Feedback-Directed Specialization of *C* (M.Eng. Thesis Abstract)

Jeremy Brown
jhbrown@ai.mit.edu

MIT AI Lab
545 Technology Square, Cambridge, MA 02139
May 12, 1995

Overview and Justification

Abstraction is a powerful tool for managing program complexity. Well-abstracted source code is far more comprehensible, maintainable, and reusable than less abstract, specialized code. In addition, because a single general routine may be called from several places within a program, a well-abstracted program is usually smaller than a more specialized one. Unfortunately, with traditional compiler technology, executables produced from hand-specialized source code generally run faster than executables produced from abstract source code; as a result, programmers frequently sacrifice abstraction's long-term virtues to the short-term goal of execution-speed.

In my M.Eng. thesis, I am studying automating specialization on the basis of profiling information gathered during actual program runs. In this approach, a given program is occasionally recompiled by a compiler which utilizes profiling information from previous program runs to specialize the program to respond optimally to observed common usage-patterns. The compiler also inserts new profiling instructions into the new executable to direct the next round of recompilation. This approach, dubbed feedback-directed specialization (FDS), has two important benefits: first, it responds dynamically to changing usage patterns; and second, recompilation is not part of runtime, so optimization can be extremely aggressive (and time-consuming.)

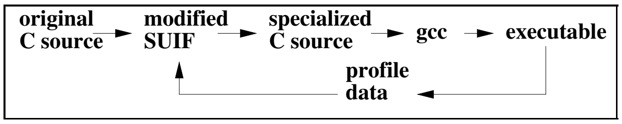
Previously, automated, dynamic response to program usage has been most visible in dynamic code generation [EP93] (DCG) systems. Executables using DCG wait until runtime to generate critical portions of code; such code has the advantage of being precisely tailored to the operating conditions actually in effect. There are two drawbacks to DCG, however: first, particular portions of code may be generated with great frequency because of characteristic user behavior; second, exacerbating this inefficiency, aggressive optimization takes too much time to perform at runtime on dynamically generated code. The result is that critical code may be generated in nearly every run of a program, but will never fully optimized. By specializing on frequently-occuring conditions, FDS avoids the first problem, albeit at the cost of having to fall back on less-specialized routines when unexpected runtime conditions occur; by recompiling outside of runtime, FDS avoids the second problem as well.

In contrast to the automated, dynamic approach taken by DCG- and FDS-based systems, the traditional method of cutting down the runtime costs of abstraction has been to specialize source code by hand. In this method, the programmer uses profiling tools to examine the behavior of a program run on test data. Profiling results are used to guide hand-optimization of the source, which is then recompiled. The advantage of this approach is that a human can change a program in arbitrarily complex and intelligent ways; optimizations may be as severe as a complete change of algorithm. Unfortunately, any hand-specializations based on profiling runs on test data may actually worsen program performance if the test data fails to accurately model end-users' data. In addition, overly specializing source code turns it into "spaghetti" code – messy source code which is hard to comprehend, maintain, or reuse. In short, when performing hand-specialization, a programmer must balance the need for executable speed against the need for future maintenance of the source code. FDS avoids all of these problems by responding automatically to actual usage conditions.

An FDS system for C

Acknowledgments: This work is conducted under the direction of Principal Research Scientist Thomas Knight Jr. This research is supported by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252. This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

For my M.Eng. thesis, I am developing a feedback-directed specialization system for the C programming language; a system flow-diagram is shown below.



SUIF [WFW+] is a research compiler designed for easy modification and extension. The modified SUIF pass performs specializations on the original, abstract C code, directed by profiling information gathered from previous program runs; it also inserts profiling instructions into its new output in order to generate feedback for the next recompilation. SUIF performs source-to-source transformations; compilation of the C output by SUIF is performed by gcc [Sta93]. Source-to-source transformation has the advantage that benchmark-comparisons against unspecialized programs compiled by gcc are straightforward, but the disadvantage that certain information (such as information about variable aliasing) cannot be represented in C.

Specializations

The feedback loop enables specializations based on empirically discovered data- and control-flow-behavior. In my system, **data-behavior based specialization** operates on programmer-annotated code blocks.[1] The annotations suggest sets of constant values that may be taken on by sets of variables in the annotated block. The resulting executable contains profiling tailored to verify the accuracy of the annotation-suggestions; subsequent recompilations implement a version of the annotated code block specialized for each data set which proves likely. The profiling also attempts to determine other likely data values for the programmer to consider in subsequent annotations. At program runtime, a light-weight case-selection mechanism chooses an appropriate specialized version of the block if applicable, or falls back upon the general case if conditions require it.[2]

In **control-flow based specializations**, feedback indicates where certain expensive specializations are profitable; the specializations enabled are "expensive" because they results in increased code size. Such specializations cannot be used by standard compilers, because without knowledge of actual program usage parameters, code-size must be generically minimized. In my system, feedback indicates points at which using such specializations speeds up the resulting executable enough to justify the increased code-size. My project utilizes two specific expensive specializations: function inlining, and generating specialized versions of general functions.

In my system, inlining is performed by individual call-site. This provides greater precision than do popular compilers such as gcc, which at best allow the programmer to suggest that certain functions be inlined everywhere they are called. The spatial cost of my system's style of inlining is $\leq$ the size of the inlined function's body per call-site inlined.

Generating specialized versions of functions has no direct counterpart in standard C compilers; the nearest analogy is perhaps using templates in C++ to produce specialized classes from general frameworks. To generate a specialized version of a function, a copy of the original function is specialized with respect to a call-site such that all constant arguments previously passed by the call are eliminated – all such constants are placed in the new copy's body, and pushed through with standard optimizations. As a result, the new version should be shorter and faster than the original function, and the caller need not pass any constant parameters. The spatial cost of generating a specialized version of a function is $\leq$ the size of the original function's body per specialized version; this cost is amortized over the number of similar call-sites that can also use this specialized version.

For a given call, the SUIF pass selectively employs several forms of profiling to decide whether to inline, generate a specialized version of the target function, or leave the call alone. These forms of profiling include PC sampling, call graph determination, and block execution frequency; the data collected tells the compiler how beneficial various specializations proved to be, allowing it to weigh these benefits against the corresponding increases in code size. The feedback reflects operating constraints from outside of the executable itself; for instance, if a specialization increases the code-size such that caching or paging effects slow execution, the slowed results will be reflected in feedback from subsequent runs. When these effects are observed, the compiler can back out specializations until the code's size has been adequately reduced.

For more information on this project, see:

Related thesis materials:
`http://www.ai.mit.edu/people/jhbrown/Thesis/home.html`

Reinventing Computing home page:
`http://www.ai.mit.edu/projects/transit/rc_home_page.html`

References

[EP93] Dawson R. Engler and Todd A. Proebsting. DCG: An Efficient Retargetable Dynamic Code Generation System. contact: todd@cs.arizonda.edu, engler@lcs.mit.edu, November 1993.

[Sta93] Richard Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., 675 Massachusetts Avenue, Cambridge, MA 02139, October 1993.

[WFW+] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, , and John Hennessy. An Overview of the SUIF Compiler System. Anonymous FTP `suif.stanford.edu:/pub/suif/suif-overview.ps`.

[1] In the future I'd like to get rid of the need for any programmer annotation of the original source, but that's a post-thesis problem.

[2] All specializations in my system are semantics preserving – the general case is always preserved where necessary to maintain full semantic equivalence with the original program.