Accurate Program Analyses for Successful Specialization of Legacy System Software

Gilles Muller, Renaud Marlet, Eugen-Nicolae Volanschi [1]

{muller,marlet}@irisa.fr
http://www.irisa.fr/compose

Compose project, IRISA / INRIA, Campus Universitaire de Beaulieu, 35042 Rennes cedex, France

Abstract

Choosing the accuracy of program analyses is a crucial issue when designing and developing a partial evaluator capable of treating realistic programs, and in particular legacy software. In this paper, we investigate the degree of accuracy of alias and binding-time analyses that is required to successfully exploit the specialization opportunities present in the Sun commercial implementation of the Remote Procedure Call protocol (RPC).

The Sun RPC implementation consists of a stack of small parameterized layers. This structure is representative of a certain programming style in operating system and network development. The analysis features that we have explored have been implemented in Tempo, a partial evaluator for C. After automatic specialization of the RPC using Tempo, we measured speedups up to 1.5 for complete remote procedure calls (including network transport) and up to 3.7 for local buffer encoding alone. This experiment suggests that partial evaluation is reaching a high level of maturity.

Key words: Partial evaluation, program analysis, program transformation, remote procedure call (RPC)

1 Introduction

Partial evaluation has seldom been applied to realistic programs, and in particular to legacy software. One reason is that most existing partial evaluator address functional and logic programming languages whereas the software industry relies

[1] Author's current address: Canon Research Centre France, Rue de la Touche Lambert, Rennes Atalante, 35517 Cesson Sévigné cedex, France. `volanschi@crf.canon.fr`

mainly on imperative and object-oriented programming. Another reason is that a partial evaluator not only has to treat large programs that use most constructs of the language, but it also has to be powerful enough to appropriately exploit the specialization opportunities present in the code.

An offline partial evaluator is divided into an analysis phase and a specialization phase. The transformations straightforwardly follow from the results of the analysis. The accuracy of the analysis phase is thus a crucial concern for both an end user and the designer of a partial evaluator. Implementing a sound and efficient analyzer is difficult, and later extending it to improve the precision may involve major changes. Choosing the precision of the analyses is thus an issue that must be addressed early in the development process of a partial evaluator. It does not mean that analyses should be as precise as possible. Too precise analyses can be worthless or even impractical as the computation time generally increases much with the accuracy.

The level of accuracy needed depends on the kind of programs that are targeted. Taking this issue into account, a large part of the work in the Compose research group concerning partial evaluation has been aimed at operating systems. Our research has been driven by realistic examples such as the Sun implementation of the Remote Procedure Call (RPC) protocol. This implementation is highly generic: it consists of a stack of small parameterized layers, which is characteristic of a certain programming style in operating system and network implementation. This software architecture calls for specialization [19].

The design of Tempo, our partial evaluator for C, was strongly influenced by the requirements of the Sun RPC case. It determined the subset of the C language to treat as well as the major functionalities and precision of the analyses. In particular, it motivated the improved binding-time analysis presented in [13] (included in this issue). Subsequently, these features have proved sufficient to optimize the Chorus IPC [37], software architectures [19], interpreters [34–36], computer graphics, and scientific computations [17,23]. These applications required an analysis precision equal or less than what was needed for the Sun RPC.

The main goal of this paper is to briefly present and assess the features that a partial evaluator *must* have to handle legacy system software such as the Sun RPC. We believe this information is fundamental for people that plan to develop a specializer, as well as for software engineers who wish to determine whether a partial evaluator matches their needs. Space constraints prevent us from providing a thorough description of the examples that support our study; detailed reports on the Sun RPC specialization experiment can be found elsewhere [20,22].

The paper is organized as follows. Section 2 provides an introduction to the RPC and describes the architecture of Sun's implementation, as well as specialization opportunities and expectations. Section 3 lists the analysis features that are required to meet those expectations. Section 4 briefly presents Tempo which implements most of these features. Section 5 gives benchmarks on the resulting specialization of the Sun RPC, and Section 6 concludes.

2 The Sun RPC Standard Protocol

The RPC protocol was introduced as a basis for the implementation of distributed services between heterogeneous machines. This protocol has become a standard in distributed operating systems design and implementation. It is notably used for implementing widespread distributed services such as NFS [32] and NIS [29]. The RPC implementation used in this paper is the commercial, 1984 copyrighted version of Sun.

Performance is a key point in RPC. A lot of research has been carried out on the optimization of the layers of the protocol [3,15,24,30,33]. However, many of these optimizations involve new protocols that are incompatible with an existing standard such as the Sun RPC.

2.1 RPC Main Features

The RPC protocol makes a remote procedure look like a local one. A call to a remote procedure is done transparently on the local machine, but the actual computation takes place on a distant machine. To that end, RPC supplies an interface between a client (on the local machine) and a server (on the remote machine) through *stub* functions, that are automatically generated from the user-specified signature of the procedure.

RPC performs two kinds of operations:

- It converts data between a local, machine-dependent representation and a network, machine-independent representation. This machine-independent data representation is standardized by the eXternal Data Representation (XDR) protocol.
- It manages the exchange of XDR-encoded messages through the network.

Our study focuses on the first point.

2.2 Specialization Opportunities

The Sun RPC is composed of a set of generic, modular micro-layers. Each layer is devoted to a small task such as managing the transport protocol (e.g., TCP or

UDP) or writing (or reading) data into the encoding buffers (memory or stream). The micro-layers may have several implementations. However, most of the time, given an application, the configuration never changes.

The layers and their configuration are the sources three main specialization opportunities: the signature of a remote procedure, communication configuration parameters, and explicit constants that are embedded in the Sun RPC routines.

- The user-defined signature of a remote procedure (i.e., the list of types for the arguments and return value) determines statically the exact sequence of encoding and decoding operations for the arguments and return value of an RPC.
- The configuration of the communication is performed at initialization time, when the user chooses the transport protocol, e.g., TCP or UDP. This choice determines the implementation of network and buffering operations.
- The Sun RPC code comprises various routines that are passed an explicit constant parameter. In particular, an invariant field of a structure that is passed as an argument to all the layers determines the coding direction. As an example, XDR routines are generic; they can perform both encoding and decoding.

2.3 Specialization Expectations

By exploiting the specialization opportunities listed above, a partial evaluator should be able to eliminate safely numerous computations that performs tests, checks, and function calls:

- *Conditions that determine if an XDR routine should either decode or encode*. Since the coding direction flag is invariant for a given transfer, the condition can be evaluated away.
- *I/O buffer overflow checking*. Because the size of the buffers, as well as the number and types of the remote procedure arguments are known, it is possible to safely evaluate away overflow checks.
- *Tests of exit status at all layers*. A failure here can only be the propagation of a buffer overflow error. The knowledge of an overflow can be safely propagated through all layers, leading to additional checks elimination.
- *Indirect function calls*. Since the configuration of the layer is fixed, indirect function calls between layers can be replaced by procedure calls.
- *Useless function calls*. Reducing code size by eliminating checks and removing indirect function calls opens new inlining opportunities.

As a result, partial evaluation tightly merges the software layers and yields a specialized implementation that merely consists of elementary copies to or from the I/O buffers.

3 Accuracy of Analyses for Specializing the Sun RPC

The heart of offline partial evaluators lies in the *binding-time analysis* (BTA) [5,16]. The BTA partitions the program into *static* and *dynamic* computations. Static computations are those that only involve values that are known at specialization time. Dynamic computations are those that may depend on values that will not be known until run time. Given actual input values, the specialization transformation phase then evaluates static computations, and leaves dynamic constructs unaffected (i.e., *residualized*). The most trivial BTA simply annotates everything as dynamic. A more precise analysis can identify more static computations, which can be performed at specialization time.

A BTA is a data-flow analysis. In the case of a language with pointers like C, it has to take aliases into account. An alias analysis must thus precede the BTA. (Doing both analyses simultaneously does not improve the overall results, as the alias analysis has no use for binding-time information.) The precision of the alias analysis affects the precision of the binding-time information, and thus the degree of specialization: superfluous aliases to memory locations that are found dynamic force a dereference to be dynamic as well. In the following, we consider the precision of both the binding-time and the alias analyses.

3.1 Intra-Procedural vs Inter-Procedural Analyses

An *intra-procedural* analysis of a function has no information about the arguments (actual parameters as well as global variables) and thus has to make conservative assumptions. On the contrary, an *inter-procedural* analysis exploits information depending on the calling contexts.

In the Sun RPC, all the parameters related to communication are stored in a structure that is passed to each of the small layered functions. The BTA must thus be inter-procedural. The alias analysis must also be inter-procedural. Otherwise (conservatively) pointer-typed arguments can point to any location; in practice this makes their dereference systematically dynamic.

3.2 Data Structure Granularity

The granularity of data structure properties determines whether the analysis assigns a single, global property to an entire C structure, or a separate property to each field of the structure (i.e., *structure polyvariance*). An intermediate alternative is to keep separate field properties, but to merge the properties of all instances of a given structure type (i.e., *structure monovariance*).

In the Sun RPC, the current coding state is stored in a structure that contains known configuration parameters (some of which are pointers) as well as pointer fields to dynamic buffers. To exploit such a partially static data structure, both the alias and the binding-time analyses must at least be structure-monovariant.

The problem with structure monovariance is that as soon as a static field becomes dynamic, it cannot later become static again. This is because setting the field of a structure instance to static does not imply that this field is static in all other instances. This case occurs in the Sun RPC code for the coding direction flag because the emission and reception of data are embedded in a retry loop to handle possible network errors. Structure polyvariance solves this problem. However, in system code in general, different instances of the same structure type tend to be used uniformly. Structure polyvariance is thus generally not needed.

3.3 Context Sensitivity

A *context-sensitive* analysis of a function determines computation properties that can depend on the local properties at each call site, as opposed to being identical for each call site (i.e., merging all possible properties). In the context of binding-time properties, this feature is also called *binding-time polyvariance*: several instances of a function with different binding times for the arguments (and the accessed store) may coexist.

Context sensitivity is useful for the integer encoding function. This function is usually called with dynamic data representing the Sun RPC arguments. Still, a static integer is encoded in each client emission: the encoding of the procedure identifier. Differentiating between the two call contexts preserves a specialization opportunity.

3.4 Flow Sensitivity

A *flow-sensitive* analysis determines computation properties that can depend on program points, as opposed to being constant for a whole function where all possible properties are merged. (This is an issue only for languages with side-effects.)

Possible run-time errors may occur in the decoding of input buffer when receiving a message. Testing for this error introduces dynamic conditions after which static information is lost. However, each branch of the corresponding conditionals can still exploit the static information. To this end, the binding time must not be a global property; it must depend on program point.

3.5 Use Sensitivity

Many of the data structures used within the Sun RPC layers are partially static. Typically, these structures are passed to a function by means of a pointer. If this pointer is static, one would expect to statically access the static fields of the structure, *and* to dynamically access the dynamic fields. However, in a traditional BTA, a computation can only be *either* static *or* dynamic. Thus, the statement that initializes the structure pointer must be considered dynamic. If the pointer is dynamic, then all uses of it are considered dynamic as well, thus destroying the interest of partially-static structures.

Motivated by several similar cases, the BTA of Tempo has been enhanced to be *use-sensitive*: it allows a computation to be considered *both* static *and* dynamic when used in static and dynamic contexts [14]. Such a computation is evaluated and exploited at specialization time, but is also present in the residualized program.

3.6 Return Sensitivity

Most of the functions in the XDR layers return a literal boolean value (expressing success or failure) selected under a static condition. However, these functions contain dynamic side-effects. Consequently, all the calls must be residualized. In this case, a traditional BTA considers that the return value is dynamic, thus inhibiting specialization of all the callers.

A *return-sensitive* BTA was designed to allow a residualized function to return a static value that can be exploited by the callers [12]. Exploiting such an analysis requires a new specialization transformation. Alternatively, the program can be rewritten just before specialization so as to return static results through global variables. The rewritten functions become `void` functions, and are called in the residual program for their dynamic side effects. Returned static values and assignments to the global variables are fully exploited and consumed at specialization time.

4 Implementation in Tempo

Our group has developed an offline partial evaluator for C, named Tempo [6,8], that implements the above features. Tempo supports traditional compile-time specialization as well as run-time specialization [9,23], data specialization [2], and incremental specialization [18]. It is publicly available [2].

[2] Tempo home page: `http://www.irisa.fr/compose/tempo`

Tempo's alias analysis is inter-procedural, structure-monovariant, flow-sensitive and context-insensitive. Tempo's BTA is inter-procedural, structure-monovariant, and sensitive to flow, context, use [14] and return [12]. In addition to these features, two manual transformations were necessary to successfully specialize the Sun RPC. Because Tempo is only structure-monovariant, we had to specialize separately the emission and the reception of network data, rather than specializing the function that sequences both operations (cf. Section 3.2). Another very simple transformation (known in the partial evaluation community as "the trick") was needed to expose a specialization opportunity: the length of received data has a static expected value; however, the actual value read from the network is dynamic because the transmission can fail. Still this value can be exploited by making it explicit (cf. [22, Section 3.4]).

Other existing partial evaluators for C implement different accuracy choices. C-Mix is a compile-time specializer [1]. Its BTA is not sensitive to flow, context, use, nor return; it handles partially static data structures only intra-procedurally, via structure splitting. DyC is a run-time specializer [10,11]. It does not include an alias analysis; aliases are handled through user annotations. The BTA is flow-sensitive but also relies on manual annotations for partially static structures, use sensitivity and inter-procedural features such as context sensitivity. Therefore, only Tempo is able to treat the Sun RPC case.

The combination of the features present in Tempo has proved to be generally sufficient for the kinds of system programs we have studied so far. However, Tempo is now also used as a back-end specializer for Java [31] using Harissa [21], a JavaVM-to-C converter. In an object-oriented language like Java, different object instances of the same class (which Harissa translates into a C structure) tend to be used differently, which results in different binding times. This calls for structure-polyvariance, both for aliases and binding times. These new analysis features are currently being implemented.

5 Benchmarks

Because it implements the feature requirements listed in Section 3, Tempo is able to exploit the specialization opportunities mentioned in Section 2.3. To assess this specialization, we have run tests that emulate the behavior of scientific parallel programs exchanging large chunks of data. The test program loops on a remote procedure call that sends and receives an array of integers. We have made two different kinds of measurements, comparing the specialized code to the original one, for different array sizes: (i) a micro-benchmark of the encoding process in the client, and (ii) a full round-trip remote procedure call that includes network transport. In order to take into account architecture-dependent aspects such as cache, memory and network, measurements have been done on two different kinds of platforms: Suns

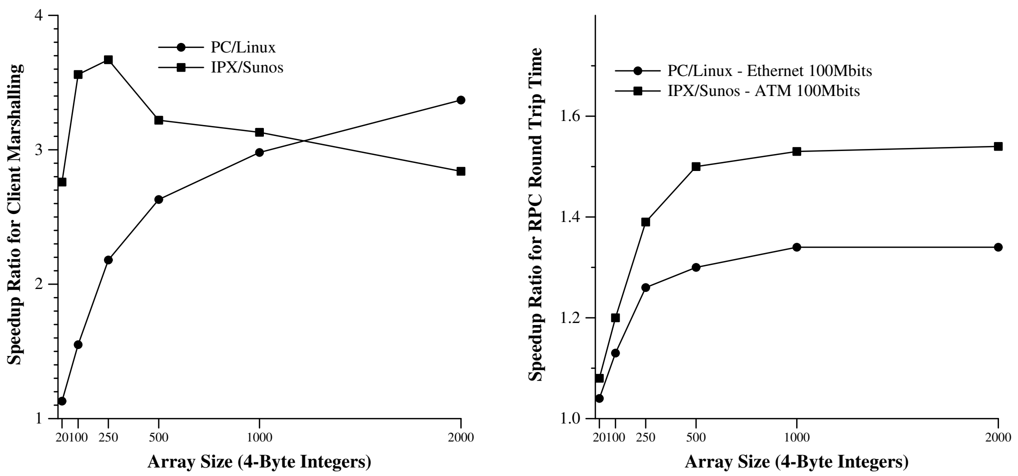

Fig. 1. Sun RPC performance improvement

4/50 connected with a 100 Mbits ATM link, and PCs Pentium 100 MHz connected with a 100 Mbits Ethernet network.

On the encoding layer, the specialized code is up to 3.7 times faster than the unspecialized code for the Sun platform. The speedup is up to 3.5 for the PC platform (see Figure 1). On the round-trip RPC, we have a speedup of up to 1.5 for the Suns, and 1.3 for the PCs. Further details on this experiment can be found in [20,22].

6 Conclusion

After carefully studying crucial features of the BTA and implementing them in Tempo, the Sun RPC has been successfully specialized and significant speedups have been obtained. This shows that applications of partial evaluation to industrial-strength programs can now be considered, and yield non-trivial results. In particular, we can now contemplate automating previous manual experiments in specialization of operating systems code [27,28].

This experiment illustrates the fact that partial evaluation is an appropriate and effective tool to suppress fine-grain modularity overhead. It should encourage engineers to write software components and applications that are generic, letting partial evaluation take care of performance issues. Adaptability should be considered more important than immediate efficiency [7,4].

Acknowledgements

The authors would like to thank the other designers and implementors of Tempo (Charles Consel, Jacques Noyé, Luke Hornof, Julia Lawall, Scott Thibault, François

Noël, Alan Sayle, Sandrine Chirokoff *et al.*) for fruitful discussions and unsparing efforts.

This research is supported partly by France Télécom / CNET grants 951W0009 and 96-1B-027.

References

[1] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[2] S. Chirokoff and C. Consel. Combining program and data specialization. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 45–59, San Antonio, TX, USA, January 1999. ACM Press.

[3] D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, September 1990. ACM Press.

[4] C. Consel. Program adaptation based on program transformation. *ACM Computing Surveys*, 28(4es):164–167, 1996.

[5] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.

[6] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.

[7] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Partial evaluation for software engineering. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.

[8] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

[9] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[10] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Annotation-directed run-time specialization in C. In PEPM'97 [25], pages 163–178.

[11] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in DyC. In PLDI'99 [26]. to appear.

[12] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In PEPM'97 [25], pages 63–73.

[13] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoretical Computer Science*, 1999. to appear.

[14] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, September 1997. Springer-Verlag.

[15] D.B. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software - Practice And Experience*, 23(2):201–221, February 1993.

[16] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[17] J.L. Lawall. Faster Fourier transforms via automatic program specialization. Research Report 3437, INRIA, Rennes, France, June 1998.

[18] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In PLDI'99 [26]. to appear.

[19] R. Marlet, S. Thibault, and C. Consel. Efficient implementations of software architectures via partial evaluation. *Journal of Automated Software Engineering*, 1999. Kluwer Academic Publishers. To appear.

[20] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[21] G. Muller and U. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, March 1999.

[22] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In PEPM'97 [25], pages 116–125.

[23] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

[24] S. O'Malley, T. Proebsting, and A.B. Montz. USC: A universal stub compiler. Technical Report TR94-10, University of Arizona, Department of Computer Science, 1994. Also in Proc. Conf. on Communications Archi. Protocols and Applications.

[25] *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.

[26] *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, US, 1–4 May 1999.

[27] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

[28] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[29] R. Ramsey. *All about administering NIS+*. SunSoft, 1993.

[30] M.D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[31] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, Lisbon, Portugal, June 1999. To appear.

[32] Sun Microsystem. NFS: Network file system protocol specification. RFC 1094, Sun Microsystem, March 1989.

[33] C.A. Thekkath and H.M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[34] S. Thibault, L. Bercot, C. Consel, R. Marlet, G. Muller, and J. Lawall. Experiments in program compilation by interpreter specialization. Research Report 3588, INRIA, Rennes, France, December 1998.

[35] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.

[36] S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device drivers: from design to implementation. In *Conference on Domain Specific Languages*, pages 11–26, Santa Barbara, CA, October 1997. Usenix.

[37] E. N. Volanschi. *An Automatic Approach to Specializing System Components*. PhD thesis, Université de Rennes I, February 1998.