Analysis of program optimization possibilities and further development

I.V. Pottosin

Institute of Informatics Systems, Siberian Division of the USSR Acad. Sci., 630090 Novosibirsk, USSR

1. Introduction

Program optimization has appeared in the framework of program compilation and includes special techniques and methods used in compiler construction to obtain a rather efficient object code. These techniques and methods constituted in the past and constitute now an essential part of the so called optimizing compilers whose goal is to produce an object code in run time, saving such computer resources as CPU time and memory. For contemporary supercomputers, the requirement of the proper use of hardware peculiarities is added.

In our opinion, the existence of a sufficiently large number of optimizing compilers with real possibilities of producing “good” object code has evidently proved to be practically significant for program optimization. The methods and approaches that have accumulated in program optimization research seem to be no lesser valuable, since they may be successfully used in the general techniques of program construction, i.e. in program synthesis, program transformation and at different steps of program development.

At the same time, there has been criticism on program optimization and even doubts about its necessity. On the one hand, this viewpoint is based on blind faith in the new possibilities of computers which will be so powerful that any necessity to worry about program efficiency will disappear. This is not true: we see that new supercomputers require compilers to have new optimizing possibilities. But on the other hand, this viewpoint reflects those real restrictions and difficulties that are inherent in the current techniques of program optimization. Objective analysis of this current state is one of the goals of this paper.

The first paper devoted to program optimization as a separate area of systems programming was published in 1969 [3]. Since that time many publications have appeared. They proposed some algorithms as well as general approaches to program optimization. In several papers (for example, [4, 36]) a catalogue of optimizing
transformations was suggested. In [34] program optimization for supercomputers was considered. Good surveys of optimization and flow analysis techniques were presented in [24, 29]. The methodical consideration of program optimization as a part of program compilation is contained in the monographs [2, 28]. Unfortunately, there are few publications about the details of the realization of particular optimizing compilers: the monograph [7] should be mentioned among them. The monograph [25] gives a full presentation of contemporary program optimization techniques for conventional computers. In papers [12, 21, 37] the possibilities of program optimization are considered as a whole. We have analyzed the evolution and the possibilities of program optimization in [39, 43], and, in a sense, this paper is a continuation.

Let us make some general remarks arising from the publications above. Modern optimization techniques implemented in existing optimizing compilers are based on automatic execution of some actions which must improve the object code according to a predefined criteria. If these actions are significant for any hardware architecture, the optimization is called machine-independent. If they are intended for a particular architecture and machine language, the optimization is machine-dependent. These actions may be oriented not to a concrete architecture but to a class of computer architectures; in this case the optimization is called machine-oriented. As a rule, these actions, especially for machine-independent optimizations, are expressed by optimizing transformations on some internal language. The list of such transformations is fixed for a particular compiler, and context conditions allowing an action are defined for each transformation. These conditions combine both the predefined optimization criteria and the requirement to preserve the program invariants, i.e., the meaning of the program as a whole, after executing the transformation.

Context conditions take into consideration the dependencies between program fragments and objects. It is possible to distinguish several kinds of optimizations: local optimization when these dependencies are given only for one elementary statement or, in the best case, for a linear block; quasilocal optimization when they are considered for a program segment with a rather simple control structure, e.g., hammok, loop body, bodies of nested loops etc., and, finally, global optimization when these dependencies are evaluated for the program as a whole. These dependencies are evaluated during a program flow analysis that is either embedded into the corresponding transformation algorithms or may be realized separately at a special stage preceding the execution of optimizing transformations or actions. Since in the case of global optimization the expenditures for flow analysis may be significant, transformation factorization should be used. Factorization consists of reducing the global optimization to a quasilocal one which is applied to some hierarchy of program control structures representing the whole program.

2. Estimation of the optimizing effect

Unfortunately, there is very little research estimating the influence of program optimization on the quality of the object code. Here we shall use the results of the
papers [8, 13, 31, 44, 45]. More detailed measurements are made for compilers with a particular input language (from Fortran to Ada), but all optimizations being estimated are language-independent and we may consider these measurements to be relevant to any imperative programming language. We understand that the number, both of the experiments and the programs considered in these experiments, is not very large, nevertheless, the results seem to be sufficiently representative to come to reliable conclusions. Some measurements described in the papers [8, 13, 45] were analysed in [39] and this analysis was republished in the monograph [25].

We shall use the measurements made in the papers above to estimate an optimizing effect. The CPU time (or memory size) optimizing effect is the ratio of CPU time (memory) of the unoptimized program to that of the optimized program. We shall distinguish three qualitative levels of the optimizing effect. The effect is invisible if a program is improved by less than 1.2 with respect to CPU time and by less than 1.1 with respect to memory size. It is visible if the improvement is up 1.2 to 2 in CPU time and up to 1.3 in memory size. It is essential if the improvement is more than 2 in CPU time and more than 1.3 in memory.

2.1.

Vasyuchkova [45] presents the measurements for the ALPHA-6 compiler with Algol 60 extension as an input language. Forty-six short programs were measured which were mainly library procedures. One of the principal goals of this study was an estimation of the loop optimization effect. In summary, loop cleaning, strength reduction, register allocation for control variables and index expression in the loops are usually referred to as loop optimization. These are all quasilocal.

The optimization effects on CPU time and memory size based on the measurements of [45] are shown in Tables 1 and 2, respectively. They are distributed according to the level of effect and the average effects are shown.

<table>
<thead>
<tr>
<th>Number of programs</th>
<th>15</th>
<th>15</th>
<th>14</th>
<th>1</th>
<th>1</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effect</td>
<td>1.0–1.2</td>
<td>1.2–2.0</td>
<td>2.0–5.0</td>
<td>5.5</td>
<td>9</td>
<td>2.0</td>
</tr>
</tbody>
</table>

* To show that the essential effect in CPU time does not contradict with the effect in memory size, the effect on memory size is shown in parentheses.

<table>
<thead>
<tr>
<th>Number of programs</th>
<th>14</th>
<th>14</th>
<th>17</th>
<th>1</th>
<th>1</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effect</td>
<td>1.0–1.1</td>
<td>1.1–1.3</td>
<td>1.3–2.0</td>
<td>2.3</td>
<td>4.0</td>
<td>1.4</td>
</tr>
</tbody>
</table>

* The effect on CPU time for the last programs is shown in parentheses.
On the basis of these measurements it is seemingly possible to make the following conclusions:

(1) Although the optimizing effect is visible on average, it is very different for different programs: approximately a third of the programs are practically unoptimized, a third have a visible effect, some more than a third of programs have an essential effect and only in a few cases is this effect large.

(2) Optimization reliability is always reached; for all the programs the effect is not less than one.

(3) Since these measurements are related to (a) small programs and (b) library (i.e., carefully written) programs, the effect should be apparently increased for medium and large programs being typical programmers’ production.

(4) There is no answer to the question of optimization cost; what expenditure is necessary to attain such effects (note that these expenditures depend on effects in a small degree, because the expenditures exist for the programs with invisible effects).

2.2.

In [8], the measurements were applied to the optimizing FOREX compiler with Fortran 77 as the source language. Some widespread constructions found by Knuth [30] in the real programs were investigated. Run time with and without loop optimizations (LO) and common subexpression elimination (CSE) was measured both in common and separately. Although the FOREX’s authors modestly considered the optimization to be local, in fact it was quasilocal. Evaluated CPU time effects are shown in the same style as previously in Tables 3–5. It should be noted

<table>
<thead>
<tr>
<th>Table 3</th>
<th>CPU time effect for CSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of programs</td>
<td>10</td>
</tr>
<tr>
<td>Effect</td>
<td>1.0–1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4</th>
<th>CPU time effect for LO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of programs</td>
<td>5</td>
</tr>
<tr>
<td>Effect</td>
<td>1.0–1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5</th>
<th>Common CPU time effect (CSE + LO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of programs</td>
<td>7</td>
</tr>
<tr>
<td>Effect</td>
<td>1.0–1.2</td>
</tr>
</tbody>
</table>
that in [8] the CPU time of the optimized (automatically) programs was compared with the CPU time of the programs written by hand and optimization was essentially exhaustive. In Table 6, we show the "underoptimization" effect which is the ratio of the CPU time of the optimized program to that of the program written by hand.

<table>
<thead>
<tr>
<th>Number of programs</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0-1.2</td>
</tr>
</tbody>
</table>

For three last programs with visible "underoptimization" effect we show the optimizing effect obtained in parentheses. The effect is highly visible but should be more. The authors [8] answered why this was so. In the first example the complex Boolean expression was not optimized, in the second one the fact that the value of the loop control variable was unused outside the loop was not taken into consideration; in the third one such properties of the arithmetic operator as commutativity and associability were not considered. We may note that some optimizing compilers take the last two possibilities into account, so this "underoptimization" effect would seem to be removed for such compilers, but the context conditions for the values of the loop control variable should be globally defined.

On the base of these measurements we may conclude that:

1. The difference in the optimizing effects is confirmed: for loop optimizations this difference is the same as above, for common subexpression elimination the number of practically unoptimized programs increases to two thirds of all the programs but when CSE and LO are simultaneously implemented, the unoptimized programs make up only 13%: this is visible, but a small percentage.

2. The optimization is reliable as above.

3. The common optimizing effect increases: about two thirds of all the programs have an essential effect because the measured programs are closer to the average programmer's production and the list of program optimizations is wider than above.

4. Comparison with programs written by hand shows that the optimization is exhaustive for a rather large percentage of programs (two thirds approximately); it is important because these optimizations are principal in optimizing compilers.

5. There exist programs for which an optimizing effect may be increased by introducing new transformations corresponding to deeper context conditions.

6. As above, the measurements do not estimate optimization costs, but the authors state that time expenses on optimization are rather small in FOREX.

2.3.

The results of measurements which are rich in the set of measured properties (but not rich in the number of measured programs) are presented in [13]. These results help us increase our knowledge about optimization possibilities. Only four programs
were measured but the set of optimizing transformations was much more than above. Implemented transformations were grouped into 3 optimization levels. Level 0 consists of dead code and common expression elimination in the linear blocks, constant propagation and other local and quasilocal optimizing transformations. Level 1 includes the level 0 transformations and such transformations as global common expression elimination, loop cleaning (code motions for loop bodies), strength reduction, etc. This level may be characterized as global optimization. Level 2 is a double execution of quasilocal and global optimization: it includes execution of level 1 and repeated execution of level 0 added by global elimination of common subexpressions and loop cleaning. This level must answer the question how many possibilities for implementing the global and quasilocal optimization exist when it has been implemented once. Compiling (optimizing) time is estimated for each level. The source language is a PL/1 subset.

On the basis of the measurements [13] we shall evaluate the optimizing effects in time and memory for each level with respect to the preceding one (for level 0 these effects are the same optimizing effects as above), the total effects for level 2, compile-time degradation coefficient as the ratio of compile-time for some level to compile-time for the preceding one (level 0 we shall estimate with respect to the lack of optimization) and total coefficient for level 2 with respect to unoptimized compilation. The results are shown in Table 7 for four measured programs $P_1$, $P_2$, $P_3$ and $P_4$.

| Table 7 |
|------------------|-----|-----|-----|-----|
|                 | Level 0 | Level 1 | Level 2 | Total |
| $P_1$ (a) Compile-time coefficient | 1.0 | 1.6 | 1.1 | 1.7 |
| (b) Effect in memory | 2.3 | 0.85 | 1.05 | 2.0 |
| (c) Effect in time | 3.1 | 1.7 | 1.05 | 5.6 |
| $P_2$ (a) | 0.9 | 1.6 | 1.1 | 1.65 |
| (b) | 1.8 | 0.95 | 1.0 | 1.7 |
| (c) | 1.8 | 1.1 | 1.0 | 2.0 |
| $P_3$ (a) | 1.05 | 1.5 | 1.1 | 1.8 |
| (b) | 1.45 | 1.05 | 1.0 | 1.5 |
| (c) | 1.4 | 1.2 | 1.0 | 1.6 |
| $P_4$ (a) | 0.85 | 1.2 | 1.1 | 1.1 |
| (b) | 2.4 | 1.1 | 1.05 | 2.8 |
| (c) | 2.5 | 1.1 | 1.05 | 2.8 |

In [13], there is no comparison with programs written by hand, but it is noted that the body (and run time, respectively) in the inner loop of $P_1$ may be decreased 1.5 times by taking into account commutativity and associativity of the operations.
It is possible to say that the source program size is medium and correlation of their sizes \( V_1 : V_2 : V_3 : V_4 \) is 5:2:4:1 if \( V_4 \) is taken to be equal to 1.

On the basis of these measurements it is seemingly possible to make the following conclusions (with restrictions connected with a small number of measured programs):

1. For all the programs the optimizing effects are at least visible (effects in memory are essential in all cases). This should be expected because the set of optimizations is much greater than above.
2. Quasilocal optimizations may be considered as exhaustive in three out of four cases because the effect of level 0 does not really differ from the effect of level 1.
3. Compile-time is not increased by quasilocal optimizations and, moreover, the compile-time may be decreased by these optimizations, because reducing the size of the intermediate text as a result of quasilocal optimization decreases the object code generation time and, consequently, the total compilation time; it is useful to take this fact into consideration in developing programming systems.
4. There exist programs for which global optimization is desirable (maybe this is relevant for sufficiently large programs as in the example this is the largest program).
5. During global optimization, the compile-time is significantly increased, but its repeated execution has no visible influence on this parameter, because the flow analysis may be carried out only once if the flow analysis results may be corrected after a transformation.
6. Global optimization criteria are such that these optimizations are not completely reliable: in two cases (\( P_1 \) and \( P_2 \)) an increase in run-time effect is accompanied by a decrease in memory effect.
7. One iteration of global optimization is apparently exhaustive, but it is difficult to give a decisive answer due to the relatively small number of measurements and limited size of the programs in [13].

2.4.

An optimizing Ada-compiler has been analyzed in [44]. Unlike the research quoted above, the measurements were made in a special benchmark consisting of well-known comparatively small programs with computational (as above) and combinatorial character. The optimization set is rather rich and has no machine-dependent optimizations; some optimization are global.

There are many interesting remarks and measurements in this paper: the individual contribution of each optimization, execution order, influence of interprocedural flow analysis, merits of internal tree representation for the program optimization, etc. We shall not discuss these problems in depth, but present in Table 8 only the results concerning optimizing the run-time effect for each program (it seems that the average effect is not interesting because the programs belong to very different application domains).
We may remark that the first four programs are essentially recursive. The memory-size effect for separate programs was not measured; the total effect for this benchmark was 1.5, i.e. essential.

From these results and other measurements in [44] it is possible to conclude:

1. The optimizing effect changes from visible for combinatorial problems to essential for computational and similar problems.

2. The stability of unessential but visible effects for combinatorial problems apparently shows that the classical optimization set is steady for computational problems (in the preceding paragraphs all the problems were computational), but has no good conformity with the combinatorial programs.

3. A small optimizing effect for recursive problems seems to be expected.

4. There is no distinct separation between quasilocal and global optimizations, but they essentially contribute to the optimizing effect in contrast to machine-independent (but machine-oriented) local optimizations.

5. All optimizations are reliable (the effect is not less than 1).

2.5.

Now we shall turn to the results in [31]. The authors proposed a method to estimate the optimizing properties of compilers. Taking the classic set of optimizing transformations: constant propagation, local and global common subexpression elimination, code motion, machine-independent loop optimization, dead code elimination etc., they estimated the weighing observed for each construction in an empirical study of programming style. The number of iterations of each of these constructions in the proposed benchmark is equal to this weight. The benchmark consisted of two equivalent parts: optimizable and non-optimizable. The first part was subject to optimizing transformations by the compiler, and the second part all transformations were made "by hand" in the source text.

The effect of "underoptimization" was measured as the ratio of the execution time of the object program produced by the optimizable part after optimizing

---

### Table 8

<table>
<thead>
<tr>
<th>Short characteristics of a program</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Permutation</td>
<td>1.4</td>
</tr>
<tr>
<td>2 Binary tree processing</td>
<td>1.4</td>
</tr>
<tr>
<td>3 Eight queens</td>
<td>1.4</td>
</tr>
<tr>
<td>4 Hanoi towers</td>
<td>1.6</td>
</tr>
<tr>
<td>5 Fast Fourier transformation</td>
<td>1.7</td>
</tr>
<tr>
<td>6 Quick sorting</td>
<td>1.8</td>
</tr>
<tr>
<td>7 Filling a cube with blocks</td>
<td>2.0</td>
</tr>
<tr>
<td>8 Real matrix product</td>
<td>2.1</td>
</tr>
<tr>
<td>9 Bubble sorting</td>
<td>2.4</td>
</tr>
<tr>
<td>10 Integer matrix product</td>
<td>2.6</td>
</tr>
</tbody>
</table>
transformations to that of the program produced by the non-optimizable part. This effect is average for a compiler, because the optimization weight corresponds to the statistical frequency of the construction in the real program flow.

The "underoptimization" effect was measured for several well-known compilers and languages: Fortran (6 compilers), C (7 compilers), Pascal (4 compilers) and Basic (1 compiler). The following results were obtained: four Fortran compilers had practically invisible effects (they ranged from 1.1 to 1.3), for two Fortran compilers the effect was visible (from 1.8 to 2.0). One C compiler had the effect equal to 1.35, three C compilers had effects from 1.45 to 1.6, two C compilers had effects close to 2 (1.9 and 1.95), and one C compiler had effect equal to 3.1. Thus, the underoptimization effect for the C compilers was at least visible in almost all the cases and was sufficient in one case. Two Pascal compilers had visible but small effects (1.4 and 1.5) and two others had sufficient effects (2.0 and 2.35). The only Basic compiler had visible effect equal to 1.8.

According to the results of the measurements we may come to the following conclusions:

(1) Judging by the wide range of the measured effects and assuming, in the worst case, the maximal "underoptimization" effect values to be obtained by compilers without optimization facilities, we may say that implementation of the traditional set of optimizing transformations decreases the execution time 2-3 times.

(2) As was expected, the richer the language, the greater the optimization effect: the difference between the object program execution time for optimizing and non-optimizing Fortran and Basic compilers is less than that for Pascal and C. Although optimization is not always needed, and the expenses on its execution are not always paid, the fact that a number of properly optimizing Fortran compilers (i.e. compilers with small "unredoptimization" effect) is more than that for the C and Pascal languages, while the effect of C and Pascal program optimization is higher than that of Fortran programs, allows us to conclude that the number of optimizing compilers is smaller than is necessary.

3. Problems and difficulties of program optimization

The measurements and estimations considered above are interesting in themselves and speak in favour of program optimization, but we shall use them further, as well as other aspects of the problem of optimizing compiler application to reveal existing problems and difficulties of program optimization.

3.1.

In the previous examples the optimizing effect was always more than 1, i.e., no programs were pessimized (Abrahams [1] warned about this danger as early as 1970). It is possible because the context conditions of each optimization guarantee
that the program does not worsen at all steps of its execution. This trend to reach optimization reliability at each step has a disadvantage; as a rule, optimizing compilers try to avoid transformations that may worsen the program for a time but provide its significant improvement afterwards. The situation is aggravated by the fact that global optimization criteria may turn out to be conflicting: increasing efficiency in execution time, the optimizing transformation may also increase the memory size demands, as was shown in Section 2.3. In automatic execution of optimizations it is not always possible to estimate the potential influence of optimizing transformations on further application of other transformations. Such estimation turns out to be especially difficult for rather complicated computer architecture (see, for example [47]).

3.2.

The program transformations in optimizing compilers are executed automatically; the same optimization set is applied to any program in the same sequence (with the exception that one or several optimizing transformations may be forbidden by a user). The set of optimizations to be realized and context conditions for optimizing compilers are chosen with orientation to the mass implementation, i.e., optimizations should be implemented in a large number of programs and context conditions should be constructively proved at an acceptable cost for all programs automatically. Thus, Boolean expression optimization is absent in the optimization set (see Section 2.2) because there are usually no complex Boolean expressions in computational programs. And similarly, these programs have no recursive procedures, so special recursive procedure transformations are not needed in the typical optimization set (see Section 2.4). Such an approach, on the one hand, sets the user free from the necessity to know something about optimization, but, on the other hand, does not allow the user to change this set or to define deeper context conditions. Each programmer who participated in optimizing compiler construction and had contact with users, knows that users exist who do not care for the fact that the programs are essentially optimized on average, but are very indignant at the fact that a compiler did not see "obvious" optimizing possibilities in their "native" programs. This is especially noticeable for supercomputers, where the cost of such omission becomes very high.

3.3.

In quite rare cases commercial optimizing compilers may produce optimized programs not equivalent to the source ones, which may result in discouraging situations for users (for example, abend in correct programs). It more often occurs because programmers do not provide correct ground for the context conditions of all optimizing transformations and make decisions based on intuition rather than on theoretical investigations. However in the intuitive approach the context conditions may turn out to be incorrect or too rough (the latter decreases the possibility
of transformations in many cases). It is worth noting, that intuition does not always help to find the best order of transformation application, if the set of optimizations is very large.

3.4.

The cost of program optimization was discussed above with respect to the CPU time, while the other aspect of the problem, i.e., the labour-consuming character of optimizing compiler development is no less significant. Wulf noted as early as 1980 [48] that the number of optimizing compilers is insufficient because it is much more difficult to construct an optimizing compiler than a simple non-optimizing one (deficiency of optimizing compilers has been also proved by the results of measurements given in Section 2.5). There are several reasons for such deficiency. Insertion of special techniques and algorithms oriented to flow analysis and optimization into context analysis and code generation makes the implementation of these phases much more difficult and, what is especially important, destroys the conceptual comprehensibility of the algorithms. Taking this effect into account, in several experimental optimizing compilers, optimization is implemented as a separate phase (see, for example [5] and [20]), which allows this phase as well as all other phases to be made clearer. This approach is still rarely used in commercial optimizing compilers.

While implementation of the majority of program optimizations does not depend on a source language and, for machine-independent implementations, it is not related to the object computer, it is practically impossible to transfer the optimization algorithms from one compiler to another, even when they have the same input language. This is connected to unjustified differences in optimizing compilation schemes and quite insignificant distinctions of internal program representation. The problems of unification and standardization of internal representation are far from being solved. Automatic compiler construction methods are rarely used in the practice of optimizing compiler development. There is a relatively small amount of research works devoted to validation of the choice of context conditions and of the order of transformation application.

3.5.

The users often “reproach” optimizing compilers with unexpected changes in the structure of the program, so conventional debugging facilities that usually report the history of program execution produce results that are hardly possible to correlate with the source program. It should be stressed that just various transpositions and removals of the statements being, as a rule, machine-independent, produce the effect mentioned above and remain unknown to the author of the program. Although it is, in principle, possible to inform the programmer about such changes, no optimizing compilers do it.
4. Program optimization prospects

During the last ten years the direction of program optimization research was rather truthfully defined. On the basis of the achievements in program optimization gained in the 1960s and 1970s and taking into account the real problems and difficulties that appeared both in research and applications, we may consider these directions to be dependent upon the usage of the approaches and methods that proved to be efficient in other areas of system programming and to be dependent on overcoming the difficulties in conventional areas of program optimization applications, i.e., compiling systems for traditional and new computer architectures.

4.1.

Optimization of programs for traditional high-level languages and von Neuman computers has been sufficiently investigated. In the Introduction we have listed the works in which (as well as in many others) the established standard sets of optimizations and practical execution algorithms are presented. The unsolved problems in this research area are mainly technological, they are related to either the technology of optimizing processor development or the technology of processor application.

As was mentioned above, one of the main difficulties in program optimization lies in the labour-consuming character of optimizing compiler design. There are several ways to solve this problem connected with different technological approaches. One of them consists of defining a clear standard scheme to implement an optimization for a particular language or class of languages. Two kinds of optimizing compilers should be distinguished for this purpose.

The first kind is represented by a compiler with quasilocal and local optimization. As the measurements of Section 2 show, such a compiler providing loop and indexed expression optimization, economy of expressions, local constant evaluation, sufficiently developed means for machine-dependent optimization and several optimizations depending on input language, reaches the limit of the possibility of optimizing a large number of programs and may be automatically used in mass applications. The machine-independent optimization in such compilers may be implemented in one or two separate scans of the program obtained after context analysis, if some additional attributes corresponding to flow analysis have been evaluated in the context analysis. Machine-dependent optimization may be done during code generation by means of the pipe-line technique. A comparatively small set of only those optimizations whose mass application is justified allows the optimizing pass implementation to be rather simple and efficient. The algorithms of this pass to a larger extent depend on the representation of the program attribute tree rather than on the input language.

The second kind is represented by a compiler oriented to execution of a large number of global optimizations. Optimizing part of such a compiler is rather difficult
and works very slowly, especially in flow analysis. It seems to be natural to separate
the optimizing part of this compiler from the proper compiler and implement it as
a relatively isolated optimizing processor. This processor may be language-independ-
dent and embedded into a multilanguage compiling system [50] or may be imple-
mented as a programming environment tool [14]. But in both cases it is very important
to define sufficiently good internal representation of the programs appropriate for
flow analysis and optimization algorithms and for the other program processing
tools as well. Taking into account that optimizing processor application is a rather
time-consuming process, the set of optimizing transformations is rich and the
optimization should be adapted to a particular program. In such processors the user
should have more influence on the process of his program optimization than in the
first kind of compilers.

A technology of optimizing compiler and processor construction needs the means
for its automatization. Two ways of automatic construction of optimizing algorithms
seem to appear. The first consists of using the so called attribute approach, when
special optimizational attributes are evaluated and then an attributed tree of the
program is transformed according to the rules of attributed transformational gram-
mars [46]. This approach seems to be well suited to constructing the first kind of
optimizing compilers. The second approach has been suggested in [20] and consists
of developing the library of technical modules used in various optimizing algorithms.
It implies availability of a large set of optimizations (the library-based approach
works just in such cases) and may be applied to optimizing processors. An interest
in automatic implementation of machine-dependent optimization has been recently
aroused and a number of approaches to such implementation, i.e. to automatic or
automatizable construction of optimizing code generators (see, for example [10]
and [22]).

Raising the level of program reusability seems to be useful in optimizing compiler
construction in no lesser degree than in the general technology of program construc-
tion. But the reuse is possible only after unification of data representation (in our
case optimized programs play the role of such data). Here the above-mentioned
problem of internal representation of the program is raised again. Examples of such
unifications are known: the DIANA language [16] that was suggested as a common
basis for all ADA processors, and the Internal Language of the BETA system [50]
which is one and the same for a wide class of input languages.

During optimization the modern technology of programming system construction
strictly connected with modularization should be taken into account. Optimization
should also be done module by module, in the process of module occurrence and
with regard to the connections between the modules. The problem of quasi-indepen-
dent module optimization is discussed in [40]. For its solution, developed inter-
module (interprocedure) flow analysis (see, for example, [14]) as well as thought-out
internal representation of the modules and convenient structure of the program
development data base/module base [42] is required.
4.2.

Supercomputer program optimization is a relatively new and important research direction. New optimizing transformations to take into account hardware-implemented possibilities of vector operations and parallel computations occur here. The transformations bringing sequential programs into vector or parallel form are mostly investigated (a rather complete review of such transformations may be found in [34]).

It is worth noting that the relations (dependencies) being defined by data flow analysis in compilers for conventional computers are insufficient for context conditions of such transformations. Data dependencies that correspond to special vector operations [49] or bind data being calculated inside the loops [6] become essential. Efficient application of such transformations requires interprocedural flow analysis [11]; the necessity for this has been mentioned above.

So, there exists a large number of results concerning transformation of sequential (mainly, Fortran) programs into efficient supercomputer programs and a number of vectorizers and optimizing compilers that use these results. There are interesting measurements [33] which show vectorizing possibilities for existing Fortran-compilers for supercomputers to be rather rich and which estimate the importance of different optimizations for these architectures. But some problems to be investigated in this direction still remain. There are no exact measurements to show how exhaustive the transformations being applied are, how close the program being received is to the most efficient program for a given supercomputer. Optimizing transformations for supercomputers have a number of special features making them different from conventional transformations for sequential computers. Thus, they depend on computer architecture to a greater extent. They include a large number of contradicting transformations: for example, loop merging and loop separation may seem to be equally useful. This requires defining a sufficiency of subtle context conditions of transformations which are restructured for any particular program and separating, in a reasonable way, machine dependency and machine independency.

When transforming sequential programs it is necessary to find program dependencies and to restructure the program on their basis. It seems however, that under non-procedural definitions, i.e., specification of computational problems, these dependencies are more obvious than in a sequential program constructed according to the specifications, and program synthesis by specification together with special transformations provide a more optimal supercomputer program than restructuring of a sequential program, where some relations visible from the specification cannot be automatically found. Further investigations in this direction will seemingly make a major contribution to supercomputer program optimization.

Studies in optimizing transformations of truly parallel programs are far from complete. Optimization of programs written in parallel and vector languages is a promising research area, especially as there exists the trend towards incorporating operations over composite data into sequential languages, from Alpha, Algol 68, PL/1 to Ada, Fortran 8X and other modern languages.
4.3.

A natural evolution of traditional program optimization has resulted in the appearance of the transformational approach to program construction. The essence of this approach is a construction of a program from specifications or from another program by applying a rich set of transformations. The important feature is an active communication with a user: the transformational approach implies automatized rather than automatic program construction.

Recently developed specification languages allowing program execution have become a convenient, easy-to-use tool for rapid prototyping and testing, together with the user, of fundamental correctness of the main requirements and decisions. But efficiency of the prototypes obtained in such a way is very far from allowing their usage on a mass scale. To be widely used, the prototype should be transformed into an efficient program. This program may be constructed by means of transformations, but the choice of the transformations together with the order of their application depends on a specification being transformed and requires significant intellectual effort. In this case, using transformation definition facilities and transformation execution tools, a programmer himself controls the transformation of the source program. This research area has been called transformational synthesis. An overview of related problems, transformations being applied and existing systems is presented in [32].

Constructing specific programs for particular cases from a universal program, so called program specialization, is another area of application of the transformational approach. During program specialization, a natural goal is set: to make the program as efficient as possible by reducing the general problem to the given particular case. Such specialization may be done by means of mixed computation [18] or concretization [26].

Both lines of the transformational approach require a technique for defining transformations together with their context conditions to be developed. Such transformations may be defined by a special language (see, for example, the CIP project [9]) or by means of a transformational machine [19, 21] that has a set of basic transformations as its operations. In the last case the process of transformation represents a program for a transformational machine.

The research on optimization in languages with automatic program synthesis or logical inference (Utopist, Prolog, Nut) are closely related to the above-mentioned program optimization development. At the same time, the special transformations that have not been studied in conventional optimization are necessary for the synthesis of efficient programs or logical inference optimization. Even the problems existing in traditional program optimization become specific when related to these languages [15, 23].

4.4.

Existing techniques and methods of program optimization include, on the one hand, acquisition of reliable information about a program behaviour for the complete
set of input data and, on the other hand, directed program transformations retaining
the meaning of the source program and allowing a program with prescribed properties
to be obtained. These possibilities of program optimization permit using the
above techniques and methods as the basis for designing program construction tools
[37] and as the basis for program processing tools at practically all stages of the
program development cycle.

As has been shown as early as 1980 [27], optimization techniques and methods
provide possibilities for constructing processors that are useful at different stages
of program construction. Optimization techniques and methods may be used:

1. In program design and specification. There may exist processors which test
project interfaces and specification consistency or allow prototypes to be constructed
as executed specifications, in an automatized way, by means of specification transfor-
mations.

2. In automatic program construction. There may exist processors which automatic-
cally optimize a program or module, or transform them in the necessary direction
(under user control), and which test some aspects of program and programming
system correctness verifying the interfaces, pointing out unlikeness and generating
program assertions to be checked by the user.

3. In debugging. There may exist processors which define the information logical
structure of the program as the basis for automatic generation of a test set or which
take this structure into account in order to put the statements gathering the execution
history into the proper places.

4. In documentation. There may exist processors which visualize information-
logical structure of the programs, estimate its characteristics, generate assertions as
additional comments and reduce the program to a more obvious form.

5. In maintenance and modification. There may exist processors which generate
specialized versions with respect to usage experience or give messages about changes
in information-logical structure corresponding to changes that have been made
during maintenance or supposed to be made, which allow changes to be additionally
controlled.

The instrumental support of program construction is especially important in
developing software for a programmer's working site [41]. It is the programmer's
working site where instrumental support of the programmer's activities in program
construction is becoming both necessary and implementable in the most natural way.

4.5.

The theory of program optimization has appeared to study optimizing program
transformations and implementation algorithms. First works on this theory were
based on the theory of program schemata [17]. Practical applications of these
theoretical studies have two main directions: validation of context conditions of
some transformation and proving that some algorithm applying a transformation
or a set of transformations provides, in a way, an optimal program (i.e., this algorithm
yields the program to which this transformation is no longer applicable, and better
program cannot be obtained by applying the transformation in any other order).
Optimization theory is also valuable from the methodological point of view: it provides the basis for constructing the model of a language-independent description of the transformation as well as the principles of constructing the internal representation of a program suitable for the algorithms of flow analysis and transformation algorithms.

Some transformations were based on such theoretical models as Yanov schemes and Lavrov schemes [17]. A wider set of machine-independent transformations (see, for example [2]) was based on standard program schemes. But, as is stated in [38], standard schemes are a very simple model and may be used in defining optimizing transformations of simplified programs only, since they cannot describe a number of real program properties. There appeared program models allowing optimization-directed properties of real programs to be defined: linear schemes, large-scale schemes, linear programs (all of them are oriented to machine-independent optimization). Special program models have been suggested for several machine-oriented optimizations. An overview of the program models for optimization is presented in [38]. By means of these models a correctness of refined context conditions for such important optimizations as loop cleaning, cleaning of recursive procedure bodies, redundant computation elimination, etc. has been proved, and the optimal character of some algorithms for conventional optimizing transformations has been shown. Since it is necessary to extend a set of executed transformations for an advanced optimizing processor, the research in this direction should be continued.

The above-mentioned studies relate to sequential computers. Similar research for supercomputers has just started. There are, of course, studies (e.g., [35]) where some transformations for supercomputers are investigated and validated, but their number is far from being sufficient. It seems that there is no general-purpose theoretical model oriented to supercomputer program optimization, but some elements of this model are beginning to appear (e.g., regular sections [11]). In my opinion, existing theoretical models of parallel programs as well as standard schemes for sequential programs are not adequate tools for investigating parallel program optimization. Thus, the theory of supercomputers and purely parallel program optimization should be further investigated.

There is an obvious necessity for theoretical investigations to introduce the transformational approach and mixed computation into programming practice. In the framework of such a methodological concept as the transformational machine, a set of basic operations should be validated, transformations of basic operations should be correctly constructed and so on. To be efficiently used, mixed computation requires the depth and correctness of the usage to be validated.

6. Conclusion

In conclusion, let us note that the development of program optimization and transformations is undoubtedly promising for the future of programming. The following arguments speak in favour of this assertion:
(1) The advent of new computer architectures; constructing efficient software for these computers seemingly requires the methods of automatic program generation to be developed with maximal regard for specific features of new architecture.

(2) Widespread use of built-in computers requiring highly efficient programs in whose construction all programming experience should be used, including the achievements of program optimization.

(3) Development of new methods of software construction, including those related to the transformational approach, program synthesis and so on.

(4) Possibilities of applying program optimization techniques and methods to raise the level of instrumental support of software development, to construct software tools for the programmer's working site.

References


Program optimization


