Tempo, A Program Specializer for C

Position Statement at the Panel Session of the ACM Dynamo 2000 Workshop

Renaud Marlet

January 18, 2000

What it is / does?

Tempo is a partial evaluator for C. It automatically specializes programs with respect to partially known inputs. Specialization can be compile-time (i.e., source-to-source transformation) as well as run-time (i.e., binary code generation). Specialized programs are more efficient (and can be smaller) than the original programs.

Tempo has been applied to in various domains such as operating systems and networking, computer graphics, scientific computation, software engineering and domain specific languages, yielding significant speedups. Tempo is being developed in the Compose group at IRISA/INRIA Rennes. It is publicly available.

How?

To operate Tempo, the user must provide declarations regarding the binding time of the program arguments, i.e., which arguments are static (can be known in advance) and which ones are dynamic (will not be known until actual execution time). A dependency analysis (preceded by an alias analysis to safely treat C pointers) then determines what parts of the program can be precomputed assuming static arguments are available. This information can be visualized by the user to assess the amount of specialization in the program.

Then, given actual values for the static arguments, all precomputable fragments of the program is evaluated ; program fragments that may depend on dynamic arguments (called code templates) are left untouched. Putting these precomputed values and code fragments together results in a specialized program.

For compile-time specialization, the dynamic code fragments consist of source code, and precomputed values are turned into their C source form. The result is the source code of the specialized program.

For run-time specialization, the dynamic code fragments are first precompiled into binary code templates using a standard C compiler. A dedicated specializer is then generated by Tempo. Running this specializer on actual values for the static arguments assembles the binary code fragments and the values being precomputed, generating a binary code that can be executed on the fly.

Why you think yours is the best approach?

The key advantages of Tempo regard features and engineering.

Automation and safety. Specialization with Tempo is automatic and non-intrusive. The user does not have to manually annotate each function to specialize, which is error-prone and makes program maintenance more difficult. The user only has to specify in Tempo configuration files: the binding time of the parameters of the specialization entry point (the root function), and the behavior (regarding binding time and side-effects) of external functions, if any. All specialization actions are then determined automatically from this initial specification.

Two manual operations are required though: slicing the part of the original program that the user wants to specialize, and installing specialized functions into the original program.

Precision. The program analyses in Tempo are accurate: they are inter-procedural and sensitive to flow, context, return, use, and structure fields.

Predictability. Tempo provides a visual representation of the program analysis results. This ensure some predictability of the transformation process: the user can assess the amount of specialization in the program.

Low breakeven point. Generating code at run time with Tempo is very fast although not optimal: code generation mainly consists in assembling precompiled binary templates. (There are little inter-template optimizations.) The resulting breakeven point (number of time the specialized code has to be used for specialization to be worth it) is typically between 3 and 50.

Easy engineering. Building a run-time specializer using Tempo's approach requires little effort: it re-uses without any change an existing, efficient, well tested, native compiler.

Regrets, if any

The precision of the analyses cannot be parameterized. (An option for doing analyses with structure field polyvariance is being worked on though.)

Tempo works on an abstract syntax tree. It includes a `goto` elimination phase to make sure that all `goto`s are rewritten into `while` loops. In some (fortunately, rare) cases, this transformation can shift static statements under dynamic control, which turns them dynamic. It thus reduces specialization opportunities. On the other hand, working on an AST is easier than dealing with a graph of basic blocks.

Tempo is a raw specialization engine. For large projects, support is needed for slicing programs (what part of the code the user wants to specialize) and installing specialized functions (when and where should a specialized function be made and run, in place of the original code). But this is more a "future work" than actual regrets.

As Tempo is just a prototype, clumsiness had sometimes to be "allowed" in the implementation. (In 5 years, more than twenty people somehow participated in the development of Tempo, most of which were students.)