Efficiency of programming in higher-level languages

H. MITTENDORF
Karlsruhe, W. Germany

1. Idea of efficiency in computer programming

The increase in software costs as compared with the cost of the hardware devices themselves has forced both vendors and users of computers to become more and more conscious of the problem of effective programming. Difficulties then arise because many people use the following definitions side by side:

a. Effective (i.e. as short as possible) program-code.
b. Effective (i.e. as little as possible) reaction-time of the program.
c. Effective labour supply, if the reserve of man-power is pruned.

Therefore, the 'effectiveness' problem can be treated exactly only if we define a relative reference value for all participants. This shall be the 'sum of all costs' of a computer installation (hardware + software). Effectiveness in this sense means the minimisation of total costs by suitable combination of the hardware configuration with a particular programming technique.

2. Historical development of programming technique

The first computers that appeared on the industrial market possessed so little core storage that programs could only be produced directly in machine code (MC). The middle fifties saw the invention of symbolic programming with the Assembler Code (e.g. in 1956 the language SOAP = Symbolic Optimal Assembly Programming at the IBM's 650). This new method of symbolic addressing was very convenient at the time. The significant sign was the 1:1 – conversion between the assembler and the machine code. For many years most people believed that this conversion was the only method of producing optimal programs in the sense of minimal code length.

The development of FORTRAN, enforced by IBM, was the beginning of a programming technique independent of the special features of a particular machine. At the same time, the concept of FORTRAN was made in such a way that an effective MC could be generated out of the source code. Some 'risky' features of high-level languages remained forbidden (e.g. very generalized I/O procedures, bit- and byte-oriented data-types, general data structures and language features for general loop-modifications). It is only now that such extensions are being discussed in conjunction with the new languages PL/1 and ALGOL 68.

There has been a general feeling that the reference value man-power would remain sufficient for a long time, i.e., it would be possible to define 'effectiveness' or 'efficiency' independently of the 'market' of system programmers. However, it is becoming very clear that it is easy to build computers but very difficult to educate and train persons capable of using these computers.

3. Programming technique today

Current developments or developments in the immediate future are indicated by the following typical signs:

a) Prices of core store
A clear tendency to lower costs can be seen in this sector. This is a result of new production techniques (wire store, later on MOS store, in the future holographic mass memories) which will bring much more saving. At present, we must assume prices of 0.15 to 0.20 DM per bit, but between 1975 and 1978 we may expect 0.05 to 0.10 DM per bit for machines with a cycle time under 1 μsec. By 1980 at the latest we may expect that the price for storage capacity will fall by a further factor of 10 or more. Perhaps by then we shall have the cheap Tera-Bit memory for each computer (with 10^{12} to 10^{14} bit and a read/write cycle time of 20 nsec).

b) Capacity of internal stores
As a consequence of falling prices, the store capacity will generally increase (also for process computers). A new method of addressing by the paging mechanism will be of great importance in this context. This method will enable the addressing of very large data blocks in spite of small data units (e.g. 16-bit words). In the immediate future we will also have for process computers a maxi-

mum store capacity of 1 million words or more. As a consequence of these facts, we can also produce core-store-resident programs by higher level languages. If a demand paging mechanism is available, one can renounce the overlay principle for large programs.

c) Programs on background stores
We also believe that external stores will become considerably cheaper than internal stores, and therefore we shall need a reasonable roll-out technique. Because of the greater capacity of these internal stores it is possible for greater parts of the software belonging to this technique to be taken over by the operating system. Therefore the application programmer will be unburdened. As a consequence, in the future it will also be possible to produce large programming systems by means of higher level languages.

d) Use of high-level languages
After the introduction of FORTRAN, a few further high-level languages (ALGOL, COBOL, PL/1) were developed. These languages can facilitate the work of the application programmer, especially in mathematical (ALGOL) and economic (COBOL) problems.

Many groups throughout the world are working on the development of special real-time languages (e.g. Process-FORTRAN, PEARL, LTPL). The newest and most modern of these languages provide for very comfortable bit-, byte- and string-handling, and for definition of special data structures. Therefore many of the tasks of the application programmers are considerably reduced. In special cases, the time for the development and testing of programs will be lowered by a factor ten.

On the one hand we find, in general, considerable facilitation of the programming technique, but, on the other hand, there are many difficulties which arise from the increasing use of computers for solving more and more diverse problems. A few of these problems are listed below:

a) 'Mathematization' of the tasks
Many interesting problems (e.g. weather-forecasting by computer, computer-aided design, linear- and dynamic-programming) now involve greater mathematical effort. The same problems arise from the urgency for mathematical treatment of many problems in commercial and management information systems. Effective effort is possible only if the highly specialized men engaged in solving these problems are economically employed and relieved of routine tasks. For this, the use of higher level languages is very important.

b) Larger volume of data
Usually this is a result of more complex problems. The higher level languages (especially ALGOL and PL/1) can bring about an important improvement in data management.

c) Higher data-structures
Problems connected with the increasing use of interactive displays working with real-time computer systems can only be overcome if we introduce new data-types – the so-called 'structures' and 'pointer-variables'. With these new elements we can solve the problem of referencing from one data-list to another connected with it.

PL/1 (using the concept of 'pointers') and ALGOL 68 (with the much more useful reference concept) give very serviceable means to the programmer. Programs for storage and treatment of complicated pictures (by suitable data-structures) can be prepared and tested much faster by these means.

d) Coordination problems in real-time-programming
An additional complication in real-time-programming is the (temporal) coordination of simultaneously running programs. For this we have the Dijkstra-Semaphores, which allow coordination of single programs. 'Higher coordinating functions', giving a real facilitation to the user, will be developed. In every case, the possibilities of coordination in a higher-level language give a formal simplification to the user. This can increase the clarity of the problem and consequently prevent the appearance of errors, especially deadlocks, in programming. All this will contribute to a more effective programming technique.

However, improvements due to revised programming techniques will be nearly counterbalanced by the increase in the complexity of new tasks to be programmed. Nevertheless, very considerable progress has been made, as compared with programming techniques at the beginning of computer development: we now have excellent higher programming languages. With these we are able to solve very extensive tasks such as computer-aided design, numerical weather-forecasting, complicated real-time process control, and so on.

4. Consequences

Maximum efficiency, in the sense of minimisation of total cost, can be reached for a given problem and given hardware installation only by careful considerations of, on the one hand, higher programming effort and, on the other hand, larger core store. Unfortunately, there are a number of basic obstacles which are not precisely measur-

able but could hinder the application of higher level languages.

a) User prejudice against higher level languages
Frequently, this follows from deficient knowledge, insufficient training and erroneous generalisation. In many cases, the very abstract and sophisticated specification (here one thinks of the ALGOL 68 report) will hinder the user, because he believes that this language is as difficult as the understanding of the language specification. The well-trained system analyser does know that the contrary is true, but the unskilled application programmer (and the number of these is much greater!) will not use such a language. Therefore in this field of application, much explanation, clarification and education must be done in the future.

b) Alleged uselessness of higher level languages for solving real-time-problems
Unfortunately, objections of this kind are often justified. We must, however, think not only of existing means (e.g. supervisor calls in FORTRAN), but we must also take into consideration that latest developments (e.g. PEARL and LTPL). When compilers are available, e.g. for PEARL, and when the usefulness of this high-level language is demonstrated for practical problems, the most conservative user will adapt himself to programming in higher level languages.

c) Hardware limitations
Core stores *will* become cheaper, but their size cannot be raised as much as one would like (because of addressing problems). This is especially true for very small computers (e.g. computers for measurement instrumentation or other special purposes); in these cases, the application of higher level languages cannot as yet be recommended.

A greater part of these objections will have lost its importance in the course of the next few years. We will therefore neglect these objections in the following quantitative considerations (especially the limitation on core stores). We can then say that we are able to perform easier and faster programming for a given problem by using a high-level language. We can thus introduce the following assumptions for a program of length L_A (made in Assembler language):

1. By using a higher level language we need only a fraction $1/n$ of the time needed for programming in assembler language ($0 < 1/n \leqslant 1$, n real).
2. By the use of this high-level language the object program is lengthened statically by a factor $V_s > 1$. In the case of very long programs, we can reach $V_s \leqslant 1$, but this is only a theoretical limitation, because it is almost impossible to make very long programs in assembler language at all. Therefore we neglect, for the moment, the dependence of V_s on program length L_A.
3. The written program shall run or be installed f times.
4. We assume fixed software costs K_{SA} in assembler programming for given program length (K_{SC} is the value for higher level languages).
5. In the same manner we assume fixed hardware costs. For this we apply the price of the additional core store in DM per word.
6. If we further consider the program dynamically running, we see that the running time can become longer by a factor V_t.
7. Finally, if the program runs in a computing centre, we additionally have to consider

M_{KSPW} = leasing cost for one word (ca. 0.0015 DM/hr)

M_{ZE} = leasing cost for central processor unit (ca. 250 DM/hr), and

t_A = absolute running time of the program (written in assembler code).

In a quantitative analysis we must compare the savings of manpower costs, that is

$$A_S = (K_{SA} - K_{SC}) \cdot L_A = K_{SA} \cdot L_A \cdot (1 - 1/n) \quad , \qquad (1)$$

with the increased hardware costs or leasing costs. Therefore, we have to consider two different cases:

A. Increase of hardware costs

If a vendor desires to sell a computer f times together with a special application program, he must find the most economical solution for the whole system. For this, the vendor could write the program in a compiler language (to save time) and pay himself for the (perhaps greater) additional hardware cost. This additional cost is for f applications of the same program:

$$A_H = f(V_s - 1) \cdot K_H \cdot L_A \qquad (2)$$

Here is a numerical example (for 24-bit word, 5000 FW/NY):

K_{SA} = 20 DM/FW K_H = 5 DM/FW
n = 3 V_s = 1.7 f = 3.

Then from Eq. (1)

$20 \cdot (1 - 1/3) = 13.34$

and from Eq. (2)

$3 \cdot (1.7 - 1) \cdot 5 = 10.5.$

Therefore, A_S is greater than A_H and the use of a compiler language (higher level language) is

more effective.

B. Increase of leasing costs

When programs in higher level languages are to be used in the closed shop of a computing centre, the costs will be higher both statically (V_s) and dynamically (V_t) (as compared with assembler programs). These higher level language programs, therefore, will need more core-storage (in units of leasing costs: $M_{KSPW} \cdot L_A \cdot (V_t \cdot V_s - 1)$) and more computing time (given by $M_{ZE} \cdot (V_t - 1)$). The sum of these two effects must be multiplied by the absolute running time t_A and by the number of applications, f. Therefore, the increase in leasing costs is then given by

$$A_M = f \cdot t_A [L_A \cdot M_{KSPW} (V_t V_s - 1) + M_{ZE}(V_t - 1)] \quad (3)$$

The value of M_{KSPW} is very low. If we assume K_H = 5 DM/FW and 20,000 total working hours (time of amortization, about 5 years), we get $M_{KSPW} = 2.5 \cdot 10^{-4}$ DM/hr.

Now consider two numerical examples:

(1) K_{SA} = 20 DM/FW (see above)

L_A = 2,000 FW

n = 3 (therefore $1 - 1/n = 0.67$).

Then, from Eq.(1),

$A_s = 2 \cdot 10^3 \cdot 20 \cdot 0.67$,

$= \underline{\underline{27{,}000 \text{ DM}}}$

f = 20

t_A = 12 min = 0.2 hr $\qquad M_{KSPW} = 2.5 \cdot 10^{-4}$ DM/hr

V_t = 1.2

V_s = 1.7 $\qquad M_{ZE} = 2.5 \cdot 10^2$ DM/hr

Then, from Eq.(3),

$$A_M = 20 \cdot 0.2 [2 \cdot 10^3 \cdot 2.5 \cdot 10^{-4} \cdot (1.2 \cdot 1.7 - 1) + 2.5 \cdot 10^2 \cdot 0.2]$$

$$= 4 [0.5(2.04 - 1) + 0.5 \cdot 10^2]$$

$$= 4(0.52 + 50)$$

$$\approx \underline{\underline{200 \text{ DM}}}$$

In this example we can neglect the additional leasing costs A_M (the first term is nearly zero).

(2) If we now increase L_A to 200,000 FW, $V_t = V_s$ to 2, t_A to 1 hr, and f to 50, we reach a very unfavourable case.

We now get

$$A_M = 50[2 \cdot 10^5 \cdot 2.5 \cdot 10^{-4} (4 - 1) + 250 \cdot 1]$$

$$= 50(50 \cdot 3 + 250)$$

$$= 50 \cdot 400$$

$$= \underline{\underline{20{,}000 \text{ DM}}}$$

Now the additional leasing costs are high, but in this case everybody will use a compiler-language (just because of the length of the program!). Beyond that, the savings of software costs are, numerically (because $L_A = 2 \cdot 10^5$ FW),

$$A_S = \underline{\underline{2\,700\,000 \text{ DM}}}$$

Summarizing: Only if t_A and/or f are large enough is it worth programming by assembler language. Quantitatively, for this purpose with t_A = 1 hr, the value of f should be 5,000 to 10,000.

It should be emphasized that, besides the economical reasons, there are many other points of view calling for the use of a compiler language, e.g.:

1. Much better form of documentation.
2. Saving of know-how when a new computer family with the same software package is used.
3. Simple changeability of the program.

For the user-programmer and the system analyst:

4. Faster programming and relieving of routine tasks.

For the producer and vendor of the data processing system:

5. Relief of own software deliveries specific for certain application problems for the customer, provided that excellent compilers for an effective high-level language are available.

For the immediate future, assembler programming will remain for only two major software fields:

On the one hand, for frequently used system programs, as the operating system itself and the various compilers, and, on the other hand, for frequently needed application programs which must have a very short code. One can find this in interrupt programs with very short response times or in the field of very small computers (e.g. computers for measurement instrumentation with a 'software modular system').

Final observations

Effectiveness, that is, minimum cost of the whole installation, especially the installation of a process computer, can only be reached by careful

consideration of all cost components. For this, we will need more and more higher level languages. But as assembler programming loses its influence, we must develop more efficient higher level languages for real-time programming. This is a very wide field for computer scientists in industry and the various research centres for computer science. Extraordinary efforts will be needed for a break-through in this field. In spite of all the difficulties, we should attempt to reach a world-wide standardization of such a language. But we must expect that the user-programmer will carefully examine these new language features and really use such language if possible. Only if both sides – the application programmer and the software designer of the language – have enough openmindedness and understanding of the problems can we expect a solution of general satisfaction.

Discussion

C. I am sure the author of this paper does not feel he has the final answer on efficiency measures, but I applaud attempts like this. They are better than none – you at least have something to make individual judgements from – and also something to change and refine as your understanding improves.