# A Compiler Analysis of Interprocedural Data Communication *

Yonghua Ding
Department of Computer Sciences
Purdue University
West Lafayette, IN 47906, USA
ding@cs.purdue.edu

Zhiyuan Li
Department of Computer Sciences
Purdue University
West Lafayette, IN 47906, USA
li@cs.purdue.edu

## ABSTRACT

This paper presents a compiler analysis for data communication for the purpose of transforming ordinary programs into ones that run on distributed systems. Such transformations have been used for process migration and computation offloading to improve the performance of mobile computing devices. In a client-server distributed environment, the efficiency of an application can be improved by careful partitioning of tasks between the server and the client. Optimal task partitioning depends on the tradeoff between the computation workload and the communication cost. Our compiler analysis, assisted by a minimum set of user assertions, estimates the amount of data communication between procedures. The paper also presents experimental results based on an implementation in the GCC compiler. The static estimates for several multimedia programs are compared against dynamic measurement performed using Shade, a SUN Microsystem's instruction-level simulator. The results show a high precision of the static analysis for most pairs of the procedures.

## 1. INTRODUCTION

This paper presents a static analysis to estimate the amount of data communication between procedures. Such an analysis is useful for transforming an ordinary program into one that runs on a distributed system for various purposes. For example, process migration [1] at procedure level has been used for load balance, fault tolerance, and mobility of computing etc. Another example is computation offloading [7, 9, 10, 13]. In a client-server distributed computing environment, the efficiency of an application program can be improved by careful partitioning of the tasks between the server and the client. The efficiency may be defined in terms of the turn-around time, the network bandwidth consumption, or, for wireless-networked handheld devices, the battery consumption. Optimal task partitioning [9, 10] depends on the tradeoff between the computation workload and the communication cost.

The analysis in this paper assumes the availability of interprocedural def-use chains [12]. A def-use chain is a connection from a `definition` for a variable to a `use` for that variable. There is a def-use chain between a definition and a use if the use can be reached from the definition along the control flow graph and call boundary without passing through another definition for that variable. The objective is to derive an upper bound on the amount of data communication between a given pair of procedures and to make such a bound as tight as possible. The analysis is assisted by a compiler-prompted minimum set of user assertions such as loop iteration counts and the size of a dynamically allocated data structure. We implemented a prototype in the GCC compiler and applied the analysis to several multimedia programs. To evaluate the precision of our static analysis, we compare the static estimates against dynamic counts collected with Shade, a SUN Microsystem's instruction-level simulator. The results show that, based on a small number (between one to eight) of user assertions, the analysis has obtained highly accurate estimates for most pairs of the procedures. And we feed our analysis results to the partition scheme [9] for computation offloading. The partition results are exactly the same as the results performed by profiling data communication.

In the rest of the paper, we illustrate a communication model (in section 2), describe the static program analysis framework (in Section 3), present experimental results (in Section 4), discuss related work (in Section 5) and make a conclusion (in Section 6).

## 2. COMMUNICATION MODEL

We first present a communication model as the basis of our compiler analysis for data communication. In our model, the communicated data is packed into a message including the variable index and its value. The receiver decodes the message and assigns the value to the variable associated with the index. In a distributed computing system, for each procedure with data communication between client and server, there are two versions of the procedure, one executed on the client side and the other on the server side. If a procedure is assigned to the client side and its data must be sent to the server, then senders of data messages are inserted in the

*This work is sponsored in part by National Science Foundation through grants CCR-0208760, ACI/ITR-0082834, and CCR-9975309, and by Indiana 21st Century Fund.


*SC* '03, November 15-21, 2003, Phoenix, Arizona, USA

```
f1() {
  x = ...;
  y = ...;
  if ( a )
    x = ...;
  for (i=0; i<20; i++) {
    if ( b )
      x = ...;
    else
      y = ...;
    call f2();
  }
}
f2() {
  ......
}
```

Original code

```
f1_client() {
  x = ...;
  y = ...;
  if ( a )
    x = ...;
  send(x, x_value);
  send(y, y_value);
  for (i=0; i<20; i++) {
    if ( b ) {
      x = ...;
      send(x, x_value);
    }
    else {
      y = ...;
      send(y, y_value);
    }
    call f2_client();
  }
}

f2_client() {
  send(f2, task_start);
  while (true) {
    receive message (var_name, value);
    if ( var_name == f2 )
      break;
  }
}
```

Client code

```
f1_server() {
  while(true) {
    receive message(var_name, value);
    if ( var_name == x )
      x = value;
    else if ( var_name == y )
      y = value;
    else if ( var_name == f2 ) {
      call f2_server();
      send(f2, task_end);
    }
    else
      break;
  }
}

f2_server() {
  ...... /* Original code */
}
```

Server code

**Figure 1: An example program and its client/server code based on our communication model**

client version of the procedure and the corresponding message receivers are inserted in the server version. The server version does not execute any of the statements except the receivers of data messages and procedure calls (if the received message is task_start of a procedure). Conversely, if a procedure is assigned to the server side and its data must be sent to the client, then senders of data messages are inserted in the server version, and the corresponding message receivers are inserted in the client version.

If procedure $P$ (assigned to the client side) calls procedure $Q$ (assigned to the server side), then the client version of $P$ will call the client version of procedure $Q$ which sends a `task_start` message to, and receives a `task_end` message from, the server version of $P$. The server version of $P$ makes the call to the server version of $Q$ after it receives `task_start`. When the server version of $Q$ returns, the server version of $P$ will send a `task_end` message to the client which ends the execution of the client version of $Q$ and continues to execute the rest of the procedure $P$ in client.

The insertion of receivers is straightforward since the version, which executes the receivers and procedure calls if the received message is task_start of a procedure, executes no other statements. The insertion of senders, on the other hand, requires information on data communication requirement. Analysis of such information is the main topic of the next section. It is important to note that, in our current work, procedures are not executed in parallel. When the server runs a procedure, the client waits for the message from the server, and vice versa. For data communication which involves global variables, the example in Figure 1 illustrates our model. In the example, we suppose there is data communication between $f1$ and $f2$, and the two procedures are partitioned into different sides. (For heap variables and procedure locals, the insertion of send and receive will be the same, but additional book-keeping code must be inserted to keep track of the memory addresses of the communicated data.)

## 3. A STATIC ANALYSIS

In this paper, we analyze an ordinary C program. There exists data communication from procedure $f1$ to procedure $f2$ if the former generates data which are used in $f2$. To estimate the total amount of data communicated from $f1$ to $f2$, we distinguish three modes of communication.

In the first mode, $f1$ calls $f2$ directly and passes values to $f2$ in the parameter list. In the second mode, $f1$ calls $f2$. At the end of $f2$, it may return values to $f1$. These two modes are quite simple to analyze and we do not go into their details. We focus, instead, on the third mode: data being communicated from $f1$ to $f2$ due to defs in $f1$ which reach the uses in $f2$. In this mode, the two functions do not necessarily have any calling relationship, i.e. it is possible that neither $f1$ nor $f2$ calls the other (directly or indirectly [1]). For example, the main function may call $f1$ first before calling $f2$, causing some values defined in $f1$ to be used by $f2$.

Our analysis is performed for one pair of procedures $(f1, f2)$ at a time, assuming that the interprocedural def-use information [12] is given. We analyze the data communication cost as if the two procedures are assigned to two different hosts already. The analysis follows the framework shown below, with the worst-case time complexity of $O(N^2)$, where $N$ is the number of procedures, multiplied by the time required to conduct the work listed below:

1. We first estimate the amount of data sent from $f1$ to $f2$ in *a single invocation of* $f1$. (Recall that $f1$ may or may not call $f2$.)
2. Since $f1$ may be called by other procedures from within loops or called recursively, we estimate the number of invocations to $f1$ which introduce data communication from $f1$ to $f2$ during the program's execution. Multiplying the amount obtained in Step 1 by the number

[1] If $f1$ calls $f3$ and $f3$ calls $f2$, we say that $f1$ calls $f2$ indirectly. In the rest of the paper, unless we state explicitly, when we say $f1$ calls $f2$, this includes indirect calls.

```
     f1() {                    f2() {
s1     g = 1;             s4     ... = g;
s2     if (...) {                }
s3       g = 2;
       }
       call f2;
     }
```

**Figure 2: An example program showing communication overcount, where $g$ is a global variable**

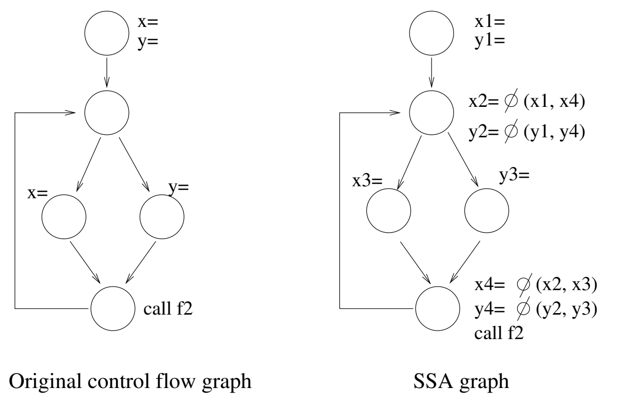


**Figure 3: An example program showing that the SSA form may introduce overcount, suppose there exist uses of $x$ and $y$ in $f2$**

of these invocations of $f1$, we get the total amount of data communicated from $f1$ to $f2$.

An important requirement for our analysis is to avoid overcount. Fig. 2 shows an example. There exist two def-use chains between procedures $f1$ and $f2$, one from $s1$ to $s4$ and the other from $s3$ to $s4$. It requires the transfer of only one piece of data from $f1$ to $f2$ by sending the data before the call site. If we estimate data communication by counting both def-use chains, we will overcount. In contrast, suppose that there exists a procedure call between $s1$ and $s2$ which eventually invokes $f2$, then it requires the transferring of two pieces of data, one generated in $s1$ and the other in $s3$.

Although we can remove overcount in the example in fig. 2 by applying static single assignment (SSA) [4] transformation, the SSA form may introduce overcount in other cases such as fig. 3 shows. After SSA transformation, there are two def-use chains, one from $x4$ to procedure $f2$ and the other from $y4$ to procedure $f2$, and the two defs are in the loop containing procedure call to $f2$, so the amount of data communication is $2n$ ($n$ is the loop count). Actually, in each iteration, there is only one of $x$ and $y$ defined in the loop can reach the use in procedure $f2$. Thus the upper bound of data communication can be as tight as $2 + n$. (The defs outside the loop introduce 2 units of data communication.) In this case, there is overcount introduced by SSA transformation when the loop count is great than 2.

Next, we discuss the two steps mentioned above in two separate subsections.

## 3.1 Evaluating Data Sent from a Single Invocation of $f1$ to $f2$

In order to minimize the overcount of data communicated from $f1$ to $f2$, we partition the defs in $f1$ into groups such that the members in the same group are re-assignments to the same variable, or are mutually exclusive, i.e. one taking place implies the other *not* taking place. We apply two kinds of grouping methods. *Sequence grouping* is used to merge defs of the same variable, and *branch grouping* is used to merge defs on mutually-exclusive branches. After grouping, we estimate the data communication represented by each def-use chain in each group, and we take the maximum as the amount represented by that group. If a def is embedded in a loop and the loop body contains a call to procedure $f2$, then the amount of data communication estimated for that def should be multiplied by the loop count. For those defs associated with arrays, pointers or structures, we need to estimate the size of modified data which will be used in the receiving procedure. We apply array reaching-definition analysis to get such estimation. We describe the two kinds of grouping methods in the following.

### 3.1.1 Sequence Grouping

To identify the candidate defs of sequence grouping, we refer to the *nearest common postdominator* (NC-PDOM).

*Definition 1* (PDOM) [15]: Let x and y be basic blocks, the Control Flow Graph (CFG) has a unique EXIT node which represents the exit of the procedure. x PDOM y $\Leftrightarrow$ every path from node y to EXIT node includes x.

*Definition 2* (NC-PDOM): The nearest common postdominator (NC-PDOM) of two basic blocks $b1$ and $b2$ is their nearest common ancestor in the PDOM tree.

**Claim 1** Two def-use chains associated with the same variable can be placed in the same group, if there exist no calls to $f2$ in any execution path from any of the two defs to their NC-PDOM.

*Proof*: In this case, only one def will actually provide the value to $f2$ through its def-use chain.

The example shown in fig. 2 illustrates the case in which two defs (in $s1$ and $s3$) can be grouped together by applying sequence grouping.

### 3.1.2 Branch Grouping

Two defs on mutually-exclusive branches can also be placed in one group if we can make sure that only one of the two defs can reach the use in $f2$ at run time. Fig. 4 shows a program example illustrating branch grouping. There exist two def-use chains, one for $g1$ and the other for $g2$, both being global variables. If the branch containing the def of $g1$ is taken, then the routine $f2$ which is called immediately after will use the value of $g2$ defined elsewhere, instead of the one defined in $f1$. In this case, the def of $g2$ in procedure $f1$ should not be counted. Similarly, if the branch containing the def of $g2$ is taken, then $g1$ used in routine $f2$ is not defined in current invocation of routine $f1$.

The situation will be different if the mutually-exclusive branches are embedded in a loop which does not contain a call to $f2$. Suppose, instead, $f2$ is called after the loop terminates. The two defs may not be placed in the same group in this case. This is because both branches may be taken in different iterations and thus both may reach routine $f2$. Suppose the data size of communication are $d1$ and $d2$ in two branches respectively, and the loop count is $n$.

```
f1() {
  if (...) {
    g1 = 1;
    f2();
  }
  else {
    g2 = 1;
    f2();
  }
}

f2() {
    ... = g1;
    .... = g2;
}
```

**Figure 4: An example program showing how to perform branch grouping, where $g1$ and $g2$ are global variables**

The amount of data communication estimated by branch grouping is $max(d1, d2) * n$, and that of non-grouping is $d1 + d2$. So in this case, only when the loop count is less than 2, branch grouping achieves tighter upper bound than non-grouping.

#### *3.1.3 Finding an optimal upper bound of data communication is NP-hard*

Generally, the optimal number of data sends depends on what control path is taken at run time. Thus it is difficult to minimize data sends for all possible control paths. If there exists a way to minimize the send operations no matter what path is taken, then it is an NP-hard problem to derive a tight upper bound on the number of send operations among all possible control paths. We reduce the problem to a special longest-path problem on DAG as follows.

- First, on the control flow graph (CFG) which is augmented by information of def-use chains, we mark the data size of defs on each node which has def-use chain reaching $f2$.
- Second, for each node in a loop, if the loop contains a call to procedure $f2$, we replace the data size marked on the node by the product of the data size and the iteration count of the loop. Otherwise, we keep the original data size. We then remove the backward edge associated with the loop from the CFG. Thus the transformed CFG is a DAG (We assume there is no GO statements for simplicity).
- Third, we apply sequence grouping on the transformed CFG as we discussed in previous section.

After the above steps, the problem of deriving the tightest bound on data sends becomes that of finding a longest path in the DAG such that the defs in the same group is counted exactly once on any path. We prove the reduced problem to be NP-hard by transforming the CNF-satisfiability problem (CNF-SAT) to this problem, and we know CNF-satisfiability problem is NP-hard [3]. We construct a DAG based on the variables in CNF-SAT as fig. 5 shows. For each variable $x$ in CNF-SAT, we construct two mutually-exclusive branches to represent the two literals ($x$ and $\neg x$). For each clause in the CNF, we associate it with a variable on the DAG. For example, the first clause is associated with variable $v1$, the second clause is associated with variable $v2$, and so on. For each literal in a clause, we insert a def of the variable on the node associated with the literal. All the defs associated with the same variable on the DAG are grouped together and the weight of each def is 1. If the weight of the special longest path on the DAG equals the number of clauses (the number of variables on the DAG), the CNF-SAT is satisfiable. Thus the CNF-SAT is reduced to the special longest-path problem on the DAG.

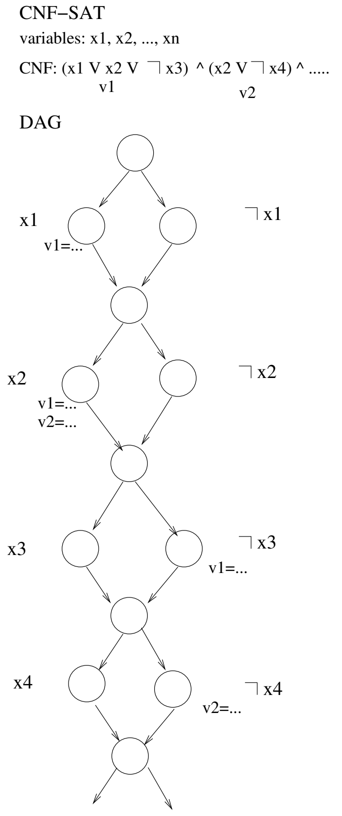


**Figure 5: CNF-SAT reduced to the special longest-path problem on DAG**

#### *3.1.4 Heuristics for Grouping and for Insertion of Senders*

To reduce overcount, we try to put those defs with large data communication into a group. Thus we sort the defs according to their weights, defined as the product of their data size and the iteration counts of the loops which contain both the def and a call to procedure $f2$. We then pick the defs one by one, in the decreasing order of weights, for grouping (using either sequence grouping or branch grouping, whichever applicable). A def can be inserted into an existing group if we can group it with each def in the group by applying sequence grouping or branch grouping. In the program shown in fig. 3, we place the two defs in the loop body into the same group by branch grouping rather than grouping the two defs of variable $x$ and grouping the two defs of variable $y$.

After the analysis and grouping, the senders are inserted using the following simple rules. In each group, the defs grouped by sequence grouping need only one sender, which is inserted at the NC-PDOM. (If the NC-PDOM is embedded in a loop which does not contain a call to procedure $f2$, then the sender can be inserted immediately after the loop.) For the rest of the defs in that same group, they are placed there by branch grouping and must belong to mutually-exclusive

```
int g;
main() {        f1() {       f2() {       f3() {
  f3();           g = 1;       ... = g;     f1();
  f2();         }            }            }
}
```

**Figure 6: An example program showing how to accumulate data communication interprocedurally**

branches. Each of these defs needs just one sender which is inserted in its definition node.

In the example shown in fig. 1, we suppose procedure $f1$ is assigned to the client side and procedure $f2$ is assigned to the server side. After we apply grouping to the defs which reach the use in procedure $f2$, we obtain three groups. The first group includes the two defs in the loop. The second group includes the two defs of variable $x$ outside the loop. The last group includes the def of variable $y$ outside the loop. Suppose the data size of variables $x$ and $y$ are the same and their message size are both $m$. The first group then has the communication amount of $20m$, and the other two groups both have the amount of $m$. So the total amount is $22m$.

## 3.2 Accumulating Data Communication in Multiple Invocations of $f1$

After estimating the amount of data sent from $f1$ to $f2$ in a single invocation of $f1$, we evaluate the number of effective invocations of $f1$ which introduce the data communication. We call an invocation of $f1$ to be an *effective* invocation if it is followed by an invocation of $f2$ before another invocation of $f1$. For example, if $f1$ is called in a loop without calling $f2$, then the number of effective invocation of $f1$ in this loop is only one, even though $f1$ may be called several times. The estimation of the number of effective invocations is done by traversing the call graph as follows.

Each edge of the call graph represents a call site. Thus, the call graph is a multigraph. In the call graph, we initially mark the count of 1 on $f1$, as the initial estimate of calls to $f1$. We mark 0 on all other procedures. We then initialize a work list such that it contains all predecessors of $f1$ in the call graph. Until the work list becomes empty, we extract a procedure, say $f3$, from it and examine all the call sites in $f3$ which lead to an effective invocation of $f1$.

To minimize overcounting the number of effective invocations to $f1$, we apply the same idea used in sequence grouping and branch grouping to the call sites. If a call site is embedded in a loop and the loop body calls $f2$, then the count is multiplied by the loop iteration count. The total count of effective invocations to $f1$ which originates from $f3$ is marked on $f3$ in the call graph. If the new count is different from the old count marked on $f3$, all predecessors of $f3$ in the call graph are added to the work list. When the work list eventually becomes empty, the count marked on the main procedure will be the total number of effective invocations of $f1$ throughout the whole program.

We use the example program in fig. 6 to illustrate the interprocedural analysis of data communication. In the example, the amount of data communicated from procedure $f1$ to $f2$ in a single invocation of $f1$ is 4 bytes. During the traversal of the call graph, we initially mark the number of effective invocations to procedure $f1$ by 1 and mark all other

```
communication = unit data communication of the SCC;
for each procedure which is in the SCC {
  for each path from the entry to a call site whose callee is in the SCC {
    if ( f1 is called in the path directly or indirectly )
      mark f1 be in head;
    if ( f2 is called in the path directly or indirectly )
      mark f2 be in head;
  }
  for each path from a call site whose callee is in the SCC to the exit {
    if ( f1 is called in the path directly or indirectly )
      mark f1 be in tail;
    if ( f2 is called in the path directly or indirectly )
      mark f2 be in tail;
  }
  if ( both f1 and f2 are in head or both f1 and f2 are in tail ) {
    communication = unit_communication * recursion_count.
    break;
  }
}
```

**Figure 7: An algorithm to compute data communication in Case 4 by traversing an SCC**

procedures by 0. Worklist initially contains $f3$. When we process procedure $f3$, there is only one group of call sites and the group contains only one call site. Thus, $f3$ is also marked by 1. We add the main procedure to the work list. During the processing of the main procedure, we find two groups of call sites, one containing a call to $f3$ and the other containing a call to $f2$. These two groups cannot be merged into a single group because $f3$ calls $f1$ and the two call sites are not on mutually-exclusive branches. Procedure $f3$ is marked by 1 and procedure $f2$ is marked by 0. Finally, the main procedure is marked by 1 which is the total number of effective invocations of $f1$. Multiplying this number by 4 bytes, we find that in total $f1$ sends 4 bytes to $f2$.

## 3.3 Handling Recursion

Recursive calls complicate the interprocedural analysis of data communication, and not all recursions can be transformed to loops by compiler. In general, it is difficult to estimate the number of times a procedure is called before recursive calls end. Therefore we prompt the user for an assertion.

Even with such a user assertion, we must carefully examine the order in which recursive calls take place, Suppose we want to compute the data communication sent from $f1$ to $f2$. During the traversal of the call graph, when we process a procedure which is in a strongly-connected-component (SCC), for each procedure $f3$ in the SCC, we examine all its calls to procedures *not* in the same SCC. After processing all such calls, we derive a count of effective invocations of $f1$ which originates from $f3$, not counting calls to other procedures in the same SCC. Repeating this for all procedures in the same SCC, we mark the SCC as a whole by an *initial count* which is the maximum count among whole procedures in it. We then classify the following sequences of recursive calls.

Case 1, both procedures $f1$ and $f2$ are in the SCC. In this case, the count marked on the SCC equals to the initial count multiplied by the recursion count supplied by the user.

Case 2, one of the two procedures $f1$ and $f2$ is in the

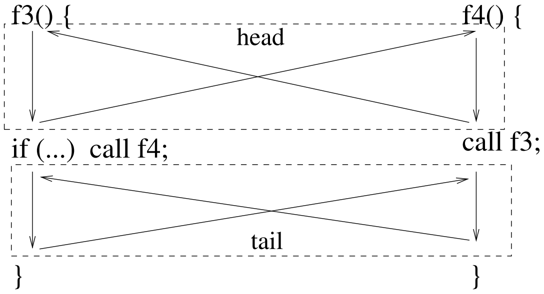


**Figure 8: The execution path in the SCC and the head, tail area**

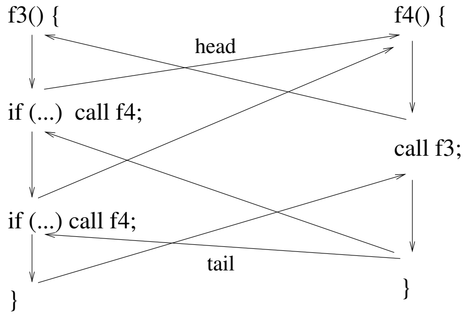


**Figure 9: The execution path in the SCC with multiple call sites in a procedure**

SCC, and the other is called by some procedure in the SCC. In this case, the count marked on the SCC is also the initial count multiplied by the recursion count.

Case 3, the SCC does not call $f1$ or $f2$. In this case, the count marked on the SCC is simply the initial count.

Case 4, neither $f1$ nor $f2$ is in the SCC, and the SCC calls both $f1$ and $f2$, and $f1$ does not call $f2$, and vice versa. Further, each procedure in the SCC has only one call site whose callee is also in the SCC. This case is handled by the algorithm in fig. 7. Fig. 8 shows the execution paths in an SCC in this case. The execution of the "head" region does not interleave with the execution of the "tail" region. If the calls to $f1$ and $f2$ are separated, one in the "head" region and the other in the "tail" region, we can transfer the data at the conjunction between The "head" and "tail" regions. Thus the count marked on the SCC should be equal to the initial count. However, if the calls to $f1$ and $f2$ are both in the "head" region or both in the "tail" region, then the two calls interleave with each other during the execution. In this case, the count marked on the SCC should be the initial count multiplied by the recursion count.

Case 5, neither $f1$ nor $f2$ is in the SCC, and the SCC calls both $f1$ and $f2$. Further, there exists one procedure in the SCC which has two or more call sites whose callees are in the SCC. In this case, the execution of the "head" region may interleave with the execution of the "tail" region, as shown in fig. 9. In this case, the count marked on the SCC should be the initial count multiplied by the recursion count.

### 3.4 Determining a Minimum Set of User Assertions

Our static analysis of interprocedural data communication requires user assertions for loop iterations, memory allocation size and recursion count. In the worst case, user assertions include loop iteration number of each loop, memory allocation size of each memory allocation, and recursion count of each recursion call. Though it may not be a mission impossible, it is a big burden to the user. In this section, we present a method to find a minimum set of user assertions.

We note all variables, which directly related with the computation of loop iteration number, memory allocation size, or recursion count, as needed variables (The value of these variables are required to get the loop iteration number etc.). With program slicing techniques [5, 14], we construct a *variable relation graph* for the given program. Fig. 10 shows an example program and its variable relation graph. In the graph, each node associates with a variable, and it has two status, "known" and "unknown". All the root nodes (with no predecessors) are input variables, and the black nodes (namely needed nodes) associate with needed variables. Directed edge (u,v) from node $u$ to node $v$ means node $v$ is depend on node $u$. One node can be derived to "known" if all of its predecessors are "known". Our objective is to find a minimum set of nodes which are initially set "known" and can induce all needed nodes to be "known". Naively, the root set and the needed nodes set are two sets that can induce all needed nodes to be "known", however, they may not be the minimum set.

To find a minimum set of user assertions, we transform the variable relation graph to a connected graph with source and sink node as following.

- First, we insert a source node on the graph, and connect the source node with each root node by a directed edge from the source node to the root node.
- Second, we insert a sink node on the graph, and connect each needed node with the sink node by a directed edge from the needed node to the sink node.
- Third, split each non-root node X with multiple out-edges into two nodes, namely node X1 and node X2. We link in-edges of original node X to node X1 and link out-edges of original node X from node X2, and connect node X1 with node X2 by a directed edge from node X1 to node X2.

Fig. 11 shows the transformed graph from the variable relation graph in fig. 10. Thus we get a directed graph with source node and sink node, and we set the capacity of each edge as 1. The objective is transformed to find a minimum s-t cut (S,T) on the new graph, and the minimum assertions set is the set of nodes satisfying any of the two following criteria.

- node $v$, if (u,v) is a cut edge from S to T and node u is the source node.
- node $u$, if (u,v) is a cut edge from S to T and node u is not the source node.

**Theorem 1**: The number of nodes in the minimum assertions set is the value of the minimum s-t cut (S,T).

**Proof**: Each node in the minimum assertions set is associated with at most one cut edge from S to T in the minimum s-t cut. Otherwise we can replace the cut with another cut with less cut edges and thus make the original cut be not a minimum s-t cut. Each cut edge is associated with at most one node in the minimum assertions set according to

```
A = x0+x1+x2;
p = malloc(A*sizeof(int));
F = 2*A;
for (i=0; i < F; i++) {
  ......
}
B = x3;
C = x3+x4;
D = x4*x5;
E = x5*x5;
for ( i=C; i < D+E; i=i+B ) {
  ......
}
```

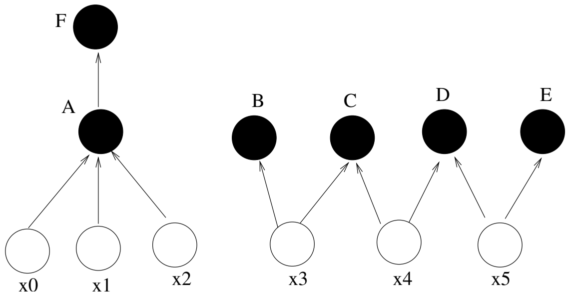


**Figure 10: An example program and its variable relation graph**

the definition of minimum assertions set. Thus Theorem 1 is proved.

**Theorem 2**: The minimum set of nodes associated with a minimum s-t cut (S,T) on the transformed graph is a minimum assertions set on the original variable relation graph.

**Proof**: We prove the theorem in two steps,
Step 1, we prove that all needed nodes can be derived by the nodes in assertion set obtained with the minimum s-t cut (S,T). If one needed node, namely node B, cannot be derived from the nodes in the assertion set, there must be a path from source node to node B, and all nodes on the path are "unknown". So the original cut is not an s-t cut since there exists a path from the source node to the sink node without a cut.
Step 2, we prove that the assertion set is a minimum set. Otherwise, there exists a smaller set which can induce all needed nodes. We construct an s-t cut on the transformed graph as following. For each node in the smaller set, if it is a non-root node with multiple out-edges, we cut between the two split nodes, if it is a non-root node with one out-edge, we cut on this out-edge. If it is a root node, we cut the edge incident on the source node and the root node. Such a cut is an s-t cut with less cut value compared with the original cut according to Theorem 1, so the original cut is not a minimum s-t cut.

There exists many algorithms one can use to solve the minimum s-t cut problem [3, 6]. In the graph in fig. 11, the minimum set of user assertions is {A, x3, x4, x5}. After the compiler determines the minimum set of user assertions, it prompts the user to provide the input value of variables in the minimum set, and it computes all other required assertions by interpretatively running the sliced code with the user assertions as inputs.

## 4. EXPERIMENTS

In this section, we first describe our implementation of interprocedural data communication analysis and then compare the results against simulation results.

### 4.1 Implementation

We have implemented the techniques described above in the GCC compiler (v3.0). We implemented new modules based on the abstract syntax tree in GCC. It is more difficult to collect information such as pointer dereferences based on the lower level (i.e. the RTL level) intermediate representation in GCC. We implemented the following new modules in GCC:

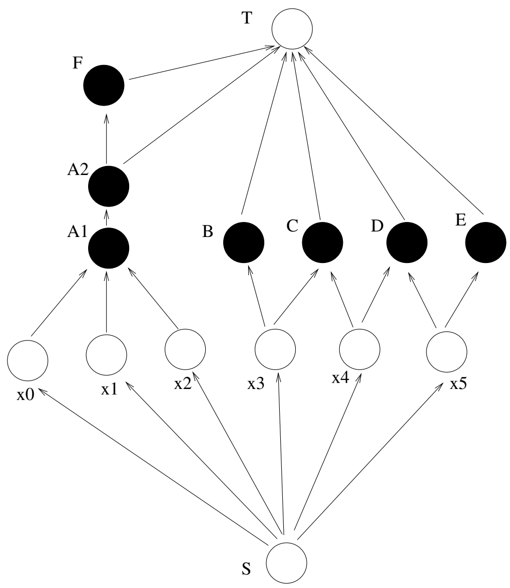


**Figure 11: A graph with source and sink transformed from a variable relation graph**

- Call graph construction
- Clean-up
- Pointer analysis
- Control flow graph construction
- Define-use chains construction
- Array reference analysis
- Evaluate data communication

In the call graph construction, we took into account function pointers and recursive functions for which we compute their strongly-connected-component (SCC). The clean-up module is implemented to ease our subsequent analyses. For example, a function call in a complex expression is split from the expression to simplify the interprocedural analysis. We perform the pointer analysis globally. For example, we can analyze a local pointer in one procedure which points to a local variable in another procedure. The construction of def-use chains is also global because a definition in one procedure may be used in another procedure through pointer or global variable. In other words, there may exist def-use

| Program | Description | Input Parameters in Simulation | Assert1 | Assert2 |
|---|---|---|---|---|
| rawcaudio | speech compression | rawcaudio<clinton.pcm>out.adpcm | 1 | 1 |
| rawdaudio | speech decompression | rawdaudio<clinton.adpcm>out.pcm | 1 | 1 |
| encode | G.721,voice compression | encode -4 -l<clinton.pcm>out.g721 | 6 | 3 |
| decode | G.721,voice decompression | decode -4 -l<clinton.pcm.g721>out.pcm | 5 | 2 |
| epic | image compression | epic test_image -b 25 | 58 | 7 |
| unepic | image decompression | unepic test_image.E test_image.out | 42 | 8 |
| toast | GSM voice transcoding | toast -fpl clinton.pcm | 9 | 6 |
| untoast | GSM voice transcoding | untoast -fpl clinton.pcm | 9 | 6 |

**Table 1: Experiment programs**

chains whose definition and use are in different procedures. After we obtain the def-use chains for the entire program, we evaluate data communication for each pair of procedures and apply sequence grouping and branch grouping in two steps as we described in previous section.

Currently we have implemented a simple array reference analysis in which we count only the number of references, but not the referenced data section. We conservatively take each reference as an instance of data communication with the size of an array element or a structure field.

## 4.2 Benchmark programs

We conducted experiments on several programs in MediaBench [8]. Table 1 lists those programs, their description and their input parameters, including the input files we use in the experiments. All the programs and input files can be found on the MediaBench web-site. The last two columns in table 1 show the effectiveness of our method to determine minimum assertion set. Column $assert1$ shows the number of original user assertions, and column $assert2$ shows the number of minimum user assertions applying our method, including the assertions of loop iteration counts and the size of dynamic memory allocation. In programs EPIC and UNEPIC, most of loop counts can be derived by three input variables $x\_size, y\_size$ and $num\_level$.

## 4.3 Dynamic counting data communication by simulation

We use the SHADE [2] simulator from SUN Microsystem to dynamically measure data communication between procedures. SHADE is an instruction-level simulator and a custom trace generator. Application programs can be executed and traced under the control of user-supplied trace analyzer. We add to the simulator a definition table which records the procedure in which the latest store occurs for each memory address. For each load the simulator will check the definition table to count data communication.

## 4.4 Results

We show our results in Figures 12 through 19. In each figure, "grouping w/o array reference analysis" means we apply our program analysis by considering array or other data structure as scalar with the unit data size of entire array or data structure. "Grouping with array reference analysis" means our program analysis, augmented with array reference analysis, can treat each array element or field of data structure as a unit data.

Since the amount of data communication between different pairs of procedures varies significantly, we normalize the

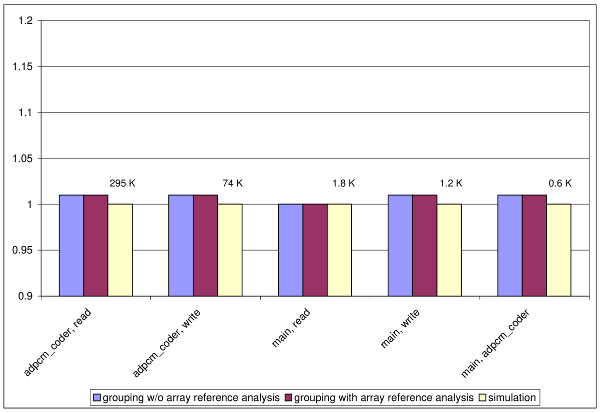


**Figure 12: ADPCM: rawcaudio**

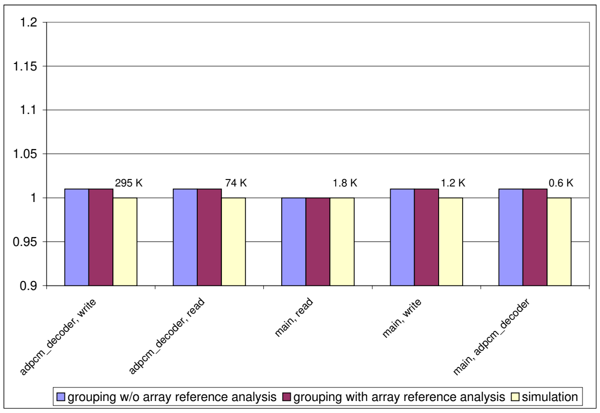


**Figure 13: ADPCM: rawdaudio**

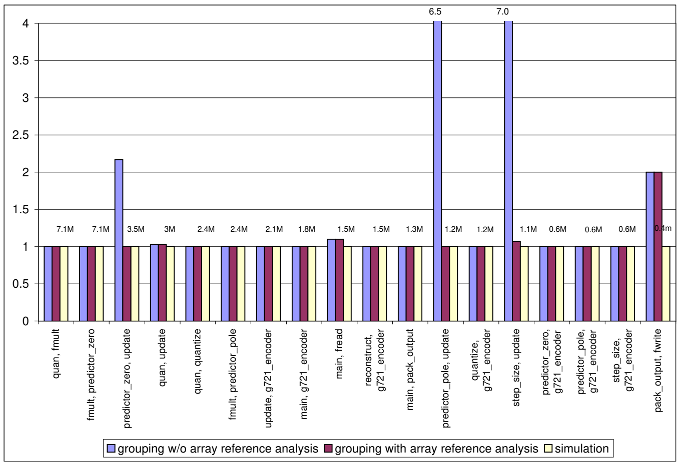


**Figure 14: G721: encode**

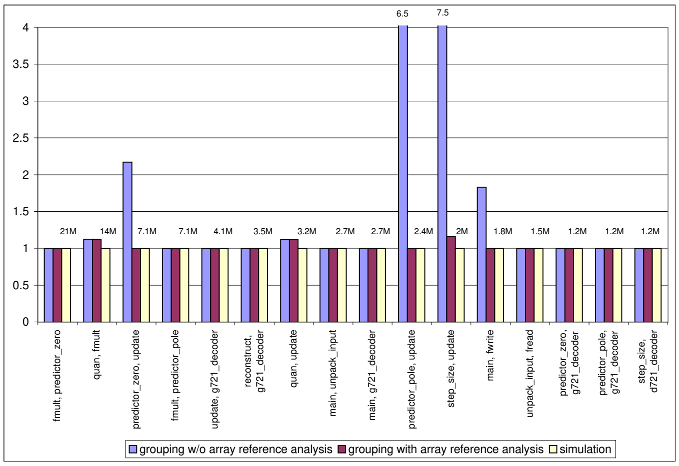


**Figure 15: G721: decode**

data communication by using the simulation results as the base and mark the actual data communication of simulation on each procedure pair. The number shown for each pair of procedures is the sum of data communication going each way. In each figure, we arrange procedure pairs in the order of data communication estimated by simulation.

Fig. 12 and 13 show the interprocedural data communication in program rawcaudio and rawdaudio of ADPCM. Our program analysis achieves results close to the simulation results and the difference is less than 1%. Fig. 14 and 15 show the interprocedural data communication in program encode and decode of G721. We obtain precise results except for one case of communication overcounting for one pair of procedures. The reason for that overcounting is due to an IF statement (which contains the procedure invocation) with a branching frequency of 50%. Fig. 16 and 17 show the interprocedural data communication in program EPIC and UNEPIC. The overcount for the pair of $internal\_filter$ and $internal\_transpose$ is partly due to the IF statement and partly due to the imprecision in the array reference analysis which over-conservatively estimates the overlap between the read and the write references. The communication between the system function $fwrite$ and $encode\_stream$ is also overcounted because we have not applied array reference analysis to system functions. Fig. 18 and 19 show the interprocedural data communication in program toast and untoast of GSM. For each program of toast and untoast, data communication exists between several hundred pairs of procedures. We choose top 20 procedure pairs to show, according to the data communication evaluated by simulation. The overcount of data communication for some of the pairs is due to insufficient array killing information in our current implementation.

## 5. RELATED WORK

Li, Wang and Xu [9] propose a partition scheme for computation offloading based on profiling information on computation time of procedures and interprocedural data communication. Their scheme can result in significant energy-saving for half of the multimedia benchmark programs. They did their analysis based on profiling information instead of on static analysis. Although profiling is more accurate than static analysis, profiling information is only valid with respect to the fixed inputs. Our static analysis requires a minimum set of user assertions. If the user assertions are identical for different inputs, the results will not change. To evaluate the precision of our analysis, we feed our analysis results to their model, and the partition results are exactly same as their results performed by profiling data communication.

Kremer, Hicks and Rehg [7] introduce a compilation framework for power management on handheld computing devices through remote task execution. They consider the procedure calls in the main routine as the candidates for remote execution and they evaluate the profitability of remote execution for each individual task separately, thus they only require the data communication between each pair of procedures among the main routine and those called in the main routine. In their paper, they do not mention how they evaluate data communication, although they present a compilation strategy to perform profitability analysis for remote execution.

Rudenko et al. [13] run a series of experiments comparing the power consumption of programs run locally with that of the same program run remotely. They consider offloading the whole program, and data communication only from input and output.

Lim et al. [11] present an affine partitioning framework to maximize parallelism while minimizing communication in programs with arbitrary loop nestings and affine data accesses. They use a greedy algorithm to minimize communication between loops.

## 6. CONCLUSIONS

In this paper, we have presented a program analysis, in conjunction with few and simple user assertions, to eval-

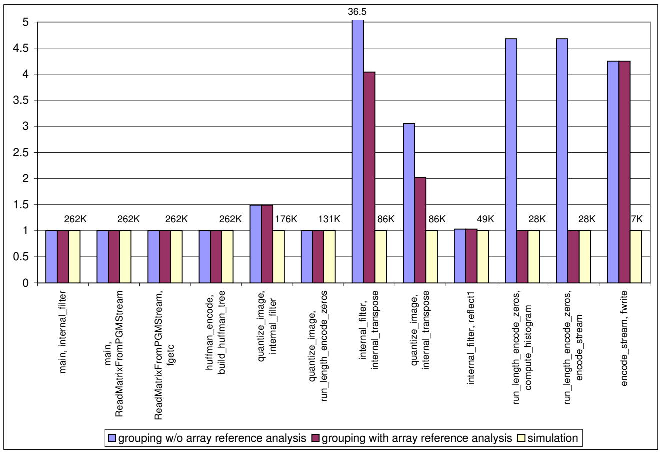


**Figure 16: EPIC**

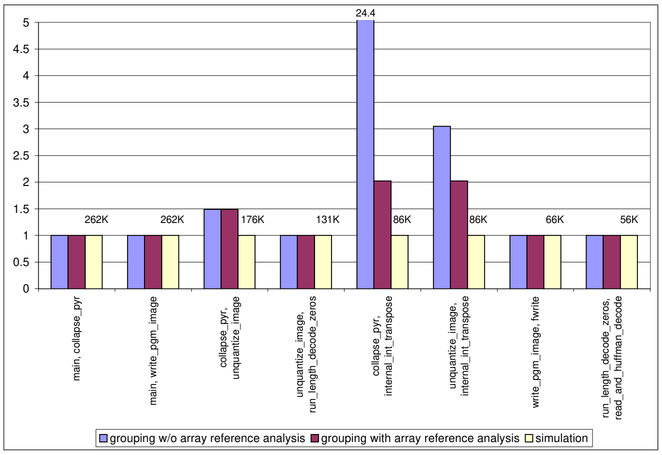


**Figure 17: UNEPIC**

uate interprocedural data communication. To reduce the overcount of data communication, we applied the sequence grouping and branch grouping. The experimental results show that the analysis is rather precise when it is aided by array reference analysis. However, we believe the analysis can be further improved by collecting accurate array section reaching definition information. We plan to integrate an array data flow analysis into our framework as the next step.

## 7. ACKNOWLEDGMENT

The authors thank Cheng Wang and Rong Xu for their helpful discussions and collecting part of the experimental data.

## 8. REFERENCES

[1] K. Chanchio and X. Sun. Data collection and restoration for heterogeneous process migration. *Software Practice and Experience*, 32(9), 2002.

[2] R. Cmelik and D. Keppel. Shade: A fast instruction set simulator for execution profiling. *Sun Microsystems Inc., Technical Report SMLI TR93-12*, 1993.

[3] H. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to algorithms. *MIT Press and McGraw-Hill*, Second Edition, 2001.

[4] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), 1991.

[5] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), January 1990.

[6] D. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4), July 1996.

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

[8] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *30th Annual International Symposium on Microarchitecture (Micro'97)*, 1997.

[9] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: A partition scheme. *Proc. of International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, November 2001.

[10] Z. Li, C. Wang, and R. Xu. Task allocation for distributed multimedia processing on wirelessly networked handheld devices. *Proc. of 16th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.

[11] A. Lim, G. Cheong, and M. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. *Proc. of 13th International Conference on Supercomputing*, June 1999.

[12] H. Pande, W. Landi, and B. Ryder. Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5), May 1994.

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

[14] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[15] M. Wolfe. High performance compilers for parallel computing. *Addison-Wesley Publishing Company*, 1996.

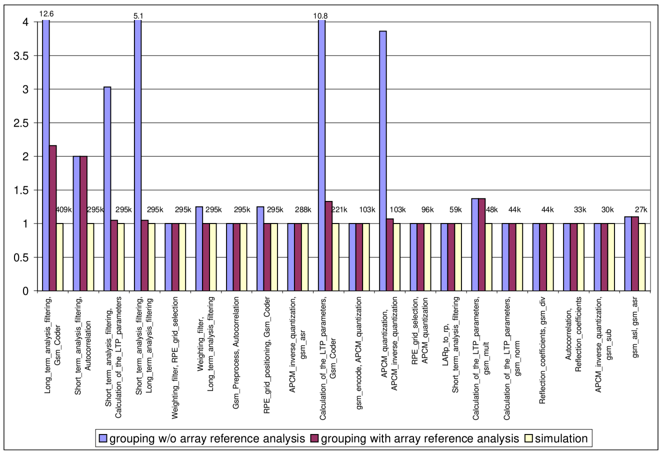


**Figure 18: GSM: toast**

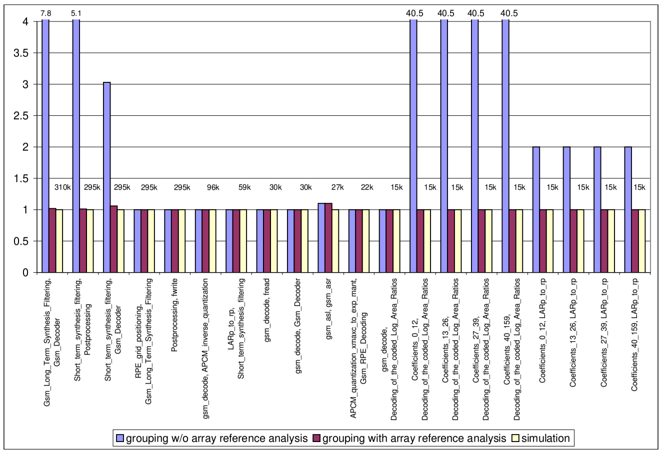


**Figure 19: GSM: untoast**