Dynamic Adaptation of the Squid web cache with Arachne

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

Dynamic Adaptation of the Squid Web Cache with Arachne

Marc Ségura-Devillechaise, Jean-Marc Menaud, Nicolas Loriant, Rémi Douence, and Mario Südholt, *École des Mines de Nantes*

Thomas Fritz, *University of British Columbia*

Egon Wuchner, *Siemens*

The Arachne aspect-oriented programming system lets developers modularize changes to networking software with little perceptible performance overhead.

Writing good software is often a challenge; writing adaptable software can be even more difficult. For example, open implementations must provide, at the time of application design, both a functional interface that exposes the services being offered and an adaptation interface that lets users customize their implementation.[1] However, because adaptable interfaces often increase complexity, common truisms such as "keep it simple, stupid" (KISS) and an emphasis on faster time to market encourage software designers to sacrifice adaptability.

Networking software illustrates this well. Because such software must meet high performance and availability constraints, developers often neglect design goals such as adaptability and modularity in the absence of immediate benefits. For example, you could write a Web cache modularly by composing an HTTP parsing library such as libwww and a threading library such as Posix and by using your operating system's virtual-memory mechanism to replicate Web pages on disk. So, for performance reasons, real-world Web cache implementations are large, monolithic applications usually written from scratch in C. For instance, the Squid open source Web cache[2] doesn't even manage resources using operating system services. Instead, Squid's event-based architecture communicates with the operating system and reacts directly to the availability of network cards and hard drives. (See other approaches in the "Related Work in Aspect Modeling" sidebar.)

Moreover, performance issues such as latency often require adaptations to networking software. Since the Internet's birth, latency has remained stable, at around 100 ms.[3,4] Within the Internet backbone, latency drops to 30 ms, almost reaching the theoretical limit imposed by the speed of light and the earth's circumference.[5] Web caches were a first attempt to cope with latency. Nowadays, networks require the continuous adaptation of legacy Web caches

0740-7459/06/$20.00 © 2006 IEEE

for techniques such as prefetching and software replication.

In legacy Web caches such as Squid, such adaptation interfaces are typically needed for functionalities that are applied across the legacy code—that is, functionalities whose code is scattered and tangled in the Web cache code's files and functions. Indeed, introducing an adaptation interface after implementing a large application results in crosscutting code and inherently defies modular design efforts.[6] Furthermore, anticipating adaptation needs has its limits; for example, developers can't plan for modifications required by security breaches arising from bugs.

Thus, using Squid as an example, we propose to design network software without adaptation interfaces, thus keeping the implementation as simple and efficient as possible. We enable adaptation on a per-need basis with Arachne, an aspect-oriented system we devised for legacy C applications featuring an expressive aspect language. Arachne lets us adapt Squid modularly without sacrificing performance or needing to plan adaptations a priori. Three examples illustrate this process: correcting security breaches, reducing latency via prefetching, and adding support for the Internet Content Adaptation Protocol (ICAP) to Squid.

The Arachne aspect system

Our system uses the Arachne aspect language and the Arachne weaver tool chain.

Aspect language

An AOP system's *join point model* defines the relevant basic execution events of a given base application. *Pointcuts* allow programmers to refer to all such events at which a functionality of interest is applied across, or *crosscuts*, the base application. At the execution events matched by pointcuts, *advice* can be used to modify the base application's execution. Arachne features two basic kinds of join points: C function calls and read/write accesses to global variables and their local aliases.

Arachne's advice language essentially consists of C function calls, introduced with the keyword `then`, which we can execute in addition to or instead of legacy function calls. We define the function called by advice in a regular C source code file that is compiled along with the aspect source code file containing the advice. By default, Arachne executes the advice instead of the join point. When the advice is omitted, Arachne executes the base program join point. The pointcut language allows programs to design join points within the set of all possible join points.

Pointcuts in Arachne match C constructions. To match the different types of join

Related Work in Aspect Modeling

AspectC[1] and AspectC++[2] extend C and C++, respectively, by an aspect model very similar to AspectJ's.[3] They all rely on source code transformation and thus cannot apply aspects to running C applications. Considering language expressiveness, both approaches provide support only for aspects that address single events and not sequences of events, as our sequence aspects do.

Toskana,[4] DAC++,[5] and Jasco[6] are three examples of dynamic weavers that, like Arachne, rewrite at runtime the compiled code that the processor executes. Toskana, however, is limited to operating system kernels. DAC++ supports only C++ applications, and Jasco targets Java programs. Thus, we cannot apply any of the three dynamic weavers to a legacy application such as the Squid Web cache.

Tools such as Dyninst[7] and Pin[8] provide APIs that support dynamic code patching as well as the binary rewriting of arbitrary assembly instructions at runtime. However, these tools work at an abstraction level much lower than the base program's higher-level programming language. While devising an aspect system on top of these might be feasible technically, the absence of a well-defined and complete relation between C source and compiled code remains a major problem. In addition, technical issues limit these approaches' feasibility. Rewriting code with Dyninst, for example, is difficult because the rewriting process might fail and corrupt the application. Pin, on the other side, lets us insert code before or after a binary instruction but does not let us replace one, so the implementation of around advice poses a problem.

References

1. Y. Coady and G. Kiczales, "Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code," *Proc. 2nd Int'l Conf. Aspect-Oriented Software Development*, ACM Press, 2003, pp. 50–59.
2. O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language," *Proc. 40th Int'l Conf. Technology of Object-Oriented Languages and Systems* (TOOLS Pacific 02), Australian Computer Soc., 2002, pp. 53–60; www.aspectc.org/fileadmin/publications.
3. G. Kiczales et al., "Aspect-Oriented Programming," *ECOOP 97—Object-Oriented Programming: 11th European Conf.* (ECOOP 97), LNCS 1241, Springer, 1997, pp. 220–242.
4. M. Engel and B. Freisleben, "Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects," *Proc. 4th Int'l Conf. Aspect-Oriented Software Development* (AOSD 05), ACM Press, 2005, pp. 51–62.
5. S. Almajali and T. Elrad, "Coupling Availability and Efficiency for Aspect-Oriented Runtime Weaving Systems," *Proc. Dynamic Aspects Workshop* (DAW 05) at Int'l Conf. Aspect-Oriented Software Development, ACM Press, 2005; www.iit.edu/~concur/publications.html.
6. D. Suvée, W. Vanderperren, and V. Jonckers, "Jasco: An Aspect-Oriented Approach Tailored for Component Based Software Development," *Proc. 2nd Int'l Conf. Aspect-Oriented Software Development* (AOSD 03), ACM Press, 2003, pp. 21–29.
7. J.K. Hollingsworth et al., "MDL: A Language and Compiler for Dynamic Program Instrumentation," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 97), IEEE CS Press, 1997, pp. 201–213.
8. C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. 2005 ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 05), ACM Press, 2005, pp. 190–200.

Like most legacy high-performance Web caches, Squid is designed around an event-based architecture.

points and to access information about the corresponding execution state, we use Arachne's pointcut language, which resembles AspectJ. For instance, with a pointcut of the form `call(void _ xcalloc(size t, size t))`, we can refer to the join point designating a call to the function `xcalloc`. The constructor `writeGlobal(var)` matches a write access to a global variable, while `write(var)` also matches accesses to the variable's local aliases—that is, aliases of global variables having local scope. The constructor `controlflow`, which takes a list of function call pointcuts as its argument, lets us denote sequences of nested function calls, similar to AspectJ's `cflow-construct`. We can also combine pointcut expressions, for example, using logical combinators such as `or` (`||`). Furthermore, *binder* expressions let us retrieve information about the execution state in which a join point occurs: we can apply `args` and `return` to function-call join points to access the arguments and the function call's return value, and we can use `value` with read and write accesses to retrieve the value being read or written. An `if(C)` pointcut lets us restrict matching to contexts where the C expression holds.

Sequence aspects, denoted by `seq(sts)`, consist of a list of steps, each of which associates a pointcut with (possibly empty) advice. A sequence *instance* is created each time the first step's pointcut matches a join point. Further steps are activated in a "greedy" fashion—that is, the step following the current one is activated as soon as its pointcut matches the current join point. We can repeat all but the first and last steps multiple times using the repetition operator*. A key feature of sequence aspects is that they can use any data bound in one step (with `arg`, `return`, or `value`) 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 is allocated for all the data the aspect is interested in. As the sequence aspect executes, the data is collected, and as soon as the last step is executed for a particular instance, it frees the associated memory. So, the Arachne implementation can represent sequences using a small part of the base program's execution history in the form of a bounded list of sequence instances, which gets updated each time a new step in the sequence occurs.

We can use Arachne's *weaver* tool to apply a set of aspects to C applications. While Arachne's pointcut and advice languages essentially resemble those of AspectJ, Arachne differs from AspectJ and similar systems in two ways:

- Arachne's aspect language features sequence aspects, which are useful for formulating protocol manipulations—in particular, the Web cache manipulations we present in this article.
- Arachne's weaving is dynamic; that is, we can adapt running C applications without stopping them. For example, dynamic weaving lets us introduce prefetching strategies or correct security breaches in a Web cache, thus preserving service availability.

Compiler and runtime tool chain

The Arachne *tool chain* runs on Pentium machines under the Linux operating system. It consists 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 (also known under Linux as a shared library). To ease interoperation 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 actual rewriting sites and their symbolic descriptions, Arachne rewrites binary code referenced by the aspect with hooks that point to the aspect DLLs. (A detailed description of the compilation and weaving process is available elsewhere.[7])

The dynamic weaver (shell command `weave`) applies an aspect DLL to a running process. The code `weave <pid> <aspect-library>` weaves the compiled aspect 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 haven't been removed from the base program. These assumptions are not uncommon: for example, the default compilation process of the Squid Web cache meets these expectations. In general, Arachne exploits binary-code and

```
seq(     /* first step : retrieve buffer and buffer size */
  call(void * xcalloc(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); )
```

Figure 1. An aspect preventing buffer overflow.

linking standards[8,9] that govern the execution of compiled files and do not depend on code patterns generated by specific compilers. Therefore, we can weave aspects into any code adhering to these standards. In addition, Arachne provides a deweaver (`deweave`) to unweave aspects from an application.

Writing aspects with Arachne

Like 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 must handle several concerns at once and do not clearly reflect execution flow. Thus, adaptations of Squid's behavior tend to require modifications at many places. As Squid amounts to several Mbytes of undocumented source code, such adaptations get very complex.

To assess Arachne, we consider both expressiveness and performance. We first focus on expressiveness: does Arachne enable us to implement useful adaptations of the Squid Web cache concisely and modularly? (Unless explicitly noted, we used the squid-2.5STABLE3 release as a test bed for our adaptations.) Our adaptation examples include removing a security threat through protocol modifications, reducing latency, and adding support for a complete network protocol (ICAP).

Correcting a security hole

In February 2002, the Computer Emergency Readiness Team 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`. A successful 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 how the two fields are manipulated by five functions relevant for handling ftp connections.

Although we could rewrite this patch as a collection of Arachne aspects, we can also write a sequence aspect that prevents a class of buffer overflows, including as-yet unreported ones (see figure 1). This sequence aspect defines a sequence of Squid functions (the *protocol*) that leads to a buffer overflow. First, the buffer length is retrieved at allocation time (call of `xcalloc`), then matches assignments made to that buffer. An advice, which replaces the faulty assignment, is attached to this step. 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 lets Arachne free the memory associated with the collected data.

In contrast to the Squid team's patch, which 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, because security threats are usually first reported only after 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.

Adding prefetching over HTTP to reduce latency

In the past two years, a number of Web browsers have started to download pages be-

```
/* 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);
```

Figure 2. Aspects for prefetching.

fore the end user requests them.[10] Such prefetching schemes trade network bandwidth for reducing end-user-perceived latency. As intermediaries, Web caches are better suited to prefetch pages than individual users are. However, Squid doesn't include prefetching. We've implemented Ken-ichi Chinen's and Suguru Yamaguchi's simple prefetching strategy,[11] which prefetches 10 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 upon an end-user request, but at the expense of doubling the consumed bandwidth.

The prefetching adaptation crosscuts the Squid functions that process HTTP requests. As figure 2 shows, we implemented it using three aspects that modify the behavior of the functions handling data reception. The first aspect starts prefetching when `clientBuildReply` creates an HTTP response. As `comm_write_mbuf` transmits a page, the second aspect retrieves the hyperlinks contained in that page. The third aspect then does the actual prefetching, retrieving a few pages among the detected hyperlinks. To avoid infinite loops, the different pieces of advice distinguish between pages requested by a regular client and those that were prefetched using the function `isPrefetch`, which retrieves the corresponding information from a hash table.

Instead of transforming an existing cache, Chinen and Yamaguchi designed the Kotetu Web cache from scratch to assess the benefits of their prefetching policy. The latest version of Kotetu consists of 38,762 lines of source code and offers fewer features than Squid. In comparison, our adaptation has no more than 1,059 LOC. Moreover, because prefetching trades bandwidth for latency, it's best used when network load is low and avoided otherwise. With Arachne, we can dynamically weave and unweave aspects and thus optimize the use of prefetching.

Adding ICAP support

Online advertisers want ads to change regularly. An end user consulting a site twice should see two different ads. The advertising providers also need to measure the audience that each ad reaches. Therefore, most of the time, ads are marked as not cacheable, thus preventing caches and reducing latency. Some researchers have suggested delegating ad insertion to Web caches.[12] The ICAP protocol was created to empower Web caches with content transformation. Each ICAP server is colocated with a proxy or cache. Every time the proxy or cache receives a request or a response, it forwards it to the ICAP server, which can then modify it if necessary. Processing continues by considering just the modified request or response that the ICAP server returns. In addition, the specification lets the ICAP server satisfy requests: in this case, the cache returns the ICAP server's response to the end user without further processing.

Table 1

Time spent in the Squid base program, Arachne, and the prefetching code

Size (Kbytes)	Miss cases				Hit cases			
	Client (sec)	Cache (sec)	Prefetching (μsec)	Arachne (μsec)	Client (sec)	Cache (sec)	Prefetching (μsec)	Arachne (μsec)
3.8	<0.1	<0.1	2	0.008	<0.1	<0.1	2	0.003
46	<0.1	<0.1	39	0.010	<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
1,182	0.6	0.6	924	0.141	0.5	0.5	919	0.142
3,875	1.3	1.3	16,679	0.677	1.0	1.0	1,801	0.446

Squid does not yet support the ICAP protocol. We used Arachne to turn Squid into an ICAP client—in other words, to dynamically add a new network protocol to Squid. First, because the standard strongly advises ICAP-enabled caches to advertise their ICAP ability, our aspects add an `X-ICAP` header to the HTTP requests and responses entering and leaving the cache. This enables content providers to send a *proxylet*—a piece of code—that runs on the ICAP server near the cache. The content provider can delegate some of its processing to the proxylet, including ad insertion. Our probe's second role is to load the ICAP adaptation in Squid when it receives a proxylet. All in all, the aspects composing the adaptation—amounting to 554 Kbytes—essentially modify the behavior of 15 Squid functions. Due to space constraints, we won't present the code here (to obtain it, see www.emn.fr/x-info/arachne/download.html).

Studying Arachne performance

We can't evaluate adaptability benefits at cache design time, so it's crucial that Arachne does not trade performance for adaptability. To study this issue, we used our collection of prefetching aspects to estimate the average overhead Arachne introduces into Squid. Measuring this required a significant modification of Squid. We've chosen to use Squid augmented with our prefetching adaptation. We built a profiling version of Arachne that tracked the time it spent in Arachne code. We compared this duration to

- the time spent in the adaptation (that is, prefetching) code,
- the time spent in Squid code to serve different Web pages, and
- the time the user needed to fetch the page from the cache.

Table 1 summarizes our results. The columns represent

- *Size*: size of the downloaded Web page
- *Client*: time needed to fetch the page
- *Cache*: time Squid needed to handle the query as reported by its own timing mechanism in the log files
- *Prefetching*: time spent running the adaptation code
- *Arachne*: time spent in the Arachne infrastructure

The time spent in prefetching code varies slightly: we used publicly available pages containing different amounts of links. In *miss* cases, the requested page has not yet been replicated in the cache. In *hit* cases, the requested page is already available in the cache. All values are averaged over 200 runs. For this particular adaptation, the time spent in Arachne never exceeded one-thousandth of the time needed to run the adaptation code, and the time needed to run the adaptation code was several orders of magnitude smaller than the time Squid required to serve a page.

We also compared the performance of Squid adapted by manually modifying its C

Table 2

PolyMix-4 results for two peak phases

	Phase 1			Phase 2		
	Arachne	Manual	Difference (%)*	Arachne	Manual	Difference (%)*
Throughput (req/sec)	5.59	5.59		5.58	5.59	
Average response time (ms)	1131.42	1146.07	+1.2	1085.31	1074.55	–1.0
Response time for a miss (ms)	2533.50	2539.52	+0.2	2528.35	2525.34	+1.8
Response time for a hit (ms)	28.96	28.76	–0.6	30.62	31.84	+3.8
Hit ratio	59.76	59.35	–0.6	61.77	62.22	+0.7
Errors	0.51	0.50	–1.9	0.34	0.34	0.0

*Negative values in difference columns indicate that the approach with Arachne was faster than the manual one.

source code to the performance of Squid adapted with Arachne. We modified the Squid source code to introduce the same prefetching strategy that our Arachne-based prefetching adaptation uses. We then benchmarked the two caches' performance with Web Polygraph.[13] After filling the cache, the PolyMix-4 workload mimicked a one-day simulation including the two request-rate peaks typically observed in production environments. Filling 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 a hundred to a thousand times slower than fetching a page that the cache has replicated locally. We performed 10 simulations using our prefetching-enabled Squid versions. The two peak phases are the most interesting cases because they stress the cache with a high throughput.

Table 2 summarizes the average results for these two phases, including the request rate and throughput from the client side. Due to the effect of prefetching, the server-side request rate was close to 43 requests per second. No significant differences exist between the manual and Arachne-based prefetching caches: 1 percent on average versus about 1.5 percent for the average variation between two simulations with the same cache version. The *miss* times (the time needed to deliver a document when it's not cached) as well as the *hit* times (the time required to deliver a document that is present in the cache) and the response times are all very similar for the two cache variants. This simulation showed no perceptible performance difference between a static prefetching integration achieved by manual source code modification and the dynamic prefetching integration Arachne performs.

Because networking developers can't assess the benefits of designing an adaptable implementation at design time, they often sacrifice adaptability for performance and simplicity. The Squid Web cache is no exception. But many reasons, ranging from the need to cope with security threats to the necessity of dealing with modifications of existing protocols, call for adaptation. Such adaptation requires unanticipated modifications of the source code and frequently crosscut the implementation.

This problem is not specific to networking software or to C. For example, we have extended Arachne to support C++ in order to adapt image generation algorithms in Siemens' medical scanners.[13] Despite our efforts, designing adaptation interfaces after program implementation is likely to continue to require significant language extensions. We plan to extend the stateful aspect-oriented sequence-based mechanism we proposed in this article to let pointcuts refer to members of C structures.

References

1. G. Kiczales, "Beyond the Black Box: Open Implementation," *IEEE Software*, vol. 13, no. 1, 1996, pp. 8–11.
2. D. Wessels, *Squid: The Definitive Guide*, O'Reilly and Assoc., 2004.
3. D.L. Mills, *Internet Delay Experiments*, Network Working Group RFC 889, Dec. 1983; www.rfc-archive.org/getrfc.php?rfc=889.

4. J. Pointek et al., "Netdyn Revisited: A Replicated Study of Network Dynamics," *Computer Networks and ISDN Systems*, vol. 29, no. 7, 1997, pp. 831–840.

5. B.-Y. Choi et al., "Analysis of Point-to-Point Packet Delay in an Operational Network," *Proc. 23rd Ann. Joint Conf. IEEE Computer and Comm. Societies* (INFOCOM 2004), IEEE Press, 2004; www.ieee-infocom.org/2004/Papers/37_4.pdf.

6. G. Kiczales et al., "Aspect-Oriented Programming," *ECOOP 97—Object-Oriented Programming: 11th European Conf.* (ECOOP 97), LNCS 1241, Springer, 1997, pp. 220–242.

7. R. Douence et al., "An Expressive Aspect Language for System Applications with Arachne," *Proc. 4th Int'l Conf. Aspect-Oriented Software Development*, ACM Press, 2005.

8. U.S.L. System Unix, *System V Application Binary Interface Intel 386 Architecture Processor Supplement*, 4th ed., Prentice Hall, 1994; www.caldera.com/developers/devspecs/abi386-4.pdf.

9. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, v. 1.2, Tool Interface Standards Committee, May 1995; www.cs.princeton.edu/courses/archive/fall05/cos217/reading/elf.pdf.

10. D. Fisher and G. Saksena, "Link Prefetching in Mozilla: A Server-Driven Approach," *Proc. 8th Int'l Workshop Web Content Caching and Distribution* (IWCW8), Kluwer Academic, 2004, pp. 283–292; http://2003.iwcw.org/papers/fisher.pdf.

11. K.-I. Chinen and S. Yamaguchi, "An Interactive Prefetching Proxy Server for Improvement of WWW Latency," *Proc. 7th Ann. Conf. Internet Soc.* (INET 97), Internet Soc., 1997; www.isoc.org/inet97/proceedings/A1/A1-3.HTM.

12. A. Gupta and G. Baehr, "Ad Insertion at Proxies to Improve Cache Hit Rates," *Proc. 4th Int'l Web Caching Workshop*, 1999; www.ircache.net/Cache/Workshop99/Papers/gupta-final.ps.gz.

13. A. Rousskov and D. Wessels, "High-Performance Benchmarking with Web Polygraph," *Software Practice and Experience*, vol. 34, no. 2, 2004, pp. 187–211.

14. T. Fritz et al., "Automating Adaptive Image Generation for Medical Devices Using Aspect-Oriented Programming," *Proc. 10th IEEE Int'l Conf. Emerging Technologies and Factory Automation* (ETFA 05), IEEE Press, 2005.

About the Authors

Marc Ségura-Devillechaise is a project leader in the research and development department at the Credit Cooperatif Bank. His research interests range from software engineering to programming language implementations. He received his PhD in computer science and a European master's degree in object-oriented and software engineering technologies from Université de Nantes. Contact him at École des Mines de Nantes-INRIA, LINA, 4, rue Alfred Kastler, 44307 Nantes Cedex 3, France; msegura501@yahoo.fr.

Jean-Marc Menaud is an assistant professor of computer science at the École des Mines de Nantes. His research interests include cache cooperative systems for large-scale distributed information systems. He received his PhD in computer science from the University of Rennes. Contact him at École des Mines de Nantes-INRIA, LINA, 4, rue Alfred Kastler, 44307 Nantes Cedex 3, France; jmenaud@emn.fr.

Nicolas Loriant is a PhD student in the Obasco Group of the École des Mines de Nantes. His research focuses on software engineering, aspect-oriented programming, and on-the-fly patching. He received his degree in computer science from Université de Nantes. Contact him at École des Mines de Nantes-INRIA, LINA, 4, rue Alfred Kastler, 44307 Nantes Cedex 3, France; nloriant@emn.fr.

Thomas Fritz is a graduate student in computer science at the University of British Columbia. He received a Diploma degree in computer science from the Ludwig Maximilians University. His research interests include software engineering, and aspect-oriented software development in particular. Contact him at the Software Practices Lab, Univ. of British Columbia, 201-2366 Main Mall, Vancouver, BC V6T 1Z4, Canada; fritz@cs.ubc.ca.

Rémi Douence is an assistant professor in computer science at Écoles des Mines de Nantes. His research focuses on programming languages in general and in AOP in particular. He received his PhD in computer science from Inria of Rennes and did a year of postdoctoral research at Carnegie Mellon University. Contact him at École des Mines de Nantes-INRIA, LINA, 4, rue Alfred Kastler, 44307 Nantes Cedex 3, France; douence@emn.fr.

Mario Südholt is a researcher in computer science at Institut National de Récherche en Informatique et en Automatique, on sabbatical from École des Mines de Nantes. His current research focuses 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 received his PhD in computer science from the Technical University of Berlin and did postdoctoral research at IRISA/INRIA. Contact him at École des Mines de Nantes-INRIA, LINA, 4, rue Alfred Kastler, 44307 Nantes Cedex 3, France; sudholt@emn.fr.

Egon Wuchner is a software engineering researcher at Siemens Corporate Technology R&D division. His research interests include the concepts, technologies, and tools needed to improve large systems' comprehensibility, maintainability, and handling of operational requirements, and thus aspect-oriented software development. Contact him at Corporate Technology, SE2, Siemens AG, Otto-Hahn-Ring 6, 81739 München, Germany; egon.wuchner@siemens.com.