

An expressive aspect language for system applications with Arachne

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

▶ **To cite this version:**

Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, et al.. An expressive aspect language for system applications with Arachne. 4th International Conference on Aspect-Oriented Software Development, Mar 2005, Chicago, United States. inria-00441355

HAL Id: inria-00441355
https://inria.hal.science/inria-00441355

Submitted on 15 Dec 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An expressive aspect language for system applications with Arachne

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

OBASCO project
École des Mines de Nantes/INRIA
4 rue Alfred Kastler
44307 Nantes Cedex 3, France
{douence,tfritz,nloriant,jmenaud,msegura,sudholt}@emn.fr

ABSTRACT

C applications, in particular those using operating system level services, frequently comprise multiple crosscutting concerns: network protocols and security are typical examples of such concerns. While these concerns can partially be addressed during design and implementation of an application, they frequently become an issue at runtime, *e.g.*, to avoid server downtime. A deployed network protocol might not be sufficiently efficient and may thus need to be replaced. Buffer overflows might be discovered that imply critical breaches in the security model of an application. A prefetching strategy may be required to enhance performance.

While aspect-oriented programming seems attractive in this context, none of the current aspect systems is expressive and efficient enough to address such concerns. This paper presents a new aspect system solving these problems. While efficiency considerations have played an important part in the design of the aspect language, the language allows aspects to be expressed more concisely than previous approaches. In particular, it allows aspect programmers to quantify over sequences of execution points as well as over alias accesses. We show how the former can be used to modularize the replacement of network protocols and the latter to prevent buffer overflows. We also present an implementation of the language as an extension of Arachne, a dynamic weaver for C applications. Finally, we show evaluations proving that Arachne is fast enough to extend high performance applications, such as the Squid web cache.

Keywords

expressive aspect language, dynamic weaving, system application

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD '05 Chicago, Illinois USA
Copyright 2005 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

1. INTRODUCTION

Real-world applications typically comprise multiple crosscutting concerns. This applies, in particular, to C-applications using operating system level services. We have examined three concerns which are typical for this domain in the context of a large application, the open source web cache "Squid." More concretely, we have considered translation of network protocols (which may be necessary for efficiency reasons), insertion of checks for buffer overflows (which are at the heart of many of today's security issues), and introduction of prefetching strategies within the cache (which can be used to enhance efficiency of the web cache). We have found that all of these concerns are scattered over large portions of the code of Squid.

Hence, these concerns are crosscutting in the sense of Aspect-Oriented Programming (AOP) [24] and aspects should therefore be a means of choice for their modularization. With respect to an AO treatment, such concerns have three important characteristics. First, the concerns must frequently be applied at runtime, *e.g.*, in order to rapidly fix a buffer overflow and thus prevent security breaches without incurring server downtime. A dynamic aspect weaver is therefore needed. Second, such concerns expose intricate relationships between execution points, *e.g.*, network protocols are most concisely expressed in terms of sequences of execution points, not individual ones. The aspect system must therefore support expressive means for the definition of aspects, in particular pointcuts. Third, efficiency is crucial in the application domain we consider.

To our knowledge, none of the current aspect systems for C meet these three requirements and is suitable for the modularization of such concerns. Moreover, requirements for dynamic weaving and efficiency often trade off with expressivity. Squid should be as efficient as possible and therefore exploits any suitable operating system and hardware particularity. It's code base is therefore difficult to understand and manipulate, thus hindering in particular modularization efforts. It is therefore not at all certain that the considered aspectization problem can be solved at all.

In this paper we propose a solution to the aspectization of such concerns of C-applications. More concretely, we provide three main contributions. First, we provide a new expressive aspect language featuring a construct for quantifi-

cation over sequences of execution points as well as over accesses to local aliases of global variables. We show how this aspect language permits concise expression of the considered concerns as aspects. Second, we present how the aspect language can be implemented efficiently through runtime weaving into binary code. Concretely, this is done by integrating the aspect language into our tool Arachne, a dynamic weaver for C-applications. Furthermore, we present how Arachne improves on our previous work μDyner. Finally, we give evidence that our approach also meets strong efficiency requirements by showing performance evaluations of aspect applications to Squid.

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 defines our aspect language. Section 4 describes its implementation within Arachne. Section 5 assesses the performance of our implementation. Section 6 describes related work. Section 7 concludes and suggests futures work.

2. MOTIVATIONS

Legacy C applications involve multiple crosscutting concerns. Many of them remain challenging, both in terms of expressiveness required to handle them properly in an aspect-oriented language and in terms of constraints posed on the weaver. This section describes three such concerns in C applications: switching the network protocol, buffer overflows and prefetching. The network protocol concern is typically scattered through the entire application. It is an issue when administrators discover at runtime that the retained protocol is not performant enough. Likewise the security threats posed by buffer overflows is a real concrete problem for administrators. While guarding all buffers against overflows might decrease performance considerably, administrators are left with no other option than accepting the trade-off between security and performance chosen at application's design time. Prefetching is another well-known crosscutting concern. Since prefetching aims at increasing performance, prefetching aspects make only sense with an efficient weaver. Yet, it is still difficult to modularize these three concerns in today aspect-oriented language. This section goes into detail of each of these examples. First, we describe the context in which the concerns arise before showing their crosscutting nature and finally explaining the lack in current aspect-oriented languages to handle them properly.

2.1 TCP to UDP protocol

HTTP was essentially designed as a file transfer protocol running on top of TCP, a connection-oriented protocol ensuring communication reliability. While the average Web page size does not exceed 8 Ko [4], the cost of retrieving a Web page is often dominated by data exchanged for control purposes of TCP rather than by the page content itself. This is not a new problem, many researches have already pointed out that TCP is not suitable for short-lived connections. While HTTP 1.1 has introduced persistent connections allowing a client to retrieve multiple pages from the same server through the same TCP connection, the number of simultaneous TCP connections is limited by operating systems. Servers have a strong incentive to close HTTP connections as soon as possible. Hence, despite the persistent connection mechanism, many studies conclude that TCP should be replaced by UDP to retrieve short pages

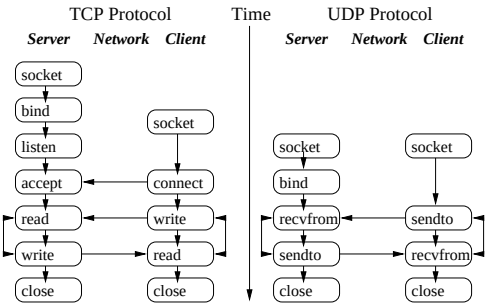


Figure 1: Typical usage of the TCP and UDP APIs.

[11, 30, 7]. Nevertheless the immediate performance improvements of this solution, the number of legacy Web applications has prevented a wide adoption. Typical legacy Web applications have to be stopped to switch the protocol. Stopping a Web cache will deprive a subnetwork from Internet connectivity. Stopping an e-commerce Web server means a loss of money. For a wide acceptance, a HTTP dialect using UDP as transport protocol should thus be deployable on demand at runtime.

In addition, replacing TCP by UDP in an application is relatively difficult. The choice of a transport protocol is usually based on standards believed to be ever-lasting and 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 on figure 1, the TCP API is built around a set of C functions to be invoked sequentially by the application. 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`). UDP uses similar but less functions. 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`). Therefore the problem is not only difficult because TCP-related function invocations are scattered in the source but because the relative order of each invocation is important in order to map it onto the appropriate function of the UDP API.

This example is typical to protocol based APIs. When such an API is used in an undisciplined way, it becomes quickly impossible to replace it by another one. Today, aspect-oriented systems lack an appropriate sequencing construct in their language. Moreover, many do not provide the ability to weave aspects dynamically.

2.2 Buffer overflows

In C, the size of an array is fixed at allocation time. According to ISO and ANSI standards [2], an invalid array access does not result in an immediate error but leads to an implementation-dependent behavior. Such behavior is increasingly exploited by hackers to circumvent security restrictions [37]. It is therefore crucial for C programmers to ensure every access to an array to be valid. On the other

hand, bound checking code is error prone: it is easy to forget to check an access and even when the access is checked, it is easy to compare the index locating the access with an inappropriate bound. Therefore, researchers have proposed to make compilers responsible for enforcing proper array access [23, 32]. The problem is that even the most efficient system (CRED [32]) slows down an application up to 130%. Moreover, most frequently used compilers like `gcc` do not support bound checking.

Today, administrators discovering a buffer overflow in production software are left with no other option than stopping the application and restarting a bug free version. This was the solution chosen when a buffer overflow was discovered in squid in [6]. While widely used, this solution suffers from three major drawbacks. First, it does not enforce continuous servicing since the service delivered by the application is not available during the update. Second, this solution entails an important information loss: an administrator has no means to learn whether the buffer overflow has been exploited by a hacker or not. Third, it misunderstands the performance trade-off, *i.e.* it is not necessary to check every array access, it is only necessary to perform enough checking to discourage hackers. Therefore, bound checking code should only run when an environment becomes hostile.

Bound checking code tends to crosscut the entire application. For example, properly written C functions accepting an array as an argument commonly take a second argument holding the array size: the first argument allows the function to access the array while the second ensures the correctness of every access it makes. In Squid, bound checking code can be found in any of the 104 "`.c`" files of its source code. On the 57635 lines composing these "`.c`" files, at least 485 deal with bound checking.

This problem fails to be handled properly in current aspect languages as they lack the ability to trigger advices upon access made through the alias of a variable. Again, many aspect-oriented systems offer only static weaving capabilities preventing the administrator to choose the trade-off security/performance suiting his needs.

2.3 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 start to fetch documents beforehand. Such prefetching schemes distinguish themselves from each other in the way 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 very likely that there is no universal perfect oracle [20]. A statically linked prefetching module is therefore inappropriate, but prefetching modules along with the necessary oracles should be loaded and unloaded on the fly.

Coady *et al.* have already pointed out the crosscutting nature of the prefetching concern in the FreeBSD operating system [13]. In our previous work considering the Squid Web cache, we reached a similar conclusion [33]. We have previously shown that this concern can be addressed with `cflow`-like constructs.

Despite potential performance improvements, prefetching also increases resource consumption (*e.g.* network prefetching consumes local storage and bandwidth). When the pressure on 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 essentially manages file descriptors, a resource only available in a limited quantity. A file descriptor is used between 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 descriptor currently in use. By comparing its value with the maximum number of file descriptors allowed by the operating system, it is possible to estimate that prefetching should be disabled or resumed.

For this consumption problem of file descriptors, the current practice of checking if prefetching should be disabled or not within the advice, is a bad practice that impedes both readability and maintainability. A mechanism is needed within the aspect language to restraint the advice execution at times where the pressure on resources is too high.

3. AN EXPRESSIVE ASPECT LANGUAGE FOR SYSTEM PROGRAMMING IN C

While AOP seems to be the 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, sequences of join points which obey a protocol and references to aliases which are local to a function definition.

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 aspectizing them using our language. We then define the join point model underlying our language precisely, followed by the definition of its syntax and informal semantics. Finally, we illustrate by example how the semantics of the aspect language can be formally defined in terms of a small-step operational semantics using the framework introduced in [15].

3.1 Example crosscutting concerns revisited

We now revisit the crosscutting concerns discussed in section 2 in order to show our language in action and give evidence for the claim that it allows such concerns to be concisely modularized.

The aspect shown in Fig. 2 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 (separated by `then`). In the example, the TCP protocol starts with a call to socket() with three constant arguments: AF_INET, SOCK_STREAM and 0. When such a call is matched, 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() with the same file descriptor fd as its first parameter is matched. In this case the values of the other parameters are bound to arguments address and length, and the original call is replaced by returnZero(). 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(),

```
seq( call(int socket(int, int, int)) && args(AF_INET, SOCK_STREAM, 0) && return(fd)
       then socket(AF_INET, SOCK_DGRAM, 0);
     call(int connect(int, struct socketaddr*, socklen_t)) && args(fd, address, length)
       then returnZero();    // where int returnZero() { return 0; }
     ( call(size_t read(int, void*, size_t)) && args(fd, readBuffer, readLength)
         then recvfrom(fd, readBuffer, readLength, 0, address, length);
       || call(size_t write(int, void*, size_t)) && args(fd, writeBuffer, writeLength)
         then sendto(fd, writeBuffer, writeLength, 0, address, length); ) *
     call(int close(int)) && args(fd) ; )
```

Figure 2: 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*)) )
```

Figure 3: An Aspect for Detecting Buffer Overflow

respectively. Note that sequences of such access are potentially translated (due to use of the repetition operator $*$). Finally, a call to close() on fd terminates the TCP protocol as well as 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.

The aspect in shown in Fig. 3 detects buffer overflows. The corresponding sequence starts when the function malloc() returns the buffer address which is then bound to 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 terminates when the memory is deallocated by calling free().

The aspect in Fig. 4 introduces prefetching in a web cache. The first `controlflow` phrase initializes prefetching when an HTTP response is built (clientBuildReply()) within the control flow of aclient 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 Join point model

A join point model defines the points in the execution of the base program to which pointcuts may refer. In our case, join points are defined by JP in the grammar shown in Fig. 5. A join point 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 $\overrightarrow{v_2}$.

```
JP     ::= callJP(val funId(val))
         | readGlobalJP(varId, val)
         | readJP(@, val)
         | writeGlobalJP(varId, val, size)
         | writeJP(@, val, size)
         | controlflowJP(funId, cfEnd)
         | controlflowstarJP(funId, cfEnd)

cfEnd  ::= callJP(val funId(val))
         | readGlobalJP(varId, val)
         | writeGlobalJP(varId, val, size)

val    ::= 0 | 1 | 2 | ...        // int
         | @0 | @1 | @2 | ...    // int*
         | ...  // values of other C types
```

Figure 5: Join point model

- A read access which 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 access to a global variable) occurs within the body of function $funId_n$. Such a join point 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 join point requires a call to $funId_{i+1}$ within the control flow of (*i.e.*, not necessarily in the body of) $funId_i$.

Two features of this join point 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 an access

```
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);
```

Figure 4: An Aspect for Prefetching

to a global variable and matching of control flow join points can be done using an atomic test on the implementation level.

3.3 Pointcuts

We are now ready to present a pointcut language (see Fig. 6) that provides constructs to match individual join points.

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

A call pointcut *PCall* selects all function call join points `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 be predicated on a boolean condition using the `if`-constructor.

A global access pointcut *PAccGlobal* selects either all read join points `readGlobalJP`(varId, val) or all write join points `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* of the form `controlflow`($PCallSig_1, ...,\ PCallSig_n,\ PCfEnd$) matches all join points of the form `controlflowJP`($\text{funId}_1, ...,\ \text{funId}_n,$ cfEnd), 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.

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

$$
\begin{array}{lcl}
Asp & ::= & AspPrim\ \ [\ \&\&\ \texttt{until(}\ AspPrim\ \texttt{)}\ ] \\
 & | & AspSeq\ \ [\ \&\&\ \texttt{until(}\ AspPrim\ \texttt{)}\ ] \\
 & & \\
AspPrim & ::= & PPrim\ Advice \\
 & & \\
AspSeq & ::= & \texttt{seq(}\ AspPrim \\
 & & \quad AspSeqElt\ [\ *\ ] \\
 & & \quad ..., \\
 & & \quad AspSeqElt\ [\ *\ ] \\
 & & \quad AspSeqElt\ \texttt{)} \\
AspSeqElt & ::= & AspPrim \\
 & | & PAcc\ Advice \\
 & | & (AspSeqElt\ ||\ AspSeqElt) \\
 & & \\
Advice & ::= & [\ \texttt{then}\ funId(\overrightarrow{pattern})\ ]\ ;
\end{array}
$$

Figure 7: Aspect language

3.4 Expressive Aspect Language

The aspect language we propose is defined in Fig. 7. Aspects *Asp* are either primitive *AspPrim*, or sequences of primitive aspects *AspSeq*.

A primitive aspect *AspPrim* combines a primitive pointcut with an advice that will be applied to all join points selected by the pointcut. If the primitive pointcut has the form $p_1 \ || \ p_2$, then all variables used in the advice have to be bound in both, p_1 and p_2.

An advice (*Advice*) is a C function call that replaces a join point in the base program execution (similarly to `around` in AspectJ). It must have the same return type as the join point it replaces: 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, hence no C function call), the original join point is executed ("instead of" the original join point). The original join point can be skipped by calling an empty C function.

A sequence aspect is composed of a sequence of primitive aspects. A sequence starts when the first primitive aspect matches. Then the second primitive aspect becomes active instead of the first one. When it matches, the third aspect becomes active instead of the second one. And so on, until the last primitive aspect in the sequence. All but the first

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

Figure 6: Pointcut language

$$
\begin{array}{lcll}
A & ::= & \mu a.A & ;\; recursive\; definition \;\; (a \in \mathcal{R}ec) \\
 & | & \text{C} \rhd \text{I};\; A & ;\; prefixing \\
 & | & \text{C} \rhd \text{I};\; a & ;\; end\; of\; sequence \;\; (a \in \mathcal{R}ec) \\
 & | & \text{C} \rhd \text{I};\; \text{STOP} & ;\; halting\; aspect \\
 & | & A'_1 \;\Box\; A'_2 & ;\; choice \\
 & | & A_1 \;||\; A_2 & ;\; parallelism \\
 & & & \\
A' & ::= & \mu a.A' & \\
 & | & \text{C} \rhd \text{I};\; A & \\
 & | & \text{C} \rhd \text{I};\; a & \\
 & | & \text{C} \rhd \text{I};\; \text{STOP} & \\
 & | & A'_1 \;\Box\; A'_2 &
\end{array}
$$

Figure 8: Tiny aspect language

and last primitive aspects can be repeated zero or multiple times by using $*$: in this case, the primitive aspect is active as long as the following one in the sequence does not match. Branching, *i.e.*, a logical 'or' between two primitive aspects, can be introduced in a sequence by the operator ||. An element of the sequence can also match a global variable of the base program and accesses to its local aliases, as soon as its address is known (*i.e.*, a previous primitive pointcut has already bound its address to a pointcut variable). Hence, an aspect matching accesses cannot start a sequence. Every join point matching the first primitive pointcut of a sequence starts a new instance of the sequence. The different instances are matched in parallel.

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

3.5 Towards a 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 could be formally defined by translating one of the example aspects into a but formal aspect language A extending that used in the formal framework of [15].

The original formal language must be extended in order to deal with halting aspects, an unbounded number of sequential aspects and arbitrary join point predicates. The grammar of the extension, our tiny aspect language, A is defined in Figure 8. In this language, C is a join point predicate (similar to our pointcut language) expressed as a conjunction of a term pattern and possibly a CLP($\mathcal{R}$) expression [21].

An aspect is either:

- A recursive definition.
- A sequence formed using the prefix operation $C \rhd I;\; X$, where X is an aspect or a recursion variable and I a piece of code (*i.e.*, an advice).
- A choice construction $A_1 \;\Box\; A_2$ which chooses the first aspect that matches a join point (the other is thrown away). If both match the same join point, A_1 is chosen.
- A parallel composition of two aspects $A_1 \;||\; A_2$ that cannot occur in choice construction.
- A halting aspect STOP.

The semantics of the protocol translation aspect (from TCP to UDP) is given in Fig. 9. 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 join point has been matched and continue to match the rest of the sequence in progress. The repetition operator $*$ is translated into recursion on variable the a_2. The branching operator || is translated into the choice operator $\Box$. Finally, the last primitive aspect of the sequence occurs as the first aspect of a choice as it has priority over the call join points *read* and *write* because of the $*$. Note that we use pattern matching in A and that the first occurrence of a variable (*i.e.*, its definition as opposed to its uses) is marked by an overbar.

Note that formal definitions such as that of the protocol translation aspect precisely define several important issues, in particular, when new instances of the sequence aspect are created, and disambiguation of potentially non-

μa_1. callJP($\overline{\text{fd}}$ socket(AF_INET, $\overline{\text{SOCK_STREAM}}$, 0)) $\triangleright$ socket(AF_INET, SOCK_DGRAM, 0);
a_1 || (callJP(a connect(fd, $\overline{\text{address}}$, $\overline{\text{length}}$)) $\triangleright$ returnZero();
μa_2. callJP(b close(fd)) $\triangleright$ skip; STOP
$\square$ callJP(c read(fd, $\overline{\text{readBuffer}}$, $\overline{\text{readLength}}$)) $\triangleright$ recvfrom(fd, readBuffer, readLength, 0, address, length); a_2
$\square$ callJP(d write(fd, $\overline{\text{writeBuffer}}$, $\overline{\text{writeLength}}$)) $\triangleright$ recvfrom(fd, writeBuffer, writeLength, 0, address, length); a_2

Figure 9: Definition of the protocol translation using the tiny aspect language

deterministic situations, *e.g.*, when two pointcuts of consecutive primitive aspects in the sequence match at the same time.

4. DYNAMIC WEAVING WITH ARACHNE

Arachne is built around two tools, 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 [33]. These techniques allow to rewrite the binary code of executable files on the fly *i.e.*without pausing the base program, as long as these files conform to the mapping defined by the Unix standard [36] between the C language and x86 assembly language. Arachne's implementation is structured as an open framework that allows to experiment with new kinds of join points and pointcut constructs. Another important difference between Arachne and μDyner is, that μDyner requires a compile time preparation of the base program [1], whereas Arachne does not. Hence Arachne is totally transparent for the base program while μDyner is not.

4.1 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 of the base program 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.

4.1.1 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* [17]. Once the aspects have been translated into C, the Arachne aspect compiler uses a legacy C compiler to generate a dynamically linked library (DLL) holding the compiled aspects.

4.1.2 The Arachne weaving process

From a user viewpoint, the Arachne `weave` and `deweave` command line offers the same syntax than 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 the 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 address space of the process through standard techniques [12].

The instrumentation kernel is transparent for the base program as the latter can not 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 Arachnes' 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.

4.2 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

[1] to insert an `nop` assembly instruction at the begin of each function

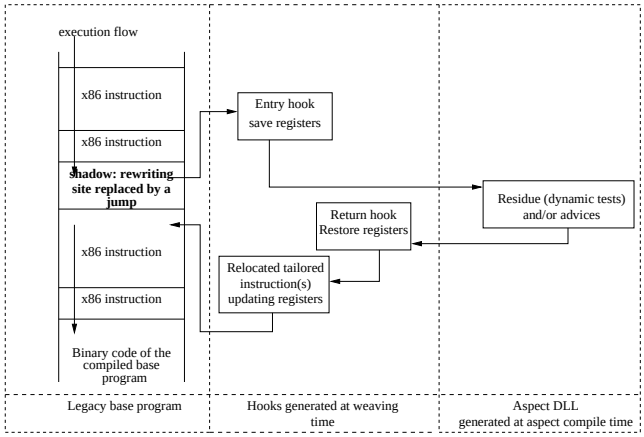


Figure 10: Generic hook operations.

Arachne's main dependencies to the underlying hardware architecture. In general, rewriting strategies need to collect information about the base program. These information typically consist of the addresses of the different shadows, their size, 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 runtime. 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 these information can be costly to retrieve, `Arachne` collects and stores it into meta-information 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.

4.2.1 *Strategies for* `call`, `readGlobal` *and* `writeGlobal`

In Arachne, `call`, `readGlobal` and `writeGlobal` allow an advice to be triggered upon a function call, aread on a global variable or a write respectively. While the implementation 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[2] 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 figure 10, 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 join point shadow are executed to perform the appropriate side effects on the processor registers. This is accomplished by relocating the instructions found at the join point shadow. Relocating the instructions makes the rewriting strategies handling read and write access to global variable independent from the instruction generated by the compiler to perform the access [3]. The limited number of x86 instructions used to invoke a function allows Arachne's rewriting strategy to exploit more efficient, relocation free, hooks.

4.2.2 *Strategies for* `controlflow` *and* `controlflowstar`

Every time a C function is called, the Linux runtime creates an activation record on the call stack [36]. Like μDyner, Arachne's implementation of the rewriting strategy for `controlflow` uses the most deeply imbricated 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 runtime. An appropriate usage of hashtables 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 n pointer comparisons. However, the residue worst case runtime for the indirect control flow operator `controlflowstar` that allows for not directly nested functions, is proportional to the base program stack depth.

4.2.3 *Strategies for* `read` *and* `write`

`read` and `write` are new join points 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 [19]. 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.

4.2.4 *Strategies for* `seq`

[2]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.

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

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 join point 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 [4]. 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 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.

4.3 Arachne limitations

Aggressive optimizations of the base program might prevent Arachne to seamlessly weave 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 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 supports two of the three optimization levels proposed by `gcc`. Stripping that removes linking information and 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 levels of optimizations provided by `gcc`.

5. 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. We then study the overhead of extending Squid with a prefetching policy. This case study shows 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.

5.1 Evaluation of Arachne's 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 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 formulate our measurements as a ratio between these two durations in order 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[5], our benchmarking infrastructure relies on the `rdtsc` assembly instruction [19]. 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[6]. The pipeline mechanism in the Pentium 4 processor entails that the speed of a piece of assembly code depends from 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, that 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, aspect-less, 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. For similar reasons, a `seq` of three invocations of such empty functions is only 3.2 time slower than the direct, aspect-less, three successive functions calls. Compared to the point-

[4]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.

[5]The POSIX `gettimeofday` is precise up to one millisecond.

[6]Strictly speaking, it forces only the preceding instructions involved in memory operations to be fully executed before going on.

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

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

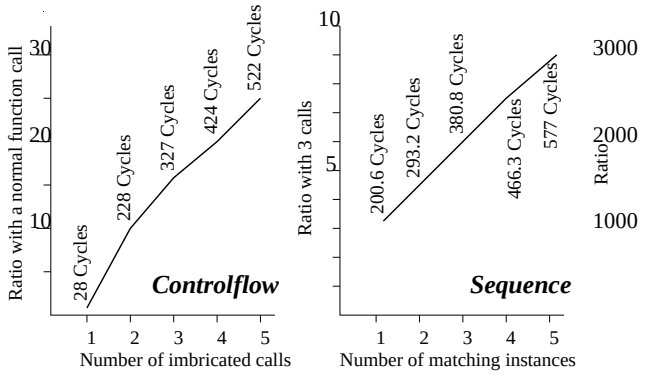


Figure 11: controlflow, seq, and read pointcut's performances

cuts 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 and restore the processor state to ensure the execution coherency, as advices are packaged as regular C functions (see also 4.2.1). It is therefore not surprising that a global variable `readGlobal` appears as being 2762 times slower than a direct, aspect-less global variable read. `read` performance can be accounted in the same way: in the absence of aspect, local variables are access 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 4.2.3. 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 runtime of these pointcuts grows linearly with the number of execution points they refer to and with the number of matching instances. Figure 11 summarizes a few experimental results for `controlflow` and `seq` proving these points.

5.2 Case Study on a real application

Since, depending on the aspect construct used, interpreting the base program with aspects can slow it down by a factor ranging between 1.3 and 9729, we studied Arachne's performance on a real world application, the Web cache Squid. We extended Squid with a prefetching policy [10]. As described in section 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 implementation using standard Web cache performance indicators: throughput, response time and hit ratio.

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

Table 2: Performances comparison between manual modification and Arachne, for prefechting policy integration in Squid

Obtaining access traces adequate to study a Web cache performance is difficult. The trace must be long enough to fill the cache. Due to 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 [31]. Polygraph is a benchmarking tool developed by the Squid team and featuring 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 [31]. Table 2 shows results of our simulation. Measures have been made during the two request peaks. The hit time and the miss time, time needed to deliver a document present, respectively not present, in the cache are very similar. It shows that differences are imperceptible between the version of Squid extended by Arachne and the one extended manually (less than 1%). Hence, even if Arachne's aspect language constructs might seem high, they are largely amortized in real applications.

6. RELATED WORK

Our work is directly related to other aspect weavers for C, approaches for expressive aspect languages, and dynamic weaving, in particular for C. In this section, we consider related work in each of these fields in turn.

Apart from μDynerand Arachne, there are few *aspect weavers for C* (or even C-like languages); some noteworthy exceptions are AspectC [13] (no available implementation), AspectC++ [34] and AspectC# [25]. All of these rely on source-code transformation and thus cannot be used for the

application of aspects to running C applications as necessitated by the applications we consider. Furthermore, none of these systems provides explicit support for aspects over sequences of join points.

There is quite a large body of work now on the notion of *expressive aspect languages* where "more expressive" typically compares to *w.r.t.* AspectJ's pointcut and advice models. Our work has been inspired by Event-based AOP [16], which aims at the definition of pointcuts in terms of arbitrary relations between events. Nevertheless, many other approaches to expressive aspect languages exist: *e.g.*, dataflow relations [27], logic programming [14], process algebras [3], graphs [5], and temporal logics [1], 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.

Dynamic weaving is commonly realized in Java through preprocessing at load-time like [8] or through the JVM Debugging Interface [29, 9]. These tools rely on bytecode rewriting techniques, have typically limited expressivity (some do not support field accesses) and incur a huge performance overhead. Dynamic weaving through modification at runtime is found infrequently for compiled languages. An exception for Java is JasCo [22] whose most recent version (0.7) supports dynamic weaving through the new instrumentation API of Java 5.

Many instrumentation techniques have been proposed to *rewrite binary code on the fly*. In these approaches, difficulty sources 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. Hence many approaches work on an intermediate representation of the binary code and source language [35]. Producing this representation first and then regenerating the appropriate binary executable code has proven to be costly both in terms of memory consumption and in CPU time.

A few other approaches have considered a direct rewriting of the binary code at runtime. Dyninst [18], Pin [26] and dynamic probes [28] allow programmers to modify any binary instruction belonging to an executable. Dyninst however relies on the Unix debugging API: `ptrace`. `ptrace` allows a third party process to read and write the base program memory. It is however highly inefficient: before using `ptrace`, the third party process has to suspend the execution of the base program and resume its execution afterwards. In comparison, Arachne uses `ptrace` at most once, to inject its kernel DLL into the base program process. In addition, Dyninst does not free the programmer from dealing with low level details. For example, it seems difficult to trigger an advice execution upon a variable access with Dyninst: the translation from the variable identifier to an effective address is left to the user. Worse, Dyninst does not grant that the manipulation of the binary instructions it performs will succeed. Dyninst uses an instrumentation strategy where several adjacent instructions are relocated. This is unsafe as one of the relocated instructions can be the target of branching instructions. In comparison, Arachne join point model has been carefully chosen to avoid these kind of issues; if an aspect can be compiled with Arachne, it can always be woven.

7. CONCLUSION AND FUTURE WORK

In this paper we have discussed three different crosscutting concerns which are typical for C-applications using OS-level services and which frequently need to be applied at runtime. We have motivated that such concerns can be expressed as aspects and have defined a suitable aspect language. This language is more expressive than those used in other aspect weavers for C in that it provides support for aspects defined over sequences of execution points as well as for variable aliases. We have presented an integration of this language into Arachne, a weaver for runtime weaving of aspects in C applications. Finally, we have provided evidence that the integration is efficient enough to apply such aspects dynamically to high-performance applications, in particular the web cache "squid."

As future work, we intend to investigate the suitability of the proposed aspect language for other C-applications. We also intend to investigate Arachne extension to the C++ language. Indeed, object-oriented programming heavily uses protocol-based interfaces collaboration (hence sequence aspects). Along with its open architecture, extending Arachne to support C++, will pave the way to a relatively language independent aspect and weaving infrastructure. Finally, Arachne's toolbox should be extended with support for aspect interactions (*e.g.*, analyses and composition operators).

8. REFERENCES

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. L. Meur. On the automatic evolution of an os kernel using temporal logic and AOP. In *Proceedings of Automated Software Engineering (ASE'03)*, pages 196–204. IEEE, 2003.

[2] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1999.

[3] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*. Springer Verlag, Sept. 2001.

[4] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, May 2000.

[5] U. Aßmann and A. Ludwig. Aspect weaving by graph rewriting. In U. W. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering (GCSE)*, Erfurt, Oct. 1999.

[6] CERT - Carnegie Mellon University. Vulnerability note vu#613459, Feb. 2002. published on line: `http://www.kb.cert.org/vuls/id/613459`.

[7] H. Chen and P. Mohapatra. Catp: A context-aware transportation protocol for http. In *International Workshop on New Advances in Web Servers and Proxy Technologies Held with ICDCS*, 2003.

[8] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In *Proceedings of the third international conference on Aspect-oriented software development*, pages 102–111. ACM Press, Mar. 2004.

[9] S. Chiba, Y. Sato, and M. Tatsubori. Using hotswap for implementing dynamic AOP systems. In

ECOOP'03 Workshop on Advancing the State of the Art in Runtime Inspection (ASARTI), 2003.

[10] K.-I. Chinen and S. Yamaguchi. An interactive prefetching proxy server for improvement of WWW latency. In *Seventh Annual Conference of the Internet Society (INET'97)*, Kuala Lumpur, June 1997.

[11] I. Cidon, A. Gupta, R. Rom, and C. Schuba. Hybrid tcp-udp transport for web traffic. In *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference (IPCCC'99)*, pages 177–184, Feb. 1990.

[12] S. Clowes. Injectso: Modifying and spying on running processes under linux. In *Black hat briefings*, 2001.

[13] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of Path-Specific customization in operating system code. In V. Gruhn, editor, *Proc. of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 88–98, New York, Sept. 10–14 2001. ACM Press.

[14] K. de Volder. Aspect-oriented logic meta programming. In P. Cointe, editor, *Meta-Level Architectures and Reflection, 2nd International Conference on Reflection*, volume 1616 of *LNCS*, pages 250–272. Springer Verlag, 1999.

[15] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LLNCS*, pages 173–188. Springer-Verlag, Oct. 2002.

[16] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 170–186. Springer Verlag, Sept. 2001.

[17] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press, 2004.

[18] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng. MDL: A language and compiler for dynamic program instrumentation. In *IEEE Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 201–213, Nov. 1997.

[19] Intel Corportation. *IA-32 Intel Architecture Software Developer's Manual*. Intel Corportation, 2001.

[20] V. Issarny, M. Banâtre, B. Charpiot, and J.-M. Menaud. Quality of service and electronic newspaper: The Etel solution. *Lecture Notes in Computer Science*, 1752:472–496, 2000.

[21] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The clp(r) language and system. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, 1992.

[22] JasCo home page. `http://ssel.vub.ac.be/jasco/`.

[23] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In M. Kamkar, editor, *Proceedings of the Third International Workshop on Automatic Debugging*, volume 2, pages 13–26, May 1997.

[24] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Jyväskylä, Finland, June 1997.

[25] H. Kim. AspectC#: An aosd implementation for C#. Master's thesis, Trinity College, 2002.

[26] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. Ispike: A post-link optimizer for the intel itanium architecture. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 15–27. IEEE Computer Society, Washington, DC, USA, Sept. 2004.

[27] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *First Asian Symposium on Programming Languages and Systems (APLAS'03)*, 2003.

[28] R. J. Moore. Dynamic probes and generalised kernel hooks interface for Linux. In USENIX, editor, *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10–14, 2000, Atlanta, Georgia, USA*, Berkeley, CA, USA, 2000. USENIX.

[29] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109, Boston, Massachusetts, Mar. 2003. ACM Press.

[30] M. Rabinovich and H. Wang. DHTTP: An efficient and cache-friendly transfer protocol for web traffic. In *INFOCOM*, pages 1597–1606, 2001.

[31] A. Rousskov and D. Wessels. High-performance benchmarking with Web Polygraph. *Software Practice and Experience*, 34(2):187–211, Feb. 2004.

[32] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*. Internet Society, Feb. 2004.

[33] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119, Boston, MA, USA, Mar. 2003. ACM Press.

[34] O. Spinczyk, A. Gal, and W. Schroeder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.

[35] A. Srivastava and A. Edwards. Vulcan: Binary transformation in a distributed environment. Microsoft Research Tech. Rpt. MSR-TR-2001-50, 2001.

[36] U. S. L. System Unix. *System V Application Binary Interface Intel 386 Architecture Processor Supplement*. Prentice Hall Trade, 1994.

[37] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, February 2003.