Making Extensibility of System Software Practical with the C4 Toolkit

Marco Yuen[+], Marc E. Fiuczynski[*], Robert Grimm[**], Yvonne Coady[+], and David Walker[*]
[+]*University of Victoria*, [*]*Princeton University*, [**]*New York University*

Abstract. This paper presents work in progress on the C4 toolkit, which is designed to bring AOSD techniques to system software written in C and which we have used for introducing aspects into the Linux 2.6 kernel. The design of our toolkit focuses on addressing key concerns raised by system software developers when handed a new programming paradigm: readability, compatibility, performance, and the preservation of existing development workflows. By addressing these concerns, we believe our toolkit can bring the benefits of AOSD techniques to any large and complex program written in C.

1 Introduction

For system software, C has been the language of choice for many years and will likely remain so for many more. Many large systems, including operating systems (Linux, Mac OS X, etc.), services (Apache, Mysql, etc.), and even applications (FireFox, IE, etc), are written in C and actively extended by its developers. Often these extensions represent *crosscutting concerns* in that they do not fit within a single program module and are scattered throughout the system sources—easily affecting a hundred files. Defining, maintaining, distributing, and applying such extensions is time consuming and error prone. Moreover, it is a significant challenge to ensure the correctness when composing a base system with multiple extensions.

More specifically, crosscutting extensions to the mainline code base are common for Linux. Embedded systems use extensions that reduce the kernel's memory footprint (e.g., [16, 24]), desktop systems use extensions supporting stronger security mechanisms that reduce the impact of viruses and worms (e.g., [15]), web hosting or time-shared systems use resource management extensions to isolate users from each other (e.g., [17, 2]), and super computer systems use special resource management modifications that scale the OS to a large number of processors and support NUMA technology (e.g., [4]).

Recognizing that crosscutting concerns are common for Linux, our work seeks to make them part of the kernel's architecture by leveraging aspect-oriented software development (AOSD) techniques. However, existing AOSD techniques target different languages—notably, Java and C++—and workflows—with aspects being developed in separation of the mainline program—while system builders continue to use C and develop their extensions inline, then extract, distribute, and inject them into different versions of the program. To reconcile this gap between AOSD efforts and systems practice, we are building the C4 toolkit, seeking to address the following research questions:

1. How can C be extended with aspects while also preserving readability, compatibility, performance and existing workflows?
2. How can program analysis automate the identification and resolution of composition conflicts?

This *work in progress* paper focuses on the first question. Aside from the lack of production compilers supporting aspects for C, we think there are two fundamental concerns that, if left unaddressed, will inhibit adoption of AOSD techniques: system software developers (1) are loath to adopt/learn new programming paradigms and (2) typically reject anything that introduces any space/time overheads (see [19] for further details).

To address the first concern we introduce a simple "dual-view" programming model for aspects. In one view, programmers can define aspects the conventional AOSD way by using a so-called "pointcut" language. We adopt AspectC [3], whose pointcut syntax is analogous to that used by AspectC++ [23] and AspectJ [14]. The dual to AspectC is C4 for CrossCutting C Code, which represents the code of an aspect directly inline with the mainline code base. C4 is a minor extension to conventional C (mainly consisting of what looks like syntactic sugar), and thereby hopefully assuages the knee-jerk rejection to new programming paradigms that systems programmers are known for. One goal of our toolkit is to seamlessly translate between the C4 and AspectC views and back again, thereby easing the maintenance task for crosscutting extensions.

To address the second concern, we have built a compiler for the C4 language that tightly integrates aspects with the mainline source, enabling aggressive optimizations at compile time. We have used this compiler to add *before*, *after*, and *around* advice as well as *introductions* to a modern Linux 2.6 kernel. There is no difference in terms of space/time overhead when the extension is added via an aspect vs. manually inlined by a systems developer. Moreover, it was easy to make the C4 compiler part of the existing Linux build system, which lets our solution be completely backward compatible with the existing development practice.

We first touch upon the motivation for building the C4 toolkit in Section 2. We then provide an overview of

the C4 toolkit and our progress so far in Section 3. We conclude in Section 4 with a discussion of pitfalls that we need to avoid.

2 Motivation

As developers of extensions to mainstream system software written in C, we have been frustrated by the lack of appropriate tools to manage the complexity involved with maintaining such extensions. Therefore, our goal for the C4 toolkit is making fine-grained extensibility of mainstream system software practical, focusing on the Linux kernel.

Major variants to a mainstream kernel such as Linux are typically created by adding kernel *extensions*. Current best practice is to implement such an extension by directly modifying the code base of a mainline Linux kernel. Developers often start with a kernel from `www.kernel.org` and then implement their extension by changing the kernel's source code in order to meet the performance or functionality requirements of their target application domain. Upon completing the development of an extension, the developers *extract* the extension with tools such as `diff` to create a "patch set." They then share the resulting patch set with others, who in turn will *integrate* the extension into their version of the kernel using tools such as `patch`.

We have written a separate paper on why `patch` should be considered harmful for maintaining kernel extensions [8]. In a nutshell, because tools such as `patch` build on textual comparison, they have two major limitations:

1. *Patch sets are brittle*. Even minor textual modification to the mainline code can cause `patch` to fail, forcing the person applying the patch set to (a) manually integrate the extension and (b) maintain this integration as the mainline code base evolves.

2. *Patch sets are difficult to compose both textually and semantically*. At the textual level, patch sets are generated against the baseline code and therefore the integration of extensions will fail when multiple patch sets modify the same code sequence(s). At the semantic level, even if patch sets apply, the developer cannot determine whether they in fact result in a correctly working kernel.

Of course, better solutions for operating system extensibility and composability have been studied extensively: Singularity [13], SPIN [1], Exokernel [6], Kea [25], MMLite [12], KNIT [22] and the OSKIT [9] are just a few examples. Moreover, Lohmann et al. [18] provide a quantitative analysis of aspects in the eCos kernel. However, these systems represent clean slate designs focused on extensibility and composability. They require fundamentally different programming models, APIs, and development practices when compared to the status quo set by mainstream operating systems such as Windows, Linux, and Mac OS X.

In contrast, we seek to address the brittleness and composition problems of patch sets, while also minimizing any changes on existing development practices. In other words, we need to build on C and rely on inline development, extraction, distribution, and integration, while also maintaining backwards compatibility, source code readability, and executable performance.

Our chosen approach is to express extensions, at the source level, as aspects—thus providing a language-supported mechanism for extracting and injecting crosscutting concerns from and into a program. By doing so, we can directly address the limitations of patch sets. First, aspects provide a well-defined specification of domain-specific features that can be separated from baseline functionality, but can also be automatically integrated again. Second, we believe that aspects enable tools that perform automatic analysis of the interaction between several crosscutting concerns and identify true semantic conflicts as opposed to the line-by-line conflicts identified by `patch`.

3 The C4 Compiler Toolkit

In this section, we discuss our aspect-oriented language enhancements, the C4 toolchain overall, and code generation in particular. In other words, we describe how we are addressing the research question of extending C with aspects but without impacting readability, compatibility, and performance.

3.1 AOSD Language Enhancements

Due to our dual-view approach to AOSD in C, we have two languages: AspectC and C4.

The AspectC language. Since aspects encapsulate concerns that crosscut many other concerns, usually in many different places (files, functions), there is a need to describe the collection of join points (points in the execution of a program affected by an aspect). This collection of join points is called a "pointcut". We use AspectC as our pointcut language for C, building on an earlier version defined by Coady [3]. The original AspectC is derived from the non-object-oriented mechanisms in AspectJ [14] for Java and represents the current state of the art in AOSD, thus providing us with a solid basis to continue our research with.

The C4 language. This language represents an AOSD-enhanced version of the C language and is designed to aid developers with defining aspects inline the

mainline code. It departs from established AOSD practice in order to support current systems practice. Without it, system builders would have to adopt a considerably less familiar programming paradigm, thus increasing the challenge to successfully adopting AOSD techniques.

Both AspectC and C4 languages support *before*, *after*, and *around* advice for execution join points. Both languages also support aspect-oriented *introductions* of global variables, functions, and types, as well as new fields to structures and unions.

There currently is no support for dynamic join points such as *cflow* and *call*. It is challenging to support these in a manner that is both efficient and correct. For *cflow*, we could adopt the implementation technique used by AspectC++[23]; while it imposes little space/time overhead, it does not work correctly in programs that use non-local returns (e.g., setjmp/longjmp, signal handlers, etc.). Supporting *call* join points efficiently in C is difficult due to the existence of function pointers, as it potentially requires a runtime check at most (if not all) call points using function pointers. While we are investigating techniques to support such dynamic join points, we believe that support for just execution join points is sufficient to alleviate a significant point of pain felt by systems programmers adding extensions to a mainline code base.

We do not present the formal definition of either language. Furthermore, the equivalent of our AspectC language is already detailed by Coady [3]. For this reason, we will focus on presenting the C4 language with a set of simple examples.

Listing 1 shows *before* and *after* advice defined in the code using the C4 language.

```
1  int foo(char **c, size_t s) {
2    /* Before advice for aspect ''bar'' */
3    aspect_before(bar) {
4      printf("I␣am␣a␣before␣advice.\n");
5      ...
6      /* execution flow falls through */
7    }
8
9    ... /* Function body */
10   return 42;
11
12   /* after advice for aspect ''bar'' */
13   aspect_after(bar) {
14     printf("I␣am␣a␣after␣advice␣(%d).\n",
15   __returned__);
16     ...
17   }
18 }
```

Listing 1: Inline before and after advice

As shown, programmers simply encapsulate the extension code they wish to add to the mainline code with either an aspect_before(*identifier*) or aspect_after(*identifier*) construct. Syntactically the before and after advice are at the beginning and end of a function, respectively.

The before advice has access to the function's formal arguments. Any changes to these arguments will be visible when control falls through the bottom of the before advice to the subsequent code, which in this case is the main function body yet could also be nested before or around advice.

When after advice is applied to a function, the C4 compiler changes the operational semantics of all return statements within that function to ensure that control flows to the after advice block. Specifically, rather than exiting from the function on line 10, the return value is stored in a local variable, called *__returned__*, which is accessible by the after advice (see line 15). The after advice can optionally change this value, which will then become the new return value of the function.

Listing 2 illustrates C4's *around* advice, which uses an aspect_around() keyword to encapsulate the advice block.

```
1  void func(int var, char *str) {
2    /* Around advice for Aspect ''foobar''
         */
3    aspect_around(foobar) {
4      printf("Inside␣around␣advice\n");
5      if(var ==  20)
6        proceed(20,"Twenty");
7      else
8        proceed(0,"ZERO");
9      /* execution flow implicitly returns
           from here */
10   };
11
12   ... /* Function body */
13 }
```

Listing 2: Inline around advice

The *around* advice appears before the function body. To enter the main function body from around advice, the programmer must explicitly use proceed([*args*]) statement(s) from the around advice.

Listing 3 shows examples of aspect-oriented *introductions*.

```
1  aspect_introduction(test) {
2    struct testFile{  char *name; int fd
         ; ...
3    };
4    typedef int myint;
5  };
6
7  struct system { int a,b,c; ...
8    aspect_introduction(memory) {
9     int new_a; /*only visible to aspect*/
10   };
11 };
```

Listing 3: Introductions

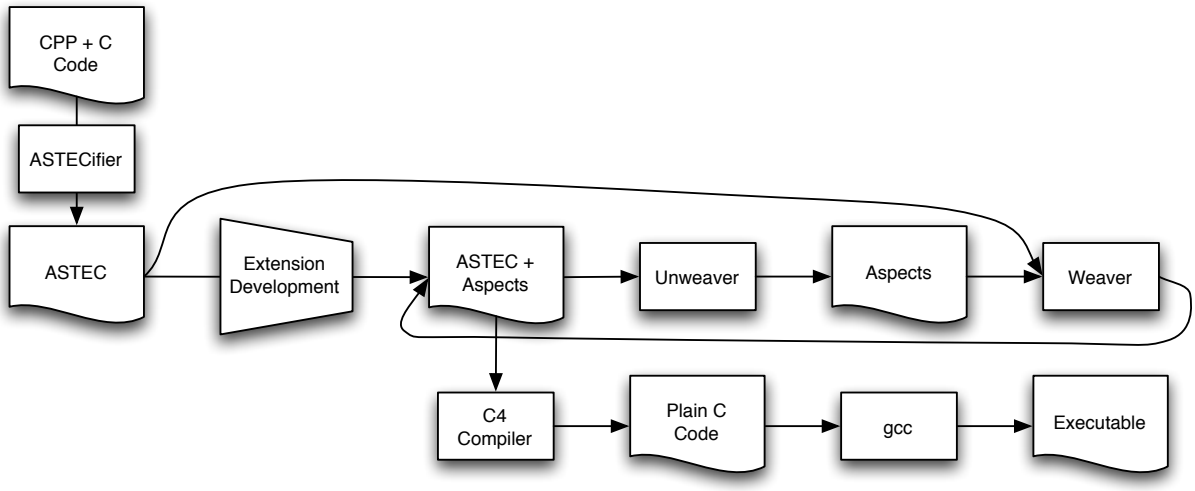

Figure 1: The interaction between C4's tools. The *ASTECifier* elevates common C preprocessor uses to the language level, thus providing our tools with a more stable basis. The *unweaver* takes an extended program as its input and extracts the corresponding aspects. The *weaver*, in turn, takes the mainline code and aspects as its inputs and produces the inline version again. The *C4 compiler* generates executable code, deferring traditional optimizations and code generation to gcc.

`C4` allows the introduction of new fields to structures and unions, as well as new functions, types, and global variables to a program. Lines 1–5 of listing 3 show the syntax for introducing a global type and a structure. Also, lines 8–10 illustrate how new fields can be inserted into an existing structure. Note that these introductions are *only* visible to the aspect that defines them. For example, a variable introduced in `aspect(foo)` will not be visible in `aspect(bar)`. Moreover, an aspect's introductions will not be visible in the normal C scope. This is necessary to avoid name clashes in the global name space of a large system.

While we support nesting of different aspect advice, we have yet to define support for composing aspects and ordering the application of advice to functions. Explicit composition support will let one aspect leverage introductions provided by another aspect. AspectC++ has support for ordering the application of aspects and we are investigating whether we can leverage their approach within both our `C4` and `AspectC` languages.

3.2 Tool Support

Our goal is to fully explore the practicality of system software extensibility using AOSD techniques. To do this requires that we build the corresponding tools. Figure 1 illustrates our tools in the context of a development work flow commonly used by system software developers — i.e., extend and test, extract tested extension, and reinject extension into mainline source base. In support of this work flow model, our tools include the *unweaver*, which extracts aspects defined using the `C4` language, the *weaver*, which injects aspects, and the *C4 compiler*, which translates C code with `C4` language extensions to executable code via a traditional compiler such as gcc (which performs traditional optimizations and code generation). These three tools are being built on top of `xtc` [10, 11], which provides a toolkit for building extensible source-to-source transformers.

While `xtc` will reduce the required engineering effort—in fact, we have already implemented a prototype compiler for `C4` on top of it (viz. Section 3.3)—we are still facing two major challenges in realizing these tools. First, the C preprocessor is based on textual substitution and thus represents a considerable obstacle to provably correct unweaving and weaving as well as to semantic analysis of aspects. To address this challenge, we will not base the `C4` language on C itself but rather on ASTEC [20], which extends C with language-level constructs for common uses of the C preprocessor [7] and thus provides a more suitable basis for our work. We expect to leverage the conversion tool created by ASTEC's designers, which (largely) automates the transition to ASTEC. However, to ensure that `C4`'s users can easily back out of both ASTEC and `C4` we will also create the tool to revert back to traditional C code.

Second, moving from `C4` to `AspectC` back to `C4` view should result in exactly the same program sources as the original sources. At a semantic level, this isomorphism captures the requirement that unweaver and weaver must not modify a program's code beyond extracting and injecting an extension. At a syntactic level,

it captures the requirement that the unweaver and weaver must preserve a program's formatting including comments to minimize any developer disruption (who, for example, would surely not appreciate if the `C4` tools reformatted *their* code according to *our* coding conventions). To address this challenge, `xtc`'s modular parser generator and code pretty printers need to be extended with support for preserving formatting and comments.

3.3 Compiler for the `C4` language

Referring back to figure 1, we have implemented a prototype `C4` compiler that can transform the `C4` language into pure C code. The compiler implementation of the inline aspect-oriented advice and introductions in `C4` is relatively straightforward. We built our `C4` compiler by extending the modular C grammar provided with `xtc`. The corresponding parser emits a full abstract syntax tree (AST) representation of the sources and thereby enables us to fully analyze the program. Specifically, we leverage this functionality to understand the precise scope within which `C4` code has been defined.

Aspect-oriented *before* and *after* advice is directly inlined into the code and assigned a new scope within the function. This prevents interference with the normal function scope and, more importantly, meets the performance criterion that is so important to system software developers: the overhead of invoking advice is indistinguishable from any other inlined function. To ensure compatibility with debugging tools such as gdb, we also specify the corresponding #line statements in the generated code to correspond to the `C4` statements versus the pure C code generated by the `C4` compiler.

As mentioned, the `C4` compiler modifies the operational semantic of `return` statements in functions to which advice has been applied. This may be a source of confusion to conventional C programmers, as the expected behavior is to exit the function. We are considering of having `C4` rewrite the return statement to an explicit goto statement at the source level. However, we intend to defer this type of change until we receive the corresponding user feedback, i.e., that changing the semantics of return is confusing them.

Aspect-oriented *around* advice is also directly inlined with the function to which it applies. It falls naturally from the `C4` definition as simply being the inlined variant of before and after advice. The implementation of `proceed()` boils down to a computed goto to the main body of the function (or the next *inner* advice when multiple aspects are composed). As the `C4` compiler generates C code rather than optimized assembly, we currently leverage a gcc extension for computed gotos to implement `proceed()` efficiently. This was a design choice we made to (a) simplify our task of making the code easily readable when debugging with gdb and (b) ensure that we always used the most optimized approach for our target environment/audience (i.e., the Linux kernel).

Aspect-oriented *introductions* of global variables, new functions, and types are defined using a name mangling technique similar to C++ in order to avoid name clashes. Similarly, introductions of fields is implemented as a nested struct/union (also using name mangling), which declares the new fields. Just as is the case for C++, we will need to change gdb to properly resolve the names of these newly introduced variables, fields, and functions. While we have not done so yet, we expect this to be a straight-forward adaption of the exiting C++ support in gdb.

We have incorporated the `C4` compiler into the Linux kernel(2.6.10-13) build process. It was able to correctly compile the kernel, which makes very heavy use of gcc extensions. We introduced some before, after, and around advice as well as introductions to the kernel to validate the correctness and scrutinize their overhead with respect to execution speed and space. This overhead was as expected: the same as manually makeing these modifications to the kernel. More importantly (at least to us), the resulting kernel image booted on real hardware and ran correctly.

Using the `C4` compiler to build the kernel is currently on the order of 4x slower than building the kernel straight with gcc. Our goal is to reduce this overhead down to 2x. While that is still quite high for a production setting, we believe it suffices to demonstrate the general utility of the overall toolkit.

4 Discussion

Our initial research goal is to support the *syntactic* separation of crosscutting concerns through aspects. On their own, syntactic separation and automatic weaving of crosscutting concerns free developers from many low-level, time-consuming, and error-prone details of maintaining extensions to software written in C, especially as the mainline code base evolves.

However, there remains at least one significant problem: in monolithic systems designed and implemented in C the number of potential join-points might be too low for aspects to be practical. Lohmann et al. [18] suggest this to be the case for the Linux kernel, especially when compared to an object-oriented kernel such as eCos. To address this problem we may leverage new techniques exposing finger-grain join points, e.g., at the statement-level [21] and possibly even at the instruction-level [5]. Until such techniques are proven, we may need to rely on refactoring the mainline code base to expose better join points.

Another problem is that there are many case where extensions to the baseline Linux kernel cannot cleanly be captured as advice. While this is a significant problem,

we do not believe it is impossible to solve. Rather, when automatic weaving/unweaving of the AOSD compliant components of an extension works seamlessly, we hope to see a groundswell of developers wanting to refactor existing mainlined extensions to the Linux kernel (and other system software written in C) into `C4`-based aspects. Why? At least for Linux, for no other reason but to contribute to the cause of decluttering the mainline code base from the myriad of extensions that have already been added in line. To prepare for this event, we will work on defining new best practices for such aspect-oriented refactoring without disrupting existing systems practices. It is precisely for this reason that the `C4` toolkit retains the "look and feel" of C as well as the existing development workflow.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, Dec. 1995.

[2] Class-Based Kernel Resource Management (CKRM). `http://ckrm.sf.net/`, 2004.

[3] Y. Coady. *Improving Evolvability of Operating Systems with AspectC.* PhD thesis, University of British Columbia, July 2003.

[4] CPUSETS for Linux. `http://www.bullopensource.org/cpuset/`, 2004.

[5] M. Engel and B. Freisleben. Using a low-level virtual machine to improve dynamic aspect support in operating system kernels. In *Proceedings of the AOSD ACPIS Workshop 2005*, pages 1–6, Chicago, IL, Mar. 2005.

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[7] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, Dec. 2002.

[8] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. `patch (1)` considered harmful. In *Proc. 10th Workshop on Hot Topics in Operating Systems*, pages 91–96, June 2005.

[9] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 38–51, 1997.

[10] R. Grimm. Systems need languages need systems! In *2nd ECOOP Workshop on Programming Languages and Operating Systems*, July 2005.

[11] R. Grimm. Better extensibility through modular syntax. In *Proc. 2006 ACM Conference on Programming Language Design and Implementation*, June 2006.

[12] J. Helander and A. Forin. Mmlite: a highly componentized system architecture. In *Proc. 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pages 96–103, New York, NY, USA, 1998. ACM Press.

[13] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steesgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR–TR–2005–135, Microsoft Research, Oct. 2005.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[15] Linux Security Modules. `http://lsm.immunix.org/`, 2004.

[16] Linux-Tiny. `http://www.selenic.com/tiny-about`, 2004.

[17] Linux VServer Project. `http://linux-vserver.org/`, 2004.

[18] D. Lohmann, F. Scheier, R. Tartler, O. Spinczyk, and W. Schroeder-Preikschat. A quantitative analysis of aspects in the OS kernel. In *Proc. 1st EuroSys*, Apr. 2006.

[19] C. Matthews, O. Stampflee, Y. Coady, J. Appavoo, M. Fiuczynski, and R. Grimm. Hey... you got your paradigm in my operating system! In *2nd ECOOP Workshop on Programming Languages and Operating Systems*, July 2005.

[20] B. McCloskey and E. Brewer. ASTEC: a new approach to refactoring C. In *Proc. of the 10th European Software Engineering Conference*, pages 21–30, Sept. 2005.

[21] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, IL, Mar. 2005.

[22] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, Oct. 2000. USENIX Annual Technical Conference.

[23] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proc. 40th International Conference on Technology of Object-Oriented Languages and Systems*, Sydney, Australia, Feb. 2002.

[24] uCLinux. `http://www.uclinux.org/`, 2004.

[25] A. C. Veitch and N. C. Hutchinson. Kea—a dynamically extensible and configurable operating system kernel. In *Proc. 3rd International Conference on Configurable Distributed Systems*, page 236, 1996.