Fortran Compilers

David Padua

University of Illinois at Urbana-Champaign

1. Introduction

The success and the glory of Fortran are its compilers.

"It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger. This belief caused us to regard the design of the **translator as the real challenge**, not the simple task of designing the language. ... To this day I believe that our **emphasis on object program efficiency rather than on language design** was basically correct. I believe that has we failed to produce efficient programs, the widespread use of language like FORTRAN would have been seriously delayed."

John Backus
FORTRAN I, II, and III
Annals of the History of Computing
Vol. 1, No 1, July 1979

- “Object programs produced by FORTRAN will be nearly as efficient as those written by good programmers.”

Programmer’s
Reference Manual
October 15, 1956

Developing effective Fortran compilers was not easy and success was not guaranteed.

“Like most of the early hardware and software systems, Fortran was **late** in delivery, and **didn’t really work** when it was delivered. At first people thought it would **never be done**. Then when it was in field test, with many bugs, and with some of the most important parts unfinished, many thought it would **never work.**

It gradually got to the point where a program in Fortran had a reasonable expectancy of compiling all the way through and maybe even running. This gradual change of status from an experiment to a working system was true of most compilers. It is stressed here in the case of Fortran only because Fortran is now almost taken for granted, as it were built into the computer hardware."

Saul Rosen

Programming Languages and Systems

McGraw Hill 1967

2. The old compilers

- Early IBM Fortran compilers (Fortran I, Fortran H) were engineering marvels.
- They introduced the seed of many compiler techniques. These led in time to powerful, general translation algorithms which are among the most beautiful creations of Computer Science.

2.1 Early techniques

- Fifty years ago, it was an open field. Practically every issue was an open problem.
- Early compiler algorithms were less general than today's algorithms
- Two examples next.

2.1.1 Operator priority

- “The lack of operator priority (often called precedence or hierarchy) in the IT language was the most frequent single cause of errors by users of that compiler” D. Knuth
- The Fortran I compiler solution:
  - Replace + and − with)) + ((and)) − ((
  - Replace * and / with) * (and) / (, respectively
  - Add ((and)) at the beginning and end resp. of expressions and “sub expressions”
- “The resulting formula is properly parenthesized, believe it or not” D. Knuth

2.1.2 DO loop optimizations

- One of the Fortran I compiler's main objective was *“to analyze the entire structure of the program in order to generate optimal code from DO statements and references to subscripted variables”.*
- Much of this is accomplished today by applying *removal of loop invariants*, *induction-variable detection*, and *strength reduction*.

- The Fortran I compiler applied a single transformation that simultaneously moved subexpressions involving loop indices to the outermost possible loop level and applied strength reduction.
- Only loop indices were recognized as induction variables. *“not practical to track down and identify linear changes in subscripts resulting from assignment statements”*

2.2 Life and compilers were simpler then

- Fortran I was only 23,500 assembly language instructions and was developed by only 6 people over three years.
- Today's compilers are much longer.
- Target machines were simpler: No caches, no parallelism, no vector extensions.

3. Fortran compilers today

- “Pure” Fortran compilers are less common today. Optimization passes are shared with other languages.
- Today, machines have become more complex
  - Vector extensions (SSE, Altivec)
  - Parallelism (multicores)
  - Caches
- Programs are more difficult to analyze due to widespread use of pointers/references and other factors.

3.1 How well do they work ?

- Evidence accumulated for many years show that compilers today fail to meet our expectations.
- Problems at all levels:
  - Detection of parallelism (numerical computing)
  - Vectorization
  - Locality enhancement
  - Uniprocessor conventional optimization algorithms (scalarization, cse, …)
- Today, most compilers include vectorization and parallelization techniques, but no clear way forward. Diminishing returns.

3.2 Why ?

- Compilers for conventional languages suffer because of
  - Inaccurate program analysis
  - Ad hoc optimization strategies
  - Uneven implementations

3.3 Improvements are needed

- With today's compiler technology, most likely, widespread parallelism will give us performance at the expense of a dip in productivity.

3.4 Today's challenge

- We face a challenge similar to that of 1957.
- Like in the 1950s need to improve programmability for performance. After all, performance is what multicore is all about.
- Like in the 1950s there is much skepticism.
- Against progress in program optimization we have
  - The myth that the automatic optimization problem is solved or insurmountable.
  - The natural desire to work on fashionable problems and "low hanging fruits"

- Perhaps the solution should resemble that of Fortran I: **Language/compiler co-design**.
- And certainly solving the problem will be as rewarding as it was for the Fortran I team:

  "In any case the intellectual satisfaction of having formulated and solved some difficult problems of translation and the knowledge and experience acquired in the process are themselves almost a sufficient reward for the long effort expended on the FORTRAN project."

  1957 Western Computer Proceeding

  Paper on the Fortran I compiler