Università di Pisa
Dipartimento di Informatica

Technical Report: TR-04-08

Accelerating Apache farms through ad-HOC distributed scalable objects repository

M. Aldinucci M. Torquati

February 10, 2004

ADDRESS: via F. Buonarroti 2, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Accelerating Apache farms through ad-HOC distributed scalable objects repository

M. Aldinucci M. Torquati

February 10, 2004

Abstract

We present HOC: a fast, scalable objects repository providing programmers with a general storage module. HOC may be used to implement DSMs as well as distributed cache subsystems. HOC is composed by a set of hot-pluggable cooperating processes that may sustain close to optimal network traffic rates. We designed an HOC-based Web cache that extends the Apache Web server and remarkably improves Apache farms performances with no modification to the Apache core code.

1 Introduction

The demand for performance, propelled by both challenging scientific and industrial problems, has been steadily increasing in past decades. In addition, broadband networks growing availability has boosted Web traffic and therefore the demand of high-performance Web servers. Distributed memory Beowulf clusters are gaining more and more interest as low cost parallel architectures meeting such performance demand. This is especially true for industrial applications that require a very aggressive development and deployment time for both hardware solutions and applications, e.g. software reuse, integration and interoperability of parallel applications with the already developed standard tools.

We present HOC (Herd of Objects Caches), i.e. a distributed *objects* repository specifically thought for Beowulf class clusters. HOC provides applications with a distributed storage manager that virtualize processing elements (PEs) primary memories into an unique common memory. As we shall see, HOC is not yet another Distributed Shared Memory (DSM), it rather implements a more basic facility. It can be used as DSMs building block as well as for other purposes. Indeed, in this paper we present the design of a cooperative cache built on top of HOC for farms of the *Apache* Web server [3]. HOC provides Apache farms with a hot-pluggable, scalable cache that considerably improves Apache farms performance with no modification to the standard Apache 2.x core code, thus meeting key industrial requirements.

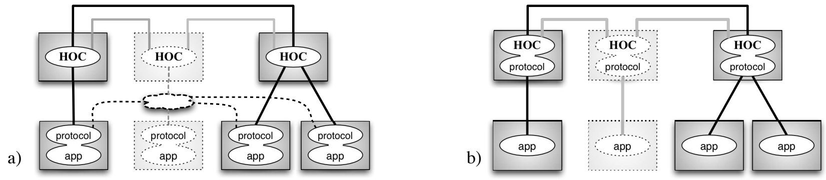


Figure 1: Typical architectural schemes of applications based on HOC. a) With data access protocol within the application. b) Extending HOC external operations.

In Sec. 2 we present HOC design principles, HOC implementation and its raw performance. In Sec. 3 we discuss Web server test-bed peculiarities and HOC+Apache architecture. Experiments on such architecture are discussed in Sec. 4. The paper is completed by conclusion.

2 HOC Design Principles

The HOC underlying design principle consists in to clearly decouple the management of computation and storage in distributed applications. The development of a parallel/distributed application is often legitimated by the need of processing large bunches of data. Therefore data storages are required to be fast, dynamically scalable and enough reliable to survive to some hardware/software failures. Decoupling helps in providing a broad class of parallel applications with these features while providing very good performances.

HOC, as DSMs, virtualizes PEs primary memories in a common distributed repository. As a matter of fact, it realizes a memory hierarchy additional layer that lays upon O.S. virtual memory, but that should be used instead of – or together with – disk-based virtual memory. The HOC implements an *external* storage facility, i.e. a repository for arbitrary length, contiguous segments of data (namely *objects*). Objects are identified by a configurable length *key* and have a *home node*, i.e. the reference node for the object (that is currently statically assigned to objects). It enables applications to fully utilize the underlying strength of a cluster, such as fast network communication speed and huge aggregated memory space. Current and near future network technologies clearly indicates that such a net-based virtual memories may perform much better than disk-based ones. Literature reports very good results both for general DSMs [4, 1] and for specific Web applications [8, 9, 6]. A large and fast data repository may be used as:

- a cache facility to improve performance of I/O bound applications;
- a primary storage facility for CPU bound applications running out-of-core on a single PE memory.

HOC, unlike some DSMs, is quite robust and simple. It does not natively implements any consistency mechanism for data copies and does not force already developed applications to be rewritten. As shown in Fig. 1, the HOC-based

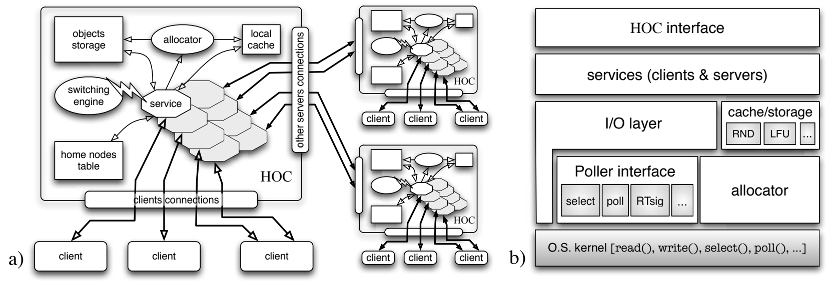


Figure 2: HOC architecture. a) Functional view. b) Implementation layers.

architecture distinguishes the raw data access layer from the protocol each application requires to access data. In many cases, for example in the context of Web caching, a data consistency mechanism is not required and may introduce unnecessary overheads. In general, HOC may be enriched with locking and consistency mechanism at the protocol level (see Fig. 1 a).

2.1 HOC implementation

A HOC server is implemented as a C++ single thread process; it relies on non-blocking I/O to manage concurrent TCP connections [12, 7]. The HOC core consists in an executor of a dynamic set of finite state machine, namely *services*, each of them realized as a C++ object. Each service reacts to socket-related events raised by O.S. kernel (i.e. connections become writable/readable, new connection arrivals, connection closures, etc.). In the case one service must wait on an event, it consolidates its state and switch the control to another one. The HOC core never blocks on I/O network operations: neither on `read()/write()` system calls nor on HOC protocol primitives like remote node memory accesses. The event triggering layer is derived from `Poller` interface [13], and may be configured to use several POSIX connections multiplexing mechanisms, such as: `select`, `poll`, and Real-Time signals[1].

HOC architecture is sketched in Fig. 2. A HOC may serve many clients, each of them exploiting many connections. Moreover, a server may cooperate with other HOCs through a configurable number of connections. Connections both with clients and other servers may be established or detached at any time during HOC lifespan by using the (optional) *Dynamic Configuration Module* (DCM). DCM keeps track of HOCs status and statistics, it enables the dynamic update of objects home, and notify all HOCs whether a new HOC is hot-added to the architecture. It can be also used as HOC interface toward back-end servers such as application and DB servers. As an example a DB-trigger may perform, through DCM, a collective operation over all HOCs in order to invalidate a given entry.

[1]The primitive choice is managed by an autoconf generated configuration script.

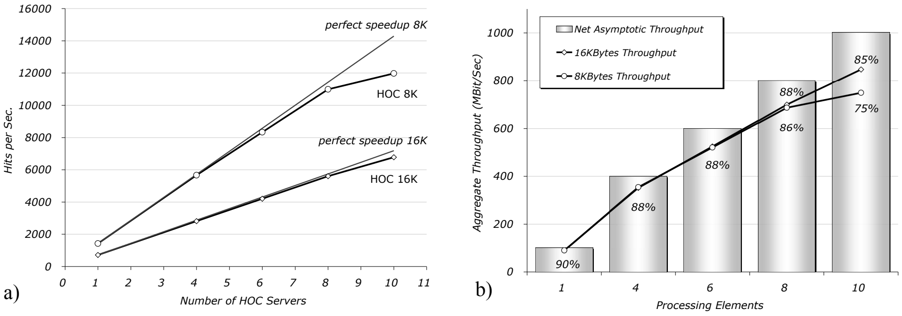


Figure 3: a) HOC scalability of served requests. b) Aggregate throughput. Experiments are referred to 200,000 `get` operations w.r.t. a data set of 40,000 objects (8K and 16K average sizes) having cyclically distributed homes among HOCs. Each HOC serves a partition of requests referred to keys that are uniformly distributed and randomly chosen in the whole data set. Each HOC have a local cache of 1,000 objects.

A connection is exclusively managed by a service. Services rely on an *allocator* in order to store objects into both a *objects storage* and a write-back *cache* facilities. They may be both managed as a cache of a configurable but fixed amount of objects and used to store server home objects and to cache remote home objects respectively. As shown in Fig. 2 b) each of them may be managed by using a replacing policy chosen at configuration time. Currently HOC comes with RND (random) and LFU (Least Frequently Used) policies, however it is designed to be easily extended with other policies. HOC offers three basic object-related services (external operations): `get`, `put`, and `remove`. Let us discuss operations behavior. Assuming to have a set of nodes each of them running an HOC, and HOC_i receive:

– `get`(x). If $home(x) = i$, a local objects $storage\_get(x)$ is issued. If otherwise $home(x) = j, j \neq i$ a local $cache\_get(x)$ and, in case of miss, a $remote\_get(x, j)$ is issued. In the latter case the $object(x)$ is stored in the i local cache. Anyway, depending on the get results, either the $object(x)$ or a miss message is sent to the client.

– `put`(x). If $home(x) = i$, $object(x)$ is put in the node i storage. Otherwise $object(x)$ is stored in the node i local cache. A capacity miss can occur in both cases. In the former case the victim object is simply deleted[2], in the latter the write-back reconciling protocol is started.

– `remove`(x). The $object(x)$ is purged from both objects storages and caches of all nodes.

[2]Whether configured as a cache, otherwise a *storage full* message is sent to the client.

a) *Raw data set*

Total size	4 GB
N. of files	100,000
N. of requests	250,000
N. of files < 256 KB	96209 (2 GB)
Static pages	100%

b) *Accesses log*

Data transfered	$\sim$ 9 GB
N. of distinct files requests	74226 (2.8 GB)
Avg. file size	36.8 KB
N. of distinct files < 256 KB	71449 (1.5 GB)
Avg. size of distinct files < 256 KB	8 KB

c) *Apache 2.0.47 MPM worker configuration (hybrid multi-threaded multi-process) [3]*

StartServers	4	MaxClients	512	ThreadPerChild	64	Log level	Notice
ServerLimit	8	MinSpareThreads	32	MaxRequestsPerChild	0	Access log	None

Table 1: Summary experimental environment. a) Raw data set characteristics. b) Access log characteristics. c) Apache Web server configuration.

2.2 HOC Performances

HOC performances has been extensively tested on several homogeneous clusters. We report here tests performed on a 21 PEs RLX Blade; each PE runs Linux (2.4.18 kernel) and is equipped with an Intel P3@800MHz, 1GB RAM, a 4200rpm disk and three 100Mbit/s switched Eth devices. As shown in Fig. 3, HOC scales very well up to maximum number of PEs available in our testing environment (10 HOCs and 10 clients). In the tests two different Ethernets are used for client-server and server-server connections. Fig. 3 b) shows that HOC can deliver an aggregate throughput that is quite close to the (quite optimistic) 100Mbit/s per PE asymptotic network bandwidth. We experienced that local cache size has a little impact on performances; it mainly acts as a network buffer.

Other tests performed on a Linux (2.4.22 kernel) cluster of 8 PEs P4@2GHz, 512MB RAM, connected through a 1Gbit/s Ethernet have confirmed for a single HOC and many clients an aggregate throughput of $\sim$91MB/s ($\sim$73% of the very optimistic 125MB/s asymptotic bandwidth) with 2048 concurrent stable connections for a test of 50,000 requests of 1MB objects. We also experienced more than 20,000 replies/s with 3,072 concurrent stable connections for a test of 200,000 requests of 512B objects. In all tested cases we experienced that HOC sustain a throughput close to the network bandwidth. HOC has been also tested on a Apple PPC-G4 cluster running MacOS 10.3.2.

3 Web Caching Test-bed

Web caches have became the standard method to ensure high-quality throughput for Web access. Web caching can be adopted at different levels: 1) Web browser storing Web objects in its local memory or disk; 2) proxy server sitting somewhere between the clients and the Web server (typically at ISP level) serving a large community of clients; 3) server accelerator (reverse proxy) sitting in front of one or more Web servers [17, 15].

Reverse proxies has been demonstrating the best solution among them. Differently from 1 and 2, they are under the site manager control and typically enable dynamic pages management through an API which allows application programs to explicitly add, delete, and update cached data [11, 10]. However, a reverse proxy may introduce unnecessary latencies due to the additional parsing and filtering of requests, including those leading to not cacheable replies. To mitigate these problems we adopt a different solution: we place a cache on back of the Web server in order to improve server performances. The cache cope with cacheable replies only and does not make any additional parsing on them. As we shall see in the next section, we implemented the solution by using a HOC-based architecture. Our approach is similar to others appeared in literature [18, 8, 6, 9]. Despite other approaches we do not designed another Web server but we propose a decoupled architecture that is composed by the standard Apache Web server and a HOC-based storage subsystem. Apache Web server is unmodified in core functionalities, we just modified the allocation policy of one optional Apache module. Therefore the architecture may benefit from Apache popularity, correctness and people expertise.

In a cache implemented on a single node (single or even multi-processor), the throughput is limited by the network interface. With cluster based Web cache the throughput can be increased simply adding more nodes. Since many HOCs may cooperate one each other, the cache throughput is not limited by the network interface, but by the aggregate cluster throughput.

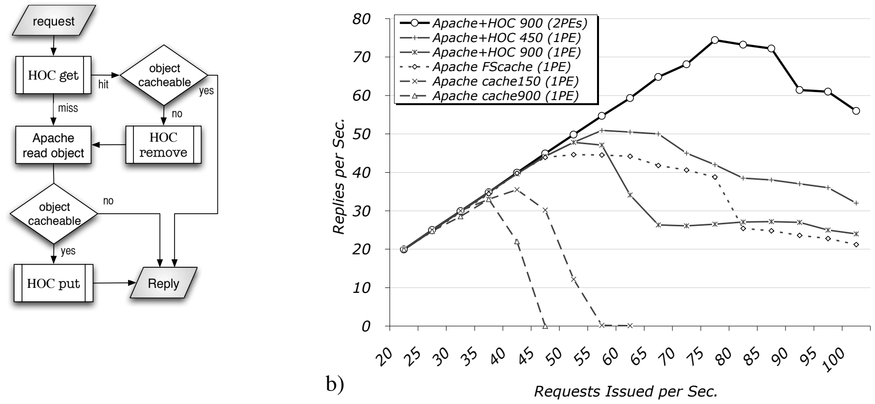
3.1 HOC+Apache Architecture

The HOC+Apache architecture is compliant to Fig. 1 a). In this case the *app* is the Apache Web server, the *protocol* is a modified version of *mod_mem_cache* Apache module [3] (compliant to RFC2616), and dashed lines are not present. In particular, *mod_mem_cache* has been modified by only substituting local memory allocation, read and write with HOC primitives as shown in Fig. 4 a). No other functionalities has been modified. Both original and modified caches are able to cache static and dynamic pages. Notice that the original version of *mod_mem_cache* implements a per process cache, thus different Apache processes never share the cache even whether they are running on the same PE.

We used as objects key the MD5 digest of the *mod_mem_cache* native key. In order to enforce correctness we include in the object to be stored into HOC original HTTP request and reply headers. Protocol does not need any additional consistency mechanism but ones ensured by *mod_mem_cache*.

4 Web Caching Experiments

We measured the performance of Apaches+HOC architecture on the RLX Blade described in Sec. 2.2. The main characteristics of the data set and accesses log are summarized in Table 1 a) and b). The data set is generated according to [5] by using a Zipf-like requests distribution with $\alpha = 0.7$. The 90% of files have small size (min. 2KB, max. 100KB, avg. ∼13KB) and the 10% of files have medium-large size (min. 101KB, max. 1MB, avg. ∼280KB). In all tests we used the Apache 2.0.47 Web server in the *MPM Worker* configuration shown



a) b)

Figure 4: a) High-level work-flow of *mod_mem_cache* changes needed to interoperate with HOC. b) Replies rate tests for stand-alone Apache and HOC+Apache architecture. Tested dataset and Apache configuration are shown in Table 1. Stand-alone Apache is tested with 900MB and 150MB native cache per process and without cache (FScache). HOC+Apache is tested with 450MB and 900MB devoted to HOC (Apache and HOC mapped on the same PE). HOC+Apache is also with Apache and HOC (900MB) on different PEs. Tests on the RLX Blade, see Sec. 2.2.

in Table 1 c). HTTP requests are issued by means of the *httperf* program [14] configured to count replies only within 1 second timeout. Each httperf is directly connected to an Apache with no switch/balancer in the middle. The whole site is replicated on all PEs running Apache.

In Fig. 4 b) we compare Apache against Apache+HOC performances; the test takes in to account three basic configurations:

i) a stand-alone Apache with no cache, with the *mod_mem_cache* (Apache native cache) exploiting both 150MB and 900MB cache per process;

ii) an Apache+HOC running on the same PE, HOC exploiting both 450MB and 900MB of total memory accessed by all Apache processes;

iii) an Apache+HOC running on different PEs, HOC exploiting 900MB of total memory accessed by all Apache processes.

As clear from Fig. 4 b), in all cases the HOC+Apache architecture overwhelms the stand-alone Apache with or without the native cache.

i) We notice that the Apache with the original cache lose its stability when the requests rate grows. This is due to a number of different factors. The foremost is the lack of a common cache storage for all processes on the same PE that leads to the replication of the same objects in several caches, and therefore to the harmful memory usage. This rapidly leads the O.S. to the swap bor-

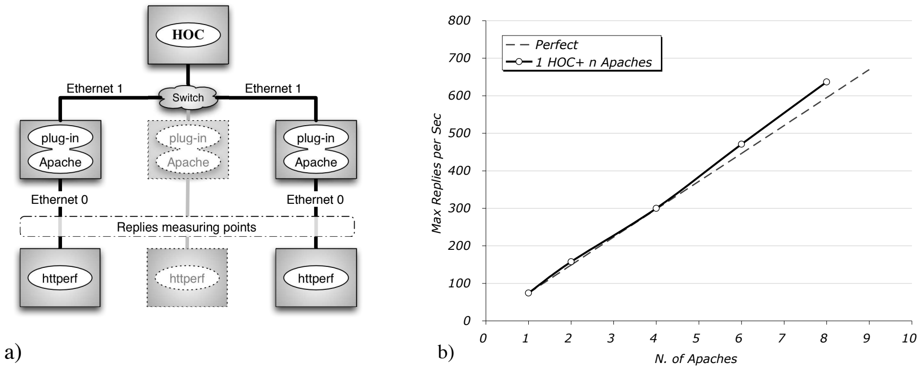


Figure 5: One HOC supporting many Apaches. a) Experiment schema. b) Replies per second speedup. HOC is configured with 900MB of objects storage. Dataset and Apache configuration are summarized in Table 1. Tests on the RLX Blade, see Sec. 2.2.

der resulting in a huge increase of reply latency (also w.r.t the limited timeout chosen in our stress test). As clear from Fig. 4 b) the Apache performs better without the native cache. In reality this behavior is due to the File System cache; during the test we measured that the FS cache grows up to $\sim$ 700MB. In this case the performance also depends on site organization on disks. In general FS cache is unsuitable for Web objects since requests do not exploit spatial and temporal locality w.r.t. disk-blocks [5]. Moreover, FS cache is totally useless for dynamic Web pages, for which we experienced that Apache native cache module is effective (as we expected from the literature [11]).

ii) In the cases the Apache and HOC are mapped on the same PE, the performance improvement is due to HOC cache sharing among Apache processes. Since the accessed files set does not fit in memory, performance are influenced by replacing policies. Here, two different caching mechanism and three policies are active at the same time: 1) the file system cache and HOC allocates memory in different bulks from the same physical memory; 2) the file system cache replacing policy is active on the first bulk, the Apache GreadyDual-Size (GDSF) and the HOC LFU replacing policies are active in cascade on the second bulk. The very complex scenario prevents a fine analysis of system bottlenecks. Indeed, the decoupling approach we followed sought to simplify the design by throwing away at least the first point.

iii) As a matter of fact, the 2PEs figures from Fig. 4 b) confirm that mapping Apache and HOC on different PEs significantly improves performances. By the way, the same tests have been performed using as HOC replacing policy the random function instead of LFU. The performance degradation is just 8.7% (from 74.4 to 67 avg. replies) even if the accessed data set does not fit on

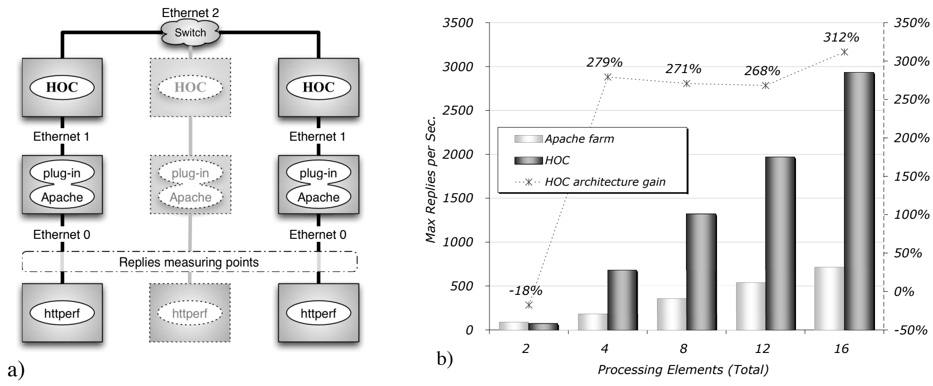


Figure 6: Many cooperating HOCs experiment. a) Experiment schema. b) Dark bars shows the replies/s speedup w.r.t. the total amount of used PEs (n-HOCs and n-Apaches). Light bars shows the same measure for a reference architecture ($2n$-Apaches on the fastest tested configuration FScache, see Fig. 4). Dashed line sketch the gain of HOC-based architecture over the reference architecture. The HOC local cache is fixed to 5,000 objects in all cases. Tests on the RLX Blade, see Sec. 2.2.

the HOC storage, that suggests that memory size have stronger impact over performances w.r.t replacing policy (in this case, at least).

Fig. 5 highlights that HOC may support many Apaches with a very good, over linear scalability due to the partition of requests among several Apaches (with induces a lower disk load). It worth noticing that HOC+8Apaches sustain an aggregate of 637 hits/s with an average file size of 8KB on the Apache-HOC link (see Table 1 b). In this case HOC works at $\sim$ 45% of maximum reply rate (As shown in Fig. 1 HOC reaches 1430 hit/s with N=1 and the same message payload).

Fig. 6 show the scalability of n Apache+HOC. Also in this case we experienced a very good scalability up to our testing environment limit reaching a 312% gain w.r.t. an Apache farm with $2n$ PEs (Apache in the fastest configuration). Whether compared with Apache with the original cache module the gain reaches the 406%. Observe that the big gap occurs from 1HOC+1Apache and 2HOC+2Apaches, where the whole set of all cacheable objects begins to fit in the aggregate memory.

5 Conclusion

We have presented HOC, a fast, scalable, hot-pluggable "storage component" for homogeneous cluster architectures. HOC is a distributed storage server; it is developed by using state-of-the-art server design technologies. As described in Sec. 2.2, this technology enables HOC to cope with a quite large number of concurrent connections and to sustain a very high throughput in both single and parallel configurations. We developed and tested a HOC-based Apache plug-in module that greatly improves Apache Web server farms performances with no modifications to the Apache core code. To our knowledge no other works have targeted the problem with no modification to the server code (see Sec. 3).

We are currently improving HOC in several directions. First, by introducing multithreading to make it scalable also on SMP boxes. Second, by integrating HOC within the ASSIST parallel programming environment [16]. Overall, we envision a complex application composed by decoupled components, each delivering a very specific service. HOC has been thought as a building block of more general applications and programming environments. In this design, the application or the programming environment run-time support is supposed to provide the correct protocol to consistently use HOC external operations. A this end we are developing a protocol that offers standard hooks to implement several DSM consistency models [2, 1].

Acknowledgments. We wish to thank M. Danelutto and M. Vanneschi for many fruitful discussions, and A. Petrocelli who has contributed in experimenting HOC.

References

[1] M. Aldinucci. eskimo: experimenting with skeletons in the shared address model. *Parallel Processing Letters*, 13(3):449–460, September 2003.

[2] G. Antoniu and L. Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. of the 6th Intl. HIPS Workshop*, number 2026 in LNCS. Springer, April 2001.

[3] Apache Software Foundation. *Apache Web Server*, 2003. (http://httpd.apache.org).

[4] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, J. Langendoen, T. Rühl, and M. F. Kaashoek. Performance evaluation of the orca shared-object system. *ACM Transactions on Computer Systems (TOCS)*, 16(1), February 1998.

[5] L. Brelau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. of the Infocom Conference*, 1999.

[6] E. Cecchet. Whoops! : A clustered web cache for dsm systems using memory mapped networks. In *Proc. of the 22nd IEEE ICDCSW Conference*, July 2002.

[7] A. Chandra and D. Mosberger. Scalability of Linux event-dispatch mechanisms. Technical Report HPL-2000-174, HP Labs., Palo Alto, USA, December 2000.

[8] G. Chen, C. L. Wang, and F. C. M. Lau. A scalable cluster-based web server with cooperative caching support. *Computation and Currency: Practice and Experience*, 15(7–8), 2003.

[9] F. M. Cuenca-Acuna and T. D. Nguyen. Cooperative caching middleware for cluster-based servers. In *Proc. of the 10th IEEE HPDC*, August 2001.

[10] B. Hines. *Livin' on the Edge: Caching Dynamic Web Content with IBM WebSphere Application Server and WebSphere Edge Server*. IBM, September 2001. (http://www-106.ibm.com/developerworks/websphere/techjournal/0109_hines/hines.html).

[11] A. Iyengar and J. Challenger. Improving Web server performance by caching dynamic data. In *Proc. of the USENIX Symposium on Internet Technologies and Systems Proceedings*, Berkeley, CA, USA, Dec. 1997.

[12] D. Kegel. *The C10K problem*, 2003. (http://www.kegel.com/c10k.html).

[13] D. Kegel. *Poller Interface*, 2003. (http://www.kegel.com/poller/).

[14] D. Mosberger and T. Jin. httperf – a tool for measuring Web server performances. In *Proc. on Internet Server Performance (WISP)*, Madison, USA, June 1998.

[15] M. Rabinovich and O. Spatsheck. *Web Caching and Replication*. A. Wesley, 2001.

[16] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12), Dec. 2002.

[17] J. Wang. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review*, 29(5), 1999.

[18] Whizz Technology. *EC-Cache Engine*, 2003. (http://www.whizztech.com/ec_cache.html).