Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

DIMMA - Performance

Ian Macmillan

Abstract

This document describes the results of performance measurements made with DIMMA, starting from release 2.0. The tests are simple ones intended to show the result of using explicit binding with QoS specification and to put these in the context of some other ORBs.

The main conclusions are that DIMMA's resource control works: in one case dramatically halving the time for an invocation. Furthermore, DIMMA's performance stands comparison with other commercial ORBs even though it has not been fully optimised.

APM.2042.00.03 **Draft** 5th August 1997

Technical Report

Distribution:
Supersedes:
Superseded by:

DIMMA - Performance

DIMMA - Performance

Ian Macmillan

APM.2042.00.03

5th August 1997

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by APM Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

APM Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 DIMMA Performance

1.1 Overview

This document describes the results of performance measurements made with DIMMA, starting from implementation release 2.0. The tests are simple ones intended to:

- compare the performance of DIMMA against that of other ORBs
- show the result of using QoS specification to vary the configuration of communication channels

Initial performance measurements made on the 2.0 base showed relatively poor performance compared with other commercial ORBs. Not surprising perhaps given that the implementation was in no way optimised for performance. A brief investigation was subsequently carried out to identify the offending areas and is described in more detail in section 1.4.

Using the results obtained from the above investigation, a modified version of DIMMA was produced (release 2.0 performance fix 1). Further studies and subsequent modifications resulted in release 2.0 performance fix 2. The results given in this document were made using the latter.

1.2 Environment

The measurements were all carried out on a dedicated Sun SPARC Ultra 1 (167MHz) with 128Mb memory and running Solaris 2.5.1. DIMMA was compiled using the `-fast` option of the SPARCworks C++ compiler version 4.1. DIMMA was built using dynamically linked shared objects (as opposed to static archive libraries).

1.3 The tests

The tests were divided into two groups:

- Simple string tests
- Tests involving more complex arguments

1.3.1 String tests

Tests were performed using a simple example to time the elapsed time at the client of:

- A null invocation
- Echo of a null string
- Echo of an arbitrary length string (54 characters)

The tests were performed using oneway operations to one interface, and normal RPC on another interface. Both interfaces were provided by the same

server. The source code for this test may be found in the DIMMA source structure, directory `$DIMMA/examples/Time`. The IDL of the interfaces is:

```
interface TestOneway
{
     oneway void test_no_param();
     oneway void test_string(in string stringVal);
};

interface TestReqReply
{
     void test_no_param();
     void test_string(inout string stringVal);
};
```

Results were recorded using both default QoS and a specific QoS designed to produce an optimised channel with the following characteristics:

- dedicated network socket
- dedicated thread to perform:
  - synchronous RPC at the client
  - all network and operation processing at the server
- dedicated buffer pools (one pool at server, one at client) each containing just one buffer

This corresponds to an `EngineeringQoS` specification of `ProcessingConcurrency` = `nullTask` and `MessageConcurrency` of one.

1.3.2 Complex argument tests

This test was based on one originating from Post Modern Computing. The source code for this test is located in the DIMMA source structures under `$DIMMA/examples/Timing`. The IDL is as follows:

```
// IDLs to do performance testing of oneway and req/reply
// interfaces
// These tests also include arguments of various types

typedef sequence<short> shortSeq;
typedef sequence<long> longSeq;
typedef sequence<float> floatSeq;
typedef sequence<double> doubleSeq;
typedef sequence<string> stringSeq;
typedef sequence<char> charSeq;

struct PerfStruct {
     short shortVal;
     long longVal;
     float floatVal;
     double doubleVal;
     char charVal;
     string stringVal;
};

typedef sequence<PerfStruct> structSeq;
```

```
interface TestOneway
{
     oneway void test_no_param();

     oneway void test_prim_args(in short shortVal,
          in long longVal,
          in float floatVal, in double doubleVal,
          in char charVal, in string stringVal);

     oneway void test_struct(in PerfStruct structVal);

     oneway void test_prim_seq(in shortSeq shortVal,
          in longSeq longVal,
          in floatSeq floatVal, in doubleSeq doubleVal,
          in charSeq charVal, in stringSeq stringVal);

     oneway void test_struct_seq(in structSeq structVal);
};

interface TestReqReply
{
     long test_prim_args(in short shortVal, in long longVal,
          in float floatVal, in double doubleVal,
          in char charVal, in string stringVal,
          inout short inoutShort, inout long inoutLong,
          inout float inoutFloat, inout double inoutDouble,
          inout char inoutChar, inout string inoutString,
          out short outShort, out long outLong,
          out float outFloat, out double outDouble,
          out char outChar, out string outString);

     long test_struct_args(in PerfStruct structVal,
          inout PerfStruct inoutStruct,
          out PerfStruct outStruct);

     long test_prim_seq(in shortSeq shortVal, in longSeq longVal,
          in floatSeq floatVal, in doubleSeq doubleVal,
          in charSeq charVal, in stringSeq stringVal,
          inout shortSeq inoutShort, inout longSeq inoutLong,
          inout floatSeq inoutFloat, inout doubleSeq inoutDouble,
          inout charSeq inoutChar, inout stringSeq inoutString,
          out shortSeq outShort, out longSeq outLong,
          out floatSeq outFloat, out doubleSeq outDouble,
          out charSeq outChar, out stringSeq outString);

     long test_struct_seq(in structSeq structVal,
          inout structSeq inoutStruct,
          out structSeq outStruct);
};
```

The test was run using default implicit binding. Runs were performed using the IIOP protocol and the ANSA flow protocol.

1.4 Initial Investigation

Initial performance measurements made on the DIMMA 2.0 base showed relatively poor performance compared with other commercial ORBs, a null RPC taking around 2300 microseconds. A short study was carried out using

Sun's `Collector` and `Analyzer` to profile the code. As these tools do not support multi-threaded applications, the measurements had to be performed using a single-threaded build of DIMMA.

The results indicated that marshalling was consuming an excessive amount of time. Specifically:

- too many calls on lowest common denominator routines such as writing a byte
- dynamic marshalling policy was causing excessive run-time overhead
- alignment method was in the top five contributors to processor usage.

Other contributors were:

- IIOP protocol's use of marshalling for message headers

Further tests using the multi-threaded build of DIMMA were attempted using the standard Unix tools `prof` and `gprof`. However, these appeared to give misleading results. Sun's `Thread Analyzer` (part of the SPARCworks/iMPact toolset) was also tried but this could not be made to work at all!

Finally an evaluation version of Pure Atria's `Quantify` tool was downloaded and fortunately proved more useful. The main culprits contributing to the inferior multi-threaded performance appear to be locks internal to the Solaris libraries. This significantly impacts the time required for operations such as `new` and `delete`. Also, a semaphore seems to be taken twice for each call to the socket abstractions `recv` and `write`, again significantly contributing to the overall time.

1.5 Further Investigation

A further study was made using `Quantify` on the 2.0_P1 DIMMA code base to investigate what could be done about the high cost of memory management and the socket I/O calls `recv` anb `write`. As a result, the following changes were made:

- Perform read-ahead when using `recv` to reduce the number of calls
- Reduce the number of dynamic memory allocation calls by:
  - adding further resource pools
  - embedding structures rather than using pointers
- Further optimising of [un]marshalling, in particular:
  - providing specific marshallers for sequences of Octet and Char
  - specific unmarshalling of IIOP object keys

1.5.1 Read-ahead caching

The use of read-ahead merits some further words as it results in a significant benefit.

IIOP messages are variable length and consist of a header, which contains the length of the entire message, and the message body which follows. IIOP performs a read of the header from the underlying TCP/IP stream and then requests the message body, having first determined its length from the header.

This results in considerable inefficiency as it requires two reads for every message and the socket `recv` call is particularly expensive in a multi-threaded environment.

The read-ahead policy addresses this by performing a fixed length read that is greater than the size of an IIOP header, on the first request from the IIOP protocol. This will often result in the entire message being read in one call to `recv`[1]. The price that must be paid for this optimisation, is that the data must subsequently be copied into the buffer provided by IIOP on the second read (the one for the message body).

However, by chosing a suitable fixed length, the drawbacks of the additional `memcpy` is more than compensated for by the reduction in `recv` calls.

1. This relies on the TCP/IP implementation returning any data that it has already received, even if this is less than that requested.

1.6 Results

The results here are based on a version of DIMMA known as 2.0 performance fix 2, or 2.0_P2 for short. This version incorporates modifications formost of the problems identified in the preceding studies.

1.6.1 String tests

The following tables document the results and are the average of 5000 invocations. Each entry represents the elapsed time in microseconds for the call to complete, as measured from the client side.

The first line of each table *No QoS* represents the time taken using explicit binding but default QoS. Explicit binding in this case only serves to remove the binding time from the measurement[1].

The second line *Client QoS* denotes a dedicated channel and thread at the client end only whilst the server uses default QoS.

The third row *Server QoS* uses a dedicated channel and thread at the server end only, the client employing default QoS.

The final row shows the results for a fully QoS configured communications channel.

Table 1.1 gives the result for DIMMA built with multi-threaded support.

Table 1.1: Multi-threaded build results

	Oneway null param	Oneway null string	Oneway test string	RPC null param	RPC null string	RPC test string
No QoS	868	1185	1291	1880	1846	1782
Client QoS	1000	980	1258	1578	1372	1377
Server QoS	280	280	318	1117	1095	1112
Full QoS	231	243	311	667	691	715

Table 1.2 illustrates the results for a single-threaded DIMMA build.

Table 1.2: Single-threaded build results

	Oneway null param	Oneway null string	Oneway test string	RPC null param	RPC null string	RPC test string
No QoS	276	269	310	745	745	761
Client QoS	213	251	280	636	637	645
Server QoS	273	266	314	640	634	647
Full QoS	198	265	263	545	563	564

Table 1.3 is provided to act as a comparison with some other ORBs. The figures for the other ORBs are taken from those produced by The Olivetti & Oracle Research Laboratory and published on their web page. Their

1. Note however that the one off binding time would likely be lost in the noise anyway since the results are the average of 5000 invocations.

measurements use a similar test scenario to the DIMMA *null string test* and the same target environment.

Table 1.3: Comparison of ORBs

ORB	Time per call (default QoS)	Time per call (explicit QoS)
omniORB2	540	n/a
Orbeline 2.0 - Release 1.51	920	n/a
HP ORB Plus 2.5	1240	n/a
Orbix 2.1	1700	n/a
DIMMA 2.0_P2	1846	691

1.6.2 Complex argument tests

Table 1.4 shows the results of running the *complex argument tests* on DIMMA, using both IIOP and ANSA flow protocols, with single and multi-threaded builds. Each table entry has two figures: the first is the elapsed time per call in

Table 1.4: Complex argument tests

	MT build IIOP	Single-threaded IIOP	MT build ANSA Flow	Single-threaded ANSA Flow
Oneway no params 0 bytes	1013 n/a	248 n/a	137 n/a	60 n/a
Oneway prim_args 125000 bytes	1172 21324	315 79300	102 244044	69 362808
Oneway struct 125000 bytes	1232 20289	288 86669	96 259370	68 368891
Oneway prim_seq 1.25e7 bytes	2815 888057	1581 1581995	519 4818524	471 5309981
Oneway struct_seq 1.25e7 bytes	3065 815632	1767 1414769	541 4620969	488 5124790
RPC prim_args 50000 bytes	2026 49345	850 117642	n/a	n/a
RPC struct_args 50000 bytes	2033 49200	847 118043	n/a	n/a
RPC prim_seq 5e7 bytes	7681 5e7 1301852	6142 1628191	n/a	n/a
RPC struct_seq 5e7 bytes	9264 5e7 1079433	7341 1362266	n/a	n/a

microseconds; the second is the throughput achieved in bytes per second.

The throughput figures for the ANSA flow protocol need to be regarded with some caution. They are based on the amount of data sent and how long it takes the transmitter to complete the transmission. However, ANSA flow is built on top of UDP and uses no acknowledgement mechanism to guarantee delivery. Therefore, it is quite likely that some of the data transmitted will be lost and never make it to the receiver.

1.7 Validity of the results

The absolute figures in the tables must be regarded with a degree of suspicion. For example, from table 1.3, the default QoS result is given as 1825 microseconds. However, figures as low as 1600 occur relatively often, even on a supposedly loaded machine. Likewise, the value of 933 microseconds for the specific QoS case has also been recorded as 840 microseconds under similar conditions.

Fortunately the trends are sufficiently reliable to draw some meaningful conclusions.

1.8 Discussion

Perhaps the most obvious result is that varying the resource configuration affects the single-threaded build far less than the multi-threaded one. The reason for this is due more to the nature of the test than anything intrinsic to either DIMMA or threading generally.

The tests are themselves single-threaded and hence there is no real contention for resources. Hence, whether a resource is allocated from a pool, or manufactured by a factory, makes little difference other than a small variation in processing overhead.

Another observation is that the single-threaded build always performs better than the multi-threaded one, irrespective of resource configuration. This is because the single-threaded build has inherent advantages over the multi-threaded one:

- there is no need to take mutual exclusion locks
- there is no thread context switching overhead

1.8.1 Use of locks

The first of these is the main reason why the multi-threaded build will always be slower, since removal of the locks is not possible by reconfiguring the resources[1]. However, it is not just the locks used directly by DIMMA that are the problem. It turns out that various Solaris libraries also take locks to ensure that their methods are reentrant.

This is more of an overhead than might be expected: some of the methods are very low level and are consequently called many times. For example, `new` and `delete`. Furthermore, these locks are not always needed within DIMMA

1. This is certainly true in the existing implementation of DIMMA. It is also difficult to see how a DPE could in general be made aware that locking is not required for a particular scenario and even if it could, how this could be used to advantage, i.e. without introducing further processing overhead.

which protects itself using its own, higher level, locks. This requires further investigation.

1.8.2 Thread context switching

Examination of the results from the multi-threaded build (table 1.1), shows a range of results depending upon the specified QoS. The difference between the default case and the full explicit QoS case is dramatic: the times for the latter are almost halved. This is almost entirely down to thread context switching, or to be more precise, the avoidance of context switching.

The default resource configuration for a communications channel in DIMMA, is to use the calling thread for message transmission, i.e right down to the operating system network interface, and a separate thread for message demultiplexing. The reason for this is one of scalability: if a thread and socket were to be allocated for every communication channel in a busy system, the resources would rapidly be exhausted.

However, for cases where a high throughput is required and there is no requirement for a large number of such channels, it is better to use a dedicated thread resource for each channel. This is demonstrated by the *Full QoS* row of table 1.1.

1.8.3 Further work

Further work is required to test the effectiveness of the DIMMA resourcing model. Tests are needed for the following scenarios:

- multi-threaded application competing for resources
- latency and jitter measurements

1.9 Conclusion

The main conclusions and observations that can be drawn from these limited tests are:

- DIMMA's resource control works - varying resource configuration via QoS specification has a significant (beneficial) impact on the results.
- DIMMA's performance stands comparison with other ORBs even though it has not been fully optimised
- The multi-threaded build suffers an overhead (18%), apparently due to the locking overhead of the Solaris multi-threaded libraries.
- More tests are required for different scenarios.

References

[APM.1994]

DIMMA Team, *DIMMA Design and Implementation*; **APM.1994**, APM Ltd., Cambridge U.K., May 1997.