Open CORBA Benchmarking

Petr Tůma, Adam Buble
Charles University, Faculty of Mathematics and Physics
Department of Software Engineering
Malostranské nám. 25, Prague, Czech Republic
phone: +420-2-21914267, fax: +420-2-21914323
e-mails: petr.tuma@mff.cuni.cz, adam.buble@mff.cuni.cz

Abstract: We present two benchmark suites for CORBA brokers, targeted at the broker vendor and user audiences respectively. The vendor suite is a result of several benchmarking projects with industrial partners, and covers the entire functionality of a CORBA broker. The user suite is simplified to give an overview of the basic factors influencing broker performance, and is complemented with an approach for tailoring the results to a specific system and mode of operation without a prohibitive loss of precision.
Keywords: CORBA, middleware, benchmarking.

1 INTRODUCTION

In 1991, CORBA [15] emerged as a middleware architecture for object communication in heterogeneous distributed environments. As the architecture developed and established itself as an industrial standard, a need for assessing the performance of its implementations became apparent [1]. In response to this need, a number of benchmarking projects appeared [2][3][4][13]. These differed in many aspects, including the benchmarked functions, the environment setup, and the results presentation. This made the interpretation and relating of different benchmarks difficult. The problem was identified by OMG in the 1998 Benchmarking PSIG Request For Information [1], and received a number of responses [5][6][7][8], summarized in the OMG White Paper On Benchmarking [9].

The White Paper styles itself as "a first step towards a uniform benchmarking methodology to be applied for various OMG technologies", and focuses mostly at outlining the issues related to benchmarking and planning the future work. This work is to take shape of a benchmarking framework and methodology, a repository for public benchmark algorithms, and a tool for automated benchmark execution. Although none of the plans materialized so far, the White Paper still voices a conviction of its contributors that there is a need for benchmarks that would be open, well understood, and easy to measure.

In typical benchmarking projects, these attributes are often considered of secondary importance. This is because the projects are done in response to particular needs of the involved parties, and those needs rarely include openness. Thus, the projects might stop short of publishing complete results [10][11][12][13], complete methodology [4], or complete sources [14]. Even if published, the project results are generally difficult to interpret and to relate to each other.

We attempt to remedy this gap by generalizing our experience in the area of benchmarking CORBA middleware, gathered in projects with MLC Systeme (now Deutsche Post Com) [10][11][12][13], Bull Soft (now Evidian) [10], IONA Technologies, and others. In chapter 2, we present an open benchmark suite for evaluating performance of CORBA middleware. In chapter 3, we follow with the guidelines on executing the suite and reporting the results. In chapter 4, we discuss the portability of the benchmark results, and demonstrate its use in both publishing and using the results.

2 OPEN BENCHMARK SUITE

The obvious goal of an open benchmark suite is to cover all commonly used broker functionality. This goal, however, has the potential of yielding too large a suite, which goes against the requirements of well understood and easy to measure benchmarks expressed in [9]. To tackle this problem, we consider two different target audiences for the benchmark results, broker vendors and broker users:

Broker vendors represent an audience that uses the benchmarks mainly to assess the performance of their particular broker, e.g. to identify a performance related bug or a bottleneck, or to evaluate a performance impact of a modification done during broker development. To satisfy these requirements, the benchmark suite needs to cover every possible aspect of broker performance in detail, including aspects specific to a particular broker implementation. The benchmark results can be very technical, because the people reading it are likely to be familiar with the technical details of the broker implementation.

Broker users represent an audience that uses the benchmarks mainly to understand what performance can be expected from a broker in their particular application, e.g. to decide what broker implementation to use, or to architect an application based on the available broker performance. To satisfy these requirements, the benchmark suite needs to be easily understandable, because the people reading it will not necessarily be experts in broker architecture and performance. The benchmark results can be simplified through generalization, although it is important to clearly note the cases where this might not work.

Although the two audiences can occasionally benefit from the benchmark results targeted at the other audience, the principal difference is significant enough to warrant separate benchmark suites.

In the following text, we describe the individual aspects of broker performance. For each aspect, we first describe what the vendor suite measures, and next identify acceptable simplifications to obtain the user suite. The simplifications are done to make the suite easy to understand, code and execute, but at the same time to keep it informative enough. We do this by analyzing the data collected in our benchmarking projects [11][12][13], and selecting those simplifications that do not loose important information. For sake of brevity, the descriptions are kept to a minimum necessary so that a person familiar with CORBA can understand what the benchmarks measure, and illustrated with selected results.

2.1 Invocation Benchmarks

For vendors: Basic invocation benchmarks measure the distribution of delivery and roundtrip times for a simple method invocation with no arguments in all available invocation modes of the broker. The results are given as a graph depicting the statistical distribution of the delivery and roundtrip times (Figure 1), and a graph depicting regular interference patterns in the results (Figure 2).

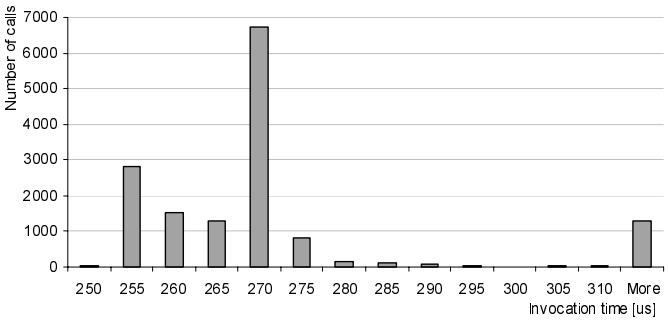

Figure 1 Distribution, roundtrip time, dynamic invocation, ORBacus 4.0.4, Linux 2.2.16, Dual Intel Pentium III 800 MHz, 512 MB RAM

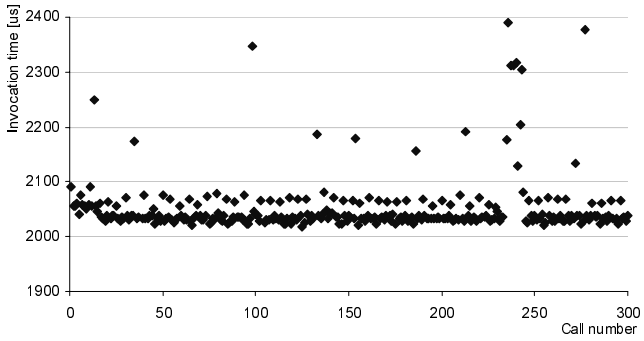

Figure 2 Regular fluctuations, roundtrip time, static invocation, ORBacus 4.0.1, Linux 2.2.16, Intel Pentium 166 MHz, 64 MB RAM

For users: For a specific broker implementation, the delivery and roundtrip times are typically very close to each other both in absolute values and in statistical distribution regardless of the invocation mode used. An acceptable simplification is to present the results of a benchmark that uses a single invocation model only.

For brokers that support Quality of Service settings, additional benchmarks have to be developed to test the individual policies. Because many current brokers do not support Quality of Service settings yet, we refer the reader to specific Quality of Service research [16] and do not attempt to simplify the benchmarks for this particular case.

2.2 Marshalling Benchmarks

For vendors: Marshalling benchmarks measure the dependence of the delivery and roundtrip times for a simple method invocation on the type, encapsulation, size and direction of the arguments. To reduce the number of combinations presented, the benchmarks first measure method invocations that pass one instance of all basic data types in all directions, which makes for some 85 methods. Next, a smaller set of representative basic types is selected, and the benchmarks measure invocations that pass instances of these inside anys, arrays, sequences, structures, unions and value types, which makes for 30 methods per type. The results are given as a graph or a table depicting the delivery and roundtrip times per type, encapsulation and direction (Figure 3).

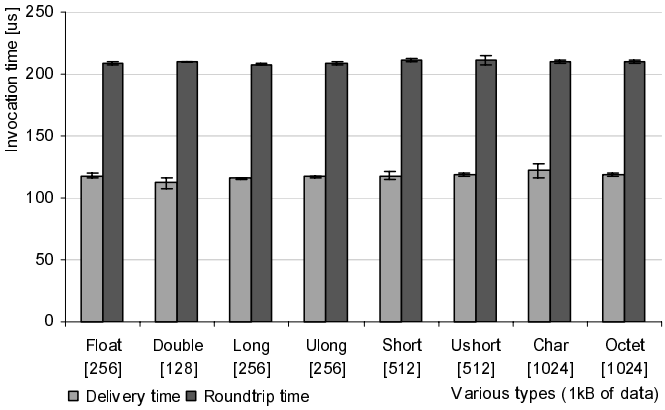

Figure 3 Delivery and roundtrip times for array types, direction in, static invocation, ORBacus 4.0.4, Linux 2.2.16, Dual Intel Pentium III 800 MHz, 512 MB RAM

To measure the dependence of the delivery and roundtrip times on the size of the data, again, a smaller set of representative basic types is selected, and the benchmarks measure invocations that pass a growing number of instances of these inside arrays and sequences. The results are given as a graph depicting the dependency of the delivery and roundtrip times on the size of the data (Figure 4).

Separate marshalling benchmarks also measure the dependency of the proxy creation times on the number of existing proxies and the number of supported proxy classes.

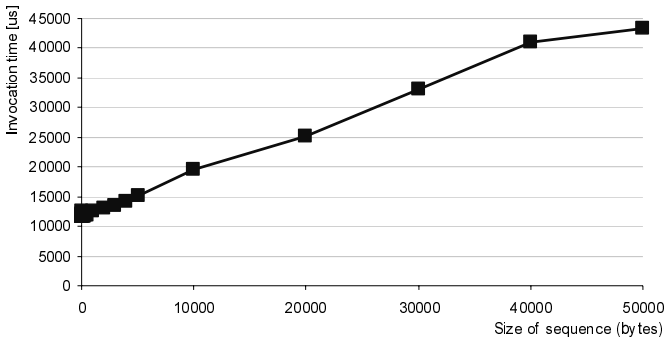

Figure 4 Roundtrip time for growing sequence<octet>, direction in, static invocation, ORBacus 4.0.1, Linux 2.2.13, AMD 486 100 MHz, 32 MB RAM

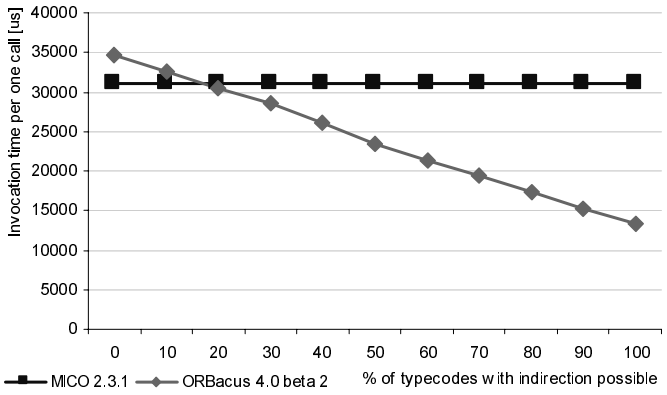

Figure 5 Typecode indirection, roundtrip time, any[100], direction in, static invocation, Windows NT 4.0 SP6, Intel Celeron 466 MHz, 128MB RAM

For users: For a specific broker implementation, the delivery and roundtrip times typically do not vary significantly with the type, encapsulation and direction of the data. An acceptable simplification is to omit the results of benchmarks that measure the impact of these factors. A notable exception is passing character data in an environment that does code page conversion, introducing an extra overhead.

The delivery and roundtrip times typically depend linearly on the size of the data. An acceptable simplification is to present the results of a benchmark that uses a single type, encapsulation and direction of the data only. Two notable exceptions are passing a large number of instances of anys and value types, where the delivery and roundtrip times depend on the number of instances and the behavior of the typecode indirection and reference graph traversal mechanisms (Figure 5).

2.3 Dispatcher Benchmarks

For vendors: Dispatcher benchmarks measure the dependence of the invocation times for a simple method invocation on the complexity of the interface the method belongs to, the complexity of the object adapter hierarchy the instance belongs to, and the servant registration policy and the number of servants registered by the object adapter.

Of the complexity of the interface, the influencing factors are the number of methods in the interface and the length of the method name. Of the complexity of the object adapter hierarchy, the influencing factors are its width and depth. These lead to straightforward benchmarks, whose results are given as a graph depicting the dependence of the invocation times on these factors.

Of the servant registration policies, the most interesting one from the performance point of view is the RETAIN policy, which searches a map of active servants during each invocation. This leads to a straightforward benchmark, whose results are given as a graph depicting the dependence of the invocation times on the number of servants registered in the RETAIN map (Figure 6).

Many broker implementations use algorithms that can exhibit the best or worst case behavior that differs significantly from the typical case to perform some of the operations described in this section. The benchmarks should take care to measure and report both the ordinary and the extraordinary behavior, possibly using the box and whisker graphs rather than usual line graphs.

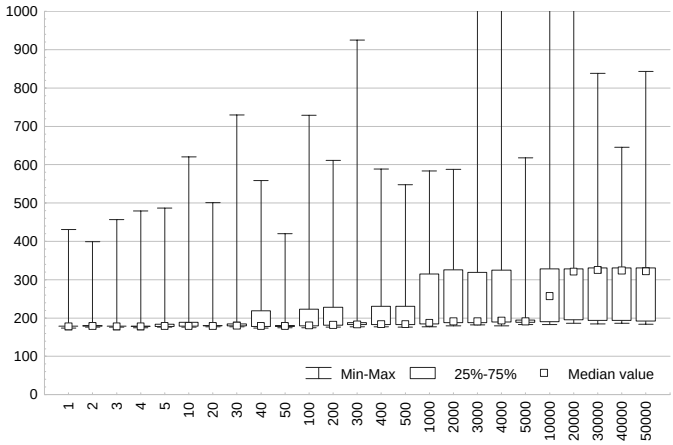

Figure 6 Roundtrip times, void ping() for increasing number of servants, static invocation, ORBacus 4.0.4, Linux 2.2.16, Dual Pentium III 800 MHz, 512 MB RAM

For users: The influence of the interface and object adapter hierarchy complexity is of a smaller importance than the influence of the number of registered servants, because it tends to exhibit itself for uncommonly complex interfaces or object adapter hierarchies only. An acceptable simplification is to present the results of a benchmark that measures the influence of the number of registered servants only.

2.4 Parallelism Benchmarks

For vendors: Parallelism benchmarks measure the dependence of the invocation times for a simple method invo-

cation on the number of threads and number of clients that issue or handle the invocations in parallel. Besides the invocation times as perceived by the individual threads, the benchmarks also measure the invocation rate as perceived by the system. The invocation times perceived by a single thread grow whenever the thread is blocked waiting for another thread, and thus describe the degree of parallelism achieved during invocation (Figure 7). The invocation rate perceived by the system increases as the ability of the system for parallel execution gets increasingly exploited, up to the point where it is fully exploited, from which the invocation rate slowly decreases as the overhead associated with the parallel execution increases. The results are given as a graph depicting the dependence of the invocation times and the invocation rate on the number of threads and number of clients that issue or handle the invocations in parallel.

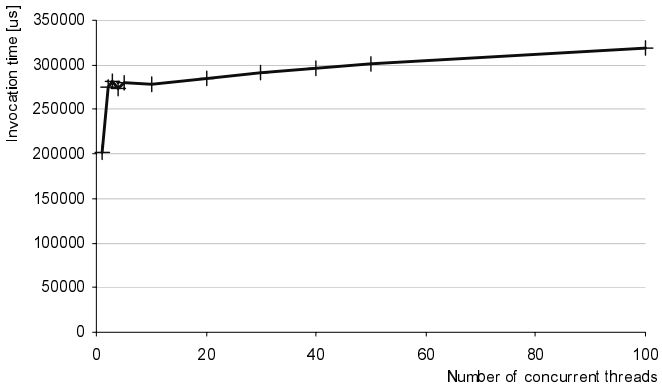

Figure 7 Roundtrip time, void ping() for increasing number of threads, static invocation, ORBacus 4.0.4, Linux 2.2.16, Dual Intel Pentium III 800 MHz, 512 MB RAM

Depending on the specific broker implementation, the parallelism benchmarks need to be run several times or extended to cover all threading strategies. Also, the distribution of the system capacity among the individual threads needs to be evaluated [17].

For users: The simplification stems from the fact that a benchmark where a large number of clients issue invocations is difficult to execute. In a local setup, the load generated by the clients interferes with the server. In a network setup, the network bottlenecks can do the same. Also, the cost of maintaining a benchmarking environment capable of executing a large number of clients in parallel is not trivial. To keep the benchmark suite easy to execute, we only include the benchmarks that use multiple threads rather than multiple clients.

2.5 Miscellaneous Benchmarks

A broker also provides functions that are not directly related to invocation. Typical examples of these are insertion and extraction of data into and from anys, string allocation, or dynamic any inspection. Straightforward benchmarks of these functions can be incorporated into the vendor suite, but are not significant enough to be included in the user suite.

The vendor suite outlined in this section provides a detailed overview of most factors influencing the broker performance. In its evolving form, it was successfully used in several benchmarking projects [12][13], and formed a basis for an EJB benchmark suite [10][11].

The user suite outlined in this section provides a rough overview of the basic factors influencing the broker performance. When implemented for C++ brokers, the suite proved easily portable, typically requiring a change of under 10 lines of code per broker. The implementation is available at http://nenya.ms.mff.cuni.cz/~bench.

3 EXECUTING AND REPORTING

The paramount criteria for executing the benchmarks and reporting the results are relevancy and repeatability. A benchmark needs to be relevant in that the measurements must reflect typical modes of operation [9], and repeatable in that the target audience must be able to reproduce the results [18].

The results of the benchmarks described in section 2 are not immediately relevant in the sense considered here, because the mode of operation used corresponds to none but most trivial problem domains. The chief differences are testing individual performance factors separately, testing with no background load, and testing with ideal network conditions.

3.1 Combining Performance Factors

A typical approach to providing relevant results is adjusting the benchmark so that it executes a mixture of operations similar to that executed by existing applications [19]. This is difficult to do for a broker, because the typical modes of operation, and therefore also the importance of the performance factors, vary significantly [9]. Our approach therefore is to characterize a specific mode of operation in terms of the individual performance factors, and then combine the results of the individual benchmarks based on this characterization.

The performance factors considered by the user suite are the size of the invocation arguments, the number of the registered servants, and the number of the active threads. The size of the arguments adds an overhead that represents time spent marshalling and transporting the arguments. The number of servants adds an overhead that represents time spent marshalling and transporting longer object references and looking up in larger tables. These overheads are additive in the sense that adding extra arguments or servants to an invocation means adding the overhead to the invocation time. The number of threads adds a scaling factor that represents the degree of parallelism of the invocations.

Knowing the typical size of invocation arguments *S*, number of registered servants *R*, and number of active threads *T* for a specific mode of operation, the results of the individual benchmarks can therefore be combined to form an estimated relevant result using a simple formula:

estimated time = (simple + size(S) – size(0) + servants(R) – servants (0)) × threads (T) / threads (1)

Here, *simple* is the simple invocation time (section 2.1), *size(x)* is the invocation time for arguments of size *x* (section 2.2), *servants(x)* is the invocation time for *x* servants (section 2.3), and *threads(x)* is the invocation time for *x* threads (section 2.4).

The estimate is based on the assumption that the factors are largely orthogonal. Our measurements indicate this condition holds until the factors start influencing each other through exhaustion of shared resources. The precision of the estimated relevant result as opposed to the measured one is illustrated in Figure 8 and Figure 9.

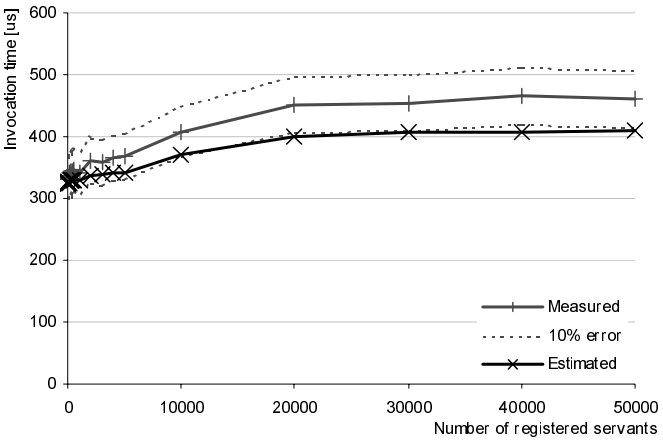

Figure 8 Roundtrip times, void ping() for increasing number of servants, static invocation, ORBacus 4.0.4, Linux 2.2.16, Dual Pentium III 800 MHz, 512 MB RAM

3.2 Incorporating Background Load

Besides the mode of operation, the observed performance of a broker also varies with the background load. It has been suggested [8] that the background load needs to be incorporated into the benchmark to make it relevant. Measurements done with a regular processor-intensive and network-intensive load created by background processes show that the background load exhibits itself as a predictable increase in the invocation times that is inversely proportional to the percentage of the processing power scheduled to the benchmarks (Figure 10).

The effects of an irregular background load cannot be incorporated into the benchmark results as easily. The simplest way to obtain precise results in irregular background load conditions is running the benchmark suite under them.

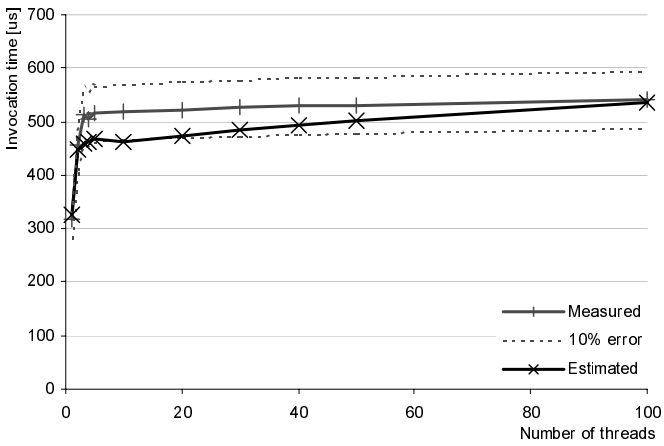

Figure 9 Roundtrip times, void ping() for increasing number of threads, static invocation, ORBacus 4.0.4, Linux 2.2.16, Dual Pentium III 800 MHz, 512 MB RAM

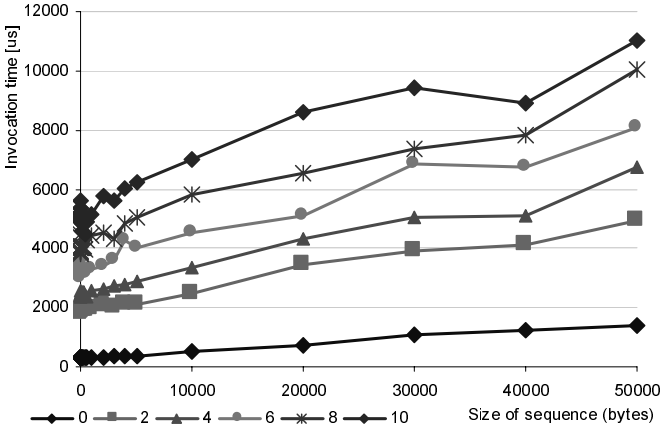

Figure 10 Roundtrip time for growing sequence<octet>, direction out, static invocation, ORBacus 4.0.3, Linux 2.2.18, Dual Pentium III 800 MHz, 512 MB RAM, lines for different numbers of background processes

3.3 Incorporating Network Conditions

Under congested network conditions, the influence of the particular broker implementation on the benchmark results tends to be quickly overshadowed by the influence of the network delays. Because a majority of brokers communicates through IIOP on top of TCP/IP implemented by the operating system protocol stack, it is best to measure their behavior under congested network conditions using the existing methodology for TCP/IP benchmarking.

3.4 Reporting Results

The report with the results of the benchmarking suite should follow the proven full disclosure approach [20][21] by specifying configuration information that is sufficient to reproduce the results. Besides the results themselves, the report should contain a description of the benchmark, a de-

scription of the form in which the results are reported, and an explanation of what the results signify and when simplifications might not apply, for each benchmark. An example of this description for a single benchmark is on Figure 11.

Typecode Indirection

Desc: The test measures the typecode indirection mechanism that prevents duplication of typecodes in GIOP messages. The called interface is:

```
typedef any ArrayAny [100];
interface TypeCodeIndirection {
   void InArrayAny (in ArrayAny Arr);
};
```

The array is filled with instances of types whose typecodes are 100 characters long. The dependency of the invocation time on the number of identical types within the array is measured.

Graph: The graph depicts the dependency of the invocation time on the number of identical typecodes passed. The X axis gives the number of instances passed within the array that have the same type; the Y axis gives the invocation time.

Result: When no dependency is observed, the ORB does not use the typecode indirection mechanism. A gradually decreasing dependency should be exhibited otherwise.

Figure 11 A report for a typecode benchmark

4 PORTABILITY OF RESULTS

Until this point, only the usual approach to benchmarking, where the benchmark results are tied to a specific hardware platform, operating system, compiler and libraries, has been considered. Although answering the basic question of *"How fast does the broker run on this system?"*, this approach fails to answer many other important questions that the target audience might ask, such as *"How fast a system do I need to get a specific performance from a specific broker?"*, *"How fast will my application with the broker run on the systems that my customers have?"*, or *"How do I compare benchmark results for different brokers running on different systems?"*.

To answer these questions, the benchmark results for a particular broker need to be abstracted from the influence of the system, on which the benchmark is executed. A general approach to the abstraction process, basically consisting of modeling the execution of the benchmarked system as a mixture of primitive operations, whose execution times combined yield the execution times of the benchmarked system, was suggested elsewhere, mostly in the operating systems research domain [22][23][24]. Two basic questions that need to be answered when using this approach are, what should be the primitive operations used, and what is the usage of a particular primitive operation by the benchmarked system. Typically, these were answered through analysis of the system calls or machine instructions used by the benchmarked system, and by profiling. Also, the precision reported was often unimpressive, for example, 30 % error in [23], 40 % error in [24].

4.1 Selecting Primitive Operations

When trying to apply the abstraction process to the broker benchmark, we are further confronted with the fact that both the broker and the system can be different for each benchmark execution. Requiring the selection and profiling of the primitive operations to be done for each combination of broker and system would break the principle of designing a benchmark that is easy to measure.

To work around this problem, we chose to select the primitive operations from those that are likely to be executed by a broker regardless of the particular implementation. The primitive operations are selected based on an analysis and experimental profiling of several broker implementations, and cover between 80 % and 90 % of the execution time:

- memory allocation for selected block sizes and counts
- memory block move for selected sizes of blocks
- thread lifecycle operations for selected thread numbers
- thread synchronization with and without blocking
- socket data transport for selected sizes of blocks

The remaining execution time is spent in various general purpose operations. Simple benchmarks are used to obtain the execution times of the primitive operations, an integer arithmetic benchmark is used to provide an approximation for the general purpose operations.

4.2 Weighting Primitive Operations

Using profiling to determine the weight of each of the primitive operations in the combined result leads to a number of technical problems. These include the inability of many profiling tools to correctly and completely report all system and library calls, to separate profiles depending on the call arguments, and to profile code with no symbol information.

Instead of profiling, we therefore use a statistical analysis of a number of collected measurements. The approach first uses the principal component analysis [25] to determine the number of independent factors contributing to the collected measurements, and then uses the incremental regression to select this many primitive operations that most contribute to the combined result and calculate their weights. Two important reasons exist for limiting the number of primitive operations to the number of independent factors determined to exist in the collected measurements. First, the larger the number of primitive operations we want to use, the larger the number of measurements needs to be collected before the statistical analysis can be done. Second, an attempt to increase the number of primitive operations above the number of independent factors causes the incremental

regression to try minimizing the error function beyond the precision the collected measurements warrant in reality. This yields false weights that appear to improve precision over the collected measurements, but because the weights cater to random fluctuations rather than reflecting reality, applying them to new measurements will fail.

Once the set of primitive operations is selected and the vector of their weights *Weights* calculated, the results of the broker benchmarks on a system for which the vector of the primitive operation timings *Ops* is known can be estimated by a simple linear combination as $Ops \times Weights$.

The straightforward application of the approach is calculating the weights for a specific broker implementation and benchmark, and then predicting the benchmark result on other systems for which the primitive operation timings are known. This can be extended to compare performance of broker implementations. The approach can also be used to select a suitable system by its primitive operation timings when the performance requirements are known.

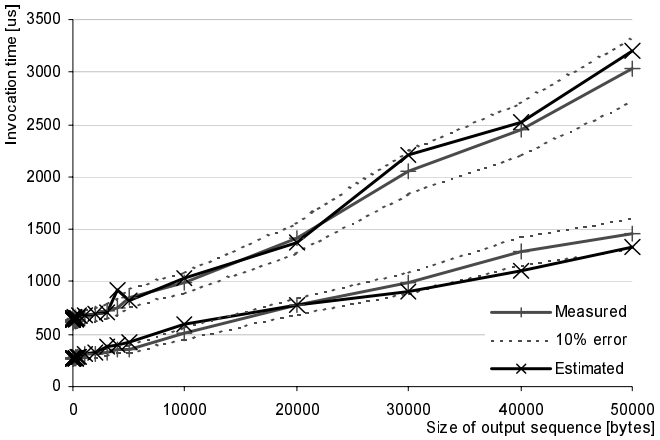

Figure 12 Roundtrip time for growing sequence<octet>, direction out, static invocation, ORBacus 4.0.1, Linux 2.2.16, AMD K6 450 MHz, 128 MB RAM, Linux 2.2.18, Dual Pentium III 800 MHz, 512 MB RAM

This approach was tested in an experiment running during the fall of 2000. A public suite of benchmarks was posted on the http://nenya.ms.mff.cuni.cz/~bench web site for several C++ brokers, together with an interface through which the participants could submit the results of the benchmarks when executed on their system. The 110 submitted results contained system configuration information and the timings for both the primitive operations and the broker benchmarks. After filtering out duplicate submissions and submissions distorted by background load, the results were used to test the precision of the abstraction. On systems ranging from AMD 486DX4/100 with 32MB RAM to dual CPU Intel Pentium III/800 with 512MB RAM, the approach predicts broker performance with a typical mean error of 5 % and the worst error rarely exceeding 15 % 5 % and the worst error rarely exceeding 15 % using five primitive operations (Figure 12).

Not surprisingly, the approach fails to predict performance when shared resources are exhausted (Figure 13). This case, however, does not represent a typical mode of operation, thus it is sufficient just to warn when the prediction failure occurs. This can be done by measuring the basic resource usage of the broker, such as memory consumed per proxy and per servant, or threads consumed per connection, and warn when the prediction is done for cases that exhaust the shared resources.

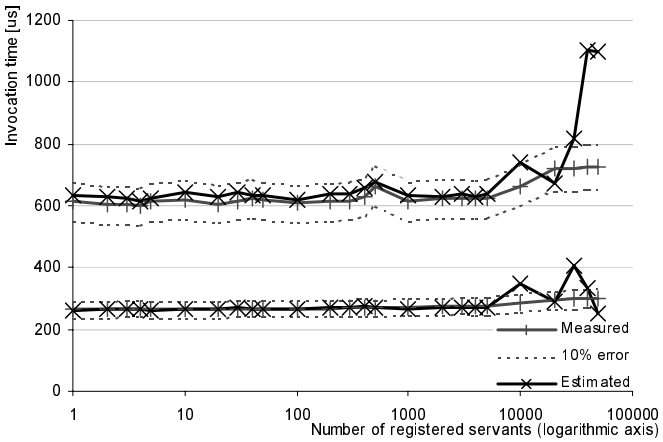

Figure 13 Roundtrip times, void ping() for increasing number of servants, static invocation, ORBacus 4.0.1, Linux 2.2.16, AMD K6 450 MHz, 128 MB RAM, Linux 2.2.18, Dual Pentium III 800 MHz, 512 MB RAM

4.3 Uses for portable results

The approach used to port performance results can answer the questions raised in the beginning of this section, and therefore is significantly more useful than usual approach to benchmarking. However, it still requires collecting measurements from at least 10–15 different systems to work. This means that it is still outside the usual possibilities of a single broker user to easily measure portable results.

Our experiment with the public suite of benchmarks, however, suggested that it is a realistic idea to set up a public web site, where results of benchmarks targeted at the broker user audience would be collected. Besides serving as a repository for code and results of the benchmarks, such a site would have enough data to provide portable and comparable results and to verify the submitted results by checking their compliance with the predicted ones. We are setting up such a web site at http://nenya.ms.mff.cuni.cz/~bench.

5 CONCLUSION

The benchmark suite for the broker vendor audience, outlined in section 2, provides a detailed overview of most factors influencing the broker performance, and was suc-

cessfully used in several benchmarking projects [12][13] with companies such as MLC Systeme, Bull Soft, IONA Technologies. To our knowledge, it is the first published outline of a benchmark suite for a CORBA broker that attempts to cover the entire functionality of the broker, complementing specialized benchmarks that focus for example on quality of service guarantees [26]. It has also been used as a basis for an EJB benchmark suite [11].

The benchmark suite for the broker user audience, outlined in section 2, provides a rough overview of the basic factors influencing the broker performance. In line with the suggestions of the OMG White Paper on Benchmarking [9], the suite is easy to execute and easy to port to different systems. It is also relevant in the sense outlined in section 3, because its results can be adjusted to reflect a particular mode of operation, and provides a good understanding of the broker performance issues. The suite has been implemented for C++ brokers and is available at http://nenya.ms.mff.cuni.cz/~bench.

An approach to porting the benchmark results has been suggested and, after tests on a wide range of systems, shown to provide very precise performance estimates. Using this approach, the benchmark results can be used to deduce system requirements from performance requirements, to compare the performance of brokers running at different systems, and other purposes that a usual benchmark does not lend itself to very well. For a future work, the approach should be verified on other than C++ platforms.

To conclude, we show that it is possible to generalize CORBA broker benchmarks into a relatively small suite, whose results can be tailored to a specific system and mode of operation without a prohibitive loss of precision.

6 ACKNOWLEDGEMENTS

The authors would like to thank members of the Distributed Systems Research Group at Charles University, headed by professor František Plášil, for cooperation on the benchmarking projects, and the industrial partners, for providing feedback and allowing us to use the results for research purposes.

7 REFERENCES

[1] Object Management Group, ORBOS PTF Benchmark RFI, OMG bench/98-05-01, 1998
[2] Callison, H.R., Butler, D.G., Real-time CORBA Trade Study, Boeing, 2000
[3] OMEX , CORBA Testbed, http://www.omex.ch, 1999
[4] Amar, V., CORBA Benchmarks Results, http://www.beust.com/virginie, 1999
[5] Plášil, P., Tůma, P., Buble, A., CORBA Benchmarking, Tech. Report No. 98/7 KSI, Charles University, 1998
[6] MITRE/Open Systems Center, Proposal to ORBOS PTF for Benchmarking CORBA Scalability, OMG bench/98-10-01, 1998
[7] Chung, M., France Telecom CNET Response to Benchmark RFI, OMG bench/98-10-05, 1998
[8] University of Helsinki, Response to ORBOS PTF Benchmark RFI, OMG bench/98-10-03, 1998
[9] Object Management Group, White Paper on Benchmarking, OMG bench/99-12-01, 1999
[10] Procházka, M., Tůma, P., Pospíšil, R., Enterprise JavaBeans Benchmarking, Tech. Report No. 4/2000 KSI, Charles University, 2000
[11] EJB Comparison Project, Final Report Public Distribution Version, Charles University, Prague, 2000
[12] CORBA Comparison Project Extension, Final Report, Charles University, Prague, 1999
[13] CORBA Comparison Project, Final Report, Charles University, Prague, 1998
[14] Schmidt, D.C., Gokhale, A., Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks, IEEE ICDCS 97, 1997
[15] CORBA 2.4.2, OMG formal/2001-02-33, 2001
[16] Schmidt, D.C., Vinoski, S., An Overview of the OMG CORBA Messaging Quality of Service (QoS) Framework, C++ Report, SIGS 3(12), March, 2000
[17] Schmidt, D. C., Evaluating Architectures for Multithreaded CORBA ORBs, CACM 10(41), 1998
[18] Mogul, J., Brittle Metrics in Operating Systems Research, Proceedings of HotOS 7, 1998
[19] Henning, J.L., SPEC CPU2000: Measuring CPU Performance in the New Millennium, IEEE/COMPUTER, July 2000
[20] SPEC CPU2000 1.1, http://www.spec.org, 2000
[21] TPC Council Policies, http://www.tpc.org, 2000
[22] Seltzer, M., Krinsky, D., Smith, K., Zhang, X., The Case for Application-Specific Benchmarking, Proceedings of HotOS 7, 1998
[23] Saavedra, R.H., Smith, A.J., Analysis of Benchmark Characteristics and Benchmark Performance Prediction, ACM Transactions on Computer Systems, 1996
[24] Brown, A., A Decompositional Approach to Performance Evaluation, Harvard University Computer Science Tech. TR-03-97, 1997
[25] Harman, H.H., Modern factor analysis, University of Chicago Press, 1967
[26] O'Ryan, C., Schmidt, D.C., Kuhns, F., Spivak, M., Parsons, J., Pyarali, I., Levine, D.L., Evaluating Policies and Mechanisms to Support Distributed Real-Time Applications with CORBA, The Journal of Theory and Practice of Object Systems 2(13), Wiley & Sons, 2001
[27] Vilicich, M., Aslam-Mir, S., Benchmark Metrics for Enterprise Object Request Brokers, Proceedings of TOOLS 99, 1999