A Short Review of High Speed Compilation

Werner Aßmann

Academy of Sciences of the G.D.R.
Institute of Informatics and Computing Techniques

Abstract:

Starting with some considerations about the need for a higher speed of program development the term "High Speed Compilation" will be introduced. High speed compilers should have a speed of more than 30,000 lines per minute. High speed programming systems must be able to run through the edit/compile/link/load/run/debug phases with a waiting time less than 10 seconds. Starting with this definitions some demands on the components of compilers and programming systems are derived. Current known and applicable methods to reach this goal are discussed. This methods are divided in "Tuning" and "Better Algorithms". The effect of incremental compilation as a newer compilation technique is shown by the INDIA programming system.

1. The Need for High Speed Compilation

The latest (and sometimes longest) stage of program development takes place in the life cycle phases

edit - compile - link - load - run - debug.

A big portion of the time needed for one cycle is only waiting time. The following example will demonstrate this situation.

To run through the above phases we needed in case of the INDIA editor (2330 lines MODULA-2) as a part of the INDIA programming system (consisting of 65 modules, code size 270K) in a production environment (many and big directories, commercial MODULA-2 compiler) on a AT-Compatible the following times:

Editor (without any actions):	*15 sec.*
Compiler:	*156 sec.*
Linker:	*221 sec.*
Start (without any actions):	*22 sec.*
Debugger (without any actions):	*139 sec.*
Total:	*553 sec.*

That means that in the above example the waiting time amounts to more than 9 minutes - lost time! It's worth mentioning that this situation didn't occur in the past with its batch processing environments.

Reason 1: Because of working in interactive environments the waiting times must be reduced to a minimum: Waiting time is lost time!

We have two other reasons to want high speed compilation. The permanently encreasing complexity of tasks solved by computers leads to much bigger programs and program systems. On the other hand the technological evolution rate demands a higher evolution rate of some kinds of programs too. Clearly we must force up the efficiency of programming.

Reason 2: The tendency to bigger programs and a higher evolution rate.

Last not least we must have a look on the used programming languages. The newer programming languages such as MODULA-2 cause a much higher effort in the compiler (and in the other components of the programming systems). Using the traditional techniques would lead to compilation times not acceptable by the users. This development will continue in the next time. The change-over to descriptive programming techniques will complicate the compiling task another one. To solve this problem new compilation techniques must be developped. Compare under this aspect the symbol table handling in FORTRAN and MODULA-2!

Reason 3: The application of higher developped programming languages.

2. What Means "High Speed Compilation"?

Clearly no exact definition of the terminus "High Speed Compilation" can be given. The ideas of the users about high speed will be very different too. Therefore a very pragmatic definition is only possible The starting point is to make some assertion about the maximal waiting time for the result of a compilation.

Our rather arbitrary definition of high speed compilation prescribes that the compilation time for a medium sized module of 2,500 lines should not exceed a period of 5 seconds. This is equivalent to:

Compilation speed >= 30,000 lines per minute.

But the above example shows that high speed compilation is only a part to reach short cycle times in the program development process. By including all the other components of a programming system such as editor, linker, loader, and debugger we can give an extended definition:

The waiting time in a "High Speed Programming System" for a full cycle must be <= 10 seconds.

The conclusion is the need not only for high speed compilers but also for

- **High Speed Loader,**
- **High Speed Linker,**
- **High Speed Debugger,** and last not least
- **High Speed File System**

as the bottleneck of all the system components.

Because the power of the used computer has a big influence of the needed time another slight different version of this definition will be given. The use of this definition lies in better possibilities comparing different products or to analyse the effort of different algorithms. The idea is the reduction to machine cycles. Using a computer with a clock rate of 25 MHz and a average number of 5 machine cycles per operation we get a maximal number of operations per line for the compiler:

Compilation effort <= 10,000 operations per line.

3. Demands on Compiler Components

If we suppose the traditional (logical) subdivision of a compiler into the components

- Lexical analysis (25%),
- Syntactical analysis (15%),
- Symbol table handling (25%),
- Context checking (10%), and
- Code generator (25%)

we can find out the necessary speed of all the components considering their share on the total effort:

- Lecical analysis: >= 120,000 lines per minute
- Syntactical analysis: >= 200,000 lines per minute
- Symbol table handling: >= 120,000 lines per minute
- Context checking: >= 300,000 lines per minute
- Code generator: >= 120,000 lines per minute

or according to the second definition:

- Lexical analysis: <= 2,500 operations per line
- Syntactical analysis: <= 1,500 operations per line
- Symbol table handling: <= 2,500 operations per line
- Context checking: <= 1,000 operations per line
- Code generator: <= 2,500 operations per line.

The percentual effort of the compiler components is estimated for programming languages like MODULA-2. For other languages and other compiler architectures other figures are possible so that more or less different results will be obtained. But essential is only the order of magnitude which can be used for statements about the quality of components.

To go a little more into details we look to the lexical analysis. If we suppose a number of 40 characters per line we have to perform the work of lexical analysis with <= 62 operations per character.

The lexical analysis of the INDIA system needs for a module of 892 lines respectively 26521 characters 3.300 seconds on a computer with 6 MHz. This means a effort of 149 operations per character. This is surprisingly not so far away from the goal figure especially because no special tuning was used till now.

4. Review of Methods in General

Three nearly orthogonal methods to increase the speed of compilers (and other programs too) exist:

- Faster hardware,
- Tuning, and
- Intelligent algorithms and data structures.

No further comment to the first method - faster hardware - will be made. Not to mention technological and economical bounds it seems unsatisfactoryly to software developpers to spent computer time without any need.

4.1. Tuning

This term will be used for methods to optimise components of a programming system such as a compiler without general changes. In principle this method can be accomplished by persons without any knowledge about the function of the program for instance by code inspection.

One simple method of this class is **Optimisation** of the program. In principle all methods known under this term from compilers can be used. The problem often lies in the contradiction to principles of structured programming. But sometimes a optimisation in space, time, and structure can be reached because some programs tends to disorder after a certain life time. Such optimisations can be:

- **Loop optimisation**
  The elimination of unaffected statements from the loop is well known. The same holds for the computation of as much as possible before entering the loop. Sometimes additional variables solve some problem. All kinds of interpreter loops must be optimised in this way.
- **Procedure call optimisation**
  The call of procedures causes in languages with recursive procedures a lot of additional actions. Another critical point is the tranfer of parameters. Therefore often called procedures should be changed to inline code or work on the base of global variables.
- **Location of variables**
  Unseen mostly by the programmer the use of imported variables and of variables declared in nested procedures is much more expensive as the use of local or global declared variables.
- **Array optimisation**
  Multidimensional arrays cause a lot of work to compute the target address. The better way is reduction to one-dimensional arrays or substituation by pointers. The last way is the most effective one but sometimes a little dangerous.

- **Reduction of run-time checks**
  Good compilers insert a lot of run-time checks into the code. This arrangement increases the security of the programs considerable. But in many cases these checks can be eliminated after a certain stage of program correctness on the base of semantic information about the domain of variables and so on.

The effect of program optimisation in this way can be estimated by analysis of the machine code of the compiler component. Some interesting insights into the properties of the used compiler (to compile the components of the programming system) are possible. The result is **Assembler-like programming**:

- **Use of efficiently compilable constructions only.**
  In dependence on the properties of compiler and machine code only such constructions of the implementation language will be used which will be translated to very efficent code. A good example is the problem of using a CASE-statement or a IF-ELSIF-ELSIF-...-statement.

- **Adaption to the used hardware.**
  The underlying components such as file system and screen handler must use very efficient access methods to reach a sufficient speed. Portability is in a direct opposition to this point.

Other methods must be used in case of interpreter techniques. By **Optimisation of abstract machines and abstract machine code** (as another word for the pairs of interpreter and associated control table) very surprising results can be reached:

- **Optimisation of control tables.**
  Interpreter techniques are in general use for the syntactical analysis by the LR(1) or LALR(1) method but also for some other components. By optimisation and/or compaction of these control tables the same effects as in a "normal" optimisation (of code for a real machine) can be reached.

- **Translation into direct executable machine code.**
  By this method the abstract machine code (the control table) will be transformed into direct executable machine code. The interpretation of the table by the abstract machine is omitted. The big size of the generated programs is a little problematic but together with optimisation very good results are reachable (see [7], [10], [16]).

4.2. Intelligent Algorithms and Data Structures

Computer science is a relatively young field. Big efforts are made in development of effective algorithms. Clearly the end of this development is not reached already. Sometimes the skilful combination of known principles has a multiple effect, sometimes a lot of research is necessary to find out a good solution for a given problem. In the field of programming systems a lot of work was done in the last years to increase the comfort (integrated programming systems, multi-window techniques and so

on). Only a few methods are characteristic of high speed efforts:

- **Use of fast search methods.**
  Hash search seems to be the fastest search method. The final result on this field is minimal perfect hash search using such a hash function that only one attempt for the search is necessary ("perfect") and there are no gaps in the hash table ("minimal"). Using this technique in the lexical analysis for the decision whether keyword or identifier should be the optimum. Sometimes the use of hash search is not possible without special precautions (sym- bol table handling in incremental compilers!).

- **Use of straightforward algorithms.**
  Algorithms without the need of decisions which branch to choose or without the need to transform data from one structure to another one are clearly better because the minor overhead. The use of minimal perfect hash functions is a good example for such algorithms.

- **Use of better system architectures.**
  Integrated programming systems working on a common data base can reach a much higher speed as systems divided traditionally into a lot of independent components. Some systems use syntax-oriented editors with a data structure which makes the task of lexical analysis superfluous.

- **Use of incremental techniques.**
  This method is the newest and most interesting one. The principle is not to throw away the results after doing some computation but to save all reusable data. These data will be used if the same computation task (with slightly different input data) arises a second time. The work of such systems is clearly more complicated but the effects are substantial. Compilers are very good candidates for this technique because of the frequent recompilation of programs with only a few changes.

The effects of incremental compilation techniques can be shown by the first results of the INDIA programming system (see [1] for further information):

The front-end compiler of INDIA works already incremental, the back-end compiler generates code for the whole program only. For the compilation of a small MODULA-2 module of 60 lines the following figures were measured (on a 6 MHz computer):

Normal (full) compilation:
10.160 sec. resp. 354 lines per minute.

Incremental deletion of one line (assign statement):
1.590 sec. resp. 2226 lines per minute
(0.220 sec. resp. 16091 lines per minute for front-end).

Incremental changing of one line (assign statement; deletion and following compilation of this line):
2.250 sec. resp. 1653 lines per minute
(0.880 sec. resp. 4091 lines per minute for front-end).

From this example we can learn that by incremental compilation

techniques compilation times proportional to the amount of the program changes and not proportional to the program size can be reached. That also means that in a good integrated programming system the compiling process can be done during the editing process (during the waiting time for the next input). From the user's point of view the compilation time is exactly null in such a system - a good contribution to a high speed programming system!

5. Review of Special Methods for some Components

The most methods referenced above can be used in all the components of a compiler or a whole programming system. But for some components very special methods are usable. In the following the attempt will be made to gather some of such methods without any pretensions to completeness.

The **Programming System Architecture** has nearly the biggest influence. The best way should be an

- Integrated programming system with a
- Commonly used internal program representation.

The **Compiler Architecture** should take into consideration the following points:

- Very few passes (but sometimes dependent on the language)
- No overlays (problem of available memory).
- Incremental work.

The **Lexical Analysis** should use

- Clever input medium (if every possible the editor data base, situated in the memory).
- Few touches of the input characters (the best solution is only once!).
- Keyword decision by MPHF (minimal perfect hash function).
- Transfer of the whole lexical analysis (or of parts) to the editor or to the syntactical analysis.

In the **Syntactical Analysis** two basic methods are available:

- LALR(1)/LR(1) or recursive descendent procedures. Newer results show that LALR(1) should enable high speed together with the other advantages.
- Elimination of chain productions (if LALR(1) method).
- Clever expression handling.

As always there are scarcely general methods for the **Semantic Analysis**:

- Limited number of "action points" during the syntactical analysis.
- Handling of imported objects (if there is a intermodule context checking) as "direct loadable" data structures (no additional data transformation!).

In the **Symbol Table Administration** search methods are the most important point:

- Use of hash search methods.

- No division into name and symbol table (double search!).

The **Code Generator** should work on the base of

- Direct code generation

because template techniques are mostly to slowly. Another solution could be the use of very fast pattern techniques.

A few statements about the other components of a programming system can be made. The **Editor** should use:

- Syntax-oriented techniques to reduce the input time and to increase the reliability.
- Data structures which reduce the work of lexical analysis.

The only possibility for the **Linker** seems to be

- Incremental linking.

To reduce the effort of the **Debugger** to gather all the necessary information about the program to test only one possibility exist:

- Use of "direct loadable data structures"

At last a remark to the underlying **File System**. If the cleverness of the file system is not high enough to handle big systems some effort is necessary such as

- Own directory handling on the base of hash search techniques.
- Own buffer administration (transfer directly into the compiler data areas).
- Use of basic access functions to the file store.

6. Results and Conclusion

Currently some results about high speed components are available. In [8] a fast lexical analyser is described with a speed 6 times faster than LEX. The scanner generator Rex described in [7] generates scanners with a speed of 180,000 to 195,000 lines per minute running on a MC 68020 processor. Much faster parsers are available: the LALR(1) parser generated by the Lalr tool of [7] has a speed of 400,000 lines per minute, another LL(1) parser generated bei Ell [7] on the base of recursive descendent procedures even 900,000 lines per minute. The LR(1) parser of [10] reaches a speed of 500,000 lines per minute on a VAX 11/780 resp. 240,000 lines per minute on an Intel 80286. In [16] a LR(1) parser with a speed of 450,000 lines per minute running on a SUN workstation is described.

These results show clearly that the available methods enable the implementation of truly high speed components. On the other hand the results mentioned above about the possibilities of incremental compilation techniques show that integrated high speed programming systems will be available in the next time.

Literature:

[1] W. Aßmann
The INDIA System for Incremental Dialog Programming
Proceedings of Workshop on Compiler Compilers and Incremental Compilation, Berlin, 1986
IIR-Informationen 2(1986)12,12-34

[2] C.C. Chang
An Ordered Minimal Perfect Hashing Scheme Based upon Euler's Theorem
Information Sciences 32(1984),165-172

[3] N. Cercone, M. Krause, J. Boates
Minimal and Almost Minimal Perfect Hash Function Search with Application to Natural Language Lexicon Design
Comp.& Math. with Appl. 9(1983)1,215-231

[4] G.V. Cormack, R.N.S. Horspool, M. Kaiserswerth
Practical Perfect Hashing
The Computer Journal 28(1985)1,54-58

[5] P. Fritzson
Incremental Symbol Processing
Proceedings of Workshop on Compiler Compiler and High Speed Compilation, Berlin, 1988 (to appear in this issue)

[6] R. Gerardy
Experimental Comparison of Some Parsing Methods
SIGPLAN Notices 22(1987)8,79-88

[7] J. Grosch
Generators for High-Speed Front-Ends
Proceedings of Workshop on Compiler Compiler and High Speed Compilation, Berlin, 1988 (to appear in this issue)

[8] V.P. Heuring
The Automatic Generation of Fast Lexical Analysers
Software-Practice and Experience 16(1986)9,801-808

[9] R.N. Horspool
Hashing as a Compaction Technique for LR Parser Tables
Software-Practice and Experience 17(1987)6,413-416

[10] T.J. Pennello
Very Fast LR Parsing
1986 ACM 0-89791-197-0/86/0600-0145 75c

[11] D.J. Rosenkrantz, H.B. Hunt
Efficient Algorithms for Automatic Construction and Compactification of Parsing Grammars
ACM Transactions on Programming Languages and Systems 9(1987)4,543-566

[12] T.J. Sager
A Polynomial Time Generator for Minimal Perfect Hash Functions
Communications of the ACM 28(1985)5,523-532

[13] T.J. Sager
A Technique for Creating Small Fast Compiler Frontends
SIGPLAN Notices 20(1985)10,87-94

[14] L. Schmitz
On the Correct Elimination of Chain Productions from LR Parsers
Intern. J. Computer Math. 15(1984),99-116

[15] W.M. Waite
The Cost of Lexical Analysis
Software-Practice and Experience 16(1986)5,473-488

[16] M. Whitney, R.N. Horspool
Extremely Rapid LR Parsing
Proceedings of Workshop on Compiler Compiler and High Speed Compilation, Berlin, 1988 (to appear in this issue)

[17] D.A. Wolverton
A Perfect Hash Function for Ada Reserved Words
Ada Letters IV(1984)1,40-44