Generalized Dynamic Probes for the Linux Kernel and Applications with Arachne

Nicolas Loriant, Jean-Marc Menaud

GENERALIZED DYNAMIC PROBES FOR THE LINUX KERNEL AND APPLICATIONS WITH ARACHNE

Nicolas Loriant Jean-Marc Menaud
Obasco Group, EMN-INRIA, LINA
4, rue Alfred Kastler
44307 Nantes, France
{Nicolas.Loriant, Jean-Marc.Menaud}@emn.fr

ABSTRACT

Finding the root cause of bugs and performance problems in large applications is a difficult task. The main reason of this difficulty is that the comprehension of such applications crosscuts the boundaries of a single process, indeed the concurrent nature of large applications requires insight of multiple threads and process and even sometimes of the kernel. In the meantime, most existing tools lacks support for simultaneous kernel and applications analysis.

In this paper, we present Arachne, a tool for runtime analysis of complex applications. While efficiency considerations have played an important role in the design of Arachne, it allows safe and runtime injection of probes inside the Linux kernel and user space applications on both function calls and variable access. It features an Aspect-Oriented language that allows to access context of execution and to compose primitive probes (for example sequence of function calls). We show how Arachne allows to easily analyze problems such as race conditions which involves complex interactions between multiple process. And finally, we show Arachne is fast enough to analyze high performance applications such as the Squid web cache.

KEYWORDS

Aspect-Oriented Programming, System analysis and debugging.

1. INTRODUCTION

Finding the root cause of bugs and performance problems in large applications is a difficult task. One of the main reasons of this difficulty is the concurrent nature of such applications. In order to understand the execution of concurrent applications, developers have to analyze the competitive execution of multiple threads and process. Because it schedules process execution, developers must also take an insight in the kernel behavior. For example, in order to prevent race conditions on files, one solution consists in monitoring file operations in the kernel. Tools such as Systemtap, Kprobes, and DTrace allows developers to dynamically insert probes in the Linux kernel. Nevertheless, file operations are very common during execution of the kernel leading to a large amount of data to analyze.

To render the analysis of large applications practicable, it is necessary to reduce the amount of information to monitor. That is possible when combining knowledge on both the application and kernel execution. For example, when monitoring file operations to prevent race conditions, not all file operations done by the applications must be watched. Using that information, it could be possible to reduce the amount of data to analyze. A similar issue arise in the Squid web cache, both user space and kernel space monitoring is necessary to report users disk usage. Nevertheless to do so, it is necessary to combine information gathered from both the kernel and from applications.

The analysis and evolution of complex applications emphasis the need for a dynamic instrumentation tool for both the kernel and applications. Moreover, the complex nature of large systems stresses the necessity of an expressive language to express the interactions of probes. In this paper we propose a solution to the analysis of large applications. More concretely, we provide the following contributions. First, we provide an expressive C-like language to concisely describes pattern of interactions between multiple execution queues. We show how this language allows to easily analyze issues such as race conditions and the Squid disk usage accounting issue mentioned earlier. Second, we present how that language can be implemented efficiently

through dynamic code injection in both kernel and user space. Finally, we give evidence that our approaches also meets strong efficiency requirements by showing performance evaluations.

The paper is structured as follows. Section 2 illustrates the Arachne aspect language and its concurrent extension through the race conditions and Squid examples. Section 3 presents the Arachne implementation. Section 4 discusses performance evaluations through micro benchmarks and the Squid disk usage accounting example. Section 5 compares our solution to related work before Section 6 concludes.

2. ARACHNE'S ASPECT LANGUAGE

In order to analyze the behavior of complex applications, developers need a language to express what events in its execution, it is necessary to monitor and how to treat those information (*e.g.*, a function call and its parameters). Hence, Aspect-Orientation appears as an adequate choice to tackle this issue. Indeed, Aspect-Oriented languages associate pointcuts and advices. A pointcut describes an event in the execution of a program, such as a function call or a read access to a variable, upon which the advice is to be executed.

In this section, we describes how we extended our aspect-oriented system, Arachne. Arachne's language is especially appropriate for system programming in C, indeed, it allows to concisely express system issues as we have shown in “An Expressive Aspect Language for System Programming with Arachne” (Douence et al.) and in “A Reflexive Extension to Arachne's Aspect Language” and in “Server Protection through Dynamic Patching” (Loriant et al.). In order to alleviate this, we first present the examples of race conditions and disk usage accounting in Squid in the Arachne language, then we'll present the language itself.

2.1 Race conditions and disk usage accounting in Squid

The aspect shown accounts disk usage from clients in Squid. This is a sequence aspect, it matches every calls to the function “**clientProcessRequest**” in Squid and stores both the process id where the call happened and IP address from which the request originated. Then, upon call of “**vfs_read**” in the kernel and originating from the process of which we stored the id, the advice call “**addClientReadDiskAccess**” to account the disk read to the client IP stored earlier.

```
seq(A: call(void clientProcessRequest(struct clientHttpRequest*)) && args(http)
        && bind (pid, GET_PID) && bind (IP_CLIENT, http->request->client_addr));
    K: call(ssize_t vfs_read(struct file*, char*, size_t, loff_t*)) && args(file, buf, count, pos)
        && if (!isSocket(file) && !isPipe(file)) && if (pid == current->pid)
        then addClientReadDiskAccess(current->pid, IP_CLIENT, size);
)
```

The second example presents a sequence aspect to solve the race condition problem in Squid. The first and last step of the sequence matches calls in Squid to functions “**stat**” and “**open**” where the target file is the same. If in between an operation occurred in the kernel on that file (but not originating from Squid), then an alarm **“race condition”** is reported when Squid tries to open that file.

```
seq(A: call(int stat(const char*, struct stat*)) && args(path_stat, buf)
         && bind (pid, GET_PID));
    K: call (ssize_t vfs_op(struct file*, char*, size_t, loff_t*)) && args(file, buf, count, pos)
         && if (file == pathtofile(path_stat)) && if (current->pid != pid));
    A: call(int open(const char*, int)) && args(path_open, flags)
         && if (path_open == path_stat)
         then alarm(“race condition”);
)
```

2.2 The Arachne language

In an Aspect-Oriented language, a joinpoint model defines the points in the execution of a program where advices can be executed. The language associates a joinpoint with an advice, the code to be executed upon matching of the joinpoint. In the Arachne language, advices are blocks of C code.The Arachne language features five types of joinpoints.

2.2.1 Function calls

Aspects on function calls matches every call of a given function (*e.g.*, “foo”). Its parameters and return value can be bound to variables to be accessed in the aspect advice. Conditions can be given for the aspect to match only if parameters and return values have particular values. Moreover, the developers can decide to execute the advice before, in-place, or upon return of the original function call.

```
call(void foo(int*)) && args(a) && if (a == 0) then {printf(“null argument\n”); exit(0);};
```

2.2.2 Read access

Aspect on read access matches every read access on a given global variable or local alias. The Arachne language distinguishes the two for performance reasons, indeed, a global variable access is an order of magnitude slower that an local alias access. In the advice, the developer can access the current value of the variable. He may also modify the variable content and execute any additional code.

2.2.3 Write access

Aspect on write access matches every write access on a given global variable or local alias. Again the two are distinguished for performance reasons. In the advice of a write access pointcut, the developer can access the value before the write and the value to be written. Again, the developer can write code to be executed before, in-place or after the access.

2.2.4 Control Flow

In control flow aspects, developers provide a suite of function names (*e.g.*, **zoo, foo**) terminated with another function name or a global variable access (read or write) (*e.g.*, **bar**). The latter is matched whenever it occurs imbricated in the suite of functions (*e.g.*, **bar** called by **foo** itself called by **zoo**). Control flow aspects comes in two variants; The first one, where the stack of functions must strictly matches the call stack and the second one where the matching is not strict, *e.g.*, **bar** called by **foo** itself called by **homer** itself called by **zoo** would match the aspect. Context information and action to be pursue in the advice of a control flow aspect depends of its last element (function call or variable access)

```
controlflow(call(void zoo(void)), call(void foo(int)),
        call(void bar(int)) && args(a) && if (a > 0) then {bar(--a);})
```

2.2.5 Sequence

A sequence aspect is composed of a sequence of primitive aspects (function call, variable aspect, control flow). 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 and last primitive aspects can be repeated zero or multiple times: in this case, the primitive aspect is active as long as the following one in the sequence does not match. 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.

2.3 Concurrent extension

We extended the Arachne language to include concurrent features. Concretely, we introduced means for the developers to place aspects in a single or multiple applications or in the kernel, and to share an aspect upon multiple applications and the kernel. To do so, we introduced: first, a notation to place aspect or part of aspect in the kernel, a given application or groups of applications, second, means to manipulate groups of applications in order to activate and deactivate aspects.

2.3.1 Aspect placement

In order to place aspects in applications or in the kernel, we introduced a placement notation for aspects. Aspects and primitive aspects can be preceded by a group (of applications) on which to inject. Hence, an

aspect can be placed on multiple applications, for example, primitive aspects of a sequence are not necessarily placed on the same applications. A group can be a named group (created with the **group** keyword) or a comma separated list of groups. A special constant group is the “K” group which correspond to the kernel. Apart from the “K” group, groups do not directly reference applications or process. The assignment of process ids to groups is made during the aspect injection, hence, aspects (even compiled) are independent from applications to ensure reusability.

2.3.2 Group manipulation

During its lifetime, a program may create threads and process, hence it is necessary to provide the ability to modify the placement of aspects. Our extension allows one to manipulate groups in order to enlarge or shrink process and threads on which aspect applies. Our extension of Arachne's aspect language allows developers to manipulate group inside aspects' advices using two primitives for group manipulation: **add** and **remove**.

3. DYNAMIC CODE INJECTION 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 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 between the C language and x86 assembly language. Arachne does not require a compile time preparation of the base program, hence, Arachne is totally transparent for the base program.

The Arachne 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. **weave** also associate programs (identified by pids through the command line argument of weave) with the group names used in the aspect file. Upon reception of a weaving request, the instrumentation kernel selects the appropriate rewriting strategies and instruments the base program (and/or the kernel) accordingly. It finally modifies the binary code of the base program to actually tie the aspects to the base program.

3.1 The Arachne aspect compilation process

The aspect compilation scheme is relatively straightforward. First, the aspect file is split into deployment units according to placement information. Then, the compiler 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. Once the aspects have been translated into C, the Arachne aspect compiler uses a legacy C compiler to generate shared libraries and/or kernel modules holding the compiled aspects.

3.2 The Arachne weaving process

The Arachne **weave** command line takes two arguments. The first is an aspect file name to wove and the second is a list that initializes the group of process declared in the aspect file with pids of process to weave in. When Arachne's **weave** receives a request to weave an aspect in a process or in the kernel and it does not contain the Arachne instrumentation kernel, **weave** loads the instrumentation kernel in the address space of the process (or the Linux kernel) through standard techniques described by Clowes in “modifying and spying on running process under Linux” or simply using loadable module in the case of the kernel.

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 that handles the different weaving requests. The instrumentation kernel allocates memory by using side effect free allocation routines. 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 would result in an application crash. After loading an aspect, the instrumentation kernel rewrites the binary code of the base program using the rewriting strategies described below.

3.3 The Arachne 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. These information typically consist of the addresses of the different rewriting locations, their size, the symbol (*i.e.* function or global variable name) they manipulate 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 and kernel files (**System.map**). However because these information can be costly to retrieve, Arachne collects and stores it into meta-information shared libraries (a loadable module in the case of the kernel) that behave as a cache. Because aspects have common data, typically a sequence of function calls in different applications need to share a common state, when an aspect file involves aspect sharing information among multiple applications and/or the kernel, Arachne places those data into a shared memory segment that is accessed concurrently by all aspects. To implement the aspect language, Arachne provides a set of eight rewriting strategies that might eventually use each other. For the sake of conciseness, the rest of this section omits control flow and sequence which are built on top of function calls and variable access aspects.

3.3.1 Function calls and global variable access

In Arachne, an advice may be triggered upon a function call, a read on a global variable or a write respectively. Arachne implements the strategy for call by rewriting function invocations found in the base program. 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 aspects do not involve any dynamic tests. The sole rewriting of the binary base program code is enough to trigger advice 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. Hooks contain a few assembly instructions that save and restore the appropriate registers before and after an advice execution. A generic approach is to have hooks save the whole set of registers, then execute the appropriate advice code before restoring the whole set of registers; finally the instructions found at the join point are executed to perform the appropriate side effects on the processor registers. This is accomplished by relocating the instructions found at the joinpoint. 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 (there exists more than 250 x86 mnemonics manipulating global variables corresponding to more that one thousand opcodes). The limited number of x86 instructions used to invoke a function allows Arachne's rewriting strategy to exploit efficient, relocation free, hooks.

3.3.2 Local alias access

Their implementation rely on a page memory protection as allowed by the Linux operating system interface (*i.e.* **mprotect**) and the Intel processor specifications. Read or write pointcut triggers a relocation of 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. PERFORMANCE EVALUATIONS

Dynamic code injection for large system analyzing and debugging will be used if it expressive enough and if its overhead is low enough for the task at hand. The purpose of this section is to study the Arachne system's performance. We first present the performance of each Arachne language construct (function calls, sequence etc) and compare it to equivalent C constructs. We then study the overhead of the disk usage counter system extension for the Squid web cache that requires code injection both in user space and kernel space. This case study shows that even if some Arachne language constructs might seems more costly than similar C constructs, this cost is largely amortized in real world applications.

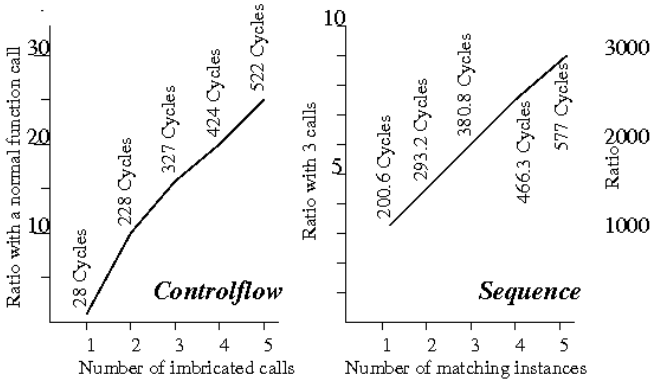
4.1 Evaluation of Arachne's language constructs

To estimate the cost of each of the Arachne's language constructs, we wrote an aspect using that construct that behave as an interpreter of the part of the base program it replaces. For each of these aspects, we compared the time to perform the operation of matching the aspect with the time required to carry out the operation natively. This approach requires to measure very short periods of time, indeed the retrieval of a global variable is done in one CPU clock cycle natively and a function call in no more that 21 cycles. Because standard time measurement APIs are not precise enough (POSIX time API are precise up to one millisecond), our benchmarking infrastructure relies on assembly instructions such as **rtdsc** and **mfence**, and **gcc** code optimization such as loop unrolling. To validate the correctness of our benchmarking protocol, we measured the time necessary to execute a **nop** assembly instruction, that requires one processor cycle according the Intel specifications. Our measures of **nop** presented a relative variation of no more than 1.6%.

The table bellow 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 dynamic test, this good result is not surprising. For similar reasons, a sequence 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 pointcuts used to delimit the different stages, the sequence 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. It is therefore not surprising that a global variable read appears as being 2762 times slower than a direct, aspect-less global variable read. The signal mechanism used in the local alias read aspect 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. Such switches to and from kernel space remain slow.

	cycles		
	Arachne	**Native**	**Ratio**
call	28	21	1.3
seq	201	63	3.2
cflow	228	42	5.4
readglobal	2762	1	2762
read	9729	1	9728

Sequence and controlflow can refer to several points in the execution of the base program (*i.e.*, different stages for sequence and different function invocations for the control flow). The runtime of these aspects grows linearly with the number of execution points they refer to and with the number of matching instances. The figure on the right summarizes a few experimental results for control flow and sequence proving these points.



4.2 Case study on the Squid web cache

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 the disk usage counter aspect described earlier. This accounting aspect is implemented as a sequence aspect scattered in the Squid application and in the Linux kernel. We based our evaluation on Web Polygraph a benchmarking tool developed by the Squid team and featuring a realistic 24 hours HTTP and SSL traffic generator and a flexible content simulator.
The table below resumes our measures. The “monitoring” results sums the number of cycle spent execution the two part of the aspect once. We distinguished the first time the “**vfs_read**” aspect is matched from the next ones, indeed, upon the first call on “**vfs_read**” the aspect allocates memory using “**kmalloc**” to hold results of the counts. During our experiment, we also measured the maximum number of requests Squid was proceeding per second. It shows the accounting aspect had a limited impact on performances (-5% requests).

		cycles	**time**
Monitoring	**first call**	16150	1.16µs
	next	220	15,9ns

5. RELATED WORK

Our work is directly related to other aspect weavers for C, and dynamic code instrumentation techniques. In this section, we consider related work in each of these fields in turn.

Apart from Arachne, there are few aspect weavers for C (or even C-like languages); some noteworthy exceptions, are AspectC (Gong and Jacobsen) (only an uncompleted implementation available), AspectC++ (Spinczyk et al), AspectC# (Kim). All of these rely on source-code transformation and thus cannot be used for the application of aspects to running C code as necessitated by the use we consider. TOSKANA, developed by Engel and Freisleben allows the runtime injection of aspects into the Linux kernel. Nevertheless, TOSKANA presents some restrictions compared to Arachne. First, TOSKANA is limited to the Linux kernel, while adapting it to user-space applications is small issue, there is no support for code spanning both the kernel and user space applications. Second, TOSKANA's aspect language only consider code injected on function calls, whereas Arachne features code injected on function calls and variable access, and higher level primitives to express sequences of function calls or nested function calls. Finally, we believe the code rewriting techniques used in TOSKANA to be unsafe in case of concurrent thread executing simultaneously in places where code is modified.

A few other approaches have considered a direct rewriting of the binary code at runtime: Dyninst (Hollingsworth et al.), Pin (Luk et al.) and DProbes (Moore). Dyninst allows programmers to modify any binary instruction belonging to an executable, however, Dyninst 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 has been carefully designed to avoid these kind of issues; if an aspect can be compiled with Arachne, it can always be woven. Prasad et al. propose Systemtap which is built on top of Kprobes. Systemtap is the most similar tool to Arachne. Indeed, Systemtap is capable of inserting probes into the Linux kernel, developers write patches in a mix of C and awk. Systemtap does not yet support probes in user-space programs but that feature is under development. Nevertheless, Systemtap is limited compared to Arachne as its language does not support higher level constructs such as nested function calls and sequences.

6. CONCLUSION

In this paper we have discussed two different analysis of complex applications which are typical of C-applications using OS-level services and which frequently need to be conducted at runtime. We have motivated that such concerns can be expressed as aspects. We proposed a language that is more expressive than those used in other analysis tools for C in that it provides support for aspects defined over sequences of execution points as well as for variable aliases. Our approach is also novel as it supports aspects crosscutting the kernel and applications boundaries. We have presented an integration of this language into Arachne. Finally, we have provided evidence that the integration is efficient enough to apply such aspects dynamically to high-performance applications, in particular the Squid web cache. As future work, we intend to extend our approach to distributed analysis with aspects spanning multiple machines. We also intend to explore Arachne extension to the C++ language.

ACKNOWLEDGEMENT

This work is supported by a regional grant from the Pays de la Loire (France).

REFERENCES

Journal

Douence, R. et al, 2006. An expressive aspect language for system applications with Arachne. *In Transactions on Aspect-Oriented Software Development*, Vol. 1, No. 1, pp 174-213.

Conference paper or contributed volume

Spinczyk, O. et al, 2002. AspectC++: an aspect oriented extension to the C++ programming language. *Proceedings of the Fortieth International Conference on Tools Pacific*. Sidney, Australia, pp. 53-60.

Engel, M. and Freisleben, B., 2005. Supporting autonomic computing functionality via dynamic operating system kernel aspects. *Proceedings of the fourth International conference on Aspect-Orient Software Development*. Chicago, ILL, USA, pp. 51-62.

Loriant, N. et al., 2006. A reflexive extension to Arachne's aspect language. *Proceedings of the 2006 AOSD workshop on Open and Dynamic Aspect Languages*. Bonn, Germany.

Loriant, N. et al., 2006. Server protection through dynamic patching. *Proceedings of the eleventh IEEE Pacific Rim Conference Symposium on Dependable Computing*. Changsha, China, pp. 343-349.

Hollingsworth, J. K. et al., 1997. MDL: a language and compiler for dynamic program instrumentation. *Proceedings of IEEE conference on Parallel Architectures and Compilation Techniques.* Yaroslavl, Russia, pp. 201-213.

Luk, C.-K. et al., 2004. A post-link optimizer for the Intel Itanium architecture. *Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization.* Washington, DC, USA, pp. 15-27.

Moore, R. J., 2000. Dynamic probes and generalized kernel hooks interface for Linux. *Proceedings of the fourth annual Linux showcase and conference*. Atlanta, GA, USA, pp. 139-145.

Prasad, V. et al, 2005. Locating system problems using dynamic instrumentation. *Proceedings of the 2005 Linux Symposium.* Ottawa, Canada, pp. 49-64 (Vol. 2).

Clowes, S., 2001. Modifying and spying on running process under Linux. *Proceedings of the 2001 Black hat conference*.

Technical reports

Gong, W. and Jacobsen, H.-A., 2006. AspectC Specification. *Middleware Systems Research Group, University of Toronto, Canada*

Kim, H., 2002. AspectC#: an AOSD implementation for C#. Master's thesis. *Trinity College.*