Improvements in Marshalling
CHAPTER 5

IMPROVEMENTS IN MARSHALLING

5.1 Introduction

Distributed Embedded Applications are targeted towards highly competitive market and are subject to frequent changes. Hence developing these applications using component architecture can reduce the software lifecycle time. Distributed Object Computing middleware like CORBA can be employed in the development of distributed embedded application due to its programming language, hardware and naming transparency. Embedded applications also face severe limitations on memory and speed. Since core CORBA faces memory and speed limitations, it cannot be directly applied to distributed embedded systems. MinCORBA [5] can be used to reduce the size of the ORB, but it does not produce time and space efficient presentation conversion routines. The main challenges faced by CORBA-based applications in presentation conversion include generation of time and space efficient stub code, fast Inter Process Communication (IPC) mechanisms, and establishment of secure communication between client and server objects. This chapter proposes the following methods to improve the presentation conversion procedure:

1) Time efficient IPC mechanisms.
2) Generation of time and space efficient stub code.

IPC mechanisms offered by the microkernel facilitate cross domain communication among components. Transparency can be achieved using IDLs. The stub code generated by the IDL compiler marshals parameters on the client side, communicates through IPC/RPC kernel primitives with the server, unmarshals the parameters on the server side and invokes the corresponding server procedure. The result returned from the server procedure is marshalled back to the client. Current IDL compilers are focused more on generating code in a portable and adaptable way than on producing efficient stubs. Sun's rpcgen [153] is not implemented using modern compiler
technologies and does not have multiple and flexible intermediate representations. Flick [154] [155] generates efficient stub code by inlining functions and using macros. Flick-generated stubs marshal data between 2 and 17 times faster than stubs produced by traditional IDL compilers, and can increase end-to-end throughput by factors between 1.2 and 3.7. But due to inlining, the size of the stubs become very large. Andreas [156][157] has reported that IDL4 produces stub code which is three times faster than that of a portable IDL compiler. It uses L4 kernel specific optimizations like flexpage, short IPC, direct stack transfer and bit stuffing. The Mach Interface Generator [158] provides a remote procedure call interface for IPC message passing in a multitasking system. The interface definition language, MIG uses, is specific to Mach Operating System. The tempo system [159] optimizes the output of the IDL compiler using partial evaluation to specialize the code at compile time by removing genericity and making it specific to a particular operating system.

The Inter Language Unification System (ILU) [160] uses Interface Specification Language (ISL) for interface definitions. It is compiled into stubs and skeletons. ILU clients and servers communicate using ILU defined communication protocols. The object interfaces provided by ILU hide implementation distinctions between different languages, different address spaces and operating system types. The Universal Stub code Compiler (USC) tool [161] minimizes the cost of marshalling / unmarshalling at compile time by annotating 'C' programming language with layout of various data types. But it does not provide run-time solutions. USC optimization is based on inlining, but the method cannot be applied to more complex, application-oriented interface definitions without incurring a significant code size overhead, since it is based on inlining. Network Interface Data Language (NIDL) [162] uses 'C' language data description with annotations and Network Data Representation (NDR) as its wire format. But USC and NIDL focus on a single language. So this chapter proposes changes in transmission medium to improve the performance of the stub code generated by the compiler in a more generic way. It also proposes methods to generate time and space efficient stub code by implementing an optimizer in the presentation generation stage of the IDL
The proposed IDL compiler is based on the insight that IDLs are true languages suitable to modern compilation. It is componentized, to bring about extensibility and reuse. The intermediate data representation facilitates many-to-many mapping between IDLs, target languages and communication protocols. It has the following advantages:

1) It can handle multiple IDLs and can map onto different target languages by using an intermediate language representation.
2) Since the compiler is componentized, it allows for the addition of new components and the reuse of existing components.
3) Since the compiler is designed as a traditional language compiler, it promotes both flexibility and optimization.
4) It allows for interoperability between different distributed object models.
5) Given a single IDL, it can map onto different target languages.
6) The generation of efficient stub code using IPC mechanisms like shared memory and sockets over TCP/IP by the back end of the compiler improves the performance of the IDL compiler.

5.2 IDL Compiler

An IDL compiler is used for static invocation of remote objects. It accepts an IDL specification and outputs an implementation of the specification. The implementation has data type declaration and stubs which take care of communication between client and server.

5.2.1 Design of the Proposed IDL Compiler

The proposed IDL compilation is done in three phases as shown in figure 5.1. These phases are explained below:

1) Front End: It reads IDL and generates a parse tree. It is language independent. Since an intermediate representation of interfaces is used, multiple IDLs can be given as input to the IDL compiler. This phase uses
lex and YACC for lexical analysis and parsing of the IDL file to produce the intermediate Parse Tree (PT) representation.

![Diagram of IDL compiler process]

**Figure 5.1: Overview of the proposed IDL compiler**

3) **Presentation generator**: It reads the PT and generates mapping onto target language declaration constructs like C++ or Java as shown in table 5.1. The output of this phase is the presentation file in a particular programming language.
Linear linked list is the data structure chosen to implement this phase of the IDL compiler. The following structures are used to generate an interface definition in C++ from a PT file. Totally there are ten structures out of which eight are dynamic and two are static. These structures are used to produce an interface in the target language from a PT file.

The following paragraphs explain the different structures in detail.

a) Static Structures

The static data structures used in the implementation of the second stage of the IDL compiler are explained below.

i) PT keyword structure: This static structure contains the keycode for all the keywords in the intermediate PT file as shown in table 5.2. This is used in the creation of the communication stub, to find out the correct keyword in Java and C++ for a particular PT type. It is used along with the keyword structure and range structure.

ii) Java keyword structure: It relates the keycode specified in the PT keyword structure and the keycode specified in the range structure with the Java and C++ keyword. It is shown in table 5.3.

---

Table 5.1: Mapping between CORBA IDL, parse tree format and Java/C++

<table>
<thead>
<tr>
<th>CORBA keyword</th>
<th>PT-KEYWORD</th>
<th>Java Keyword</th>
<th>C++ Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>PT-NAMESPACE</td>
<td>package</td>
<td>header file</td>
</tr>
<tr>
<td>Interface</td>
<td>PT-INTERFACE</td>
<td>interface</td>
<td>class</td>
</tr>
<tr>
<td>Short</td>
<td>PT-INTEGER</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>Char</td>
<td>PT-CHAR</td>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>Float</td>
<td>PT-FLOAT</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>arrays</td>
<td>PT-ARRAY, kind and range</td>
<td>kind array</td>
<td>kind array</td>
</tr>
<tr>
<td>const</td>
<td>PT-CONST</td>
<td>public static final</td>
<td>const</td>
</tr>
<tr>
<td>struct</td>
<td>PT-STRUCT</td>
<td>public final class</td>
<td>struct</td>
</tr>
<tr>
<td>enum</td>
<td>PT-ENUM</td>
<td>public final class</td>
<td>enum</td>
</tr>
<tr>
<td>union</td>
<td>PT-UNION</td>
<td>public final class</td>
<td>union</td>
</tr>
<tr>
<td>typedef</td>
<td>PT-INDIRECT</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>void</td>
<td>PT-VOID</td>
<td>void</td>
<td>void</td>
</tr>
<tr>
<td>unsigned long</td>
<td>PT-SCALAR</td>
<td>long</td>
<td>int</td>
</tr>
<tr>
<td>any</td>
<td>PT-ANY, PT-TYPE-TAG, PT-TYPED</td>
<td>Maps onto runtime data type</td>
<td>void*</td>
</tr>
</tbody>
</table>

Maps onto runtime data type: void*
Table 5.2: PT Keyword structure

<table>
<thead>
<tr>
<th>PT Keyword</th>
<th>Keyword value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sot</td>
<td>1</td>
</tr>
<tr>
<td>Scope</td>
<td>2</td>
</tr>
<tr>
<td>Name</td>
<td>3</td>
</tr>
<tr>
<td>PT-NAMESPACE</td>
<td>4</td>
</tr>
<tr>
<td>PT-INTERFACE</td>
<td>5</td>
</tr>
<tr>
<td>PT-INTEGER</td>
<td>6</td>
</tr>
<tr>
<td>PT-CHAR</td>
<td>7</td>
</tr>
<tr>
<td>PT-FLOAT</td>
<td>8</td>
</tr>
<tr>
<td>PT-ARRAY</td>
<td>9</td>
</tr>
<tr>
<td>PT-CONST</td>
<td>10</td>
</tr>
<tr>
<td>PT-STRUCT</td>
<td>11</td>
</tr>
<tr>
<td>PT-ENUM</td>
<td>12</td>
</tr>
<tr>
<td>PT-UNION</td>
<td>13</td>
</tr>
<tr>
<td>PT-EXCEPTION</td>
<td>14</td>
</tr>
<tr>
<td>PT-INDIRECT</td>
<td>15</td>
</tr>
<tr>
<td>PT-VOID</td>
<td>16</td>
</tr>
<tr>
<td>PT-SCALAR</td>
<td>17</td>
</tr>
<tr>
<td>PT-OPTIONAL</td>
<td>18</td>
</tr>
<tr>
<td>PT-FWD-INTRFC</td>
<td>19</td>
</tr>
<tr>
<td>PT-ANY</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 5.3: Java/C++ keyword value structure

<table>
<thead>
<tr>
<th>Java Keyword</th>
<th>C++ keyword</th>
<th>Keycode</th>
</tr>
</thead>
<tbody>
<tr>
<td>package</td>
<td>header file</td>
<td>4</td>
</tr>
<tr>
<td>interface</td>
<td>class</td>
<td>5</td>
</tr>
<tr>
<td>int</td>
<td>long int</td>
<td>6</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td>7</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>8</td>
</tr>
<tr>
<td>string</td>
<td>array of characters</td>
<td>9</td>
</tr>
<tr>
<td>public static final</td>
<td>const</td>
<td>10</td>
</tr>
<tr>
<td>public final class</td>
<td>struct</td>
<td>11</td>
</tr>
<tr>
<td>public final class</td>
<td>enum</td>
<td>12</td>
</tr>
<tr>
<td>public final class</td>
<td>union</td>
<td>13</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>14</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>15</td>
</tr>
<tr>
<td>void</td>
<td>void</td>
<td>16</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>17</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>18-21</td>
</tr>
<tr>
<td>any</td>
<td>none</td>
<td>22</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>23</td>
</tr>
<tr>
<td>boolean</td>
<td>int</td>
<td>24</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>25</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>26</td>
</tr>
<tr>
<td>short</td>
<td>short int</td>
<td>27-29</td>
</tr>
</tbody>
</table>

Table 5.4: Range structure

<table>
<thead>
<tr>
<th>Range</th>
<th>Keycode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>24</td>
</tr>
<tr>
<td>64</td>
<td>25</td>
</tr>
<tr>
<td>0-4294967295</td>
<td>26</td>
</tr>
<tr>
<td>-2147483648-2147483647</td>
<td>6</td>
</tr>
<tr>
<td>-32768-2767</td>
<td>27</td>
</tr>
<tr>
<td>0-65535</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>-32768-32767</td>
<td>29</td>
</tr>
</tbody>
</table>

b) Dynamic Structures

The dynamic data structures used in the implementation of the second stage of the IDL compiler are explained below. Their size varies at runtime.
i) **Range Structure:** It is used to identify the correct data type while generating stub code. Table 5.4 gives the range of a data type and its corresponding keycode.

ii) **Identifier Structure:** This structure contains the main keyword information in the PT file like the module information, interface information and constant variable information. It has the following fields:

- Slot Number of a keyword.
- Scope of the keyword in a module.
- User defined name of a keyword, which includes, module name and interface name.
- Kind of identifier which specifies whether the identifier is module name or interface name.
- Address of the nodes that correspond to the slot and the address of the next module number.

iii) **Operation Structure:** It contains information for the various operations contained in a particular interface. For each operation a separate node is created. The identifier structure stores the address of this node. It has the following fields:

- Address of the parameter list.
- Return type.
- Whether the operation's return type is direct or indirect.
- Name of the indirect variables.

iv) **Parameter Structure:** It contains the information regarding parameters passed in the operation/method. It has the following fields:

- Name of the parameter.
- Direction of the parameter. It can be IN, OUT, IN-OUT.
- Parameter kind.
- Parameter range.
- Whether the variable is indirect or not.
- Name of the indirect variable.
v) **Constant Structure:** It has the information regarding constant variables. The identifier structure stores the address of this structure. It has the following fields:
- Kind of the constant variable
- Name of the constant variable.

vi) **Indirect data representation structure:** It contains information about the typedef statement. It has the following fields:
- Slot Number of the indirect data.
- Scope of the data.
- User defined name of the indirect data.
- Kind of name.
- Range of the name.
- A forward reference, which specifies the interface count, depending upon the interface level in which typedef is done. If typedef is done at the start of the module, then its value is taken as -1.

vii) **Structure variables table:** This dynamic table has an entry for each structure defined in the PT file. It has the following fields:
- Member name.
- Member type.
- Slot number.
- Pointer to the next variable.

viii) **Keyword list table:** This dynamic table stores a list of PT keywords in the PT file, its keycode and their equivalent C++/Java keywords.

An optimizer is incorporated in the presentation generation stage of the compiler to generate time and space efficient stub code as proposed in section 5.5.

3) **Back End:** It reads the interface file generated by the presentation generator and generates skeletons and stubs for a particular transmission mechanism and message format. It is independent of the IDL and presentation rules. Conventional CORBA stubs use IIOP for
communication. Since it is slow, modules are incorporated in the back end to produce stubs with faster IPC mechanisms. The back end supports RMI/JRMP, RMI/IIOP and CORBA/IIOP communication mechanisms over TCP/IP. It can also produce the following types of stub code which use different IPC mechanisms:

**a) Shared memory:** Conventional IPCs like FIFO, pipes and message queues require at least 4 copy operations. It includes input file to client copy through kernel, client to pipe copy, pipe to server copy, and server to output file copy through kernel as shown in figure 5.2. Using shared memory, the number of copy operations can be reduced. When a client calls a function, the function name and arguments are copied into shared memory area. Only the key values to the shared memory area are passed to the server side, hence the networking time is reduced. The server program reads the function details using key values passed to it. The result is then marshaled back to the client as shown in figure 5.3.

**b) Multiple shared memory segments:** Here multiple segments of shared memory is used to access different arguments. Each shared memory segment is taken as a single buffer. A fixed size is allocated to a buffer and the arguments passed are divided and written into the buffers. Each argument can be placed into one or more buffers.

![Figure 5.2: Data movement using FIFO/pipes/message queues](image1)

![Figure 5.3: Data movement using shared memory](image2)
c) Threading multiple shared memory segments: Threading allows a program to perform multiple tasks simultaneously. A pool of threads is created and a thread is allotted to each shared memory segment as shown in figure 5.4. Reading and writing from/to segment is done using this thread.

From a logical point of view, multithreading means multiple lines of a single program can be executed at the same time. Under Unix, forking a process creates a child process with a different address space for both code and data. However, fork() creates a lot of overhead for the operating system, making it a very CPU-intensive operation. By starting a thread instead, an efficient path of execution is created while still sharing the original data area with the parent.

Since each buffer is of a fixed size, the number of buffers needed is calculated from the size of arguments passed. A pool of threads is created and from this pool, a thread is taken for each shared memory and reading/writing on the shared memory is controlled by that thread. But this method is only suitable for transfer of large...
amounts of data, since there is always overhead involved in context switching of the threads.

d) Sockets over TCP/IP: In this method, the function call and its parameters are copied into a common buffer and passed to the server through the TCP/IP sockets. The contents of the buffer received through the sockets are read and the parameters are arranged in the form of a system call. The function is executed and the return value is sent back to the client. This method can be used for communication between homogeneous systems.

e) Multiple Sockets: In this method, multiple sockets can be established along with threads to transfer large amounts of data, but it is not suitable to transfer small amounts of data due overhead involved in context switching of threads.

f) GIOP over UDP: GIOP specifications have been implemented over UDP. A simple acknowledgement has been incorporated to check for reliability. This method facilitates multicasting to IIOP. Conventional IIOP is an implementation of GIOP over TCP/IP and hence does not facilitate multicasting.

5.2.2 Experimental Results

A comparative study of the different IPC mechanisms in the stub code produced by the back end of the proposed IDL compiler has been performed from the developer’s standpoint. For a developer the most important performance criterion is the total time taken by a remote method invocation. This time is defined as the round trip time (RTT). RTT is the time that elapses between the initiation of a method invocation by the client until the results are returned to the client. To evaluate the performance of the stub code, the following measurements are taken:

1) The effect of the different data types as parameters and return values on the results: RTT has been measured for the following:

   i) simple data types like float, char and long.
ii) data arrays of float and character type.

iii) user defined data type namely structure.

2) The effect of the data size on the results: Performance results for different data arrays like float and character from size 10 up to size 10000 have been gathered. The response time in this distributed configuration is approximately Client CPU time + network time (outbound) + Server CPU time + network time (inbound). Each interface provides two methods. One method accepts a data type as a parameter and has no return value, called as send procedure. The second method has no parameters but returns the data type as a return value, called as receive procedure. The measurements have been carried out with Redhat Linux ver 7.1 as the operating system.

The following paragraphs explain the performance of the different data types.

1) Performance of Primitive Data Types

The following observations are made on the primitive data types:

a) In the case of the primitive data types like float as shown in figure 5.5 a and 5.5 b, shared memory usage shows the best performance. Similar results were observed for other primitive data types likes char, long and short.

b) The order of performance for primitive data types is shared memory, multiple buffers, multiple buffers with threads, sockets and multiple sockets.

c) It is observed that using shared memory the networking time is less, since only the key values are passed. On the other hand the networking time using sockets is more since the entire data has to pass through the network.

d) Since the number of copy operations using shared memory is less, it is faster when compared to sockets.

e) Shared memory needs more time to marshal and unmarshal the arguments, whereas sockets spend more time in networking the data values as shown in table 5.5.
f) Since time is needed for context switching it is seen that multiple buffers with threads take more time. Around 45% of the total time is spent in context switching of the threads.

g) It is also observed that, while receiving the data elements the amount of time spent on marshalling and unmarshalling is less when compared to sending the data elements.

![Figure 5.5a: Marshall, Network, Unmarshall and Total Time to send float (Primitive) data](image1)

![Figure 5.5b: Marshall, Network, Unmarshall and Total Time to receive float (Primitive) data](image2)
Performance of the stub code has also been evaluated for a user defined data type, namely structure with character, float, long, string and float array as its members. It is observed that the performance of the structure (figures 5.6 a and b) is similar to that of primitive data types. It is also observed that the order of performance is shared memory, multiple buffers, multiple buffers with threads, sockets and multiple sockets, as in the case of primitive data types.
Table 5.5: Percentage of marshal, unmarshal and networking time out of total time taken to send / receive data values

<table>
<thead>
<tr>
<th>Method</th>
<th>% marshal time</th>
<th>% unmarshal time</th>
<th>% networking time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared memory (Basic data types)</td>
<td>35%</td>
<td>10%</td>
<td>45%</td>
</tr>
<tr>
<td>Sockets (Basic data types)</td>
<td>10%</td>
<td>10%</td>
<td>50%</td>
</tr>
<tr>
<td>Sending arrays (shared memory)</td>
<td>50%</td>
<td>10%</td>
<td>10%</td>
</tr>
<tr>
<td>Sending arrays (Other methods)</td>
<td>5%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>Receiving arrays (shared memory)</td>
<td>10%</td>
<td>10%</td>
<td>50%</td>
</tr>
<tr>
<td>Receiving arrays (sockets)</td>
<td>30%</td>
<td>5%</td>
<td>50%</td>
</tr>
<tr>
<td>Receiving arrays (other methods)</td>
<td>5%</td>
<td>5%</td>
<td>15%</td>
</tr>
</tbody>
</table>

3) Performance of Arrays

The following observations were made on arrays of character and float data types as shown in table 5.5:

a) While sending arrays using shared memory, around 50% of the total time is spent on marshaling and around 10% of the total time is spent in networking and unmarshalling elements.

b) While receiving arrays using shared memory, more time is spent in the network (around 50%) and less time is spent in marshalling and unmarshalling elements.

c) For all the other methods, while sending arrays, only 5% of the total time is spent in marshalling and unmarshalling the arguments and around 15% of the total time is spent on networking. The rest of the time is spent in context switching of the threads and buffer allocation.

d) While receiving arrays using sockets, more time is spent in the network (around 50%), 30% of the time is spent in marshalling and 5% of the time is spent in unmarshalling elements.
e) For all the other methods, while receiving arrays, only 5% of the total time is spent in marshalling and unmarshalling the arguments and around 15% of the total time is spent on networking. The rest of the time is spent in context switching of the threads and buffer allocation.

The following observations were made on character and float arrays of varying sizes (figures 5.7 a and 5.7 b) as shown in table 5.6:

a) On using shared memory, as the number of elements in the array increases, the total time taken to pass the array elements also rapidly increases. It is around 98% for sending arrays and 35% for receiving arrays.

b) The rate of increase for sockets is around 98% while sending arrays and 90% while receiving arrays.

c) The rate of increase for multiple sockets is around 30%. But for less number of elements, it takes more time. So multiple sockets can be used in places where a large amount of data transfer is needed.

d) The rate of increase for multiple buffers is around 90%.

e) The rate of increase for multiple buffers and threads is around 95%. This is mainly due to the context switching time involved when using threads.

f) While sending arrays of large size, the order of performance is multiple buffers, multiple sockets, multiple buffers with threads, shared memory and sockets.

Table 5.6: Percentage increase in time to send/receive an array of 10000 elements with respect to an array of 10 elements

<table>
<thead>
<tr>
<th>Method</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared memory</td>
<td>98%</td>
<td>35%</td>
</tr>
<tr>
<td>Sockets</td>
<td>98%</td>
<td>90%</td>
</tr>
<tr>
<td>Multiple sockets</td>
<td>30%</td>
<td>30%</td>
</tr>
<tr>
<td>Multiple buffers</td>
<td>90%</td>
<td>90%</td>
</tr>
<tr>
<td>Multiple buffers and sockets</td>
<td>95%</td>
<td>95%</td>
</tr>
</tbody>
</table>

1: percentage increase in send time for an array of size 10000 with respect to an array of size 10.
2: percentage increase in receive time for an array of size 10000 with respect to an array of size 10.
4) Performance of IIOP/UDP

The purpose of Unreliable Multicast Inter-ORB Protocol (MIOP) is to provide a common mechanism to deliver GIOP request and fragment messages through multicast. The default transport specified for MIOP is IP Multicasting through UDP/IP2 provides the ability to perform connectionless multicast. This requires that IDL operations should have one-way semantics. The default communication medium to transfer information between stub and skeleton is IIOP/TCP. A Common Data Representation (CDR) is used to marshal the request parameters and results. The communication medium was later modified for IIOP/UDP to provide multicasting support. It was found that
the networking speed improved. The percentage improvement in networking speed achieved shown in table 5.7. This was especially true for constructed data type like struct and arrays.

Table 5.7: Improvement in networking speed achieved on using IIOP/UDP in comparison with IIOP/TCP

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Improvement in Networking Speed of IIOP/UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct</td>
<td>58%</td>
</tr>
<tr>
<td>char</td>
<td>6%</td>
</tr>
<tr>
<td>long</td>
<td>9%</td>
</tr>
<tr>
<td>bool</td>
<td>42%</td>
</tr>
<tr>
<td>char array (size 100)</td>
<td>58%</td>
</tr>
<tr>
<td>union</td>
<td>24%</td>
</tr>
</tbody>
</table>

5) Comparison of the proposed methods with Existing IDL Compilers

![Comparison of RTT of the proposed methods with existing ORBs](image)

Figure 5.8: Comparison of RTT of the proposed methods with existing ORBs

The performance of the stub code generated by the IDL compiler with the proposed IPC mechanisms is compared with that of three other IDL compilers – Visibroker, ORBacus and Orbix as shown in figure 5.8. The results of the RTT tests are as follows:
a) The relative speed for primitive data types is in the order shared memory, Visibroker, ORBacus, multibuffers, multiple buffers with threads, sockets and Orbix and multiple sockets.
b) For arrays the performance is in the order multiple buffers, multiple buffers with threads, Visibroker, ORBacus, Shared memory, sockets, multiple sockets and Orbix.
c) The time to pass an array or sequence depends on the length of the data being passed.
d) There is a constant overhead of invocation, irrespective of argument sizes especially on using threads for multiple buffers and multiple sockets.

The performance of the stub code on different transmission mediums has been evaluated and it is found that shared memory and multiple buffers perform well. The order of performance is shared memory, multiple buffers, multiple buffering using threads, sockets and multiple sockets. It is observed that using shared memory, the networking and the number of copy operations is reduced. Around 45% of the total time is spent in context switching of the threads when multithreading is used. While sending arrays using shared memory, around 50% of RTT is spent on marshalling. While using sockets, less time is spent on marshalling when compared to networking, since all the parameters are transmitted.

A reuse analysis is conducted on the extent to which the IDL compiler modules used for generation of stub code for CORBA/IIOP in Java can be reused for the generation of stub code for CORBA/IIOP in C++. Totally there are three stages in compilation. Out of this the front end is completely reused for the generation of PT file. In the presentation generator about 60% of the code is reused. The following reuse measures are evaluated:

a) **Reuse leverage**: It is concerned with the number of reused objects.

\[
\text{Reuse leverage} = \frac{\text{Number of objects reused}}{\text{Number of objects built}} = \frac{1}{3} = 0.33
\]
b) **Reuse percentage:** It is concerned with the number of reused lines of code.

Reuse percentage = (Reused lines of code) / (Total lines of code)

= 45%

c) **Quality improvement:** It is due to the reusability of code. The defect rate of reused code is around 0.9 defects/KLOC and for newly developed components, it is around 5.1 defects/KLOC [132]. So when the components are reused, the quality of the system improves as the defect rate is reduced. The quality of the system is improved by 27% due to the reuse of the code.

d) **Productivity** of the system is also improved, since the number of hours needed to complete the coding for the compiler for different target languages is reduced, due to the reuse of the different modules.

This componentized design of IDL compiler not only aids in generation of stub code which facilitates efficient communication but also brings about interoperability between CORBA and DCOM/RMI as explained in section 5.3.

5.3 Proposed Method of Static Interoperability between CORBA and DCOM

The middleware architectures COM and CORBA differ in the following ways:

- CORBA is a specification in the form of IDLs given by OMG. It is an open standard. COM is an implementation for Microsoft Corporation. Thus CORBA objects are portable across platforms, while COM is not.
- CORBA objects use OMG IDL while COM objects use Microsoft IDL.
- In CORBA the objects interact through the ORB with IIOP as the basic communication medium. In COM, Object Remote Procedure Call (ORPC) is used.
- A unique OR references CORBA objects. COM objects are accessed through a pointer to one of the objects interfaces. Interfaces and their
implementations in COM are identified through their Interface Identifier (IID) and Class Identifier (CLSID).

- The lifespan of the COM object is controlled by reference counting whereas the ORB controls the CORBA object's lifespan.
- The base of all interfaces in COM is IUnknown. It returns a pointer to the interface of the CORBA object. The base of all CORBA objects is CORBA::Object. It performs object registration, OR generation and skeleton instantiation.
- The functionality of multiple inheritances in CORBA is done using multiple interfaces to an object in COM.
- While CORBA uses Interface repositories (IFR) to store definitions of data types and interfaces, COM uses type libraries.
- While CORBA interfaces can be updated without changing its repository identifier, COM interfaces need a new identifier.

Due to these differences, a bridge is needed to transparently couple objects from both the models. The functionality of the bridge includes the following:

- Transparent access to objects in other models.
- Usage of data types in one model as native types in another model.

Since CORBA is platform independent, IIOP is used as a communication medium between the two models. Bridging can be of two types – static and dynamic. A dynamic bridge serves as a generic bridge that maps at runtime. It uses IFR and dynamic invocation interface (DII) at the CORBA side and typelib at the COM side. Using this method, there is no need to update the bridging software and it occupies less memory. But this bridge is complex and has poor performance. So a static bridge is used. A proxy is generated from the IDL by the IDL compiler. It takes care of marshalling the code and making the calls between both the systems. It is interface-specific and is suitable for applications whose interfaces are well defined. This method is less complex and has better performance.
Bridging requires a view object. A client in a COM environment can interact with a CORBA target through a COM view object and vice versa. The client treats the view object like a real object. It aids in providing transparent communication to the target object through IIOP. The view object is a component of the COM-CORBA bridge and can be located remotely/locally in the COM environment as shown in figure 5.9.

Ronan [163] and Mike [164] discuss the factors influencing the design of COM-CORBA bridges to establish communication between COM and CORBA objects. Qiwang [165] presents a detailed study on the interoperability between DCOM and CORBA objects. Inprise Application server can be used for integration of COM, CORBA and EJB applications[166]. CORBA – DCE interoperability based on CORBA specifications was established by Vandana [167], while CORBA – COM interoperability based on CORBA specifications was established by Edwin [168], DCOM clients cannot use CORBA objects, and CORBA clients cannot utilize DCOM objects, due to incompatible object system infrastructures. Bridges can bind clients to services at compile time or support dynamic client-server bindings. It is found that static bridges are on the order of five times faster than dynamic bridges. Hence our implementation uses a static bridge. The proposed design of the IDL compiler facilitates it, by mapping the COM and CORBA IDLs onto an intermediate representation from which stubs and skeletons are generated.

---

![Figure 5.9: COM/CORBA interworking with a local bridge](image)

---
The static bridge is designed with the following characteristics:

- A generic IDL compiler has been designed to map COM and CORBA IDLs to an intermediate representation from which stubs and skeletons are generated.
- The location of the bridge is local to the COM environment.
- COM/CORBA interworking specification is used to handle translation of types and OR between the two models.
- The implementation repository and registry entries are cached for faster lookup.
- Threading is used to allow multiple clients to contact a single CORBA server using IIOP/TCP as a communication medium.
- IIOP/UDP is also tried as a communication medium to provide multicasting and to improve the networking speed.

5.3.1 COM/CORBA interworking specifications

In conformance with the COM/CORBA interworking specifications given by OMG the following issues have to be considered for the design of the static bridge:

a) **Interface mapping**: The mapping of CORBA IDL to COM IDL takes care of the following:
   - Mapping of interfaces, primitive and constructed data types.
   - Mapping of CORBA ORs to COM interface pointers and viceversa.
   - Mapping of multiple interface inheritance in CORBA to multiple interfaces in COM
   - Mapping of CORBA attributes to get and set methods in COM

b) **Interface composition mapping**: CORBA interfaces that inherit from a single parent are mapped to an MIDL interface that derives from the mapping for parent interfaces. CORBA's multiple inheritance maps onto a single inheritance model, producing a single linear collection of interface elements.
c) **Lifetime comparison**: COM objects are reference counted and destroyed when clients refer to them. CORBA objects decouple and remove themselves from memory. They have to be explicitly destroyed.

d) **Binding Objects**: For interworking with CORBA objects, the COM views are created and registered in active object registry. Using GetObject function in ICORBAfactory, CORBA objects are mapped to COM views. COM views of CORBA object expose ICORBAFactory interface. It creates new CORBA object instances and binds to existing OR by name. It creates COM view, creates CORBA objects and binds to the view and returns the bound view to the caller.

For interworking COM objects to CORBA views, SimpleFactory interface is designed to map COM to IClassFactory, allowing CORBA clients to create and bind to COM objects. It maps COM objects to CORBA views. IClassFactory has CreateInstance function to create COM objects. The above-mentioned specifications are implemented in our model using the static bridge and a generic IDL compiler.

5.3.2 **Generic IDL Compiler**

The front end of the IDL compiler (figure 5.1) scans and parses the COM and CORBA IDLs and generates an intermediate interface notation called as the Parse Tree (PT). The mappings from COM and CORBA IDLs to PT format is shown in table 5.8. This intermediate file as discussed in section 5.2 is used to map multiple IDLs (COM/CORBA) and to produce a COM/CORBA view of the object as shown in figure 5.10. A module to scan COM IDL and map onto PT is implemented. The back end of the IDL compiler produces the stubs and the view objects as shown in figure 5.10.
Figure 5.10: Proposed Model of Static COM-CORBA

Table 5.8: Mapping different IDL declarations to PT

<table>
<thead>
<tr>
<th>CORBA IDL type</th>
<th>COM IDL Type</th>
<th>C++</th>
<th>PT-def type</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>void</td>
<td>void</td>
<td>PT_VOID</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>int</td>
<td>PT_INTEGER</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td>char</td>
<td>PT_CHAR</td>
</tr>
<tr>
<td>enum</td>
<td>enum</td>
<td>enum</td>
<td>PT_ENUM</td>
</tr>
<tr>
<td>octet</td>
<td>byte</td>
<td>short</td>
<td>PT_SCALAR</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>int</td>
<td>PT_INTEGER</td>
</tr>
<tr>
<td>unsigned short</td>
<td>unsigned short</td>
<td>unsigned int</td>
<td>PT_INTEGER</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>int</td>
<td>PT_INTEGER</td>
</tr>
<tr>
<td>unsigned long</td>
<td>unsigned long</td>
<td>unsigned int</td>
<td>PT_INTEGER</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>float</td>
<td>PT_FLOAT</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>double</td>
<td>PT_FLOAT</td>
</tr>
<tr>
<td>long double</td>
<td>long double</td>
<td>long double</td>
<td>PT_FLOAT</td>
</tr>
<tr>
<td>fixed</td>
<td>Non conformant arrays</td>
<td>array</td>
<td>PT_ARRAY</td>
</tr>
<tr>
<td>Non fixed length array</td>
<td>Conformant arrays</td>
<td>array</td>
<td>PT_ARRAY</td>
</tr>
<tr>
<td>string</td>
<td>LPSTR[string,unique]char *</td>
<td>String</td>
<td>PT_ARRAY of PT_CHAR</td>
</tr>
<tr>
<td>wstring</td>
<td>BSTR, LPWSTR[string,unique]char */string</td>
<td></td>
<td>PT_ARRAY of PT_CHAR</td>
</tr>
<tr>
<td>struct</td>
<td>struct</td>
<td>struct</td>
<td>PT_STRUCT</td>
</tr>
<tr>
<td>union</td>
<td>union</td>
<td>union</td>
<td>PT_UNION</td>
</tr>
<tr>
<td>any</td>
<td>VARIANT</td>
<td>Void *</td>
<td>PT_TYPED</td>
</tr>
<tr>
<td>typedef</td>
<td>typedef</td>
<td>none</td>
<td>PT_INDIRECT</td>
</tr>
</tbody>
</table>
5.3.3 Mapping between OR and UUID

This section explains the mapping procedure of COM clients to CORBA server and CORBA clients to COM server.

a) COM client and CORBA server: Once the server is started, it caches the implementation repository at a particular port and address and sends the details to the clients. To create a new CORBA object, the COM client contacts the CreateObject method in ICORBAFactory. It then contacts the QueryInterface method in IUnknown, which connects to the implementation repository, passes the interface name and returns the port number and address (OR) in which the server object is located. It also spawns a new process for the object. The OR is then sent through IUnknown and ICorbaFactory to the COM client as shown in figure 5.11.

b) CORBA client and COM server: Once the COM server is started, it caches the registry details at a particular port and address and sends the details to the clients. To create a new COM object, The CORBA client contacts the CreateObject method of ISimpleFactory, which contacts the
registry, creates a COM instance through ClassFactory and returns the UUID to the CORBA client through ISimpleFactory as shown in figure 5.12. It also spawns a new process for the COM object.

Once the object reference has been obtained by the view for the target CORBA object and the CORBA object has been created, the stub and skeleton generated by the IDL compiler takes care of all the marshalling and unmarshalling details. Similarly, once the UUID of the target COM instance has been returned, the stub and skeleton take care of marshalling and unmarshalling the parameters.

On the server side, a linked list has been implemented to cache OR/UUID. The server listens to a particular port for a request from the client. On getting the interface name, it looks up the cache and returns OR/UUID. It spawns a new process for that object and resumes listening for the other client requests. On receiving client request in CDR format, it extracts the values and calls the function and marshals back the value to the client. The client code creates an object for ICorbaFactory/ISimpleFactory. It calls CreateObject method to get the OR/UUID for that object. It then performs the
operation using this identifier. The operations in stub code marshal the arguments and send it to the server.

5.4 Proposed Method of Static Interoperability between CORBA and RMI

The back end of the componentized IDL compiler can generate stubs for RMI, RMI/IIOP and CORBA as shown in figure 5.1. The following files are produced by the back end, depending upon the communication protocol:

1) CORBA/IIOP: The stub code produced has the following files:
   - Helper class files are used to check for the variables or the array limits.
   - Holder class files are produced only if the operations have an OUT or INOUT parameter
   - Stub class file takes care of marshalling and unmarshalling information
   - Implementation base class takes care of invocation and networking information

2) RMI/JRMP: The stub file produced serves as a client side proxy for the remote object. It performs functions such as initiating the call and marshalling the arguments to the marsh stream.

3) RMI/IIOP: The stub code produced has the following files:
   - Stub class file: It performs the same functions as RMI/JRMP.
   - Tie class file: It is an IIOP protocol server-side entity that contains a method, which dispatches calls to the actual remote object implementation.

RMI provides ease of programming, but lacks interoperability because it uses JRMP for communication. Since legacy software is programmed in languages other than Java, application developers need CORBA. The main advantage of CORBA is its interoperability. Since CORBA works over IIOP protocol, it cannot interoperate with RMI/JRMP as shown in figure 5.13. But RMI/IIOP supports dual export of RMI/IIOP objects to both JRMP and IIOP.
simultaneously and hence promotes interoperability between RMI/JRMP and CORBA/IIOP as shown in figures 5.14 and 5.15.

Figure 5.13: RMI and CORBA lack direct interoperability

Figure 5.14: RMI and CORBA interoperability through RMI/IIOP

Figure 5.15 RMI and CORBA static Interoperability through RMI/IIOP
Thus interoperability between the different models can be achieved using the proposed compiler as shown in figure 5.1. Further, an equivalent interface for an IDL can be obtained from the presentation generator, which can be used to create stubs and skeletons for all three different distributed object models.

5.5 Automated Generation of Time and Space Efficient Stub Code

CORBA offers two different invocation mechanisms. They are Static Interface Invocation (SII) and Dynamic Interface Invocation (DII) mechanisms. Dynamic object invocation is through Dynamic Invocation Interface on the client side and Dynamic Skeleton Interface on the server side. DSI receives requests dynamically from the client. Thus it is generic and flexible. Its stubs and skeletons occupy less space, but it is slow as type checking occurs at run time. The static invocation of the remote object occurs through the stubs and skeleton generated by the IDL compiler. It uses Static Invocation Interface (SII) and Static Skeleton Interface (SSI). They are defined by the IDL and are tied to the specific object. Hence it is less generic and is rigid. It occupies more space, since it is generated for all the operations in the IDL file, but it is fast.

There are three different SII mechanisms. They vary in execution speed and size of the stubs. They are described below:

i) **Interpretation:** For interpreted code, the stub compiler translates a composite type into a set of commands, one for each component. The stub code generated by the compiler is compact, generic and unified. So more conversion instructions can be kept in the instruction cache. This saves the memory access time. Its main disadvantage is its low execution speed especially when the data is a deeply nested constructed type. Further, using these stubs causes larger amount of data flows through the data cache. Additional cost is also incurred by executing extra instructions.
ii) **Compilation:** Procedure – driven code translates type declarations into procedures that are fast but occupy more space. So for systems like distributed embedded systems with restricted memory sizes, compiled marshalling routines can reach code sizes that are unacceptable. In the compiled stubs, instruction cache has multiple copies of the same code if remote invocations take place at a high rate. Also, it is less generic as it does not support data whose type is not known at compile time.

iii) **Inlining:** Here the encoding/decoding procedures are inlined. Hence the speed is very high, but its size becomes prominent. The increased code size due to inlining causes more cache miss ratio.

OpenFusion e*ORB C Edition (CE) and ORBExpress do not possess DII and DSI features. IONA Orbix treats DII and SII as two different invocation mechanisms. DII implementations must be optimized for performance-sensitive applications on high-speed networks [143]. In addition, the CORBA 2.0 DII specification must be improved to support application portability and optimal performance. For DII, it is desirable to permit reuse of requests if the operations are one-way and the parameter values do not change [169][170]. Just-in-time compilation techniques [171] can be used to improve speed. Here switching between compiled / interpreted code can be also done manually or automatically. Pitmann [124] proposes using hybrid code between interpretation and compilation. A two-level programming model with a well-defined boundary is used to benefit both from the code density of interpreted virtual machines and from the speed of native code execution without considering inlining and DII mechanisms. It requires manual intervention for optimisation. Hoschka [172][173][174] discusses about using control flow analysis to generate time and space efficient stubs using greedy methods. It aims to strike a balance between interpretation and compilation. A Markov model in combination with a heuristic branch predictor is used for estimating execution frequencies. By investing only 25% of the code size of fully optimized code, a performance improvement of 55% to 68% can be achieved. Since greedy method is used for optimization, it does not always produce optimal results. Only time objective is considered and it is used to generate
stubs under space constraint. Hence the proposed optimizer in the presentation generation stage of the IDL compiler uses genetic algorithms.

5.5.1 Optimization in Presentation Generation

The proposed model of the IDL compiler as shown in figure 5.1 is adopted to bring about time and space optimisation. An optimizer is incorporated in the presentation stage to decide the operations that are to be compiled, interpreted or inlined under static invocation mechanism and the operations that are to be called dynamically. It refers to the interface repository to get and update the number of times a type is accessed for SII as shown in figure 5.16. The interface repository of CORBA systems maintains the operation signatures which can be used for type checking for DII and the dynamic frequency count. The static frequency count is obtained from the parse tree generated by the front end. The optimization can be done using a single objective or using multiple objectives. Single objective optimizer generates the hybrid stub code using time objective with space constraint, whereas the multiple objective optimizer uses both time and space objectives.

![Figure 5.16 : Optimisation in Presentation Generator](image)

5.5.2 Estimation of Hybrid Conversion Routines

The criteria to be considered to estimate the functionality of hybrid conversion routines are listed below:

1) **Estimation of time and space benefits**: The nodes in the parse tree represent various data elements. The types of these data elements can be
primitive or composite. There is no difference in speed and the size of the code generated by the compiled and interpreted routines for primitive types. The differences between compiled and interpreted stubs (table 5.9) when a composite data type is marshaled are listed below:

- An interpreter marshals all the composite types by splitting into its primitive types.
- The interpreter template requires one command for the constructor type and one command for each field of the structure. Thus the time difference between compilation and interpretation is $1+n$, where $n$ is the number of elements in the structure.
- For a union, only one of the elements of the union is used at run time. So the time difference between interpretation and compilation is 2.
- Each element of the array is marshaled individually. So the time difference is $1+i$, where $i$ is the number of elements in the array.

<table>
<thead>
<tr>
<th>Type constructor</th>
<th>Time savings due to compilation ($\Delta t$)</th>
<th>Space savings due to interpretation ($\Delta s$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>$1+n$</td>
<td>$1+n$</td>
</tr>
<tr>
<td>Union</td>
<td>2</td>
<td>$1+n$</td>
</tr>
<tr>
<td>Array</td>
<td>$1+i$</td>
<td>2</td>
</tr>
</tbody>
</table>

where,

- $n$ - number of elements in the structure/union
- $i$ - array size

For a structure and union the code template of the constructor node and all its fields are included in the stub code when compiled, whereas the code template of only the constructor node is included in the stub code when it is interpreted. Thus the compiled code occupies an extra space for each element of the structure. The space difference between compilation and interpretation in the case of a structure is $1+n$, where $n$ is the number of elements in the structure. Each element of the array is marshaled individually. So the space difference between compilation and interpretation is 2.
2) Estimation of frequency of occurrence of a data type: The static frequency count is obtained from the parse tree generated by the front end using a conversion flow graph. Since the behavior of the presentation conversion routine exhibits locality, wherein most of the runtime is spent in a small number of nodes in the flow graph, an automatic prediction of the frequency of the types is obtained from an analysis of the interface specifications. A conversion flow graph represents the flow of control between the individual type conversion routines that are invoked for the conversion of a specific composite type. Nodes in the graph represent individual conversion routines and the arcs represent the transfer of control between the conversion routines. The conversion of each node is determined by the frequency with which a node is visited multiplied by the profit due to time savings in compilation. The overall cost of presentation conversion is obtained by summing the conversion cost of each node. For example, let us consider a CORBA IDL file as shown in figure 5.17. The equivalent conversion graph is represented in figure 5.18. From this graph, the frequency with which a node is visited is calculated as shown in table 5.10.

Figure 5.17 : An Example CORBA IDL to estimate frequency of occurrence of a data type
Table 5.10: Estimation of frequency for different data types in figure 5.13

<table>
<thead>
<tr>
<th>SNo</th>
<th>Name</th>
<th>Type</th>
<th>Frequency</th>
<th>profit</th>
<th>Space constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>st1</td>
<td>struct</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>st2</td>
<td>struct</td>
<td>4</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>st3</td>
<td>struct</td>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>st4</td>
<td>struct</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>st5</td>
<td>Struct</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>ut</td>
<td>union</td>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>st2-arr</td>
<td>array</td>
<td>1</td>
<td>13</td>
<td>2</td>
</tr>
</tbody>
</table>

3) Design of the optimization algorithm: Two different genetic algorithms have been proposed. They are based on single objective and multiple objectives. In single objective genetic algorithm, time objective is considered under space constraint. In multiple objective genetic algorithm, both time and space objectives are considered. These algorithms are discussed in sections 5.5.3 and 5.5.4.
Each gene of the chromosome is represented in ‘0’ or ‘1’. ‘0’ represents that the type is interpreted and ‘1’ represents that the type is compiled. In SOGA, when all the genes are compiled for the example CORBA IDL given in figure 5.17, the structure of the chromosome is “1111111” and the extra space needed by compiled code is 28 (calculated from the sum of the elements in space constraint column of table 5.10). So the chromosomes are chosen with maximum profit with space constraint of 14. Structure of a randomly generated chromosome for the example in table 5.17 is “0010010” with profit 9 and space constraint of 14. The structure of the best chromosome generated is “1101101” with a profit of 39 and constraint of 14. From this around 30% of the data types specified by the genes in the chromosome is interpreted and 70% is compiled when SOGA is used. The result is obtained after 14 generations.

In MOGA, when all the genes are compiled, the structure of the chromosome is “1111111”. Structure of a randomly generated chromosome for the example in figure 5.17 is “1010110” with fitness 0.325. The structure of the best chromosome generated is “1100001” with a fitness of 0.7. From this it can be seen that for example given in figure 5.17, 65% of the genes in the chromosome is interpreted and 35% is compiled when MOGA is used. This result is obtained after 14 generations. In table 5.11 we can see that MOGA gives a better fit solution when compared to SOGA method.

Table 5.11: Chromosome Structure for CORBA IDL in figure 5.17

<table>
<thead>
<tr>
<th>SNo</th>
<th>chromosome</th>
<th>SOGA Profit</th>
<th>Constraint</th>
<th>MOGA fitness</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111111</td>
<td>0</td>
<td>28</td>
<td>0.3</td>
<td>Fully compiled</td>
</tr>
<tr>
<td>2</td>
<td>0000000</td>
<td>0</td>
<td>0</td>
<td>0.7</td>
<td>Fully interpreted</td>
</tr>
<tr>
<td>3</td>
<td>0010010</td>
<td>9</td>
<td>14</td>
<td>0.41</td>
<td>Randomly generated chromosome - 75% interpreted and 25% compiled</td>
</tr>
<tr>
<td>4</td>
<td>1101101</td>
<td>39</td>
<td>14</td>
<td>0.59</td>
<td>Best chromosome for SOGA - 71% interpreted and 29% compiled</td>
</tr>
<tr>
<td>5</td>
<td>1010110</td>
<td>16</td>
<td>9</td>
<td>0.325</td>
<td>Randomly generated for MOGA - 43% interpreted and 57% compiled</td>
</tr>
<tr>
<td>6</td>
<td>1100001</td>
<td>33</td>
<td>20</td>
<td>0.7</td>
<td>Best chromosome for MOGA - 65% interpreted and 35% compiled</td>
</tr>
</tbody>
</table>
5.5.3 Proposed Optimisation using SOGA

The criteria to be considered to automate the generation of hybrid conversion routines are listed below:

1) **Estimation of time benefits:** An estimate of time benefit of compilation over interpretation is done in section 5.5.2.

2) **Estimation of frequency of occurrence of a data type:** An estimation of frequency of occurrence of a data type is discussed in section 5.4.2.

3) **Design of the optimization algorithm:** The compiled stub code gives time savings and the interpreted stub code gives space savings. The proposed method uses a combination of compilation and interpretation, to produce an efficient stub code. The objective of the optimization problem is to achieve a maximum speed under the constraint of fitting the elements into a given memory space. The memory space is considered as the knapsack. So the optimization problem reduces to a single objective 0/1 knapsack problem.

The total size allocated for the compiled presentation conversion routine is considered as the size of the knapsack. The sack is fitted to half its capacity with code for compiled data types. The rest of the capacity of the sack is filled with code for interpreted data types.

The optimization problem is formalized using the following variables:

- \( C \) : capacity of the knapsack. It is the size constraint when all the code is compiled.
- \( S_{\text{opt}} \) : total size of the presentation conversion code after optimization. It is the total size of the compiled code and interpreted code.
- \( T \) : total execution time of the presentation conversion code before optimization.
- \( T_{\text{opt}} \) : total execution time of the presentation conversion code after optimization (combination of compiled and interpreted code).
- \( s_i \) : size of the interpreted code template for node \( i \).
sc : size of the compiled code template for node i.

tc : compiled code time for marshalling node i.

ti : interpreted code time for marshalling node i.

fi : frequency of occurrence of node i in the interface specification tree. If a node i occurs n times then it is included as n genes in the chromosome or as n individual items in the knapsack.

xi : 1 if node 'i' is compiled;
0 if node 'i' is interpreted.

The objective of the optimization problem is to generate marshalling routine in such away that it minimizes T_{opt} under the constraint that S_{opt} does not exceed a given a maximal code size.

Thus 0/1 knapsack problem is written as,

\[ S_{\text{opt}} = \sum_{i=1}^{n} x_i s_c + (1-x_i) s_i \] \hspace{1cm} (5.1)

\[ T_{\text{opt}} = \sum_{i=1}^{n} f_i^*((x_i t_c + (1-x_i) t_i) \] \hspace{1cm} (5.2)

Each node in the parse tree has a profit of \( p_i = (x_i t_c + (1-x_i) t_i) \). The aim of the optimization problem is to maximize \( T_{\text{opt}} \) subject to the constraint

\[ \sum_{i=1}^{n} s_c x_i \leq C_1, \text{ where } C_1=C/2. \] \hspace{1cm} (5.3)

Since only relative performance of compiled and interpreted code is needed, the compiled code size \( C_1 \) is restricted to half of the maximum capacity (C) of the knapsack.

The following methods are used to solve the knapsack problem:

a) **Greedy method**: In this method, the elements are arranged in the knapsack in such a way that they are sorted by the speed/size ratio. The elements are inserted into the knapsack by continually going through the list.
of elements following the critical element and including each element that fits into the knapsack capacity. The greedy policy of taking as many as possible items may leave an empty space in the knapsack, too small to accommodate the remaining elements, but large enough for the overall value density to be well below that of other elements. So the greedy method may sometimes reach the local maximum instead of the global maximum.

**b) Exhaustive method:** In this method, all the combinations of elements are selected and their speed/size ratio is evaluated and the best combination is chosen. Since this method is exhaustive, it is time consuming. This method does not work for large IDLs due to the exponential number of combinations.

**c) Proposed Method using Genetic algorithms:** Since the knapsack problem has NP hard characteristic, a genetic algorithm can be applied to solve the problem. This is faster than the exhaustive method and provides a better solution than the greedy method.

The fitness function to be maximized is

\[ \sum_{j=1}^{i=n} f^*(x_i t_c + (1-x_i) t_b) \]  \hspace{1cm} \text{(5.4)}

Subject to the constraint

\[ \sum_{j=1}^{i=n} (s_c x_i) \leq C_1, \text{ where } C_1=\frac{C}{2}. \]  \hspace{1cm} \text{(5.5)}

where, C is the total size of presentation conversion routine when compiled.

Genetic algorithms create a string of numbers that represent the solution. Each data type is represented as a gene in the chromosome.

The following steps are carried by the SOGA (figure 5.19) to solve the problem:

1) Generate initial population of chromosomes of random compositions of the genes. 0 represents interpreted type and 1 represents compiled type. Repeat steps 2 and 3 until the maximum number of generations.
Figure 5.19: Flowchart for Genetic Algorithm

- Gen=0
  - Create initial random population
  - The length of the chromosome (L) is the number of data types in IDL
  - Number of chromosomes (N) in each generation is twice the length of the chromosomes
- Evaluate fitness of each individual
  - Gen=Gen+1
  - Select 10% best individuals based on the fitness - elitism
  - Copy into new population
  - Select randomly two individuals
    - Perform crossover with 65% probability
    - Insert cross-over offspring into new population
  - Select an individual randomly
    - Perform mutation with 1% probability
    - Insert mutant into new population
  - Choose 'n' best individuals after selection, crossover and mutation
- Termination? (best fit (Gen-1) = best fit (Gen))
  - Yes
    - Designate result
    - End
  - No

2) Evaluate the fitness function and the constraint for each chromosome in the population.

3) Create a new population of chromosomes by doing the following:
   a) Elitism method of selection of chromosomes is used.
   b) Create new chromosomes by using one point cross over and random cross over point with cross over probability 0.65.
   c) Create a new chromosome by mutation. Mutation is done with probability 0.01, when populations of some of the consecutive generations do not vary much.

4) The best chromosome gives the solution to the problem.

   The structure of a randomly generated chromosome and the final chromosome which gives the solution to the problem in figure 5.17 is shown in table 5.11. From the structure of the final chromosome, it can be seen that structures st1, st2, st4, st5 and array st_arr are to be compiled, whereas st3 and union ut are to be interpreted. This accounts for 70% compilation and 30% interpretation for the example given in figure 5.17.

5.5.4 Proposed Optimisation using MOGA

   Faster convergence to optimal solution can be achieved by considering both time and space objectives and using domain specific heuristics. The criteria to be considered to automate the generation of hybrid conversion routines are listed below:

1) Estimation of time and space benefits: An estimation of time and space benefits of the hybrid code generated by the IDL compiler is done as given in section 5.5.2.

2) Estimation of frequency of occurrence of data types: An estimation of the frequency of occurrence of data types is done as presented in section 5.5.2.
Design of the optimisation algorithm: The proposed method uses a combination of compilation and interpretation, to produce an efficient hybrid stub code. The objective of the optimization problem is to achieve a tradeoff between the speed and size of the stub code generated by the IDL compiler. The memory space is considered as the knapsack. The objectives are time savings due to compilation and space savings due to interpretation. So the optimization problem reduces to a multiple objective 0/1 knapsack problem.

The objective function \( F \) is designed in such a way that for compiled type, the speed difference is taken and for interpreted types the size difference is taken.

\[
F = w_1 \sum_{i=1}^{j=n} \Delta s_i / S_{\text{max}} + w_2 \sum_{j=1}^{j=n} \Delta t_i / T_{\text{max}}
\]

\[
\forall i \ni x_i=1, \forall j \ni x_j=0 \text{ and } \sum_{i=1}^{j=n} \Delta t_i < T_{\text{max}} \text{ and } \sum_{j=1}^{j=n} \Delta s_j < S_{\text{max}} \ldots \ldots \ldots (5.6)
\]

where,

- \( s_i, s_c \) is the size of interpreted and compiled code templates.
- \( t_i, t_c \) is the execution time for interpreted and compiled code templates.
- \( f_i \) is the frequency of node \( i \) in the interface tree.
- \( \Delta s_i \) is the space saving due to interpretation.
  \[
  \Delta s_i = s_c - s_i \quad \ldots \ldots \ldots (5.7)
  \]
- \( \Delta t_i \) is the time saving due to compilation
  \[
  \Delta t_i = f_i \ast (t_i - t_c) \quad \ldots \ldots \ldots (5.8)
  \]
- \( S_{\text{max}} \) is the maximum size difference achieved when all the code is interpreted.
- \( T_{\text{max}} \) is the maximum time difference achieved when all the code is compiled.
- \( x_i = 1 \) if the node is compiled
  \[
  = 0 \text{ if the node is interpreted}
  \]
- \( w_1 \) and \( w_2 \) are weight vectors with the following relationship
w_2 = (1 - w_1) \quad \ldots \ldots \ldots \ldots (5.9)

Using this method, it is not needed to know the maximum capacity to fit the stub code since a tradeoff between the speed and size is achieved in the objective function itself. Genetic algorithms can be used to solve NP hard problems like 0/1 knapsack problem. Gottleib [175] explains the various genetic algorithms that can be used for constrained optimisation problems. Deb [176] explains about multiobjective optimisation using evolutionary algorithms. Optimisation [177] can be done using genetic algorithms for conventional programming languages. CM-SRM [178] technique can be used for faster convergence of the results. The performance of the transposition operator is better than a cross-over operator for a 0/1 knapsack problem [179]. On applying varying mutations [180] in parallel for deterministic and self-adaptive varying mutation GAs the performance of the 0/1 knapsack problem can be enhanced.

Two different Genetic Algorithms were studied and domain specific heuristics were applied to bring about fast convergence of the results. They are explained below:

a) Multiple Objective Genetic Local Search (MOGLS) method: MOGLS as shown in figure 5.20 is used to strike a balance between code size and execution speed.

The fitness function to be maximized is given by:

\[
w_1 \cdot \sum_{j=1}^{n} \Delta s_j / S_{\text{max}} + w_2 \cdot \sum_{j=1}^{n} \Delta t_j / T_{\text{max}}, \forall i \geq x_i = 1, \forall j \geq x_j = 0 \quad \ldots (5.10)
\]

where, \(w_1 = 1 - w_2\). \(w_1\) and \(w_2\) are weights. The weight vector is \(\lambda(w_1, w_2)\).

MOGLS is a meta heuristic algorithm that hybridizes recombination operators with local heuristics. Its goal is to generate a good approximation to the non-dominated set. A population is randomly generated, and 1% of the genes having best \(\Delta t\) value is compiled to generate CS. It is a local domain specific heuristic. A random weight vector is assigned to the \(i^{th}\) solution in the population and its fitness is calculated. In this way, all the population
members are assigned fitness. From this current state (CS) population the Temporary Elite Population (TEP) is chosen. Crossover and mutation (CM) are performed on TEP with probabilities 0.65 and 0.1 respectively. In CM, 10% of TEP having best Δs is interpreted (domain specific heuristic). This is the mutation operation. Each solution (x) of the new TEP is compared with the solutions in TEP. If it is better than the worst solution in TEP, then it is added to CS. The potentially non-dominated points (PP-potentially pareto-optimal solution) is updated with the new solution (x) by

1. Adding x to PP if no solution in PP dominates x.
2. Removing from PP all solutions dominated by x.

In this method,

- The diversity in the non-dominated solution is maintained by using random weight vectors for each solution, thereby emphasizing solutions, which may lead to a different solution in the Pareto-optimal region.

- Faster convergence is obtained by using domain specific heuristics.

Figure 5.20: Multiple Objective Genetic Local Search

b) Multiple Objective Crossover and Mutation, Self-Reproduction and Mutation (MOCM-SRM) method: We also propose to use MOCM-SRM to generate time and space efficient stubs. MOCM-SRM uses two types of cooperative and competitive genetic operators - Crossover and Mutation (CM)
and Self-Reproduction and Mutation (SRM) to produce offspring and assign them specific roles. Offspring created by both operators compete for survival through an extinctive selection mechanism as shown in figure 5.21.

The crossover-mutation (CM) operator creates offspring by conventional and successive mutation with small mutation probability. Here 1% of the genes having best Δt value is compiled. This operator has the role to propagate beneficial genetic information into the population by combining segments from parent individuals. It creates $\lambda_{CM}$ offspring.

The self-reproduction and mutation (SRM) operator is an adaptive mutation operator with dynamic mutation probability varying from high to low. Here at least 10% of the genes having best Δs value are interpreted. This operator has the role of introducing diversity into the population by creating offspring that cannot be created by CM. It creates $\lambda_{SRM}$ offspring.

Offspring created by CM and SRM compete for survival through ($\mu,\lambda$) proportional selection. It selects best $\mu$ offspring by discarding low fitness value from $\lambda = \lambda_{CM} + \lambda_{SRM}$.

c) Proposed Parallel MO-CM-SRM method: We propose to use MO parallel CM-SRM method to achieve a balance between the size and execution speed of the stub code that is generated. In this method CM and SRM operations are done in parallel using Posix threads (pthreads). Pthreads are light weight
5.5.5 Performance comparison of the genetic algorithms

Consider an example IDL file (figure 5.22 a) given as input to IDL compiler. The front end generates a parse tree from the IDL specifications as shown in figure 5.22 b. This acts as the conversion graph from which frequency, time difference, space difference and profit are calculated. Figure 5.22 c gives the performance of the optimizer using genetic algorithms with respect to the greedy method. From the graph we can observe that within 25 generations it reaches the greedy value and after 100 generations it generates a better solution when compared to the greedy method. Thus the main advantage of this method is that it gives a better solution than the greedy method for large IDL files, even though it takes a little more compile time than greedy method in generating the solution. When compared to the exhaustive method, this method is faster. The input to the optimizer stage is obtained by scanning the IDL file and identifying the type, frequency, $\Delta s$ and $\Delta t$ values for the different elements of the IDL file. The dynamic frequency of execution of the data types in the operation is also obtained from the interface repository.

A comparison of the performance of greedy, exhaustive, MOGLS and MOCM_SRM is shown in the figure 5.23. It is observed that MOCM_SRM performs better than MOGLS and MOGLS performs better than the greedy method. MOCM_SRM also converges faster to the solution than MOGLS. The results obtained through MOCM_SRM are also comparable to the exhaustive method. From figure 5.23, it can be seen that when the number of data types exceed 20, exhaustive method becomes infeasible.
module static_freq
{
    typedef struct stu_id
    {
        string idl;
    } tid;

typedef struct nm
{
    string fn;
    string mn;
    string ln;
} tname;

typedef struct id_det
{
    tname name;
    tid id;
} tid_det;
...
}

Figure 5.22 a: Example CORBA IDL file

Figure 5.22 b: Parse Tree generated by front end of the IDL compiler for the input IDL in figure 5.24 a

Figure 5.22 c: Optimisation using greedy and GA methods
A parallel method of MOCM_SRM uses pthreads to do CM and SRM operations in parallel. Figure 5.24 compares the performance of the sequential and parallel implementations of MOCM_SRM methods. As the parallel implementation of MOCM_SRM is efficient, it is used by the optimizer to decide which operations are to be compiled. The back end generates the stub code based on this decision.

Figure 5.23: Comparison of exhaustive, Greedy, MOGLS and MOCM_SRM

Figure 5.24: Performance of Serial and parallel MOCM_SRM

Figure 5.25 shows the pareto curve in criterion space. It is also known as the trade-off curve. It shows the variation of the two objectives with respect to the number of data types compiled and interpreted. In the figure 5.25, fit0 stands for the first objective, namely, fitness of the data types that are interpreted and fit1.
stands for the second objective, namely, fitness of the data types that are compiled. To obtain an optimal tradeoff, around 70% of the data types are to be interpreted and 30% are to be compiled (figure 5.25).

5.5.6 Modeling the Hybrid invocation of CORBA objects

CORBA offers two different invocation mechanisms, namely, Static Invocation Interface (SII) and Dynamic Invocation Interface (DII). There are three different SII mechanisms. They are interpretation, compilation and inlining. This section proposes a method of generating a hybrid stub code, which is optimized for code space and execution speed. SII mechanisms in turn can be interpretation, compilation and inlining by considering the dynamic frequency of access of the operations and the static frequency of the occurrence of the data types in the interface. MOCM-SRM is used to automate this optimal trade-off. The optimizer is implemented in the presentation generator stage of the compiler. The following criteria have to be considered to automate the generation of hybrid conversion routines:

a) Estimation of time and space benefits

The front end of the IDL compiler generates the parse tree from the CORBA IDL file. Each node in the parse tree represents the data elements in the interface definition. The data elements can be of either primitive or composite type. There is no time or space difference when the primitive types are statically or dynamically called. When called dynamically, an extra time to refer to interface repository (IR) for type checking before marshalling has to be taken into account. A composite type is split into its primitive types before marshalling dynamically. In table 5.12 the columns Δ s42 and Δ t42 represent the difference between statically and dynamically marshaled stubs.
Table 5.12: Comparison of performance of different types of stubs

<table>
<thead>
<tr>
<th>Composite type</th>
<th>Interpreted</th>
<th>Compiled</th>
<th>Inlined</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Delta s^{32}_1$</td>
<td>$\Delta s^{31}_1$</td>
<td>$\Delta s^{21}_1$</td>
<td>$\Delta t^{23}_1$</td>
</tr>
<tr>
<td>Struct</td>
<td>1+n</td>
<td>1+n+c1*n</td>
<td>c1*n</td>
<td>1+n</td>
</tr>
<tr>
<td>Array</td>
<td>2</td>
<td>2+c1*μ</td>
<td>c1*μ</td>
<td>1+μ</td>
</tr>
<tr>
<td>Union</td>
<td>1+n</td>
<td>1+n+c1*n</td>
<td>c1</td>
<td>2</td>
</tr>
</tbody>
</table>

where
- $n$ is the number of elements in the structure.
- $\mu$ is the size of the array.
- $c1$ is a constant which specifies the number of times an encoding procedure is called. Here extra space is needed to place code for each procedure call in cache.
- $c2$ is a constant which specifies the number of times an encoding procedure is called. Each call results in an increase in speed as the code is in cache and the overhead of the procedure call is avoided.
- $I$ is time for data type checking in the interface repository.

For a structure, the code template of the constructor node and all its fields are included in the stub code when compiled. Thus the compiled code occupies an extra space for each element of the structure. Since the dynamic and interpreted stubs require one command for the constructor type and one for each field of the structure, it takes an extra time to execute commands for each field of the structure. Thus there is time difference between Dll and Sll stubs. The union type has the same space difference between Dll and Sll as the structure. Since only one of the elements of the union is used at run time, the time difference between Dll and Sll mechanisms is 2. Each element of the array is marshaled individually. So the space difference between Dll and Sll is 2. A loop is used to execute the command to marshal the elements of the array. Since each element of the array is marshaled by a separate command, an extra time is taken to execute it. So the time difference is 1+n, where $n$ is the number of elements in the array.
The improved performance of the interpreted stubs over compiled and inlined stubs is shown in the columns $\Delta s_{32}$ and $\Delta s_{31}$ of table 5.12. The main advantage of the interpreted stub is its space savings over compiled and inlined stubs. The same argument as the difference between DII and SII stubs holds for interpreted and compiled stubs also. In the case of the inlined stubs an extra space is needed in cache to hold the expanded stub procedures also.

The improved performance of the compiled over inlined and interpreted stubs is represented in columns $\Delta s_{21}$ and $\Delta t_{23}$ of the table 5.12. The advantage of compiled over dynamic code is represented by the column $\Delta t_{24}$. The main advantage of the compiled stub over inlined stub is its space savings, whereas the advantage of compiled stub over interpreted stub is time savings. The improved performance of the inlined stubs over compiled and interpreted stubs is its time savings as shown in columns $\Delta t_{12}$ and $\Delta t_{13}$ respectively.

b) Estimation of the frequency

An optimizer in the stub generator must decide which type of stub should be used for invoking an operation, based on the static frequency of occurrence of the nodes in the parse tree and the dynamic frequency with which the operations are invoked by the clients. The static analysis gives the relative frequency of occurrence of the different nodes for all traces of the program. To take into account the actual frequency of the occurrence of the types, a count of the number of times a type is accessed is maintained in the interface repository. This is accessed by the optimizer in the compiler statically.

c) Optimisation algorithm

The compiled stub code gives time savings and the dynamic stub code gives space savings and is more flexible. Further, the different static invocation mechanisms, like, interpreted, compiled and inlined differ in their time and space requirements. The objective of the optimization problem is to
achieve a maximum speed and size difference under the constraint of fitting the elements into a given memory space. The memory space is considered as the knapsack. The multiple objectives are time difference and space difference. So the optimization problem reduces to a multiple objective 0/1 knapsack problem.

The objective function is designed in such a way that for interpreted and dynamic stubs, the space difference is taken and for inlined stubs the time difference is taken. For compiled type, the speed difference with respect to interpreted and the space difference with respect to inlined are considered.

![Space vs time graph for Sll](image)

Figure 5.26: Space vs time graph for Sll

- \( s_1, s_2, s_3 \) is the size of inlined, compiled and interpreted code templates for node \( i \) (figure 5.26)
- \( s_4 \) is the size of Dll code template for a type \( i \)
- \( t_1, t_2, t_3 \) is the execution time for inlined, compiled and interpreted code templates for node \( i \)
- \( t_4 \) is the execution time of Dll code template for a type \( i \)
- \( f_1 \) is the static frequency of node \( i \) in the interface tree.
- \( f_2 \) is the dynamic frequency with which node \( i \) is called.

\[
f_i = f_1 \times f_2 \quad \ldots (5.11)
\]

- \( \Delta s_{32} \) is the space saving achieved by interpreter with respect to compiler with weight \( w_1 \);
- \( \Delta s_{32} = s_3 - s_2 \quad \ldots (5.12) \)
- \( \Delta s_{31} \) is the space saving achieved by interpreter with respect to inlining with weight \( w_2 \);
\( \Delta s31_i = s3_i - s1_i \) \hspace{1cm} \ldots \ldots (5.13)

\[ \text{fit(3)} \text{ fitness of the interpreter} \]

\[ = w_1 \Delta s32_i / \sum_{i=1}^{n} \Delta s32_i + w_1 \Delta s31_i / \sum_{i=1}^{n} \Delta s31_i \]

\[ \ldots \ldots (5.14) \]

\( \Delta s21_i \) is the space saving achieved by compiler with respect to inlining with weight \( w_3 \);

\( \Delta s21_i = s2_i - s1_i \) \hspace{1cm} \ldots \ldots (5.15)

\( \Delta t23_i \) is the time saving achieved by compiler with respect to interpreter with weight \( w_4 \);

\( \Delta t23_i = f_1 \ast (t2_i - t3_i) \) \hspace{1cm} \ldots \ldots (5.16)

\( \Delta t24_i \) is the time saving achieved by SII with respect to DII with weight \( w_6 \);

\( \Delta t24_i = f_2 \ast (t2_i - t4_i) \) \hspace{1cm} \ldots \ldots (5.17)

\[ \text{fit(2)} \text{ fitness of the compiler} \]

\[ = w_3 \Delta s21_i / \sum_{i=1}^{n} \Delta s21_i + w_4 \Delta t23_i / \sum_{i=1}^{n} \Delta t23_i + w_5 \Delta t24_i / \sum_{i=1}^{n} \Delta t24_i \]

\[ \ldots \ldots (5.18) \]

\( \Delta t12_i \) is the time saving achieved by inlining with respect to compilation with weight \( w_6 \);

\( \Delta t12_i = f_1 \ast (t1_i - t2_i) \) \hspace{1cm} \ldots \ldots (5.19)

\( \Delta t13_i \) is the time saving achieved by inlining with respect to interpreter with weight \( w_7 \);

\( \Delta t13_i = f_1 \ast (t1_i - t3_i) \) \hspace{1cm} \ldots \ldots (5.20)

\[ \text{fit(1)} \text{ fitness of the inliner} \]

\[ = w_6 \Delta t12_i / \sum_{i=1}^{n} \Delta t12_i + w_7 \Delta t13_i / \sum_{i=1}^{n} \Delta t13_i \]

\[ \ldots \ldots (5.21) \]

\( \Delta s42_i \) is the space saving achieved by DII mechanism with respect to SII mechanism with weight \( w_8 \);

\( \Delta s42_i = s4_i - s2_i \) \hspace{1cm} \ldots \ldots (5.22)
fit(4) fitness of the dynamic code

\[ \text{fit}(i) = \sum_{i=n}^{i=n} \Delta s_{42_i} i / \sum_{i=1}^{i=1} \Delta s_{42_i} \] ........(5.23)

\[ \sum_{i=1}^{i=8} w_i=1 \] ........(5.24)

Object function \[ F= \sum_{i=1}^{i=4} \text{fit}(i) \] ........(5.25)

**5.5.7 Design of the proposed Hybrid Invocation Mechanism**

The sequence of steps involved in the proposed hybrid invocation mechanism as shown in figure 5.27 is listed below:

0) The server registers the object in the implementation repository (IMR).

1) When the client refers to a remote object, it searches in the IMR.

2) IMR checks if the object is activated.

3) If not, it activates the object. The address and port number are stored in the IMR.

4) Address and port number of the object is returned to the client.

5) When a client invokes an operation, it checks the interface repository (IR) to see if the operation is to be compiled, inlined, interpreted or dynamically invoked. It also increments the dynamic frequency of access of the operation, which is maintained in the interface repository.

6) The decision on the mode of operation invocation taken in step 5 is returned to the client.

7) The client dynamically invokes the object through DII, Internet Interoperable Protocol (IIOP) and DSI
   a) Client invocation is sent to the DII code. It refers to the interface repository for type checking. It converts the code into Common Data Representation (CDR).
   b) The request passes through IIOP.
c) DSI is a generic skeleton interface. It takes care of unmarshalling the code from CDR to local machine format. It also uses interface for type checking.

8) The client statically invokes the object through compiled stubs
   a) The stubs generated by the IDL compiler, converts the code into Common Data Representation (CDR).
   b) The request passes through IIOP.
   c) The Skeleton takes care of unmarshalling the code from CDR to local machine format.

Figure 5.27: Proposed Hybrid Invocation Mechanism
10) The client statically invokes the object through interpreted stubs
   a) The generic stub code based on the DII library, converts the code into
      Common Data Representation (CDR).
   b) The request passes through IIOP.
   c) The generic Skeleton based on DSI library, takes care of
      unmarshalling the code from CDR to local machine format.

5.6 Performance of hybridized stub code

   This section illustrates the performance of hybrid stub code
   between compilation and interpretation generated by the proposed IDL
   compiler. It also presents the experimental results of the performance of
   the hybrid stub code between SII and DII mechanisms on a CORBA
   marshalling engine and on a RT-CORBA implementation, namely, Zen.

5.6.1 Performance of hybridized stub code between compilation and
   interpretation

   A stub code generator to generate CORBA stub code was developed.
   The measurements have been taken in Linux 7.1 OS. Table 5.13 compares
   the performance of purely compiled, purely interpreted and hybrid stub code
   for an interface having primitive types and an interface with composite types.
   The primitive types tested include character, float, double, short and long. The
   composite types include unions, structures and arrays. The hybrid stub code
   combines interpretation and compilation. It uses MOCM_SRMI to identify
   whether a type is to be compiled or interpreted.

   From table 5.13, it is observed that the speed of the hybrid stub is
   comparable to that of compiled stub, but it occupies less space. This
   difference is more prominent in the case of an interface with composite data
   types.
### Table 5.13: Performance Comparison of Compiled, Interpreted and Hybrid Stub Code

<table>
<thead>
<tr>
<th>Interface Type</th>
<th>Time taken (μsec)</th>
<th>Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interface with primitive types</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiled stub</td>
<td>2082</td>
<td>Skeleton: 4855, Stub: 3190</td>
</tr>
<tr>
<td>Interpreted stub</td>
<td>2855</td>
<td>Skeleton: 1941, Stub: 989</td>
</tr>
<tr>
<td>Hybrid stub</td>
<td>2217</td>
<td>Skeleton: 4332, Stub: 2764</td>
</tr>
<tr>
<td><strong>Interface with composite types</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiled stub</td>
<td>1281</td>
<td>Skeleton: 5215, Stub: 3433</td>
</tr>
<tr>
<td>Interpreted stub</td>
<td>20006</td>
<td>Skeleton: 1941, Stub: 989</td>
</tr>
<tr>
<td>Hybrid stub</td>
<td>1976</td>
<td>Skeleton: 3866, Stub: 2171</td>
</tr>
</tbody>
</table>

### 5.6.2 Performance of hybridized stub code between Static and Dynamic Invocation Mechanisms

Figure 5.28 compares the performance of compiled, interpreted, inlined and dynamically invoked operations having boolean and float arrays of different sizes as their parameters. It is seen that due to the reference made to the interface repository by the dynamic stub code to get the data type to be marshaled, the time taken for method invocation is very large. Interpreted...
invocations take a little less time when compared to Dll, but each element has to be marshaled separately. Compiled and inlined invocation takes much less time.

Figure 5.28 b: Performance of inline, compiled, interpreted and dynamic invocation stubs for boolean arrays of different sizes.

Figure 5.29: Percentage of time spent in encode, networking, demarshalling and other activities for static and dynamic invocation mechanisms

Figure 5.29 shows the percentage of time spent on marshalling, networking, demarshalling and other activities out of the total time in invoking
an object. Other activities include, function invocation time taken from the skeleton code and referring the interface repository to get the data types for marshaling. The measurements have been taken for float array of size 256 for the different methods. Similar results were observed for other data types also. It is seen that in static method (inline, compile and interpret) of invocation less time is spent on other activities. This is mainly because it knows the data types to be marshaled during the compile time itself, and generates the stub code accordingly.

![Figure 5.30: Pareto-optimal Graph for Hybrid Code Generation](image)

Weight calculation for the various objects in the MOGA plays a crucial role in deciding the efficiency of MOGA. For this pareto-optimal graph is used. Figure 5.30 shows the pareto curve in criterion space. It shows the variation of the four objectives with respect to percentage of data types being interpreted, compiled, inlined and dynamically called. From the pareto-optimal graph, the percentage of interpretation, compilation with inlining and dynamic invocation have been indentified as 30%, 25% and 20% respectively, when the interpretation, compilation with inlining and dynamic curves meet the compilation curve. Percentage compilation is got by subtracting the summation of interpretation, compilation with inlining and dynamic from 100%.
Table 5.14 compares the performance of purely compiled, purely interpreted, purely inlined, purely dynamic and hybrid stub code for an interface having primitive types and composite types. This interface is chosen to check the effect of the stub code on different possible combinations of data types that have an effect on the performance. The primitive types tested include character, float and short.

Table 5.14: Performance comparison of hybrid invocation mechanism with standard methods

<table>
<thead>
<tr>
<th>Type</th>
<th>stub code size(KB)</th>
<th>skeleton code size(KB)</th>
<th>time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic invocation</td>
<td>Total: 54.9</td>
<td>Total: 55.6</td>
<td>910</td>
</tr>
<tr>
<td>Interpreted stubs</td>
<td>Total: 54.9</td>
<td>Total: 55.6</td>
<td>890</td>
</tr>
<tr>
<td>Inlined stubs</td>
<td>Stub: 5.38</td>
<td>Skeleton: 6.72</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM: 26.38</td>
<td>RAM: 27.72</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>Total: 104.29</td>
<td>Total: 104.29</td>
<td></td>
</tr>
<tr>
<td>Compiled stubs</td>
<td>Stub: 5.38</td>
<td>Skeleton: 6.72</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM: 25</td>
<td>RAM: 25</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>Total: 104.29</td>
<td>Total: 104.29</td>
<td></td>
</tr>
<tr>
<td>Hybrid stubs with Dll for primitive types</td>
<td>Stub: 4.06</td>
<td>Stub: 4.06</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM: 25.06</td>
<td>RAM: 25.06</td>
<td>590</td>
</tr>
<tr>
<td></td>
<td>Total: 58.06</td>
<td>Total: 58.06</td>
<td></td>
</tr>
<tr>
<td>Hybrid stubs with inlining for primitive types</td>
<td>Stub: 4.06</td>
<td>Stub: 4.06</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM: 85.06</td>
<td>RAM: 85.06</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>Total: 58.96</td>
<td>Total: 58.96</td>
<td></td>
</tr>
</tbody>
</table>

The composite types include arrays, structures and union. Arrays of type float (with size 256) and char (with size 1024) and boolean (with size 256) are included. Three structures have been considered. One structure has primitive data types short, long, char, float and double as its elements. The second structure has arrays of size 10 elements belonging to short, long, char, float, long and boolean as its elements. The third structure is a combination of the first two structures. The union has its members of type boolean, float and short. The hybrid marshalling includes a combination of static and dynamic methods. It uses MOCM_SRM to identify the mode for each operation in the interface. The output of the optimizer has identified that inlined stub code is to be generated for primitive types. Float array and the union are to be interpreted, boolean and character arrays and structures are to be compiled.
From table 5.14, it is observed that the speed of the hybrid marshalling is comparable to that of compiled stub, but it also occupies less space. It is also found that it is much faster than dynamic invocation mechanism. The hybrid stub code generated using this technique occupies less space when compared to compiled code and inlined code. It performs better than the interpreted code and dynamic invocation mechanism.

ZEN is a Real-time CORBA ORB implemented using Real-time Java, thereby combining the benefits of these two standard technologies. Zen’s architecture is based on the concept of layered pluggability. It has eight core ORB services. They are Object adapters, Message buffer allocators, GIOP message handling, CDR Stream readers/writers, Protocol transports, Object resolvers, IOR parsers, and Any handlers. These services are removed out of the ORB to reduce its memory footprint and increase its flexibility. The remaining portion of code is called the ZEN kernel. Each ORB service itself is decomposed into smaller pluggable components that can be loaded into the ORB only when needed. This pluggable design makes ZEN a good research platform, because alternative implementations of various ORB components can be plugged in and profiled with standard benchmarks to determine their utility. The proposed hybrid code generation model was tested over ZEN.

Table 5.15: Details of an example CORBA IDL for Hybrid code generation

<table>
<thead>
<tr>
<th>Size of IDL</th>
<th>Total no. of Operations</th>
<th>Total no. of user defined types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>Medium</td>
<td>24</td>
<td>11</td>
</tr>
<tr>
<td>Large</td>
<td>60</td>
<td>17</td>
</tr>
</tbody>
</table>

The measurements have been taken for three different CORBA IDLs with the number of operations and data types shown in table 5.15. Table 5.16 compares the performance of the hybrid stub code with the performance of the conventional static and dynamic stubs. From the table 5.16, it can be seen that size of dynamic stubs is less when compared to the static stubs. It is also seen that the time for static method of invocations is less when compared to
the dynamic method invocations. In Zen already some of the marshalling procedures are inlined. So when inlining is used for operation calls, only there is a marginal increase in speed as seen in table 5.16. The hybrid stubs take more space when compared to dynamic stubs, but is less when compared to static stubs. Further, it is faster than dynamic stubs. The total time for invoking all the operations (28 msec) sequentially is less when compared to the dynamic method which takes 41.7 msec, but it is more when compared to the static methods. Since the invocations of the server operations can occur in parallel processes for each different invocation methods, the total invocation time can be reduced in hybrid invocation to 9 msec. It is found that the invocation time is much less when compared to the compile method which takes 14 msec. The percentage heap space savings and hard disk space savings achieved by hybrid stubs when compared to conventional compiled stubs is around 54% and 9%. The time savings achieved by hybrid stubs when compared to dynamic stubs is 33%. The parallel invocation of objects further increases the performance of the hybrid stub code by 36% when compared to compiled stubs.

<table>
<thead>
<tr>
<th>Types of invocations</th>
<th>Stubs Size (KB)</th>
<th>Class Size (KB)</th>
<th>Hard Disk Space (KB)</th>
<th>Heap Size (Bytes)</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted</td>
<td>32.6</td>
<td>38.7</td>
<td>71.3</td>
<td>Total: 772432</td>
<td>21.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stub: 888</td>
<td></td>
</tr>
<tr>
<td>Compiled</td>
<td>74.2</td>
<td>70.3</td>
<td>144.5</td>
<td>Total: 773288</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stub: 1912</td>
<td></td>
</tr>
<tr>
<td>Inlined</td>
<td>73.7</td>
<td>70.0</td>
<td>143.7</td>
<td>Total: 773152</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stub: 910</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>48.0</td>
<td>45.2</td>
<td>93.2</td>
<td>Total: 196578</td>
<td>41.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stub: 240</td>
<td></td>
</tr>
<tr>
<td>Hybrid</td>
<td>130.8</td>
<td></td>
<td></td>
<td>Total: 772400</td>
<td>Parallel: 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stub: 883</td>
<td>Total: 28</td>
</tr>
</tbody>
</table>

Table 5.16 : Performance Comparison of Hybrid Stubs with conventional methods for small IDL
Table 5.17: Performance Comparison of Hybrid Stubs with conventional methods for medium sized IDL

<table>
<thead>
<tr>
<th>Types of invocations</th>
<th>Stubs Size (KB)</th>
<th>Class Size (KB)</th>
<th>Hard Disk Space (KB)</th>
<th>Heap Size (Bytes)</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted</td>
<td>347</td>
<td>433</td>
<td>780</td>
<td>Total: 374024</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stub: 51064</td>
<td></td>
</tr>
<tr>
<td>Compiled</td>
<td>225</td>
<td>200</td>
<td>425</td>
<td>Total: 826048</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stub: 29832</td>
<td></td>
</tr>
<tr>
<td>Inlined</td>
<td>221</td>
<td>198</td>
<td>419</td>
<td>Total: 1115296</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stub: 99976</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>99</td>
<td>90</td>
<td>189</td>
<td>Total: 322032</td>
<td>313</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stub: 2784</td>
<td></td>
</tr>
<tr>
<td>Hybrid</td>
<td>410</td>
<td></td>
<td></td>
<td>Total: 795600</td>
<td>Parallel:67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stub: 18056</td>
<td>Total:139</td>
</tr>
</tbody>
</table>

From tables 5.17 and 5.18, it can be seen that the performance of the hybrid stub code is much better when the size of the interface increases. It is found that space savings and time savings achieved using this method is more prominent on using large interfaces. The time savings achieved by hybrid code when compared to dynamic stubs range from 11% (for medium sized IDL) to 56% (for large sized IDL) using this method. The heap space savings achieved in hybrid code when compared to compiled stubs is 44% (for medium sized IDLs) and 36% (for large sized IDLs) based on the size of the interfaces when compared to static methods. The hard disk space savings achieved in hybrid code when compared to compiled stubs is 4% for both medium and large sized IDLs based on the size of the interfaces when compared to static methods. The parallel invocation of objects, further
increases the speed of the hybrid stub code by 23% for medium sized IDLs and 34% for large sized IDLs when compared to compiled stubs.

To conclude, in static methods of invocation like compilation with inlining, compilation and interpretation less time is spent in marshalling, because these procedures are generated at the compile time itself. Since each individual type has to be marshaled separately in an interpreter, it takes more time for marshalling than a compiler. For compiling with inlining more RAM space is needed when compared to compilation. For dynamic marshalling the type information has to be got from the interface repository during runtime only, hence it is slow. The hybrid method of invocation uses a combination of static and dynamic methods. Hence, it is faster than the dynamic invocation method, and occupies less space than the static invocation method. As each invocation takes place through a persistent ORB connection, the time required for invocation is further reduced. Thus the proposed method produces a stub code which is optimized in terms of both time and space and is highly suitable for embedded applications.

5.7 Summary

The componentized IDL compiler treats the stub generation problem as a programming language translation problem. It uses intermediate representations, modularized front and back ends, localizing source and target language specifications, interoperability between different distributed object models and a framework organization that encourages reuse of software. The backends of the componentized IDL compiler facilitates the generation of stub code with fast IPC mechanisms. The time taken to marshal and unmarshal the arguments and to pass the data from client to server over the network has been measured for the different datatypes using shared memory and socket optimisation techniques. It is observed that the use of shared memory minimizes the network time because it does not involve marshalling of data into the buffers. But the marshal and unmarshal time is increased due to the time taken to write into or read from the shared memory. On the other hand, the use of sockets minimizes the time taken for marshalling and unmarshalling but increases the network time and the
number of copy operations. Using threads more time is taken in context switching of the threads, but it is useful to pass large amounts of data.

A static bridge to establish communication between COM and CORBA objects using the componentized IDL compiler and COM/CORBA interworking specifications has been designed. The entire process is transparent, as COM clients are not aware that they are communicating with the CORBA servers and vice-versa. Threading has been used to spawn a new process for every client. Also IIOP/UDP has been tried as a new communication medium to improve the networking speed and to bring about interoperability. Interoperability between RMI and CORBA through RMI/IIOP is also achieved using the componentized design of the IDL compiler.

An optimizer has been built in the presentation generation stage of IDL compiler to automatically strike a balance between code size and execution speed of the stub code generated by the compiler. The performance of the hybrid (compiled and interpreted) stub generated using MOGA and SOGA has been analysed. It is observed that the speed of the hybrid stub is comparable to that of compiled stub, but it occupies less space. This difference in performance is more prominent in the case of an interface with composite data types.

A hybrid code is also generated to strike a balance between SII mechanisms, namely inlined, compiled, interpreted and DII mechanisms. It is observed that the speed of the hybrid marshalling is comparable to that of compiled stub, but it also occupies less space. It is also found that it is much faster than dynamic invocation mechanism. For an example interface, it has been found that the size of the hybrid stub code is almost the same as that of dynamic stub and its speed is also increased by 45%. The performance of the hybridized stub code has been tested on ZEN, a RT-CORBA ORB. A maximum of 54% savings in heap space and 11% savings in hard disk space when compared to compilation and 56% in time savings when compared to dynamic invocation has been achieved in Zen ORB. Using persistent ORB connections, the speed of the hybrid stub code generated was further
increased by 36% when compared to the conventional static invocation methods used in ZEN.