}(v_i, d)V_{si}(v_j, d)c_{csd}(v_i, v_j, d) + \neg V_{co}(v_j, d)V_{ci}(v_i, d)c_{scd}(v_i, v_j, d) \tag{2}$$

- **Task Scheduling Cost**

  Task scheduling cost is the overhead due to task scheduling via remote procedure calls between the server and the client. On each edge $e = (v_i, v_j) \in E$, if v_i is assigned to the client ($M(v_i) = 0$) and v_j is assigned to the server ($M(v_j) = 1$), there will be a client-to-server task scheduling cost $c_{cst}(v_i, v_j)$. If v_i is assigned to the server ($M(v_i) = 1$) and v_j is assigned to the client ($M(v_j) = 0$), there will be a server-to-client task scheduling cost $c_{sct}(v_i, v_j)$. The total task scheduling cost can be expressed as:

$$\sum_{(v_i,v_j) \in E} \neg M(v_i)M(v_j)c_{cst}(v_i, v_j) + \neg M(v_j)M(v_i)c_{sct}(v_i, v_j) \tag{3}$$

- **Data Registration Cost**

  The data registration cost is the overhead due to the registration mechanism. For each dynamically allocated data item d, if d is accessed on both the server and the client, ($N_c(d) = 1, N_s(d) = 1$), there will be a data registration cost $c_a(d)$. The total data registration cost can be expressed as:

$$\sum_{d} N_c(d)N_s(d)c_a(d) \tag{4}$$

3.2 Cost Formula

We now derive the cost formula for cost analysis. For computation cost, we have:

$$c_s(v) = \sum_{i \in v} t_s(i) * \mathbf{r(i, v)}$$
$$c_c(v) = \sum_{i \in v} t_c(i) * \mathbf{r(i, v)}$$

where $t_c(i)$ and $t_s(i)$ are the average time to execute instruction i on the client and on the server respectively, and $r(i, v)$ is the execution count of instruction i in task v.

For data communication cost, we have:

$$c_{csd}(v_i, v_j, d) = t_{scd}(d) * \mathbf{r(v_i, v_j)}$$
$$c_{scd}(v_i, v_j, d) = t_{csd}(d) * \mathbf{r(v_i, v_j)}$$

where $t_{csd}(d)$ and $t_{scd}(d)$ are the time to transfer data d from the client to the server and from the server to the client respectively. $r(v_i, v_j)$ is the execution count of the control edge (v_i, v_j). $t_{csd}(d)$ and $t_{scd}(d)$ can be written as:

$$t_{csd}(d) = t_{csh} + t_{csu} * \mathbf{s(d)}$$
$$t_{scd}(d) = t_{sch} + t_{scu} * \mathbf{s(d)}$$

where t_{csh} and t_{sch} are the data transfer startup time, t_{csu} and t_{scu} are the unit data transfer time and $s(d)$ is the data size.

For task scheduling cost, we have:

$$c_{cst}(v_i, v_j) = t_{cst} * \mathbf{r(v_i, v_j)}$$
$$c_{sct}(v_i, v_j) = t_{sct} * \mathbf{r(v_i, v_j)}$$

where t_{cst} and t_{sct} are the average time for client-to-server and server-to-client task scheduling respectively, and $r(v_i, v_j)$ is the execution count of control edge (v_i, v_j).

For data registration cost, we have:

$$c_a(d) = t_a * \mathbf{r(d)}$$

where t_a is the average data registration time and $r(d)$ is the execution count of the statement that allocates data d.

In the formula above, all the values in bold face are runtime values which should be expressed as functions of input parameters. The other values are constant values which are measured by experiments using synthesized benchmarks.

3.3 Program Flow Constraints

Next, we use program flow constraints to obtain the execution count r and the dynamically allocated data size s. We expressed r and s as functions of input parameter vector $\vec{\lambda}$. Worst case execution time (WCET) estimation techniques [4] use program flow constraints among execution counts r to get an upper-bound on the program execution time. We extend their flow constraints for both r and s:

- The execution count of the program entry node is 1.
- For a dynamic allocation statement whose allocation size is expressed as a function $S(\vec{\lambda})$, we know:

  $$s = rS(\vec{\lambda})$$

  where s is the total size of the data allocated by this statement and r is the execution count of the same statement.
- For a loop whose loop trip count is expressed as a function $L(\vec{\lambda})$, we know:

  $$r_b = r_i L(\vec{\lambda})$$

  where r_i is the execution count of the loop header and r_b is the execution count of the loop body.
- For a branch statement whose branch condition is expressed as a function $B(\vec{\lambda})$ such that the true branch is taken when $B(\vec{\lambda}) = 1$ and the false branch is taken when $B(\vec{\lambda}) = 0$, we know:

  $$r_t = r_i B(\vec{\lambda})$$
  $$r_f = r_i - r_t$$

  where r_t is the execution count of the true branch, r_f is the execution count of the false branch and r_i is the execution count of the branch header.
- For each node n in the control flow graph, we have:

  $$\sum r_i = \sum r_o$$

  where r_i is the execution count of an entry edge of node n and r_o is execution count of an exit edge of node n.

Symbolic analysis techniques [5] such as forward expression substitution and inductive variable recognition are used to find the dynamic data sizes, loop trip counts and branch conditions that are expressed as functions of input parameter vector $\vec{\lambda}$.

3.4 User Annotations

A program may contain features which make it hard (if not impossible) to express r and s as functions of $\vec{\lambda}$. If the optimal program partitioning depends on such unknown r and s values, user annotations are required. We do not limit user annotations to constant values, as was done in many WCET analysis techniques [4]. We allow user annotations expressed as functions of $\vec{\lambda}$, whose values vary with different run time parameter values.

There exist unknown variable values which do not affect the optimal program partitioning. This commonly happens when the different branches of an *IF* statement have approximately equal workload and data access amount. In this case, the branches have little effect on the program partitioning decision, even though we do not know the exact execution count for each branch. This also happens in loops where the ratio between the computation cost and the data communication cost is the same in each iteration of the loop. In this case, the exact value of loop trip count only affects the value of total execution cost, but not the optimal program partitioning decision.

Our parametric algorithm discussed in the next section has an important advantage that it can be used to determine the necessary user annotations. We treat each unknown r or s as a dummy parameter in our parametric partitioning problem and then solve it in the normal way. The dummy parameters which do not occur in the solution will not affect the program partitioning decision. Hence, we only need annotations for the dummy parameters that appear in the solution.

4. PARAMETRIC PARTITIONING ALGORITHM

4.1 The Problem Formulation

The optimal program partitioning problem can be expressed as:

Problem 1: Assign binary values to variables M, V_{si}, V_{so}, V_{ci}, V_{co}, N_s and N_c subject to program partitioning constraints in section 2.4 and minimize the sum of the cost in (1) - (4).

We have the following theorem.

Theorem 1: With given cost values c_c, c_s, c_{csd}, c_{scd}, c_{cst}, c_{sct} and c_a, problem 1 can be reduced to a single-source single-sink **min-cut network flow** problem [2].

Proof:

First, we show how the overall reduction works. We build a min-cut network flow problem (N, A, c, s, t), where N is the node set, A is the arc set, c is the capacity on each arc, s is the source node and t is the sink node. We represent the terms M, V_{si}, V_{so}, $\neg V_{ci}$, $\neg V_{co}$, N_s and $\neg N_c$ by nodes in N. We solve the min-cut network flow problem to cut all the nodes into two sets, the source set S containing s and the sink set T containing t, such that the term represented by a node in S has value 1 and the term represented by a node in T has value 0.

The following two steps build the arc set and define a capacity on each arc such that the min-cut network flow problem is equivalent to problem 1:

1. All the constraints in problem 1 can be normalized into the form $X \Rightarrow Y$, where X and Y are terms. Constraint $X = Y$ can be normalized to $X \Rightarrow Y$ and $Y \Rightarrow X$. Constraint $\neg X \Rightarrow \neg Y$ can be normalized to $Y \Rightarrow X$. We then represent $X \Rightarrow Y$ by an arc $X \rightarrow Y$ with infinity capacity. Therefore if X is cut to S, then Y must also be cut to S. This is equivalent to the constraint $X \Rightarrow Y$. The constant values 1 and 0 in the constraints are represented by node s and node t respectively. This is because s is always cut to S and t is always cut to T.

2. All the costs in problem 1 can be normalized into the form $\neg YXc$, where X and Y are terms and c is the cost value. The cost Xc can be normalized as $\neg 0Xc$, and $\neg Xc$ as $\neg X1c$. We then represent cost $\neg YXc$ by an arc $X \rightarrow Y$ with capacity c. Therefore, if X is cut to S and Y is cut to T, the cut value equals c. This is equivalent to the cost $\neg YXc$.

After these steps, we can know that the minimum cut in the flow network corresponds to an optimal solution to Problem 1. The resulting min-cut network flow problem is equivalent to problem 1. $\diamond$

4.2 A Parametric Algorithm

With all the cost values expressed as functions of $\vec{\lambda}$, the optimal program partitioning problem can be expressed as the following parametric min-cut network flow problem ($\vec{\lambda}$, N, A, c, s, t):

Problem 2:

$$\min \sum_{\substack{a=(n_i,n_j)\in A \\ P(n_i)-P(n_j)=1}} c(a,\vec{\lambda})$$

subject to:

$$\begin{aligned} 0 \le P(n) \le 1, \qquad & \forall n \in N \\ P(s) = 1, \quad P(t) = 0 & \end{aligned} \tag{5}$$

where $\vec{\lambda}$ is the parameter vector, N is the node set, A is the arc set, $c(a, \vec{\lambda})$ is the capacity on arc a expressed as a function of $\vec{\lambda}$, s is the source node and t is the sink node. $P(n)$ is a binary cut variable for each node n. $P(n) = 1$ represents the fact that n is cut to source set S and $P(n) = 0$ represents the fact that n is cut to sink set T.

With the value of $\vec{\lambda}$ given, we can calculate $c(a, \vec{\lambda})$ and, using max-flow/min-cut algorithms, we can find the minimum cut P. However, the value of $\vec{\lambda}$ is only available at run time. Performing the max-flow/min-cut algorithm at run time introduces unacceptable run-time overhead, as the best known algorithms run in $O(n^3)$ time complexity. We statically solve the parametric min-cut network flow problem and find the set $\mathcal{Z}$ of pairs $(P, \mathcal{H})$, such that P is the minimum cut of problem 2 for all parameter values $\vec{\lambda} \in \mathcal{H}$. At run time, we can easily find the set $\mathcal{H}$ that contains the current value of $\vec{\lambda}$. Based on $\mathcal{H}$, the corresponding minimum cut P is determined.

The following theorem is the foundation of our parametric algorithm. It comes from the equivalence between the maximum flow and the minimum cut of a flow network [2].

Theorem 2: A cut P that satisfies constraint (5) is a minimum cut of problem 2 with parameter value $\vec{\lambda}$ if and only if there exists a flow f satisfying the following constraints:

$$\begin{aligned} \textbf{Flow}: \quad & \sum_{a=(n,.)\in A} f(a) = \sum_{a=(.,n)\in A} f(a), \\ & \forall n : n \ne s, n \ne t, n \in N \\ \textbf{Capacity}: \quad & 0 \le f(a) \le c(a,\vec{\lambda}), \quad \forall a : a \in A \\ \textbf{Opt 1}: \quad & f(a) = c(a,\vec{\lambda}), \quad \forall a : \quad a = (n_i, n_j) \in A, \\ & \qquad\qquad P(n_i) - P(n_j) = 1 \\ \textbf{Opt 2}: \quad & f(a) = 0, \quad \forall a : \quad a = (n_i, n_j) \in A, \\ & \qquad\qquad P(n_i) - P(n_j) = -1 \end{aligned} \tag{6}$$

Proof:

$\Rightarrow$ If P is a minimum cut of problem 2 with parameter value $\vec{\lambda}$, from the equivalence between the maximum flow and the minimum cut of a flow network, there exists a maximum flow f satisfying constraints of **Flow** and **Capacity**. Moreover, because the maximum flow value of f is equivalent to the minimum cut value of P, constraints of **Opt1** and **Opt2** must be satisfied. Otherwise, the flow value of f will be smaller than the cut value of P.

$\Leftarrow$ If there exists a flow f satisfying (6), due to the constraints of **Opt1** and **Opt2**, it is easy to verify that the flow value of f is equivalent to the cut value of P. Since a cut value is never less than a flow value, P must be a minimum cut of problem 2 with parameter value $\vec{\lambda}$.

From theorem 2, we immediately have the following lemma.

Lemma 1: For a given cut P satisfying constraint (5), the following set contains all the parameter values $\vec{\lambda}$ which make P a minimum cut:

$$\{\ \vec{\lambda} \mid \exists f : f,\ P,\ \vec{\lambda} \text{ satisfy constraints (6) }\} \qquad (7)$$

Based on the discussions above, we devise the following iterative algorithm to solve the parametric min-cut problem:

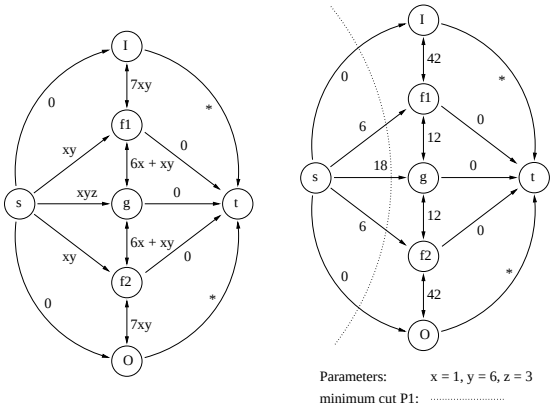
Algorithm 2:
Input: $\mathcal{X}$ - set of parameter values $\vec{\lambda}$
Output: $\mathcal{Z}$ - set of pairs $(P, \mathcal{H})$ such that P is a minimum cut of problem 2 for all the parameter values $\vec{\lambda} \in \mathcal{H}$.

1: $\mathcal{Z} = \phi$
2: while $(\mathcal{X} \neq \phi)$ {
3: Choose a sample parameter value $\vec{\lambda} \in \mathcal{X}$;
4: Find a minimum cut P for problem 2
with parameter value $\vec{\lambda}$;
5: With P, compute set $\mathcal{H}$ defined by (7);
6: Add $(P, \mathcal{H})$ to $\mathcal{Z}$;
7: $\mathcal{X} = \mathcal{X} - \mathcal{H}$;
8: }

In the above, $\mathcal{X}$ and $\mathcal{H}$ are sets of $\vec{\lambda}$ expressed as constraints on $\vec{\lambda}$. A max-flow/min-cut algorithm can be used to find the minimum cut P in step 4. It is not hard to use polyhedral operations to perform step 2, 3, 5 and 7, if $c(a, \vec{\lambda})$ are linear functions. For a nonlinear function $c(a, \vec{\lambda})$, we approximate it as a new parameter independent of $\vec{\lambda}$. This approximation expands the set X, which may produce false min-cut solutions. Since no parameter values at run time will correspond to the false solutions, the false solutions cause no harm except incurring the time to check for nonexistent parameter ranges.

As an example, we show how the algorithm works on the program in Figure 1. We divide the program into 5 tasks, I, $f1$, g, $f2$ and O, where tasks I and O represent the tasks for input and output operations respectively. The parametric min-cut problem is shown in Figure 6 (a), where s and t are the source node and sink node respectively. For simplicity, we modify the original graph to eliminate the data validity state nodes and replace them with the equivalent data communication costs between tasks. The flow network simplification method discussed in later section can automatically do that. For simplicity of discussion, we ignore the task scheduling costs and the registration costs. All the costs marked on arcs existing s are client computation costs and all the costs marked on arcs entering t are server computation costs. The other costs are data communication costs between the tasks. Tasks I and O must run on the client, so they have zero client computation costs and infinite server computation costs (denoted by $*$).

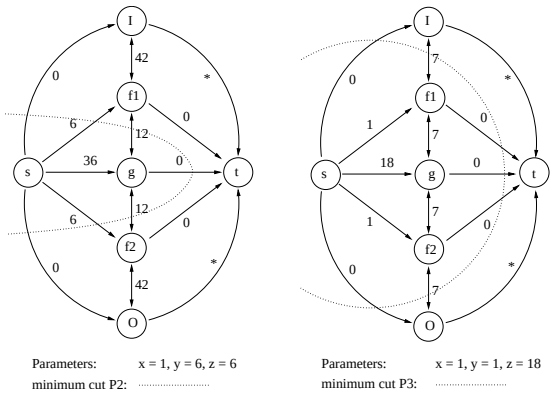
In our parametric algorithm, we first arbitrarily select parameter values and get, for example, $x = 1$, $y = 6$ and $z = 3$. Then we get the min-cut problem and its minimum cut $P1$ in Figure 7 (b). With $P1$, we find the parameter range $R1$: $z \leq 12$ and $yz \leq 12 + 2y$, for which $P1$ is the minimum cut. In the second iteration, we arbitrarily select parameter values in the remaining parameter range and get, for example $x = 1$, $y = 6$ and $z = 6$. Then we get the min-cut problem and its minimum cut $P2$ in Figure 7 (a). With $P2$, we find the parameter range $R2$: $6 \leq 5y$ and $12 + 2y \leq yz$,



(a) (b)

Figure 6: Illustration of Parametric Algorithm (1)

for which $P2$ is the minimum cut. In the third iteration, we arbitrarily select parameter values in the remaining parameter range and get, for example $x = 1, y = 1, z = 18$. Then we get the min-cut problem and its minimum cut $P3$ in Figure 7 (b). With $P3$, we find the parameter range $R3$: $5y \leq 6$ and $12 \leq z$, for which $P3$ is the minimum cut.



(a) (b)

Figure 7: Illustration of Parametric Algorithm (2)

Thus, we obtain three optimal partitioning decisions $P1$, $P2$ and $P3$ for three parameter ranges $R1$, $R2$ and $R3$. Interestingly, although all the costs depend on the parameter x, the optimal program partitioning decisions do not depend on x.

5. IMPLEMENTATION

We implement our parametric analysis in GCC. We use a flow and context insensitive point-to analysis algorithm similar to [1] and apply symbolic analysis in the SSA form. Our symbolic analysis works in a demand-driven way as in [13]. Only the symbols which occur in loop trip counts, dynamic allocation size expressions and branch conditions are backward substituted.

We implement our parametric algorithm with *PolyLib* [8], a library which performs polyhedral operations. PolyLib provides basic functions for our parametric algorithm. The Chernikova algorithm used in PolyLib has $O(n^{\lfloor \frac{d}{2} \rfloor})$ com-

plexity, where n is the number of constraints and d is the polyhedron dimension. This is the theoretical best one can expect for operations on polyhedron because the size of output is of the same order.

5.1 Implicit Functions

Our parametric algorithm does not restrict the computation and communication costs to be closed-form functions. Since all the parameter value sets in Algorithm 2 are expressed as constraints, we can also express cost functions in implicit form as constraints.

Programs often contain many conditional expressions in branch and switch statements. For cost functions which depend on such conditional expressions, the following lemma is useful for transforming a nonlinear function into implicit functions of linear constraints.

Lemma 2: Nonlinear function $f(x, y) = x * y$, where $x \in \{0, 1\}$ and $y \in [N, M]$, is equivalent to the following implicit linear form:

$$\begin{aligned} x * N \leq \quad & f(x, y) \quad \leq x * M \\ y - (1 - x) * M \leq \quad & f(x, y) \quad \leq y - (1 - x) * N \end{aligned}$$

We can easily verify that Lemma 2 is true for both values of x. Lemma 2 can be extended to function $f(x_1, x_2, ..., x_n, y) = x_1 * x_2 * ... * x_n * y$, where $x_i \in \{0, 1\}$ and $y \in [N, M]$. We first transform it into:

$$\begin{aligned} f_1(x_1, x_2, ..., x_n, y) &= x_1 * f_2(x_2, x_3, ..., x_n, y) \\ f_2(x_2, x_3, ..., x_n, y) &= x_2 * f_3(x_3, x_4, ..., x_n, y) \\ &... \\ f_n(x_n, y) &= x_n * y \end{aligned}$$

We then use Lemma 2 to transform each function f_i into an implicit linear form.

We can also extend Lemma 2 to function $f(x, y) = x * y$, where $x \in \{0, 1, ..., n\}$ and $y \in [N, M]$. In this case, we first transform it into:

$$\begin{aligned} f(x, y) &= f_1(x_1, y) + f_2(x_2, y) + ... + f_n(x_n, y) \\ f_1(x_1, y) &= x_1 * y \\ f_2(x_1, y) &= x_2 * y \\ &... \\ f_n(x_n, y) &= x_n * y \end{aligned}$$

where $x_i \in \{0, 1\}$. We then use Lemma 2 to transform each function f_i into an implicit linear form.

5.2 Degeneracy Problem

Due to the degeneracy problem in linear systems, the parameter ranges for different optimal partitioning decisions may overlap with each other. This makes our parametric partitioning solutions depend on the choice of minimum cuts in max-flow/min-cut algorithms. For example, in Figure 8 (a), the small oval denotes the parameter range for partitioning $P1$ and the large oval denotes the parameter range for partitioning $P2$. If the algorithm finds the optimal partitioning $P2$ first, there will be only one optimal partitioning decision $P2$ with the larger parameter range. However, if the algorithm finds partitioning $P1$ first, there will be two optimal partitioning decisions: $P1$ and $P2$.

In our implementation, in order to reduce the number of optimal partitioning decisions, we use a simple heuristic to compare the parameter ranges between each pair of optimal partitioning decisions.

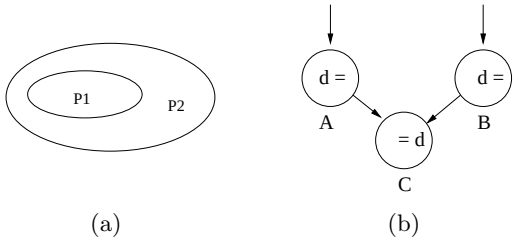


Figure 8: Implementation Issues

5.3 Path Sensitivity Problem

In our current method, the assignment of a task is fixed throughout the run-time execution, regardless the control path which leads to its execution. Such path insensitivity makes our program partitioning model simpler. However, it may lose certain opportunities for computation offloading. For example, in the $TCFG$ shown in Figure 8 (b), task A and B write to data d and task C reads data d. If we partition the program such that task A runs on the server and task B runs on the client, there will be a data transfer, either from A to C or from B to C, depending on the task assignment of C. It is better to assign task C in such a way that C runs on the server if the program control flow goes from A to C and runs on the client if the program control flow goes from B to C. In this way, there will be no data transfer.

In our implementation, we alleviate our limitation of path insensitivity by inlining small functions based on heuristics to achieve certain path sensitivity. In future work, we will consider a general context sensitive program partitioning model for computation offloading.

5.4 Flow Network Simplification

The flow network reduced from problem 1 contains much redundancy due to the infinity capacities. We simplify the flow network based on the following heuristic before applying our parametric algorithm.

Heuristic: For minimum cut network problem $G = (N, A, c, s, t)$, two nodes $n_i \in N$ and $n_j \in N$, can be merged into a single node without losing the optimal solution if the following constraints are satisfied:

$$\begin{aligned} c(n_i, n_j) &\geq \sum_{(n_j, n_k) \in A, n_k \neq n_i} c(n_j, n_k) \\ c(n_j, n_i) &\geq \sum_{(n_k, n_j) \in A, n_k \neq n_i} c(n_k, n_j) \end{aligned}$$

It is easy to verify that, under above constraints, a cut on edge (v_i, v_j) is never better than a cut on all edges (v_j, v_k), and a cut on edge (v_j, v_i) is never better than a cut on all edges (v_k, v_j). Therefore we can merge node v_i and v_j to a single node. We repeatedly check the above constraints and merge the nodes until nothing can be merged.

6. EXPERIMENT

The client used in our experiments is an HP IPAQ 3970 Pocket PC which has a 400MHZ Intel XScale processor. The server is a P4 2GHz Dell Precision 340 desktop machine.

Table 3: Test programs

Program Name	Description	No. of Parameters	No. of Source Lines
rawcaudio	ADPCM in Mediabench, Speech Compression	1	205
rawdaudio	ADPCM in Mediabench, Speech Decompression	1	178
encode	G.721 in Mediabench, CCITT Voice Compression	4	1118
decode	G.721 in Mediabench, CCITT Voice Decompression	4	1248
fft	FFT in Mibench, Discrete Fast Fourier Transforms	3	332
susan	susan in Mibench, Photo Processing	12	2122

Table 4: Parametric Analysis Results

Program	No. of Tasks	No. of Annotations	No. of Partitioning Choices	Analysis Time (s)
rawcaudio	10	2	1	164
rawdaudio	10	2	1	185
encode	107	4	4	2247
decode	87	4	4	2159
fft	26	3	2	748
susan	95	13	3	3482

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. Besides the program execution time, we also measure the program energy consumption. We connect an HP 3459A high precision digital multimeter to measure the current drawn by the handheld device during program execution. In order to get a reliable and accurate reading, we disconnect the batteries from both IPAQ and the extension pack and we use an external 5V DC power supply instead. Further, we make use of the built-in trigger mechanism in the multimeter. After the test program starts running, the IPAQ triggers the meter to do the reading with a high frequency. The trigger stops when the test program finishes. According to our real measurement, the overhead associated with the triggering interrupts is less than 0.5% and the readings are consistent over repeated runs.

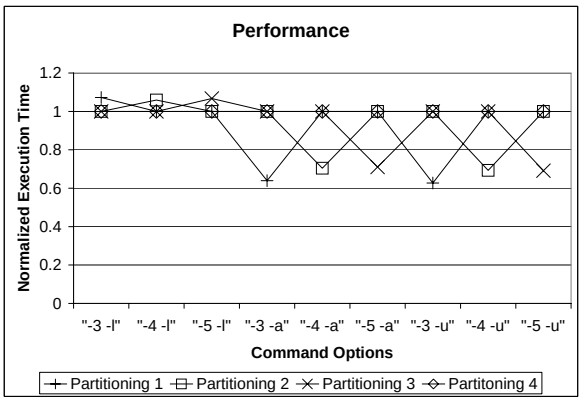


Figure 9: G.721 encode with different options

Table 3 gives the information for the benchmark programs used in our experiments. Programs *rawcaudio* and *rawdaudio* have only one option, which is the input file name. We use the input data size as a parameter. Programs *encode* and *decode* have three command options: the coding method, the audio data format and the input file name. These two benchmarks for the G.721 standard use unbuffered I/O functions, but real G.721 applications use buffers. We modify the code by using buffered I/O and set the buffer size as a new parameter. As we shall see in the experimental results, the buffer size greatly affects the partitioning decision.

Program *fft* has three command options: the sinusoid number, the sample number and a flag for inverse fft. Program *susan* has 12 parameters, 10 of which are command options. The other two are the photo sizes in each dimension (x dimension and y dimension).

6.1 Parametric Partitioning Analysis Result

Table 4 gives our parametric partitioning analysis results. The simplest programs are *rawcaudio* and *rawdaudio*. Their computation cost increases with the input data size. However, our parametric analysis finds that the best way of program execution is always to run the whole program locally, no matter what the input data size is. This is because the communication cost for computation offloading also increases with the input data size. The tradeoff between computation and communication cost remains invariant.

For programs *encode* and *decode*, our parametric partitioning algorithm finds that the input data size does not affect the partitioning decision. The different values of other parameters generate 4 different choices of program partitioning.

For program *fft*, our analysis finds that the sinusoid number and the flag for inverse fft have no effect on partitioning decision. However, the sample number affects the partitioning decision, and there exist two choices of optimal program partitioning decision. Program *susan* has many parameters which lead to 3 kinds of optimal program partitioning.

6.2 Experimental Results

Figure 9 shows the performance results of different partitioning decisions under different coding methods and audio file format compositions for *G.721 encode*. Option *-3* uses G.723 coding for 24kpbs bandwidth, option *-4* uses G.721 coding for 32kbps bandwidth, and option *-5* uses G.723 cod-

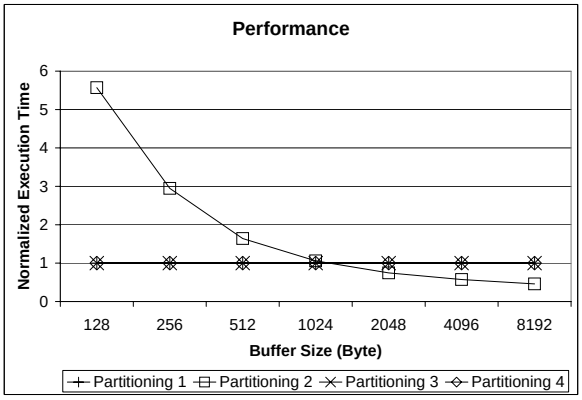


Figure 10: G.721 encode with different buffer sizes

ing for 40kbps bandwidth. Option *-l* works on linear PCM audio data, option *-a* works on a-law audio data and option *-u* works on u-law audio data. All the data in this figure are normalized such that the local program execution time is 1. We see that no single partitioning decision always performs best under all command options. Each partitioning decision can be the best partitioning under certain command options. This fact justifies our parametric analysis for adaptive computation offloading.

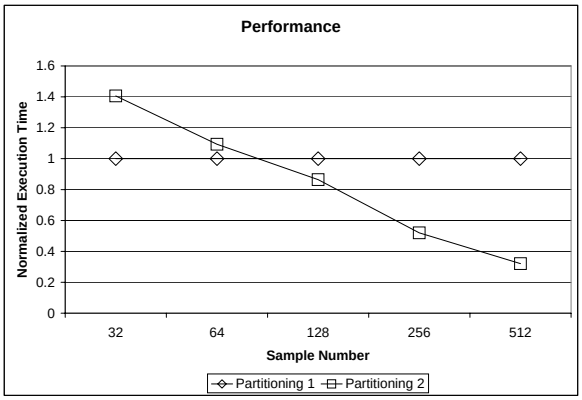


Figure 11: fft with different sample number

Figure 10 shows the performance results of different program partitioning decisions under different I/O buffer sizes for *G.721 encode*. An interesting point here is that the four choices of program partitioning have only two different effects under different buffer sizes. Three of them (Partitioning 1, 3 and 4) have exactly the same effect under different buffer sizes. This is because we use one particular coding method (*-4*) and audio file format (*-l*) to get the performance under different buffer sizes. Under this particular combination of the coding method and the audio file format, the program execution follows a particular execution path, on which partitioning 1, 3 and 4 are exactly the same. We also see from this figure that the buffer size greatly affects the partitioning decisions. Any fixed choice of partitioning may lead up to 60% performance decrease from the optimal choice.

Figure 11 shows the performance results under different sample numbers for *fft*. Again we see that no fixed partitioning decision remains the optimal for all test cases.

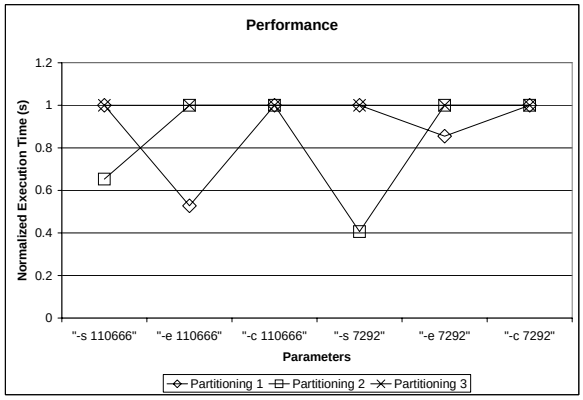


Figure 12: susan with different parameters

We select 6 representative parameter values and show the performance results for *susan* in Figure 12. Option *-s* performs photo smoothing, option *-e* recognizes edges in the photo and option *-c* recognizes corners in the photo. The horizontal axis also shows the input file size. Notice that partitioning 3 always performs the worst in these experiments. According to our analysis, however, it is the best when the input photo is smaller than hundreds of pixels. Such an extreme case is rare in reality. Therefore, it is not used in the experiments.

6.3 Prediction Error

Figure 13 shows the prediction errors of different program partitioning decisions under different command options for the program *G.721 encode*. All the data in the figure are the ratios between the predicated costs and corresponding measured costs. We see that, although we can not guarantee the accuracy of our cost analysis, in the experiments, the prediction errors are all within 10%.

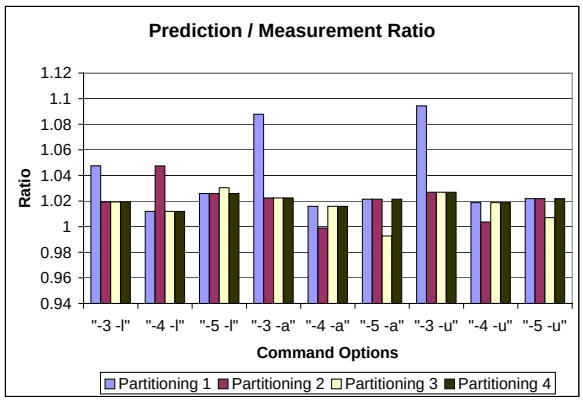


Figure 13: Prediction Error

For all the parameter values shown in Figures 9 to 12, our parametric analysis correctly determines the corresponding optimal choice of partitioning. The average performance improvement of computation offloading over local program execution is about 37%, excluding the execution instances where the whole program executes locally. Energy measurement results for our test programs show that the aver-

age electrical current and voltage do not differ much with different program partitioning decisions, therefore the total energy consumption improves roughly in proportion to the execution time.

7. RELATED WORK

Program profiling information has been used in many program optimization techniques [7] [3]. Profiling information is only useful when the specific program behaviors vary little with different program execution instances. Optimal program partitioning for computation offloading depends on the tradeoff between the program computation workload and communication cost, which often varies strikingly with different program inputs.

Rudenko, Reiher, Popek and Kuenning [11] experiment with computation offloading at process level. They show significant energy saving with computation offloading on laptops. Since they offload an entire program, there is no analysis for program partitioning.

Early computation offloading techniques determine program partitioning with either static analysis [9] or program profiling information [10]. These techniques do not adapt the program partitioning to different program execution instances. As we have seen in our experiments, different program execution contexts may lead to quite different optimal program partitioning decisions.

Worst case execution time (WCET) estimation techniques [4] can be viewed as a special case of parametric analysis. Instead of estimating the program execution time for different input parameters, WCET estimates the worst execution time for all possible program input parameter values.

Feautrier [6] presents a parametric integer programming algorithm for array data flow analysis. We use parametric analysis and parametric partitioning algorithm for adaptive optimization problem of computation offloading.

8. CONCLUSION

The experimental results in our work have shown that the optimal partitioning for computation offloading may vary with different program execution instances. The performance penalty can be quite high if a fixed partitioning decision is used for all execution instances. We have presented a parametric program analysis to transform the program such that the optimal partitioning decision can be made at run time according to the run-time parameter values. In future work, we hope to extend our parametric analysis for other program optimization problems. This would provide a powerful approach to complement profile-guided schemes.

9. REFERENCES

[1] L. O. Andersen. Program analysis and specialization for the C programming language. *PhD thesis, DIKU, University of Copenhagen*, 1994.

[2] J. Bang-Jensen and G. Gutin. Graphs: theory, algorithms, and applications. *Springer-Verlag, London*, 2001.

[3] P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. Profile-guided automatic inline expansion for c programs. *Software Practice and Experience*, 22(5):349–369, 1992.

[4] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. of RTSS'00, 21st IEEE Real-Time Systems Symposium*, 1998.

[5] T. Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *The Journal of Supercomputing*, 12(3):227–252, 1994.

[6] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22:243–268, 1988.

[7] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Proc. of IEEE International Conference on Computer Languages*, 1998.

[8] P. Jansson and J. Jeuring. Polylib - a library of polytypic functions. In *Informal Proceedings Workshop on Generic Programming (WGP'98)*, 1998.

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

[10] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: A partition scheme. In *International Conference on Compiler, Architecture and Synthesis for Embeded Systems*, pages 238–246, November 2001.

[11] 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.

[12] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Proc. of Eighth Heterogeneous Computing Workshop (HCW'99)*, 1999.

[13] P. Tu and D. Padua. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers. In *Proc. of the 1995 International conference on Supercomputing*, 1995.

[14] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, 1995.