

Dynamic Adaptation of the Squid Web Cache with Arachne

Marc Ségura-Devillechaise, Jean-Marc Menaud, Nicolas Lorient, Thomas Fritz, Rémi Douence, Mario Südholt, and Egon Wuchner.

Abstract—Networking software, in particular popular web caches as Squid, are highly optimized for execution speed and therefore dispense with several common software design principles, e.g., a modular architecture. However, this is an important impediment to their adaptation to new requirements, such as the extension of an existing web cache protocol or closing a security hole. Furthermore, such adaptations typically crosscut the applications’ legacy code.

In this paper, we investigate the use of Arachne, a system for Aspect-Oriented Programming of C-applications. We present three different realistic case studies of adaptations of the Squid web cache: correction of a security hole, prefetching introduction, and a protocol extension. These case studies show, in particular, that Arachne’s expressive aspect language, in particular its notion of sequence aspects, allows for the concise modularization of these adaptations. Furthermore, we give evidence that Arachne’s dynamic weaver allows such adaptations to be performed without a perceptible performance overhead.

WRITING (good) software is often a challenge. Writing adaptable software is even more difficult. For example, an open implementation must provide a functional interface exposing the services offered by the application along with an adaptation interface allowing the customization of the implementation [1]. However, since an adaptation interface often increases complexity, common wisdoms, such as “Keep It Simple, Stupid (KISS)” and “time to market”, encourage software designers to sacrifice adaptability.

Networking software is a perfect illustration of the current situation. Since this software has to meet high performance and availability constraints, design goals such as adaptability and modularity are frequently neglected in the absence of immediate benefits brought. For example, a web cache - a software running on a machine sitting between web servers and clients and locally replicating requested web pages to increase performance - could be written modularly by composing an `http` parsing library like `libwww`, a threading library such as the `posix` one, and by using the operating system’s virtual memory mechanism to replicate web pages on disk. For performance reasons, real world web cache implementations are instead large monolithic C applications usually written from

scratch. For instance, the open-source web cache Squid does not even manage resources using operating system services. Instead, Squid’s event-based architecture communicates with the operating system and directly reacts to the availability of network cards and hard drives.

Such software is hard to maintain and to adapt. But issues such as latency frequently require adaptations to networking software. Latency is defined as the delay separating the emission and the reception of a piece of information (e.g., an IP packet or a chunk of `http` data) between two machines. Latency is limited by the speed of the physical signal used to exchange information. Since Internet birth, latency has remained stable with an average value around 100 ms [2], [3]. Within the Internet backbone, the latency drops to 30 ms almost reaching the theoretic limit imposed by the speed of light and the earth perimeter [4]. Web caches have been a first attempt to cope with latency. Nowadays, in order to reduce latency, the exploitation of software replication techniques has become a business and continuous adaptation of existing legacy web caches for techniques like prefetching is required.

This shows the typical contradicting forces that software designers have to reconcile: performance and simplicity but also adaptability and modularity. Using Squid as an example, this paper proposes to design network software without adaptation interfaces, thus keeping the implementation as simple and efficient as possible. Adaptation interfaces should be defined on a per-need basis. In Squid, as generally in legacy web caches, such adaptation interfaces are typically needed for functionalities which crosscut the legacy code, i.e., functionalities whose code is scattered and tangled in the different files and functions comprising the web cache code. Indeed it is a common observation underlying the field of Aspect-Oriented Programming [5] that the introduction of adaptation interfaces into large applications after the event results in crosscutting code and inherently defies efforts of a modular design. Furthermore, anticipation of adaptation needs has its limits: many cannot reasonably be planned for, e.g., modifications required by security breaches arising from bugs.

In this article we propose to realize adaptations on a per-need basis with Arachne, an aspect-oriented system we devised for the C programming language. The key contribution of this paper is to show that by using Arachne it is possible to adapt Squid in a modular way without sacrificing high performance, even without the need to plan adaptations a priori. To this end, we show how three realistic adaptations can be performed: correcting security breaching, reducing latency via prefetching introduction, and adding support for

Marc Ségura-Devillechaise, Jean-Marc Menaud, Nicolas Lorient, Rémi Douence, and Mario Südholt are members of the OBASCO project, École des Mines de Nantes - INRIA, LINA, 4, rue Alfred Kastler, 44307 Nantes Cedex 3, France, e-mail: {msegura,jmenaudo, nloriant, douence, sudholt}@emn.fr.

Thomas Fritz is a member of the Software Practices Lab, University of British Columbia, 201-2366 Main Mall, Vancouver, BC V6T 1Z4, Canada, e-mail: fritz@cs.ubc.ca.

Egon Wuchner is a member of the Corporate Technology, SE2, Siemens AG, Otto-Hahn-Ring 6, 81739 München, Germany, e-mail: Egon.Wuchner@siemens.com.

the Internet Content Adaptation Protocol (ICAP) to Squid. We illustrate how the Arachne language, especially its sequence aspect feature, enables the simple formulation of protocol manipulations, and illustrate through the examples that Arachne’s dynamic weaving process enables such adaptations to be realized without perceptible overhead.

The rest of this article is structured as follows: the first section describes the Arachne language and runtime toolbox. Section II discusses three case studies of how we used Arachne to adapt the Squid web cache. The third section presents some performance evaluations. Section IV compares Arachne to other aspect-oriented systems.

I. THE ARACHNE ASPECT SYSTEM

In this section we describe the Arachne aspect language and how it is used in the context of the Arachne weaver tool chain.

A. Aspect language

Pointcuts in Arachne match constructions of the C programming language, the most frequent implementation language for web caches. Advice essentially consists of calls to C functions, which may be executed in addition to or instead of legacy function calls. Finally, a tool, Arachne’s weaver, can be used to apply a set of aspects to C applications. While Arachne’s pointcut and advice languages are essentially similar to those of AspectJ, Arachne is set apart from AspectJ and similar systems by two main features. First, Arachne’s aspect languages features sequence aspects, which are useful for the formulation of protocol manipulations, in particular, the web cache manipulations presented in this article. Second, weaving is dynamic, i.e., adaptations can be applied to running C applications without stopping them. Dynamic weaving benefits, for example, the introduction of prefetching strategies and the correction of security breaches into a web cache, because it preserves its service availability.

1) *Joinpoints*: The join point model of an AOP system defines the relevant basic execution events, i.e., those events which can be referred to in pointcuts and at which occurrences advice may modify the program execution. Arachne features two basic kinds of join points: (i) C function calls, (ii) read and write accesses to global variables and local aliases to them.

2) *Pointcuts*: Arachne provides a pointcut language, which is similar to AspectJ, to match the different types of join points described above and access information about the corresponding execution state. For instance, with a pointcut of the form `call(void * xcalloc(size_t, size_t))` a call to the function “xcalloc” with the given signature can be denoted. The constructor `writeGlobal(var)` matches a write access to a global variable, while `write(var)` also matches accesses to the variable’s local aliases, i.e., aliases of global variables having local scope. The constructor `controlflow`, which takes a list of function call pointcuts as argument, allows to denote sequences of nested function calls, similar to AspectJ’s `cflow-construct`. Pointcut expressions can be combined, e.g., using logical combinators, such as “or” (`||`). Furthermore, “binder” expressions allow programmers to retrieve information about the execution state in which a join point occurs:

`args` and `return` can be applied to function call join points to access the arguments and the return value of the function call; `value` can be used with read and write accesses and allows to retrieve the value being read or written. Finally, an `if(C)` pointcut enables matching to be restricted to contexts where the expression C holds.

3) *Advice*: Advice consists of C function calls introduced with the keyword `then`. The function called by advice must be defined in a regular C source code file compiled along with the aspect source code file containing the advice. By default Arachne replaces the execution of the join point by the execution of the advice. When the advice is omitted, the base program join point is executed.

4) *Sequence aspects*: Sequence aspects, denoted by `seq(sts)`, consist of a list of steps each of which associates a pointcut with a (possibly empty) advice. A sequence instance is created each time the pointcut of the first step matches a join point. Further steps are activated in a “greedy” fashion, i.e., the step following the current one is activated as soon as its pointcut matches. All but the first and last steps can be repeated zero or multiple times by using the repetition operator ‘*’.

A key feature of sequence aspects is that any data bound in a step (with `arg`, `return`, or `value`) can be used in a later step. This is especially useful to match the value of a variable of the base program and accesses to its local aliases: one step can bind an address (using `arg`, `return`, or `value`) and subsequent steps can restrict accesses with `read` and `write` to that address.

Every time a join point matches the pointcut of the aspect’s first step, a sequence instance is created and space for all the data an aspect is interested in is allocated. During the execution of the sequence aspect, the data is collected, and as soon as the last step of the sequence is executed for a particular instance, the associated memory is freed. Sequences can therefore be realized using a small representation of the history of the base program execution in form of a list of such sequence instances which is updated when a new step in the sequence occurs.

B. Compiler and runtime tool chain

The Arachne tool chain runs on PENTIUM machines under the Linux operating system. It is composed of three tools: an aspect compiler, a weaver and a deweaver. The compiler, shell command `acc`, transforms aspect source code into an aspect dynamic link library (DLL)¹. To ease interoperability with legacy code, `acc` can link the aspect DLL it is compiling with other DLLs and with static libraries or object files produced by other compilers. Based on mappings between the symbolic descriptions of rewriting sites and the actual rewriting sites, Arachne rewrites binary code referenced by the aspect with hooks that point to the aspect DLLs. (A more detailed description of the compilation and weaving process is given in [6].)

The dynamic weaver, shell command `weave`, applies an aspect DLL to a running process. `weave <pid> <aspect-library>` weaves the compiled aspect library

¹also known under Linux as a shared library

into the process identified by the process id `pid`. The weaver supposes that the base program has been compiled without function inlining and that the symbols generated at compile-time have not been removed from the base program. These assumptions are not uncommon: the default compilation process of the `squid` web cache for example meets these expectations. In general, Arachne exploits the interface standards for binary code and linking information [7], [8] which govern the execution of a compiled file and do not depend on code patterns generated by specific compilers. Therefore aspects can be woven into any code adhering to these standards. In addition, Arachne provides a `deweaver`, `deweave` to deweave aspects from an application.

II. WRITING ASPECTS WITH ARACHNE

As most legacy high-performance web caches, Squid is designed around an event-based architecture, which breaks down event handling and request processing into many different functions and uses function pointers and state machines to drive execution. For performance reasons, functions are overloaded with several concerns at once and do not clearly reflect the flow of execution from event detection to response building. Thus, adaptations of Squid’s behavior tend to require modifications at many different places. As Squid amounts to several megabytes of undocumented source code such adaptations get very complex.

To assess Arachne, we consider in the following two criteria: expressiveness and performance. We first focus on expressiveness: does Arachne allow to implement useful adaptations of the Squid web cache² concisely and modularly? Our adaptation examples range from a very simple one where Arachne is used to remove a security threat via protocol modifications to reduce latency by prefetching documents to the inclusion of a complete network protocol (ICAP).

A. Correcting a security hole

In February 2002, the CERT issued a vulnerability note on Squid versions 2.3 and 2.4, pointing out a buffer overflow in the FTP authentication mechanism. The function `rfc1378_escape_part` could overflow the buffer’s `base_href` and `title_url` fields contained in the structure `FtpStateData`. An appropriate exploitation could result in denial of service attacks, thus compromising latency guarantees.

The Squid team corrected the mistake by distributing a patch to apply to the Squid source code. This patch alters the way the functions `ftpStateFree`, `ftpListingStart`, `ftpBuildTitleUrl`, `ftpListDir`, and `ftpReadSize` manipulate the fields `title_url` and `base_href` of the structure `FtpStateData`.

While it is possible to rewrite this patch as a collection of Arachne aspects correcting this specific buffer overflow, we can also write a sequence aspect preventing a class of, i.e., containing against yet unreported, buffer overflows. The code

of this aspect is presented in figure 1. This sequence aspect defines a sequence of Squid functions, the protocol, which lead to a buffer overflow. The first step of this sequence aspect is an aspect retrieving the buffer length at allocation time (call of `xcalloc`). The second matches assignments made to that buffer. An advice is attached to this step which replaces the faulty assignment. This advice uses the regular C function `reallocAndWrite` to resize and copy the appropriate data into the buffer. The last step indicates that the buffer is no longer used and allows Arachne to free the memory associated with the collected data.

In contrast to the patch written by the Squid team that requires in-depth comprehension of the parsing of ftp requests, our sequence aspect is based on simple knowledge about buffer creation and use. In addition, as security threats are usually first reported once the cache is in production, weaving aspects on the fly is a great advantage. The traditional approach - patching the source code, recompiling it, stopping the running version, and starting the new version - would have at least implied the loss of the web pages replicated in RAM. Therefore the traditional recompilation approach significantly degrades latency as RAM is faster than disk memory.

B. Adding prefetching over HTTP to reduce latency

In the last two years, a number of Web browsers have started to download pages before they are requested by the end-user [9]. Such prefetching schemes trade network bandwidth to reduce the end-user perceived latency. As intermediaries, web caches are better suited to prefetch pages than individual users. However Squid does not include prefetching features. We have implemented the simple prefetching strategy proposed by Chinen and Yamaguchi [10] that prefetches ten hyperlinks referenced in an HTML page served by the cache. Benchmarks showed that this strategy doubles the number of pages served directly from the local cache storage upon an end user request at the expense of doubling the consumed bandwidth.

The prefetching adaptation crosscuts the Squid functions which process HTTP requests. As shown in figure 2 it has been implemented using three aspects which modify the behavior of the functions handling the data reception. The first aspect starts prefetching on the creation of a HTTP response by `clientBuildReply`. During the transmission of a page by `comm.write_mbuf`, the second aspect retrieves the hyperlinks contained in the page. The third aspect then does the actual prefetching and retrieves a few pages among the detected hyperlinks. To avoid infinite loops, the different pieces of advice distinguish between pages requested by a regular client or for prefetching purposes using the function `isPrefetch`, which retrieves the corresponding information from a hashtable.

Instead of transforming an existing cache, Chinen and Yamaguchi designed and the Web cache `kotetu` from scratch to assess the benefits of their prefetching policy. The latest version of `kotetu` consists of 38762 lines of source code and offers fewer features than Squid. In comparison, the source code size of our adaptation does not exceed 1059 lines of code. Moreover, as prefetching trades bandwidth for latency,

²Unless explicitly noted, we used `squid-2.5STABLE3` as a test bed for our adaptations.

```

seq( /* first step : retrieve buffer and buffer size */
    call(void * xmalloc(size_t, size_t)) && args(numberOfElements, elementSize) && return(buffer) ;
    /* second step : identify and replace faulty assignments */
    write(buffer) && size(writtenSize) && value(newValue) && if(writtenSize > numberOfElements * elementSize)
        then reallocAndWrite(buffer, allocatedSize, writtenSize, newValue); *
    /* third step : free memory associated to sequence when buffer is freed */
    call(void xfree(void*)) && args(buffer); )

```

Fig. 1. An Aspect Preventing Buffer Overflow

```

/* start prefetching on creation of HTTP response */
controlflow(call(void clientSendMoreData(void*, char*, size_t)),
            call(HttpReply * clientBuildReply(clientHttpRequest*, char*, size_t))
            && args( request, buffer, bufferSize ))
    then startPrefetching(request, buffer, bufferSize);

/* retrieve hyperlinks during page transmission */
controlflow( call(void clientSendMoreData(void*, char*, size_t)),
            call(void comm_write_mbuf(int, MemBuf, void*, void*))
            && args(fd, mb, handler, handlerData) && if(! isPrefetch(handler)) )
    then parseHyperlinks(fd, mb, handler, handlerData);

/* prefetch pages on completion of write */
call(void clientWriteComplete(int, char*, size_t, int, void*))
    && args(fd, buf, size, error, data) && if(! isPrefetch(data))
    then retrieveHyperlinks(fd, buf, size, error, data);

```

Fig. 2. Aspects for Prefetching

prefetching is used best when the load on the network is low and to avoid it otherwise. With Arachne, you have the ability to dynamically weave and unweave aspects and thus to optimize the use of prefetching.

C. Adding ICAP support

Online advertising is difficult. First, providers want advertisements to be changed on a regular basis. An end user consulting a site twice should see two different advertisements. Second, the advertising providers need to measure the audience gathered by each advertisement. Therefore, most of the time, advertisements are marked as not cacheable, thus preventing caches to reduce latency. Some suggested to delegate the insertion of advertisements to Web caches [11]. Empowering Web caches with content transformation has motivated the Internet Content Adaptation Protocol (ICAP). An ICAP server is co-located with a proxy or a cache. Every time the latter receives a request or a response, it forwards it to the ICAP server. The ICAP server is then given the opportunity to modify it if desired. Processing continues by considering solely the modified request or response returned by the ICAP server. In addition, the specification allows the ICAP server to satisfy a request: in this case, the response returned by the ICAP server is returned by the cache to the end user without further processing. Squid does not yet support the ICAP protocol.

We used Arachne to turn Squid into an ICAP client. In other words, we exploited Arachne to dynamically add a new network protocol to Squid. First, since ICAP-enabled caches

are strongly advised to advertise their ICAP ability, our aspects add a *X-ICAP* header to the HTTP requests and responses entering and leaving the cache. This allows content providers to send a *proxylet*, i.e. a piece of code, that runs on the ICAP server near the cache. Thus the content provider can delegate some of its processing to the proxylet, including the insertion of advertisements. The second role of our probe is to load the ICAP adaptation in Squid upon a proxylet reception. All in all, the aspects composing the adaptation essentially modify the behavior of 15 Squid functions amounting to 554 KB. Due to space constraints we will not present the code here.

III. STUDYING ARACHNE PERFORMANCE

Adaptability benefits can not be evaluated at cache design time. It is therefore crucial that Arachne does not trade performance for adaptability. To study this point, we now estimate the average overhead introduced in Squid by Arachne using our prefetching aspect collection. Finally we compare the performance of Squid adapted by modifying its C source code by hand to the performance of Squid adapted with Arachne.

Measuring the average overhead introduced by Arachne by itself requires a significant modification of Squid. We have chosen to use Squid augmented with our prefetching adaptation. We built a profiling version of Arachne that tracks the time spent in the code generated under the hood by Arachne. We have compared this duration to the time spent in the adaptation code (i.e., in the prefetching code), to the time spent in Squid code to serve different Web pages and to the

TABLE I
TIME SPENT IN THE SQUID BASE PROGRAM, IN ARACHNE AND WITHIN THE PREFETCHING ADAPTATION CODE.

All values are averaged over 200 runs. The size column indicates the size of the downloaded Web page. The client column reports the time observed using `time` to fetch the page. The cache column gives the time needed by Squid to handle the query as reported by its own timing mechanism in the log files. The prefetching column reports the time spent running the adaptation code measured through the `rdtsc` instruction. The Arachne column measures the time spent in the Arachne infrastructure evaluated through the `rdtsc` instruction. An attentive reader will note some slight variations in the time spent in prefetching code: we have used publicly available pages containing different amounts of links. A miss corresponds to the case where the requested page has not yet been replicated in the cache. A hit describes the case where the requested page is already available in the cache.

Size (KB)	Client (sec)	Cache (sec)	Prefetching (μ sec)	Arachne (μ sec)	Client (sec)	Cache (sec)	Prefetching (μ sec)	Arachne (μ sec)
3.8	<0.1	<0.1	2	0.008	<0.1	<0.1	2	0.003
46	<0.1	<0.1	39	0.01	<0.1	<0.1	39	0.007
70	0.1	<0.1	113	0.014	<0.1	<0.1	110	0.011
90	<0.1	<0.1	122	0.016	<0.1	<0.1	122	0.014
163	<0.1	<0.1	154	0.028	<0.1	<0.1	283	0.045
196	0.2	0.2	365	0.055	0.2	0.2	356	0.020
301	0.1	<0.1	43	0.044	<0.1	<0.1	42	0.051
1182	0.6	0.6	924	0.141	0.5	0.5	919	0.142
3875	1.3	1.3	16679	0.677	1	1	1801	0.446

(a) Miss case

(b) Hit case

duration needed for the user to fetch the page from the cache. Table I summarizes our results. For this particular adaptation, the time spent in Arachne never exceeds one thousandth of the time needed to run the adaptation code. And the latter is several orders of magnitude smaller than the time required by Squid to serve a page.

To conclude our performance evaluations, we have compared the overhead of using Arachne to the overhead of adaptations made by manual modification of the source code. We have manually modified the Squid source code to introduce the same prefetching strategy used by our Arachne-based prefetching adaptation. We have then benchmarked the performance of the two caches with Web Polygraph [12]. After filling up the cache, the POLYMIX-4 workload mimics a one-day simulation including the two request rate peaks observed in production environments. Filling up the cache is a necessary and lengthy operation before evaluating its performance: for example, requesting a page that the cache has to fetch from the Web is usually hundred to thousand times slower than fetching a page that the cache has replicated locally. We realized ten simulations using our prefetching enabled Squid versions. The two peak phases are the most interesting cases since they stress the cache with a high throughput. Table II summarizes the average results for these two phases. The request rate and throughput are all given from the client side. Due to the effect of prefetching, the server side request rate was close to 43 req/s. Table II shows that there are no significant differences between the manual and Arachne-based prefetching caches: 1% on the average, when the average variation between two simulations with the same cache version is about 1.5%. The miss times, i.e., the time to deliver a document when it is not present in the cache, as well as the hit times, i.e., the time required for a document present in the cache, and also the response times are all very similar for the two cache variants. This simulation shows that there is no perceptible performance difference between a static prefetching integration achieved by manual source code modification and the dynamic prefetching integration performed by Arachne.

IV. RELATED WORK

AspectC [13] and AspectC++ [14] extend C and C++, respectively, by an aspect model very similar to AspectJ's one [5]. They all rely on source-code transformation and thus cannot apply aspects to running C applications. Furthermore, considering language expressiveness both approaches provide only support for aspects addressing single events and not sequences of events like our sequence aspects do.

Toskana [15], DAC++ [16], and JAsCo [17] are three examples of dynamic weavers that, like Arachne, rewrite at runtime the compiled code executed by the processor. Toskana however is limited to operating system kernels. DAC++ supports only C++ applications and JAsCo targets Java programs. Thus, none of the three dynamic weavers can be applied to a legacy application such as the Squid web cache.

Tools like Dyninst [18] and Pin [19] provide APIs that support binary rewriting of arbitrary assembly instructions at runtime as well as dynamic code patching. However, these tools work at an abstraction level much lower than the base program's higher-level programming language. While it might technically be feasible to devise an aspect system on top of them, the absence of a well-defined and complete relation between C source and compiled code remains a major problem. In addition, technical issues limit the feasibility of these approaches. Rewriting code with Dyninst, for example, is difficult as the rewriting process might fail. Pin on the other side does only allow to insert code after or before a binary instruction but not to replace one and thus the implementation of around advice poses a problem.

V. CONCLUSION

Networking software has to meet strong performance and availability constraints. As the benefits of designing an adaptable implementation can not be assessed at design time, adaptability is often sacrificed for the sake of performance and simplicity. The Squid Web cache is no exception. But many reasons ranging from the need to cope with security

TABLE II
POLYMIX-4 RESULTS FOR TWO MAIN PHASES

	ARACHNE Phase1	Manual Phase1	ARACHNE Phase2	Manual Phase2	diff Phase1 -Phase2
Throughput (req/s)	5.59	5.59	5.58	5.59	
Average response time (ms)	1131,42	1146,07	1085,31	1074,55	1,2 % - -1%
Response time for a miss (ms)	2533,50	2539,52	2528,35	2525,34	0,2% - 1,8 %
Response time for a hit (ms)	28,96	28,76	30,62	31,84	-0,6 % - 3,8 %
Hit Ratio	59,76	59,35	61,77	62,22	-0,6% - 0,7 %
Errors	0.51	0.50	0.34	0.34	-1,9 % - 0%

threats to the necessity of dealing with modifications of existing protocols call for adaptations. These adaptations require unanticipated modifications of the source code and frequently crosscut the implementation.

To cope with these problems, we have devised Arachne, a dynamic weaver for legacy C application featuring an expressive aspect language. This paper has shown that by using Arachne it is possible to support seemingly irreconcilable software quality attributes: performance and simplicity as well as adaptability and modularity. We have shown that an application like the Squid web cache can be modified without fixing adaptation interfaces a priori or sacrificing performance. Concretely, we have shown how three realistic cases of adaptations for web caching can be concisely modularized using Arachne's aspect language, in particular its sequence aspect feature, and shown through some benchmarks that the application of these aspects to a running Squid application results in no perceptible overhead.

REFERENCES

- [1] G. Kiczales, "Beyond the black box: Open implementation," *IEEE Softw.*, vol. 13, no. 1, pp. 8–11, 1996.
- [2] D. L. Mills, "RFC 889: Internet delay experiments," Dec. 1983.
- [3] J. Pointek, F. Shull, R. Tesoriero, and A. Agrawala, "Netdyn revisited: a replicated study of network dynamics," *Comput. Netw. ISDN Syst.*, vol. 29, no. 7, pp. 831–840, 1997.
- [4] B.-Y. Choi, S. Moon, Z.-L. Zhang, K. Papagiannaki, and C. Diot, "Analysis of point-to-point packet delay in an operational network," in *IEEE Infocom*, Hong Kong, Mar. 2004.
- [5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, ser. Lecture Notes in Computer Science, vol. 1241, New York, NY, June 1997, pp. 220–242.
- [6] R. Douence, T. Fritz, N. Lorient, J.-M. Menaud, M. Sgura-Devillechaise, and M. Sdholt, "An expressive aspect language for system applications with arachne," in *Proceedings of the 4th international conference on Aspect-oriented software development*. Chicago, USA: ACM Press, Mar. 2005.
- [7] U. S. L. System Unix, *System V Application Binary Interface Intel 386 Architecture Processor Supplement*. Prentice Hall Trade, 1994.
- [8] TIS Committee, "Tool interface standard (TIS) executable and linking format (ELF) specification," May 1995, version 1.2.
- [9] D. Fisher and G. Saksena, "Link prefetching in mozilla: A server-driven approach," in *Proceedings of Eighth International Workshop on Web Content Caching and Distribution*, Sept. 2003.
- [10] K.-I. Chinen and S. Yamaguchi, "An interactive prefetching proxy server for improvement of WWW latency," in *Seventh Annual Conference of the Internet Society (INET'97)*, Kuala Lumpur, June 1997.
- [11] A. Gupta and G. Baehr, "Ad insertion at prxies to improve cache hit rates," in *Proceedings of the fourth International Web Caching Workshop*, Apr. 1999.
- [12] A. Rousskov and D. Wessels, "High-performance benchmarking with Web Polygraph," *Software Practice and Experience*, vol. 34, no. 2, pp. 187–211, Feb. 2004.
- [13] Y. Coady and G. Kiczales, "Back to the future: a retroactive study of aspect evolution in operating system code," in *Proceedings of the 2nd international conference on Aspect-oriented software development*. Boston, MA: ACM Press, Mar. 2003, pp. 50–59.
- [14] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An aspect-oriented extension to the C++ programming language," in *40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, Feb. 2002.
- [15] M. Engel and B. Freisleben, "Supporting autonomic computing functionality via dynamic operating system kernel aspects," in *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2005, pp. 51–62.
- [16] S. Almajali and T. Elrad, "Coupling availability and efficiency for aspect-oriented runtime weaving systems," Proc. of the Dynamic Aspects Workshop (DAW'05) at AOSD, Mar. 2005.
- [17] D. Suvée, W. Vanderperren, and V. Jonckers, "Jasco: an aspect-oriented approach tailored for component based software development," in *Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM Press, 2003, pp. 21–29.
- [18] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng, "MDL: A language and compiler for dynamic program instrumentation," in *IEEE Conference on Parallel Architectures and Compilation Techniques (PACT)*, Nov. 1997, pp. 201–213.
- [19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2005, pp. 190–200.



Marc Ségura-Devillechaise works at the research and development department of the Credit Coopératif bank France. Before receiving his PhD the Ecole des Mines de Nantes (EMN). He graduated from the EMOOSE (European Master in Object-Oriented and Software Engineering technologies) program. His research interests ranges from software engineering to programming language implementations. He can be reached through email at msegura at emn.fr.



Jean-Marc Menaud defended his Ph.D Thesis in Computer Science at the University of Rennes 1 in January 2000 in the Solidor research group of IRISA/INRIA on the following subject: Cache cooperative system for large scale distributed information systems. He is currently an assistant professorship at the Ecole des Mines de Nantes, France, and he joined the Obasco group to do his research in September 2000.



Nicolas Lorient is a graduate student in Computer Science of the University of Nantes, France. He is currently a PhD student in the Obasco Group of the École des Mines de Nantes. His research interest are software engineering, Aspect-Oriented Programming, and On-the-fly patching. He can be reached through email at nloriant at emn.fr.



Thomas Fritz is a student of Computer Science at the Ludwig-Maximilians-University Munich and a future graduate student at the University of British Columbia. His research interests include software engineering, in particular aspect-oriented software development. His email address is fritz at cs.ubc.ca.



Rémi Douence has been an assistant professor at Ecoles des Mines de Nantes in France since 1998 following doctoral studies in computer science at Inria of Rennes and a year of postdoc research at CMU. He is interested in programming languages (semantics, analysis, compilation...) in general and in AOP in particular.



Mario Südholt has been an assistant professor at Ecole des Mines de Nantes in France since 1997 following doctoral studies in computer science at Technical University of Berlin and a year of postdoc research at Irisa/Inria in Rennes. Currently, he is an INRIA researcher on secondment from EMN. His current research focus is on the formal definition and realization of expressive approaches for Aspect-Oriented programming and support for composition based on more powerful notions of component interfaces. He has served several times on the program

committee of the main conference on AOSD.



Egon Wuchner works at the research and consultancy department called Corporate Technology, Software Engineering of Siemens AG, Germany. He has been working in the field of software architecture and relevant technologies for building distributed systems. One of his research topics are concepts, technologies and tools to improve the developmental qualities of large systems, e.g. their handling of operational requirements, their comprehensibility and maintainability. Thus, he has been taking a closer

look at Aspect-oriented Software Development for several years. His email address is egon.wuchner at siemens.com.