A SOURCE-LEVEL TRANSFORMATION FRAMEWORK FOR RPC-BASED DISTRIBUTED PROGRAMS

Tae-Hyung Kim
James M. Purtilo

Institute for Advanced Computer Studies
and Computer Science Department
University of Maryland
College Park, MD 20742

Abstract:

The remote procedure call (RPC) paradigm has been a favorite of programmers who write distributed programs because RPC uses a familiar procedure call abstraction as the sole mechanism of operation. The abstraction helps to simplify programming tasks, but this does not mean that the resulting program's RPC-based flow of control will be anything close to ideal for high performance. The purpose of our research is to provide a source-level transformation framework as an alternative way to implement an RPC-based distributed program, so that the code can be optimized through program analysis techniques. This paper describes the transformation tools we have constructed towards this end.

This research has been supported by Office of Naval Research and a grant from the Siemens Corporation.

1 INTRODUCTION

As communication in the LAN-based networks of workstations gets faster, such systems are becoming viable environments for running parallel applications. Even though still there is an order of magnitude difference in the speed of latency and transmission rate from tightly coupled parallel machines, the network of workstations has two noteworthy advantages. First, it provides an opportunity to have a parallel machine virtually with no extra costs. Workstations are ubiquitous and most of them have been underutilizing at most of times. Second, the virtual parallel machine can be constructed so as to take advantage of some special resource locally available on some of the network hosts, for example graphic processors or vector processors.

In either case, the task of creating software for such a system is far more demanding than for a single host. Two representative programming models for distributed memory machines are available for programmers, message passing (MP) and distributed share memory (DSM). Message passing primitives [12, 22, 24] are expressive enough to program for efficiency; however, they are too low-level to write large distributed programs. Programmers are fully responsible for matching send/receive pairs, allocating buffers, and marshaling/unmarshaling data correctly. Programming under DSM systems [9] eases such difficulties, but the resulting programs suffer efficiency due to false sharing and coherence maintaining overhead. (Some researchers claimed that DSM would be more efficient in some particular applications that have irregular communication patterns [23].)

Our work presented in this paper is an effort to strike a compromise between these two models, using the RPC paradigm for writing distributed programs plus a source transformation framework for improving performance. Procedure call abstraction has been favored since early programming era because it contributes to construct a well structured modular program, which allows to reuse existing modules and helps write and maintain a large program by giving a clear view of its structure. The RPC paradigm adopts a widely used and understood procedure call abstraction as the sole mechanism of remote operations; thus it simplifies distributed programming by abstracting from details of communication and synchronization.

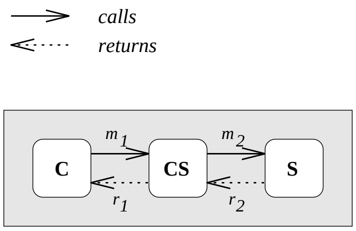
In fact, a distributed program is usually written by a number of different abstraction layers. It is natural to implement each layer of abstraction as a distinct module (or procedure). Following such a natural flow of concept would help write large distributed programs. However, this fact does not necessarily mean that the ideal flow of resulting program for high performance should be consistent with the conceptual flow of RPC paradigm. Two problems should be addressed to adopt RPC paradigm for high performance distributed programming. First, the parallelism is inhibited under the paradigm since the caller blocks until the requested service is finished while we want to make use of the time between sending requests and getting the responses back. Second, an unnecessary communication is likely to occur especially when a system is layered and implemented on the basis of modularization. For example, the communication between far distant layers might require a series of communications between a series of adjacent layers.

The main problem of traditional stub generation based methods [6, 13] for implementing RPC paradigm is that it just adopts the natural flow of modularization as its actual flow of the program. We observe low performance of executing program in this way because the ideal program flow does not conform to the flow of modularization, and the situation is aggravated especially if data transmission and communication latency is getting significant unlike conventional procedure calls in sequential programming.

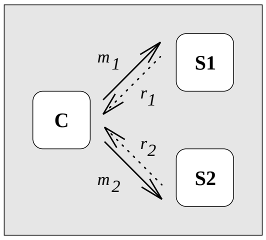
To cope with the discrepancy between the conceptual flow to write a program and the ideal flow to run a program, we present a source transformation framework for RPC-based distributed programs, which is intended for program optimization. It has several advantages over conventional stub generation: (1) it can be safely parallel – correctness is kept because it is transformed under preserving given dependence constraints, (2) using fine grained message passing primitives to implement an RPC statement gives an opportunity for further code optimization through static program analysis techniques, and (3) modularization is not discouraged because the actual communication paths will be restructured optimally based on the given control and data dependences rather than the modular structure as written. Communication optimization has been an important issue to compiling SPMD (Single Program Multiple Data) programs where communication is necessary to access non-local data [2, 25]. Our work is focused on optimization of general RPC-based distributed programs which is not of SPMD form.

2 MOTIVATION

This section illustrates situations to motivate source-level transformation of RPC-based distributed programs for higher performance. The RPC paradigm adopts the model of client-server computing; caller and callee correspond to client and server, respectively. Traditional researches



(a) Parallelism in depth



(b) Parallelism in breadth

Figure 1: Basic topologies in client-server model from the perspective of optimization.

on improving RPC program performance have focused on reducing latency and transmission time within this pairwise form [16, 5, 14].

When this simple topology extends to a *network* of client-server model computing, more advanced optimization other than just efficient pairwise hooking between client and server is called for. Figure 1 shows two basic topologies to form an application of networked client-server computation. These two topologies are basic units where we can account for our source-level transformation techniques for higher performance. The optimization goals are "enhancing parallelism" (**Goal 1**) and "minimizing communication overhead" (**Goal 2**).

In Figure 1 (a), the client module **C** calls its server **CS** and it successively calls **S** to fulfill **C**'s request. The module **CS** plays both roles respectively to the module **C** and **S**. We name it *parallelism in depth* because the parallelism stretches in depth in the calling graph; in other words, the parallelism exists merely in the direction of depth because a client has a single server. Normally, **C** and **CS** are blocked when **S** is working on. Parallelism in depth (**Goal 1**) can be exploited only if communication can be overlapped with client's computation. Towards **Goal 2**, the performance can be improved if we can establish a direct message passing path between an indirect client-server relationship like that of **C** and **S**. Direct message passing means that the recipient can get the data earlier, and consequently, it can start what it is supposed to do earlier. We need to assure that m_2 is equal to m_1 to make a direct message passing between **C** and **S**.

In Figure 1 (b), the client module **C** calls the server **S1** and then **S2**. We name it *parallelism in breadth* because the parallelism exists in the direction of breadth in the calling graph; it implies that multiple servers are associated with a client. Normally, **S1** and **S2** cannot run in parallel due to synchronous nature of RPC. However, the parallel execution of the two servers should be allowed, unless there is a data dependence between **S1** and **S2**. For the special case that the value of $r1$ is equal to the value of $m2$, we can establish a direct message passing path between **S1** and **S2**, which is for the **Goal 2**. This can be a significant improvement for certain cases like when **C** calls **S1** and **S2** in a same loop.

The purpose of this paper is to present an analysis technique for determining whether a programmer's optimization goals can be exploited, while at the same time allowing that programmer to express his application structure in terms of RPCs. Our technique transforms an RPC statement into a set of low-level message passing primitives to enable an RPC-based distributed program to run in an optimized way. Though not discussed in this paper, an experimental environment implementing these techniques has been successfully implemented in an environment called CORD [17].

3 CONSTRAINTS ON SOURCE TRANSFORMATION FOR RPC

Exploitation of parallelism is limited by data and control dependences in the program and resource constraints of the executing environment. Dependence constraints are directly related to the semantics of a program. Executing dependent statements simultaneously or in different order may change the original semantics of a program. Resource constraints limit the maximum degree of parallelism that can be attained when the degree of potential parallelism is higher than the number of available resources. Program transformation to improving the performance must be guided by given program dependence and resource constraints.

In Figure 2 (a), the execution order between S_1 and S_2 must be preserved because S_2 uses the value of `x` which is defined by S_1. This is a flow (true) data dependence denoted by $S_1\ \delta\ S_2$. The order between S_2 and S_3 must be also preserved, otherwise the value of `z` at S_2 may be changed by S_3. This is an anti dependence denoted by $S_2\ \delta^{-1}\ S_3$. The order between S_1 and S_4 must be preserved as well because they have same variable `x` to store the results. This is an output dependence denoted by $S_1\ \delta^{o}\ S_4$. Anti and output dependences are "spurious" ones because they can be disappeared if we rename the associated variables properly; for example, `z` in S_3 to `zz` and `x` in S_4 to `xx` like in Figure 2 (b). Such a renaming releases the imposed execution order constraints by spurious dependences, as a result, executing them in parallel is possible.

S_1: `x = f(x);`
S_2: `y = g(x, z);`
S_3: `z = h(v);`
S_4: `x = h(w);`

(a) Original code

S_1: `x = f(x);`
S_2: `y = g(x, z);`
S_3: `zz = h(v);`
S_4: `xx = h(w);`

(b) After renaming

S'_1: `Send(f(), x);`
S'_2: `Send(h(), v);`
S'_3: `Send(h(), w);`
S'_4: `x = Receive(f());`
S'_5: `Send(g(), x, z);`
S'_6: `y = Receive(g());`
S'_7: `z = Receive(h());`
S'_8: `x = Receive(h());`

(c) After transformed

Figure 2: Eliminating spurious data dependences for parallelization

Suppose there is a sequential (client) program, where two arbitrary statements S_1 and S_2 are totally ordered with respect to $\prec$: i.e. $S_1 \prec S_2$ denotes S_1 is executed before S_2. The task of parallelization can be rephrased by that of converting the total ordering $\prec$ into a partial ordering $\prec_P$ under certain semantic-preserving constraints as follows. The relation $\prec_P$ is an irreflexive partial ordering[1] defined as follows.

1. If S_1 is executed before S_2, then $S_1 \prec_P S_2$.
2. If $S_1 \prec_P S_2$ and $S_2 \prec_P S_3$, then $S_1 \prec_P S_3$.

If two statements, S_1 and S_2, are not related by the $\prec_P$ relation, then we say these two statements can be executed in parallel. If, however, $S_1 \prec_P S_2$, then it is possible for statement S_1 to causally affect statement S_2.

When we transform RPC statements into statements of message passing primitives, we have following ordering relation per RPC due to the *law of causality* by which a reply can be received only after the proper request has been sent out. Suppose that a client program has a set of RPC statements R:

$$\forall S \in R : S \overset{\text{transform}}{\Longrightarrow} S^{snd} \prec_P S^{recv} \tag{1}$$

The statement S^{snd} only *uses* variables whereas the S^{recv} only *defines* variables. This behavioral difference in accessing variables between S^{snd} and S^{recv} helps to widen the gap by placing S^{snd}

[1]It resembles the *happened–before* relation on a set of distributed events [18]. While the associated events in that relation are distributed, the relation $\prec_P$ is an ordering between statements in a single program. Since a statement cannot be executed before itself, it is irreflexive.

as early as possible and placing S^{recv} as late as possible under following constraints (2), (3), and (4) if there are any data dependences on other RPCs. It practically implies that other useful statements can be executed during a wait for an RPC statement.

We assume that the execution time of a statement in a single program is negligible compared to the time for S^{recv}, which is for server processing time plus communication time to get back to the client. We even ignore the time for a program to finish S^{snd} assuming that the underlying message passing system immediately takes the control after executing S^{snd} to complete the send. For example, a relation like $S_1^{snd} \prec_P S_2^{snd}$ is of no significance, and is regarded as parallel. Consequently, the outstanding number of S^{snd} implies the potential degree of parallelism in an RPC-based distributed program. This implies that we do not need any special constructs like **parbegin** and **parend** to express parallelized form after source transformation. The statement S^{recv} is a blocking one with non-negligible executing time because the message becomes ready to receive only after passing considerable amount of time for remote computation and message transmission over the network. Therefore, S^{recv} is the only order preserving primitive to maintain program semantics imposed by data dependences like δ, δ^o and δ^{-1}, as follows:

$$\forall S_1, S_2 \in R: \quad S_1 \ \delta \ S_2 \ \wedge \ S_1 \prec S_2 \Rightarrow S_1^{recv} \prec_P S_2^{snd} \tag{2}$$

$$\forall S_1, S_2 \in R: \quad S_1 \ \delta^o \ S_2 \ \wedge \ S_1 \prec S_2 \Rightarrow S_1^{recv} \prec_P S_2^{recv} \tag{3}$$

$$\forall S_1, S_2 \in R: \quad S_1 \ \delta^{-1} \ S_2 \ \wedge \ S_1 \prec S_2 \Rightarrow S_1^{snd} \prec_P S_2^{recv} \tag{4}$$

The ordering imposed by (2) serializes two RPCs because we get an execution sequence of $S_1^{snd} \prec_P S_1^{recv} \prec_P S_2^{snd} \prec_P S_2^{recv}$ by combining with (1). It is inevitable due to the true data dependence between the two RPCs. However, the restrictions of (3) and (4) apparently do not inhibit parallelism because S_1^{snd} and S_2^{snd} that trigger the server computations are still independent and can run simultaneously.

Suppose that all function calls in Figure 2 are remote ones. Each remote procedure is a unit of our parallelization; in other words, a client module is called "parallelized" if it can call more than two remote functions at the same time by sending proper requests to distinct sites. Figure 2 (c) shows the transformed code by means of *send* and *receive* message passing primitives. In summary, the constraint (1) produces relations of $S_1' \prec_P S_4'$, $S_2' \prec_P S_6'$, $S_3' \prec_P S_7'$, and $S_5' \prec_P S_8'$; (2) imposes $S_4' \prec_P S_5'$ because of $S_1 \ \delta \ S_2$; (3) imposes $S_4' \prec_P S_8'$ because of $S_1 \ \delta^o \ S_4$; (4) imposes $S_5' \prec_P S_7'$ because of $S_2 \ \delta^{-1} \ S_3$. As a result, three RPCs in statements S_2, S_3 and S_4 run in parallel while all

RPCs in statements S_1–S_4 are executed sequentially since they are totally ordered with respect to $\prec$ in the original code.

When a server is replicated to release resource constraints, the regarding concern is to balance the load with respect to all available server processes so that there are no idle ones while others are busy. If we do not consider server process migration among processors, this is for the case of balancing requests to a set of replicated server processes of identical functionality. For example, if a server function like `h()` in Figure 2 (c) is replicated, two requests to the same remote function `h()` at S_2' and S_3' should be sent to distinct sites for the purpose of load balancing. Such an issue has been discussed from the perspective of configuration level optimization in a separate paper [17]. In the following section, we present a source transformation framework under constraints (1)–(4) for optimizing RPC-based distributed programs, with preserving the control dependences as well.

4 TRANSFORMATION FRAMEWORK

When an RPC is implemented through traditional stub generation based methods [6, 8, 13], a stub performs following three functions.

1. **Communication:** RPC arguments are transmitted to the "remote" callee via communication network, and the result is back to the caller.
2. **Synchronization:** The caller is suspended until the result is ready to receive.
3. **Data conversion:** A data type in a caller machine needs not to be identical to the "remote" machine. For flexibility as well as convenience[2], a data type is converted into a standardized type like XDT [10] (*encoding*) before converted into a specific type (*decoding*).

Since those three functions can be implemented by a combination of message passing primitives that are provided by underlying MP systems [12, 24] or (distributed) operating systems, we approach the transformation problem from RPC statements to a series of such low-level primitives

[2]Without having external data conversion, if L different languages and M different machines are intermixed in a distributed application, then potentially $(L \times M)^2$ cases of data conversion must be used [13].

in a static manner. The transformation based RPC implementation approach opens an opportunity to apply various static analysis techniques for optimizing RPC-based distributed programs, which have been shown useful for automatic parallelization. As we assemble those MP primitives to implement an RPC statement, which is regarded as a "big" statement, we have freedom to place each low-level primitive appropriately interspersed in a module in order to achieve our aforementioned goals of enhanced parallelism and minimized communication. The transformation merely affects client parts. The transformation is performed at client side to implement its remote procedure call. A server is synthesized to start with *prologue* part that receives various requests from all eligible clients, and to end with *epilogue* part that sends the result to the actual destination(s) rather than its caller (Section 5).

In this section we present the heart of our algorithm, which is our approach to hasten RPC argument passing as early as possible (over the distinct modules), and to delay receiving the return value as late as possible, according to the result of **def** and **use** analysis to the variables involved. We do this in a three-step process. First, all RPCs in an application are enumerated to be "*positionally different*"[3], and represented by a call tree (Section 4.2). Next, **use-def** chains for RPC arguments and **def-use** chains for a return value are evaluated by data flow analysis (Section 4.3). Finally, global optimization is performed to meet our optimization goals. A special consideration on RPC in a loop body is discussed to show the effectiveness of source transformation framework for implementing RPC. That is, to reveal more aggressive transformation techniques to optimize distributed programs. We start our discussion by giving basic definitions for those presentations.

4.1 Definitions

Suppose there is a distributed program P that is composed of k different software modules, $M_1, M_2, \ldots, M_k$ running at distinct sites. $RPC_1, RPC_2, \ldots, RPC_k$ are sets of *positionally different* occurrences of RPCs that are imported in $M_1, M_2, \ldots, M_k$, respectively. We use the terms *module* and *function* interchangeably.

$DUC_m(l)$ (Def-Use-Chain) is a set of reachable uses of a definition to a variable l in a module m. $UDC_m(r)$ (Use-Def-Chain) is a set of reaching definitions of a variable associated with use of a

[3]Even if there is only one imported RPC in a client module, the remote procedure can be called several times at different places in the client. All of these occurrences are for the the same RPC, but they are considered *different* because they may have different data flow in terms of argument passing and result returning; i.e. they are distinguished by position.

variable r in a module m.

Receive_Request(r) denotes a set that contains every sources of arguments, which form a *request* for a remote call r. Let $|r|$ be the number of arguments for r. Then, *Receive_Request*(r) can be written by $\{t_i \mid t_i = (s_i, v_i), 1 \leq i \leq |r|\}$, where s_i is the module that defines the value of the i-th actual argument of the call, and v_i is the variable that contains the value in the module; i.e. $v_i \in UDC_{s_i}(a_i)$ where a_i is the i-thththth actual argument. The default (i.e. unoptimized) state of *Receive_Request*(r) is a set of tuples of an original caller and argument variables because the arguments are expected to be sent from a caller module when r is called. For example, if "`l = f(`$v_1, \ldots, v_n$`)`" is an RPC statement in m, the default contents of *Receive_Request*(r) will be given by $\{m : v_1, \ldots, m : v_n\}$. One of the effects of our optimization algorithm is to change the default states of the tuples; i.e. change of a variable v_i to a reaching definition, or change of a source module m to the more original source that has provided the value to m.

Send_Result(r) denotes a set that contains every recipients of r. Conventionally, this is a singleton as the caller is the only recipient of the return value. It can be written by $\{t \mid t = (d, v)\}$, where d denotes a destination module in $\{M_1, \ldots, M_k\}$ that receives the return value, and v denotes a variable that will contain it. Multiplicity in the set has two implications. First, it provides an opportunity to take advantage of multicasting if available, which is faster than a series of point-to-point communications. Second, the result is delivered directly to a module that uses it.

No need to have *Send_Request*(r) and *Receive_Result*(r) additionally because of the duality between *send* and *receive* primitives.

4.2 Call Tree Construction

Every occurrences of RPC in an application should be distinguished so as to construct their own optimized paths of the data flows. For example, in "`a = f(x); b = f(y)`", the first call to `f()` has a data flow on the variable `x` that is different from that on the variable `y` in the next call, and same for `a` and `b`. We construct a Call Tree (CT) as a way to represent all *positionally different* calls as well as control dependences on them. The call tree is defined as follows.

Definition 4.2 Let P denote an RPC-based distributed program. The call tree of P is an unordered tree $CT = (V, E)$, where

- The vertices V represent a set of modules involved in every RPC statements in P. In addition, there is a distinguished vertex "root", which represents the root of the tasks; it denotes a main program in P. The remaining vertices are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$ where each of these sets is a tree.
- The edges E represent caller–callee relationships. That is, an edge $(v, w) \in E$ means v directly calls w. An edge carrys a control dependence predicate defined in the following. An edge without having a control predicate means control independence (*always true*).

The tree forms a calling hierarchy in a program; the parent node is a caller (client), and the children nodes are its callees (servers). An edge represents a *positionally different* RPC, which conveys flow sensitive information [4, 7, 15]. We also define "$v \stackrel{+}{\Rightarrow} w$" to mean that node v can reach node w via one or more control dependence edges. □

A *control flow graph* [11] is a directed graph $CFG = (CV, CE)$ with unique node $Entry, Exit \in CV$ such that there exists a path from $Entry$ to every node in CV and a path from every node to $Exit$; $Entry$ has no incoming edges, and $Exit$ has no outgoing edges. We will use a module name for the node $Entry$ to avoid multiple $Entry$ nodes appear in CT consisted of multiple modules. An edge in CE is annotated by a control predicate that determines the control flow of the edge. We assume T on single outgoing edge (no branch), which denotes *true*, so it means the edge is always selected from the caller. The $(v - w)$ denotes the control predicate on an edge (v, w) among all outgoing edges from v. If $P_{v_1 v_n}$ is a path from v_1 to a node v_n, which is $<v_1, \ldots, v_n>$, the control predicate that determines the execution of node w, denoted by $Cpred(P_{v_1 v_n})$, is $(v_1 - v_2) \wedge \ldots \wedge (v_{n-1} - v_n)$. If there are n different paths $P_1, \ldots, P_n$ from v_1 to v_n that are all reachable paths from v_1 to v_n, the control predicate for node v_n from v_1 is $Cpred(P_1) \vee \cdots \vee Cpred(P_n)$, where $Cpred(P_i)$ represents a control predicate for path P_i.

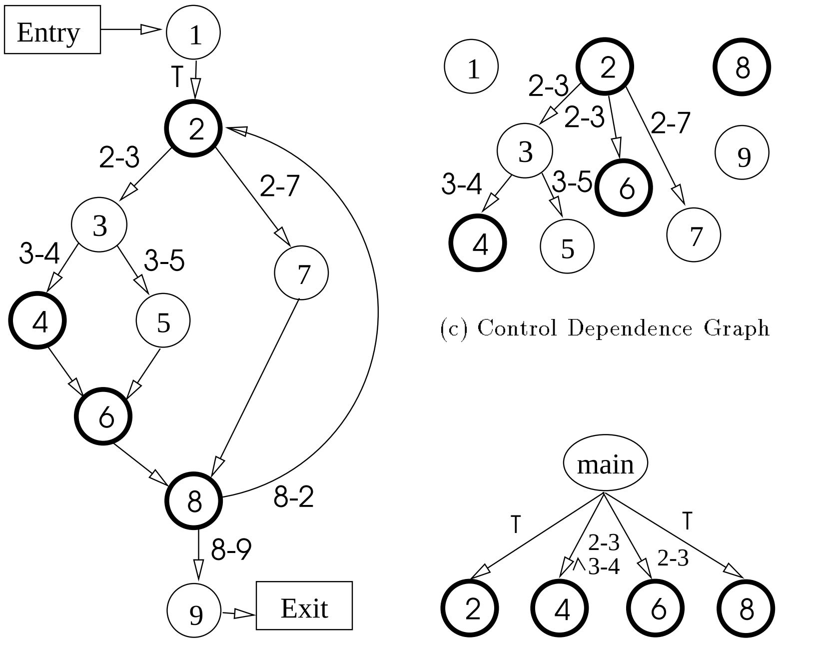
The call tree CT of a sample program along with its CFG and CDG is shown in Figure 3 [4]. The control predicates will be used to construct optimized server module with low-level message passing primitives in the following section. The control predicates can be evaluated through either CFG or CDG. Following example illustrates how to evaluate the control predicates on edges in CT from CFG in Figure 3.

Example:

[4] In the SSA representation of the figure, a join node for the loop construct is omitted for brevity.

```
main()
{
[1]; /* defs on x1, y1, z1 */
loop {
    [2]: y2 = f1(x1);
    if ([2] - [3]) {
        [3];
        if ([3] - [4])
            [4]: z2 = f2(y2);
        else
            [5];
        [6]: w2 = h(x1);
    }
    else if ([2] - [7])
        [7];
    [8]: v2 = g(φz(z1, z2));
}
printf(y, z, w, v);
}
```

(a) An example



(b) Control Flow Graph

(c) Control Dependence Graph

(d) Call Tree

Figure 3: An example: CFG and CDG to construct Call Tree.

$$
\begin{aligned}
Cpred(Entry - \boxed{4}) &= Cpred(< E, 1, 2, 3, 4 >) \;\vee\; Cpred(< E, 1, 2, 3, 5, 6, 8, 2, 3, 4 >) \;\vee \\
&\quad\; Cpred(< E, 1, 2, 7, 8, 2, 3, 4 >) \\
&= [(2-3) \wedge (3-4)] \;\vee\; [(2-3) \wedge (3-5) \wedge (8-2) \wedge (3-4)] \;\vee \\
&\quad\; [(2-7) \wedge (8-2) \wedge (2-3) \wedge (3-4)] \\
&= [(2-3) \wedge (3-4)] \;\vee\; [(2-7) \wedge (8-2) \wedge (2-3) \wedge (3-4)] \\
&= (2-3) \wedge (3-4) \qquad\qquad [\text{OR}-\text{simplification on } (2-3) \wedge (3-4)]
\end{aligned}
$$

The above method requires finding all reachable paths and simplifying boolean expressions; it is computationally expensive. Control dependence [11] captures the essential control flow relationships in a program. Informally, for node v and w in CFG, w is *control dependent on* v if v can directly affect whether w is executed or not. Thus, if nodes v, w are control independent with each other ($(v, w) \notin E$ in CDG) or they are control dependent on same other node x with same label on it (for example, $\boxed{3}$ and $\boxed{6}$ with label of (2–3)), they are free to be executed in parallel, unless there are data dependences between these nodes. The control predicate for a node

v in CT is always from the entry point of the module, which calls the remote procedure in the node v, to the node v: i.e., $Cpred(Entry, v)$. It can be determined by taking control labels of all of its ancestors in CDG. All reachable paths to the node (say, $p_1, \ldots, p_n$) are sought first in CDG. Then, similarly as in case of using CFG, $Cpred(Entry, v)$ is $Cpred(p_1) \vee \cdots \vee Cpred(p_n)$, where $Cpred(p_i)$ is $(w_1 - w_2) \wedge \cdots \wedge (w_{n-1} - w_n) \wedge (w_n - v)$, if the path p_i is $<w_1, \ldots, w_n, v>$. Obviously, if there is no reachable path in CDG, it gives T. Simplification of the expression is not necessary because the control dependence information is already summarized in CDG; for instance, $Cpred(Entry - 4)$ in CT is $(2-3) \wedge (3-4)$ as there is only path $<2, 3, 4>$ in the CDG.

A simple algorithm to construct a call tree is given below.

Algorithm 4.2 *Call Tree Construction*

Input:

1. All involved modules $M_1, \ldots, M_k$
2. All $r \in RPC_1 \cup \cdots \cup RPC_k$ where $server(r), client(r) \in \{M_1, \ldots, M_n\}$

Output: $CT = (CV, CE)$ as defined in Definition 4.2.

Begin
    $E = \phi$;
    $V = \{M_1\}$;
    Let RPC_1 **be** $\{v_1, \ldots, v_m\}$;
    $V = V \cup RPC_1$;
    $E = E \cup \{(M_1, v_1), \ldots, (M_1, v_m)\}$;
    Evaluate control predicates for the new edges;
    For all $v \in RPC_1$
        $Compute(v)$;
End

function $Compute(r)$ {
    $s = server(r); c = client(r)$;
    $R = RPC_s$;
    Let R **be** $\{w_1, \ldots, w_n\}$;
    $V = V \cup R$;
    $E = E \cup \{(c, w_1), \ldots, (c, w_n)\}$;
    Evaluate control predicates for the new edges;
    For all $w \in R$
        $Compute(w)$;
}

4.3 Initialization

Suppose R is an RPC statement that we want to translate into R^{snd} and R^{recv}. Before global optimization, we need to characterize the set of proper data dependences for variables involved in R^{snd} and R^{recv} in a compact way through data flow analysis. In effect, it initializes $Receive\_Request(R)$ and $Send\_Result(R)$. Let $t_{rr} = (s_{rr}, v_{rr})$ and $t_{sr} = (d_{sr}, v_{sr})$ be elements of $Receive\_Request(R)$ and $Send\_Result(R)$, respectively. Since the initialization phase works within a module, the s_{rr} and d_{sr} are not changed after initialization. But v_{rr} is replaced with the latest definition that reaches to its use as an argument, and v_{sr} is replaced with the earliest use that is reached by the return value of the RPC. This is to widen the gap between the R^{snd} and R^{recv}; in effect, we can eliminate the spurious data dependences in this way as illustrated in Section 3. The more the gap is attained, the more statements can be executed during executing an RPC.

Node v *dominates* node w, denoted by $v \Delta w$, if v appears on every path from $Entry$ to w [1]. Node v *immediately dominates* node w iff $v \Delta w$ and there is no node x such that $v \Delta x$ and $x \Delta w$. In a dominator tree (DT) of a CFG, the children of a node v are all immediately dominated by v. When v is a closer descendent to x than y in the DT, the dominator x is called *closer* to v than y. Node v *post-dominates* node w, denoted by $v \Delta_p w$, if v appears on every path from w to $Exit$ [11]. Node v *immediately post-dominates* node w iff $v \Delta_p w$ and there is no node x such that $v \Delta_p x$ and $x \Delta_p w$. In a post-dominator tree (PDT), the children of a node v are all immediately post-dominated by v. When v is a closer descendent to x than y in the PDT, the post-dominator x is called *closer* to v than y. Then, the effect of initial transformation is described concisely as follows, where $\{d_1, \ldots, d_m\}$ is a UDC set for an argument variable in R, and $\{u_1, \ldots, u_n\}$ is a DUC set for an l-value of R:

Property 1 *R^{snd} is the closest common post-dominator to $\{d_1, \ldots, d_m\}$.*

Property 2 *R^{recv} is the closest common dominator to $\{u_1, \ldots, u_m\}$.*

In case that UDC or DUC is empty, the location of R^{snd} or R^{recv} has no restriction in terms of data dependences; the former could be an error, and the latter will be treated in global optimization. As the properties describe, the algorithm to find such R^{snd} and R^{recv} is straightforward; i.e. compute the proper UDC and DUC sets [1], and find the least common ancestors for those elements of the sets, in the PDT and DT, respectively, and repeat for the next RPC. Notice that the R^{snd} and R^{recv} locations of the current RPC must be determined before computing UDC and DUC sets for the next RPC, because they may affect those sets for the following RPCs. Now we want to assure that the algorithm satisfies the semantic-preserving constraints in (1)–(4) as shown in Section 3.

Theorem 4.1 *Property 1 and 2 satisfy the Constraint (1) in Section 3.*

Proof: Obvious. □

Theorem 4.2 *Property 1 and 2 satisfy the Constraint (2) in Section 3.*

Proof: Suppose S_1 is data dependent on S_2 w.r.t a variable x. By **Property 1**, S_2^{snd} follows any reaching definitions on x, obviously including the definition by S_1^{recv}. If S_2^{snd} has to precede S_1^{recv}, S_2 must not be a reachable use from S_1, by **Property 1**, or equivalently, S_1 must not be a reaching definition to S_2, by **Property 2**, both of which contradict the data dependence between S_1 and S_2. □

Theorem 4.3 *Property 2 satisfies the Constraint (3) in Section 3.*

Proof: Suppose S_1 is output dependent on S_2 w.r.t a variable x. Let x_1, x_2 be the l-values of the definitions by S_1, S_2, respectively. Suppose there exists a $u \in DUC(x_1)$ such that it is preceded by one of $DUC(x_2)$ in the CFG. Then it means that the use u is preceded the definition of x_2, i.e. the definition of x_1 is killed by x_2 at this point. This is impossible because v must be in $DUC(x_2)$ then. Thus, all members of $DUC(x_1)$ precede those of $DUC(x_2)$. That is, the maximum depth of $DUC(x_1)$ is shallower than the minimum depth of $DUC(x_2)$ in the DT. Therefore, the least common ancestor node of $DUC(x_1)$, which is S_1^{recv}, precedes the least common ancestor node of $DUC(x_2)$, which is S_2^{recv}, in other words, $S_1^{recv} \prec_p S_2^{recv}$. □

Theorem 4.4 *Property 1 and 2 satisfy the Constraint (4) in Section 3.*

Proof: Suppose S_1 is anti dependent on S_2 w.r.t a variable x. Let x_{old} be the used variable in S_1^{snd}. Let x_{new} be the l-value of the new definition by S_2^{recv}. Suppose that $S_1^{snd} \prec_p S_2^{recv}$ cannot be satisfied by the **Property 1**; i.e., $S_2^{recv} \preceq_p S_2^{snd}$ is possible after the transformation. To make it possible, some uses in $DUC(x_{new})$ must precede (for '$\prec$') or be equal to (for '$=$') some definitions in $UDC(x_{old})$. This is impossible, by the definitions of UDC and DUC sets. □

Further optimization, which starts from the initialized sets, extends the idea of "widening gap between the statements of R^{snd} and R^{recv}" to a global (interprocedural) level. It may change the conventional call/return pattern because the third module may send a request or receive a result to pursue an optimal data flow path.

4.4 Global Optimization

This phase is to seek a direct message passing path that could be a series of message passing at the interprocedural level. Sending out a message m at a module y to a module z, if that is sent by a module x, is an unnecessary communication because it can be replaced with a direct communication between y and z: i.e. replacing $x \rightarrow y \rightarrow z$ with $x \rightarrow z$. To make this simplification possible, we have to know that the message m is not killed at y before sending out to z and not used for other purpose than relaying to z. Even if m is used at y, seeking a direct message passing path between x and z is still worthwhile because z can receive it earlier than being sent via y. From the viewpoint of each procedure, the interprocedural data flow equations to this end can be expressed as following recursive forms where the $\textsf{Called}(P)$ is the set of remote procedures called directly from P [4]:

$$Use(P) = LocalUse(P) \bigcup_{Q \in \textsf{Called}(P)} Use(Q) \tag{5}$$

$$Def(P) = LocalDef(P) \bigcup_{Q \in \textsf{Called}(P)} Def(Q) \tag{6}$$

Since there are no global or reference variables, we can rewrite those equations as following concrete forms:

$$Use(P) = LocalUse(P) \bigcup RetUse(P) \bigcup_{Q \in \textsf{Called}(P)} Call(Q) \tag{7}$$

$$Def(P) = LocalDef(P) \bigcup ArgDef(P) \bigcup_{Q \in \textsf{Called}(P)} Return(Q) \tag{8}$$

A local use is a use that is not used as an argument in an RPC statement or as a return value. Non-local uses are two kinds; $RetUse(P)$ is a use for returning a value (single variable return only), and $Call(Q)$ is a set of variables that are used in calling a remote procedure Q. A local definition is a definition that is not defined by an RPC or by argument passing. Non-local definitions are two kinds; $ArgDef(P)$ is a definition that is passed through an argument, and $Return(Q)$ is a definition by the return value of of a remote procedure Q.

On the other hand, from the viewpoint of each RPC, where we are interested in seeking *true definitions* and *true uses* associated with the call, the $Receive\_Request(r)$ and $Send\_Result(r)$

sets can be defined as follows:

$$Receive\_Request(r) = RR_{server(r)}(M_1) \cup \cdots \cup RR_{server(r)}(M_k) \tag{9}$$

$$Send\_Result(r) = SR_{server(r)}(M_1) \cup \cdots \cup SR_{server(r)}(M_k) \tag{10}$$

$RR_{server(r)}(M_i)$ is a set of variables that are defined at M_i in order to be used at $server(r)$ module. $SR_{server(r)}(M_i)$ is a set of uses of a return value of $server(r)$ at M_i. Recalling the message passing for argument(s) and return value passing is the only way to interact between distinct modules, these two sets can be defined as follows:

$$RR_s(c) = \begin{cases} Use(s) \cap Def(c) \equiv Call(s) \cap Def(c) & \text{if } c \text{ calls } s \text{ directly} \\ RR_s(t_1) \cap RR_{t_1}(t_2) \cap \cdots \cap RR_{t_n}(c) & \text{if } c \stackrel{+}{\Rightarrow} s\ (<c, t_1, \ldots, t_n, s>) \\ \phi & \text{otherwise} \end{cases} \tag{11}$$

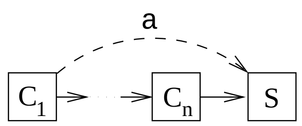
$$SR_s(c) = \begin{cases} Def(s) \cap Use(c) \equiv Return(s) \cap Use(c) & \text{if } c \text{ calls } s \text{ directly} \\ SR_s(t_1) \cap SR_{t_1}(t_2) \cap \cdots \cap SR_{t_n}(c) & \text{if } c \stackrel{+}{\Rightarrow} s\ (<c, t_1, \ldots, t_n, s>) \\ \phi & \text{otherwise} \end{cases} \tag{12}$$

$Use(s)$ in Eq. (11) can be replaced with $Call(s)$ because the passed arguments are the only variables that are used in the server module s, providing that there are no aliasing and reference variables. Similarly, $Def(s)$ in Eq. (12) can be replaced with $Return(s)$, because the return value is the only definition that can be defined by a remote procedure s. Notice that Eq. (11) and Eq. (12) are dual only if $Def(c)$ in Eq. (11) is $Return(s')$ and $Use(c)$ in Eq. (12) is $Call(s')$ (i.e. all other terms are null in Eqs. (7), (8)), which implies that a return value of an RPC s' is used to call s when c calls s.

Consequently, if we compute each *definition* and *use* sets as shown in Eqs. (7) and (8), we can compute $Receive\_Request(r)$ and $Send\_Result(r)$ sets adequately. That is, no message is transmitted unnecessarily any more. As the intraprocedural **def-use chains** and **use-def chains** have already been computed in an initialization phase, we are ready to solve Eqs. (9)–(12).

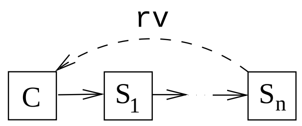
Interprocedural data path may be analyzed if there is a chain of procedure calls, i.e. $c \stackrel{n}{\Rightarrow} s$ when $n > 1$. Solving these equations directly is not realistic, however, they give us an insight to solve them. To see if a solution exists in Eq. (11), we need to check out if an argument is passed

```
C_1()          ...  C_{n-1}(arg_type a)   C_n(arg_type a)   S(arg_type a)
{                   {                     {                 {
...                 ...                   ...               /* do S() */
x = C_2(a);         x = C_n(a);           x = S(a);         return(...);
...                 ...                   ...               }
}                   }                     }
```



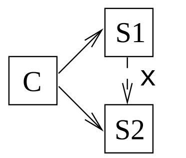
(a) Call path optimization

```
C()             S_1()              ...  S_{n-1}()              S_n()
{               {                       {                      {
...             /* do S_1() */          /* do S_{n-1}() */     /* do S_n() */
x = S_1(...);   rv = S_2(...);          rv = S_n(...);         return(rv);
y = x + 2;      return(rv);             return(rv);            }
...             }                       }
}
```



(b) Return path optimization

```
C()
{
    x = S1(...);
    y = S2(x);
    ...
}
```



(c) Mixed path optimization

Figure 4: Example code shapes for global optimization

without having changed from c to s along the call path. For example, as shown in Figure 4 (a), if the client $\mathtt{C}_n$ sends a value a to the server S and the value a is an input argument provided by its caller $\mathtt{C}_{n-1}$, then the server S can receive the argument value directly from $\mathtt{C}_{n-1}$, and ultimately up from $\mathtt{C}_1$. Similarly, to see the same thing in Eq. (12), we need to check out if a return value from s is returned again in c. As shown in Figure 4 (b), if the client C receives a result from the server $\mathtt{S}_1$ and the value is the return value from its server $\mathtt{S}_2$, then the client C can receive the value directly from $\mathtt{S}_2$, and ultimately from $\mathtt{S}_n$. Finally, if the client C sends a request of a value x for an RPC "$\mathtt{y = S_2(x)}$" and the value is actually defined by another RPC "$\mathtt{x = S_1(\cdots)}$" then the value of x can be directly sent from the module $\mathtt{S}_1\mathtt{()}$ to the module $\mathtt{S}_2\mathtt{()}$ as shown in Figure 4 (c). As this is a mixed case, it is checked by solving $SR_{s1}(c) \cap RR_{s2}(c)$. Notice that the *parallelism in breadth* cannot be exploited as this is the case that $m2$ is data dependent on $r1$ in Figure 1. Interestingly, however, this optimization converts the original program to exploit *parallelism in depth* between $\mathtt{S}_1\mathtt{()}$ and $\mathtt{S}_2\mathtt{()}$ based on the special fact that $r1$ is equal to $m2$.

Algorithm 4.4: *Optimization*
Input:
1. All $r \in Enumerated$ where $server(r) \in \{M_1, \ldots, M_n\}$
2. Initialized $Receive\_Request(r)$ and $Send\_Result(r)$

Output: Optimized $Receive\_Request(r)$ and $Send\_Result(r)$.
Begin
  for each $t_{rr} \in Receive\_Request(r)$ **do**
    while $(val(t_{rr}) \in ArgDef(r, source(t_{rr})))$ **do**
        $r \leftarrow previous(r)$;
        $source(t_{rr}) \leftarrow client(r)$;
        $val(t_{rr}) \leftarrow UDC_{client(r)}(ACTUAL(r, t_{rr}))$;
    endwhile
  endfor /* Call path optimization */
  for each $t_{sr} \in Send\_Result(r)$ **do**
    while $(val(t_{sr}) \in RetUse(r, dest(t_{sr})))$ **do**
        $r \leftarrow previous(r)$;
        $dest(t_{sr}) \leftarrow client(r)$;
        $val(t_{rr}) \leftarrow DUC_{client(r)}(LVALUE(r))$;
    endwhile /* Return path optimization */
    for each sibling edge r_{sib} of a node $dest(t_{sr})$ in CT **do**
        if $(val(t_{sr}) \in Call(r_{sib}, server(r_{sib})))$
            $new\_t_{sr} \leftarrow CreateTupleSR(r)$;
            $dest(new\_t_{sr}) \leftarrow server(r_{sib})$;
            $val(new\_t_{sr}) \leftarrow FORMAL(r_{sib}, t_{sr})$;
            $Send\_Result(r) \leftarrow Send\_Result(r) \cup \{new\_t_{sr}\}$;
            for each $t_{rr} \in Receive\_Request(r_{sib})$ **do**
                if $(val(t_{rr}) \in Return(r, client(r)))$
                    $source(t_{rr}) \leftarrow server(r)$;
                    $val(t_{rr}) \leftarrow RETVAL(r)$;
            endfor
    endfor /* Mixed path optimization */
    if $(t_{sr} \notin LocalUse(r, client(r)))$
        $Send\_Result(r) \leftarrow Send\_Result(r) - \{t_{sr}\}$;
  endfor
End

Figure 5: Global optimization algorithm

Other than seeking a direct path, a message passing path can be eliminated if a return value is not used in the caller module except being used as an argument for another RPC; the corresponding *send_result* and *receive_result* pair collapses. Figure 5 summarizes the algorithm for the global data path optimization. In the figure, $previous(r)$ is a previous edge in CT, which is a previous remote procedure call before r is made.

4.5 Loop Transformation

Many research works have been focused on loop transformations in various parallel compilers,

```
for (i = 0; i < N; i++) {
    a[i] = a[i] + c;
    x[i+1] = callrpc(a[i], x[i]);
}
```

(a) Original loop

⇒

```
for (i = 0; i < N; i++)
    a[i] = a[i] + c;
for (i = 0; i < N; i++)
    x[i+1] = callrpc(a[i], x[i]);
```

(c) After loop distribution

⇓

```
for (i = 0; i < N; i++) {
    a[i] = a[i] + c;
    send_req(callrpc(), a[i], x[i]);
    x[i+1] = recv_res(callrpc());
}
```

(b) Transformed RPC into MP primitives

⇒

```
for (i = 0; i < N; i++)
    a[i] = a[i] + c;
for (i = 0; i < N; i++)
    send_req(callrpc(), a[i], x[i]);
for (i = 0; i < N; i++)
    x[i+1] = recv_res(callrpc());
```

(d) After loop distribution

Figure 6: Loop distribution and call streaming

for loops are hot spots in the program [21]. We are interested in transforming a loop as well, especially when RPC statements are surrounded by a loop construct. Executing an RPC is involved in rather longer delay. Aggregating remote messages can drastically reduce the inter-networking overhead by sharing the overhead by multiple messages. If a loop contains RPCs, the chance to reduce the overhead through aggregation is higher [19], thus careful loop transformation provides good opportunity for aggregation.

Loop distribution (also called *loop splitting* or *loop fission*) breaks a single loop into multiple loops with the same iteration space but each enclosing a subset of the statements in the original loop [20]. It is used to improve instruction and data locality by shortening loop bodies and to allow parallelism that is hindered by loop-carried dependences in the original loop. The latter effect is important in applying the technique to a loop that contains RPC statements. An original loop shown in Figure 6 (a) can be distributed as in (c). It surely eliminates the flow dependence between two statements in the loop, however, the "`callrpc()`" over the iteration space cannot run in parallel in (c) even if we assume there are replicated servers for the procedure, because RPC in each iteration is synchronous. If we transform RPC statements into statements of message passing primitives according to our transformation based approach, the original loop in Figure 6 (a) would be transformed into (b), and then (d) after loop distribution.

Recalling our assumption that S^{snd} takes negligible amount of time, N different calls are able to be placed if there are N servers. Even if only a server is available, the calls can be streamed,

	Receive_Request$_{INI}$	Send_Result$_{INI}$	Receive_Request$_{OPT}$	Send_Result$_{OPT}$
$\boxed{2}$	(main, x_1)	(main, y_2)	(main, x_1)	(main,y_2) & (f_2,Arg1)
$\boxed{4}$	(main, y_2)	(main, z_2)	(f_1, RetVal)	(main,z_2) & (g, Arg1)
$\boxed{6}$	(main, x_1)	(main, w_2)	(main, x_1)	(main,w_2)
$\boxed{8}$	(main,z_1) | (main,z_2)	(main,v_2)	(main,z_1) | (f_2,RetVal)	(main,v_2)

Table 1: Receive_Request and Send_Result sets after initialization and global optimization

so it reduces the cost of transmitting the call and reply messages because the streamed calls and replies can be buffered and sent to allow us to amortize the overhead of kernel calls and the transmission delays over several calls. It is called *call streaming*, which was proposed to effectively support asynchronous calls with an aid of a special data type called "*promises*" [19]. Our method presents a static solution for call streaming without employing special programming language constructs. Moreover, connecting the output of one remote procedure to the input of another is also automatically achieved as presented in the previous section while results must be returned to the original caller before sending them on the next stream in [19].

Data aggregation to amortize kernel overhead and the transmission delays over several calls can be achieved transparently by an aid of underlying MP systems or statically by an aid of compiler that properly generates finer grain MP primitives. For instance, `send_res()` can be composed of finer primitives of `msg_decode(); msg_send()`. Employing an efficient buffer allocation mechanism is encouraged during compiling an RPC program under our transformation based approach.

5 MODULE SYNTHESIS

The synthesis phase involves implementing the transformations resulted from optimizing call behaviors. This must account for correct program behaviors in spite of drastically changed communication paths between caller and callee. As we establish all possible RPC occurrences in CT, the computed control dependences must be properly reflected in module synthesis so that the dependences are preserved at run-time.

Table 1 shows the contents of *Receive_Request* and *Send_Result* sets for each RPC in the example of Figure 3, after initialization in Section 4.3 and global optimization by Algorithm 4.4. All

```
main() {
  [1]; /* defs on x1, y1, z1 */
  loop {
    send_req(f(), x1);
    [2]; /* null */
    if ([2] – [3]) {
      send_req(h(), x1);
      [3];
      if ([3] – [4]) {
        [4]; /* null */
        y2 = recv_res(f());
        send_req(f(), y2);
      }
      else
        [5];
      [6]; /* null */
    }
    else if ([2] – [7])
      [7];
    [8]; /* null */
    if ([2] – [3] ∧ [3] – [4])
      z2 = recv_res(f());
    send_req(g(), φ(z1, z2));
    if ([2] – [3])
      w2 = recv_res(h());
    v2 = recv_res(g());
  }
  printf(y, z, w, v);
}
```

(a) After initialization only

```
main() {
  [1]; /* defs on x1, y1, z1 */
  loop {
    send_req(f(), x1);
    [2]; /* null */
    if ([2] – [3]) {
      send_req(h(), x1);
      [3];
      send_ctrl(f(), [2] – [3]
                   ∧ [3] – [4]);
      if ([3] – [4])
        [4]; /* null */
      else {
        send_req(g(), z1);
        [5]; }
      [6]; /* null */
    }
    else if ([2] – [7])
      [7];
    [8]; /* null */
    y2 = recv_res(f());
    if ([2] – [3] ∧ [3] – [4])
      z2 = recv_res(f());
    if ([2] – [3])
      w2 = recv_res(h());
    v2 = recv_res(g());
  }
  printf(y, z, w, v);
}
```

(b) After global optimization

```
f(/* int a */) {
f1_p: a = recv_req(main(), x1); end_p
f2_p: a = RetVal of f(); end_p
      /* do f(): original source */
f1_e:
   send_res(main(), y2);
   c1 = recv_ctrl(main());
   if (c1) { /* [2] – [3] ∧ [3] – [4] */
      goto f2_p;
   }
end_e
f2_e:
   send_res(main(), z2);
   send_res(g(), Arg1);
end_e
}

g(/* int a */) {
a=recv_req(f()) || recv_req(main());
      /* do g(): original source */
send_res(main(), v2);
}

h(/* int a */) {
a = recv_req(main());
      /* do h(): original source */
send_res(main(), w2);
}
```

(c) Servers after global optimization

Figure 7: Transformed client and server modules for the previous example in Figure 3

are unary functions (single argument) in the example; i.e. $Receive\_Request(r)$ has a single element. In $Receive\_Request_{INI}(\boxed{8})$, '|' denotes an 'or', i.e. it implies that there are two reaching definitions. In $Send\_Result_{OPT}(\boxed{2})$ and $Send\_Result_{OPT}(\boxed{4})$, '&' denotes an 'and', so the result is sent to the both destinations.

We will use three pairs of message passing primitives: `send_req`, `recv_req`, `send_res`, `recv_res`, `send_ctrl`, `recv_ctrl`. The suffixes "`_req`", "`_res`", and "`_ctrl`" (abbreviations for "request", "result", and "control", respectively) are merely used to represent a usage of a primitive; basically, `send` and `receive` primitives suffice to implement. Thus, a pair of `send_req` and `recv_req` forms a "call" part in an RPC. A pair of `send_res` and `recv_res` forms a "return" part. A control message is necessary because a point of a request receiving or a result sending at server needs not to follow the actual control flow in the client module. So, before finalizing some task — which has proceeded in advance as much as possible, the server finally needs that actual control information. The control information during module synthesis is prepared in CT by Algorithm 4.2.

Client part of the call is the module that RPC statements are replaced with proper message passing primitives that are interspersed over the source program as determined through optimization process. In Figure 7 (a) and (b), the original positions of RPC statements are commented by "null".

Server part of the call is the module of which original code is surrounded by *prologue* and *epilogue* parts. A pair of prologue and epilogue is prepared for each *positionally different* RPC. For example, there are two calls for `f()` at $\boxed{2}$ and $\boxed{4}$, thus two pairs of prologue and epilogue are synthesized in Figure 7 (c). As explained before, this is because two different calls to the same remote function have different control and data flow behaviors.

Data availability is the only firing condition to perform *that* particular call; in other words, selecting a request to perform is done non-deterministically. Since no ordering relation is given between those multiple pairs, it looks like we have to use parallel language constructs to express the situation, but not necessarily. This kind of construct can be implemented by special message passing primitives that allows a *non-blocking* receive operation — for example, in Polylith system [22], `mh_readselect()` allows us to read the next message to arrive on *any* interface (it will be blocked if no message arrives), then `mh_readback()` completes the receipt. *Non-blocking* receive primitives are commonly supported by other MP systems like PVM [24] and MPI [12].

Finally, let's consider what has been improved in Figure 7 (b) from (a). There is no difference regarding the degree of parallelism, that is constrained by inherent data dependences. However, if '$\boxed{2} \to \boxed{4} \to \boxed{6} \to \boxed{8}$' is a call sequence to be taken, the message passing path of '$\mathtt{main} \to \mathtt{f}_1 \to \mathtt{main} \to \mathtt{f}_2$' is simplified by '$\mathtt{main} \to \mathtt{f}_1$' and '$\mathtt{main} \to \mathtt{f}_2$', and '$\mathtt{f}_2 \to \mathtt{main} \to \mathtt{g}$' is simplified by '$\mathtt{f}_2 \to \mathtt{g}$'. Moreover, the execution of `g()` is hastened by hoisting the corresponding '`send_req`' primitive up to the point before $\boxed{5}$ (if $\boxed{3} - \boxed{4}$ flow is taken), or by receiving the necessary argument earlier directly from `f()` (if $\boxed{3} - \boxed{5}$ flow is taken).

6 CONCLUSION

Some problems are inherently parallel so that it would be rather difficult to write them under procedure call abstraction; for example, problems that need group communication primitives such as broadcast or multicast. But many practical applications can be naturally expressed in a modular way using procedure call abstraction [3]. The previous unavailability of proper optimization methods discouraged programmers from using the otherwise simpler paradigm for high performance programming, in spite of its convenience. We have presented a source-level transformation framework for RPC-based distributed programs, whose goal is to parallelize independent RPC tasks and to optimize the communication paths according to the constraints of data and control dependences. Programming directly in terms of message passing primitives may still give programmers the maximum ability to write high-performance programs in distributed systems, but this freedom comes at a high price in programmer time and effort, and reduces the programmer's freedom to port, upgrade or reuse the component program units, especially if the hardware configurations in the networks of workstations are highly changeable.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–138, June 1993.

[3] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, Vol. 23(1), March 1991.

[4] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communication of the ACM*, Vol. 21(9):724–736, September 1978.

[5] B. N. Berstad, T. E.Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, Vol. 8(8):37–55, February 1990.

[6] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, Vol. 2(1):39–59, February 1984.

[7] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, June 1988.

[8] J. Callahan and J. M. Purtilo. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering*, Vol. 17(6):626–635, June 1991.

[9] J. B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency.* PhD thesis, Rice University, sep 1993.

[10] J. R. Corbin. *SUN RPC: The art of distributed applications: programming techniques for remote procedure calls.* Springer-Verlag, 1991.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and systems*, 9(3):319–349, July 1987.

[12] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings Supercomputing '93*, 1985.

[13] P. B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, Vol. 13(1), January 1987.

[14] D. K. Gifford and N. Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems*, Vol. 6(3):258–283, August 1988.

[15] Mary W. Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, October 1990.

[16] D. B. Johnson and W. Zwaenepoel. The Peregrine high–performance RPC system. *Journal of Software Practice and Experience*, Vol. 23(2):201–221, February 1993.

[17] T.-H. Kim and J. M. Purtilo. Configuration-level optimization of RPC-based distributed programs. In *Proceedings of the 15th International Conference On Distributed Computing Systems*, May 1995.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, Vol. 21(7):558–565, July 1978.

[19] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 260–267, June 1988.

[20] D. A. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communication of the ACM*, Vol. 29(12):1184–1201, December 1986.

[21] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[22] J. M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and systems*, Vol. 16(1):151–174, January 1994.

[23] J. P. Singh, A. Gupta, and M. Levoy. Parallel Visualization Algorithms: Performance and architectural implications. *IEEE Computer*, pages 45–55, jul 1994.

[24] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, dec 1990.

[25] R. von Hanxleden and K. Kennedy. Give–N–Take — A balanced code placement framework. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 107–120, June 1994.