Jacobsen, Hans-Arno; Weissman, Boris

Working Paper

Towards high-performance multithreaded CORBA servers

SFB 373 Discussion Paper, No. 1998,111

Provided in Cooperation with:
Collaborative Research Center 373: Quantification and Simulation of Economic Processes, Humboldt University Berlin

Suggested Citation: Jacobsen, Hans-Arno; Weissman, Boris (1998) : Towards high-performance multithreaded CORBA servers, SFB 373 Discussion Paper, No. 1998,111, Humboldt University of Berlin, Interdisciplinary Research Project 373: Quantification and Simulation of Economic Processes, Berlin,
https://nbn-resolving.de/urn:nbn:de:kobv:11-10060819

This Version is available at:
https://hdl.handle.net/10419/61279

Towards High–Performance Multithreaded CORBA Servers *

Hans - Arno Jacobsen†
Humboldt–Universitaet zu Berlin
Institut fuer Wirtschaftsinformatik
D–10178 Berlin
jacobsen@wiwi.hu-berlin.de

Boris Weissman
International Computer Science Institute
1947 Center Street
Berkeley, CA 94703
borisv@icsi.berkeley.edu

Abstract *Parallel platforms have become widely available. Moderately priced commodity SMPs are now manufactured by most major hardware vendors. Platform independent software environments, emphasizing a transparent programming model for building distributed applications, are rapidly emerging.*

In this paper we demonstrate how to combine the transparency characteristics of these environments with the high–performance features of the affordable server technology. We integrate the thread–per–request concurrency model into CORBA servers while providing high–performance. We demonstrate that thread creation overhead can be minimal and is merely attribute to the thread package used. We introduce and evaluate optimization techniques for increasing overall server performance. These techniques are based on increasing locality of reference for the client–server interaction.

Keywords: High–performance object request brokers, CORBA, mutlithreaded servers.

1 Introduction

With advances in microprocessor technology, parallel platforms have become widely available. Moderately priced commodity SMPs are now manufactured by most major hardware vendors.

At the same time platform independent software environments, such as CORBA [1], DCOM [2], and DCE [3], are rapidly emerging. These platforms aim at providing a transparent programming model for the development of portable and interoperable distributed applications while enabling efficient client–server computing.

The increase in affordable processing power, due to the availability of cheap SMPs on the one hand, and the growing need for high–performance computational servers in distributed environments on the other hand, make SMPs the platform of choice for distributed application servers.

However, to benefit from the increased processing power, an adequate programming model must be exposed to the application designer, so that the high–performance features and the available parallelism can be exploited by the application.

Little work has been done within the interoperability platforms specifying bodies, such as the OMG (CORBA) to integrate parallel programming models into the standards. It is therefore difficult to exploit the high–performance features of current server technology. CORBA offers language bindings for C, C++, SmallTalk, Java, Ada, and Eiffel, but does not leave room to integrate parallel extensions into the language binding in a non–proprietary and portable manner.

Some work has emerged [4, 5] that surveys implementation techniques for designing multithreaded servers based on the features CORBA provides. Proprietary extensions to support multithreaded CORBA servers have also been implemented for several commercial ORBs [6, 7]. An engineering solution for multithreading CORBA clients has been proposed by Hellemans *et al.* [8]. Attempts at extending the CORBA object model to handle data parallel computations have been introduced by Keahey and Gannon [9].

Despite this growing interest, no attempt, has so far, been made to explicitly quantify the performance improvements obtained from different server implementation techniques. No optimizations have been investigated to combine the transparency characteristics of CORBA with the high–performance features of SMPs to enable high–performance computational servers for distributed environments.

In this work we demonstrate how to integrate the thread–per–request concurrency model into CORBA servers while providing high–performance. In this model each incoming request is associated with a separate thread. The implementation is based on CORBA interceptors and is therefore fully portable. We have ex-

*In: Proceedings of The International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98),

†This research was supported by the German Research Society (DFG), SFB 373/A3.

tended the MICO CORBA ORB [10] implementation to concurrently process client requests.

We demonstrate that thread creation overhead can be minimal and is merely attribute to the thread package used. Thus user–exposed thread–pooling techniques, proposed by Schmidt and Vinoski [4] need not be applied. We feel that such exposure does not support an intuitive parallel programming model and is not necessary at this level of abstraction. As our performance study shows, a thread–per–request style of invocation is fully satisfactory. This latter point is often neglected which led to the proposal of using pre–spawned thread–pools to reduce thread creation overhead (cf. [4, 11]).

Furthermore, we propose and investigate techniques for increasing multithreaded server performance. These techniques are based on increasing locality of reference for the client–server interaction. By rearranging the server–side arriving request stream, such that requests from the same client are executed in a maximally long non–interspersed sequence on the same processor, we can achieve substantial performance improvements, over the execution of requests in the order of arrival. We measure server throughput and monitor the cache miss rate for a set of benchmarks to illustrate our findings. The benchmarks have been developed to correspond to different client–server interaction scenarios, inspired from emerging distributed applications, such as distributed transaction processing systems (e.g., ATM (automated teller machine) and loan–granting applications, online trading etc.).

The rest of the paper is organized as follows. Section 2 reviews the features of the CORBA standard relevant to our work, outlines the benefits of multithreading for computational server design, and introduces the thread package used. Section 3 presents techniques for preserving locality of reference and motivates our microbenchmark suite. In Section 4 we experimentally validate the techniques discussed.

2 Multithreaded CORBA servers

2.1 Multithreaded server programming

Threads are a programming abstraction to identify and utilize potential parallelisms in programs. A thread serves as the unit of execution. It constitutes a sequence of processing steps together with a program pointer, a stack pointer, and processor registers. A thread executes within the context of a process address space which it shares, as well as other process resources, with a potentially unlimited number of other threads.

This multithreaded programming model is well suited for implementing CORBA servers for two key reasons:

1. It provides a simple methodological model that allows the association of threads with user transactions (single requests, group of requests, a session, and a service), thereby providing a clean abstraction, resulting in a simpler program design.
2. It exploits the parallelism and high–performance features of SMPs.

Several alternative client–server interaction schemes have been proposed in the literature, e.g., [12, 4], for multithreading servers, such as:

- thread–per–request: a thread is associated with each incoming request
- thread–per–session: a thread is associated with each connecting client
- thread–per–transaction: a thread is associated with each individual transaction
- thread–per–object: a thread is associated with each object on the server–side
- thread–per–service: a thread is associated with each service provided on the server–side

Advantages and disadvantages of a number of these alternative interaction schemes is discussed in [12, 4]. As we will show later the thread–per–request scheme provides sufficient performance, and, at the same time, a clean programming abstraction.

2.2 The CORBA specification

The Common Object Request Broker Architecture (CORBA) is a standard for distributed computing which has been developed by the Object Management Group (OMG) [1], a consortium of independent companies. CORBA aims at providing a uniform communication infrastructure for building distributed applications. It supplies a unifying framework for interoperating software components, operating on various hardware platforms, running different operating systems. Furthermore, CORBA aims at providing programming language transparency.

The CORBA standard does not foresee explicit support for concurrent method invocation in its design. Client–server communication is based either on synchronous RPC–style invocation, or on deferred asynchronous invocation. Deferred invocation allow a client to send off multiple requests in sequence, and subsequently poll for results. A oneway invocation scheme is additionally provided. It allows a client to send off a request and continue processing. No knowledge about the successful completion of the request is communicated back to the client. Transfer semantic of such invocations is best effort.

2.3 Active Threads

Active Threads [14] is an extensible and portable high–performance thread system. It is based on an event driven architecture capable of supporting different scheduling policies. Active Threads supports user–extensible scheduling that exploits temporal and spatial locality. Active Threads hide hardware dependencies such as the number of CPUs. Instead, the user is provided with a virtual processor abstraction. At the application level, threads can be scheduled to run on virtual processors. Active Thread clients are encouraged to schedule threads that are likely to share data to run on the same virtual processor. Such data dependent scheduling annotations are likely to produce substantial performance benefits. No precise knowledge of the memory hierarchy is necessary, although some information may be beneficial. Virtual processors are multiplexed over the available cpus. The application can change the mapping by supplying customized schedulers. A library of schedulers is provided and can be extended by the application.

The Active Threads runtime provides basic thread services: thread initialization, start-up and context switch, thread stack management. A variety of extensible synchronization objects including spinning and two-phase locks, semaphores, barriers, and condition variables are supported. Much of the runtime is machine independent. Hardware dependent services are captured in the Machine-Dependent Layer.

3 Increasing server throughput

CORBA aims at providing a highly transparent programming environment for developing distributed applications (cf. Section 2). Remote method invocations appear just like local invocations, i.e., a proxy object acts on behalf of the remote object in the client's address space. To process a remote invocation at the server side, data and instructions of the receiving object have to be loaded into the cache possibly replacing objects from previous and concurrent invocations. Abstractly speaking, a working set is built up in the server–side cache reflecting the activity of the client–server interaction. To increase performance, i.e., decrease the cache miss rate, it would be desirable to maintain the working sets in the cache as long as possible.

In general, request streams from independent clients will arrive at the server in a random order, i.e., alternate client requests arrive randomly interspersed. Processing of request in a first–come–first–serve (FCFS) fashion has negative effects on the cache hit rate. By rearranging requests such that sequences of 'related' requests are executed in a non–interspersed fashion, possibly on the same processor, would reduce the cache miss rate and hence improve performance.

One crucial aspect of this scheme is the scheduling of requests, i.e., which requests should be serviced next and how long should an idle processor wait before executing a request. One simple heuristic is to pick a request from the largest, yet unserviced, group of requests. This is to maximize the locality of reference potentially obtainable for the selected group by executing all its requests on the same processor. A group of such requests is usually associated with an interacting client or transaction.

As the number of clients increases, server processors are likely to often switch among request streams. To lower the number of such switches, a processor that becomes idle may switch to a stream with the largest number of available requests only when this number is above a threshold. It may even be beneficial for an idle processor to wait for a certain interval before switching to a different request stream. One possibility is to wait for the same time that it takes to reload a request's working set in cache before switching to a different stream (the price of such reload is unavoidable for a switch). Many variations of these policies are possible. In this work we study the case outlined above and defer the investigation and evaluation of alternative policies.

Many distributed applications benefit from cache reuse. Take, for example, any application that involves multistep transaction processing, like ATM (automated teller machine systems), loan granting systems, distributed database query processing, database servers, online trading, and the like. In all these cases, multiple independent clients will asynchronously initiate requests to a server. Due to the inherent asynchronity, will the arrival order of requests be random and a priori unpredictable at the server side. On the other hand, network services, such as directory and naming services, that involve a simple one–step transaction would not benefit considerably, since such lookups do not exhibit locality.

We use the simple thread–per–request concurrency model to implement the parallel server. The key benefits of this model include the following:

1. simplicity of the model,
2. good semantics match — logically separate requests are handled by physically different threads,
3. simplicity of the implementation — no need to maintain thread pools or any other queuing and load–balancing data structures. This functionality is usually already handled by the thread system and should not be duplicated.

Our implementation is based on the MICO CORBA ORB [10] and its implementation of request level interceptors [1, 15]. MICO is adopting interceptors as they are specified in [15].

Interceptors are application objects whose operations are invoked by the ORB in a pre–defined order. Thus they may be used to invoke operations at different stages of the processing of a remote method invocation. This feature can for example be used for logging, authentication, and message transformation, among others. We have used it to associate incoming requests with threads, to implement the thread–per–request concurrency model.

4 Performance evaluation

4.1 Microbenchmarks

To investigate the performance effects of the server thread locality management, we have designed a series of microbenchmarks. Our goal was to keep the benchmarks simple and yet to be able to vary the memory reference patterns in order to observe the effects of thread locality in different contexts.

In our base benchmark, a series of clients send streams of requests to the server. Each request results in a number of memory accesses with a stride that can be varied. We consider the cases of read and update operations. Update involves reading and modifying a word. All requests from the same stream access the same memory locations. The working sets of different streams on the server do not overlap. The server executes on the SMP, and the client processes run on single processing workstations connected by Ethernet.

The experimental parameters involved are: `p` — the number of processors used by the server to handle the incoming client requests; `c` — the number of clients; `s` — the stride size in bytes; `n` — the number of memory accesses.

In all experiments, the server is using a thread–per–request concurrency model — a separate server thread is created for each incoming request [4]. In the base case, the server threads are scheduled over the processors on the server SMP in the first–come–first–serve (FCFS) order. We then examine the performance implications of scheduling the requests from the same stream to execute on the same server processor, if possible. We are mostly interested in the overall server throughput in terms of the number of requests per second. We also examine the contributing factors such as the number of cache misses on the server machine.

4.2 Hardware Platforms

We use several platforms in our experiments. The server of our main platform is a 4–way SPARCStation–10. The client machines are UltraSPARC–1 workstations connected to the server by Ethernet.

To measure different server cache statistics, we use an 8cpu Sun Enterprise 5000 server (the UltraSPARC cpus have the necessary performance monitoring hardware unavailable on the HyperSPARC cpus).

Table 4 summarizes the relevant characteristics of the hardware platforms. The UltraSPARC cpus have 2 level caches: separate on–chip instruction and data caches (I–cache and D–cache), and a unified secondary cache (E–cache). For Enterprise 5000, the E–cache miss penalty is 50 cycles if the line in question is not cached by any other processors and 80 cycles otherwise.

4.3 Experiments

4.3.1 Concurrency model

We use the simple thread–per–request concurrency model to implement a parallel server (cf. Section 3). Many sources dismiss this model based on performance and resource utilization considerations [4]. We, however, argue that the poor thread creation, synchronization, and scalability properties are merely artifacts of the particular implementations rather than the features inherent in the model. For instance, with Active Threads [14], it is possible to achieve thread creation and synchronization latencies of a few microseconds on a variety of different architectures. This is usually only a fraction of a percent of the TCP/IP protocol handling cost. Therefore, thread overheads may have virtually no impact on the CORBA servers based on TCP/IP and the associated OS kernel trap overheads. Active Threads can also be configured to allocate thread stacks lazily as needed. For sufficiently independent requests, this results in similar memory consumption as that demonstrated by the other more complex server concurrency models (thread pools, etc.).

Figure 1 shows throughputs for a CORBA server running on the 4–way SS10 SMP relative to a single–processor server (p=4,c=4,s=16,n=8192). The server employs a simple FCFS policy for thread scheduling. The left side of the figure deals with the standard invocations and the right–hand side characterizes the "oneway" invocations. Both figures demonstrate fairly good scalability and thus substantiate our claim that a thread–per–request model is not inherently expensive, Furthermore, we show in the following sections that the throughputs are constrained by the thread locality scheduling properties rather than the high thread management overheads. We will also show how taking the locality properties into account can substantially improve the throughputs within the basic framework of the thread–per–request concurrency model.

System	Level	Size	Block/ page size	Line count	Associativity	Miss penalty (to next level)
SS10 SMP,						
4 50 Mhz cpus,	L2	256Kb	32	8K	direct	50
HyperSPARC	DRAM	128M	4K	32K	full	–
Sun E5000,	I–cache	16Kb	32	512	2-way	3 cycles
8 167 Mhz cpus	D–cache	16Kb	32	512	direct	3 cycles
UltraSPARC-1	E–cache	512Kb	64	8K	direct	50/80
	DRAM	512Mb	8K	64K	full	-
client Ultra-1,	I-cache	16Kb	32	512	2-way	3 cycles
167 Mhz	D-cache	16Kb	32	512	direct	3 cycles
	E-cache	512Kb	64	8K	direct	42 cycles
	DRAM	128Mb	8K	16K	full	–

Table 1: Hardware platform characteristics.

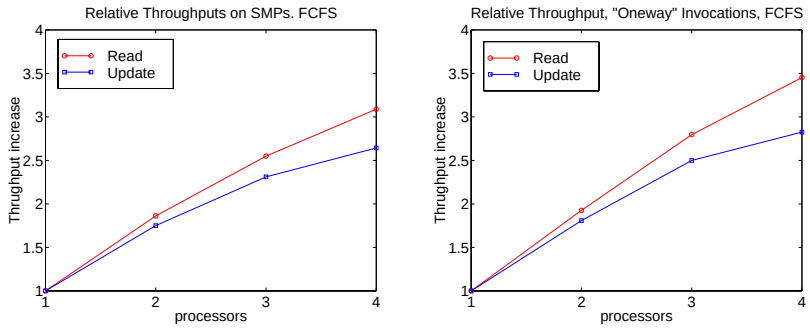

Figure 1: Throughput for server running on 4–way SS10 SMP relative to a single–processor server (p=4,c=4,s=16,n=8192).

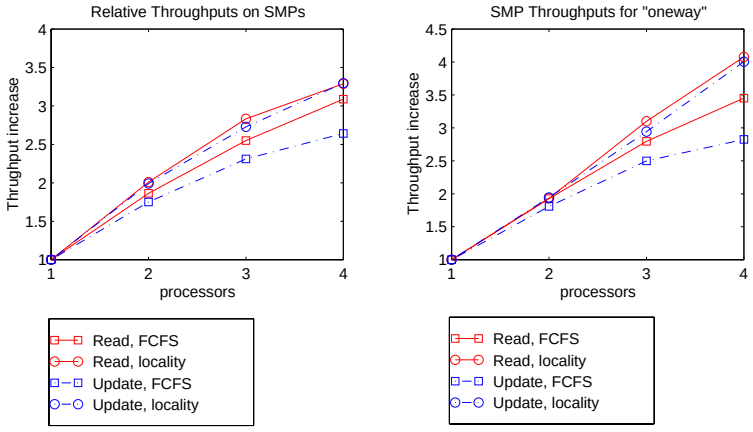

Figure 2: Throughput for server running on 4–way SS10 SMP relative to a single–processor server (p=4,c=4,s=16,n=8192), comparing FCFS policy with locality policy.

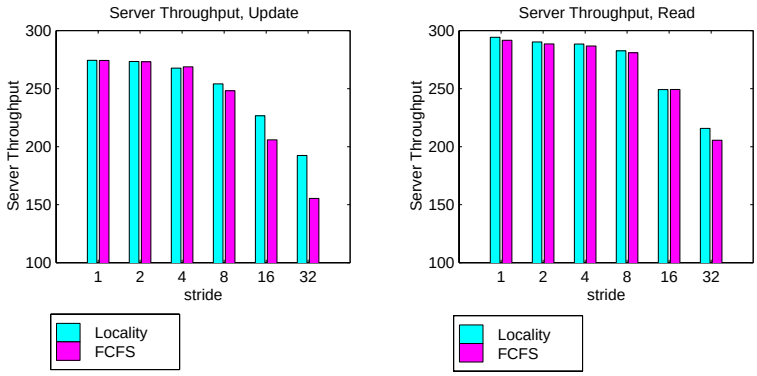

Figure 3: Absolute throughputs for 4 cpu servers for different access strides.

4.3.2 Thread locality and throughputs

To illustrate the importance of the thread locality management, we have replaced the server's default FCFS policy with a policy that respects the locality of requests within the streams. Active Threads permits annotations to tag the threads that belong to the same logical group at the creation time. Active Threads attempts to execute threads within the same group on the same processor, if possible. However, when a processor becomes idle and no threads from a previous group is available, the processor switches to a group with the largest number of threads.

The ORB dispatch code has been modified to annotate threads that handle requests from the same stream with unique identical tags. The resulting server speedups for the benchmark from the previous section are shown in Figure 2.

Figure 2 also repeats the original SMP speedup curves corresponding to the FCFS policy. Taking thread locality into account results in significant performance gains and demonstrates that a thread–per–request concurrency model does not necessarily have negative performance implications.

Figures 3 and 4 provide further quantitative insights into the importance of locality management. The figures show absolute throughputs for 4 cpu servers for different access strides. As the stride increases, the cache reload cost is amortized over a fewer number of accesses. Therefore, the relative importance of preferentially servicing requests whose working sets are already in cache (even partially) increases. The figures also indicate that "oneway" invocations are relatively more sensitive to locality management. For instance, for updates with stride s=16, the locality policy results in 21% improvement for "oneway" invocations and only 9% gain for the regular calls. This is due to a smaller absolute communications cost and hence the server "computation" makes up a larger portion of the request lifetime.

To study the effects of the number of clients, we have fixed the stride at 32, and varied the number of client processes (for more than 16 clients, we have multiplexed a number of client processes over the same set of workstations). Figure 5 demonstrates the effects of the locality policies as the number of clients increases. We have also modified our base locality policy. For a large number of clients, server processor are likely to often switch among request streams. To lower the number of such switches, a processor that becomes idle switches to a stream with a largest number of available requests, but only when this number is above a threshold. In a more practical setting, in the absence of threads from the previously serviced stream, it may be beneficial for an idle processor to wait for a certain interval before switching to a different stream. One possible heuristics is to wait for the same time as it takes to reload a request working set in cache before switching to a different stream (the price of such reload is unavoidable for a switch).

4.3.3 Locality and Secondary Caches

The lower throughputs of the FCFS policy is mostly explained by the greater number of the secondary cache misses when processors often switch between request streams. Many modern processors such as PentiumPro [16], UltraSPARC [17], RS6000 [18] have performance monitoring hardware that enables user–level access to the external cache miss counters.

We have instrumented our server code to read the performance instrumentation counters (PICs) of the UltraSPARC processor [17]. We then repeated our experiments on the Sun Enterprise 5000 server with 8 167Mhz UltraSPARC–1 cpus. Figure 6 corresponds to the following configuration p=8, c=64, s=64. We varied the

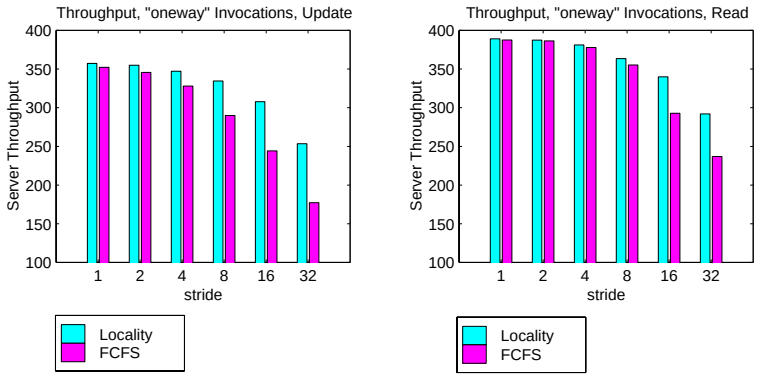

Figure 4: Absolute throughputs for 4 cpu servers for different access strides.

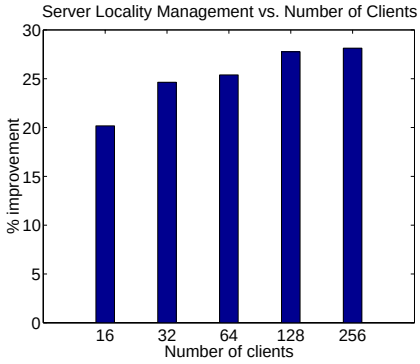

Figure 5: Effects of the locality policies as the number of clients increases.

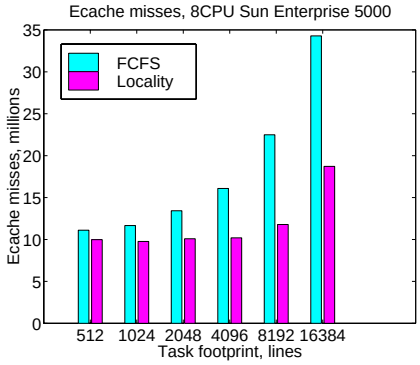

Figure 6: Total number of E–cache misses.

size of the footprint each request leaves on the server from 512 lines to 16384 lines (each E–cache line is 64 bytes). Figure 6 presents the total number of E–cache misses (a sum of E–cache misses across all server processors). For a large range of footprints (up to 8192 lines), the total number of E–cache misses for the locality policy remains unaffected by the increased footprint size. In contrast, the number of misses for the FCFS policy is proportional to the footprint size. For very large requests that do not fit in the secondary caches, the number of E–cache misses is affected by the footprint size for the locality policy as well. However, since the secondary caches of the modern servers tend to be fairly large, locality scheduling is likely to be a significant factor for many applications. For instance, the E–cache of Sun Enterprise server can be up to 4Mb [19], B–cache of DEC AlphaServer 4100 is also up to 4Mb [20], the external cache of HP Exemplar is 1Mb [21].

5 Conclusion

We have quantified the performance gains exploiting request locality in a distributed system. The distributed applications that may benefit the most from this scheme are transaction processing systems, where many clients interact with a server through heavy weight transactions. The developed optimizations aim at exploiting locality of reference between the client–transaction and the server.

We aim at the further development of policies to schedule threads from different groups of waiting requests. The developed optimization techniques trade off fairness for performance, i.e., a particular earlier arriving request may have to wait for the benefit of increasing the overall server throughput. We aim at analytically quantifying this problem. This will allow us to formulate policies that take fairness considerations into account while keeping performance within a specified bound, i.e., introduce the notion of quality of service in the threaded

execution model.

References

[1] OMG. The Common Object Request Broker Architecture and Specification. Revision 2.1. Technical Report, Object Management Group, 1997.

[2] Microsoft and Digital Equipment Corporatio. Distributed Component Object Model Specification. Technical Report, Microsoft, October 1996. Draft version 1.0 edition.

[3] Open Software Foundation. OSF Distributed Computing Environment Rationale. Cambridge, MA, 1990.

[4] D. C. Schmidt and S Vinoski. Object interconnections. comparing alternative programming techniques for multi–threaded servers. *SIGS C++ Report*, February-August 1996. Columns 5, 6, 7.

[5] R. Orfalli, D. Harkey, and J. Edwards. *Client Server programming with Java and CORBA*. John Wiley & Sons, INC., 1997.

[6] Iona Technologies. Orbix/orbixweb. http://www.iona.com/Orbix/index.html.

[7] OOC. Omnibroker. http://www.ooc.com/ob.html.

[8] P. Hellemans, F. Steegmans, H. Vanderstraeten, and H. Zuidweg. Implementation of hidden concurrency in CORBA clients. In O. Spaniol, C. Linnhoff-Popien, and B. Meyer, editors, *Trends in Distributed Systems: CORBA and Beyond*, pages 30–42. Springer Verlag, October 1996.

[9] K. Keahey and D. Gannon. Pardis: A parallel approach to corba. In *International Symposium on High Performance Distributed Computing*, pages 31–9, Los Alamitos, CA, USA, Aug 1997. IEEE Comput. Soc.

[10] A. Puder and K. Römer. *MICO is CORBA — A CORBA 2.0 compliant implementation.* dpunkt.verlag, Heidelberg, 1998.

[11] S. Baker. *Corba Distributed Objects : Using Orbix.* Addison Wesley, 1997.

[12] Jr. H. W. Lockhart. *OSF DCE. Guide to developing distributed applications.* McGraw–Hill Inc., 1994.

[13] S. Vinoski. CORBA Integrating Diverse Applications Within Distributed heterogeneous Environments. *IEEE Communications Magazine*, 14(2), Feb 1997.

[14] B. Weissman. Active Threads: An extensible and portable light-weight thread system. Technical Report, ICSI TR-97-036, Nov 1997.

[15] Alcatel, Hewlett-Packarad Company, Lucent technology Inc., et al. Realtime corba. Technical Report, Object Managment Group, January 19 1998. OMG document number orbos/98-01-08.

[16] Intel Corporation. *Pentium Pro Family Developer's Manual.* Intel Corporation, December 1995. Volume 3: Operating System Writer's Guide.

[17] Sun Microsystems. *UltraSPARC-1 User's Manual*, 1996.

[18] E. H. Welbon, C. C. Chan-Nui, D. J. Shippy, and D. A. Hicks. The power2 performance monitor. IBM internal paper.

[19] Sun Microsystems. The ultra enterprise 1 and 2 server architecture. Technical Report, Sun Microsystems, April 1996. Technical White Paper.

[20] M. B. Steinman, G. J. Harris, A. Kocev, V. C. Lamere, and R. D. Pannell. The alphaserver 4100 cached processor module architecture and design. *Digital Technical Journal*, April 1997.

[21] Hewlett Packard. Hp exemplar technical servers. Technical report, Hewlett Packard, 1998. Technical Specification. Available at http://www.hp.com/wsg/products/servers/servhome.html.