*Published in: Proceedings «Second IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems», Williamsburg, Virginia Sep. 1-3 1993.*

# Control flow graph analysis for automatic fast path implementation

*Philipp Hoschka, Christian Huitema*
*INRIA, Projet RODEO, 2004, Route des Lucioles B.P. 93, 06902 Sophia-Antipolis Cedex (France)*
*e-mail: {hoschka | huitema}@sophia.inria.fr*

A common approach to building high performance protocol software is the implementation of so-called fast paths. A fast path increases the execution speed of the most frequently used path through the protocol software. However, today, a fast path implementation requires extensive manual performance analysis and intuition on behalf of the protocol programmer. This paper presents work in progress whose ultimate goal is to build a protocol compiler that automates fast path implementation. We argue that, starting from a formal description of the protocol, the standard compiler technique of flow graph analysis can be adapted to automatically predict the most frequently executed path in a protocol. This prediction is a prerequisite for automatic fast path implementation. We demonstrate our approach on the automatic implementation of a fast path in presentation conversion routines.

## 1. Introduction

A standard heuristic in computer science says that most programs spend 80% of their execution time in 20% of their code (80/20 rule). Protocol software is no exception to this rule, as indicated by the amount of recent work that deals with optimization of the most frequent case in different protocols (header prediction/header templates ([Clar89], [Jaco90], [John93]), integrated layer processing ([Clar90], [Abbo92b]), Morpheus language [Abbo92a]). Generally speaking, the goal of these efforts is to implement a so-called *fast path* through a protocol, i.e. to minimize the number of instructions in the most frequently executed path of the protocol.

Today, the protocol programmer usually determines the most frequently executed path manually by a combination of two approaches - profiling of existing protocol implementations and intuition on behalf of the programmer (e.g. by following heuristics such as «data transfer operations occur more often than control operations»). We believe that the prediction of the most frequently executed path in a protocol can be automated.

Automation of protocol optimization is especially important for the recent trend towards flexible, highly modular protocol architectures that allow for protocol configurations that are application-specific (e.g. [O'Mal92], [Hosc93], [Schm93]). With a configurable protocol architecture, the most frequently executed path changes with each protocol configuration, rendering manual frequency prediction and fast path implementation difficult or even impossible.

In this paper, we discuss an approach for automating the prediction of the most frequently executed path in a protocol.This prediction is based on control flow analysis. Analysis of the control flow graph of a program is standard practice in optimizing compilers for general-purpose programming languages [Aho86]. For a general-purpose program, the nodes of a flow graph consist of sequences of instructions with sequential flow of control that are terminated by a conditional or unconditional goto instruction. Goto's correspond to the edges in a control flow graph, which represent the flow of control in the program. By an analysis of a program's control flow graph an optimizing compiler can generally predict quite accurately the execution frequencies of the single nodes in the control flow graph, and thus control the application of code optimi-

sations.

Control flow graph analysis has to start from a formal description. In the area of communication protocols, the only formal descriptions widely used in practice today are the interface descriptions used to generate presentation conversion routines. Thus, in Section 2 we show how an important practical problem in the generation presentation conversion routines - the size/speed trade-off between procedure-driven and table-driven conversion routines - can be automated by flow graph analysis. In Section 3, we discuss the requirements and potential benefits of a generalization of flow graph analysis to a full protocol behavior specification. Section 4 contains our conclusion and outlook.

## 2. Automatic fast path implementation for presentation conversion

### 2.1 Presentation conversion

A standard technique for solving the problem of incompatible data representations in different modules of a distributed system is to use a common network data representation format, e.g. BER (Basic Encoding Rules) or XDR (eXternal Data Representation). In an application using a common network data representation, any data item has to be converted between the local data representation and the network data representation before it can be sent/received by the application.

A network data representation format generally offers a set of primitive data types such as integer, real and string types, and a set of type constructors to define constructed types. Type constructors offered are generally an operator to build record/struct types, an operator to mark components of a record/struct as optional, an operator to build a variant record/union types and an operator to build arrays.

Constructed types are generally described in an interface description language such as ASN.1 [ISO88], XDR [Sun86] or IDL [OMG93]. From an interface description language specification, a so-called stub generator can automatically generate the presentation conversion routines necessary to convert between the local and the network data representation.

### 2.2 Procedure-driven versus table-driven presentation conversion

Presentation conversion routines can be implemented by two alternative techniques: procedure-driven or table-driven. In a procedure-driven implementation, the conversion routine for a constructed type T consists of an explicit procedure call for each of the components of the type. In a table-driven implementation, the explicit procedure calls are replaced by a sequence of interpreter commands that are stored in a table. For each conversion of type T, the commands in the table are interpreted on protocol run-time by a generic interpreter routine.

Procedure-driven presentation conversion is much faster than a table-driven conversion. In a simple experiment with our ASN.1-based stub generator MAVROS [Huit91], we measured the time to convert an ASN.1 sequence type consisting of 10 integer fields from the local C data format to BER using a procedure-driven routine and a table-driven routine. The procedure-driven routine was about 6 times faster than the table-driven routine (720 us : 4250 us encoding time for 100 conversions on a SUN Sparc SS 10)[1].

However, procedure-driven conversion requires much more code space than table-driven conversion. Experiments at the University of British Columbia with a complex ASN.1 interface specification (X.500)

1. The difference in execution speed between table-driven and procedure-driven conversion are not specific to our ASN.1 compiler implementation. Similar experiments were conducted with the snacc ASN.1 compiler conducted at the University of British Columbia. In these experiments, procedure-driven conversion was found to be about 4 times faster than table driven conversion (about 350 s :1400 s for encoding and decoding the personnel record benchmark one million times on a MIPS R3260 ([Samp93a], [Samp93b])).

show that procedure-driven conversion routines can take up about 18 times more code space than the equivalent table-driven routines (480 KB : 27 KB object code size [Samp93b]).

Large code size of procedure-driven presentation conversion routines is a particular problem for machines with memory restrictions such as mobile systems or PCs. For instance, the problem occurs frequently in a PC implementation of the complex access protocols to OSI services such as X.400 electronic mail or the X.500 directory service.

A solution to optimize the size/speed trade-off between procedure-driven and table-driven routines is to implement a fast path for presentation conversion. Such a fast path uses procedure-driven conversion routines only for the most frequently executed conversion routines, and table-driven routines for the rest of the conversions. According to the 80/20-rule, fast path implementation for presentation conversion should work well in practice.

To confirm this intuition, we conducted a simple experiment: we randomly chose eighteen X.400 P1 PDUs (delivery reports and messages) from the message queue of the MTA at INRIA Sophia-Antipolis. We then calculated the average frequency of each type per message in the trace. Out of the 30 constructed types in the X.400 P1 ASN.1 specification eight types were on average used several times per message (all of the frequently used types were part of addressing and tracing protocol fields). Six types were on average used once per message, and eleven types less than once per message. Five of the 30 types were not used at all - consequently, their conversion routines were never executed for the trace data.

### 2.3 Analysis of presentation conversion flow graphs

One possible approach to control whether a stub generator generates procedure-driven or table-driven conversion routines is to leave the decision to the programmer. Several ASN.1 stub generators offer the programer the choice between table-driven and procedure-driven routines at least on the level of ASN.1 modules ([Onio89], [Samp93b]). However, a fast path implementation for presentation conversion with this approach requires a lot of effort on behalf of the programmer, in particular for a big interface specification with many type definitions. A better approach is to exploit the fact that interface descriptions are written in a formal language, and that this formal language can be easily be converted into a flow graph representation.

The control flow between the individual type conversion routines that are invoked for the conversion of a particular constructed type can be represented by a flow graph which we will call a *conversion flow graph*. In the conversion flow graph, struct-types that contain only primitive types correspond to strictly-sequential conversion instructions within a node. Type references correspond to edges. Recursive data types and arrays correspond to loop edges in the conversion flow graph. Union types and optional components of a struct type correspond to non-loop edges.

The conversion flow graph is input into a *prediction algorithm* that ranks the nodes of the conversion flow graph according to their predicted run-time frequency. Intuitively, we expect nodes that represent array types and recursive types to occur high up in the predicted ranking, since they are part of loops. Moreover, a type that is referenced often by other type definitions will very likely occur frequently on run-time (the importance of types with multiple references is another result of our analysis of X.400 P1 mentioned in the previous section). Finally, optional components of a structure type can be expected to occur less often on run-time than non-optional components.

For loop detection in the conversion flow graph we can choose one of the many loop detection algorithms developed for optimizing compilers [Aho86]. Moreover, the prediction algorithm has to analyze the acyclic graph that results from deleting all loop edges from the conversion flow graph. This is necessary to further refine the ranking of nodes, in particular in a flow graph that contains only a few loops, and many types with multiple references. A naive algorithm to predict the run-time frequency of a node x in an acyclic graph is to first count the number of different paths from the root to this node. Then, for each leaf node, we count

the number of paths from the leaf node to node x. The sum of these two numbers gives a good indication of how often the type represented by the node will occur at run-time. However, depending on the exact structure of the acyclic graph, the run-time of this frequency prediction algorithm might be high.

The ranking generated by the prediction algorithm can be used by the stub generator to decide whether the conversion routine for a particular type should be procedure-driven or table-driven. One possible code generation strategy for a stub compiler is to first generate table-driven routines for all nodes of the conversion flow graph. Then, table-driven routines are replaced with procedure-driven routines in the order of the predicted frequencies of the nodes, until some given upper limit on object code size is reached. Another, simpler strategy is to generate procedure-driven routines only for a fixed number of X most frequently used nodes. Additionally, other optimisations such as inlining of conversion routines for primitive types can be applied to the predicted most frequently executed nodes.

## 3. Extension to full protocol flow graph analysis

Frequency prediction based on a conversion flow graph will usually result in a prediction of good accuracy if the type specifications used are sufficiently complex. This is true for applications that exchange data types that have a complex structure such as e-mail messages. It is also true for data types that result from the combination of PDUs of several different layers according to the mapping rules of a particular protocol configuration.

However, basing the frequency prediction solely on the type information can be misleading, e.g. when an array type is contained in a PDU that is seldom used in the protocol. Moreover, some interface descriptions may not be rich/structured enough to allow a ranking of the types solely based on conversion flow graph analysis. Thus, the accuracy of the predictions should be further improved by analyzing a full *protocol flow graph*.

A protocol flow graph is a generalization of a conversion flow graph. It represents the dynamic behavior of the protocol. Thus, in addition to conversion instructions, a general protocol flow graph contains send and receive instructions as well as conditional branch instructions that test received values and local protocol state. A protocol flow graph corresponds to the specification of a protocol automaton, extended by conversion instructions. This information is not only useful to get better results in fast path implementation for presentation conversion, but also necessary to automate other fast path implementations such as header prediction that require data flow analysis of the protocol.

Several initial representations can be converted into a protocol flow graph as an intermediate representation. For instance, the protocol flow graph can be constructed from a flow graph of a program written in a general-purpose programming language by deleting all nodes from the general program graph that do not contain communication-specific instructions. Alternatively, the protocol flow graph can be constructed from a special-purpose protocol specification language that describes the protocol automaton and the PDU syntax of a particular protocol.

## 4. Conclusion and Outlook

We presented an approach for automating the prediction of the most frequently executed path in a protocol implementation. The automation is based on an analysis of a control flow graph that is derived from a formal specification of the protocol PDU syntax and the protocol behavior. Preliminary, simple experiments indicate that the standard control flow analysis approach to code optimization will be transferable to the optimization of automatically generated protocol software. Further confirmation will require measurements of the accuracy of the prediction algorithms on representative protocols and protocol traces.

***Acknowledgments.*** Ellen Siegel provided valuable comments and feedback. This work was made possible by an individual grant of the Human Capital and Mobility Programme HCM of the Commission of the Eu-

ropean Communities (grant no. ERBCHBICT920005).

## References

[Aho86] Aho, Alfred, Ravi Sethi and Jeffrey Ullman. Compilers - Principles, Techniques and Tools. Reading: Addison-Wesley, 1986.

[Abbo92a] Abbott, Mark and Larry Peterson. «A Language-Based Approach to Protocol Implementation.» SIGCOMM '92, 27-38, 1992.

[Abbo92b] Abbott, Mark and Larry Peterson. «Automated Integration of Communication Protocol Layers.» TR 92-24. Tucson: Department of Computer Science, 1992.

[Clar89] Clark, David, Van Jacobson, John Romkey and Howard Salwen. «An Analysis of TCP Processing Overhead.» IEEE Communications Magazine, June 1989, 23-29.

[Clar90] Clark, David and David Tennenhouse. «Architectural Considerations for a New Generation of Protocols». SIGCOMM '90, 200-208, 1990.

[Hosc93] Hoschka, Philipp. «Towards tailoring protocols to application specific requirements». IEEE INFOCOM '93 Proceedings, Volume 2. Los Alamitos: IEEE Computer Society Press, 1993, 647-653.

[Huit91] Huitema, Christian. «MAVROS: Highlights on an ASN.1 compiler». Internal working paper. Sophia-Antipolis: INRIA, Projet RODEO, 1991.

[ISO88] International Standards Organization. Specification of Abstract Syntax Notation One: ISO 8224, 1988.

[Jaco90] Jacobson, Van. «4BSD TCP Header Prediction.» Computer Communication Review, 20, 2 (April 90), 13-15

[John93] Johnson, David and Willy Zwaenepoel. «The Peregrine High-performance RPC System.» Software Practice and Experience, 23, 2 (February 93), 201-221.

[O'Mal92] O'Malley, Sean and Larry Peterson. «A Dynamic Network Architecture.» ACM Transactions on Computer Systems, 10, 2 (May 92), 110-143.

[OMG93] Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 1.1. OMG Document Number 91.12.1. 1993.

[Onio89] Onions, Julian and Marshall Rose. «The Applications Cookbook.» Stefferud, Einar, Ed. Proceedings of the Fourth International Symposium on Computer Message Systems. Amsterdam: North-Holland, 1989, 217-231.

[Samp93a] Sample, Micheal and Gerald Neufeld. «Implementing efficient encoders and decoders for network data representations». IEEE INFOCOM '93 Proceedings, Volume 3. Los Alamitos: IEEE Computer Society Press, 1993, 1144-1153.

[Samp93b] Sample, Micheal. Private communication.

[Schm93] Schmidt, D. B. Stiller, T. Suda, A. Tantawy and M. Zitterbart. «Language Support for Flexible, Application-Tailored Protocol Configuration». To be published in: 18th Conference on Local Computer Networks, 1993.

[Sun86] Sun Microsystems Inc. XDR: External Data Representation Standard. RFC 1014, SRI Network Information Center, June 1987.