Selecting better-performing alternative code using run-time profiling feedback

Ping-Shun Huang <pshuang@ai.mit.edu>*
May 1, 1995

MIT Artificial Intelligence Laboratory
545 Technology Square, Cambridge, MA 02139

Abstract

The underlying theme for work in the Reinventing Computing group is exploring software and hardware technologies engineered to adapt to the underlying changes in base technologies that will radically change the computing landscape of the 21ST century. Quasistatic computing is a term we have coined to describe the effects of incorporating technologies into fielded computing systems which will allow them to evolve while they are in actual use, rather than remain rigid over their lifetimes unless manually upgraded. These technologies would track shifts in patterns of usage and optimize accordingly, and adapt software to the performance characteristics of computing platforms, which change both over time and between contemporary individual machines.

One way of obtaining the benefits of quasistatic computing is to modify how compilers work. Today's optimizing compilers have come an astoundingly long way in the past decades, by increasing the scope and sophistication of analysis and program transformations, but they rely almost entirely on static analysis. Static analysis can only give incomplete and inaccurate estimates of program performance at best. Runtime profiling feedback can provide a much more complete and accurate picture. With additional information, the compiler can then try more radical, speculative transformations, and ascertain which proved to be beneficial in a particular instance. In other words, incorporating feedback allows the compiler to run experiments.

Ideally, a compiler supporting quasistatic computing would take unannotated source code and automatically analyze it for potential loci for this kind of experimentation. A stepping stone to such a *smart* compiler would be a *clever* compiler — one which relies on programmer annotations to help it decide what profiling feedback to gather and how to organize its analysis. Although this requires additional manual effort, note that using a *clever* compiler should still be much easier than traditional manual performance optimization, since the programmer does not need to examine profiler output by hand, and retains the quasistatic computing advantage that the program will dynamically evolve itself for different usage patterns and different computing platforms.

My M. Eng. thesis deals with implementing several such annotations for C source code: quasistatic `if-then-else` and quasistatic parameters. The quasistatic `if-then-else` annotation (`qif`) looks like a normal C `if` or `if-else-if` chain, listing alternative pieces of code to be run when that part of the

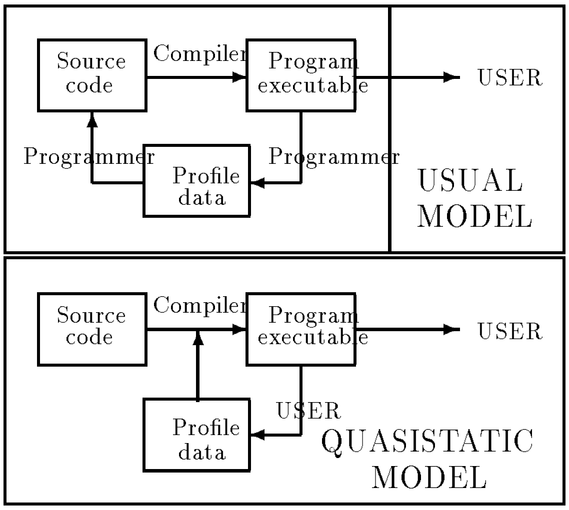

Figure 1: Automating the profiling feedback loop.

*This work is conducted under the direction of Thomas Knight Jr. Ping-Shun Huang and research in the Reinventing Computing group are supported in part by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

program executes. However, instead of the program making the choice at run-time, the compiler chooses one of the pieces of alternative code at compile-time. The quasistatic parameter annotation (`qint`) looks like a normal C integer definition, except that instead of optionally specifying a single initial value, the programmer specifies the possible range of values which the parameter may take on, and then writes the rest of his or her code so that it will function correctly for any value of the paramter from that range. The programmer can give additional hints to the compiler by adding `qinfluence` annotations that pick out the parts of the program that are most strongly affected by particular `qint`'s.

Current Status

I have implemented support for these annotations by modifying existing passes and writing additional new passes for SUIF, a C compiler from Stanford University. SUIF was chosen because it was intended for supporting compiler research efforts, and as such, has some nice features including a well-defined tree-structured intermediate format, which makes performing program transformations relatively easy; its disadvantages include the small size of its user community, backend support for MIPS-based systems only, and relatively weak traditional scalar optimizations by today's standards. However, by using SUIF to perform source-to-source transformations and then compiling with the GNU `gcc` compiler, several of these disadvantages are ameliorated.

At the current time, I've finished constructed the supporting system framework and modified and written the necessary SUIF passes. Over the next few weeks, I hope to illustrate that these annotations allow a programmer to obtain noticeable improvements in program performance and across different platforms with reasonably minimal effort, both by writing demonstration programs from scratch and by modifying existing bodies of code (e.g., benchmarks from the SPEC92 suite). I also plan to work on improving how one of the new SUIF compiler pass chooses between the different code alternatives and different parameter values; since the alternatives and parameters interact to create an potentially exponentially large "search space", some of the standard space-searching heuristics probably should be applied (e.g., hill-climbing). More domain-specific search heuristics can be developed: for example, alternatives and parameters which can only affect performance of parts of the program which are not hot spots can be ignored *a priori* during the searching process.

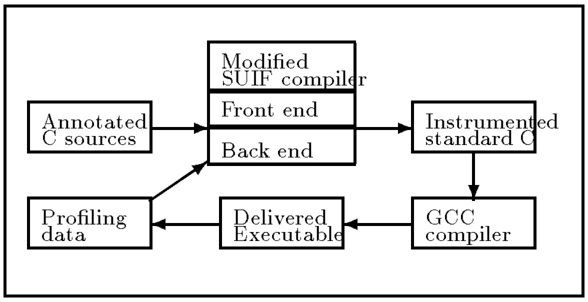

Figure 2: Quasistatic constructs implementation.

Even after those improvements, there would still be many problems with this instantiation of quasistatic computing ideas; for example, the system constructed requires that software publishers provide an intermediate form of their programs, which may be more prone to reverse-engineering, and that end-user systems be equipped with compilers. However, the core notion being explored is more general, and hence implementation flaws should not detract from the potential benefits of incorporating experimentation and observation into optimizing compilers.

Note

A copy of my completed thesis should be available several weeks after the thesis deadline (26 May 1995). If you have access to the World Wide Web, check the URL http://www.ai.mit.edu/projects/transit/rc_home_page.html. If you do not have WWW access, or if you have problems finding it, please contact me by email at `pshuang@ai.mit.edu`.

References

[BDE94] Jeremy Brown, Andre DeHon, Ian Eslick, et al.: "The Clever Compiler: Proposal for a First-Cut Smart Compiler." Transit Note 101, MIT Artificial Intelligence Laboratory, January 1994.

[D94] Andre DeHon: "A Proposal for Alternative Code: a quasistatic-if-then-else and quasistatic variables." Transit Note 110 (working draft), MIT Artificial Intelligence Laboratory, July 1994.

[K71] Donald E. Knuth, "Empirical Study of FORTRAN Programs." *Practice and Experience*, vol. 1, pg. 105-133.

[S91] Alan Dain Samples, *Profile-driven Compilation.* U.C. Berkeley CSD-91-627, April 1991.