MEASUREMENT OF PROGRAM IMPROVEMENT ALGORITHMS

John COCKE and Peter W. MARKSTEIN
International Business Machines, Inc., Thomas J. Watson Research Center
Yorktown Heights, New York USA

Abstract: This paper reports on the effectiveness of various code optimization techniques. These techniques are generally found in the literature, but a short informal description will be given of each. These include common subexpression elimination, code motion, dead code elimination, strength reduction, reassociation and linear test replacement, constant propagation, subsumption, and register allocation which is based on graph coloring.

All of these techniques have been embodied in a compiler which is a variant of PL/1, whose purpose is to produce good system code. A discussion will be given of those changes to PL/1 which make the production of good code easier, such as removal of on-conditions, use of offsets instead of pointers, and minor restrictions on declarations of bit strings.

The techniques are in general machine and language independent. At the present time we generate code for six different machines, most of which are experimental. The performance data will be based on System/370 because of availability of machine time. In particular, the register allocation techniques and its machine independent aspects will be discussed.

1. INTRODUCTION

1.1 Objectives

Algorithms for code improvement during the compilation process have appeared in the literature for over ten years. This paper presents measurements of the effects of applying some of these algorithms, including the additional times to apply the algorithms, and the changes in running time and code space of the compiled programs.

The code improvements with which this paper is concerned are designed to overcome limitations under which code is usually generated. Generally, code is produced when some grammatical construct is recognized; the LALR-produced parser-generators is an example of this approach. However, insufficient context is usually available for such a code generator to produce efficient code. Nonetheless code improvement algorithms when applied to code so generated can result in final code approaching and even surpassing hand coded assembly language programs. These improvements are achieved by exploiting global relationships which were unavailable during the parsing phase. It should be stressed that the code improvement (or optimization) algorithms are not designed to overcome poor coding practice or inefficient algorithms.

1.2. The Source Language

The compiler used for measuring the effects of code improvement is for the PL/1L (PL/1-like) language, which is an experimental system programming language at IBM Research. PL/1L is a PL/1 derivative designed to allow compilation into efficient object code, and which requires that validity checking be carried out at run time when it cannot be ascertained during compilation time.

Certain PL/1 features were omitted from PL/1L in order to remove obstacles to good global code improvement. ON conditions were removed, because the complex program flow that these conditions imply tends to inhibit good common subexpression elimination and register allocation. However, all programs are compiled as though SUBSCRIPTRANGE was enabled, but so that detected violations lead to program termination. Of course, the code which checks for subscript range violations is subjected to all the code improvement transformations.

Unrestricted pointers are also not permitted in PL/1L, since they impose unacceptably heavy penalties when full validity checking is required. Offsets into areas are available.

Bit strings of arbitrary or variable size are also not permitted in PL/1L because their implementation is too inefficient. By restricting declarations of bit strings to specify a fixed, constant length no longer than the machine word size, the commonly encountered uses of bit strings in system programming yield code of assembly programming quality. Longer bit strings can be obtained by using vectors of bit strings.

On the other hand, PL/1L has improvements in parameter passing dictions. By declaring parameters as being passed by value, or declaring that a subprogram does not use internal static storage, more efficient calling sequences are possible. It becomes feasible to write one or two line subprograms, if such programs require no prologue or epilogue.

The OFFSET attribute has been made orthogonal to other attributes, thus allowing efficient address computations to be encoded in PL/1L. Similarly, logical operations on integers, or arithmetic on bit strings can be specified without implying conversions. The arithmetic precision rules, while restricted, have also been simplified so that the system programmer can easily predict the effects of his arithmetic computations.

The viability of this language has been demonstrated by developing the PL/1L compiler in the intersection of the PL/1 and PL/1L languages. Initially, the compiler was itself compiled with PL/1; after development had proceeded sufficiently, the PL/1L compiler was bootstrapped, and further development has exploited unique PL/1L features to the point where the compiler can no longer be compiled by PL/1. Several operating systems for experimental machines have also been produced in PL/1L.

1.3 The Compiler

The PL/1L compiler uses a LALR-generated parser-generator to produce intermediate language (IL) text for an idealized machine with an unlimited number of high speed registers. In other respects, the IL is a relatively low level interface which exposes most of the details of a computation. There are some exceptions; for example MIN and MAX are IL primitives to avoid exposing additional program flow. After the code improvement algorithms have been applied,

final code selection transforms IL text into object code for the target machine.

Each code improvement algorithm is implemented as a stand-alone program which transforms the intermediate language program into another program in the same intermediate language. In this way, it is easy to experiment with the order of applying optimizing transformations, and the effects of the transformations can be measured singly, and in concert with others.

The code improvement algorithms are based on known algorithms which are mathematical in nature. Code selection tricks are avoided. The transformations are insensitive to the dictions employed in the higher level language. Loops, controlled by DOs or less structured techniques, but otherwise equivalent, are both amenable to the same optimizations.

The machine independence of these code improvement algorithms is demonstrated by the fact that the PL/1L compiler has final code selection (or back end) phases for six different machines at present: System/370, and five experimental machines. While the IL was originally designed for one of the experimental machines, the compiler does produce high quality code for all the target machines. With the exception of register allocation, the remainder of the optimizations are insensitive to the target machine.

Even the source language does not effect the optimization algorithms. A PASCAL compiler is under development at IBM Research. It is implemented by merely providing a PASCAL→IL front end. Thereafter, code improvement (and final code selection) run as originally written for PL/1L without any regard to the original source language. It appears that these techniques are applicable to a wide class of languages that are more general than FORTRAN, but not as general as SETL.

The PL/1L compiler was produced by eight people who, during the several years of its development, contributed many ideas which are reported in this paper. These people are: Marc Auslander, Gregory Chaitin (who did much of the work on register allocation), Richard Goldberg, Martin Hopkins, Peter Markstein, Peter Oden, Philip Owens, and Henry Warren.

A description of the code improvement techniques used by the PL/1L compiler is given in Section 2.

Interprocedural analysis [8] has not been used in the PL/1L compiler, but will be considered in the future.

1.4 Measurements

There are an unlimited number of sequences in which the code improvement transformations can be applied to the test programs. Obviously, just a small set of the transformation sequences could be applied and measured. Hopefully from the sequences chosen, the relative value of extra compile time versus improved code benefits can be assessed by the reader. Again, with six back ends, measurements could be made to demonstrate whether the improved IL could be translated to each target machine with equal effectiveness. Unfortunately limitations on computer time have led us to make most measurements only on System/370.

The programs chosen to be compiled and measured are more than kernels. The optimization transformations do, in fact, perform exceptionally well on small kernels. However, the real gain will be in the ability to apply program improvement to large stretches of code in spite of all the commonly arising obstacles to optimization. The chosen programs are also self-contained, so that the measurement of execution time of compiled programs is not overshadowed by the weight of fixed, precompiled library routines.

The measurement results are presented in Section 3.

2. OPTIMIZATION TECHNIQUES

2.1 Terminology

Some basic definitions are needed in the discussion of code improvement algorithms.

A **straight line block** is any sequence of IL instructions such that control must enter the sequence at the first instruction. With the possible exception of the last (resp. first) instruction, every instruction in such a block has a unique successor (resp. predecessor) which is in the sequence. A **basic block** is a straight line block of maximal length.

For the analyses which underlie code improvement, the IL program is first decomposed into basic blocks, and a **flow graph** is then constructed. Each node in the flow graph corresponds to a basic block, and the directed edges between the nodes correspond to the flow of control between the basic blocks.

A **strongly connected region** is a subgraph in which there exists a path between any nodes in the subgraph such that the path lies entirely within the subgraph. A **single entry strongly connected region** is a strongly connected region in which only one node, called the header, has predecessors outside the region. In this paper, strongly connected regions will be understood to be single entry, unless otherwise stated.

An **extended basic block** (headed by a distinguished node n) is a connected subgraph containing n such that all nodes of the subgraph except possibly n have exactly one predecessor. An extended basic block is simply a tree of basic blocks.

2.2 Value Numbering

Value numbering is a technique for recognizing those computations in basic blocks which yield identical results. Computations are examined in execution order within a basic block. Two distinct situations can arise. In the first case a computation may be encountered which is formally identical to a previously encountered computation such that all of the operands are unchanged between the two encounters. In this event, the second computation is redundant, and it may be eliminated from the IL text.

In the second case, a computation may be encountered which can be be shown to yield the same result as a previously encountered computation, although the computations are not formally identical. The second computation, called a discovered identity, may be replaced by copying the result of the first computation.

$$\begin{array}{l} z = x * y \\ a = copy\ x \\ z = x * y \\ b = a * y \end{array} \quad becomes \quad \begin{array}{l} z = x * y \\ a = copy\ x \\ b = copy\ z \end{array} \qquad (2.2.1)$$

In the left hand column of (2.2.1), the third line is formally identical to the first, and is redundant. The fourth line is an example of a discovered identity. The right column shows the improved code.

On most computers "copy" is the fastest operation when the source and target are both high speed registers, so that value numbering never degrades the IL program. However, the "copy" operations may lead to further improvement by subsumption (see 2.5).

Every operand, if encountered before its definition, is assigned a unique integer, called a value number. For each computation, a hash table is searched for a tuple consisting of the operator and the value numbers of its operands. If found, the hash table yields a value number for the computation. If the target of the computation already has the same value number, a redundant expression has been found. Otherwise, if another register has the computation's value number, then the computation can be replaced by a register copy operation. If the tuple is not in the hash table, a new value number is stored with the tuple in the hash table, and the target of the computation is given this value number.

Arrays in storage must be treated with care since it is impractical to associate a value number with each array element. Aliasing due to declarations or use of subprogram arguments pose similar problems, and are discussed in [1].

By arranging to remove items from the hash table and to regress value numbers assigned to registers, it is possible to use value numbering on extended basic blocks [1].

Whenever all operands of a computation are known constants, the computation can be evaluated and the target of the computation then also holds a known constant. Such constant propagation is carried out concurrently with value numbering. Of particular importance is the case when the action of a conditional branch or validity check can be determined. Either the conditional branch or the validity check can be removed, or the conditional branch can be transformed into an unconditional branch; an edge is removed from the flow graph, which in turn enhances the effectiveness of other global optimization algorithms.

2.3 Global Common Subexpression Elimination

It is not known whether value numbering in all its generality can be extended to an arbitrary flow graph. In particular, no uniform procedure which recognizes discovered expressions over arbitrary flow is known. However, it is possible to determine whether two formally identical computations yield the same result.

An expression e is **available** at entry to block b if e is computed along every path from the entry node to b, and if the computation of e at entry to block b would yield the same result as the most recently encountered computation of e before b, regardless of the path. If for each basic block, the set *avail(b)* of available expressions were known, then redundant formally identical computations can easily be recognized by traversing the code in b in execution order, and updating *avail(b)* as each instruction of b is examined. Any computation which is in *avail(b)* when it is examined is redundant.

The sets *avail(b)* can easily be computed by first determining for each basic block, the sets of **downward exposed expressions**, *dex(b)*, and **not-killed expressions**, *nok(b)*.

An expression e is in *dex(b)* if it is computed in b, and none of its operands are subsequently altered in b.

An expression is in *nok(b)* if none of its operands are altered in block b.

Notice that *nok(b)* and *dex(b)* can be determined by examining only the instructions in basic block b in execution order. In terms of these sets, *avail(b)* must obey the following relationship:

$$avail(b) = \bigcap_{p \in predecessor(b)} \{[avail(p) \cap nok(p)] \cup dex(p)\} \qquad (2.3.1)$$

A maximal solution to (2.3.1) can be found by setting *avail(b)* to the empty set for the program entry node, and to the set of all expressions for every other node, and then applying (2.3.1) to all nodes of the graph until *avail(b)* remains unchanged for all basic blocks b. This method is known to converge [2] but the PL/1L compiler uses a non-iterative technique to determine the *avail* sets [3].

With either technique, the set operations are most easily implemented if sets are represented as bit vectors in which each bit position corresponds to a specific expression within the program. The unions and intersections are then realized by fast logical-or and logical-and operations.

2.4 Code Motion

It may happen that a computation in a strongly connected region uses operands which are invariant within the region. If the computation were performed just prior to entering the region, the computation within the region would become redundant and could be eliminated. This combination of code insertion and redundant code elimination gives the illusion of code moving from regions of high execution frequency to regions of lower execution frequency. Several methods have been published for determining the candidate expressions for code motion, and the locations at which to insert them [3, 4]. The PL/1L compiler uses a scheme which exploits knowledge of the strongly connected regions of the program [5]. It is possible to create an ordered list *L* of all the strongly connected regions such that if a strongly connected region contains other strongly connected regions, the interior regions appear earlier in the list.

The regions are processed in the order in which they appear in *L*. For each region, equation (2.3.1) is solved just for the nodes of the region under two initializations, which differ only in the initial assignment of *avail(h)*, where h is the entry node of the region. All the other nodes are initialized to the set of all expressions. One solution, *p(b)*, is found by initializing *avail(h)* to the set of all expressions, and represents those expressions available at b assuming that all expressions were available on entry to the region. The other solution, *d(b)*, is found by initializing *avail(h)* to the empty set, and represents those expressions available at b assuming that no expressions were available on entry to the region. Then for any expression e:

- $ep(b) \cap d(b)$ implies that e is available on entry to block b,
- $ep(b) \cap \overline{d(b)}$ implies that e would be available on entry to block b if it were available on entry to the scr,
- $\overline{ep(b)} \cap d(b)$ is impossible, and
- $\overline{ep(b)} \cap \overline{d(b)}$ implies that if e is computed in block b, its first evaluation is not redundant.

The set of expressions $p(b) \cap \overline{d(b)}$ are candidates for placement into an artificially inserted basic block whose predecessors are the predecessors of the region's entry node, h, and whose successor is h. Considerations of profitability and safety may inhibit moving some of these expressions. For example, in (2.4.1), J/I cannot safely be moved out of the loop, because if I were equal to 0, the "improved" code would cause a divide-by-0 exception, whereas the original code would not. The definitions of **not killed** and **downward exposed** can be extended to strongly connected regions, which reduce the processing time for regions with embedded strongly connected regions.

```
do k = 0 to N;
  if i ≠ 0 then
    a(k) = j / i;
end;
```
(2.4.1)

2.5 Subsumption

Whenever the IL contains an instruction of the form $x = copy\ y$, it is possible that this instruction might not require any code in the final object program. This fortunate circumstance arises whenever no use of x occurs after a redefinition of y, and no use of y occurs after a redefinition of x. In this case, x and y are merely different names for the same object. In the PL/1L compiler, register subsumption is determined as a byproduct of register allocation. In effect, if register x is subsumed by register y, they will both be allocated to the same real register prior to final code generation, which simply does not generate any code whenever a real register is copied onto itself.

2.6 Use-Def Chains

Use-def chains denotes a map from points in the IL where operands are used to points in the IL where the operands are defined. This map, and its inverse play a key role in most of the remaining code improvement algorithms to be described. In practice, the use-def chains are not produced explicitly; instead, several maps from basic blocks to set of definition points are computed, from which the necessary use-def information can be derived by examining a basic block, if necessary.

By examining only the contents of a basic block, the following sets can be determined:

- *uses(b):* If an expression e is used as an operand in basic block b before it is computed, then d is in *uses(b)* for every definition point d of the expression e.
- *thru(b):* If an expression e is not computed in b, then d is in *thru(b)* for every definition point d of the expression e.
- *dnx(b):* If an expression e, defined at point d within b is downward exposed in b, then d is in *dnx(b)*.

Two global maps from basic blocks to points of definition must be computed:

- *live(b):* If there exists a path from the entrance of b to a use of e that does not pass through a computation of e, then d is in *live(b)* for every definition point d of e.
- *reach(b):* If there is a path from the definition of e at d to block b without passing through another computation of e, then d is in *reach(b)*.

The *reach* and *live* sets can be computed from the following relationships in much the same manner as (2.4.1) is solved for *available* expressions [2, 6].

$$live(b) = uses(b) \cup thru(b) \cap \left[\bigcap_{s \in successor(b)} live(s) \right] \qquad (2.6.1)$$

$$reach(b) = \bigcup_{p \in predecessor(b)} [dnx(p) \cup \{reach(p) \cap thru(p)\}] \qquad (2.6.2)$$

For basic block b, $reach(b) \cap uses(b)$ gives the definitions which are used in block b, but reach it from another basic block, and from these sets, the use-def chains can be derived.

2.7 Dead Code Elimination

An expression e is dead if there exists no path from its computation to a use of e. The *live* sets permit dead code to be detected and eliminated on a basic block basis. For basic block b, compute the set of expressions which are dead on exit from b as:

$$dead(b) = \left\{ e \mid \exists \text{ definition point of } e \text{ in } \bigcap_{s \in successor(b)} \overline{live(s)} \right\} \qquad (2.7.1)$$

Traverse b in reverse execution order. For each computation c do the following:

if $c \in dead(b)$ the computation is dead and can be deleted;
otherwise, add c to *dead(b)* and remove all of c's operands from *dead(b)*.

Dead code can arise in many ways through no shortcoming of the source language program. For example, a reasonable way for the statement $C = length\ (A \mid\mid B)$ to be compiled is as follows: $A \mid\mid B$ is the first subexpression recognized by the parser. Unaware of the context in which $A \mid\mid B$ will be used, it generates code to compute it. Assuming that L' is the IL operator to find the length of a string, the IL code generated is:

$$\begin{aligned} L'T &= L'A + L'B \\ T &= concat(A, B) \end{aligned} \qquad (2.7.2)$$

Next, *length* $(A \mid\mid B)$, or equivalently, *length* (T) is recognized and expanded as the built-in length operator, producing as IL for the original statement:

$$\begin{aligned} L'T &= L'A + L'B \\ T &= concat(A, B) \\ C &= L'T \end{aligned} \qquad (2.7.3)$$

If no use is actually made of $A \mid\mid B$, then the second IL statement in (2.7.3) is dead, and it will be discarded by dead code elimination. Notice that the parser did not special case the expansion of an argument to a built-in function.

Dead code elimination is not commutative with other code improvement algorithms. Consider the following code fragment:

```
if length (A || B) < 16 then
  C = A || B;                    (2.7.4)
else C = '';
```

Performing dead code elimination before common subexpression elimination yields superior code, because the computation of $A \mid\mid B$ will only appear in the translation of the *then* clause. On the other hand, if common subexpression elimination were performed first, the computation of $A \mid\mid B$ in the *then* clause would be eliminated, thereby freezing the computation in the unconditional portion of the program.

Dead code elimination should (also) be performed as the last code improvement because the other code improvements tend to induce computations to become unused.

2.8 Global Constant Propagation

The use-def chains allow the effects of computations with constants to be propagated throughout a program, regardless of flow. In an initial scan of an IL program, all points of definition of constant values are collected in a constant-definition list. For each element on the list, the use-def chains identify all computations which use the constant as an operand. For such a computation, if all its operands are constants (the use-def chains can be used again to determine

where the operands are defined) the computation is replaced by a copy of the constant value, and this definition point and its value is placed at the end of the constant-definition list. When the entire list has been scanned, the process terminates.

2.9 Strength Reduction

Strength reduction is the process of replacing complex operations with simpler ones in strongly connected regions, and copying the complex operation to a point just prior to the region's entry node. (Code motion is a special case, where the simpler operation is no operation at all!) In the PL/1L compiler, strength reduction is used to transform multiplications which often arise from subscripted variables, into additions.

A quantity which varies as a linear function of the number of loop iterations is called an **induction variable**. To replace a product of an induction variable I and a loop-constant c, designate a new register t to hold the current value of I*c. t must be initialized by actually computing I*c before control enters the strongly connected region. Wherever I is incremented in the region by a quantity q, precede that instruction by incrementing t by c*q. These points are determined from the use-def chains. Of course, c*q will be a loop-constant, and should be computed prior to entering the loop. (c*q might later be determined to be a constant by global constant propagation).

This technique can easily be extended to strength reduce products of induction variables. Just as with code motion, strength reduction should operate on regions as given by list L (see Section 2.4). If a node represents an inner strongly connected region, no instructions in such a node need be considered further. Products which were copied to points outside an inner region may themselves be strength reduced in an outer region.

If, after strength reduction, the induction variable I would be dead except for use in a loop closure test, the test may be rewritten as a comparison of the new register t against the other comparand, multiplied by c. (2.9.1) is an example to which strength reduction, global constant propagation, and dead code elimination have been applied

```
loop: R = I * 6                    t = I * 6
      - - -                        loop: R = t
      - - -                              - - -
      I = I + 1        becomes           - - -          (2.9.1)
      compare I : 100                    t = t + 6
      branch - if - low loop             compare t : 600
                                         branch - if - low loop
```

2.10. Reassociation and Essential Computations

An **essential computation** is a computation whose result is stored in memory, is used to access memory, is printed, or is passed to a subprogram as an argument. For each essential computation in a strongly connected region, the full algebraic expression is constructed in terms of quantities available on loop entry. It may happen that the algebraic expansions of the essential computations will reveal common subexpressions which were not apparent from the formal identities. By using the associative rule to rearrange these computations, the discovered common subexpressions can be exploited to reduce the actual computations in a loop. This is a relatively new code improvement technique which has not yet been reported in the literature. In some cases examined, as much as 50% of the code in inner loops can be eliminated in this manner.

2.11 Anchor Pointing

Consider the following statement:

if A | B then go to L; (2.11.1)

This statement can be interpreted as:

If A then go to L;
if B then go to L; (2.11.2)

(2.11.2) avoids the logical-or operation and avoids testing B altogether if A is true. A similar transformation is used to replace if-clauses using the logical-and operator. If A or B are expressions involving logical operations, these operations likewise can be replaced by a pair of conditional branches.

2.12. Register Allocation

Each of the above optimizations improves the IL program. However, before final code generation takes place, the unlimited set of IL registers must be mapped onto the finite set of registers in the real target machine. If all the IL registers can be mapped onto the real registers, then the full benefit of the other optimizations can be realized; otherwise the contents of some of the IL registers must temporarily be spilled, that is, they must be kept in storage.

The first step in register allocation is to prepare an **interference graph**. Each register in the IL program, as well as the real registers, are represented by a node. Two registers which cannot coexist in a real register are said to **interfere**. For each interference, an edge is inserted into the graph. The interference graph is then colored; that is, "colors" are assigned to each node in the graph under the restriction that nodes joined by an edge may not have the same color. The minimum number of colors required, R, is called the **chromatic number.**

Because the determination of the chromatic number is NP-complete [7], the PL/1L compiler uses a heuristic to color the interference graph. (In all practical cases investigated, the heuristic succeeded in finding the chromatic number. It appears that real programs do not lead to very complex interference graphs.) If the chromatic number does not exceed the number of real registers, then register allocation is achieved by assigning to each IL register, the real register which has the same color.

Subsumption is accomplished as a byproduct of building the interference graph by not regarding a register copy operation as producing an interference between the source and target registers. After the interference graph is constructed, each register copy operation is reexamined. If the source and target do not interfere, these registers can share the same real register, and their nodes in the interference graph are coalesced, so that these nodes will per force be assigned the same color.

The algorithms for spilling are the most heuristic of the code improvement algorithms, and will not be described here. When spilling is required, the heuristic forces a sufficient number of computations into memory so that the number of concurrently live expressions in registers does not exceed a fixed quantity, N. If the chromatic number is still too high, N is reduced by 1. Spilling and recoloring are repeated until the chromatic number does not exceed the number of real registers.

Setting N to four less than the number of real registers has avoided iterated spilling in practice. As an option, N can be started at 2 more than the number of available registers, in an attempt to spill as little as possible, but this choice usually lead to several iterations, and can be very time consuming.

We consider the IL instruction $A = P + Q$ to further illustrate the the effects of register allocation. This can be mapped into one System 370 instruction.

```
AR  A,Q
```

if A, P, and Q are all in registers at the time of the operation, and if P is dead afterward, so that it can share a register with A. If all the quantities must be in memory, the final code could become:

```
L   x,P
A   x,Q
ST  x,A
```

3. MEASUREMENTS

3.1 Results

The PL/1L compiler consists of five phases, each of which is itself a set of PL/1L programs. It can apply code improvement at four different levels, called OPT(0) to OPT(3), with OPT(3) providing the highest level of optimization. For the purposes of this paper, an OPT(-1) has been added which avoids as much of the code improvement as possible, so that comparisons could be made with totally unenhanced code. Table I shows the sequence of code improvement algorithms applied for each optimization level.

The standard PL/1L compiler is produced by compiling all the components at OPT(3). A control compiler was built by recompiling all components at OPT(0). We found that the standard compiler ran from 22% to 27% faster than the control compiler. When using the control compiler, the code generation phases were from 14% to 20% slower, while the lexical analysis and code improvement phases were from 26% to 31% slower.

The results of compiling and executing several programs are shown in Table II. (USEDEF, a component of the compiler, solves the boolean system of equations given in Section 2.6. PUZZLE is a program which reconstructs some dissected plane figures. IPOO is one of the programs used to verify the Four Color Theorem, and HEAPSORT is an in-place, in-memory sort.) HEAPSORT is unique among these test programs in that it is written in the intersection of PL/1 and PL/1L. HEAPSORT was also coded in FORTRAN using the same program structure. The results of using the IBM PLIOPT and FORTRAN H compilers are also given in Table II.

3.2 Observations

Perhaps the most striking conclusion to be drawn from these tables is that elimination of all optimizations by use of OPT(-1) does not even save compilation time. The sheer volume of unenhanced code that must ultimately be processed for register allocation and the production of the final object code consumes more time than is required for value numbering.

The principal difference between OPT(0) and the higher levels of code improvement is that OPT(0) does not perform global code motion and global common subexpression elimination. While these code improvement algorithms decrease IL code, they often require that values be kept in registers over long stretches of code, consequently increasing the chromatic number of the register interference graph. The higher the chromatic number, the greater the probability of requiring spill code during register allocation, which is a time consuming process. The effects of spilling can be seen in the compilation times of PUZZLE, and IPOO, each of which has a subprogram which requires spilling with OPT(1) but not with OPT(0).

If the statistics for the OPT(-1) compilations are ignored, the largest incremental gain in object code efficiency arises when OPT(1) is used in place of OPT(0), thereby exploiting global relationships in code improvement. Even though code space may increase due to spilling, execution efficiency shows improvement. The gains achieved by OPT(2) and OPT(3) are relatively smaller. For programs which do not require spill code, OPT(3) offers no advantages over OPT(2).

The statistics for USEDEF, which is a subprogram in the PL/1L compiler, shows the effects of applying these transformations to a program that references complex structures in a nest of loops. The OPT(0)-produced code runs 71% slower than OPT(1)-produced code, and 85% slower than the OPT(3) produced code.

The increase in code space which sometimes occurs when applying a higher optimization level results from register allocation having to insert spill code. However, execution time decreases even in such cases.

Even PL/1L's OPT(0) produced more time and space efficient code for HEAPSORT than the other compilers did at their highest optimization level. However, in this example, the DO-index is not used directly as a subscript, which disables many of FORTRAN's excellent capabilities for optimizing DO-loops.

In experimenting with different orders of applying the code improvement algorithms, we found that omission of value numbering at optimization levels higher than OPT(0) sometimes produced better results! The identities discovered and exploited by value numbering destroy some of the formal identities which are used in the determination of movable expressions during global commoning and code motion. To remedy this defect, when global commoning is applied, it should be applied at least once before value numbering is ever attempted.

At the time these measurements were made, reassociation (see Section 2.10) was not yet implemented. In USEDEF, for example, we have observed that this code improvement technique will reduce the inner loops of the IL program by 50%. The effect on the final code would be even greater, because the improved IL code also has a substantially lower chromatic number.

Linear test replacement was also not implemented in time for these measurements. As a consequence, for a machine such as the IBM System 370/168 with fast hardware multiplication, some programs with spill code in inner loops actually were degraded by strength reduction. When USEDEF was compiled at OPT(2) without strength reduction, the resultant code executed in 0.120 sec. On the other hand, for some of the experimental machines, strength reduction is worthwhile even without linear test replacement.

Constant propagation did not yield as much benefit was was expected. We discovered that some known constants, such as those declared in INITIAL attributes, were not exposed in the IL. Some of the code generation strategies must be modified in our compiler before the benefits of this code improvement technique can be assessed.

OPTIMIZATION LEVEL

Code improvement transformation	-1	0	1	2	3
dead code elimination		x	x	x	x
value numbering		x	x	x	x
local constant propagation		x	x		
global commoning, code motion			x	x	x
dead code elimination			x	x	x
strength reduction			x	x	x
macro expansion†	x	x	x	x	x
dead code elimination				x	x
value numbering				x	x
local constant propagation				x	x
global commoning code motion				x	x
register allocation: N=R+1*	x	x	x	x	
N=R+2*					x

Table I
Sequence of Code Improvement Transformations
applied at Optimization Levels of PL/1L compiler

† See Sec. 1.3, #1
* See Sec. 2.12 for definitions of N and R

OPTIMIZATION LEVEL

		-1	0	1	2	3
USEDEF 360 lines	Compilation Time (sec.)	19.7	19.7[4]	31.7[4]	34.2[4]	51.2[4]
	Code Space (bytes)	12138	5386	6390	6098	5942
	Execution Time (sec.)	.720	.230	134	129	.124
PUZZLE 154 lines	Compilation Time	6.2	5.7	9.3[1]	10.2[1]	14.7[1]
	Code Space	2790	1682	1778	1782	1698
	Execution Time	1.33	0.73	0.67	0.67	0.62
IPOO 295 lines	Compilation Time	9.8	10.3[3]	15.5[2]	17.3[2]	20.5[2]
	Code Space	4908	3404	3232	3216	3156
	Execution Time	5.88	4.25	3.61	3.59	3.51
HEAPSORT PL/1L 84 lines	Compilation Time	2.2	1.9	2.3	2.5	2.5
	Code Space	1024	432	384	368	368
	Execution Time	5.60	2.26	2.12	2.02	2.02
HEAPSORT PL/1	Compilation Time		0.83		0.96	
	Code Space		740		700	
	Execution Time		4.31		4.00	
HEAPSORT FORTRAN	Compilation Time		0.26		0.33	0.38
	Code Space		674		490	442
	Execution Time		4.83		2.88	2.88

Table II

1 1 of two subprograms require spill-code
2 2 of six subprograms require spill-code
3 1 of six subprograms requires spill-code
4 4 of four subprograms require spill code

REFERENCES

[1] J. Cocke, & J.T. Schwartz, "Programming Languages and Their Compilers", Courant Institute of Mathematical Sciences, N.Y.U., 1970.

[2] M.S. Hecht, "Flow Analysis of Computer Programs", Elsevier North-Holland, New York, 1977.

[3] J.T. Schwartz, "On Programming - An Interim Report on the SETL Project. Installment II: The SETL Language and Examples of Its Use", Courant Institute of Mathematical Sciences, N.Y.U., 1973.

[4] E. Morel & C. Renvoise, "Global Optimization by Supression of Partial Redundancies", CACM Vol 22, No.2, pp. 96-103, 1979.

[5] R. Tarjan, "Depth First Search and Linear Graph Algorithms", SIAM J. Comput. Vol 1, p. 146-160, 1972.

[6] F.E. Allen & J. Cocke, "A Program Data Flow Analysis Procedure", CACM Vol 19, p 137-147, 1976.

[7] A.V. Aho, J.E. Hopcroft, & J.D. Ullman, "The Design and Analysis of Computer Algorithms", Addison Wesley, Reading, Mass., 1974.

[8] B.K. Rosen, "Data Flow Analysis for Procedural Languages", JACM Vol. 26, p. 322, 1979.