An expressive aspect language for system applications with Arachne

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

An Expressive Aspect Language for System Applications with Arachne

Rémi Douence[1,⋆], Thomas Fritz[2,⋆⋆], Nicolas Loriant[1], Jean-Marc Menaud[1], Marc Ségura-Devillechaise[1], and Mario Südholt[1,⋆]

[1] OBASCO project,
École des Mines de Nantes - INRIA, LINA,
4, rue Alfred Kastler,
44307 Nantes Cedex 3, France
{douence, nloriant, jmenaud, msegura, sudholt}@emn.fr
[2] Gruppe PST,
Institut für Informatik,
Ludwig-Maximilians-Universität München,
Oettingenstraße 67,
80538 München, Germany
fritz@informatik.uni-muenchen.de

Abstract. Security, networking and prefetching are typical examples of concerns which crosscut system-level C applications. While a careful design can help to address these concerns, they frequently become an issue at runtime, especially if avoiding server downtime is important. Vulnerabilities caused by buffer overflows and double-free bugs are frequently discovered after deployment, thus opening critical breaches in running applications. Performance issues also often arise at run time: in the case of Web caches, e.g., a prefetching strategy may be required to increase performance. Aspect-oriented programming is an appealing solution to solve these issues. However, none of the current dynamic aspect systems is expressive and efficient enough to support them properly in the context of C applications. Arachne is a new aspect system specifically designed to address these issues. Its aspect language allows aspects to be expressed concisely using a sequence construct for quantification over function calls and accesses through variable aliases. Arachne enables aspects to be woven "on the fly" in running legacy applications. We show how these abilities can be used to prevent security breaches, to modularize the replacement of network protocols by more efficient ones, and to introduce prefetching in Web caches. We present two formal semantics for Arachne: one which defines in abstract terms the main properties of the sequence construct, and a second one which enables reasoning about the actual implementation. Following a detailed presentation of Arachne's implementation, we give performance evaluations showing that Arachne is fast enough to extend high-performance applications, such as the Squid Web cache.

⋆ This work has been supported by AOSD-Europe (http://www.aosd-europe.net).
⋆⋆ Part of this work was done during the author's stay at École des Mines de Nantes.

A. Rashid and M. Aksit (Eds.): Transactions on AOSD I, LNCS 3880, pp. 174–213, 2006.

1 Introduction

Real-world applications are typically made of a number of different concerns. System-level C applications are no exception: security considerations, network concerns, caching and prefetching concerns are usually scattered in the entire program code. Furthermore, there is a strong need to isolate and manipulate these concerns at run time, especially in server environments whose downtime must be minimal. Security breaches such as double-free bugs and buffer overflows might be discovered after server deployment. Hardware resources might turn out to be undersized calling, for instance, for use of more appropriate network protocols or for the inclusion of prefetching strategies within Web caches. The Web cache Squid [1] is a typical illustration of this situation. First, we have found that several such concerns are scattered over large portions of the code of Squid. Second, such a Web cache should not be stopped in order to avoid performance loss by keeping caches filled continuously. Similarly, a buffer overflow should be fixed without incurring server downtime.

Potentially aspect-oriented programming (AOP) [2] should allow one to properly modularize and manipulate crosscutting concerns such as those we have identified for the Squid Web cache. Furthermore, Squid is designed to be as efficient as possible and therefore exploits any suitable operating system and hardware particularity. Its code base is therefore difficult to understand and manipulate, thus hinting at the use of specialized aspect systems instead of traditional means for modularization. However, these concerns exhibit three characteristics which make difficult the application of basic aspect technology. First, any of these concerns expose intricate relationships between execution points: network protocols, e.g., are most concisely expressed in terms of sequences of execution points, not individual ones. Second, as motivated above, the concerns need to be manipulated "on the fly" once the application is running. A dynamic aspect weaver is therefore needed. Finally, their lack of modularization at design time typically results from performance considerations. Use of aspect-oriented (AO) techniques in this context must only degrade efficiency to a very small extent.

To our knowledge, none of the current aspect systems for C is suitable for the modularization of such concerns. In particular, no existing aspect systems meets the three requirements introduced above, e.g., dynamic weavers often trade efficiency for expressivity. This paper summarizes our attempt to treat such concerns as aspects using the Arachne system. The core to our solution is a new expressive aspect language providing a sequence construct which allows us to quantify over function call events and access to local aliases of global variables. A main contribution of this paper is to show how sequences allow us to facilitate nontrivial evolution tasks of legacy systems software. Technically, we show how they support the proper modularization of the four concerns introduced above. Its implementation is based on binary code rewriting techniques and allows aspects to be woven dynamically in running C applications (which uses implementation techniques quite different from load-time or dynamic weaving in, e.g., Java-based aspect systems).

The paper is structured as follows. Section 2 presents the motivating concerns we identified within Squid. Section 3 shows how to modularize these concerns as aspects and presents the Arachne aspect language. The language is defined in Sect. 3, where two formal semantics for this language are presented: an abstract one defining the main properties of the language and an implementation-level one that allows us to reason about the code executed by the Arachne tool. Section 5 describes Arachne's implementation. Section 6 assesses the performance of our implementation. In Sect. 7, we discuss related work, and we conclude in Sect. 8.

2 Motivation

Legacy C applications involve multiple crosscutting concerns. Many of them remain challenging, both in terms of expressiveness required to handle them properly in an AO language and in terms of constraints posed on the weaver. In this section we discuss four such concerns in C applications: memory management problems caused by double-free bugs, buffer overflows, switching network protocols and Web cache prefetching. The security threats posed by double-free bugs and buffer overflows are typically scattered over the entire application. Since guarding all buffers against overflows or monitoring memory manipulations might considerably decrease performance, administrators are often left with no other option than accepting the trade-offs between security and performance chosen at design time of an application. Likewise, switching network protocols is a real problem for administrators facing bandwidth problems. Prefetching is another well-known crosscutting concern that traditionally require similar trade-offs [3, 4]. Since prefetching aims at increasing performance, prefetching aspects make only sense with an efficient weaver. Yet, it is still difficult to modularize these four concerns in today's AO languages. In this section, we first describe the contexts in which the different concerns arise before giving evidence of their crosscutting nature and finally motivating the lack of appropriate means of expression in current AO languages.

2.1 Double-Free Bugs

Unix systems introduced the `brk` system call, allowing programs to dynamically resize the heap. Later on, the standard C library has provided the `malloc` interface that acts as a layer between applications and the system. It allocates large chunks of memory through `brk` and fragments these chunks for the application, thus providing a more efficient and finer-grained interface for dynamic memory manipulation.

For performance reasons the GNU C library performs no sanity check on use of the `malloc` interface: freeing a nonallocated memory chunk leads to an implementation-dependant behavior, most frequently a segmentation fault. This has widely been exploited by hackers to build denial of service attacks [5]. In order to deal both with performance and fragmentation issues, the GNU C library implementation stores information such as the list of free chunks, the chunk size and other management information within the heap itself. If an application tries

to free a nonallocated memory chunk, hackers can exploit the GNU C memory layout to take control of the application by corrupting its memory [6]. To protect against these so-called double-free bugs (which occur frequently because of erroneously freeing a memory location twice), a safe implementation of the `malloc` interface, which can be selected at load time, is provided by the GNU C library. Nevertheless, this safe implementation turns out to be very inefficient and is rarely if ever used. Hence, administrators discovering that an application contains a double-free bug cannot ensure security without deploying a bug-free version of the application. This requires the application to be stopped at the cost of potentially trashing the work in progress.

Despite the fact that Squid implements its own heap manipulation API,[1] it has recently been proven to be vulnerable to double-free bugs [7]. In Squid, memory allocation is a crosscutting concern: 71% of the `.c` files, which constitute its source code, contain direct references to the heap manipulation API.

In order to ensure continuous servicing without hurting performance, sanity checks on double-free calls should be limited to untrusted code (i.e., external libraries) or to periods when the environment is known to be hostile. Adding sanity checks to memory manipulation code may affect an entire application source code as dynamic memory allocation is highly common in string and buffer manipulations. In this case, an AO system is likely to permit users to improve on the lengthy and error-prone process of manually adding sanity checks. Furthermore, it should be helpful to address the security versus performance trade-off.

2.2 Buffer Overflows

In C, the size of an array is fixed at allocation time. According to ISO and ANSI standards [8], an invalid array access, i.e., an access out of the bounds of the array, does not result in an immediate error but leads to an implementation-dependent behavior. These vulnerabilities are increasingly exploited by computer worms such as CodeRed [9, 10], Slammer [11, 12] and Blaster [13, 14], and cause billions of dollars worth of damage [15]. Today, about 50% of vulnerabilities reported by CERT [16] arise from buffer overflows, and buffer-overflow attacks present the most common security attacks on software systems [17, 18].

A typical buffer-overflow attack tries to modify the memory by injecting code and altering the control flow so that the attacker gains control of the machine [19]. The most common buffer-overflow attack, the so-called stack smashing, overwrites the return address of a function on the stack with an address pointing to previously inserted malicious code (Fig. 1). This overwriting of the return address is possible as the program does not check if input exceeds the bounds of the buffer, and thus the attacker can overwrite code adjacent to the buffer. Once the function returns, the control is handed to the malicious code, and the attacker may get control over the machine [20]. A simple echo server in C containing such a buffer-overflow vulnerability is shown in Fig. 2. The code lacks

[1] Squid can also be configured to use its own heap manipulation routines on top of GNU `malloc`.

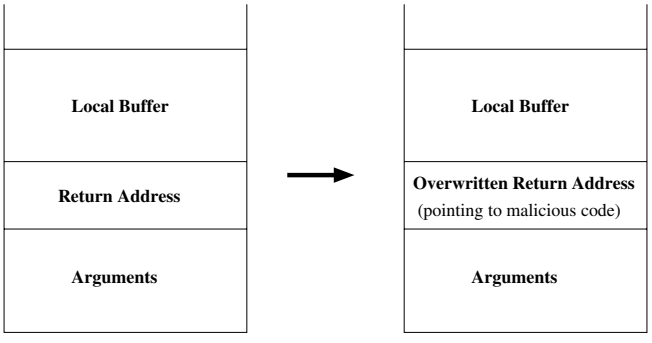

Fig. 1. Buffer-overflow attack overwriting the return address of a function

```
void echo() {
  char* in = malloc(255);
  gets(in);              /*read user input*/
  printf("%is\n",in);    /*display it*/
  free(in);
}
```

Fig. 2. Echo server in C with buffer-overflow vulnerability

a test whether the user input exceeds the size of the array, and an attacker could easily exploit this vulnerability as described.

Therefore, it is crucial to ensure every access to a buffer to be in its bounds. But bound-checking is error-prone and easily forgotten, and it is infeasible to detect all buffer-overflow vulnerabilities by statically analyzing code [20]. Several buffer overflow detectors have thus been proposed. Some of these approaches do not protect against all attacks, like StackGuard [21]. This approach is based on placing a dummy value between the stack data and the return address and then checking whether it has been altered or not. Thus, it just detects attacks overwriting everything along the stack. Bound-checkers, on the other hand, detect all buffer-overflow attacks as they check all buffer accesses. But approaches like Cyclone [22] or CCured [23], which are based on bound-checking, imply changes to the code. Cyclone is a "safe dialect of C" [22]. To prevent safety violations, the approach requires a subset of the C language to be used, e.g., by restricting pointer arithmetic, and the corresponding compiler performs static analysis and inserts run-time checks. CCured is a program transformation system that statically analyzes the program by classifying pointers and, depending on the classification, also adds run-time checks. Compilers have also been proposed that enforce proper array access by bound-checking [15, 24][2] without requiring code changes. But even the most efficient of these compilers, CRED [15], incur an overhead of up to 130%. Moreover, most frequently used C compilers, like the `gcc` compiler, do not support bound-checking.

[2] `http://sourceforge.net/projects/boundschecking/`

Also, with respect to performance, most approaches are too generic. They check every buffer access, even if the environment is not hostile and there is no vulnerability. However, as bound-checking is expensive, it should only run on buffer-overflow vulnerabilities [25].

Nowadays, administrators discovering a buffer-overflow vulnerability in a running application are mostly left with no other option than stopping the application and restarting a bug-free version, as done in Squid [26]. However, this technique does not conserve the continuous service property required by applications like the Squid Web cache. Furthermore, by stopping the application, the administrator has no means to know whether and how the vulnerability has been exploited, and thus this technique entails an important loss of information.

Bound-checking code tends to crosscut the entire application. In Squid, bound-checking code can be found in any of the 104 `.c` files of its source code. Of the 57,635 lines composing these `.c` files, at least 485 relate to bound-checking.

This problem fails to be handled properly in current aspect languages since they lack the ability to trigger advices upon access made through the alias of a variable. Furthermore, many AO systems offer only static weaving capabilities, preventing the administrator from choosing the trade-off between security and performance that suits his needs.

2.3 TCP to UDP Protocol

The Hypertext Transfer Protocol (HTTP) [27] is the primary method of the World Wide Web to transfer information over the Internet. The most frequently used communication protocol underlying HTTP is the Transmission Control Protocol (TCP) [28]. TCP is a connection-oriented protocol ensuring reliable communication by explicitly setting up and tearing down connections. While TCP is used as the underlying transport protocol of HTTP, it is not well-suited for short-lived connections exchanging only little data. However, short interactions comprise a significant amount of Web traffic. According to a study conducted on the soccer World Cup Web site of 1998 [29], the average request size is about 4 KB, and there are results that 40% of the Web traffic can even fit into a single datagram of 1500 bytes, making up the size of a maximum transfer unit (MTU) of Ethernet [30]. Thus, the cost of a Web interaction is dominated by data exchanged for control purposes of the TCP connection rather than the actual requested data. Furthermore, HTTP 1.1 has introduced persistent connections, allowing a client to retrieve multiple pages from the same server through the same TCP connection. However, the number of simultaneous TCP connections is limited by operating systems, and thus servers have a strong incentive to close HTTP connections as soon as possible.

Therefore, as also supported in [30, 31, 32], it seems beneficial to use the User Datagram Protocol (UDP) [33]. UDP incurs much less overhead for connection establishment than TCP as the underlying transport protocol of HTTP for short-lived connections and thereby reduces the overhead induced by TCP.

In spite of the corresponding potential performance gains, the existence of a large number of legacy Web applications and the corresponding adaptation costs

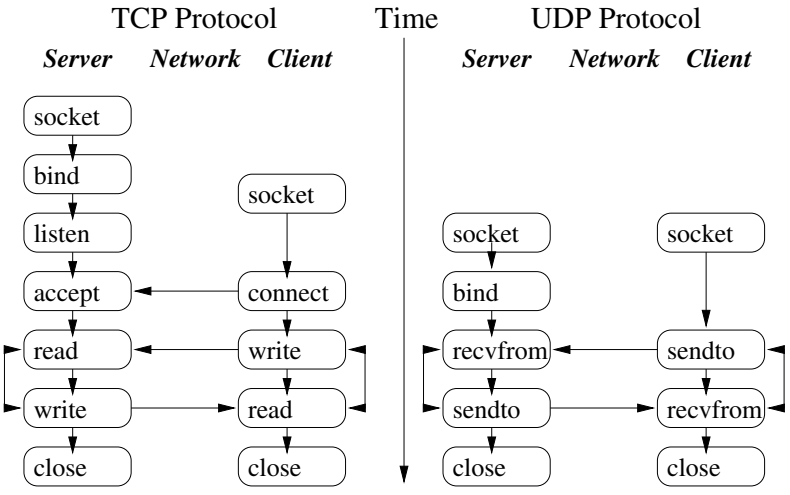

Fig. 3. Typical usage of the TCP and UDP APIs

have hindered widespread adoption of this solution. In particular, a complete redesign of legacy applications is typically not reasonable. Besides the corresponding development costs, deployment of the modified applications is problematic. Existing approaches to application deployment require stopping the legacy Web application to switch the protocol. This, however, does not satisfy the continuous servicing property inherent in such applications and, for example, in the case of an e-commerce Web server, causes a direct loss of money. Therefore, one may swap the application between different machines to avoid shutting down the service, but this requires redundant servers, which are often not affordable for small companies. For wide acceptance, a HTTP dialect using UDP as transport protocol should thus be deployable on demand at run time.

In addition, replacing TCP by UDP is relatively difficult in an application. The choice of a transport protocol is usually based on standards believed to be everlasting and is made at an early design stage. Hence, no particular effort is made to localize this design decision in a single piece of code. For example, despite a modularization effort, the TCP API provided by the operating system is used directly in 7 of the 104 `.c` source files of the Squid Web cache.

As shown in Fig. 3, the TCP API is built around a set of C functions to be invoked sequentially by the application [34]. In a properly written program, TCP functions are first used to establish the connection (typically with `socket`, `connect`, `bind` and `listen`), exchange data through the connection (typically with `read` and `write`) and then close it (typically `close`). Similarly, UDP applications first direct the operating system to dedicate the appropriate resources to exchange data (typically with `socket` and `bind`), then exchange data through these resources (typically with `sendto` and `recvfrom`) before releasing them (typically with `close`). Hence, the problem is not only difficult because TCP-related

function invocations are scattered but also because the relative order of each invocation is important in order to map it onto the appropriate UDP function. Furthermore, there can be several connections at the same time, i.e., several clients that connect with one server, and each connection can be in a different state.

2.4 From Fetching to Prefetching

Operations like retrieving a file on a local disk or over the Web can be sped up if the underlying software anticipates user requests and fetches documents in advance of explicit requests. Such prefetching schemes differ from one another by how they predict future user requests. These "oracles" actually prevent a clean encapsulation of prefetching in a single module communicating with the rest of the application through well-defined interfaces since predictions are based on information meant to be private to other modules. In addition, it is obvious that there is no universally perfect oracle [35]. A statically linked prefetching module is therefore inappropriate; instead, prefetching modules along with the necessary oracles should be loaded and unloaded on the fly. Because of their crosscutting nature, prefetching modules including such oracles are better written with aspects, as motivated by Coady et al. for file prefetching in the FreeBSD OS [3] and our previous work considering the Squid Web cache [4].

Despite potential performance improvements, prefetching also increases resource consumption (e.g., network prefetching consumes local storage and band width). When the need for such resources is too high, prefetching computation competes for them against regular user requests and slows down their treatment instead of speeding it up. In such cases, prefetching should therefore be temporarily disabled. Squid, for instance, essentially manages file descriptors, a resource only available in a limited quantity. A file descriptor is used by the underlying operating system and applications to describe a network connection or a file on the disk. Squid's file descriptor management is based on a global variable that tracks the number of file descriptors currently in use. By comparing its value with the maximum number of file descriptors allowed by the operating system, it is possible to evaluate whether prefetching should be disabled or activated.

Using current AO technology, enabling/disabling of prefetching depending on the number of open file descriptors would be handled within advice by explicitly managing a corresponding state and triggering the corresponding actions. This is bad practice because it impedes both readability and maintainability. A mechanism is needed within the aspect language to restrict advice execution at times where resource usage is too high.

3 An Expressive Aspect Language for System Programming in C

While AOP is an obvious choice to tackle the crosscutting concerns introduced above, none of the existing AO systems provides explicit support for some of their essential elements, in particular, references to aliases which are local to a function, and joinpoint sequences for protocols.

In this section we introduce a new aspect language for system programming in C that allows such crosscutting concerns to be expressed concisely. In order to make this point, we first revisit the examples by concisely "aspectizing" them using our language. (Note that our aspect language is expressive in the sense that it enables the concise definition of certain types of aspects, especially compared to other tools for system-level manipulations, but it is not necessarily more expressive than existing approaches in a language-theoretic sense.) We then define the joinpoint model underlying our language precisely, followed by the definition of its syntax and informal semantics. Its formal semantics is the subject of the following section.

3.1 Example Crosscutting Concerns Revisited

We now revisit the concerns discussed in Sect. 2 in order to show our language in action and to give evidence that it allows such concerns to be concisely modularized. Our motivating examples are reordered following increasing complexity of the language constructs involved.

Double-Free Bugs. The aspect shown in Fig. 4 detects double-free bugs. It uses two sets, `addMalloc` and `addFree`, which are initially empty to collect addresses that have been allocated and freed as exemplified by the first advice. The second advice checks whether these sets are consistent when `free(buffer)` is to be called. First, when the current address has already been freed previously, the advice terminates the execution of the application. Second, when `buffer` does not belong to `addMalloc`, either the aspect has been dynamically woven *after* the corresponding call to `malloc` that returned `buffer` and this call to `free(buffer)`

```
void* checkMalloc(size_t size) {
        void* buffer = malloc(size);
        addMalloc = addMalloc ∪ {buffer};
        addFree = addFree \ {buffer};
        return buffer;
}

void checkFree(void* buffer) {
        if (buffer ∈ addFree) exit(error);
        else if (buffer ∉ addMalloc) warning();

        free(buffer);
        addFree = addFree ∪ {buffer};
        addMalloc = addMalloc \ {buffer};
}

call(void* malloc(size_t)) && args(size) then checkMalloc(size);

call(void free(void*)) && args(buffer) then checkFree(buffer);
```

Fig. 4. An aspect for detecting double-free bugs

is correct, or the aspect has been dynamically woven *after* the previous call to `free(buffer)` and the current call is a bug. These two cases cannot be distinguished, so the advice only prints a warning. The memory is freed, and the two sets of addresses are maintained. Note that, if the user does not care about warnings, the aspect can be simplified by suppressing the set `addMalloc`.

TCP to UDP Protocol. The aspect shown in Fig. 5 translates transport protocols from TCP to UDP. A protocol defines a sequence of function calls, so the top-level operator of this aspect is `seq`. The sequence aspect syntactically consists of a list of pairs of pointcut and advice, with the pairs being separated by ";". In the example, the TCP protocol starts with a call to socket() with three arguments that are bound to family, type and protocol and compared to constants (AF_INET, SOCK_STREAM and 0) in the if-expression. When such a call is matched for which the comparisons in the if-expression also evaluates to true, the second parameter is replaced by SOCK_DGRAM as required by the UDP protocol. The result of this transformed call, the file descriptor, is bound to fd by `return`(fd). Then the next call to connect() is matched for which the same file descriptor has to be the first parameter (achieved by binding it to fd1 and comparing it to fd in an if-expression). In this case, the values of the other parameters are bound to arguments address and length, and the original call is replaced by returnZero(), which simulates a successful connection establishment by returning zero and doing nothing else. Indeed, there is no connect step in the UDP protocol. After that, calls to read() and write() (using the "or" on aspects: ||) on the same file descriptor fd are translated to UDP recvfrom() and sendto(), respectively. Note that sequences of such access are potentially repeatedly translated (due to use of the repetition operator "$*$"). Finally, a call to close() on the same file descriptor fd terminates the TCP protocol as well as

```
seq( call(int socket(int, int, int)) && args(family, type, protocol)
        && if((family == AF_INET) && (type == SOCK_STREAM)
                && (protocol == 0))
        && return(fd)
       then socket(AF_INET, SOCK_DGRAM, 0);

     call(int connect(int, struct socketaddr*, socklen_t))
       && args(fd1, address, length)  && if(fd1 == fd)
       then returnZero();    // where int returnZero() { return 0; }

     ( call(size_t read(int, void*, size_t))  && args(fd2, readBuffer, readLength)
         && if(fd2 == fd)
         then recvfrom(fd, readBuffer, readLength, 0, address, length);
       || call(size_t write(int, void*, size_t))
           && args(fd3, writeBuffer, writeLength)  && if(fd3 == fd)
           then sendto(fd, writeBuffer, writeLength, 0, address, length); ) *

     call(int close(int)) && args(fd4) && if(fd4 == fd) ; )
```

Fig. 5. An aspect for switching transport protocols, from TCP to UDP

```
seq( call(void * malloc(size_t))
       && args(allocatedSize) && return(buffer) ;

     write(buffer) && size(writtenSize)
       && if(writtenSize > allocatedSize)
       then reportOverflow(); *

     call(void free(void*)) && args(b1) && if(b1 == buffer) ; )
```

Fig. 6. An aspect for detecting buffer overflow

```
require Number_Of_Fd as int*;
require Squid_MaxFd as int*;

controlflow(call(void clientSendMoreData(void*, char*, size_t)),
            call(HttpReply * clientBuildReply(clientHttpRequest*, char*, size_t))
            && args( request, buffer, bufferSize ))
   then startPrefetching(request, buffer, bufferSize);
&& until(writeGlobal(int * Number_Of_Fd) && if((*Number_Of_Fd) *
100/(*Squid_MaxFd) ≥ 75) ; )

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);

call(void clientWriteComplete(int, char*, size_t, int, void*))
   && args(fd, buf, size, error, data) && if(! isPrefetch(handler))
   then retrieveHyperlinks(fd, buf, size, error, data);
```

Fig. 7. An aspect for prefetching

the UDP protocol and thus is not modified (i.e., there is no `then` clause). This last step is required to free the variables used in the sequence (here, fd, address and length). Indeed, this aspect can use numerous (instances of these) variables when it deals with interleaved sequences, as each call to socket() creates a new instance of the sequence.

Buffer Overflows. The aspect shown in Fig. 6 detects buffer overflows. The corresponding sequence starts when the function malloc() returns the buffer address that is then bound to the buffer. Then, each time this address is accessed (through a global variable or a local alias) the size of the data to be written is compared with the size of the initially allocated memory. If the former exceeds the latter, an overflow is indicated. The sequence ends when the memory is deallocated using free().

From Fetching to Prefetching. The aspect in Fig. 7 introduces prefetching in a Web cache. The first `controlflow` phrase initializes prefetching when an

HTTP response is built (clientBuildReply()) within the control flow of a client request (clientSendMoreData()). The `until` clause stops prefetching when the number of connection becomes too large, a situation where prefetching would effectively degrade performance. The second `controlflow` phrase analyzes hyperlinks in a page being transmitted (i.e., when comm_write_mbuf() is called within the control flow of clientSendMoreData()). Finally, the last call phrase prefetches hyperlinks analyzed by the second aspect. It does so by replacing the method call to clientWriteComplete() with retrieveHyperlinks(). Finally, note that the two `require` clauses at the top of the aspect declare the types of the global variables of the base program used in the aspects.

3.2 Joinpoints

A joinpoint model defines the points in the execution of the base program to which pointcuts may refer. In our case, joinpoints are defined by JP in the grammar shown in Fig. 8. A joinpoint is either:

- A call of a function `callJP`(v$_1$ funId($\overrightarrow{v_2}$)) with function name *funId*, return value v_1 and a vector of arguments $\overrightarrow{v_2}$.
- A read access that comes in two variants: `readGlobalJP`(varId, v) denotes reading a global variable with name *varId* holding the value v; `readJP`(@, v) denotes reading a global variable or a local alias with address @ holding the value v.
- Write access, which also comes in two variants: `writeGlobalJP`(varId, v, size) denotes assignment to a global variable with name *varId* of the value v of size *size*. `writeJP`(@, v, size) denotes assignment to a global variable or a local alias with address @ of the value v of size *size*.
- A cflow expression `controlflowJP`($\overrightarrow{funId}$, c), where $\overrightarrow{funId} = [funId_1, .., funId_n]$ is a stack of function names, and c (either a function call or an

$$
\begin{array}{lll}
JP & ::= & \texttt{callJP}(\text{val funId}(\overrightarrow{val})) \\
 & | & \texttt{readGlobalJP}(\text{varId, val}) \\
 & | & \texttt{readJP}(@, \text{val}) \\
 & | & \texttt{writeGlobalJP}(\text{varId, val, size}) \\
 & | & \texttt{writeJP}(@, \text{val, size}) \\
 & | & \texttt{controlflowJP}(\overrightarrow{funId}, \text{cfEnd}) \\
 & | & \texttt{controlflowstarJP}(\overrightarrow{funId}, \text{cfEnd}) \\
 & & \\
cfEnd & ::= & \texttt{callJP}(\text{val funId}(\overrightarrow{val})) \\
 & | & \texttt{readGlobalJP}(\text{varId, val}) \\
 & | & \texttt{writeGlobalJP}(\text{varId, val, size}) \\
 & & \\
val & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \quad \texttt{// int} \\
 & | & \texttt{@0} \mid \texttt{@1} \mid \texttt{@2} \mid \ldots \quad \texttt{// int*} \\
 & | & \ldots \quad \texttt{// values of other C types}
\end{array}
$$

Fig. 8. Joinpoint model

access to a global variable) occurs within the body of function $funId_n$. Such a joinpoint requires a call to $funId_{i+1}$ within the body of $funId_i$.
- A cflow expression `controlflowstarJP`($\overrightarrow{funId}$, c), where $\overrightarrow{funId} = [funId_1, .., funId_n]$ is a *partial* stack of function names, and c (either a function call or an access to a global variable) occurs within the *control flow* of function $funId_n$. Such a joinpoint requires a call to $funId_{i+1}$ within the control flow of (i.e., not necessarily in the body of) $funId_i$. Therefore, in contrast to the preceding cflow expression, no direct nesting is required, but the functions and the final execution point c may be nested at arbitrary depth within the preceding function.

Two features of this joinpoint model may be surprising at first sight: distinction of accesses to aliases from those to global variables and explicit representation of control flow expressions. Both are motivated by our quest for efficiency and are grounded in strong implementation constraints in the context of dynamic weaving of binary C code: An access to a local alias is several magnitudes slower than that to a global variable, and matching of control flow joinpoints can be done using an atomic test on the implementation level.

3.3 Pointcuts

We now present a pointcut language (Fig. 9) that provides constructs to match individual joinpoints.

Primitive pointcuts are defined by *PPrim* and comprise three basic pointcuts: matching calls, global variable accesses and control flow joinpoints. Primitive pointcuts can also be combined using a logical "or", noted ||.

A call pointcut *PCall* selects all call joinpoints `callJP`(val funId($\overrightarrow{val}$)), i.e., all calls to a function matching the signature *type funId*($\overrightarrow{type}$), where the arguments of the function can be bound to pointcut variables using argument binder `args`($\overrightarrow{pattern}$) and the return value can be bound to a pointcut variable using a return clause `return`(pattern). The two constructs `args`($\overrightarrow{pattern}$) and `return`(pattern) can also provide pattern matching by using values (or already bound pointcut variables) in *pattern*. Pointcuts can also depend on a Boolean condition using the `if`-constructor.

A global access pointcut *PAccGlobal* selects either all read joinpoints, i.e., `readGlobalJP`(varId, val), or all write joinpoints `writeGlobalJP`(varId, val, size) on the global base program variable *varId*. In these cases, the read or written value can be bound to a variable using `value`(pattern). In addition, the size of the written value can be bound with `size`(varName). Pattern matching can also be used for variable access.

A control flow pointcut *PCf*, which is of the form `controlflow`($PCallSig_1$, ..., $PCallSig_n$, *PCfEnd*), matches `controlflowJP`(funId$_1$, ..., funId$_n$, cfEnd) joinpoints, where the function identifier in $PCallSig_i$ is $funId_i$. Similarly, a control flow pointcut may match a global variable access for a given stack configuration. The pointcuts of the form `controlflowstar`(...) select calls or global variable accesses in a stack context, allowing for calls that are not directly nested within one another.

$$
\begin{array}{lll}
PPrim & ::= & PCall \\
 & | & PAccGlobal \\
 & | & PCf \\
 & | & PPrim \,||\, PPrim \\
 & & \\
PCall & ::= & PCallSig\ [\ \&\&\ \texttt{args(}\ \overrightarrow{pattern}\ \texttt{)}\ ]\ [\ \&\&\ \texttt{return(}\ \text{pattern}\ \texttt{)}\ ] \\
 & & [\ \&\&\ PIf\ ] \\
PCallSig & ::= & \texttt{call(}\ \text{type funId}(\overrightarrow{type})\ \texttt{)} \\
 & & \\
PIf & ::= & \texttt{if(}\ \text{expr}\ \texttt{)}\ \ [\ \&\&\ PIf\ ] \\
 & & \\
PAccGlobal & ::= & \texttt{readGlobal(}\ \text{type varId}\ \texttt{)}\ [\ \&\&\ \texttt{value(}\ \text{pattern}\ \texttt{)}\ ]\ [\ \&\&\ PIf\ ] \\
 & | & \texttt{writeGlobal(}\ \text{type varId}\ \texttt{)}\ [\ \&\&\ \texttt{value(}\ \text{pattern}\ \texttt{)}\ ] \\
 & & [\ \&\&\ \texttt{size(}\ \text{pattern}\ \texttt{)}\ ]\ [\ \&\&\ PIf\ ] \\
 & & \\
PCf & ::= & \texttt{controlflow(}\ PCallSigList,\ PCfEnd\ \texttt{)} \\
 & | & \texttt{controlflowstar(}\ PCallSigList,\ PCfEnd\ \texttt{)} \\
PCallSigList & ::= & PCallSig\ \ [\ ,\ PCallSigList\ ] \\
PCfEnd & ::= & PCall \mid PAccGlobal \\
 & & \\
PAcc & ::= & \texttt{read(}\ \text{var}\ \texttt{)}\ [\ \&\&\ PIf\ ] \\
 & | & \texttt{write(}\ \text{var}\ \texttt{)}\ [\ \&\&\ \texttt{size(}\ \text{pattern}\ \texttt{)}\ ]\ [\ \&\&\ PIf\ ] \\
 & & \\
pattern & ::= & var \mid val
\end{array}
$$

Fig. 9. Pointcut language

Finally, *PAcc*, an access pointcut for a global variable or all of its local aliases, matches all joinpoints of the form `readJP` or `writeJP`.

3.4 Aspect Language

The aspect language we propose is defined in Fig. 10. Aspects *Asp* are either primitive aspects *AspPrim*, or sequences of primitive aspects *AspSeq*.

Both primitive and sequence aspects can be combined with requirement statements. A requirement statement is needed for each function or global variable of the base program used in the aspect. Similar to the declaration of a function before its first use in a C file, e.g., in a header file, a function or global variable with identifier *Id* has to be specified in a requirement statement `require` *Id* `as` *Type*; before it can be used in an aspect.

A primitive aspect *AspPrim* combines a primitive pointcut with an advice that will be applied to all joinpoints selected by the pointcut. An advice (*Advice*) is a C function call that replaces a joinpoint in the base program execution (similarly to `around` in AspectJ). It must have the same return type as the joinpoint it replaces, that is, the type of the global variable in case of a read access, `void` for a write access and the return type of the function for a call. When the advice is empty (no `then` clause), the original joinpoint is executed. The original joinpoint can be skipped by calling an empty C function.

$$
\begin{array}{lll}
\mathit{Asp} & ::= & \mathit{RequireStmt}\ \ \mathit{Asp} \\
 & | & \mathit{AspPrim}\ \ [\ \&\&\ \texttt{until(}\ \mathit{AspPrim}\ \texttt{)}\ ] \\
 & | & \mathit{AspeSeq}\ \ [\ \&\&\ \texttt{until(}\ \mathit{AspPrim}\ \texttt{)}\ ] \\
 & & \\
\mathit{RequireStmt} & ::= & \texttt{require}\ \ \mathit{Id}\ \ \texttt{as}\ \ \mathit{Type}\ \texttt{;} \\
 & & \\
\mathit{AspPrim} & ::= & \mathit{PPrim}\ \ \mathit{Advice} \\
 & & \\
\mathit{AspSeq} & ::= & \texttt{seq(}\ \ \mathit{AspPrim} \\
 & & \qquad\quad \mathit{AspSeqElts} \\
 & & \qquad\quad \mathit{AspSeqElt}\ \texttt{)} \\
 & & \\
\mathit{AspSeqElts} & ::= & \mathit{AspSeqElt}\ \ [\mathit{AspSeqElts}] \\
 & | & \mathit{AspSeqElt}\ \ *\ \ \ [\mathit{AspSeqElts}] \\
 & & \\
\mathit{AspSeqElt} & ::= & \mathit{AspPrim} \\
 & | & \mathit{PAcc}\ \mathit{Advice} \\
 & | & (\mathit{AspSeqElt}\ ||\ \mathit{AspSeqElt}) \\
 & & \\
\mathit{Advice} & ::= & \ \texttt{;} \\
 & | & \texttt{then}\ \mathit{funId}(\overrightarrow{\mathit{pattern}})\ \texttt{;} \\
 & & \\
\mathit{pattern} & ::= & \mathit{var} \\
 & | & \mathit{value}
\end{array}
$$

Fig. 10. Aspect language

A sequence aspect is composed of a sequence of primitive aspects. A sequence instance is created when the pointcut of the first primitive aspect matches. The following primitive aspects in the sequence are activated as soon as the corresponding pointcut matches (i.e., a primitive aspect has priority over its predecessor if both match). All but the first and last primitive aspects can be repeated zero or multiple times by using the operator "$*$". Branching, i.e., a logical "or" between two primitive aspects in a sequence, is supported by the operator $||$. Different sequence instances are (conceptually) matched in parallel.

A primitive or a sequence aspect a can be used in combination with an expression `until`(a_1), to restrict its scope. In this case, once a joinpoint has been matched by a, the execution of a proceeds as previously described until a_1 matches.

To conclude the presentation of our language, note that it does not include some features, such as named pointcuts as arguments to `controlflow`s, and conjunctive terms, which are not necessary for the examples we considered but which could easily be added. (As an aside, note that such extensions of the pointcut language may affect the computability of advanced algorithmic problems, such as whether a pointcut matches some part of any base program [36].)

4 Formal Semantics for Expressive Aspects

In the previous sections, we have given an informal semantics of our aspect language. We now illustrate how the aspect language can be formally defined by means of two different semantics:

- A semantics translating our aspects language into an extension of the language used in the formal framework of [37]. This semantics abstracts from most implementation details but allows a clear and succinct definition of the main properties of our sequence construct.
- A semantics providing a translation scheme into the actual C implementation used in the Arachne tool. This semantics has been harnessed to establish correctness arguments about and and thus guide the implementation of our tool.

4.1 An Abstract Formal Semantics

Douence et al. [37, 38] have introduced a generic framework for AOP supporting stateful crosscuts, i.e., pointcuts with explicit state. Without relying on any specific programming language, they have applied this framework to the formal definition of aspects and for certain kinds of reasoning techniques over aspects. In the case of our aspect language, their language must be extended in order to deal with halting aspects, an unbounded number of sequential aspects executed in parallel and arbitrary joinpoint predicates. The grammar of our extended version, our tiny aspect language, is defined in Fig. 11. In this language, aspect expressions A consist of parallel combinations of aspects. C is a joinpoint predicate (similar to our pointcut language) expressed as a conjunction of a term pattern and possibly an expression from the constraint logic programming language CLP($\mathcal{R}$) [39].

An aspect A' is either:

- A parallel composition of two aspects $A_1 \ || \ A_2$.
- A recursive definition.
- A sequence formed using the prefix operation $C \rhd I;\ X$, where X is an aspect, a recursion variable, or a halting aspect STOP, and I a piece of code (i.e., an advice).

$$
\begin{array}{lll}
A & ::= A' & \\
 & \mid \ A \ || \ A & \textit{; parallelism} \\
 & & \\
A' & ::= \mu a.A' & \textit{; recursive definition } (a \in \mathcal{R}ec) \\
 & \mid \ \mathrm{C} \rhd \mathrm{I};\ A & \textit{; prefixing} \\
 & \mid \ \mathrm{C} \rhd \mathrm{I};\ a & \textit{; end of sequence } (a \in \mathcal{R}ec) \\
 & \mid \ \mathrm{C} \rhd \mathrm{I};\ \mathrm{STOP} & \textit{; halting aspect} \\
 & \mid \ A' \ \Box \ A' & \textit{; choice}
\end{array}
$$

Fig. 11. Tiny aspect language

– A choice construction $A_1 \Box A_2$ (A_1, A_2 must not be parallel expressions) which chooses the first aspect that matches a joinpoint (the other is thrown away). If both match the same joinpoint, A_1 is chosen.

One can think of a stateful aspect A (as well as A') as a kind of transition system. Thereby, an aspect is always in a certain state in its execution, e.g., at rule $\mathrm{C} \triangleright \mathrm{I}$ (which is the head of a sequence of rules, which in turn is possibly part of a more complex expression), and waiting on a joinpoint to match C. If a joinpoint matching C occurs, the aspect executes I and advances to the next state, i.e., the next rule in the sequence.

Protocol Translation. The semantics of the protocol translation aspect (from TCP to UDP) is given in Fig. 12. A sequence can have several instances. This is translated into the language A by the expression $a_1 \;\; || \;\; ...$, which starts a new sequence a_1 once the first joinpoint has been matched and continues to match the rest of the sequence in progress. The repetition operator "$*$" is translated into recursion on the variable a_2. The branching operator $||$ of the source language is translated into the choice operator $\Box$ of A. Finally, the last primitive aspect of the sequence occurs as the first aspect of a choice to get priority over the joinpoints *read* and *write* because of the repetition marked by "$*$". Note that we use joinpoint patterns with variables, where an overbar marks the first occurrence of a variable (i.e., its definition in opposition to a use) and subsequent variable occurrences without overbar mark variable uses (e.g., to use the value of the file descriptor *fd* in argument positions).

Buffer Overflow Detection. The semantics of the aspect for detecting buffer-overflows is given in Fig. 13. This definition reports overflows after memory for a buffer has been allocated until a joinpoint matches the `free` crosscut, in which case the sequence instance corresponding to the freed buffer will be stopped.

These examples demonstrate that this style of semantics clearly exhibits the advantages stated in the beginning by concisely defining three important properties of our sequence aspect:

1. A sequence can have several instances, as for each joinpoint matching the pointcut of the first primitive aspect, a new sequence instance is created. The parallel operator $a_1 \;\; || \;\; ...$ in the translation of the sequence aspect

$$
\begin{array}{l}
\mu a_1.\ \texttt{callJP}(\overline{\mathrm{fd}}\ \mathrm{socket}(\mathrm{AF\_INET}, \mathrm{SOCK\_STREAM}, 0)) \triangleright \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{socket}(\mathit{AF\_INET}, \mathit{SOCK\_DGRAM}, 0); \\
\quad a_1 \;||\; (\ \texttt{callJP}(\overline{\mathrm{var}_1}\ \mathrm{connect}(\mathrm{fd}, \overline{\mathrm{address}}, \overline{\mathrm{length}})) \triangleright \mathrm{returnZero}(); \\
\qquad\quad \mu a_2.\ \texttt{callJP}(\overline{\mathrm{var}_2}\ \mathrm{close}(\mathrm{fd})) \triangleright \mathrm{close}(\mathrm{fd});\ \mathrm{STOP} \\
\qquad\qquad\quad \Box\ \texttt{callJP}(\overline{\mathrm{var}_3}\ \mathrm{read}(\mathrm{fd}, \overline{\mathrm{readBuffer}}, \overline{\mathrm{readLength}})) \triangleright \\
\qquad\qquad\qquad\qquad \mathit{recvfrom}(\mathit{fd}, \mathit{readBuffer}, \mathit{readLength}, 0, \mathit{address}, \mathit{length}); a_2 \\
\qquad\qquad\quad \Box\ \texttt{callJP}(\overline{\mathrm{var}_4}\ \mathrm{write}(\mathrm{fd}, \overline{\mathrm{writeBuffer}}, \overline{\mathrm{writeLength}})) \triangleright \\
\qquad\qquad\qquad\quad \mathit{recvfrom}(\mathit{fd}, \mathit{writeBuffer}, \mathit{writeLength}, 0, \mathit{address}, \mathit{length}); a_2
\end{array}
$$

Fig. 12. Definition of the protocol translation using the tiny aspect language

$$\mu a_1.\ \texttt{callJP}(\overline{\text{buffer}}\ \text{malloc}(\overline{\text{allocatedSize}})) \rhd \text{malloc}(\text{allocatedSize});$$
$$a_1 \ ||\ \mu a_2.\ \texttt{callJP}(\overline{\text{var}_1}\ \text{free}(\text{buffer})) \rhd \text{free}(\text{buffer});\ \text{STOP}$$
$$\Box\ \texttt{writeJP}(\text{buffer}, \overline{\text{var}_2}, \overline{\text{writtenSize}})$$
$$\&\&\ (writtenSize > allocatedSize) \rhd reportOverflow()\ ;\ \ a_2$$

Fig. 13. Definition of the buffer overflow aspect using the tiny aspect language

expresses this property. Once the first joinpoint has been matched, a new sequence a_1 is started and the rest of the sequence in progress continues to match in parallel.

2. The last step in a sequence aspect determines the finalization of sequence instances. When a joinpoint matches the pointcut of the last sequence element and a sequence instance is in a state waiting for such a joinpoint, i.e., the instance has already passed all previous steps of the sequence, the advice of the last step is executed and then the instance is terminated. In A', the finalization is expressed by STOP, which terminates the corresponding sequence.
3. The star operator $*$ attached to a sequence step, besides expressing repetition, causes the following step to have priority over its predecessor. The choice operator $\Box$ and the order of arguments of the choice in the translation ensure this property.

Note that formal definitions such as that of the protocol translation aspect and the buffer overflow detection aspect precisely define several important issues, which are somewhat implicit in the sequence aspect construct. In particular, they define when new instances of the sequence aspect are created: A new sequence instance is created once the first step in the sequence is matched, i.e., sequences are implicitly in scope of a repetition. The abstract semantics could be used, e.g., to formally prove that two instances match when a joinpoint matches the first as well as another step of a sequence. Furthermore, they disambiguate potentially nondeterministic situations, e.g., when two pointcuts of consecutive primitive aspects in the sequence match at the same time. Finally, this style of semantics clearly abstracts from implementation details, e.g., how the sequence state is represented in the implementation.

4.2 An Implementation-Level Semantics

Due to its abstractness, the semantics presented in the previous section illustrates certain properties of Arachne's aspect language very clearly, e.g., when new sequence instances are created. However, it abstracts from many details that are relevant, in particular, to judge the correctness of the Arachne tool: most important the above semantics abstracts from the generated C code, the C run-time environment and the concrete weaving process. In order to support a detailed understanding of the Arachne tool we have therefore developed an implementation-level formal semantics, which we present in this section.

This implementation-level semantics — in the remainder of this section the term "semantics" always refers to the implementation-level semantics — is

formulated as a denotational semantics [40] whose valuation functions define transformations from aspects into the corresponding C code executed by the Arachne tool. Technically, the valuation functions map syntactic categories of our language to a list of code generation functions. The *code generation functions* define code that handles the initialization of aspects, the dynamic conditions that are used to check whether a joinpoint actually matches the pointcut of an aspect as well as calls to the advice.

In addition, the generated code contains symbolic references to rewriting sites, i.e., places in the base program that have to be rewritten. The exact sites to be rewritten are first known at run time, as only then the aspect is woven into the base program. After the aspect code is generated, it will be compiled into a dynamic link library (DLL).[3] At weave time, the aspect DLL will then instruct Arachne to instrument the base program at the appropriate places and once such a site is encountered at run time, the dynamic predicates are tested and the advice function eventually executed.

This semantics therefore helps understanding of the Arachne tool by the following two characteristics:

- code generation functions providing a structured presentation of the executed C code
- a notion of rewriting sites providing an explicit representation of the weaving process

This way it is concrete enough to serve for correctness considerations of our tool, while being abstract enough to enable such considerations compositionally in terms of structural entities.

In this section we first present the denotational semantics in the context of a concrete example aspect and the evaluation of that aspect by means of the semantics. Second, we present a detailed overview of the semantics (a complete account can be found in [41]).

Example: Semantics of a Control Flow-Based Aspect. In order to illustrate the semantics and provide some information about the complexity in using the semantics (which cannot be completely avoided since it enables, in fine, derivation of the executed C code), we first discuss a concrete transformation. The following example shows an aspect that executes an advice `action(x)` when the function `h` is called within a control flow path on which functions `f` and `g` have already been called (see Listing 1 for the aspect definition).

Figure 14 presents three steps resulting from the application of the valuation functions of the semantics to the preceding aspect definition:

(a) The initial transformation step introduces initialization code and calls the valuation function corresponding to the aspect at hand (here **AP**).

[3] A library that is linked to a process/application at run time rather than at compile time and can be shared between several processes (called "shared object libraries" under Unix).

```
A⟦controlflow(call(int f(int)),call(long g(short)),
              call(float h(double))&&args(x))
         then action(x);⟧

   = (step a)
     createAspectInitialization(1);
     createAspectCompletionGuard();
     AP⟦controlflow(call(int f(int)),call(long g(short)),
                    call(float h(double))&&args(x))
              then action(x);⟧(1)

   ...

   = (step b)
     createAspectInitialization(1);
     createAspectCompletionGuard();
     defineAF_ACTION(1,1, action(x) );
     defineMacro(NUMBER_OF_JPS,1,1);
     defineJPMacro(1,1);
     let (c,d) = PCSL⟦call(int f(int)),call(long g(short))⟧("",0)
        in defineCF_BEGIN(1,0,c,d);
           defineCF_END_FC(1,0);
     PC⟦call(float h(double))&&args(x))⟧(1,0)
     createPrimitiveAspect(1);

   ...

   = (step c)
     createAspectInitialization(1);
     createAspectCompletionGuard();
     defineAF_ACTION(1,1, action(x) );
     defineMacro(NUMBER_OF_JPS,1,1);
     defineJPMacro(1,1);
     defineCF_BEGIN(1,0,""f","g"",2);
     defineCF_END_FC(1,0);
     defineORIGINAL_FC(1,"float","h","double","x");
     createEntryPointFunctionFC(1,0,"float","h","double","x",
                                                 "double x");
     createJoinPointFunCall(1,0,"h");
     createPrimitiveAspect(1);
```

Fig. 14. Example: Transformation of an control flow-based aspect (excerpt)

```
controlflow(call(int f(int)),call(long g(short)),
            call(float h(double)) && args(x)) then action(x);
```

Listing 1. Aspect using a control flow pointcut

(b) An intermediate step which enables matching of the call to `h` (via the pointcut valuation function **PC**) after the call sequence `f;g` has been matched (valuation function **PCSL**)).
(c) The final step represents the complete aspect code. This definition is given in terms of functions manipulating macro definitions (which, in turn, correspond to "real" C macros). The use of macro manipulation functions allows the semantics to be expressed quite concisely while still completely defining the executing code. The final step makes explicit, e.g., that pointcut matching is specialized w.r.t. the concrete number of joinpoints (through the use of `NUMBER_OF_JPS`), which governs how many concrete joinpoint macros can be instantiated (via `defineJPMacro`).

Listing 2 shows the executed code, i.e., once all the macro definitions and manipulations resulting from the final step of the transformation shown in Fig. 14 have been resolved.

This code (which is actually executed code, not some pseudocode) that will be generated by the compiler consists of initialization code, the advice and entrypoint of the aspect, a file guard and an aspect structure.

When the compiled aspect DLL is loaded, the initialization code (lines 1–4) triggers the automatic initialization of the aspect. Thereby the aspect is added to the active aspects and Arachne's kernel instruments all sites in the base program affected by the aspect structure (lines 32–54), which in the example are all function calls to `h`. Once the base: program executes a site rewritten for the aspect, the guard (lines 6–10) of the aspect, which indicates the progress of the weaving process, is checked to see whether it is true or false. In case all affected sites have been rewritten, the guard is set to true and the corresponding entrypoint function (lines 12–30) is invoked. In the entrypoint function the dynamic part of the pointcut is checked, to see whether the joinpoint really matches. In our example, the stack is checked for the functions `f` and `g` that are specified in the `controlflow` pointcut. If the dynamic predicate of the pointcut holds for the joinpoint, the advice/action (line 24) is executed; otherwise the original function is executed (lines 27–28).

The aspect structure (lines 32–54) consists of an array of joinpoints affected by the aspect (lines 35–48). Each joinpoint in the array specifies an entrypoint function (line 42), has a type and, in case of a function call or a global read or write access, additionally specifies a function identifier or a variable identifier, respectively. In the example, the only joinpoint of the array is a function call joinpoint, and thus the function identifier is specified (line 43). The `AllocatorAPI` provides an interface for the dynamic allocation and deallocation of structures, and each structure required for an aspect thus has a pointer to it (lines 8, 15, 34, 37, 41).

Overview of the Implementation-Level Semantics. Figure 15 shows a typical excerpt of the semantics itself. Valuation functions typically map syntactic entities to lists of code generation functions (see the signature of the valuation function **A**). They may, however, also depend on context information, e.g., the

```
static Aspect * __aspect_1__; static void initAspect_1()
__attribute__ ((constructor)); static void initAspect_1() {
    AspectsInFile->api->add(AspectsInFile,__aspect_1__)
}

static Guard * __fileGuard__ = & (Guard) {
    (GuardAPI*) & __guardAPI__,
    (AllocatorAPI*) & __transparentAllocatorAPI__,
    false
};

float entryPointOfJoinPoint_1_0(double x) {
    static CFlow * cflow = & (CFlow) {
            (CFlowAPI*) & __CFLowAPI__,
            (AllocatorAPI*) & __transparentAllocatorAPI__,
            (char* []) {"f","g"},
            UNKNOWN_EIP_2_FUNCTION_ADDRESS
         };
    static boolean init = false;
    if(!init) {
       defaultLoader->api->loadJoinpoint(defaultLoader,cflow);
    }
    if(CHECK_STACK(cflow->functions,cflowFunctions,2,1,1)){
            return action(x);
    }
    else{
      return ((float (*) (double))
        (defaultAspectLoader->api->getSymbolsByName("h")))(x);
    }
}

static Aspect* __aspect_1__ = & (Aspect) {
    (AspectAPI*) & __AspectAPI__,
    (AllocatorAPI*) & __transparentAllocatorAPI__,
    & (ArrayOfJoinpoint) {
        (ArrayOfJoinpointAPI*) & __ArrayOfJoinpointAPI__,
        (AllocatorAPI*) & __transparentAllocatorAPI__,
        (Joinpoint*) {
            Joinpoint) &(void*[]) {
                (JoinpointAPI*) &__FunctionCallJoinpointAPI__,
                (AllocatorAPI*) &__transparentAllocatorAPI__,
                (void*) entryPointOfJoinPoint_1_0,
                "h"
            };
        },
        1,
        1
    },
    UNKNOWN_SOURCE,
    UNKNOWN_PATH,
    UNKNOWN_LINE,
    TO_STRING("hand␣generated"),
    __fileGuard__
}
```

Listing 2. Example: Generated code for a control flow aspect (Listing 1)

```
A:    Aspect ⟼ CG*

A⟦AspSeq⟧ =
            createAspectInitialization(1);
            createAspectCompletionGuard();
            AS⟦AspSeq⟧(1)


PCF:        ControlFlowPointcut ⟼( INT × INT ) ⟼CG*

PCF⟦controlflow(PCallSigList,PCall)⟧(n,m) =
            let (l,d) = PCSL⟦PCallSigList⟧("",0)
                    in defineCF_BEGIN(n,m,l,d);
                    defineCF_END_FC(n,m);
            PC⟦PCall⟧(n,m)


PCSL:       FunctionCallSignature* ⟼(STRING × INT) ⟼(STRING × INT)

PCSL⟦call(Type FunId(TypeList))⟧(s,i) =
            (concat(s,concat("\"",concat(S⟦FunId⟧ ,"\""))),i+1)

PCSL⟦call(Type FunId(TypeList)),PCallSigList⟧(s,i) =
            PCSL⟦PCallSigList⟧(concat(concat(s,","),
                        concat("\"",concat(S⟦FunId⟧ ,"\""))),i+1)
```

Fig. 15. Valuation function of implementation-level semantics (excerpt)

function identifiers (e.g., the string arguments in the signature of function **PCF**) and position arguments in a sequence of a list of code generation functions (see the signatures of functions **PCF** and **PCSL**). The valuation function **A** defines how aspects are transformed by introducing aspect intialization code and then calling the valuation function of the current aspect construct (sequences, in the excerpt). The valuation function **PCF** defines that control-flow pointcuts are translated by first generating test code (by means of **PCSL**) for the sequence of calls in whose flow the final call has to occur, setting the macros `CF_BEGIN`, `CF_END_FC` which allow to test such a context and, finally, generating test code for the final call (using **PC**).

5 Dynamic Weaving with Arachne

Arachne is built around two tools (Fig. 16), an aspect compiler and a runtime weaver. The aspect compiler translates the aspect source code into a compiled library that, at weaving time, directs the weaver to place the hooks in the base program. The hooking mechanisms used in Arachne are based on improved techniques originally developed for μDyner [4]. These techniques allow users to rewrite the binary code of executable files on the fly, i.e., without pausing the

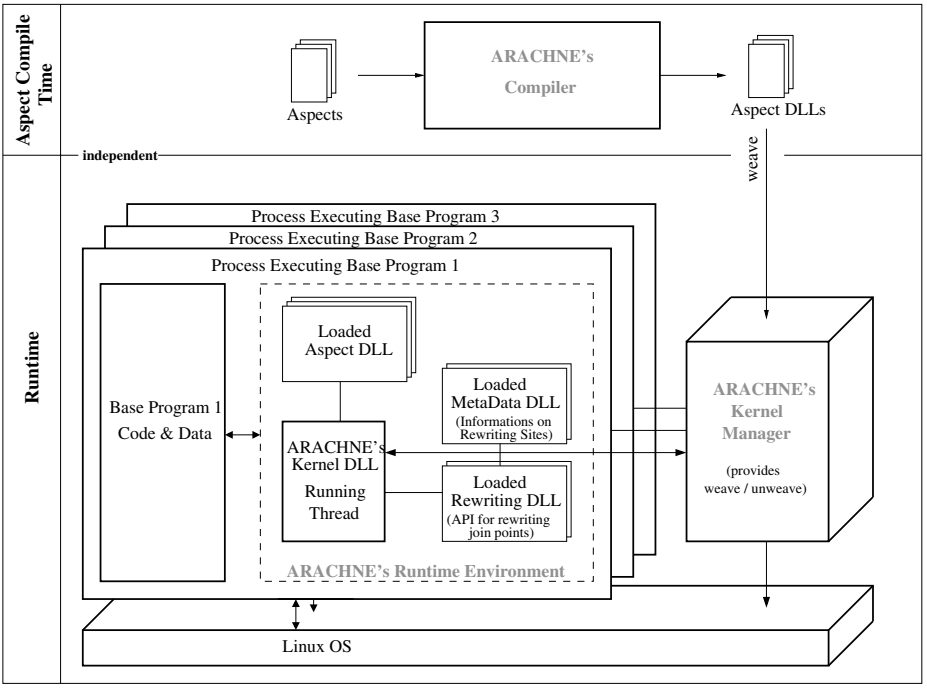

Fig. 16. Arachne's architecture

base program, as long as these files conform to the mapping defined by the Unix standard [42] between the C language and x86 assembly language. Arachne's implementation is structured as an open framework that allows one to experiment with new kinds of joinpoints and pointcut constructs. Another important difference between Arachne and μDyner is that μDyner requires a compile time preparation of the base program, whereas Arachne does not. Hence Arachne is totally transparent for the base program while μDyner is not.

5.1 The Arachne Open Architecture

The Arachne open architecture is structured around three main entities: the aspect compiler, the instrumentation kernel and the different rewriting strategies. The aspect compiler translates the aspect source code into C before compiling it. Weaving is accomplished through a command line tool `weave` that acts as a front end for the instrumentation kernel. `weave` relays weaving requests to the instrumentation kernel loaded in the address space of the program through Unix sockets. Upon reception of a weaving request, the instrumentation kernel selects the appropriate rewriting strategies referred by the aspects to be woven and instruments the base program accordingly. The rewriting strategy consults the pointcut analysis performed by the aspect compiler to locate the places where the binary code of the base program needs to be rewritten. It finally modifies the binary code to actually tie the aspects to the base program.

With this approach, the Arachne core is independent of a particular aspect, of the aspect language, of the particular processor architecture and of a particular base program. In fact, all dependencies to aspect language implementation are limited to the aspect compiler. All dependencies to the operating system are localized in the instrumentation kernel and, finally, all dependencies to the underlying hardware architecture are modularized in the rewriting strategies.

The Arachne Aspect Compilation Process. The aspect compilation scheme is relatively straightforward: It transforms advices into regular C functions. Pointcuts are rewritten as C code driving hook insertions into the base program at weaving time. There are, however, cases where the sole introduction of hooks is insufficient to determine whether an advice should be executed. In this case, the aspect compiler generates functions that complement the hooks with dynamic tests on the state of the base program. These dynamic tests are called *residues* in AspectJ, and the rewritten instructions within the base program the *shadow* [43]. Once the aspects have been translated into C, the Arachne compiler uses a legacy C compiler to generate a dynamically linked library (DLL) for the compiled aspects.

The Arachne Weaving Process. From a user viewpoint, the Arachne `weave` and `deweave` command line programs the same syntax as μDyner's version. They both take two arguments. The first identifies the process to weave aspects in or deweave aspects from, and the second indicates the aspect DLL. However, Arachne can target potentially any C application running on the machine, while μDyner was limited to applications compiled with it running on the machine. When Arachne's `weave` receives a request to weave an aspect in a process that does not contain the Arachne instrumentation kernel, it loads the kernel in the process address space using standard techniques [44].

The instrumentation kernel is transparent for the base program, since the latter cannot access the resources (memory and sockets essentially) used by the former. Once injected, the kernel creates a thread with the Linux system call: `clone`. This thread handles the different weaving requests. Compared to the POSIX `pthread_create` function, the usage of `clone` allows the instrumentation thread to prevent the base program to access its sockets. The instrumentation kernel allocates memory by using side-effect-free allocation routines (through the Linux `mmap` API). Because the allocation routines are side-effect-free, Arachne's memory is totally invisible to the base program. It is up to the aspect to use Arachne's memory allocation routines or base program-specific allocation functions. This transparency turns out to be crucial in our experiments. Legacy applications such as Squid use dedicated resource management routines and expect any piece of code they run to use these routines. Failures will result in an application crash.

After loading an aspect, the instrumentation kernel rewrites the binary code of the base program. These rewriting strategies are not included in the kernel and must be fetched on demand by each loaded aspect.

5.2 Rewriting Strategies

Rewriting strategies are responsible for transforming the binary code of the base program to effectively tie aspects to the base program at weaving time. These strategies localize Arachne's main dependencies to the underlying hardware architecture. In general, rewriting strategies need to collect information about the base program. This information typically consists of the addresses of the different shadows, their sizes, the symbol (i.e., function or global variable name) they manipulate, their length, etc. In order to keep compiled aspects independent from the base program, this information is gathered on demand at run time. The mapping between a symbol name in the base program source code and its address in memory is inferred from linking information contained in the base program executable. However, because this information can be costly to retrieve, `Arachne` collects and stores it into metainformation DLLs. These DLLs behave as a kind of cache and lessen the problem of collecting the information required to instrument the base program. To implement our aspect language, Arachne provides a set of eight rewriting strategies that might eventually use each other.

Strategies for `call`, `readGlobal` and `writeGlobal`. In Arachne, `call`, `readGlobal` and `writeGlobal` allow an advice to be triggered upon a function call, a read on a global variable or a write, respectively. While the implementation

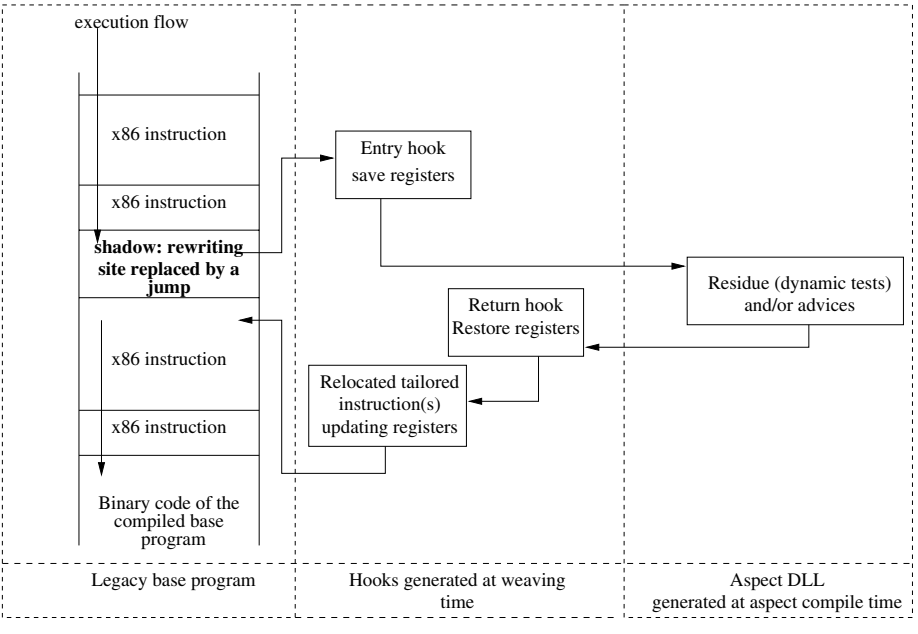

Fig. 17. Generic hook operations

of `readGlobal` and `writeGlobal` in Arachne is close to the one in μDyner, Arachne implements the strategy for `call` by rewriting function invocations found in the base program. μDyner instead rewrites the function body of the callee. On the Intel architecture, function calls benefit from the direct mapping to the x86 `call` assembly instruction that is used by almost, if not all, compilers. Write and read accesses to global variables are translated into instructions using immediate, hard-coded addresses within the binary code of the base program. By comparing these addresses with linking information contained in the base program executable, Arachne can determine where the global variable is being accessed. Therefore those primitive pointcuts do not involve any dynamic tests. The sole rewriting of the binary base program code is enough to trigger advice and residue[4] executions at all appropriate points.

The size of the x86 `call` instruction and the size of an x86 jump (`jmp`) instruction are the same. Since the instruction performing an access to a global variable involves a hard-coded address, x86 instructions that read or write a global variable have at least the size of a x86 `jmp` instruction. Hence at weaving time, Arachne rewrites them as a `jmp` instruction to a hook. Hooks are generated on the fly on freshly allocated memory. As shown in Fig. 17, hooks contain a few assembly instructions that save and restore the appropriate registers before and after an advice (or shadow) execution. A generic approach is to have hooks save the whole set of registers, then execute the appropriate residue and/or advice code before restoring the whole set of registers. Finally, the instructions found at the joinpoint shadow are executed to perform the appropriate side effects on the processor registers. This is accomplished by relocating the instructions found at the joinpoint shadow. Relocating the instructions makes the rewriting strategies handling read and write access to global variables independent from the instruction generated by the compiler to perform the access.[5] The limited number of x86 instructions used to invoke a function allows Arachne's rewriting strategy to exploit more efficient, relocation-free hooks.

Strategies for `controlflow` and `controlflowstar`. Every time a C function is called, the Linux runtime creates an activation record on the call stack [42]. Like μDyner, Arachne's implementation of the rewriting strategy for `controlflow` uses the most deeply nested function call (or global read or write access) in the control flow pointcut as shadow. This shadow triggers a residue. This residue uses the activation record's chaining to check whether the remaining function calls of the control flow are on the call stack maintained by the Linux run time. An appropriate usage of hash tables that store the linking information contained in the base program executables can thereby decrease the cost of determining if a specific function is the caller of another to a pointer comparison. Therefore, the residue for a `controlflow` with n directly nested functions implies exactly

[4] Residues (i.e., dynamic tests on the base program state) are required when these primitive pointcuts are combined with conditional pointcuts or when pattern matching is involved.

[5] About 250 x86 instruction mnemonics can directly manipulate a global variable. This corresponds to more than 1000 opcodes.

n pointer comparisons. However, the residue worst-case run time for the indirect control flow operator `controlflowstar` that allows for not directly nested functions is proportional to the base program stack depth.

Strategies for `read` and `write`. `read` and `write` are new joinpoints not included in μDyner that have been added to the latest version of Arachne. Their implementation relays on a page memory protection as allowed by the Linux operating system interface (i.e., `mprotect`) and the Intel processor specifications [46].[6] A `read` or `write` pointcut triggers a residue to relocate the bound variable into a memory page that the base program is not allowed to access and adds a dedicated signal handler. Any attempt made by the base program to access the bound variable identified will then trigger the execution of the previously added signal handler. This handler will then inspect the binary instruction trying to access the protected page to determine whether it was a read or a write access before eventually executing the appropriate advice.

Strategies for `seq`. Like `read` and `write`, `seq` is a new language feature of Arachne. μDyner offers no equivalent construct. Arachne's rewriting strategy of this operator associates a linked list to every stage inside the sequence except the last one. Each stage in a sequence triggers a residue that updates these linked lists to reflect state transitions of currently matching execution flows. Upon matching of the first pointcut of the first primitive aspect in the `seq`, a node is allocated and added to the associated linked list. This node contains a structure holding variables shared among the different pointcuts within the sequence. Once a joinpoint matches a pointcut of an primitive aspect denoting a stage in the sequence, Arachne consults every node in the linked list associated with the previous stage and executes the corresponding advice.[7] Arachne eventually updates the node and, in the absence of a $*$, moves it to the list associated with the currently matched pointcut. If the matching pointcut corresponds to the end of the sequence, structures are not moved into another list but are freed. Our aspect compiler includes an optimization where structures are allocated from a resizable pool, and upon a sequence termination, structures are not freed but returned to the pool.

5.3 Limitations of Arachne

Aggressive optimizations of the base program might prevent Arachne from seamlessly weaving aspects. Two optimizations are not yet supported by Arachne. First, if the compiler inlines a function in another one within the binary code of the base program, the Arachne weaver will fail to properly handle pointcuts referring to that function. Second, control flow pointcuts are based on the chaining

[6] Even if this implementation is Linux/x86 specific, it is applicable to arbitrary architectures supporting memory paging.

[7] In case the previous stage pointcut was used with a star $*$, Arachne examines nodes from linked list associated with the last two previous stages, and so on, until a not-starred primitive aspect in the sequence is reached.

of activation records. On the x86 architecture, in leaf functions, optimizing compilers sometimes do not maintain this chaining to free one register for the rest of the computation. This, however, has not been a problem during our experiments as we used the open-source C compiler `gcc`. Arachne does not require the base program's source code in order to weave aspects, however it relies on linking information embedded within the executable to determine where the program code must be rewritten. Hence the stripping of symbols from executables as well as aggressive optimizations that break the interoperability between compilers and/or debuggers are incompatible with Arachne. In practice, Arachne can be used on applications compiled like Squid with two of the three `gcc` optimization levels.

6 Performance Evaluation

Aspect-oriented solutions will be used if the aspect system's language is expressive enough and if the aspect system overhead is low enough for the task at hand. The purpose of this section is to study Arachne's performance. We first present the speed of each Arachne language construct and compare it to similar C language constructs. Second, we study the overhead of extending Squid with a prefetching policy. Third, we measure the overhead induced by protecting the Washington University's FTP server `wu-ftpd` from a buffer-overflow vulnerability. These two case studies show that even if the cost of some Arachne aspect language constructs might be high compared to C language constructs, this overhead is largely amortized in real applications.

6.1 Evaluation of the Language Constructs

This performance evaluation focuses on studying the cost of each construct of our aspect language. To estimate the cost for each construct of our aspect language, we wrote an aspect using this construct that behaves as an interpreter of the base program. For example, to study the performance of `readGlobal`, we wrote an aspect whose action returns the value of the global variable referred to in the pointcut, i.e., we wrote aspects behaving like the base program. For each of these aspects, we compare the time required to perform the operation matching the pointcut, in case the operation is interpreted by the woven aspect, with the time required to carry out the operation natively (without the woven aspect). For example, to study the performance of `readGlobal`, we first evaluate the time needed to retrieve the global variable value through the code generated by the C compiler `gcc` without any aspect woven and compare this value to the time needed to retrieve the global variable value through the aspect once it has been woven in the base program. We express our measurements as a ratio between these two durations to abstract from the experimentation platform.

This approach requires the ability to measure short periods of time. For instance, a global variable value is usually retrieved (`readGlobal` in our aspect language) in a single clock tick. Since standard time measurement APIs were

not precise enough, our benchmarking infrastructure relies on the `rdtsc` assembly instruction [45]. This instruction returns the number of clock cycles elapsed since power up. The Pentium 4 processor has the ability to dynamically reorder the instructions it executes. To ensure the validity of our measurement, we thus insert `mfence` instructions in the generated code whose execution speed is being measured. An `mfence` forces the preceding instructions to be fully executed before going on. The pipeline mechanism in the Pentium 4 processor entails that the speed of a piece of assembly code depends on the preceding instructions. To avoid such hidden dependencies, we place the operation whose execution time is being measured in a loop. We use `gcc` to unroll the loop at compile time, and we measure the time to execute the complete loop. This measure divided by the number of loop repetitions yields an estimation of the time required to execute the operation. The number of times the loop is executed is chosen after the relative variations of the measures, i.e., we increased the number of repetitions until ten runs yields an average relative variation not exceeding 5%. To check the correctness of our experimental protocol, we measured the time needed to execute a `nop` assembly instruction, which requires one processor cycle according to the Intel specification. The measures of `nop` presented a relative variation of 1.6%.

Table 1 summarizes our experimental results. Using the aspect language to replace a function that returns immediately is only 1.3 times slower than a direct, aspectless call to that empty function. Since the aspect compiler packages advices as regular C functions, and because a `call` pointcut involves no residue, this good result is not surprising. When an access to a global variable is replaced by an advice execution, the hooks generated by the rewriting strategy need to prepare the processor to call the advice function. This increases the time spent in the hooks. A `seq` of three invocations of empty functions is only 3.2 times slower than the direct, aspectless, three successive functions calls. Compared to the pointcuts used to delimit the different stages, the `seq` overhead is limited to a few pointer exchanges between the linked lists holding the bound variable. On Intel x86, global variable accesses benefit from excellent hardware support. In the absence of aspects, a direct global variable read is usually carried out in a single unique cycle. To trigger the advice execution, the Arachne runtime has to save

Table 1. Speed of each language construct used to interpret the base program compared to a native execution

	Execution times (cycles)		
	Arachne	Native	Ratio
`call`	28±2.3%	21±1.9%	1.3
`seq`	201±0.5%	63±1.7%	3.2
`cflow`	228±1.6%	42±1.8%	5.4
`readGlobal`	2762±4.3%	1±0.2%	2762
`read`	9729±4.9%	1±0.6%	9729

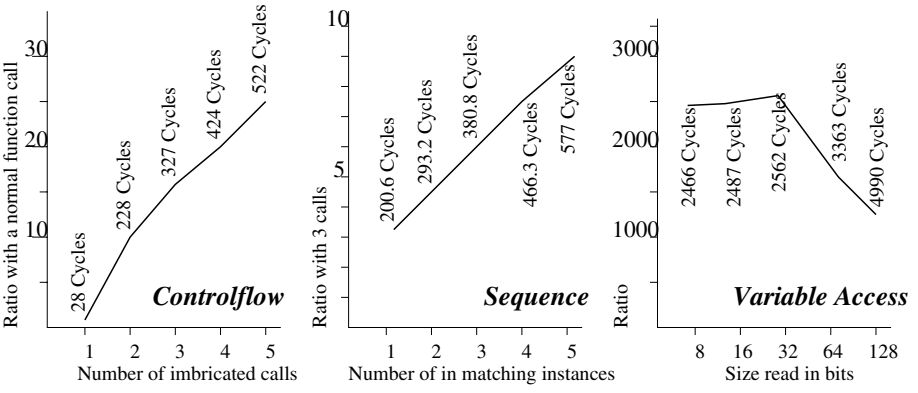

Fig. 18. `controlflow`, `seq` and `readGlobal` performances

and restore the processor state to ensure the execution coherency, as advices are packaged as regular C functions (see also Sect. 5.2). It is therefore not surprising that a global variable `readGlobal` appears as being 2762 times slower than a direct, aspectless global variable read. `read` performance can be accounted in the same way: In the absence of aspect, local variables are accessed in a single unique cycle. The signal mechanism used in the `read` requires that the operating system detects the base program attempt to read into a protected memory page before locating and triggering the signal handler set up by Arachne, as shown in Sect. 5.2. Such switches to and from kernel space remain slow. Using `read` to read a local variable is 9729 times slower than retrieving the local variable value directly, without aspects.

`seq` and `controlflow` can refer to several points in the execution of the base program (i.e., different stages for `seq` and different function invocations for the `controlflow`). The run time of these pointcuts grows linearly with the number of execution points they refer to and with the number of matching instances. Variable access pointcut performance varies depending on the size of the data accessed. Indeed, on IA32 architectures, an access to a variable smaller or equal to 32 bits is performed atomically in one processor cycle, while time to access a variable larger than 32 bits grows linearly with the variable size. Hence, the overhead of an aspect replacing an access to an up to 32-bit variable is constant and beyond amortized corresponding to the variable size. Figure 18 summarizes a few experimental results for `controlflow`, `seq` and `readGlobal` that provide evidence for these performance propositions.

6.2 Case Study on a Real Application

Since executing a base program with aspects can slow it down by a factor ranging between 1.3 and 9729, depending on the aspect construct used, we studied Arachne's performance on a real-world application, the Web cache Squid. We extended Squid with a prefetching policy [46]. As described in Sect. 3.1, we implemented this policy as a set of aspects and made a second implementation

of this policy by editing the Squid source code and recompiling it. This section compares the performance of these two implementations using standard Web cache performance indicators: throughput, response time and hit ratio.

A tool which seemingly is appropriate for such a real-world experience is the traces generated during Web cache executions. However, obtaining access traces adequate to study a Web cache performance is difficult. The trace must be long enough to fill the cache. Because of privacy issues, traces are usually not publicly available. Since traces do not include the content of the accessed pages, these pages must be downloaded again. In the meantime, the page contents may have changed and even the URLs may have disappeared.

Instead of traces, we based our evaluation on Web Polygraph [47]. Polygraph is a benchmarking tool developed by the Squid team that features a realistic HTTP and SSL traffic generator and a flexible content simulator.

We filled up the cache and simulated a one-day workload with its two request rate peaks observed in real-life environments [47]. Table 2 shows some results of our simulation. Measures have been made during the two request peaks. All measures, be it the hit time, the miss time, the time needed to deliver a document presenting the cache or not, are very similar, independent of Arachne being used or not. These measures prove that differences are imperceptible between the version of Squid extended by Arachne and the one extended manually (less than 1%). Hence, even if the cost of some of Arachne's aspect language constructs might seem high, they are largely amortized in real applications. To give a typical example observed on our experimental platform: in case of a cache hit, a 3.8-MB page was retrieved in a single second, the time spent in prefetching advices amounted to 1801 μsec, and the time spent within Arachne to execute the hooks and dynamic tests was 0.45 μsec. In a miss case, on the average, a client

Table 2. Performance comparison between manual modification and Arachne, for prefetching policy integration in Squid

	Arachne	**Manual**	
	Top1 **Top2**	**Top1** **Top2**	**Diff (%)**
Throughput (request/s)	5.59 5.58	5.59 5.59	–
Response time (ms)	1131.42 1085.31	1146.07 1074.55	1.2 – –1
Miss response time (ms)	2533.50 2528.35	2539.52 2525.34	0.2 – 1.8
Hit response time (ms)	28.96 30.62	28.76 31.84	–0.6 – 3.8
Hit ratio	59.76 61.77	59.35 62.22	–0.6 – 0.7
Errors	0.51 0.34	0.50 0.34	–1.9 – 0

retrieved the same page in 1.3 seconds, 16679 μsec were spent in the advices and 0.67 μsec within Arachne itself.

6.3 A Second Case Study: `wu-ftpd`

We also applied Arachne on the real-world application `wu-ftpd` (Washington University file transfer protocol Daemon), a widely deployed file transfer protocol service. It constitutes the basis for development of several other ftp servers, e.g., BSD ftpd, ProFTPD.

We performed measurements on `wu-ftpd` applying an aspect for the correction of a buffer overflow. We chose a buffer-overflow vulnerability identified in the s/key authentication mechanism discovered in 2004 and referenced by the Common Vulnerabilities and Exposures under the identifier `CVE-2004-0185`.

In order to evaluate `wu-ftpd`'s performance, we used `dkftpbench` [48], a benchmarking tool for FTP servers. `dkftpbench` permits users to stress ftp servers by faking client connections using automata. Each fake client authenticates to the server, retrieves a particular file and disconnects. `dkftpbench` constantly creates new automata and then permits users to measure instant/average/maximum numbers of simultaneous users. We recorded our measurements between two machines: one for `dkftpbench` and one for `wu-ftpd` over a 100-Mb/s ethernet. `wu-ftpd` was running on a Pentium 4, 3.3 GHz, with 512-MB RAM. Each file to be retrieved was 5-MB long, and network bandwidth dedicated to each client was set so that the network was never subject to congestion.

We measured the maximum number of users simultaneously served by `wu-ftpd` when running unprotected and protected with our aspect. Results show no significant difference between the two versions, which allow for 1008 and 1012 simultaneous users, respectively. This demonstrates that even if our aspect constructs might seem to consume an important amount of local resources, they are clearly reasonably applicable in real-world situations by permitting users to protect applications against attacks without impacting performance significantly.

Nevertheless, performance penalties could be significant. Indeed, frequent use of the `read` construct could greatly slow down program execution. Such a situation would arise, for example, when every single buffer in an application should be protected from overflowing. However, one of the main characteristics of our approach is that it supports the selective modification of system-level applications using aspects. Furthermore, those situations seem to be quite unlikely anyway: for all applications we have encountered, `read/write` and `readGlobal/writeGlobal` pointcuts have been marginal compared to `call` pointcuts.

7 Related Work

Aspect-oriented research currently focuses on object-oriented languages. Apart from μDyner and Arachne, there are few *aspect weavers for C* (or even C-like languages). AspectC [3] and AspectC++ [49] are two noteworthy exceptions.

They both rely on source-code transformation and solely weave aspects at compile time. DAC++ [50] and Toskana [51] are dynamic weavers for C++ and C. DAC++ is built around a metaobject protocol enabling the run time instrumentation required to weave aspects at run time. Since C++ does not include a standard metaobject protocol, the base program has to be compiled with a dedicated specific compiler. Toskana weaves aspects in a running Linux kernel. It does not allow users to weave aspects in user applications. In addition, the joinpoint model in Toskana is limited to function calls. Hence, none of these weavers is suitable to modularize and dynamically compose the concerns we considered.

There is quite a large body of work now on the notion of *expressive aspect languages*, where "more expressive" typically compares to AspectJ's pointcut and advice models. Our work has been inspired by Event-based AOP [52], which aims at the definition of pointcuts in terms of arbitrary relations between events. Nevertheless, many other approaches to expressive aspect languages exist. For example, data-flow relations [53], logic programming [54], process algebras [55], graphs [56] and temporal logics [57] have all been proposed as a basis for the definition of expressive aspect languages. However, few of these encompass dynamic weaving, and only the latter has been applied to C code under efficiency considerations similar to our setting (but using a static approach to weaving).

Research on explicit sequence pointcuts and aspects is still in its infancy. Sequential aspects were first introduced by Douence et al. with the notion of stateful aspects [37, 38]. They exploited the underlying notion of regular sequence aspects — which are thus of more restricted expressiveness than the sequence aspects considered in this article — to analyze aspect interactions, and a prototype supporting arbitrary relations between joinpoints for Java was implemented [58]. However, this prototype is based on static weaving and does not allow dynamic modification of aspects. Regular sequence aspects have also been integrated in the Java-based JAsCo aspect system [59]. In [60] a specialized language was proposed to define pointcuts as sequences of method calls in Java. A pointcut is associated with a single advice, which is executed at the end of the sequence. This approach does not support advice attached to the middle of a sequence. Moreover, this Java-based tool supports static source-code-only weaving.

Another class of techniques relevant to our work is dynamic binary code instrumentation, which has already been widely studied. These techniques were used in the first computers [61]. In these techniques, difficulty issues range from the complexity to rewrite binary code to the lack of a well-defined relationship between source code and the compiler-generated binary code. Pin [62] and Dyninst [63] enable programmers to modify any binary instruction belonging to an executable. Based on a just-in-time translation, Pin is very efficient but is limited to insert code before or after a binary instruction of the base program. This prevents Pin from serving as a back end for an aspect system using around-like aspect. Dyninst does not suffer from this limitation; it is designed around the Unix debugging API: `ptrace`. After suspending the base program execution, this

API allows a third-party process to read and write the base program memory. In comparison, Arachne suspends the base program at most once to inject, with `ptrace`, its kernel DLL into the base program process. In addition, instrumentation schemes written with Dyninst are not ensured to be reliable: Dyninst's implementation relocates several adjacent instructions. Since one of the relocated instructions can be a branching instruction target, the instrumentation success depends on the base program considered. In comparison, Arachne's joinpoint model has been devised to avoid these kind of issues by design.

Finally, there are many Java-based approaches to nonstatic code weaving, i.e., *dynamic weaving* and *load-time weaving*. Load-time weaving refers to the process of instrumenting the base program when the execution environment — the Java virtual machine, for instance — transfers it from disk storage to memory-executable structures. To provide aspect deployment at run time they have to prepare, i.e., instrument, the base program at load time and thus can imply a nonnegligible overhead, even in the absence of aspects [64, 65, 66, 67]. Contrary to those approaches, Arachne does not weave aspects at load time but dynamically at run time and does not require any anticipation of aspect weaving at load time. Furthermore, Arachne keeps a clear separation at run time between the base program code and the aspect code, so that aspects can be unwoven without leaving residues in the base program code. Some approaches, most notably JAsCo [68], Steamloom [69], JBoss AOP [70], Spring AOP [71] and AspectWerkz (which is currently under integration with AspectJ) support run-time weaving of aspects for Java. As Java-based approaches they cannot be applied to solve the legacy code problems we consider. Furthermore, they do not provide Arachne's fine-grained weaving (weaving on the level of processor instructions). Finally, since they rely on particularities of the Java platform (such as the debugging interface or Hotswap), the incurred performance overhead is large to very large compared to that of Arachne, and the implementation techniques themselves are not transferable to C.

8 Conclusion and Future Work

Technical issues such as double-free bugs and buffer overflows, networking, and prefetching are typical examples of concerns which crosscut system-level C applications: in many real-world legacy applications such as the Squid Web cache, these concerns are scattered over the entire program source code. Since security breaches and insufficient resources are often discovered after deployment, and because downtime must be avoided in many application contexts, there is a growing need to modularize and manipulate these concerns at runtime.

Security, networking and prefetching are appealing candidates for modularization using aspects. However, basic aspect-oriented techniques are not applicable due to the complex relationships between executions points these aspects are required to account for. We have proposed an aspect language enabling us to specify these relationships involving sequences of execution points as well as for variable aliases. We have shown how to successfully modularize security,

networking and prefetching concerns within this aspect language. Furthermore, we have presented two formal semantics for this language that clearly express different properties of the language due to their different abstraction levels.

The Arachne tool implements this language. It can weave and deweave aspects dynamically in running legacy C applications like Squid or the `wu-ftpd` ftp server. We have provided detailed evidence that the performance of these two applications, including modifications by Arachne aspects, competes with (optimal) manual source code modifications.

As future work, we intend to investigate how unexpected interactions between aspects can be detected at compile time and at weaving time. Unexpected interactions can occur when two aspects refer to the same joinpoint or by sharing variables. Detecting these interactions would greatly ease the development of large aspects libraries. Another lead for future work is to exploit the better modularization of system-level functionalities by Arachne aspects for the testing of such functionalities. We also plan to integrate debugging information support to Arachne. Because this is the missing link between source and binary code, that information should permit users to overcome aggressive optimizations performed by compilers.

References

[1] Wessels D. Squid: *The Definitive Guide*. O'Reilly, 2004

[2] Kiczales G., Lamping J., Menhdhekar A., Maeda C., Lopes C., Loingtier J.M., Irwin J. Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) *Proceedings European Conference on Object-Oriented Programming, LNCS vol. 1241,* Springer, Jyväskylä, Finland, pp. 220–242, 1997

[3] Coady Y., Kiczales G., Feeley M., and Smolyn G. Using AspectC to improve the modularity of path-specific customization in operating system code. In: Gruhn, V. (ed.) *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE-01).* Volume 26, 5 of SOFTWARE ENGINEERING NOTES, ACM, New York, pp. 88–98, 2001

[4] Ségura-Devillechaise M., Menaud J.M., Muller G., and Lawall J. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, ACM, pp. 110–119, 2003

[5] Arce I., Levy E. An analysis of the slapper worm. *IEEE Security and Privacy* 1:82–87, 2003

[6] Solar Designer: JPEG COM Marker Processing Vulnerability in Netscape Browsers. http://www.openwall.com/advisories/OW002-netscape-jpeg/ (1997)

[7] Ubuntu: Squid Proxy Cache Double Memory Free Vulnerability. http://www.security.nnov.ru/Idocument338.html (2005)

[8] American National Standards Institute: ANSI/ISO/IEC 9899-1999: *Programming Languages — C.* American National Standards Institute, New York, 1999

[9] CERT Coordination Center: CERT Advisory CA-2001-13 Buffer Overflow in IIS Indexing Service DLL. http://www.cert.org/advisories/CA-2001-13.html (2001)

[10] CERT Coordination Center: "Code Red" Worm Exploiting Buffer Overflow in IIS Indexing Service DLL (CERT Incident Note IN-2001-10). http://www.cert.org/incident_notes/IN-2001-08.html (2001)

[11] US-CERT (United States Computer Emergency Readiness Team): Microsoft SQL Server 2000 contains stack buffer overflow in SQL Server Resolution Service (Vulnerability Note VU#484891). http://www.kb.cert.org/vuls/id/484891 (2002)
[12] CERT Coordination Center: CERT Advisory CA-2003-04 MS-SQL Server Worm. http://www.cert.org/advisories/CA-2003-04.html (2003)
[13] US-CERT (United States Computer Emergency Readiness Team): Microsoft Windows RPC vulnerable to buffer overflow (Vulnerability Note VU#568148). http://www.kb.cert.org/vuls/id/568148 (2003)
[14] CERT Coordination Center: CERT Advisory CA-2003-20 W32/Blaster worm. http://www.cert.org/advisories/CA-2003-20.html (2003)
[15] Ruwase O. and Lam M.S. A practical dynamic buffer overflow detector. In: *Proceedings of the 11th Annual Network and Distributed System Security Symposium,* Internet Society, San Diego, CA, 2004
[16] CERT Coordination Center: CERT/CC advisories. http://www.cert.org/advisories/ (1988)
[17] Wagner D., Foster J.S., Brewer E.A., and Aiken A. A first step towards automated detection of buffer overrun vulnerabilities. In: *Network and Distributed System Security Symposium*, Internet Society, San Diego, CA, pp. 3–17, 2000
[18] Cowan C., Wagle P., Pu, C., Beattie S., and Walpole J. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In: *DARPA Information Survivability Conference and Exposition (DISCEX).* Vol. 2, Hilton Head Island, SC, USA, 119–129, IEEE 2000
[19] Wilander J. and Kamkar M. A comparison of publicly available tools for dynamic buffer overflow prevention. In: *Proceedings of the 10th Network and Distributed System Security Symposium*, Internet Society, San Diego, CA, pp. 149–162, 2003
[20] Larochelle D. and Evans D. Statically detecting likely buffer overflow vulnerabilities. In: *Proceedings of the 10th USENIX Security Symposium,* USENIX, Washington, DC, pp. 177–190, 2001
[21] Cowan C., Pu C., Maier D., Walpole J., Bakke P., Beattie S., Grier A., Wagle P., Zhang Q., and Hinton H. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: *Proc. 7th USENIX Security Conference*, USENIX, San Antonio, TX, pp. 63–78, 1998
[22] Jim T., Morrisett G., Grossman D., Hicks M., Cheney J., and WangY. Cyclone: A safe dialect of C. In: *Proceedings of the USENIX Annual Technical Conference*, USENIX, Monterey, CA, pp. 275–288, 2002
[23] Condit J., Harren M., McPeak S., Necula G.C., and Weimer W. CCured in the real world. In: *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ACM, San Diego, CA, pp. 232–244, 2003
[24] Jones R. and Kelly P. Backwards-compatible bounds checking for arrays and pointers in C programs. In: Kamkar, M. (ed.) *Proceedings of the Third International Workshop on Automatic Debugging.* Vol. 2, Linköping, Sweden, Linköping Electronic Articles in Computer and Information Science, pp. 13–26, 1997
[25] Keromytis A.D. "Patch on demand" saves even more time? *IEEE Computer*, 37:94–96, 2004
[26] US-CERT (United States Computer Emergency Readiness Team): Squid Proxy Server contains buffer overflow in parsing of the authentication portion of FTP URLs (Vulnerability Note VU#613459). http://www.kb.cert.org/vuls/id/613459 (2002)

[27] Berners-Lee T., Fielding R., Frystyk H. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0 (1996) Status: INFORMATIONAL.
[28] Postel J. Transmission Control Protocol. RFC 793. http://www.rfc-editor.org/rfc/rfc793.txt (1981)
[29] Arlitt M., Jin T. A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14:30–37, 2000
[30] Cidon I., Gupta A., Rom R., Schuba C. Hybrid TCP-UDP transport for web traffic. *Technical Report 99-71*, Sun Microsystems Laboratories, Palo Alto, CA, 1999
[31] Rabinovich M. and Wang H. DHTTP: An efficient and cache-friendly transfer protocol for web traffic. In: *IEEE INFOCOM*, pp. 1597–1606, 2001
[32] Chen H. and Mohapatra P. CATP: A context-aware transportation protocol for HTTP. In: *International Workshop on New Advances in Web Servers and Proxy Technologies Held with ICDCS*, Providence, RI, USA, pp. 922–927, 2003
[33] Postel J. User datagram protocol. RFC 768. http://www.rfc.net/rfc768.html (1980)
[34] Comer D., Stevens D. Internetworking with TCP/IP, Volume III — Client-Server Programming and Applications for the BSD Socket Version. Volume III. Prentice Hall, 1993
[35] Issarny V., Banâtre M., Charpiot B., Menaud J.M. Quality of service and electronic newspaper: The Etel solution. *LNCS vol. 1752*, pp. 472–496, 2000
[36] Lieberherr K.J., Palm J., Sundaram R. Expressiveness and complexity of crosscut languages. *Technical Report NU-CCIS-04-10*, Northeastern University, 2004
[37] Douence R., Fradet P. and Südholt M. A framework for the detection and resolution of aspect interactions. In: *GPCE'02: Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, *LNCS vol. 2487*, Springer, Pittsburgh, PA, USA, pp. 173–188, 2002
[38] Douence R., Fradet P., and Südholt M. Composition, reuse and interaction analysis of stateful aspects. In: *AOSD'04: Proc. of 3rd International Conference on Aspect-Oriented Software Development*, ACM, Lancaster, UK, pp. 141–150, 2004
[39] Jaffar, J., Michaylov, S., Stuckey, P.J., Yap, R.H.C. The clp(r) language and system. *ACM Trans. Program. Lang. Syst.*, 14:339–395,1992
[40] Schmidt D.A. Denotational semantics - A methodology for language development. Allyn and Bacon, http://www.cis.ksu.edu/~schmidt/text/densem.html (1986)
[41] Fritz T. An expressive aspect language with arachne. *Master's thesis*, Ludwig-Maiximilians-Universität München, 2005
[42] System Unix U.S.L.: System V application binary interface intel 386 architecture processor supplement. Prentice Hall Trade, 1994
[43] Hilsdale E. and Hugunin J. Advice weaving in AspectJ. In: *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, ACM, pp. 26–35, 2004
[44] Clowes S. Injectso: Modifying and spying on running processes under linux. In: *Black Hat Briefings*, 2001
[45] Intel Corportation: IA-32 Intel Architecture software developer's manual. *Intel Corportation*, 2001
[46] Chinen K.I. and Yamaguchi S. An interactive prefetching proxy server for improvement of WWW latency. In: *INET'97: Seventh Annual Conference of the Internet Society*, Internet Society, Kuala Lumpur, Malaysia, 1997
[47] Rousskov A., Wessels D. High-performance benchmarking with Web Polygraph. *Software Practice and Experience*, 34:187–211, 2004

[48] Kegel, D. dkftpbench. http://www.kegel.com/dkftpbench/ (2000)
[49] Spinczyk O., Gal A., and Schröder-Preikschat W. AspectC++: An aspect-oriented extension to the C++ programming language. In: *Proceedings of the Fortieth International Conference on Tools Pacific*, Australian Computer Society, Sydney, Australia, pp. 53–60, 2002
[50] Almajali S. and Elrad T. Coupling availability and efficiency for aspect-oriented runtime weaving systems. In: *DAW'05: Proceeding of the 2nd Dynamic Aspects Workshop at AOSD*, Chicago, IL, pp. 47–56, 2005
[51] Engel M. and Freisleben, B. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In: *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, ACM, New York, pp. 51–62, 2005
[52] Douence R., Motelet O., and Südholt M. A formal definition of crosscuts. In: Yonezawa, A., Matsuoka, S. (eds.) *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, *LNCS vol. 2192*, Kyoto, Japan, Springer, Berlin Heidelberg New York, pp. 170–186, 2001
[53] Masuhara H. and Kawauchi K. Dataflow pointcut in aspect-oriented programming. In: Ohori, A. (ed.) *APLAS'03: First Asian Symposium on Programming Languages and Systems*, *LNCS vol. 2895*, Beijing, China, Springer, Berlin Heidelberg New York, pp. 105–121, 2003
[54] de Volder K. Aspect-oriented logic meta programming. In: Cointe, P. (ed.) *Meta-Level Architectures and Reflection, 2nd International Conference on Reflection*, *LNCS vol. 1616*, Saint Malo, France, Springer, Berlin Heidelberg New York, pp. 250–272, 1999
[55] Andrews J.H. Process-algebraic foundations of aspect-oriented programming. In: Yonezawa, A., Matsuoka, S. (eds.) *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, *LNCS vol. 2192*, Kyoto, Japan, Springer, Berlin Heidelberg New York, pp. 187–209, 2001
[56] Aßmann U. and Ludwig A. Aspect weaving with graph rewriting. In: Czarnecki, K., Eisenecker, U.W. (eds.) *GCSE: Generative Component-Based Software Engineering*, Erfurt, Germany, pp. 24–36, 1999
[57] Åberg R.A., Lawall J.L., Südholt M., Muller G., and Meur A.F.L. On the automatic evolution of an OS kernel using temporal logic and AOP. In: *ASE 2003: Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, IEEE Computer Society, Montreal, Canada, pp. 196–204, 2003
[58] Douence R., Südholt M. A model and a tool for event-based aspect-oriented programming (eaop). *Technical Report 02/11/INFO*, École des mines de Nantes (2002) French version published in Proc. of LMO'03, Hermes Sciences,
[59] Vanderperren W., Suvee D., Cibran M.A., and De Fraine B. Stateful aspects in JAsCo. In: *SC'05: Proc. of the 4th Int. Workshop on Software Composition*, *LNCS vol. 3628*, Springer, Berlin Heidelberg New York, 2005
[60] Allan C., Avgustinov P., Christensen A.S., et al. Adding trace matching with free variables to AspectJ. In: Gabriel, R.P. (ed.) *OOPSLA'05: ACM Conference on Object-Oriented Programming, Systems and Languages*, ACM, 2005
[61] Aspray W. John von Neumann's contributions to computing and computer science. *Annals of the History of Computing*, 11:189–195, 1989
[62] Luk C.K., Cohn R., Muth R., Patil H., Klauser A., Lowney G., Wallace S., Reddi V.J., and Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. In: *PLDI: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, ACM, Chicago, IL, pp. 190–200, 2005

[63] Hollingsworth J.K., Miller B.P., Goncalves M.J.R., Naim O., Xu, Z., and Zheng L. MDL: A language and compiler for dynamic program instrumentation. In: *PACT: Proceedings of the 6th Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, San Francisco, CA, USA, pp. 201–213, 1997

[64] Chiba S. Load-time structural reflection in Java. In: *ECOOP 2000: Sophia Antipolis and Cannes*, *LNCS vol. 1850*, France, Springer, Berlin Heidelberg New York, pp. 313–336, 2000

[65] Pawlak R., Seinturier L., Duchien L., and Florin G. JAC: A flexible solution for aspect-oriented programming in Java. In: *Proceedings of Reflection'01. LNCS vol. 2192*, Springer, Berlin Heidelberg New York, pp. 1–24, 2001

[66] Popovici A., Alonso G., and Gross T.R. Just-in-time aspects: Efficient dynamic weaving for Java. In: *AOSD: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, ACM, pp. 100–109, 2003

[67] Chiba S. and Nakagawa K. Josh: An open AspectJ-like language. In: Murphy, G.C., Lieberherr, K.J. (eds.) *AOSD: Proceedings of the Third International Conference on Aspect-Oriented Software Development*, ACM, pp. 102–111, 2004

[68] Suvée D., Vanderperren W., and Jonckers V. JasCo: An aspect-oriented approach tailored for component-based software development. In: Press, A. (ed.) *AOSD'03: Proc. of 2nd International Conference on Aspect-Oriented Software Development*, pp. 21–29, 2003

[69] Bockisch C., Haupt M., Mezini M., and Ostermann K. Virtual machine support for dynamic join points. In: *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, ACM, New York, pp. 83–92, 2004

[70] JBoss Inc: JBoss AOP. http://jboss.com/products/aop. (2005)

[71] Spring Framework: Spring AOP. http://www.springframework.org/. (2005)