Configuration-level optimization of RPC-based distributed programs

Tae-Hyung Kim
James M. Purtilo

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

Abstract:

Many strategies for improving performance of distributed programs can be described abstractly in terms of an application's overall configuration. But previously those techniques would need to be implemented manually, and the resulting programs, though yielding good performance, are more expensive to build and much less easy to reuse. This paper describes research towards an automatic system for introducing performance improvement techniques based upon an application's configuration description.

With oversight by Office of Naval Research, this research is supported by ARPA/ISTO in conjunction with the Common Prototyping Language project.

1 INTRODUCTION

Writing distributed programs is difficult for programmers, and even more difficult when high performance is required. Many mechanisms to achieve better performance in distributed programming have been proposed [3, 12, 13, 14, 19]; however, in practice these mechanisms are hard to utilize, and do not take into account the burden placed on programmers who already encounter difficulty in writing functionally correct programs. Furthermore, most of these mechanisms are expressed by special programming language constructs for specifying the exact semantics on communication and synchronization [3]. Such languages are not good at accommodating the programming skills of those who are already accustomed to conventional programming languages like C.

A great deal of this difficulty in developing large distributed programs arises at the early stages of program development, when the relationship between modules' functionality, their interactions and overall performance is hard discern. For a given module's functionality as dictated by some design, it is possible to implement many program units, each having some different calling conventions, servicing style and communication properties, yet all maintaining the same functionality. Previously this flexibility in how to implement the module resulted in burden to the programmer, who was tasked with selecting one of the implementations based upon too little information, and who then would be faced with great programming burden should one of those decisions need to be changed later. We now show how to exploit that flexibility.

Many of the individual mechanisms for improving performance, as cited above, have been implemented experimentally in the past, albeit without considering the programmer or designer cost. However, fewer have been implemented for evaluation in broader execution environments. Since many mechanisms can be expressed in terms of the high-level configuration of application modules, we sought to derive a practical adaptation system for configuration level programming. This approach would allow programmers to express performance improvement techniques abstractly (in terms of the configuration, instead of the low-level implementation), and then prepare appropriate implementations automatically.

Module interconnection activity is understood to be an essentially distinct and different intellectual activity from that of implementing individual modules, that is "programming-in-the-large" is distinct from "programming-in-the-small" [9]. Analogously, this observation applies to performance programming as well. Decisions concerning how a configuration might be adapted in order to allow use of performance improvement mechanisms are inherently different from the task of tailoring individual program units and their interfaces to execute as dictated by the abstract decision. Thus, each module is written to satisfy its functional requirements while each configuration program is written to specify performance related as well as interconnection related information. Many existing performance oriented mechanisms can be achieved by using ordinary modules with proper configuration programs and source-to-source translation techniques. This frees programmers from making extensive amounts of manual adaptations for various performance configurations.

The objective of our research has been to provide an adaptation system, to allow practical employment of existing performance improvement techniques; to suggest new techniques; and to allow programmers the freedom to study the impact of various techniques – in concert with one another, as desired – upon the application. As the programmer's original implementation of a module is translated under this system, each RPC statement is translated into a set of fine grained message passing primitives, and the source is translated to introduce the particular techniques specified at the configuration level. This builds upon the MIL (Module Interconnection Language) approach [5, 21] for distributed programming, where the original MIL specification is intended for structural presentation of interfaces between interacting processes. We append performance related specifications onto each interface specification in a MIL. As the performance factors are isolated from the module programming level, changing that information in order to fine tune the performance requires not whole changes in source modules, but regeneration of new executables for the performance configuration.

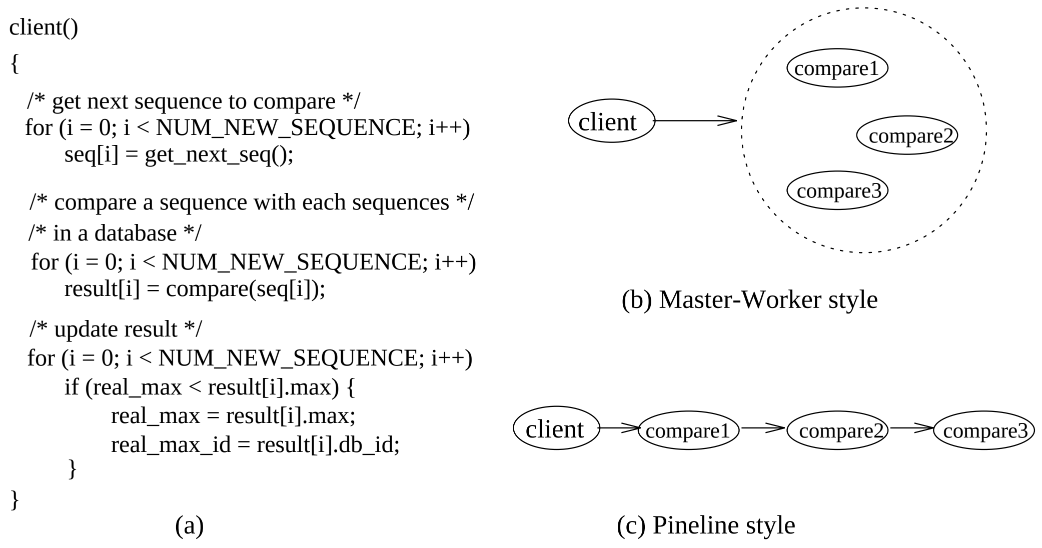
As a result of this work, programmers will have a practical and straight forward way to employ optimization techniques which previously were costly to introduce, and very costly to subsequently change. This will let them tune and experiment with the impact of techniques on their application, ultimately with benefits to both performance and development costs. From a research point of view, we hope to have a practical way to evaluate many proposed mechanisms for performance improvement, using real systems. In addition, we are studying the possibility that unwanted interference and interaction may arise when several optimization steps are employed simultaneously. The purpose of this paper is to describe the overall concept of configuration level optimization, motivate the requirements for our experimental system and to describe the evaluation activity in progress.

2 MOTIVATION

This section presents a concrete example to motivate the optimization of RPC-based distributed programs. The example we will discuss involves DNA sequences, an conceptually straight forward problem whose solutions, though very intricate in implementation, are conceptually simple and may admit several types of parallelism.

This is essentially a data structure problem: when a new DNA sequence is discovered, geneticists want to find out how and which previously known sequences the new one resembles. Suppose we have tens or hundreds of newly discovered sequences that are to be compared to a large database of existing sequences. Suppose the length of each sequence is variant, and so is the comparison time. Figure 1 (a) is a client (or *master*) module that initiates the required number of comparison tasks.

Two basic parallelizing approaches to the DNA example illustrate the problems that we are dealing with: one approach performs many sequential comparisons simultaneously as shown in Figure 1 (b), which is known as a master/workers model (database is replicated to each server),



```
client()
{
  /* get next sequence to compare */
  for (i = 0; i < NUM_NEW_SEQUENCE; i++)
       seq[i] = get_next_seq();

  /* compare a sequence with each sequences */
  /* in a database */
  for (i = 0; i < NUM_NEW_SEQUENCE; i++)
       result[i] = compare(seq[i]);

  /* update result */
  for (i = 0; i < NUM_NEW_SEQUENCE; i++)
       if (real_max < result[i].max) {
              real_max = result[i].max;
              real_max_id = result[i].db_id;
       }
}
```

Figure 1: Simple DNA sequence search

and the other constructs a pipeline of a series of sub-comparison modules by decomposing a large database to many small ones as shown in Figure 1 (c). The former reflects "data parallelism" and the later, "functional parallelism".

RPC [4, 8] is a popular paradigm for distributed programming since it simplifies distributed program construction by abstracting away from details of communication and synchronization. However, it inhibits us from expressing many useful considerations like scheduling, load balancing, stream computation and so forth, which are crucial to improving the performance of a distributed program. The problems for this example, which make direct use of conventional RPC inappropriate to high performance distributed computing, may be summarized as follows:

1. **Load balancing:** Server replication is a basic way to improve throughput. However, the performance of a replicated server can be degenerated to that of the bottleneck process(or) unless a proper load balancing scheme is used. In Figure 1 (b), no workers should be idle while others are busy. So far, RPC in itself does not make any association with load balancing. Previous RPC systems for multiple servers like PARPC [20] and MultiRPC [22] have been devised, but they do not deal with load balancing since their main purpose is fault tolerance rather than good performance.

2. **Scheduling:** In our example, the length of each DNA sequence varies, so does comparison time. In this situation, if the longest sequence is assigned to an unfortunate process at a late time near the end of all computations, only that process will be busy while others sit idle. This problem can be solved if the longest sequence is serviced first. To do this, the RPC server must be constructed to service tasks with respect to their given priorities.

3. **Parallelism:** RPC is synchronous in nature. A client must wait to get a response for its

```
module client {                         module server {
   source = "C" "local" ::                 source = "C" "remote" ::
   entrypoint = "main"  ::                 define interface compare
   use interface compare                      : pattern = { string }
      : pattern   = { string }                : returns = { integer }
      : accepts   = { integer }               : interface = "null" ::
      : interface = "stdio" ::          }
}
               module DNA_seq_search {
                  instance client ::
                  instance server: standalone ::
                  bind client compare server compare ::
                  interface = "stdio" ::
               }
```

Figure 2: Basic configuration for DNA sequence search example

call before calling another server. Preparing multiple servers or multi-stage pipelines may not be of much use if a synchronous RPC is used for remote interaction as then only one server may be activated by a client. Parallelism can be sought if the gap between *send* and *receive* primitives is widened to allow more useful computations during the wait for a result.

4. **Length of communication paths:** RPC can lengthen communication paths unnecessarily if involved modules form a computation network (like the *trellis* model in Chapter 8 of [7]) because of its two–way communication protocol. For instance, in Figure 1 (c), an intermediate result in each stage of the *compare* module must go back to the client first before being delivered to the next stage. An optimization step that eliminates such unnecessary communication paths is called for.

This example illustrates the several dimensions open to programmers, and serves to help us state simply our objective: since each of the above types of improvement admits several strategies for success, and also each can be characterized in terms of the application's configuration level description, we seek a development environment where developers may implement modules in terms of RPC interfaces (which are comparatively simple constructs), yet separately be able to express performance improvement strategies in terms of the configuration description. Figure 2 shows the basic configuration program for the example of Figure 1; it represents (in the notation of our system to be described) the conceptual starting point for configuration programmers who wish to experiment with different optimization techniques. After programmers express directions in terms of this configuration, the system should tailor all executables to be consistent with both specifications.

3 REQUIREMENTS FOR CONFIGURATION OPTIMIZATION

Section 2 exposed some limitations of using RPC for high performance distributed programs, and in doing so suggested some dimensions by which improvement can be achieved. This also makes it clear that we can separate what programmers should be able to do and what tools can do as follows:

1. High-level decisions regarding performance factors that affect overall performance should be specified in the programming-in-the-large level so that module reusability can be enhanced, especially in the process of performance tuning. Programmers should be able to specify those decisions independently.
2. High-level decisions regarding performance factors should be automatically realized and optimized with low-level message passing primitives.

The purpose of this section is to discuss in greater detail the various strategies by which performance can be improved by configuration level annotation. This will identify which features will be used for optimal realization of RPC (Section 3.1) and expression of the load balancing scheme (Section 3.2).

3.1 Performance Factors

Performance benefits are realized as latency and throughput improvements. A distributed program is composed of clients, servers and their interactions. We distinguish the task of performance improvement of a distributed program from the perspectives of its three components. Namely, clients should be able to make multiple requests (parallelism), load must be balanced among servers (load balancing), and interprocess communication and its overhead must be minimized (communication optimization). We will elaborate on factors that affect performance and what we can do to improve performance in the following subsections. Sections 3.1.1, 3.1.2 and 3.1.3 present those factors from the viewpoints of clients, servers and their interactions, respectively. All of these factors are related in module interactions rather than functionality; thus they will be represented at the interconnection programming level. When these factors are separated from individual module construction, the modules themselves can be more easily programmed as well as more reusable [9].

3.1.1 Calling Style A synchronous call is a call whereby the client blocks the call until the server completes it [4]. An asynchronous call does not block the client, and replies can be received as they are needed. To date, the decision on calling style is not the programmer's (for example, calls may be synchronous only [4] or they may be asynchronous only [2, 19, 24]), or the decision has to be made at module programming level by use of different library routines [8]. If we let this decision be separate from RPC statement, the modules will remain reusable for different calling styles. Therefore, in devising requirements for a configuration level optimization system, an asynchronous RPC should be implemented by separating the *send_request* primitive and the *receive_result* primitive to allow other useful operations in the midst of remote service. Synchronous calls would be implemented by their placement in sequence in a client module. Thus, a server module does not distinguish whether a server is called synchronously or asynchronously. It implies that the same server can be called asynchronously for one client and synchronously for another client in the same application.

The calling style should be easily prescribed by programmers in terms of a **use** clause in the module specification. Consider the module *client* in Figure 2, which calls the remote procedure *compare*. If programmers want to specify an asynchronous call, they should be able to state it in the **callstyle** expression as follows:

```
use interface compare
: callstyle = "async"
: pattern = { string }
: accepts = { integer } ::
```

3.1.2 Servicing Style

When the length of a service queue is long, throughput can be improved by the choice of a good "servicing style." Servicing style can be characterized by scheduling policy and server replication. Scheduling policy determines the desirable order of requests to be serviced. Usually the order of service is fixed by arrival time. Scheduling generalizes the order – i.e. other parameters besides arrival time are considered to determine the order of service. For example, the length of a DNA sequence to be compared may be a parameter that determines such an order as mentioned in section 2. Server replication improves throughput as well because the load is distributed among replicated servers, although load balance is crucial to good performance.

As with calling style, the module specification for expressing scheduling and replication features should be simple for programmers to assign. Illustrating one way this might appear for the introductory example, is

```
module server {
  source = "C" "remote" ::
  define interface compare
  : priority = "strlen(x)"
  : replication = "harvey.cs.umd.edu,bugs.cs.umd.edu,thumper.cs.umd.edu"
  : pattern = { string }
  : returns = { integer }
  : interface = "null" ::
}
```

Here the **priority** attribute is an expression, which would use valid syntax within the module *compare* in order to evaluate a priority. As we hoped to assign a higher priority to the longer sequence, evaluating `strlen(x)` produces the right order of priorities. The *compare* module is replicated in its simplest form here, while load balancing will be considered in section 3.2.

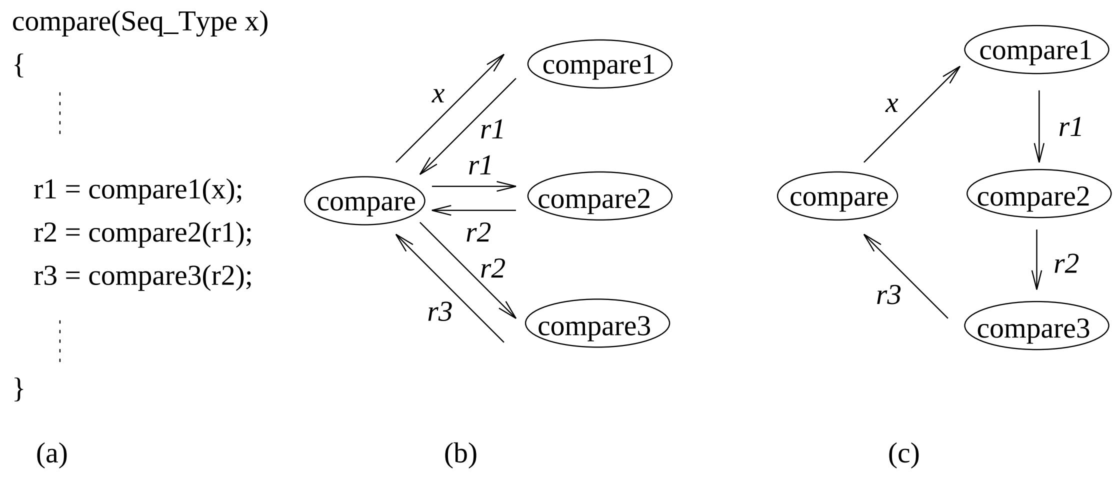
3.1.3 Communication Style

A communication pattern in distributed programs occurs in three different forms: intermittent, incremental and bulk rate data transfer. A conventional RPC protocol covers only the case of intermittent data transfer, i.e. when the number of messages between client and server is not too big or too frequent. An incremental pattern of communication occurs when we try to exploit pipeline concurrency for a chain of clients and servers as in Figure 1 (c) and Figure 3. This pattern, which forces a single computation to be decomposed into a series of distinct RPCs, reduces the server's performance since it is inactive between calls unless the synchronous behavior of RPC has been changed. Also, if we want to send bulk data by a series of RPCs, the communication performance is severely limited since it is not possible to aggregate data of successive procedure calls from a single client. Even worse, contemporary RPC systems are optimized to transmit limited amounts of data (usually less than 10^3 bytes) per call. To support the incremental and bulk rate data transfer, wherein conventional RPC systems performance suffers severely, a new communication model called *remote pipe* [15] has been devised. In the framework we are motivating, these patterns may be efficiently handled with automatic communication optimization if programmers specify which communication pattern will appear.

Once that information has been provided, there would be three ways to improve communication performance: (1) choice of proper transport, (2) reduction of kernel overhead by data aggregation and (3) elimination of unnecessary communication. The best transport protocol depends on the amount of data to be transferred. In other words, the connection-less transport protocol (UDP: User Datagram Protocol) works best for the intermittent data transfer pattern, and the connection-oriented transport protocol (TCP: Transmission Control Protocol) for the incremental and bulk rate data transfer pattern. Data aggregation allows us to amortize the overhead of kernel calls. If the size of aggregated data is increased, the throughput is increased, and if it is decreased, then the latency is reduced. Programmers can control high throughput vs. low latency by assigning the size of aggregated data to a particular server. Unnecessary communication is unavoidable in conventional RPC implementation as illustrated in Figure 3. Figure 3 (b) is optimized to (c) by elimination of the unnecessary communication paths. Figure 4 illustrates optimization to communication parallelization. A value 'v' is supposed to be transmitted to two destinations. Figure 4 (c) is optimized because module 'h' can receive 'v' independently of module 'g'; moreover, the value 'v' can be multicast if multicasting primitives are available in the underlying message passing environments.

3.2 Load Balancing

Fox *et al.* [11] demonstrated that the SPMD model is a natural paradigm for a large number of problems in science and engineering. This model can similarly be expressed by RPC paradigm with the aid of replication expressions in configuration programs, but load must be balanced among workers to insure good performance. We provide a systematic way to customize proper load balancing schemes for an RPC to replicated servers. Once programmers decide task distribution ratio, task migration paths for load balancing, and load balancing policies, then the resulting codes are automatically generated.

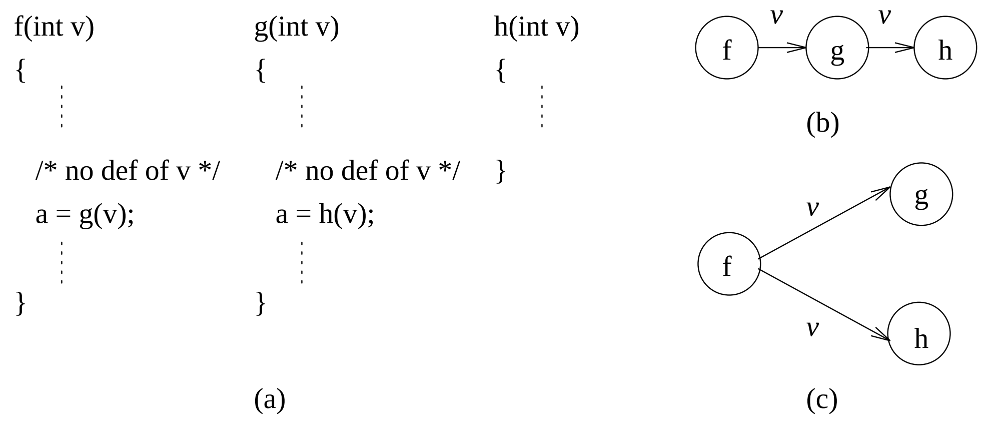


```
compare(Seq_Type x)
{
    ⋮

  r1 = compare1(x);
  r2 = compare2(r1);
  r3 = compare3(r2);

    ⋮
}
```

Figure 3: Communication optimization for figure 1 (c)



```
f(int v)               g(int v)               h(int v)
{                      {                      {
    ⋮                      ⋮                      ⋮

  /* no def of v */      /* no def of v */    }
  a = g(v);              a = h(v);
    ⋮                      ⋮
}                      }
```

Figure 4: Communication paths for a sequence of RPCs

3.2.1 Solution 1: Static load distribution

Static load distribution is a simple approach to load balance. The tasks generated by master process are distributed to the pool of worker processes according to the statically defined task distribution ratio, which is decided by programmers based on the average performance of participating workstations. The task distribution ratio is the only parameter in this scheme. Since load distribution is a client side concern, an attribute **loadratio** is needed in the **use** clause. The ratio description is matched with the **replication** attribute in the corresponding **define** clause as follows:

```
use interface compare          define interface compare
: callstyle = "async"          : replication = "harvey.cs.umd.edu,bugs.cs.umd.edu,
: loadratio = "1:2:3"                    thumper.cs.umd.edu"
        ⋮                                ⋮
```

3.2.2 Solution 2: Demand-driven load distribution

Simple dynamic load balancing can be achieved through demand-driven load distribution, which does not need to migrate tasks among workers. When a master process receives a result from a worker, it sends another task to the worker as the load situation of the worker has decreased due to the recent finish, i.e. receiving a result is regarded as a demand for another task. This scheme contains two problems. First, the master process can generate a bottleneck [16]. For example, if there are 1000 workers and a master needs 10^{-3} second to prepare and send a task, the master would create a bottleneck unless the average time for each worker to finish a task is greater than a second. Furthermore, if all workers took the same amount of time to finish their own tasks, the finish replies would come in burst, and this would cause a bottleneck, too. Second, the scheme does not allow overlap between communication and computation because the next task can not be issued unless the current one has been finished.

To alleviate these problems, watermarking can be used. It was originally used to control overload [7], but it can also be used to avoid underload, which is caused by latency. Good watermark enables a master to send a stream of task service requests; as a result, a worker does not sit idle while demanding more tasks. This requires a change in calling style, represented by "`async-demand(`*num*`)`", where *num* is a watermark value. In the following module specification, *client1* and *client2* use static and demand-driven load distribution, respectively.

```
module client1 {
        ...
  use interface compare
  : callstyle = "async"
  : loadratio = "1:1:1"
        ...
}
module client2 {
        ...
  use interface compare
  : callstyle = "async-demand(5)"
        ...
}
```

```
module server {
  source = "C" "remote" ::
  define interface compare
  : priority = "strlen(x)"
  : replication = "harvey.cs.umd.edu,
                   bugs.cs.umd.edu,
                   thumper.cs.umd.edu"
  : pattern = { string }
  : interface = "null" ::
}
```

3.2.3 Solution 3: Dynamic load balancing

When load balance cannot be reached through the above load distribution methods, tasks should migrate. Many dynamic load balancing algorithms have been devised for such an efficient migration [6, 10, 17, 18, 23]; they are characterized by the following parameters which distinguish them. Load balancing algorithms can be fine tuned when programmers can change those factors conveniently.

- **Topology:** Topology determines the shape of task migration paths. A fully connected topology provides a way to gain load balance in any case, but with some system overhead due to periodic load state exchange. The overhead can be cut through simplified topology. A compromise must be sought between reduced overhead and load balancing gains.

- **Transfer policy:** Transfer policy determines whether load has to migrate at a particular load state. The decision can be made based on local or global load information.
- **Location policy:** Location policy determines which process initiates the migration and which process should be the source or the destination in this migration: sender-initiated, receiver-initiated or mix of them.[1]
- **Selection policy:** Selection policy determines how many work load units are to migrate.

Ideally, a **replication** attribute would consists of a list of the following expression, which describes a customized topology, transfer, location and selection policy.

$$M_1\ (oc_1,\ uc_1\ )\ \#\ M_2\ (\ oc_2,\ uc_2\ )\ :\ [\gamma]^2$$

At least one condition out of "oc_1, uc_1, oc_2, uc_2" must appear, otherwise the task migration cannot be initiated. The following two examples illustrate the use of a **replication** expression.

Example 1: "`harvey.cs.umd.edu(oc=(load>=10))#bugs.cs.umd.edu:[1/2]`"
There is a task migration path between "`harvey`" and "`bugs`". The task migration will occur when the amount of load is more than 10 units. It is sender-initiated. Half ($\gamma = 1/2$) of the remaining work loads will migrate.

Example 2: "`harvey(oc=(load>=10))#bugs(uc=(load==0)):[1/2]`"
Same as above except "`bugs`" can initiate task migration, which is a receiver-initiated policy, when it sits idle (`load==0`).

In summary, a server module specification is given below, which forms a dynamic load balancing scheme that has a circular topology and sender-initiated task migration policy.

```
module server {
   source = "C" "remote" ::
   define interface compare
   : priority = "strlen(x)"
   : replication = "harvey.cs.umd.edu(oc=(L>=10))#bugs.cs.umd.edu:[1/2],
                  bugs.cs.umd.edu(oc=(L>=20))#thumper.cs.umd.edu:[1/3],
                  thumper.cs.umd.edu(oc=(L>=10))#harvey.cs.umd.edu:[1/4]"
   : pattern = { string }
   : interface = "null" ::
}
```

[1]With a *sender-initiated* policy, an overloaded process will look for a destination to export load to. With a *receiver-initiated* policy, an underloaded process will look for a destination to import load from.

[2]M_1, M_2: host address that has one replica of the server module.
oc_i, uc_i: a condition that decides overloading or underloading for M_i, respectively.
'#': a linkage between M_1 and M_2.
γ: the number of load units to migrate (fraction).

```
a.cfg             Makefile client.cl server.cl all.cl
client.c          x.client.c x.client.h
server.c    ⟹     x0.server.c x0.server.h x1.server.c x1.server.h
                  x2.server.c x2.server.h x3.server.c
                  x3.server.h
```

Figure 5: Generated files from user provided modules using CORD

4 DEVELOPING APPLICATIONS IN CORD

The previous section has characterized the various forms of optimization which are possible to discuss in terms of an application configuration. We have been developing a support environment called CORD (Configuration-level Optimization for RPC-based Distributed programs) to allow us to experiment with introduction of such adaptations at low cost.

The CORD system is primarily an integration effort at the University of Maryland, and builds extensively upon prior results in the area of software interconnection. The configuration language chosen for expressing modules and their compositions is derived from the Polylith module interconnection language, and the distributed run time environment chosen is likewise the software bus behind Polylith. Basic tools for preparing applications to run in this environment are already available within the Polygen system [5], although they are to be tailored to attain our source translation (rather than stub generation) principle. Therefore the principle thrust of our effort has been to add a source translator (**gen_trans**) to the suite of Polygen tools. The source translator operates differently depends on whether a module is a client or a server from given RPC's viewpoint: for a client module, it performs data flow analyses to place message passing primitives optimally, and for a server module, it generates proper codes to implement particular servicing styles described in configuration programs. Though we have omitted the statement of algorithms from this paper in the interests of space, the translation is straight forward, because we can decide the earliest time to send a request and the latest time to receive a result based on DUC_l (Definition-Use-Chain of l-value) and UDC_r (Use-Definition-Chain of r-value) sets, that can be evaluated through **use-def** and **def-use** analyses [1].

The development of an application in CORD consists of a number of steps. At some point, each module used in the application must be given an implementation, each dealing with interfaces in generic RPC terms, of course. Since performance decisions that occur in module interactions are decoupled from the module programming level, module functionality is the only concern in this step.

The second step is to define an application using the module interconnection and performance configuration. In the next step, CORD generates all necessary files for an executable automatically with respect to the configuration program. Figure 5 shows the automatically generated files from the user provided files, which are source programs `client.c, server.c` and the configuration program `a.cfg`, using CORD. (In the figure, it is assumed that the `server.c` is replicated

	Initially the user has source codes ("client.c, server.c") and configuration file ("a.cfg")
% config < a.cfg > a.pl	*Creates prolog assertions for the configuration*
% prolog < a.pl > a.pkg	*Given inference engine ("package.pl"), generates the packaging information ("a.pkg") to satisfy the packaging goal according to the generated assertions*
% gen_imake < a.pkg > Imakefile	*Creates Imakefile file from the packaging information*
% imake -T "Imake.tmpl"	*Creates Makefile using a prepared Imake template ("Imake.tmpl")*
% make	*Creates executables according to the interface generation, source translation, and compilation information in the Makefile*
	The following output is from commands called from the makefile
gen_header client < a.pkg > client.h	*generates a header file for the client.c*
gen_header server < a.pkg > server.h	*generates a header file for the server.c*
gen_trans client.c < a.pkg > x.client.c	*translates from the original user code client.c to x.client.c*
gen_trans server.c < a.pkg > x.server.c	*translates from the original user code server.c to x.server.c*
gen_module client < a.pkg > client.cl	*generates the client specification*
gen_module server < a.pkg > server.cl	*generates the server specification*
gen_cluster < a.pkg > all.cl	*generates the application specification*
csc client.cl	*compiles the client specification into client.co*
csc server.cl	*compiles the server specification into server.co*
csc all.cl	*compiles the application specification into all.co*
csl -o all client.co server.co all.co	*creates a root executable that executes client and server*
cc -o client x.client.c -lith	*compiles the x.client.c component and creates client*
cc -o server x.server.c -lith	*compiles the x.server.c component and creates server*
⋮	⋮

Figure 6: Script for the design (user commands prefixed by a % prompt)

to four distinct machines.) This step follows the similar packaging process in Polygen, which deals with automatic adaptations for divergent structural and geometric configurations. The interaction between modules in distinct sites, which is an RPC, is resolved by generating client and server stubs automatically by Polygen. CORD does not generate stubs but translates source codes in which every RPC is replaced with a set of message passing primitives interspersed for the purpose of optimization. The script of the entire process, which includes both user commands and the execution of the configuration program, is shown in Figure 6. The tools that are involved in this process, are summarized as follows:

- **config** generates prolog assertions (`a.pl`), which encodes facts about the modules and bindings in the configuration, from user provided configuration (`a.cfg`).
- **prolog**: After reading the assertions (`a.pl`), the CORD uses **prolog** inferencing mechanism to search for satisfying the goal, which asks the possibility to create an application for the

	harvey	rimfire	thumper	highpower
Sync	216	103	86	57
Async	125	59	52	30

(a) Single server case

	Type 1	Type 2	Type 3
ALL	34	26	17

(b) Multi-server with various configurations

Table 1: Measured time (in seconds) to compute Mandelbrot set on [0.5,-1.8] to [1.2,-1.2] with 200×200 pixel window used

configuration described in `a.cfg` by means of the available tools in the environment. This inference results in a package information (`a.pkg`) if successful.

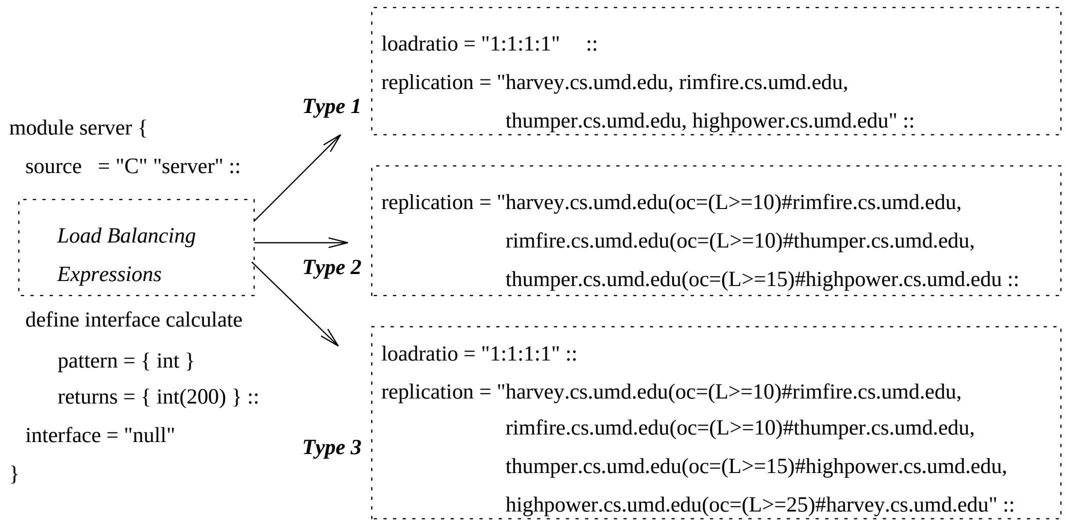
- **gen_imake**: Using the package information (`a.pkg`), **gen_imake** generates an `Imakefile` to create a `Makefile` for an application. A UNIX `imake` is used to generate a `Makefile` from a provided template in CORD.
- **gen_poly_module** generates a MIL program (`.cl`) for the module descriptions[3].
- **gen_poly_cluster** generates a MIL program (`.cl`) for the application description.
- **gen_poly_hdr** generates a header file for each module if necessary.
- **gen_trans** generates translated source code to realize RPCs using message passing primitives, and proper codes for scheduling and/or load balancing.

The final step is to execute the application, identify performance bottlenecks using a performance measurement tool, and repeat the process from the second step until the resulting performance is satisfactory.

The evaluation of CORD's effectiveness overall is currently under way, as is our study of the potential for unintended interference between various configuration level optimization strategies. Nevertheless, it is already possible at this date to suggest the potential for CORD in helping programmers to discover desirable optimization opportunities at low cost. We do illustrate this using the Mandelbrot example, using a generically coded C implementation built in the Polylith system. In this implementation, a sub-task is to compute the set for one row in 200×200 pixel window, therefore 200 RPCs will be made to complete the whole computation. This formulation of the problem increases traffic beyond that of alternative implementation, but makes the effect of any optimization strategies more easily measured for illustration.

Table 1 shows timing results when we execute this Mandelbrot program for various performance improvement alternatives, where the programmer may select each mechanism by making only a simple attribute change in the module specification as in Figure 7. Table 1 (a) compares the

[3]The components of the MIL program are the module descriptions and the application description. See [21].



```
module server {
  source   = "C" "server" ::
      Load Balancing
      Expressions
  define interface calculate
      pattern = { int }
      returns = { int(200) } ::
  interface = "null"
}
```

Type 1

```
loadratio = "1:1:1:1"     ::
replication = "harvey.cs.umd.edu, rimfire.cs.umd.edu,
               thumper.cs.umd.edu, highpower.cs.umd.edu" ::
```

Type 2

```
replication = "harvey.cs.umd.edu(oc=(L>=10)#rimfire.cs.umd.edu,
               rimfire.cs.umd.edu(oc=(L>=10)#thumper.cs.umd.edu,
               thumper.cs.umd.edu(oc=(L>=15)#highpower.cs.umd.edu ::
```

Type 3

```
loadratio = "1:1:1:1" ::
replication = "harvey.cs.umd.edu(oc=(L>=10)#rimfire.cs.umd.edu,
               rimfire.cs.umd.edu(oc=(L>=10)#thumper.cs.umd.edu,
               thumper.cs.umd.edu(oc=(L>=15)#highpower.cs.umd.edu,
               highpower.cs.umd.edu(oc=(L>=25)#harvey.cs.umd.edu" ::
```

Figure 7: Module specification for various load balancing schemes

performance between synchronous and asynchronous RPC where the computation is run on each of several different servers in turn. (To be concrete, '`harvey`' is SparcStation IPC, '`rimfire`' is SparcStation IPX, '`thumper`' is SparcStation 2, and '`highpower`' is SparcStation 10: the broad spectrum of computing power in these machines is intentional to cause load imbalance in the later load balancing test.) Asynchronous RPC is better because it allows to overlap server computation with communication. Table 1 (b) shows timing results when all four machines are cooperating for the computation. Each row in the Table 1 (b) indicates the type of load balancing among four servers. Type 1 is when tasks are distributed equally in spite of divergence in computing power – the performance is degenerated to that of harvey, the slowest machine (see "$34 \approx 125/4$"). Type 2 is when the task migration paths are linearly connected, i.e. "client $\rightarrow$ harvey $\rightarrow$ rimfire $\rightarrow$ thumper $\rightarrow$ highpower." Type 3 is when the paths are circular and the client distributes the equal number of tasks to all servers initially. The CORD system allows us to track down these configurations towards better performance without having to worry about extensive amount of manual adaptations. Each of the execution scenarios shows performance that is comparable to a manually coded counterparts, yet these were achieved without extensive manual intervention on the part of programmers.

5 CONCLUSION

We have described a system called CORD that allows configuration-level optimization of RPC-based distributed programs. Because it automatically adapts the application at the source level, CORD encourages programmers to experiment with various performance improvement strategies in order to discover the best for their environment and data. 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.

Though this description is still one of work in progress, the availability of CORD places us well on our way to supporting an overall hypothesis: that isolating many types of performance factors from the module programming level, and deferring such decisions to the configuration level, can decrease the cost of developing and tailoring application programs, while at the same time achieving overall performance comparable to manually tailored counterparts. Programmers can have conceptual simplicity and high performance at the same time.

References

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

[2] A. L. Ananda, B. H. Tay, and E. K. Koh. Astra – An asynchronous remote procedure call facility. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 172–179, 1991.

[3] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, Vol. 21(3):260–322, September 1989.

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

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

[6] Clemens H. Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, Vol. 19:1221–1234, 1993.

[7] N. Carriero and D. Gelernter. *How to write parallel programs: A first course.* MIT Press.

[8] John R. Corbin. *SUN RPC:The art of distributed applications.* Springer-Verlag.

[9] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, Vol. 2(2), June 1976.

[10] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, Vol. 12(5):662–675, May 1986.

[11] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, 1988.

[12] N. H. Gehani. Concurrent C. *Journal of Software Practice and Experience*, Vol. 16:821–844, September 1986.

[13] N. H. Gehani. Message passing in concurrent C: Synchronous versus asynchronous. *Journal of Software Practice and Experience*, Vol. 20(6):571–592, June 1990.

[14] W. M. Gentleman. Message passing between sequential processes: The reply primitive and the administrator concept. *Journal of Software Practice and Experience*, Vol. 11:435–466, May 1981.

[15] 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.

[16] J. L. Gustafson, R. E. Benner, M. P. Sears, and T. D. Sullivan. A radar simulation program for a 1024-processor hypercube. In *Proceedings of SuperComputing 1989*, pages 96–105, 1989.

[17] Philip Krueger and Niranjan G. Shivaratri. Adaptive location policies for global scheduling. *IEEE Transactions on Software Engineering*, Vol. 20(6):432–444, June 1994.

[18] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, Vol. 13(1):32–38, January 1987.

[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] Bruce Martin, Charles Bergan, and Brian Russ. PARPC: A system for parallel remote procedure calls. In *Proceedings of the International Conferences on Parallel Processing*, pages 449–452, 1987.

[21] James Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, Vol. 16(1):151–174, January 1994.

[22] M. Satyanarayanan and E. H. Siegel. MultiRPC: A parallel remote procedure call mechanism. Technical Report CMU-CS-86-139, Carnegie-Mellon University, 1986.

[23] Jianjian Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Computing*, Vol. 20:853–868, 1994.

[24] E. F. Walker, R. Floyd, and P. Neves. Asynchronous remote operation in distributed systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 253–259, 1990.