Software architecture adaptive compilers

Jonne Itkonen

Department of
Mathematical Information Technology
University of Jyväskylä Finland
E-mail: ji@mit.jyu.fi

Abstract

As software grows in size and complexity, compilers that can adapt to software are needed, to ease the developers in the quest for code optimised by multiple objectives simultaneously. In this paper, we consider briefly the possibility of compilers adapting by software architecture.

1. Introduction

Current compiler technology concentrates on compiling optimised code in general cases. Although compilers for specific application domains have been written, compilers that adapt are still rare. Developers do have quite rich possibilities to modify the code generation of the compiler via command line arguments, but finding the optimal choice of parameters can be a burdensome task that requires strong expertise of not only the application domain, but of the specific compiler and computer architecture used.

Programmes are also compiled for a specific or generalised computer architecture. This might lead to contradictions betweenlocally and externally developed software. This possibility has lead to the practice of continuously optimising compilers, that using data gathered from the execution of the programme, try to recompile parts of the programme more optimally. This technique of continuous compiling has been applied to Java JIT compilers [1, 16], to Self [7] and to C compilers [14].

The question this paper is asking is: Can the description of the architecture of the software be used when trying to find an optimal adaptation for a compiler? Can a compiler automatically adapt to compile code optimised for the architecture of the software?

The concept of adaptive compilers is introduced in Section 2. Section 3 describes briefly the concepts of software architecture and architecture description languages. Section 4 gives some possibilities, where using Lisp would help in writing adaptive compiler environments.

2. Adaptive compilers

Cooper et al. call an ordered list of selected transformations to be applied to the code to be compiled as the *compilation sequence* [3]. The traditional way of writing compilers includes one or several compilation sequences chosen by the developers of the compiler. The user of the compiler can then choose from these sequences, or do some small scale adjustments to the sequences by using command line arguments. However, these sequences tend to be general approximations, compilation sequences suitable for many domains instead of carefully adjusted to a particular domain, although exceptions can be found (for example, in the field of embedded software [3]).

Adaptive compilers try to find out the best possible compilation sequence. This can be done by changing the order or the parameters of the transformations in the compilation sequence. The objectives for the generated code usually are smaller size, faster execution speed or smaller power consumption. Data, on which the chosen compilation sequence is based on, can be gathered by executing the application and collecting run-time data[1], inspecting the results of the compilation, or analysing the source code.

Another branch of adapting compilers is the lazily optimising compilers [7], that optimise code when the need arises. When executing the code for the first time, compiler just compiles the code, making no detailed optimisations. During the execution of the programme, certain run-time data is gathered and analysed, and the compiler is invoked to recompile parts of code that have less than acceptable performance.

There exists a wide variety of adaptive compilers, from those written for languages like C/C++, Java and Oberon, to those written for languages like Scheme, Self and Smalltalk. These compilers are mostly of the run-time analysing flavour. What seems to be missing is compilers that adapt to a given description of the architecture of the software.

[1] Similarly to a testing method sometimes called *gamma testing*.

3. Software architecture and architecture description languages

One widely used definition for software architectures can be found in [2, p. 21]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements[2], the externally visible properties of those elements, and the relationships among them.

The architecture of the software is designed to ease the communication between the stake-holders, and to reason about the fulfilment of functional and non-functional requirements. Software architecture is documented from several different points of view. Use of UML and similar graphical notations completed with textual descriptions of details of the architecture is the current state of the art. Use of software architecture description languages has, alas, diminished since the middle of the 1990.

Software architecture description languages (ADL) are used to describe software architectures on a level that would make possible the manual or automatic reasoning about the architecture, and to aid in the construction of the software. ADLs are quite like Module Interconnection Languages (MIL) [4, 15], but compared to MILs, ADLs give more emphasis to connectors between elements of the architecture. This distinction is also valid between ADLs and programming languages or object-oriented modelling languages, such as UML. Arguably, UML can be extended to an ADL, but the success of UML can also be seen as a hindering factor for the development of ADLs. An extensive classification and comparison of ADLs was written by Medvidovic [13].

4. How Lisp can help?

Lisp could be used to implement an ADL, which would describe the architecture of the software written with the target language, and drive the compilation and instantiation of this software. The compiler for the target language should be written in Lisp to get most out of the Lisp based ADL.

4.1. Lisp as an architecture description language

Lisp has many features build into it to present many kinds of meta-data, from properties to the metaobject protocol [9]. Using these, an ADL similar to, for example, Acme [5] could be implemented. The architectural description of the software could then be used to choose a candidate for a compilation sequence.

[2] In older version of the referenced book, elements in the definition were called “components”.

As a side note, this architectural description could also be used to assure the architecture during the evolution of the software. The description of architecture could be compared to the structure of the software acquired by applying different metrics. This is the area of our research currently, first results published in [8]. Details of the method used can be found in [11].

4.2. Compilers written in Lisp

If a Lisp compiler for a specific language is written, a highly modularised architecture should be supposed. The developer should be able to easily change the compiler, for example, by adding and deleting transformations of compilation and optimisation, and by changing their order or arguments, even their implementation.

There has been quite an interesting events ongoing that could develop into a C language compiler written in Lisp. Recent thread in Usenet news group comp.lang.lisp, where Paul F. Dietz teased the readers by suggesting that the GNU C Compiler (gcc [6]) should be rewritten in Lisp[3]. Perhaps this was the primus motor for Brian Mastenbrook to challenge the welfare of comp.lang.lisp community with the announcement of *sexpc*, a s-expressions to C translation[4]. As Mastenbrook himself writes:

> sexpc is a program for translating a s-expression based syntax tree for the C language into actual C source code. [12]

4.3. Feedback from the executable

Aspect-Oriented Programming [10], AOP, which has its roots near the Lisp community, is a technique to handle the implementation of features like logging and memoizing, that are spread all around the code. AOP collects these features to clear abstractions, that are the woven into the code in points denoted by pointcuts. Aspects are usually written in a suitable language, which is not necessarily the same language the programme is written in.

Using AOP, the code to inspect the execution and report that back to the compiler can be woven into the programme by the weaver. This feedback is then analysed, the values of the objective function calculated, and the results reflected in the compilation sequence of the compiler. The feedback could also be used by comparing it to the description of the architecture of the software, to ensure that the run-time structure of the programme is valid.

[3] Message-id: `KfycncAJjJ7V-r6jXTWcqQ@dls.net`

[4] Message-id: `010320042329298654%NOSPAMbmastenbNO-SPAM@cs.indiana.edu`. sexpc is available at `http://www.common-lisp.net/project/sexpc`.

5. Conclusion

In this paper we have described adaptive compilers, and considered the possibility of compilers that could adapt their compilation sequence to match the architecture of the software to be compiled. This paper has presented some possibilities of research, not answers nor implementations of such kind of compilers. Also, the possibility of Lisp based architecture description languages, to instruct the compiler adaptation and to assure the architecture during the evolution of the software, was considered. The research should direct to implementing a Lisp based ADL, and to implement a compiler written in Lisp, that then could be used to experiment different architectures and their impact on compiling.

One interesting question is, how are software developers encouraged to use this ADL? Seems like ADLs are not quite high on the hype meter, so the ADL and compiler should give pretty solid results, that are competitive with the results of the current compilers. Not an easy task, but a task worth to seek for.

References

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno JVM. In *Conference on Object-Oriented*, pages 47–65, 2000.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.

[4] F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2):80–86, June 1976.

[5] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[6] GNU Compiler Collection, GCC. `http://www.gnu.org/software/gcc/gcc.html`.

[7] U. Hölzle. Adaptive optimization for Self: Reconciling high performance with exploratory programming, 1994.

[8] J. Itkonen, M. Hillebrand, and V. Lappalainen. Application of relation analysis to a small java software. In *Proceedings of CSMR 2004*, pages 233–239, Tampere, Finland, March 2004. IEEE Computer Society.

[9] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The art of metaobject protocol*. MIT Press, 1991.

[10] 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. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[11] J. Krajewski. QCR – A methodology for software evolution analysis. Master's thesis, Technical University of Vienna, April 2003. Available online at `http://www.infosys.tuwien.ac.at/teaching/thesis/online/Krajewski/krajewski.pdf`.

[12] B. Masterbrook. S-expression to C translation -manual, March 2003. Available at `http://www.common-lisp.net/project/sexpc/repos/sexpc/documentation.html`.

[13] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[14] M. P. Plezbert and R. K. Cytron. Does "just in time" = "better late than never". In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 120–131. ACM Press, 1997.

[15] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *J. Syst. Softw.*, 6(4):307–334, 1986.

[16] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 180–195. ACM Press, 2001.